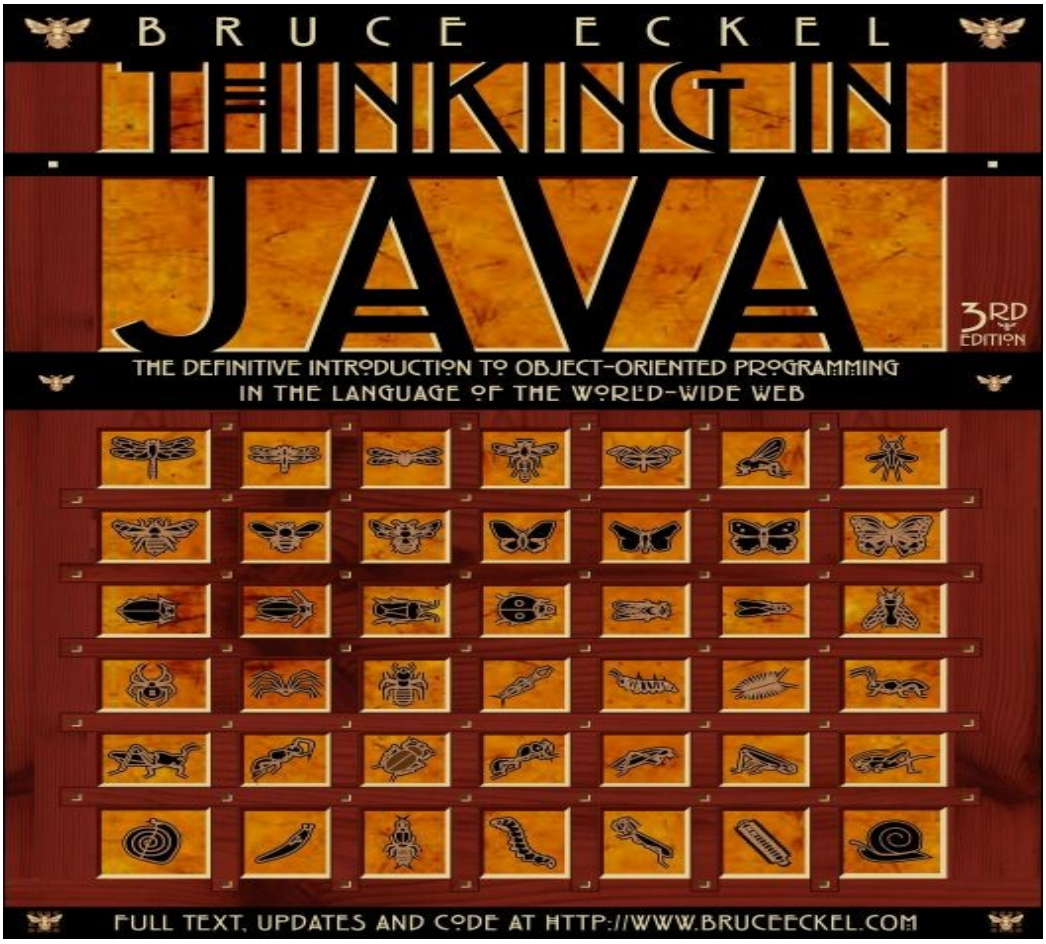
Penser en java

Seconde édition

Bruce Eckel



Traducteurs pour cette version française :

- *Cédric BABAULT*
- *Phillipe BOITE*
- *Yannis CHICHA*
- *Nate CULWELL-KANAREK*
- *Jérome DANNOVILLE*
- *Florence DEFAIX*
- *Armel FORTUN*
- *Daniel LE BERRE*
- *Anthony PRAUD*
- *Jérome QUELIN*
- *RACZY*
- *Olivier THOMANN*
- *Jean-Pierre VIDAL.*

Mise en page pour cette version:

- *Stéphane BIGNET*

Liens des différents sites internet qui ont fournis les sources:

- http://penserenjava.free.fr/
- http://bruce-eckel.developpez.com/livres/java/traduction/tij2/

Table des matières

Préface

J'ai suggéré à mon frère Todd, qui est en train de migrer du hardware vers le software, que la prochaine grande révolution serait l'ingénierie génétique.

Nous créerons bientôt des organismes dédiés à la fabrication de nourriture, de carburant, de plastique ; ils digéreront la pollution et, de manière générale, nous permettront de maîtriser la manipulation du monde réel, ceci pour un coût minime comparé à celui d'aujourd'hui. J'ai prétendu que la révolution informatique serait vraiment peu de chose au regard de cela.

Puis j'ai réalisé que j'étais en train de commettre une erreur triviale chez les auteurs de science-fiction : oublier le propos de la technologie (ce qui est évidemment très facile à faire en science-fiction). Un écrivain expérimenté sait qu'on ne raconte jamais une histoire à propos de choses, mais de gens. La génétique aura un très grand impact sur nos vies, mais je ne suis pas persuadé qu'elle éclipsera la révolution informatique (qui d'ailleurs rend possible la révolution génétique) — ou au moins la révolution de l'information. L'information, c'est essentiellement se parler les uns les autres : bien entendu, les voitures, les chaussures, et surtout les maladies génétiques sont importantes, mais en définitive ce ne sont que des faux-semblants. La vrai question est notre relation au monde. Ainsi en est-il de la communication.

Ce livre est un cas. Une majorité d'amis pensa que j'étais soit vraiment hardi soit légèrement dérangé pour mettre tout cela sur le Web. « Pourquoi quelqu'un voudrait-il l'acheter ? » me disaient-ils. Si j'avais eu un tempérament plus conservateur, je ne m'y serais pas pris de cette manière ; en réalité je ne voulais pas écrire un livre supplémentaire de style traditionnel sur les ordinateurs. Je ne savais ce qui en aurait résulté si je n'avais pas agi ainsi, mais en tout cas ce fut la meilleure chose que j'ai jamais réalisée avec un livre.

Tout d'abord, chacun commença à m'envoyer des correctifs. Ce fut un processus très amusant, parce que mes amis avaient fureté dans chaque coin et recoin et repéré les erreurs techniques aussi bien que grammaticales, ce qui me permit d'éliminer les fautes de toutes sortes que j'aurais laissé passer sans cela. Dans ce travail, ils ont été tout simplement formidables, commençant très souvent par me dire « bon, je ne dis pas ça pour critiquer... » pour me mettre ensuite sous le nez un ensemble d'erreurs que je n'aurais jamais été capable de trouver par moi-même. Je crois que ce fut une espèce de travail de groupe, et cela a réellement ajouté quelque chose de spécial à ce livre.

Mais ensuite, j'ai commencé à entendre ceci : « OK, très bien, c'est bien gentil vous avez fait une version électronique, mais moi j'aurais préféré une version complète imprimée chez un vrai éditeur ». Alors j'ai beaucoup travaillé afin que chacun puisse l'imprimer dans un format adéquat, mais cela n'a pas suffis à résorber la demande d'un livre « publié ». La plupart des gens n'ont pas envie de lire le livre à l'écran dans son intégralité, et l'idée de transporter un paquet de feuilles volantes, même impeccablement imprimées, ne leur conviendrait pas davantage (de plus, je pense que ce n'est pas forcément économique en terme de toner). Après tout il semblerait que la révolution informatique ne risque pas de mettre les éditeurs au chômage. Un étudiant suggéra toutefois que cela pourrait servir de modèle pour l'édition du futur : les livres seraient d'abord publiés sur le Web, et, à condition qu'ils rencontrent un certain intérêt, on les coucherait sur papier. Actuellement, une grande majorité de livres représentent des désastres financiers, et cette nouvelle approche pourrait peut-être rendre l'industrie de l'édition plus rentable.

Ce livre a été par ailleurs une expérience enrichissante pour moi. Ma première approche de Java fut « juste un nouveau langage de programmation », ce qu'il est de fait par ailleurs. Mais, le

temps passant, et au fur et à mesure que je l'étudiais plus en profondeur, je commençais à m'apercevoir que son propos fondamental était différent de celui de tous les langages que j'avais connu auparavant.

Programmer, c'est gérer la complexité : complexité du problème que l'on veut résoudre, superposée à la complexité de la machine sur laquelle il va être résolu. Bien des projets de programmation ont avorté à cause de cette complexité. Je voudrais maintenant dire que, de tous les langages de programmation que je connaisse, aucun n'a été conçu pour gérer la complexité du développement et de la maintenance de programmes [1]. Bien entendu, dans la conception des langages, beaucoup de décisions ont été prises en gardant la complexité présente à l'esprit, mais, dans chaque exemple et à partir d'un certain moment, d'autres problèmes ont surgi qui furent considérés comme essentiels et par suite intégrés à la mixture. Fatalement, ces nouveaux problèmes furent de ceux qui envoyèrent finalement les programmeurs « droit dans le mur » avec ce langage. Par exemple, C++ se devait de posséder une compatibilité ascendante avec C (afin de favoriser la migration des programmeurs C), ainsi qu'une certaine efficacité. Ces deux buts étaient très utiles et ont grandement participé au succès de C++, mais dans le même temps ils ont généré une complexité supplémentaire qui a eu pour résultat d'empêcher certains projets d'arriver à terme (bien entendu, vous pouvez toujours vous en prendre à la responsabilité des programmeurs et/ou de leur encadrement, mais, si un langage pouvait vous aider à repérer vos erreurs, pourquoi ne le ferait-il pas ?). Autre exemple, Visual Basic (VB) était lié à BASIC, lequel n'était pas vraiment conçu comme un langage extensible, et par suite toutes les extensions superposées à VB se sont traduites par une syntaxe vraiment horrible et impossible à maintenir. Perl a une compatibilité ascendante avec Awk, Sed, Grep ainsi que d'autres outils Unix qu'il était censé remplacer, le résultat est qu'il est souvent accusé de produire du « code-à-écrire-seulement » (c'est à dire qu'on est incapable de se relire quelques mois plus tard). D'un autre côté, C++, VB, Perl, et d'autres langages comme Smalltalk, de par leur conception, ont abordé le problème de la complexité, et par là se révèlent remarquablement efficaces dans la résolution de certains types de problèmes.

Ce qui m'a le plus frappé alors que je commençais à comprendre Java est quelque chose qui s'apparente à l'incroyable finalité de diminuer la complexité *pour le programmeur*. Un peu comme si l'on disait « rien n'est important, mis à part réduire le temps et la difficulté pour produire un code robuste ». Dans les premières versions de Java, cette finalité s'est traduite par un code qui ne tournait pas très vite (bien que l'on ait fait de nombreuses promesses concernant la vitesse de Java) mais il n'empêche que cela a réduit le temps de développement de manière stupéfiante : la moitié, ou moins, du temps nécessaire à la création d'un programme équivalent en C++. Ce seul résultat pourrait déjà se traduire par une incroyable économie de temps et d'argent, mais Java ne s'arrête pas là. Il s'attache à encapsuler toutes les tâches complexes qui ont pris de l'importance, comme le multithreading et la programmation réseau, au moyen de fonctionnalités du langage ou de bibliothèques rendant parfois ces tâches triviales. Et pour finir, il traite quelques problèmes d'une réelle complexité : programmes multi-plate-forme, changements dynamiques de code, sans oublier la sécurité, chacun d'entre eux pouvant s'adapter à votre gamme de difficultés, depuis un « obstacle » jusqu'à un « point bloquant ». Ainsi, malgré les problèmes de performance que nous avons vus, les promesses de Java sont énormes : il est capable de faire de nous des programmeurs encore plus productifs.

Le Web est l'un des lieux où cela se révèle le plus. La programmation réseau a toujours été difficile, et Java la rend facile (et les concepteurs du langage Java travaillent à la rendre encore plus facile). La programmation réseau, c'est ce qui fait que nous parlons entre nous de manière plus pertinente et meilleur marché que nous ne l'avons jamais fait avec le téléphone (l'émail à lui seul a révolutionné bien des entreprises). Alors que nous nous parlons de plus en plus, des choses sensationnelles commencent à émerger, peut-être plus incroyables que celles promises par l'ingénierie gé-

nétique.

Dans tous les cas — en créant des programmes, en travaillant en groupe pour créer des programmes, en concevant des interfaces utilisateur permettant aux programmes de dialoguer avec l'utilisateur, en faisant tourner des programmes sur différents types de machine, en écrivant facilement des programmes qui communiquent au travers de l'Internet — Java accroît la bande passante de la communication entre les gens. Je pense que le but de la révolution de la communication n'est certainement pas de transmettre de grandes quantités de bits ; la vraie révolution sera là lorsque nous serons tous capables de nous parler plus facilement : de personne à personne, mais aussi en groupe, et, pourquoi pas, sur la planète entière. J'ai entendu dire que la prochaine révolution verra l'émergence d'une espèce d'esprit global associant un grand nombre de personnes à un grand nombre d'interconnexions. Il se peut que Java soit, ou pas, l'outil de cette révolution, mais en tout cas cette possibilité m'a fait comprendre qu'il n'était pas insensé de tenter d'enseigner ce langage.

Préface à la 2ème édition

Les lecteurs de la première édition de ce livre ont fait énormément de commentaires élogieux à son propos, ce qui m'a été très agréable. Toutefois de temps à autres l'un ou l'autre s'est plaint, et l'une des critiques récurrentes est « le livre est trop gros ». Dans mon esprit, je dois dire que si « trop de pages » est votre seule plainte, c'est une faible critique (on se souvient de l'empereur d'Autriche se plaignant du travail de Mozart : « trop de notes ! », mais n'allez pas penser que je cherche d'une manière ou d'une autre à me comparer à Mozart). De plus, je peux seulement supposer qu'une telle demande provient d'une personne qui en est encore à prendre conscience de l'étendue du langage Java lui-même, et qui n'a pas encore vu ou lu les autres livres traitant du sujet — par exemple, ma référence favorite, *Core Java* de Cay Horstmann & Gary Cornell (Prentice-Hall), qui a tellement grossi qu'on a dû le scinder en deux volumes. Malgré cela, une des choses que j'ai essayé de faire dans cette édition a été d'éliminer les parties obsolètes, ou tout au moins non essentielles. Je n'ai pas eu de scrupules en faisant cela car l'original existe toujours sur le site Web ainsi que sur le CD ROM qui accompagne ce livre, sous la forme de la première édition du livre, librement téléchargeable *(http://www.BruceEckel.com)*. Si c'est l'ancienne version qui vous intéresse, elle existe encore, et ceci est un merveilleux soulagement pour un auteur. Par exemple, vous pouvez remarquer que le dernier chapitre de l'édition originale, « Projects », a disparu ; deux des projets ont intégré d'autres chapitres, et le reste n'avait plus d'intérêt. Pareillement, le chapitre « Design Patterns », devenu trop gros, a fait l'objet d'un livre séparé (également téléchargeable sur le site Web). Ainsi, logiquement, le livre devrait être plus mince.

Mais, hélas, il n'en sera pas ainsi.

Le plus gros problème est le continuel développement du langage Java lui-même, et en particulier l'extension de l'API qui nous promet de nous procurer une interface standard pour tout ce que nous pourrions imaginer (et je ne serais pas surpris de voir paraître une API « JCafetiere » pour couronner le tout). Le propos de ce livre n'est certainement pas de parler de ces API, d'autres auteurs se chargeront de cette tâche, mais certaines questions ne peuvent être ignorées. Parmi les plus importantes, Java « côté serveur » (principalement les Servlets et les Java Server Pages, ou *JSP*), qui représentent réellement une excellente solution au problème du World Wide Web, pour lequel nous avons découvert que les divers navigateurs Web ne sont pas suffisamment consistants pour traiter la programmation côté client. En outre, un problème reste entier, celui de créer facilement des applications qui se connectent à des bases de données, qui supervisent des transactions, qui gèrent la sécurité, etc., ce que traitent les Enterprise Java Beans (EJBs). Ces sujets se retrouvent dans le chapitre anciennement nommé « Programmation Réseau », appelé maintenant « Informatique Distribuée »,

un sujet qui est en passe de devenir essentiel. Ce chapitre a également grossi afin d'inclure une vue d'ensemble de Jini (prononcez « djini », ce n'est pas un acronyme, mais un nom), qui est une technologie de pointe permettant de concevoir différemment les interconnexions entre applications. Et, bien entendu, le livre a évolué pour utiliser tout au long des exemples la bibliothèque de composants graphiques Swing (GUI, Graphics User Interface, Interface Graphique Utilisateur, NdT). Ici aussi, si vous vous intéressez aux anciennes versions Java 1.0/1.1, vous les trouverez dans le livre que vous pouvez télécharger gratuitement à http://www.BruceEckel.com (et qui est également inclus dans le CD ROM fourni avec cette nouvelle édition ; vous en saurez davantage à ce sujet un peu plus tard).

Outre les quelques nouvelles fonctionnalités du langage Java 2 et les corrections effectuées dans tout le livre, un autre changement majeur est le chapitre sur les collections (chapitre 9), qui met maintenant l'accent sur les collections Java 2 utilisées tout au long du livre. J'ai également amélioré ce chapitre pour traiter plus en profondeur certaines facettes des collections, entre autres expliquer comment fonctionne une fonction de hashing (afin que vous sachiez comment en créer une convenablement). Il y a eu d'autres changements, tels que la réécriture du Chapitre 1, la suppression de quelques appendices ainsi que d'autres parties qui ne me paraissent plus indispensables pour le livre imprimé, mais ce fut le plus gros des suppressions. D'une manière générale, j'ai essayé de tout revoir, d'enlever de cette 2e édition tout ce qui n'était plus indispensable (mais que l'on peut trouver sous la forme électronique de la première édition), de traiter les modifications, et d'améliorer tout ce qui pouvait l'être. Comme le langage continue d'évoluer — mais toutefois pas à la même allure vertigineuse qu'auparavant — il ne fait pas de doute que de nouvelles éditions de ce livre verront le jour.

Je dois m'excuser auprès de ceux qui persistent dans leur critique à propos de la taille du livre. Que vous me croyiez ou non, j'ai travaillé dur pour le rendre plus mince. Malgré sa taille, je pense qu'il existe assez d'alternatives pour vous satisfaire. D'une part, le livre existe sous forme électronique (sur le site Web, ainsi que sur le CD ROM accompagnant ce livre), ainsi lorsque vous prenez votre ordinateur portable vous pouvez également emporter le livre sans supplément de poids. Si vous êtes réellement partisan de la minceur, il existe des versions Palm Pilot (quelqu'un m'a dit qu'il lirait le livre au lit sur l'écran rétro-éclairé de son Palm afin de ne pas déranger sa femme. Je ne peux qu'espérer que cela l'aidera à glisser dans les bras de Morphée). Si vous le préférez sur papier, je connais des personnes qui impriment un chapitre à la fois et l'emmènent dans leur attaché-case afin de le lire dans le train.

Java 2

Alors que j'écris ceci, la sortie de la version 1.3 du *Java Development Kit* (JDK) de Sun est imminente, et les modifications proposées pour le JDK 1.4 ont été publiées. Bien que ces numéros de version soient encore dans les « uns », la manière standard de se référer à une version du JDK 1.2 ou supérieur est de l'appeler « Java 2 ». Ceci souligne les modifications significatives entre le « vieux Java » — qui possède beaucoup de verrues, ce que je critiquais dans la première version de ce livre — et la nouvelle version du langage, améliorée et plus moderne, comportant bien moins de verrues et beaucoup de compléments, ainsi qu'une conception agréable.

Ce livre est écrit pour Java 2. J'ai la grande chance de dominer l'ancien langage et de n'écrire que pour le nouveau langage amélioré, parce que l'ancienne information existe encore dans la 1re édition sur le Web et sur le CD ROM (ce qui représente votre source d'information si vous utilisez une version antérieure à Java 2). D'autre part, et parce que n'importe qui peut librement télécharger le JDK depuis java.sun.com, cela signifie qu'en écrivant sur Java 2 je n'oblige personne à une

contrainte budgétaire élevée en lui imposant une mise à jour.

Il y a toutefois une nuance. JDK 1.3 possède quelques améliorations que j'aimerais réellement utiliser, mais la version de Java actuellement fournie pour Linux est le JDK 1.2.2. Le système Linux (voir http://www.Linux.org) est un développement très important en conjonction avec Java, parce qu'il est en train de devenir rapidement la plus importante plate-forme serveur — rapide, fiable, robuste, sécurisée, bien maintenue, et gratuite, une vraie révolution dans l'histoire de l'informatique (je ne me souviens pas avoir jamais rencontré l'ensemble de ces fonctionnalités dans aucun outil auparavant). Et Java a trouvé une très importante niche dans la programmation côté serveur sous la forme des *Servlets*, une technologie qui est une énorme amélioration de la programmation CGI traditionnelle (ceci est décrit dans le chapitre « Informatique Distribuée »).

Aussi, malgré le fait que j'aimerais n'utiliser que les toutes nouvelles fonctionnalités de Java, il est essentiel que l'ensemble puisse être compilé sous Linux, et donc que lorsque vous installerez le code source et que vous le compilerez sous cet OS (avec le dernier JDK) vous puissiez constater que l'ensemble peut être compilé. Toutefois, vous verrez que le texte est parsemé çà et là de notes à propos des fonctionnalités du JDK 1.3.

Le CD ROM

Un autre bonus de cette édition est le CD ROM que vous trouverez à la fin du livre. J'ai naguère rejeté cette idée, pensant que mettre quelques Ko de code source sur cet énorme CD n'était pas justifié, et préféré à cela que les gens le téléchargent depuis mon site Web. Cependant, vous allez voir tout de suite que ce CD ROM représente autre chose.

Le CD contient le code source que l'on trouve dans le livre, mais il contient aussi l'intégralité du livre, sous différents formats électroniques. Mon préféré est le format HTML, parce qu'il est rapide et complètement indexé — vous cliquez simplement sur une entrée de l'index ou de la table des matières et vous vous retrouvez immédiatement à l'endroit voulu dans le livre.

Toutefois la plus grande partie des 300 Mo du CD consiste en un ensemble multimédia complet nommé *Thinking in C : Foundations for C++ & Java.* A l'origine, j'avais délégué à Chuck Allison la création de ce « séminaire sur CD ROM » en tant que produit à part entière, puis j'ai décidé de l'inclure dans les secondes éditions de *Thinking in C++* et de *Thinking in Java* après avoir vécu, lors d'un séminaire, l'arrivée de personnes dépourvues de connaissances suffisantes en langage C. Leur propos était apparemment « je suis un bon programmeur et je n'ai *pas envie* d'apprendre le C, mais plutôt C++ ou Java, c'est pourquoi je compte passer rapidement sur C pour aller directement à C++/Java ». Peu après leur arrivée au séminaire, ils prennent conscience que le pré requis de la connaissance de la syntaxe du C se trouve là pour d'excellentes raisons. En incluant le CD ROM dans le livre, nous nous assurons que chaque participant à un séminaire a une préparation suffisante.

Le CD permet également d'élargir l'audience du livre. Même si le chapitre 3 (Contrôle du flux de programme) traite de parties fondamentales de Java provenant du langage C, le CD est une introduction en douceur, et à l'inverse du livre suppose de la part de l'étudiant une moindre connaissance de la programmation. Le CD étant inclus dans le livre, j'espère que davantage de personnes intégreront le cercle de la programmation Java.

[1]j'ai enlevé ceci de la 2ème édition : je pense que le langage Python est très proche de faire exactement cela. Voir http://www.Python.org.

Avant-propos

Tout comme n'importe quel langage humain, Java permet d'exprimer des concepts. S'il y parvient, il deviendra un moyen d'expression considérablement plus simple et plus souple que n'importe quelle alternative, alors même que les problèmes augmentent en taille et en complexité.

Il est impossible de considérer Java uniquement sous l'angle d'une collection de fonctionnalités — beaucoup de fonctionnalités perdent leur sens hors de leur contexte. On ne peut utiliser la somme des parties que si l'on pense en termes de *conception*, et non simplement en termes de code. Pour appréhender Java de cette manière, il faut comprendre les problèmes qui lui sont propres et ceux qui relèvent de la programmation en général. Ce livre traite de problèmes de programmation, en quoi ce sont des problèmes, et quelle est l'approche de Java pour les résoudre. Ainsi, l'ensemble de fonctionnalités que je présente dans chaque chapitre est basée sur ma manière de résoudre un certain type de problèmes au moyen de ce langage. Par cette démarche j'espère peu à peu amener le lecteur au point où « penser Java » lui deviendra naturel.

Je garderai constamment à l'esprit qu'il faut que chacun se construise un modèle de pensée permettant de développer une profonde connaissance du langage ; lorsqu'il rencontrera un problème ardu il devra être capable d'en alimenter ce modèle et d'en déduire la solution.

Pré requis

Ce livre part du principe que le lecteur est un familier de la programmation : il sait qu'un programme est un ensemble d'instructions, il sait ce que sont un sous-programme, une fonction, une macro-instruction, un ordre de contrôle tel que « if » ainsi qu'une structure de boucle telle que « while », etc. Toutefois, il a certainement appris cela de différentes façons, par exemple en programmant avec un macro-langage ou bien en utilisant un outil tel que Perl. Si vous faites partie de ceux qui sont à l'aise avec les idées de base de la programmation, vous lirez ce livre sans problème. Bien entendu, le livre sera plus *facile* pour les programmeurs C et encore plus pour les programmeurs C++, mais n'abandonnez pas pour autant si vous n'avez aucune expérience de ces langages (en revanche, préparez-vous à travailler dur ; par ailleurs, le CD multimédia fourni avec ce livre vous amènera rapidement à comprendre la syntaxe de base du langage C nécessaire à l'apprentissage de Java). Je vais introduire les concepts de la programmation orientée objet (POO) et les mécanismes de contrôle de base de Java, ainsi le lecteur en aura connaissance, et rencontrera dans les premiers exercices les instructions de base du contrôle de flux de programme.

Bien qu'il soit souvent fait référence aux fonctionnalités des langages C et C++, il ne s'agit pas d'un aparté pour initiés, mais au contraire d'aider tous les programmeurs à mettre Java en perspective avec ces deux langages, qui, après tout, sont ses parents. Je vais essayer de simplifier ces références et d'expliquer à un programmeur ne connaissant ni C ni C++ tout ce que j'imagine être peu familier pour lui.

Apprendre Java

J'ai commencé à enseigner C++ à l'époque où était édité mon premier livre *Using C++*(Osborne McGraw-Hill, 1989). Enseigner la programmation est devenu ma profession ; depuis 1989 j'ai vu bien des hochements de tête, de visages vides, ainsi que beaucoup d'expressions d'incompréhension chez maint auditoire à travers le monde. Lorsque je me mis à donner des cours chez moi, pour

des groupes plus réduits, je découvris que même ceux qui souriaient et hochaient la tête étaient déconcertés face à de nombreux problèmes. J'ai découvert aussi, alors que je présidais le cursus C++ durant plusieurs années à la Software Development Conference (et plus tard le cursus Java), que moi-même ainsi que les autres conférenciers avions tendance à traiter trop de choses trop rapidement. Finalement, à cause des différences entre les niveaux de mes auditeurs tout autant que de la manière dont je présentais mon exposé, j'aurais fini par perdre une partie de mon auditoire. Je me suis posé beaucoup de questions, mais, faisant partie de ceux qui rechignent au cours magistral (et chez beaucoup de gens, je crois qu'une telle attitude ne peut provenir que du souvenir de l'ennui que distillent de tels cours), j'ai voulu faire en sorte que tout le monde reste éveillé.

À une certaine période, je terminais mes différents cours sous la pression des besoins. C'est ainsi que j'ai fini par enseigner par essais et erreurs (une technique qui marche bien également dans la conception des programmes Java), et finalement j'ai réalisé un cours qui utilise tout ce que j'ai appris grâce à mon expérience d'enseignant — un cours qu'il me serait agréable de donner durant longtemps. Il consiste à s'attaquer au problème de l'apprentissage par touches discrètes et faciles à intégrer, et lors d'un séminaire impromptu (la situation idéale pour enseigner) chaque courte leçon est suivie d'exercices. Je donne maintenant ce cours dans des séminaires Java publics, que l'on peut trouver sur le site http://www.BruceEckel.com. (Le séminaire d'introduction est également disponible sur le CD ROM, son contenu est disponible sur le même site Web.)

Le retour d'expérience que me procure chaque séminaire m'aide à modifier et recentrer mon discours jusqu'à ce qu'il devienne un bon moyen d'enseignement. Mais ce livre est plus qu'une simple compilation de notes de séminaires : j'ai tenté d'y intégrer autant d'informations que je le pouvais, et je l'ai structuré afin que chaque sujet mène droit au suivant. Enfin, plus que tout, le livre est conçu pour aider le lecteur solitaire qui se bat avec un nouveau langage de programmation.

Buts

Comme mon précédent livre *Thinking in C++*, celui-ci a été structuré pour enseigner le langage. En particulier, ma motivation est de faire en sorte qu'il puisse me servir pour enseigner le langage dans mes propres séminaires. Lorsque je pense à un chapitre du livre, je me demande s'il pourrait faire l'objet d'une bonne leçon lors d'un séminaire. Mon but est d'avoir des chapitres courts pouvant être exposés en un temps raisonnable, suivis par des exercices réalisables dans une situation de salle de classe.

Dans ce livre je me suis donné comme buts :

1. présenter le cours pas à pas afin que le lecteur assimile chaque concept avant d'aller plus avant ;
2. utiliser des exemples qui soient aussi simples et courts que possible. De temps en temps, cela me détournera des problèmes « du monde réel », mais j'ai remarqué que les débutants sont généralement plus satisfaits de comprendre chaque détail d'un exemple qu'ils ne sont impressionnés par la portée du problème qu'ils cherchent à résoudre. Il y a également une limite à la taille du code qui peut être assimilé dans une situation de cours magistral, limite qu'il faut impérativement ne pas dépasser. A ce sujet je devrais recevoir quelques critiques pour avoir utilisé des « exemples jouets », et je les accepte volontiers, avec le prétexte que ce que je présente est utile, pédagogiquement parlant ;
3. enchaîner soigneusement la présentation des fonctionnalités afin que l'on ne rencontre jamais quoi que ce soit qui n'ait jamais été exposé. Bien entendu, ce n'est pas toujours possible, et, dans de telles situations, je donne une brève description en introduction ;

4. montrer ce que je pense être important concernant la compréhension du langage, plutôt qu'exposer tout mon savoir. Je crois que l'information est fortement hiérarchisée, qu'il est avéré que 95 % des programmeurs n'ont pas besoin de tout connaître, et que cela déroute tout le monde et ajoute à leur impression de complexité du langage. Pour prendre un exemple en C, connaissant par coeur le tableau de priorité des opérateurs (ce qui n'est pas mon cas), il est possible d'écrire un code astucieux. Mais en y réfléchissant un instant, ceci risque de dérouter le lecteur et/ou le mainteneur de ce code. Il est donc préférable d'oublier la priorité des opérateurs, et d'utiliser des parenthèses lorsque les choses ne sont pas claires ;

5. maintenir chaque section assez concentrée de telle manière que le temps de lecture - et le temps entre les exercices - soit court. Non seulement cela maintient l'attention et l'implication des auditeurs lors d'un séminaire, mais cela donne au lecteur une plus grande impression de travail bien fait ;

6. vous munir de bases solides afin que votre connaissance soit suffisante avant de suivre un cours ou lire un livre plus difficiles.

Documentation en ligne

Le langage Java et les bibliothèques de Sun Microsystems (en téléchargement libre) sont accompagnés d'une documentation sous forme électronique, que l'on peut lire avec un navigateur Web, et en pratique chaque implémentation tierce de Java possède un système de documentation équivalent. La plupart des livres publiés à propos de Java dupliquent cette documentation. Soit vous l'avez déjà, soit vous pouvez la télécharger, et donc ce livre ne la reprendra pas, excepté lorsque c'est nécessaire, parce qu'il sera généralement plus rapide de trouver la description d'une classe au moyen d'un navigateur plutôt que dans le livre (de plus, la documentation en ligne sera probablement davantage à jour). Ce livre fournira certaines descriptions de classes supplémentaires lorsqu'il sera nécessaire de compléter la documentation afin de comprendre un exemple particulier.

Les chapitres

Ce livre a été conçu en gardant une seule chose à l'esprit : la manière dont les gens apprennent le langage Java. Le retour d'information des auditeurs de séminaires m'a aidé à découvrir les parties difficiles qui justifient un autre éclairage. Dans les domaines où je fus ambitieux, où j'ai ajouté trop de fonctionnalités dans un même temps, j'ai fini par comprendre - au travers du processus d'enseignement - que si l'on ajoute de nouvelles fonctionnalités, on doit les expliquer complètement, et que cela peut dérouter les étudiants. Je me suis donc donné beaucoup de mal pour introduire aussi peu que possible de nouveaux concepts en un même temps.

Le but est donc d'enseigner une seule fonctionnalité par chapitre, ou à la rigueur un petit ensemble de fonctionnalités associées, en évitant les liaisons avec des fonctionnalités supplémentaires. De cette manière, il est possible d'assimiler chaque chose dans le contexte de la connaissance actuelle avant d'aller plus loin.

Voici une brève description des chapitres contenus dans le livre, qui correspondent aux leçons et exercices de mes séminaires.

Chapitre 1 : Introduction sur les Objets

Ce chapitre est une vue d'ensemble de ce qu'est la programmation orientée objet, y compris la réponse à la question de base « Qu'est-ce qu'un objet », ce que sont une interface et une implémen-

tation, l'abstraction et l'encapsulation, les messages et les fonctions, l'héritage et la composition, ainsi que le polymorphisme qui est d'une très haute importance. On y trouve également une vue d'ensemble de la manière dont les objets sont créés avec les constructeurs, où se trouvent les objets, où les ranger une fois créés, ainsi que le magique ramasse-miettes (*garbage collector)* qui détruit tous les objets devenus inutiles. D'autres questions seront abordées, comme le traitement des erreurs par les exceptions, le multithreading pour des interfaces utilisateur réactives, la programmation réseau et l'Internet. On y apprendra ce qui rend Java spécial, pourquoi il a tant de succès, ainsi que l'analyse et la conception orientées objet.

Chapitre 2 : Tout est Objet

Avec ce chapitre on arrive au point où l'on peut écrire un premier programme Java. Il doit donc donner une vision d'ensemble des choses essentielles, entre autres : le concept de *référence* à un objet ; comment créer un objet ; une introduction aux types primitifs et aux tableaux ; comment ils sont détruits par le ramasse-miettes ; comment toute chose est en Java un nouveau type de données (class) et comment créer vos propres classes ; les fonctions, leurs arguments et leur valeur de retour ; la visibilité des noms et l'utilisation de composants provenant d'autres bibliothèques ; le mot clef **static** ; les commentaires et la documentation intégrée.

Chapitre 3 : Contrôler le Déroulement du Programme

Ce chapitre commence avec tous les opérateurs provenant de C et C++. On y découvre les pièges classiques des opérateurs, le changement de type, la promotion et la priorité. Suivent le classique contrôle de flux de programme, les instructions de rupture de séquence déjà connues pour avoir été rencontrées dans d'autres langages de programmation : le choix avec *if-else*, la boucle avec *for* et *while* ; comment sortir d'une boucle avec *break* et *continue* aussi bien qu'avec les *break étiquetés* et les *continue étiquetés* (qui représentent le « goto manquant » en java) ; la sélection avec *switch*. Bien que la majorité de ces fonctionnalités ressemblent au code C et C++, il existe certaines différences. De plus, tous les exemples sont écrits en pur Java, afin de mieux montrer à quoi ressemble le langage.

Chapitre 4 : Initialisation et Nettoyage Mémoire

Ce chapitre commence par décrire le constructeur, lequel garantit une initialisation correcte. La définition du constructeur débouche sur le concept de surcharge de fonction (dans la mesure où plusieurs constructeurs peuvent coexister). La suite est une discussion sur le processus de nettoyage mémoire, qui n'est pas toujours aussi simple qu'il semblerait. Normalement, il suffit simplement d'abandonner un objet lorsqu'on n'en a plus besoin, et le ramasse-miettes finira par libérer la mémoire. Cette partie explore le ramasse-miettes ainsi que quelques-unes de ses particularités. Le chapitre se termine par une vision plus centrée sur l'initialisation : initialisation automatique des membres, spécification de l'initialisation des membres, ordre d'initialisation, initialisation **static** et initialisation des tableaux.

Chapitre 5 : Cacher l'implémentation

Ce chapitre traite de la manière dont le code est mis en paquetages, et pourquoi certaines parties d'une bibliothèque sont visibles alors que d'autres sont cachées. Il s'intéresse tout d'abord aux mots clefs **package** et **import**, qui sont en relation avec la gestion des paquetages au niveau fichier et permettent de construire des bibliothèques de classes. Il examine ensuite le problème sous l'angle des chemins de dossier et des noms de fichiers. Le reste du chapitre traite des mots clefs **public**, **private** et **protected**, du concept de l'accès « amical » (accès par défaut, NdT), et de ce que signifient les différents niveaux de contrôle d'accès utilisés dans divers contextes.

Chapitre 6 : Réutilisation des Classes

Le concept d'héritage se retrouve dans pratiquement tous les langages de POO. Il s'agit de prendre une classe existante et d'étendre ses fonctionnalités (ou tout aussi bien les modifier, c'est le sujet du chapitre 7). L'héritage consiste toujours à réutiliser du code en gardant la même « classe de base », et en modifiant simplement certaines choses çà et là afin d'obtenir ce que l'on veut. Toutefois, l'héritage n'est pas la seule manière de créer de nouvelles classes à partir de classes existantes. Il est également possible d'encapsuler un objet dans une nouvelle classe au moyen de la *composition*. Ce chapitre explique ces deux méthodes de réutilisation du code en Java, et comment les utiliser.

Chapitre 7 : Le Polymorphisme

Si vous appreniez par vous-même, il vous faudrait neuf mois pour découvrir et comprendre le polymorphisme, l'une des pierres angulaires de la POO. Des exemples simples et courts montreront comment créer une famille de types au moyen de l'héritage et comment manipuler les objets dans cette famille par l'intermédiaire de leur classe de base. Le polymorphisme de Java permet de traiter de manière générique tout objet d'une famille, ce qui signifie que la plus grande partie du code n'est pas liée à une information spécifique sur le type. Ceci rend les programmes extensibles, et donc leur développement et leur maintenance plus simples et moins onéreux.

Chapitre 8 : Interfaces & Classes Internes

Java fournit une troisième voie pour la réutilisation du code, avec l'*interface*, qui est une pure abstraction de l'interface d'un objet. L'**interface** est bien plus qu'une simple classe abstraite dont on aurait poussé l'abstraction à l'extrême, puisqu'il vous permet de développer une variation sur l'« héritage multiple » du C++, en créant une classe qui peut être transtypée vers plus d'un type de base.

Au premier abord, les classes internes ressemblent à un simple mécanisme permettant de cacher le code : on place des classes à l'intérieur d'autres classes. Vous apprendrez toutefois que la classe interne fait plus que cela - elle connaît la classe enveloppante et peut communiquer avec elle - et il est certain que le style de code que l'on écrit au moyen des classes internes est plus élégant et plus clair, bien que cela représente pour la plupart d'entre vous une nouvelle fonctionnalité nécessitant un certain temps d'apprentissage avant d'être maîtrisée.

Chapitre 9 : Stockage des Objets

Un programme qui manipule un nombre fixe d'objets dont la durée de vie est connue ne peut être que clair et très simple. Mais généralement, les programmes créent de nouveaux objets à différents moments, qui ne seront connus que lors de l'exécution. De plus, avant l'exécution, on ne connaît ni le nombre ni parfois le type exact des objets qui seront nécessaires. Afin de résoudre ce problème général de la programmation, nous devons pouvoir créer n'importe quel nombre d'objets, à n'importe quel moment, n'importe où. Ce chapitre explore en profondeur la bibliothèque fournie par Java 2 pour ranger les objets durant leur existence : les tableaux simples et les conteneurs plus sophistiqués (structures de données) comme **ArrayList** et **HashMap**.

Chapitre 10 : Traitement des Erreurs au Moyen des Exceptions

Java a pour philosophie de base qu'un code mal écrit ne sera jamais exécuté. Autant que possible, le compilateur repère les problèmes, mais parfois les problèmes - aussi bien une erreur de programmation qu'une condition d'erreur naturelle survenant lors de l'exécution normale du programme - ne peuvent être détectés et traités qu'au moment de l'exécution. Java possède un traitement des erreurs par les exceptions pour s'occuper de tout problème survenant pendant l'exécution. Ce chapitre examine comment fonctionnent en Java les mots clefs **try**, **catch**, **throw**, **throws**, et **fi-**

nally ; quand lancer des exceptions ; et ce que l'on doit faire si on les intercepte. Il expose aussi les exceptions standard de Java, comment créer vos propres exceptions, ce qu'il advient des exceptions dans les constructeurs, et comment sont localisés les codes de traitement d'exception.

Chapitre 11 : le Système d'E/S de Java

En théorie, on peut diviser n'importe quel programme en trois parties : entrée, traitement, et sortie des données. Ceci suggère que les E/S (entrées/sorties) représentent une part importante de n'importe quel problème. Ce chapitre étudie les différentes classes fournies par Java pour lire et écrire des fichiers, des blocs mémoire, ainsi que la console. Il montre la distinction entre E/S « vieux style » et E/S « nouveau style » Java. Il examine également le processus consistant à prendre un objet, le transformer en flux (de manière à pouvoir le ranger sur disque ou l'envoyer à travers un réseau) puis le reconstruire, ce qui est pris en charge par la *sérialisation des objets* de Java. Il présente également les bibliothèques de compression de Java, utilisées dans le format de fichier Java ARchive (JAR).

Chapitre 12 : Identification Dynamique de Type

L'identification dynamique de type de Java (Run-Time Type Identification, RTTI) permet de connaître le type exact d'un objet à partir d'une référence sur le type de base. Habituellement, on préfère ignorer intentionnellement le type exact d'un objet et laisser au mécanisme de liaison dynamique de Java (polymorphisme) le soin d'implémenter la signification correcte pour ce type. Mais de temps en temps il est très utile de connaître le type réel d'un objet pour lequel on n'a qu'une référence sur le type de base. Souvent cette information permet d'implémenter plus efficacement un traitement spécial. Ce chapitre explique à quoi sert la RTTI, comment l'utiliser, et comment s'en débarrasser lorsqu'on n'en a plus besoin. Enfin, il introduit le mécanisme de *réflexion* de Java.

Chapitre 13 : Créer des Fenêtres et des Applets

Java est livré avec la bibliothèque GUI « Swing », qui est un ensemble de classes traitant du fenêtrage d'une manière portable (NdT : sur différentes plates-formes). Ces programmes fenêtrés peuvent être soit des applets soit des applications autonomes. Ce chapitre est une introduction à Swing et à la création d'applets pour le World Wide Web. Il introduit aussi l'importante technologie des « JavaBeans », fondamentale pour la création d'outils de développement de programmes destinés au Développement Rapide d'Applications (RAD, Rapid-Application Development).

Chapitre 14 : Les Threads Multiples

Java fournit un moyen de créer de multiples sous-tâches concurrentes, appelées threads, s'exécutant dans le contexte d'un même programme (mis à part le cas où la machine possède plus d'un processeur, ceci n'a que l'apparence de sous-tâches multiples). Bien qu'on puisse les utiliser n'importe où, l'utilisation des threads est plus évidente lorsqu'il s'agit de créer une interface utilisateur réactive comme, par exemple, lorsqu'un certain processus gourmand en ressources système en cours d'exécution empêche un utilisateur d'utiliser un bouton ou d'entrer des données. Ce chapitre examine la syntaxe et la sémantique du multithreading en Java.

Chapitre 15 : Informatique Distribuée

Toutes les fonctionnalités et bibliothèques de Java semblent vraiment faites les unes pour les autres lorsqu'on commence à écrire des programmes qui travaillent en réseau. Ce chapitre explore la communication au travers des réseaux et sur l'Internet, ainsi que les classes fournies par Java pour faciliter cela. Il introduit les concepts très importants de *Servlets* et des *JSPs* (pour la programmation « côté serveur »), ainsi que la connectivité aux bases de données, *Java DataBase Connectivity* (JDBC), et l'invocation de méthodes distantes, *Remote Method Invocation* (RMI). Et, pour finir,

une introduction aux nouvelles technologies *JINI*, *JavaSpaces*, et *Enterprise JavaBeans* (EJB).

Annexe A : Passage & Retour d'Objets

Etant donné qu'en Java seules les références permettent d'appréhender les objets, le concept de « passer un objet à une fonction » et celui de « retourner un objet depuis une fonction » ont quelques conséquences intéressantes. Cette annexe explique ce qu'il faut savoir afin de gérer les objets à l'entrée et à la sortie d'une fonction, et montre également la classe **String**, qui utilise une approche différente du problème.

Annexe B : L'interface Java Natif (JNI)

Un programme Java entièrement portable a de sérieux inconvénients : la vitesse, et l'incapacité d'accéder à des services spécifiques de la plate-forme. Connaissant la plate-forme sur laquelle sera exécuté le programme, il est possible d'accélérer spectaculairement certaines opérations en les transformant en *méthodes natives*, qui sont des fonctions écrites dans un autre langage de programmation (actuellement, seuls C/C++ sont supportés). Cette annexe procure une courte introduction à cette fonctionnalité, suffisante pour qu'on puisse créer des exemples simples utilisant cette interface avec un code autre que Java.

Cet annexe est un ensemble de suggestions qui vous aideront dans la conception et le codage de bas niveau de votre application.

Annexe D : Ressources

Une liste des livres sur Java que m'ont paru particulièrement utile.

Exercices

Je me suis aperçu que des exercices simples sont très utiles pour consolider les connaissances des étudiants lors d'un séminaire, on en trouvera donc un ensemble à la fin de chaque chapitre.

La plupart d'entre eux sont conçus afin d'être assez simples pour être réalisés dans un temps raisonnable dans le contexte d'une salle de classe, pendant que l'instructeur vérifie que tous les étudiants ont assimilé le sujet de la leçon. Quelques exercices sont plus pointus, afin d'éviter l'ennui chez les étudiants expérimentés. La majorité est conçue pour être réalisés rapidement, ainsi que pour tester et perfectionner les connaissances. Quelques-uns présentent des difficultés, mais jamais de difficulté majeure. (Je présume que vous les découvrirez par vous-même — ou plutôt qu'ils vous trouveront).

Les solutions des exercices se trouvent dans le document électronique *The Thinking in Java Annotated Solution Guide*, disponible pour un faible coût sur http://www.BruceEckel.com.

Le CD ROM Multimédia

Deux CD ROM multimédia sont associés à ce livre. Le premier est fourni avec le livre lui-même : *Thinking in C*, décrit à la fin de la préface, et consiste en une préparation à ce livre qui décrit la syntaxe C nécessaire à la compréhension de Java.

Il existe un deuxième CD ROM Multimédia, basé sur le contenu du livre. Ce CD ROM est un produit séparé et contient la **totalité** du séminaire d'une semaine de formation Java « Hands-On Java ». J'y ai enregistré plus de 15 heures de conférence, synchronisées avec des centaines de diapositives d'information. C'est un accompagnement idéal, dans la mesure où le séminaire est basé sur ce livre.

Le CD ROM contient toutes les conférences du séminaire de formation en immersion totale de cinq jours (il ne traite pas de l'attention portée aux cas particuliers !). Nous espérons que cela définira un nouveau standard de qualité.

Le CD ROM « Hands-On Java » est uniquement disponible en le commandant directement sur le site *http://www.BruceEckel.com*.

Le Code Source

L'ensemble du code source de ce livre est disponible en freeware sous copyright, en une seule archive, en visitant le site Web *http://www.BruceEckel.com*. Afin que vous soyez certains d'obtenir la dernière version, cette adresse est celle du site officiel pour la distribution du code et de la version électronique du livre. Il existe des versions miroir du livre électronique et du code sur d'autres sites (dont certains sont référencés sur le site *http://www.BruceEckel.com*), mais il est préférable de rendre visite au site officiel afin de s'assurer que la version miroir est la plus récente. Vous êtes autorisés à distribuer le code à des fins d'enseignement ou d'éducation.

Le but essentiel du copyright est d'assurer que la source du code soit correctement citée, et d'éviter que le code soit utilisé sans autorisation dans un médium imprimé. (Tant que la source est citée, l'utilisation d'exemples provenant du livre ne pose généralement pas problème).

Chaque code source contient une référence à l'annonce suivante du copyright :

```
//:!:CopyRight.txt
Copyright ©2000 Bruce Eckel
Source code file from the 2nd edition of the book
"Thinking in Java." All rights reserved EXCEPT as
allowed by the following statements:
You can freely use this file
for your own work (personal or commercial),
including modifications and distribution in
executable form only. Permission is granted to use
this file in classroom situations, including its
use in presentation materials, as long as the book
"Thinking in Java" is cited as the source.
Except in classroom situations, you cannot copy
and distribute this code; instead, the sole
distribution point is http://www.BruceEckel.com
(and official mirror sites) where it is
freely available. You cannot remove this
copyright and notice. You cannot distribute
modified versions of the source code in this
package. You cannot use this file in printed
media without the express permission of the
author. Bruce Eckel makes no representation about
the suitability of this software for any purpose.
It is provided "as is" without express or implied
warranty of any kind, including any implied
warranty of merchantability, fitness for a
particular purpose or non-infringement. The entire
```

```
risk as to the quality and performance of the
software is with you. Bruce Eckel and the
publisher shall not be liable for any damages
suffered by you or any third party as a result of
using or distributing software. In no event will
Bruce Eckel or the publisher be liable for any
lost revenue, profit, or data, or for direct,
indirect, special, consequential, incidental, or
punitive damages, however caused and regardless of
the theory of liability, arising out of the use of
or inability to use software, even if Bruce Eckel
and the publisher have been advised of the
possibility of such damages. Should the software
prove defective, you assume the cost of all
necessary servicing, repair, or correction. If you
think you've found an error, please submit the
correction using the form you will find at
www.BruceEckel.com. (Please use the same
form for non-code errors found in the book.)
///:~
```

Vous êtes autorisés à utiliser le code pour vos projets ainsi qu'à des fins d'éducation (ceci incluant vos cours) à la condition de conserver le copyright inclus dans chaque fichier source.

Typographie et style de code

Dans ce livre, les identificateurs (de fonction, de variable, de nom de classe) sont écrits en **gras**. La plupart des mots clefs sont également en gras, à l'exception de ceux qui sont si souvent utilisés, tels que « class », que cela en deviendrait ennuyeux.

J'utilise un style de code particulier pour les exemples de ce livre. Ce style suit les règles que Sun utilise lui-même dans pratiquement tous les codes que l'on peut trouver sur son site (voir *java.sun.com/docs/codeconv/index.html*), et semble être pris en compte par la plupart des environnements de développement Java. Si vous avez lu mes autres livres, vous avez pu remarquer que le style de codage de Sun coïncide avec le mien — ce qui me fait évidemment plaisir, bien que je n'y sois pour rien. Le sujet du style de format demanderait des heures de débat assez chaud, aussi je vais simplement dire que je ne prétends pas imposer un style correct au travers de mes exemples, j'ai seulement mes propres motivations pour faire ainsi. Puisque Java est un langage de programmation indépendant de la forme, vous pouvez continuer à utiliser le style qui vous convient.

Les programmes de ce livre sont des fichiers directement inclus, au moyen du traitement de texte, depuis des fichiers ayant déjà subi une compilation. Par suite, le code imprimé dans le livre ne doit pas provoquer d'erreurs de compilation. Les erreurs qui *pourraient* entraîner des messages d'erreur lors de la compilation sont mis en commentaires au moyen de *//!* de manière à être facilement repérées et testées par des moyens automatiques. Les erreurs découvertes et rapportées à l'auteur feront d'abord l'objet d'une modification du code source distribué, puis, plus tard, d'une révision du livre (qui sera également disponible sur le site Web *http://www.BruceEckel.com*).

Les versions de Java

Je me réfère généralement à l'implémentation Sun de Java pour déterminer la démarche correcte.

Depuis le début, Sun a fourni trois versions majeures de Java : 1.0, 1.1 et 2 (laquelle est appelée version 2 même si les versions du JDK de Sun continuent à être numérotées 1.2, 1.3, 1.4, etc.). La version 2 semble définitivement mettre Java en lumière, en particulier lorsqu'il est question des outils d'interface utilisateur. Ce livre en parle et a été testé avec Java 2, bien que je fasse de temps en temps des concessions aux fonctionnalités futures de Java 2 afin que le code soit compilable sous Linux (avec le JDK Linux disponible alors que j'écrivais ceci).

Si vous désirez apprendre les versions antérieures du langage non couvertes par cette édition, la première édition de ce livre est librement téléchargeable à l'adresse url : *http://www.BruceEckel.com*, vous la trouverez également dans le CD livré avec ce livre.

Attention : lorsqu'il m'a fallu mentionner les versions antérieures du langage, je n'utilise pas les numéros de sous-révision. Dans ce livre je fais uniquement référence à Java 1.0, Java 1.1, et Java 2, afin de me prémunir contre les erreurs typographiques qui pourraient résulter de futures sous-révisions de ces produits.

Seminars and mentoring

(non traduit, les personnes intéressées par les séminaires de Bruce Eckel devant à priori maîtriser l'anglais) My company provides five-day, hands-on, public and in-house training seminars based on the material in this book. Selected material from each chapter represents a lesson, which is followed by a monitored exercise period so each student receives personal attention. The audio lectures and slides for the introductory seminar are also captured on CD ROM to provide at least some of the experience of the seminar without the travel and expense. For more information, go to *http://www.BruceEckel.com*.

My company also provides consulting, mentoring and walkthrough services to help guide your project through its development cycle — especially your company's first Java project.

Errors

(non traduit car cela concerne les erreurs relevées dans la version anglaise du livre de Bruce Eckel) No matter how many tricks a writer uses to detect errors, some always creep in and these often leap off the page for a fresh reader.

There is an error submission form linked from the beginning of each chapter in the HTML version of this book (and on the CD ROM bound into the back of this book, and downloadable from *http://www.BruceEckel.com*) and also on the Web site itself, on the page for this book. If you discover anything you believe to be an error, please use this form to submit the error along with your suggested correction. If necessary, include the original source file and note any suggested modifications. Your help is appreciated.

À propos de la conception de la couverture du livre

La couverture de *Thinking in Java* est inspirée par le Mouvement des Arts et Métiers Américain (American Arts & Crafts Movement), qui commença peu avant le XXe siècle et atteignit son zénith entre 1900 et 1920. Il vit le jour en Angleterre en réaction à la fois contre la production des ma-

chines de la Révolution Industrielle et contre le style hautement ornemental de l'ère Victorienne. Arts & Crafts mit l'accent sur la sobriété, les formes de la nature telles que les voyait le mouvement « art nouveau" (en français dans le texte, NdT), le travail manuel, et l'importance des travailleurs et artisans particuliers, et encore n'ont-ils pas dédaigné l'utilisation des outils modernes. Il y a beaucoup de ressemblances avec la situation actuelle : le tournant du siècle, l'évolution des débuts inexpérimentés de la révolution informatique vers quelque chose de plus raffiné et significatif pour les individus, et l'engouement pour la connaissance du métier de programmeur face à la fabrication industrielle de code.

Je considère Java de la même manière : une tentative pour élever le programmeur au-dessus de la mécanique du système d'exploitation afin de l'amener à devenir un « artisan du logiciel ».

L'auteur, tout autant que le concepteur du livre et de sa couverture (qui sont amis depuis l'enfance) ont trouvé leur inspiration dans ce mouvement, et tous deux possèdent des meubles, lampes etc. soit originaux, soit inspirés par cette période.

L'autre thème de cette couverture suggère une boîte servant à la présentation des spécimens d'insectes recueillis par un naturaliste. Ces insectes sont des objets, qui sont placés dans des boîtes-objets. Les boîtes-objets sont elles-mêmes placées dans la « couverture-objet », ce qui illustre le concept fondamental d'agrégation en programmation orientée objet. Bien entendu, ceci n'est d'aucune utilité pour un programmeur, mais crée une association avec les « punaises » (*« bugs »)*, ici les punaises ont été capturées, probablement tuées dans un bocal à spécimen, et pour finir confinées dans une petite boîte pour être exposées, comme pour montrer la capacité de Java à trouver, montrer, et soumettre les bugs (ce qui est vraiment l'un de ses attributs les plus puissants).

Remerciements

En premier, merci à tous les associés qui travaillèrent avec moi pour encadrer des séminaires, faire du consulting, et développer des projets éducatifs : Andrea Provaglio, Dave Bartlett (qui a par ailleurs fortement contribué au Chapitre 15), Bill Venners, et Larry O'Brien. J'ai apprécié leur patience alors que je continuais de développer le meilleur modèle permettant à des personnes indépendantes comme nous de travailler ensemble. Merci à Rolf André Klaedtke (Suisse) ; Martin Vlcek, Martin Byer, Vlada & Pavel Lahoda, Martin the Bear, et Hanka (Prague) ; ainsi que Marco Cantu (Italie) pour leur hébergement lors de ma première tournée de conférences improvisée en Europe.

Merci aussi à la « Doyle Street Cohousing Community » pour m'avoir supporté durant les deux années nécessaires à la rédaction de cette première édition (ainsi que pour m'avoir supporté dans l'absolu). Bien des remerciements à Kevin et Sonda Donovan pour m'avoir accueilli dans leur magnifique Crested Butte, Colorado, l'été où je travaillais à la première édition du livre. Merci aussi à tous les amis résidents de « Crested Butte » et au « Rocky Mountain Biological Laboratory » qui m'ont si bien accueilli.

Merci également à Claudette Moore de « Moore Literary Agency » pour son énorme patience et sa constance à m'obtenir exactement ce je désirais.

Mes deux premiers livres ont été publiés sous la houlette de l'éditeur Jeff Pepper aux éditions Osborne/McGraw-Hill. Jeff est arrivé au bon endroit au bon moment à Prentice-Hall, il a débroussaillé le chemin et fait tout ce qu'il fallait pour rendre cette expérience de publication très agréable. Merci, Jeff — cela a compté pour moi.

J'ai une dette spéciale envers Gen Kiyooka et sa compagnie Digigami, qui m'ont gracieusement fourni un serveur Web les premières années. Cela a représenté pour moi une aide inestimable.

Merci à Cay Horstmann (co-auteur de *Core Java*, Prentice-Hall, 2000*)*, D'Arcy Smith (Symantec), et Paul Tyma (co-auteur de *Java Primer Plus*, The Waite Group, 1996), pour m'avoir aidé à clarifier les concepts du langage.

Merci aux personnes qui ont pris la parole dans mon cursus Java à la Software Development Conference, aux étudiants de mes séminaires, pour m'avoir posé les bonnes questions qui m'ont permis de clarifier mes cours.

Je remercie spécialement Larry et Tina O'Brien, qui m'ont aidé à mettre mon séminaire sur le CD ROM original *Hands-On Java* (vous en saurez davantage sur *http://www.BruceEckel.com*).

Beaucoup de gens m'ont envoyé des correctifs et je reste en dette avec eux, mais je dois remercier particulièrement (pour la première édition) : Kevin Raulerson (qui repéra des tonnes d'énormes bugs), Bob Resendes (tout simplement incroyable), John Pinto, Joe Dante, Joe Sharp (les trois, fabuleux), David Combs (beaucoup de corrections grammaticales et d'éclaircissements), Dr. Robert Stephenson, John Cook, Franklin Chen, Zev Griner, David Karr, Leander A. Stroschein, Steve Clark, Charles A. Lee, Austin Maher, Dennis P. Roth, Roque Oliveira, Douglas Dunn, Dejan Ristic, Neil Galarneau, David B. Malkovsky, Steve Wilkinson, ainsi qu'une foule d'autres. Prof. Ir. Marc Meurrens déploya beaucoup d'efforts de publication et réalisa la version électronique de la première édition du livre disponible en Europe.

J'ai rencontré dans ma vie une avalanche de personnes très techniques et très intelligentes qui sont devenues des amis mais qui étaient également peu communes et qui m'ont influencé en ce qu'elles pratiquaient le yoga ainsi que d'autres formes de spiritualité, ce qui m'a instruit et inspiré. Ce sont Kraig Brockschmidt, Gen Kiyooka, et Andrea Provaglio (qui aida à la compréhension de Java et de la programmation en général en Italie, et qui est maintenant aux Etats-Unis associé à l'équipe MindView).

Ce ne fut pas une grande surprise pour moi de découvrir que ma connaissance de Delphi m'a aidé à comprendre Java, car ces deux langages ont beaucoup de concepts et de décisions de conception de langage en commun. Des amis fanatiques de Delphi m'ont aidé à devenir plus perspicace à propos de ce merveilleux environnement de programmation. Ce sont Marco Cantu (un autre italien — il se pourrait qu'être imprégné de culture latine donne certaines aptitudes pour la programmation ?), Neil Rubenking (qui se nourrissait de culture yoga/végétarienne/Zen avant de découvrir les ordinateurs), et bien entendu Zack Urlocker, un vieux copain avec qui j'ai parcouru le monde.

La perspicacité et l'aide de mon ami Richard Hale Shaw m'ont été fort utiles (celles de Kim également). Richard et moi avons passé de concert beaucoup de mois à donner des séminaires et à tenter de trouver l'enseignement parfait pour les auditeurs. Merci également à KoAnn Vikoren, Eric Faurot, Marco Pardi, ainsi qu'à toute l'équipe du MFI. Merci particulièrement à Tara Arrowood, qui me rendit confiance à propos des possibilités de conférences.

La conception du livre, de la couverture, ainsi que la photo de couverture sont dûs à mon ami Will-Harris, auteur et concepteur connu (*http://www.Will-Harris.com*), qui jouait au collège avec des lettres transfert en attendant l'invention de la publication assistée par ordinateur, tout en se plaignant de mes grognements à propos de mes problèmes d'algèbre. Toutefois, j'ai produit moi-même mes pages prêtes à imprimer, et donc j'assume mes erreurs de frappe. Microsoft® Word 97 for Windows a été utilisé pour écrire le livre et Adobe Acrobat pour créer les pages offset ; le livre est sorti directement des fichiers PDF d'Acrobat (pour rendre hommage à l'ère électronique, je me trouvais outre-mer les deux fois où la version finale du livre fut produite — la première édition fut envoyée depuis Le Cap en Afrique du Sud et la seconde depuis Prague). Les polices de caractères sont *Georgia* pour le corps du texte et *Verdana* pour les titres. La police de couverture est *ITC Rennie*

Mackintosh.

Merci aux groupes qui ont créé les compilateurs : Borland, le Blackdown group (pour Linux), et bien entendu, Sun.

Je remercie particulièrement tous mes maîtres et tous mes étudiants (qui furent en même temps mes maîtres). Le plus plaisant de mes maîtres en écriture fut Gabrielle Rico (auteur de *Writing the Natural Way*, Putnam, 1983). Je garderai précieusement le souvenir d'une formidable semaine à Esalen.

Liste non exhaustive de mes collaborateurs : Andrew Binstock, Steve Sinofsky, JD Hildebrandt, Tom Keffer, Brian McElhinney, Brinkley Barr, Bill Gates au *Midnight Engineering Magazine*, Larry Constantine et Lucy Lockwood, Greg Perry, Dan Putterman, Christi Westphal, Gene Wang, Dave Mayer, David Intersimone, Andrea Rosenfield, Claire Sawyers, d'autres italiens (Laura Fallai, Corrado, Ilsa, et Cristina Giustozzi), Chris et Laura Strand, les Almquists, Brad Jerbic, Marilyn Cvitanic, les Mabrys, les Haflingers, les Pollocks, Peter Vinci, la famille Robbins, la famille Moelter (ainsi que les McMillans), Michael Wilk, Dave Stoner, Laurie Adams, les Cranstons, Larry Fogg, Mike et Karen Sequeira, Gary Entsminger et Allison Brody, Kevin Donovan et Sonda Eastlack, Chester et Shannon Andersen, Joe Lordi, Dave et Brenda Bartlett, David Lee, les Rentschlers, les Sudeks, Dick, Patty, et Lee Eckel, Lynn et Todd, et leurs familles. Et, bien entendu, Papa et Maman.

Collaborateurs Internet

Merci à tous ceux qui m'ont aidé à réécrire les exemples au moyen de la bibliothèque Swing, ou pour d'autres choses : Jon Shvarts, Thomas Kirsch, Rahim Adatia, Rajesh Jain, Ravi Manthena, Banu Rajamani, Jens Brandt, Nitin Shivaram, Malcolm Davis, ainsi qu'à tous ceux qui se sont exprimés. Cela m'a réellement aidé à mettre le projet à jour.

Chapitre 1 - Introduction sur les « objets »

La révolution informatique a pris naissance dans une machine. Nos langages de programmation ont donc tendance à ressembler à cette machine.

Mais les ordinateurs ne sont pas tant des machines que des outils au service de l'esprit (« des vélos pour le cerveau », comme aime à le répéter Steve Jobs) et un nouveau moyen d'expression. Ainsi, ces outils commencent à moins ressembler à des machines et plus à des parties de notre cerveau ou d'autres formes d'expressions telles que l'écriture, la peinture, la sculpture ou la réalisation de films. La Programmation Orientée Objet (POO) fait partie de ce mouvement qui utilise l'ordinateur en tant que moyen d'expression.

Ce chapitre présente les concepts de base de la POO, y compris quelques méthodes de développement. Ce chapitre et ce livre présupposent que vous avez déjà expérimenté avec un langage de programmation procédural, bien que celui-ci ne soit pas forcément le C. Si vous pensez que vous avez besoin de plus de pratique dans la programmation et / ou la syntaxe du C avant de commencer ce livre, vous devriez explorer le CD ROM fourni avec le livre, *Thinking in C: Foundations for C++ and Java*, aussi disponible sur *www.BruceEckel.com*.

Ce chapitre tient plus de la culture générale. Beaucoup de personnes ne veulent pas se lancer dans la programmation orientée objet sans en comprendre d'abord les tenants et les aboutissants. C'est pourquoi nous allons introduire ici de nombreux concepts afin de vous donner un solide aperçu de la POO. Au contraire, certaines personnes ne saisissent les concepts généraux qu'après en avoir vu quelques mécanismes mis en oeuvre ; ces gens-là se sentent perdus s'ils n'ont pas un bout de code à se mettre sous la dent. Si vous faites partie de cette catégorie de personnes et êtes impatients d'attaquer les spécificités du langage, vous pouvez sauter ce chapitre - cela ne vous gênera pas pour l'écriture de programme ou l'apprentissage du langage. Mais vous voudrez peut-être y revenir plus tard pour approfondir vos connaissances sur les objets, les comprendre et assimiler la conception objet.

Les bienfaits de l'abstraction

Tous les langages de programmation fournissent des abstractions. On peut dire que la complexité des problèmes qu'on est capable de résoudre est directement proportionnelle au type et à la qualité de nos capacités d'abstraction. Par « type », il faut comprendre « Qu'est-ce qu'on tente d'abstraire ? ». Le langage assembleur est une petite abstraction de la machine sous-sous-jacente. Beaucoup de langages « impératifs » (tels que Fortran, BASIC, et C) sont des abstractions du langage assembleur. Ces langages sont de nettes améliorations par rapport à l'assembleur, mais leur abstraction première requiert une réflexion en termes de structure ordinateur plutôt qu'à la structure du problème qu'on essaye de résoudre. Le programmeur doit établir l'association entre le modèle de la machine (dans « l'espace solution », qui est l'endroit où le problème est modélisé, tel que l'ordinateur) et le modèle du problème à résoudre (dans « l'espace problème », qui est l'endroit où se trouve le problème). Les efforts requis pour réaliser cette association, et le fait qu'elle est étrangère au langage de programmation, produit des programmes difficiles à écrire et à maintenir, et comme effet de bord a mené à la création de l'industrie du « Génie Logiciel ».

L'autre alternative à la modélisation de la machine est de modéliser le problème qu'on tente de résoudre. Les premiers langages tels que LISP ou APL choisirent une vue particulière du monde (« Tous les problèmes se ramènent à des listes » ou « Tous les problèmes sont algorithmiques », res-

pectivement). PROLOG convertit tous les problèmes en chaînes de décisions. Des langages ont été créés en vue de programmer par contrainte, ou pour programmer en ne manipulant que des symboles graphiques (ces derniers se sont révélés être trop restrictifs). Chacune de ces approches est une bonne solution pour la classe particulière de problèmes pour laquelle ils ont été conçus, mais devient une horreur dès lors que vous les sortez de leur domaine d'application.

L'approche orientée objet va un cran plus loin en fournissant des outils au programmeur pour représenter des éléments dans l'espace problème. Cette représentation se veut assez générale pour ne pas restreindre le programmeur à un type particulier de problèmes. Nous nous référons aux éléments dans l'espace problème et leur représentation dans l'espace solution en tant qu'« objets ». (Bien sûr, on aura aussi besoin d'autres objets qui n'ont pas leur analogue dans l'espace problème). L'idée est que le programme est autorisé à s'adapter à l'esprit du problème en ajoutant de nouveaux types d'objet, de façon à ce que, quand on lit le code décrivant la solution, on lit aussi quelque chose qui décrit le problème. C'est une abstraction plus flexible et puissante que tout ce qu'on a pu voir jusqu'à présent. Ainsi, la POO permet de décrire le problème avec les termes mêmes du problème plutôt qu'avec les termes de la machine où la solution sera mise en oeuvre. Il y a tout de même une connexion avec l'ordinateur, bien entendu. Chaque objet ressemble à un mini-ordinateur ; il a un état, et il a à sa disposition des opérations qu'on peut lui demander d'exécuter. Cependant, là encore on retrouve une analogie avec les objets du monde réel - ils ont tous des caractéristiques et des comportements.

Des concepteurs de langage ont décrété que la programmation orientée objet en elle-même n'était pas adéquate pour résoudre facilement tous les problèmes de programmation, et recommandent la combinaison d'approches variées dans des langages de programmation *multiparadigmes*. [2]

Alan Kay résume les cinq caractéristiques principales de Smalltalk, le premier véritable langage de programmation orienté objet et l'un des langages sur lequel est basé Java. Ces caractéristiques représentent une approche purement orientée objet :

1. **Toute chose est un objet.** Il faut penser à un objet comme à une variable améliorée : il stocke des données, mais on peut « effectuer des requêtes » sur cet objet, lui demander de faire des opérations sur lui-même. En théorie, on peut prendre n'importe quel composant conceptuel du problème qu'on essaye de résoudre (un chien, un immeuble, un service administratif, etc...) et le représenter en tant qu'objet dans le programme.

2. **Un programme est un ensemble d'objets se disant les uns aux autres quoi faire en s'envoyant des messages.** Pour qu'un objet effectue une requête, on « envoie un message » à cet objet. Plus concrètement, on peut penser à un message comme à un appel de fonction appartenant à un objet particulier.

3. **Chaque objet a son propre espace de mémoire composé d'autres objets.** Dit d'une autre manière, on crée un nouveau type d'objet en créant un paquetage contenant des objets déjà existants. Ainsi, la complexité d'un programme est cachée par la simplicité des objets mis en oeuvre.

4. **Chaque objet est d'un type précis.** Dans le jargon de la POO, chaque objet est une *instance* d'une *classe*, où « classe » est synonyme de « type ». La plus importante caractéristique distinctive d'une classe est : « Quels messages peut-on lui envoyer ? ».

5. **Tous les objets d'un type particulier peuvent recevoir le même message.** C'est une caractéristique lourde de signification, comme vous le verrez plus tard. Parce qu'un objet de type « cercle » est aussi un objet de type « forme géométrique », un cercle se doit d'accepter

les messages destinés aux formes géométriques. Cela veut dire qu'on peut écrire du code parlant aux formes géométriques qui sera accepté par tout ce qui correspond à la description d'une forme géométrique. Cette *substituabilité* est l'un des concepts les plus puissants de la programmation orientée objet.

Un objet dispose d'une interface

Aristote fut probablement le premier à commencer une étude approfondie du concept de *type* ; il parle de « la classe des poissons et la classe des oiseaux ». L'idée que tous les objets, tout en étant uniques, appartiennent à une classe d'objets qui ont des caractéristiques et des comportements en commun fut utilisée directement dans le premier langage orienté objet, Simula-67, avec son mot clef fondamental **class** qui introduit un nouveau type dans un programme.

Simula, comme son nom l'indique, a été conçu pour développer des simulations telles qu'un guichet de banque. Dans celle-ci, vous avez un ensemble de guichetiers, de clients, de comptes, de transactions et de devises - un tas « d'objets ». Des objets semblables, leur état durant l'exécution du programme mis à part, sont groupés ensemble en tant que « classes d'objets » et c'est de là que vient le mot clef **class**. Créer des types de données abstraits (des classes) est un concept fondamental dans la programmation orientée objet. On utilise les types de données abstraits exactement de la même manière que les types de données prédéfinis. On peut créer des variables d'un type particulier (appelés *objets* ou *instances* dans le jargon OO) et manipuler ces variables (ce qu'on appelle *envoyer des messages* ou des *requêtes* ; on envoie un message et l'objet se débrouille pour le traiter). Les membres (éléments) d'une même classe partagent des caractéristiques communes : chaque compte dispose d'un solde, chaque guichetier peut accepter un dépôt, etc... Cependant, chaque élément a son propre état : chaque compte a un solde différent, chaque guichetier a un nom. Ainsi, les guichetiers, clients, comptes, transactions, etc... peuvent tous être représentés par la même entité au sein du programme. Cette entité est l'objet, et chaque objet appartient à une classe particulière qui définit ses caractéristiques et ses comportements.

Donc, comme la programmation orientée objet consiste en la création de nouveaux types de données, quasiment tous les langages orientés objet utilisent le mot clef « class ». Quand vous voyez le mot « type » pensez « classe » et inversement [3].

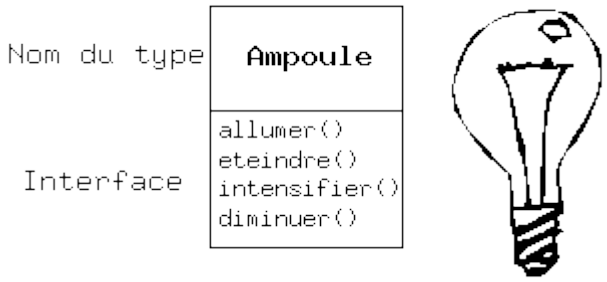
Comme une classe décrit un ensemble d'objets partageant des caractéristiques communes (données) et des comportements (fonctionnalités), une classe est réellement un type de données. En effet, un nombre en virgule flottante par exemple, dispose d'un ensemble de caractéristiques et de comportements. La différence est qu'un programmeur définit une classe pour représenter un problème au lieu d'être forcé d'utiliser un type de données conçu pour représenter une unité de stockage de l'ordinateur. Le langage de programmation est étendu en ajoutant de nouveaux types de données spécifiques à nos besoins. Le système de programmation accepte la nouvelle classe et lui donne toute l'attention et le contrôle de type qu'il fournit aux types prédéfinis.

L'approche orientée objet n'est pas limitée aux simulations. Que vous pensiez ou non que tout programme n'est qu'une simulation du système qu'on représente, l'utilisation des techniques de la POO peut facilement réduire un ensemble de problèmes à une solution simple.

Une fois qu'une classe est créée, on peut créer autant d'objets de cette classe qu'on veut et les manipuler comme s'ils étaient les éléments du problème qu'on tente de résoudre. En fait, l'une des difficultés de la programmation orientée objet est de créer une association un-à-un entre les éléments de l'espace problème et les éléments de l'espace solution.

Mais comment utiliser un objet ? Il faut pouvoir lui demander d'exécuter une requête, telle

que terminer une transaction, dessiner quelque chose à l'écran, ou allumer un interrupteur. Et chaque objet ne peut traiter que certaines requêtes. Les requêtes qu'un objet est capable de traiter sont définies par son *interface*, et son type est ce qui détermine son interface. Prenons l'exemple d'une ampoule électrique :




```
Ampoule amp = new Ampoule();
amp.allumer();
```

L'interface précise *quelles* opérations on peut effectuer sur un objet particulier. Cependant, il doit exister du code quelque part pour satisfaire cette requête. Ceci, avec les données cachées, constitue l'*implémentation*. Du point de vue de la programmation procédurale, ce n'est pas si compliqué. Un type dispose d'une fonction associée à chaque requête possible, et quand on effectue une requête particulière sur un objet, cette fonction est appelée. Ce mécanisme est souvent résumé en disant qu'on « envoie un message » (fait une requête) à un objet, et l'objet se débrouille pour l'interpréter (il exécute le code associé).

Ici, le nom du type / de la classe est **Ampoule**, le nom de l'objet **Ampoule** créé est **amp**, et on peut demander à un objet **Ampoule** de s'allumer, de s'éteindre, d'intensifier ou de diminuer sa luminosité. Un objet **Ampoule** est créé en définissant une « référence » (**amp**) pour cet objet et en appelant **new** pour créer un nouvel objet de ce type. Pour envoyer un message à cet objet, il suffit de spécifier le nom de l'objet suivi de la requête avec un point entre les deux. Du point de vue de l'utilisateur d'une classe prédéfinie, c'est tout ce qu'il est besoin de savoir pour programmer avec des objets.

L'illustration ci-dessus reprend le formalisme UML (*Unified Modeling Language*). Chaque classe est représentée par une boîte, avec le nom du type dans la partie supérieure, les *données membres* qu'on décide de décrire dans la partie du milieu et les *fonctions membres* (les fonctions appartenant à cet objet qui reçoivent les messages envoyées à cet objet) dans la partie du bas de la boîte. Souvent on ne montre dans les diagrammes UML que le nom de la classe et les fonctions publiques, et la partie du milieu n'existe donc pas. Si seul le nom de la classe nous intéresse, alors la portion du bas n'a pas besoin d'être montrée non plus.

L'implémentation cachée

Il est plus facile de diviser les personnes en *créateurs de classe* (ceux qui créent les nouveaux types de données) et *programmeurs clients* [4] (ceux qui utilisent ces types de données dans leurs applications). Le but des programmeurs clients est de se monter une boîte à outils pleine de classes réutilisables pour le développement rapide d'applications (RAD, Rapid Application Development en

anglais). Les créateurs de classes, eux, se focalisent sur la construction d'une classe qui n'expose que le nécessaire aux programmeurs clients et cache tout le reste. Pourquoi cela ? Parce que si c'est caché, le programmeur client ne peut l'utiliser, et le créateur de la classe peut changer la portion cachée comme il l'entend sans se préoccuper de l'impact que cela pourrait avoir chez les utilisateurs de sa classe. La portion cachée correspond en général aux données de l'objet qui pourraient facilement être corrompues par un programmeur client négligent ou mal informé. Ainsi, cacher l'implémentation réduit considérablement les bugs.

Le concept d'implémentation cachée ne saurait être trop loué : dans chaque relation il est important de fixer des frontières respectées par toutes les parties concernées. Quand on crée une bibliothèque, on établit une relation avec un programmeur client, programmeur qui crée une application (ou une bibliothèque plus conséquente) en utilisant notre bibliothèque.

Si tous les membres d'une classe sont accessibles pour tout le monde, alors le programmeur client peut faire ce qu'il veut avec cette classe et il n'y a aucun moyen de faire respecter certaines règles. Même s'il est vraiment préférable que l'utilisateur de la classe ne manipule pas directement certains membres de la classe, sans contrôle d'accès il n'y a aucun moyen de l'empêcher : tout est exposé à tout le monde.

La raison première du contrôle d'accès est donc d'empêcher les programmeurs clients de toucher à certaines portions auxquelles ils ne devraient pas avoir accès - les parties qui sont nécessaires pour les manipulations internes du type de données mais n'appartiennent pas à l'interface dont les utilisateurs ont besoin pour résoudre leur problème. C'est en réalité un service rendu aux utilisateurs car ils peuvent voir facilement ce qui est important pour leurs besoins et ce qu'ils peuvent ignorer.

La deuxième raison d'être du contrôle d'accès est de permettre au concepteur de la bibliothèque de changer le fonctionnement interne de la classe sans se soucier des effets que cela peut avoir sur les programmeurs clients. Par exemple, on peut implémenter une classe particulière d'une manière simpliste afin d'accélérer le développement, et se rendre compte plus tard qu'on a besoin de la réécrire afin de gagner en performances. Si l'interface et l'implémentation sont clairement séparées et protégées, cela peut être réalisé facilement.

Java utilise trois mots clefs pour fixer des limites au sein d'une classe : **public**, **private** et **protected**. Leur signification et leur utilisation est relativement explicite. Ces s*pécificateurs d'accès* déterminent qui peut utiliser les définitions qui suivent. **public** veut dire que les définitions suivantes sont disponibles pour tout le monde. Le mot clef **private**, au contraire, veut dire que personne, le créateur de la classe et les fonctions internes de ce type mis à part, ne peut accéder à ces définitions. **private** est un mur de briques entre le créateur de la classe et le programmeur client. Si quelqu'un tente d'accéder à un membre défini comme **private**, ils récupèreront une erreur lors de la compilation. **protected** se comporte comme **private**, en moins restrictif : une classe dérivée a accès aux membres **protected**, mais pas aux membres **private**. L'héritage sera introduit bientôt.

Java dispose enfin d'un accès « par défaut », utilisé si aucun de ces spécificateurs n'est mentionné. Cet accès est souvent appelé accès « amical » car les classes peuvent accéder aux membres amicaux des autres classes du même package, mais en dehors du package ces mêmes membres amicaux se comportent comme des attributs **private**.

Réutilisation de l'implémentation

Une fois qu'une classe a été créée et testée, elle devrait (idéalement) représenter une partie de code utile. Il s'avère que cette réutilisabilité n'est pas aussi facile à obtenir que cela ; cela demande de l'expérience et de l'anticipation pour produire un bon design. Mais une fois bien conçue, cette

classe ne demande qu'à être réutilisée. La réutilisation de code est l'un des plus grands avantages que les langages orientés objets fournissent.

La manière la plus simple de réutiliser une classe est d'utiliser directement un objet de cette classe, mais on peut aussi placer un objet de cette classe à l'intérieur d'une nouvelle classe. On appelle cela « créer un objet membre ». La nouvelle classe peut être constituée de n'importe quel nombre d'objets d'autres types, selon la combinaison nécessaire pour que la nouvelle classe puisse réaliser ce pour quoi elle a été conçue. Parce que la nouvelle classe est composée à partir de classes existantes, ce concept est appelé *composition* (ou, plus généralement, *agrégation*). On se réfère souvent à la composition comme à une relation « possède-un », comme dans « une voiture possède un moteur ».

(Le diagramme UML ci-dessus indique la composition avec le losange rempli, qui indique qu'il y a un moteur dans une voiture. J'utiliserai une forme plus simple : juste une ligne, sans le losange, pour indiquer une association [5].)

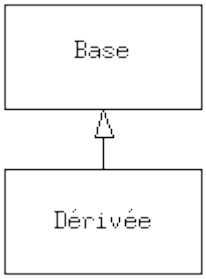
La composition s'accompagne d'une grande flexibilité : les objets membres de la nouvelle classe sont généralement privés, ce qui les rend inaccessibles aux programmeurs clients de la classe. Cela permet de modifier ces membres sans perturber le code des clients existants. On peut aussi changer les objets membres lors la phase d'exécution, pour changer dynamiquement le comportement du programme. L'héritage, décrit juste après, ne dispose pas de cette flexibilité car le compilateur doit placer des restrictions lors de la compilation sur les classes créées avec héritage.

Parce que la notion d'héritage est très importante au sein de la programmation orientée objet, elle est trop souvent martelée, et le nouveau programmeur pourrait croire que l'héritage doit être utilisé partout. Cela mène à des conceptions ultra compliquées et cauchemardesques. La composition est la première approche à examiner lorsqu'on crée une nouvelle classe, car elle est plus simple et plus flexible. Le design de la classe en sera plus propre. Avec de l'expérience, les endroits où utiliser l'héritage deviendront raisonnablement évidents.

Héritage : réutilisation de l'interface

L'idée d'objet en elle-même est un outil efficace. Elle permet de fournir des données et des fonctionnalités liées entre elles par concept, afin de représenter une idée de l'espace problème plutôt que d'être forcé d'utiliser les idiomes internes de la machine. Ces concepts sont exprimés en tant qu'unité fondamentale dans le langage de programmation en utilisant le mot clef **class**.

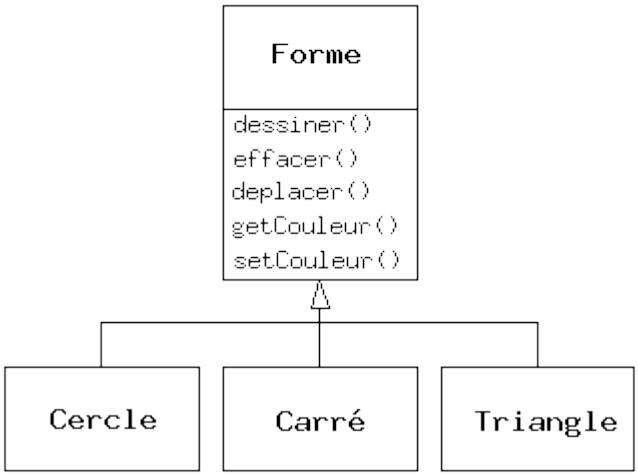
Il serait toutefois dommage, après s'être donné beaucoup de mal pour créer une classe de devoir en créer une toute nouvelle qui aurait des fonctionnalités similaires. Ce serait mieux si on pouvait prendre la classe existante, la cloner, et faire des ajouts ou des modifications à ce clone. C'est ce que l'*héritage* permet de faire, avec la restriction suivante : si la classe originale (aussi appelée classe de *base*, *superclasse* ou classe *parent*) est changée, le « clone » modifié (appelé classe *dérivée*, *héritée*, *enfant* ou *sousclasse*) répercutera aussi ces changements.



(La flèche dans le diagramme UML ci-dessus pointe de la classe dérivée vers la classe de base. Comme vous le verrez, il peut y avoir plus d'une classe dérivée.)

Prenons l'exemple d'une machine de recyclage qui trie les détritus. Le type de base serait « détritus », caractérisé par un poids, une valeur, etc.. et peut être concassé, fondu, ou décomposé. A partir de ce type de base, sont dérivés des types de détritus plus spécifiques qui peuvent avoir des caractéristiques supplémentaires (une bouteille a une couleur) ou des actions additionnelles (une canette peut être découpée, un container d'acier est magnétique). De plus, des comportements peuvent être différents (la valeur du papier dépend de son type et de son état général). En utilisant l'héritage, on peut bâtir une hiérarchie qui exprime le problème avec ses propres termes.

Un autre exemple classique : les « formes géométriques », utilisées entre autres dans les systèmes d'aide à la conception ou dans les jeux vidéos. Le type de base est la « forme géométrique », et chaque forme a une taille, une couleur, une position, etc... Chaque forme peut être dessinée, effacée, déplacée, peinte, etc... A partir de ce type de base, des types spécifiques sont dérivés (hérités) : des cercles, des carrés, des triangles et autres, chacun avec des caractéristiques et des comportements additionnels (certaines figures peuvent être inversées par exemple). Certains comportements peuvent être différents, par exemple quand on veut calculer l'aire de la forme. La hiérarchie des types révèle à la fois les similarités et les différences entre les formes.



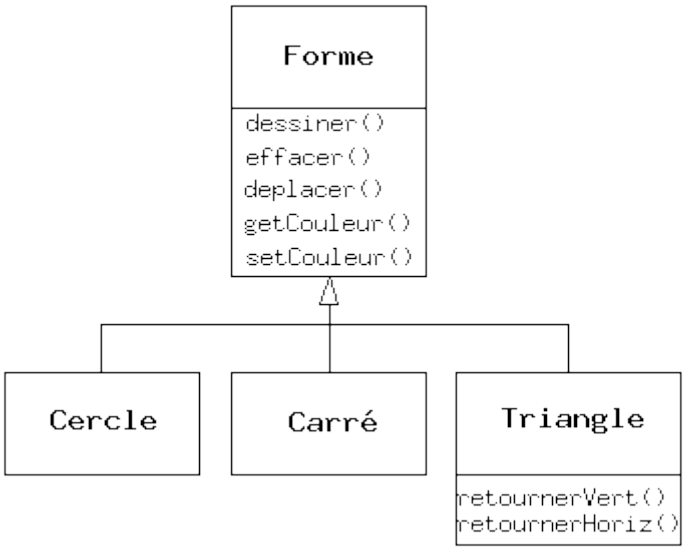
Représenter la solution avec les mêmes termes que ceux du problème est extraordinairement bénéfique car on n'a pas besoin de modèles intermédiaires pour passer de la description du problème à la description de la solution. Avec les objets, la hiérarchie de types est le modèle primaire, on passe donc du système dans le monde réel directement au système du code. En fait, l'une des diffi-

cultés à laquelle les gens se trouvent confrontés lors de la conception orientée objet est que c'est trop simple de passer du début à la fin. Les esprits habitués à des solutions compliquées sont toujours stupéfaits par cette simplicité.

Quand on hérite d'un certain type, on crée un nouveau type. Ce nouveau type non seulement contient tous les membres du type existant (bien que les membres privés soient cachés et inaccessibles), mais plus important, il duplique aussi l'interface de la classe de la base. Autrement dit, tous les messages acceptés par les objets de la classe de base seront acceptés par les objets de la classe dérivée. Comme on connaît le type de la classe par les messages qu'on peut lui envoyer, cela veut dire que la classe dérivée *est du même type que la classe de base*. Dans l'exemple précédent, « un cercle est une forme ». Cette équivalence de type via l'héritage est l'une des notions fondamentales dans la compréhension de la programmation orientée objet.

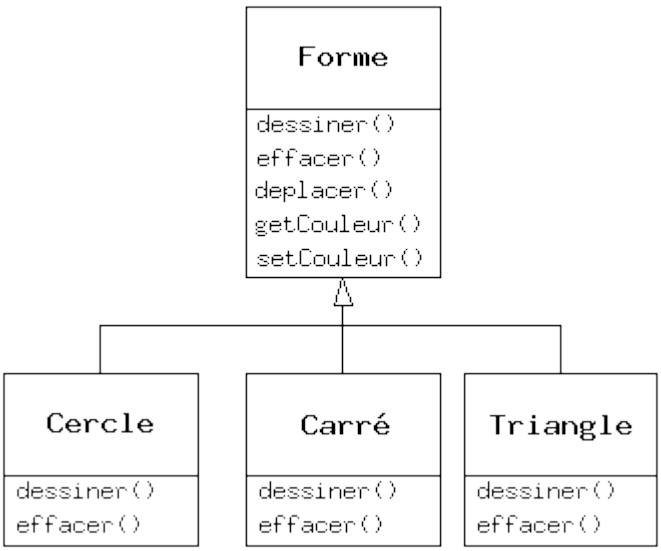
Comme la classe de base et la classe dérivée ont toutes les deux la même interface, certaines implémentations accompagnent cette interface. C'est à dire qu'il doit y avoir du code à exécuter quand un objet reçoit un message particulier. Si on ne fait qu'hériter une classe sans rien lui rajouter, les méthodes de l'interface de la classe de base sont importées dans la classe dérivée. Cela veut dire que les objets de la classe dérivée n'ont pas seulement le même type, ils ont aussi le même comportement, ce qui n'est pas particulièrement intéressant.

Il y a deux façons de différencier la nouvelle classe dérivée de la classe de base originale. La première est relativement directe : il suffit d'ajouter de nouvelles fonctions à la classe dérivée. Ces nouvelles fonctions ne font pas partie de la classe parent. Cela veut dire que la classe de base n'était pas assez complète pour ce qu'on voulait en faire, on a donc ajouté de nouvelles fonctions. Cet usage simple de l'héritage se révèle souvent être une solution idéale. Cependant, il faut tout de même vérifier s'il ne serait pas souhaitable d'intégrer ces fonctions dans la classe de base qui pourrait aussi en avoir l'usage. Ce processus de découverte et d'itération dans la conception est fréquent dans la programmation orientée objet.



Bien que l'héritage puisse parfois impliquer (spécialement en Java, où le mot clef qui indique l'héritage est **extends**) que de nouvelles fonctions vont être ajoutées à l'interface, ce n'est pas toujours vrai. La seconde et plus importante manière de différencier la nouvelle classe est de *changer* le comportement d'une des fonctions existantes de la superclasse. Cela s'appelle *redéfinir* cette fonc-

tion.



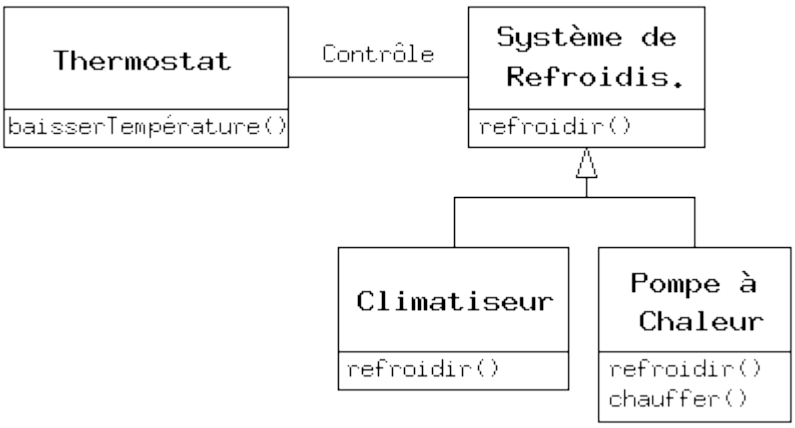
Pour redéfinir une fonction, il suffit de créer une nouvelle définition pour la fonction dans la classe dérivée. C'est comme dire : « j'utilise la même interface ici, mais je la traite d'une manière différente dans ce nouveau type ».

Les relations est-un vs. est-comme-un

Un certain débat est récurrent à propos de l'héritage : l'héritage ne devrait-il pas *seulement* redéfinir les fonctions de la classe de base (et ne pas ajouter de nouvelles fonctions membres qui ne font pas partie de la superclasse) ? Cela voudrait dire que le type dérivé serait *exactement* le même que celui de la classe de base puisqu'il aurait exactement la même interface. Avec comme conséquence logique le fait qu'on puisse exactement substituer un objet de la classe dérivée à un objet de la classe de base. On fait souvent référence à cette *substitution pure* sous le nom de *principe de substitution*. Dans un sens, c'est la manière idéale de traiter l'héritage. La relation entre la classe de base et la classe dérivée dans ce cas est une relation *est-un*, parce qu'on peut dire « un cercle *est une* forme ». Un test pour l'héritage est de déterminer si la relation est-un entre les deux classes considérées a un sens.

Mais parfois il est nécessaire d'ajouter de nouveaux éléments à l'interface d'un type dérivé, et donc étendre l'interface et créer un nouveau type. Le nouveau type peut toujours être substitué au type de base, mais la substitution n'est plus parfaite parce que les nouvelles fonctions ne sont pas accessibles à partir de la classe parent. On appelle cette relation une relation *est-comme-un* [6]; le nouveau type dispose de l'interface de l'ancien type mais il contient aussi d'autres fonctions, on ne peut donc pas réellement dire que ce soient exactement les mêmes. Prenons le cas d'un système de climatisation. Supposons que notre maison dispose des tuyaux et des systèmes de contrôle pour le refroidissement, autrement dit elle dispose d'une interface qui nous permet de contrôler le refroidissement. Imaginons que le système de climatisation tombe en panne et qu'on le remplace par une pompe à chaleur, qui peut à la fois chauffer et refroidir. La pompe à chaleur *est-comme-un* système de climatisation, mais il peut faire plus de choses. Parce que le système de contrôle n'a été conçu que pour contrôler le refroidissement, il en est restreint à ne communiquer qu'avec la partie refroidissement du nouvel objet. L'interface du nouvel objet a été étendue, mais le système existant ne

connaît rien qui ne soit dans l'interface originale.



Bien sûr, quand on voit cette modélisation, il est clair que la classe de base « Système de refroidissement » n'est pas assez générale, et devrait être renommée en « Système de contrôle de température » afin de pouvoir inclure le chauffage - auquel cas le principe de substitution marcherait. Cependant, le diagramme ci-dessus est un exemple de ce qui peut arriver dans le monde réel.

Quand on considère le principe de substitution, il est tentant de se dire que cette approche (la substitution pure) est la seule manière correcte de modéliser, et de fait *c'est* appréciable si la conception fonctionne ainsi. Mais dans certains cas il est tout aussi clair qu'il faut ajouter de nouvelles fonctions à l'interface d'une classe dérivée. En examinant le problème, les deux cas deviennent relativement évidents.

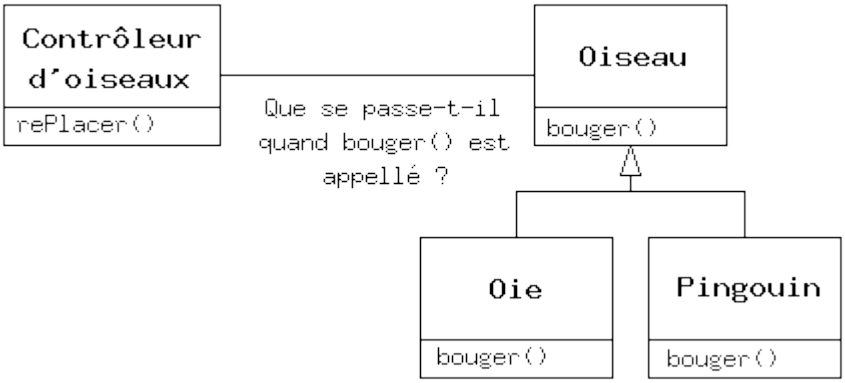
Polymorphisme : des objets interchangeables

Il arrive qu'on veuille traiter un objet non en tant qu'objet du type spécifique qu'il est, mais en tant qu'objet de son type de base. Cela permet d'écrire du code indépendant des types spécifiques. Dans l'exemple de la forme géométrique, les fonctions manipulent des formes génériques sans se soucier de savoir si ce sont des cercles, des carrés, des triangles ou même des formes non encore définies. Toutes les formes peuvent être dessinées, effacées, et déplacées, donc ces fonctions envoient simplement un message à un objet forme, elles ne se soucient pas de la manière dont l'objet traite le message.

Un tel code n'est pas affecté par l'addition de nouveaux types, et ajouter de nouveaux types est la façon la plus commune d'étendre un programme orienté objet pour traiter de nouvelles situations. Par exemple, on peut dériver un nouveau type de forme appelé pentagone sans modifier les fonctions qui traitent des formes génériques. Cette capacité à étendre facilement un programme en dérivant de nouveaux sous-types est important car il améliore considérablement la conception tout en réduisant le coût de maintenance.

Un problème se pose cependant en voulant traiter les types dérivés comme leur type de base générique (les cercles comme des formes géométriques, les vélos comme des véhicules, les cormorans comme des oiseaux, etc...). Si une fonction demande à une forme générique de se dessiner, ou à un véhicule générique de tourner, ou à un oiseau générique de se déplacer, le compilateur ne peut savoir précisément lors de la phase de compilation quelle portion de code sera exécutée. C'est

d'ailleurs le point crucial : quand le message est envoyé, le programmeur ne *veut* pas savoir quelle portion de code sera exécutée ; la fonction dessiner peut être appliquée aussi bien à un cercle qu'à un carré ou un triangle, et l'objet va exécuter le bon code suivant son type spécifique. Si on n'a pas besoin de savoir quelle portion de code est exécutée, alors le code exécuté lorsque on ajoute un nouveau sous-type peut être différent sans exiger de modification dans l'appel de la fonction. Le compilateur ne peut donc précisément savoir quelle partie de code sera exécutée, donc que va-t-il faire ? Par exemple, dans le diagramme suivant, l'objet **Contrôleur d'oiseaux** travaille seulement avec des objets **Oiseaux** génériques, et ne sait pas de quel type ils sont. Cela est pratique du point de vue de **Contrôleur d'oiseaux** car il n'a pas besoin d'écrire du code spécifique pour déterminer le type exact d'**Oiseau** avec lequel il travaille, ou le comportement de cet **Oiseau**. Comment se fait-il donc que, lorsque **bouger()** est appelé tout en ignorant le type spécifique de l'**Oiseau**, on obtienne le bon comportement (une **Oie** court, vole ou nage, et un **Pingouin** court ou nage)



La réponse constitue l'astuce fondamentale de la programmation orientée objet : le compilateur ne peut faire un appel de fonction au sens traditionnel du terme. Un appel de fonction généré par un compilateur non orienté objet crée ce qu'on appelle une *association prédéfinie*, un terme que vous n'avez sans doute jamais entendu auparavant car vous ne pensiez pas qu'on puisse faire autrement. En d'autres termes, le compilateur génère un appel à un nom de fonction spécifique, et l'éditeur de liens résout cet appel à l'adresse absolue du code à exécuter. En POO, le programme ne peut déterminer l'adresse du code avant la phase d'exécution, un autre mécanisme est donc nécessaire quand un message est envoyé à un objet générique.

Pour résoudre ce problème, les langages orientés objet utilisent le concept d'*association tardive*. Quand un objet reçoit un message, le code appelé n'est pas déterminé avant l'exécution. Le compilateur s'assure que la fonction existe et vérifie le type des arguments et de la valeur de retour (un langage omettant ces vérifications est dit *faiblement typé*), mais il ne sait pas exactement quel est le code à exécuter.

Pour créer une association tardive, Java utilise une portion spéciale de code en lieu et place de l'appel absolu. Ce code calcule l'adresse du corps de la fonction, en utilisant des informations stockées dans l'objet (ce mécanisme est couvert plus en détails dans le Chapitre 7). Ainsi, chaque objet peut se comporter différemment suivant le contenu de cette portion spéciale de code. Quand un objet reçoit un message, l'objet sait quoi faire de ce message.

Dans certains langages (en particulier le C++), il faut préciser explicitement qu'on souhaite bénéficier de la flexibilité de l'association tardive pour une fonction. Dans ces langages, les fonctions membres ne sont *pas* liées dynamiquement par défaut. Cela pose des problèmes, donc en Java

l'association dynamique est le défaut et aucun mot clef supplémentaire n'est requis pour bénéficier du polymorphisme.

Reprenons l'exemple de la forme géométrique. Le diagramme de la hiérarchie des classes (toutes basées sur la même interface) se trouve plus haut dans ce chapitre. Pour illustrer le polymorphisme, écrivons un bout de code qui ignore les détails spécifiques du type et parle uniquement à la classe de base. Ce code est *déconnecté* des informations spécifiques au type, donc plus facile à écrire et à comprendre. Et si un nouveau type - un **Hexagone**, par exemple - est ajouté grâce à l'héritage, le code continuera de fonctionner aussi bien pour ce nouveau type de **Forme** qu'il le faisait avec les types existants. Le programme est donc extensible.

Si nous écrivons une méthode en Java (comme vous allez bientôt apprendre à le faire) :

```
void faireQuelqueChose(Forme f) {
  f.effacer();
  // ...
  f.dessiner();
}
```

Cette fonction s'adresse à n'importe quelle **Forme**, elle est donc indépendante du type spécifique de l'objet qu'elle dessine et efface. Si nous utilisons ailleurs dans le programme cette fonction **faireQuelqueChose()** :

```
Cercle c = new Cercle();
Triangle t = new Triangle();
Ligne l = new Ligne();
faireQuelqueChose(c);
faireQuelqueChose(t);
faireQuelqueChose(l);
```

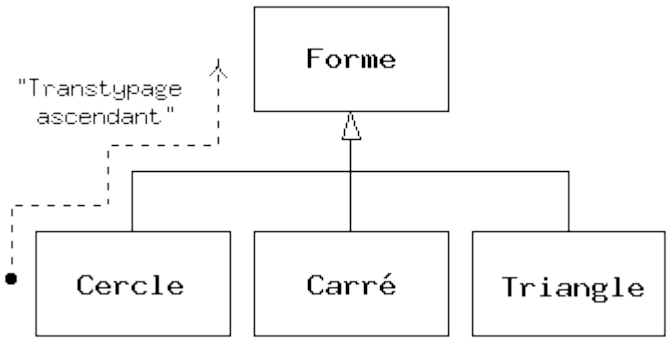
Les appels à **faireQuelqueChose()** fonctionnent correctement, sans se préoccuper du type exact de l'objet.

En fait c'est une manière de faire très élégante. Considérons la ligne :

```
faireQuelqueChose(c);
```

Un **Cercle** est ici passé à une fonction qui attend une **Forme**. Comme un **Cercle** *est-une* **Forme**, il peut être traité comme tel par **faireQuelqueChose()**. C'est à dire qu'un **Cercle** peut accepter tous les messages que **faireQuelqueChose()** pourrait envoyer à une forme. C'est donc une façon parfaitement logique et sûre de faire.

Traiter un type dérivé comme s'il était son type de base est appelé *transtypage ascendant*, *surtypage* ou *généralisation* (upcasting). L'adjectif ascendant vient du fait que dans un diagramme d'héritage typique, le type de base est représenté en haut, les classes dérivées s'y rattachant par le bas. Ainsi, changer un type vers son type de base revient à remonter dans le diagramme d'héritage : transtypage « ascendant ».



Un programme orienté objet contient obligatoirement des transtypages ascendants, car c'est de cette manière que le type spécifique de l'objet peut être délibérément ignoré. Examinons le code de **faireQuelqueChose()** :

```
f.effacer();
 // ...
 f.dessiner();
```

Remarquez qu'il ne dit pas « Si tu es un **Cercle**, fais ceci, si tu es un **Carré**, fais cela, etc... ». Ce genre de code qui vérifie tous les types possibles que peut prendre une **Forme** est confus et il faut le changer à chaque extension de la classe **Forme**. Ici, il suffit de dire : « Tu es une forme géométrique, je sais que tu peux te **dessiner()** et t'**effacer()**, alors fais-le et occupe-toi des détails spécifiques ».

Ce qui est impressionnant dans le code de **faireQuelqueChose()**, c'est que tout fonctionne comme on le souhaite. Appeler **dessiner()** pour un **Cercle** exécute une portion de code différente de celle exécutée lorsqu'on appelle **dessiner()** pour un **Carré** ou une **Ligne**, mais lorsque le message **dessiner()** est envoyé à une **Forme** anonyme, on obtient le comportement idoine basé sur le type réel de la **Forme**. Cela est impressionnant dans la mesure où le compilateur Java ne sait pas à quel type d'objet il a affaire lors de la compilation du code de **faireQuelqueChose()**. On serait en droit de s'attendre à un appel aux versions **dessiner()** et **effacer()** de la classe de base **Forme**, et non celles des classes spécifiques **Cercle**, **Carré** et **Ligne**. Mais quand on envoie un message à un objet, il fera ce qu'il a à faire, même quand la généralisation est impliquée. C'est ce qu'implique le polymorphisme. Le compilateur et le système d'exécution s'occupent des détails, et c'est tout ce que vous avez besoin de savoir, en plus de savoir comment modéliser avec.

Classes de base abstraites et interfaces

Dans une modélisation, il est souvent souhaitable qu'une classe de base ne présente qu'une interface pour ses classes dérivées. C'est à dire qu'on ne souhaite pas qu'il soit possible de créer un objet de cette classe de base, mais seulement pouvoir surtyper jusqu'à elle pour pouvoir utiliser son interface. Cela est possible en rendant cette classe *abstraite* en utilisant le mot clef **abstract**. Le compilateur se plaindra si une tentative est faite de créer un objet d'une classe définie comme **abstract**. C'est un outil utilisé pour forcer une certain conception.

Le mot clef **abstract** est aussi utilisé pour décrire une méthode qui n'a pas encore été implémentée - comme un panneau indiquant « voici une fonction de l'interface dont les types dérivés ont hérité, mais actuellement je n'ai aucune implémentation pour elle ». Une méthode **abstract** peut seulement être créée au sein d'une classe **abstract**. Quand cette classe est dérivée, cette méthode

doit être implémentée, ou la classe dérivée devient **abstract** elle aussi. Créer une méthode **abstract** permet de l'inclure dans une interface sans être obligé de fournir une portion de code éventuellement dépourvue de sens pour cette méthode.

Le mot clef **interface** pousse le concept de classe **abstract** un cran plus loin en évitant toute définition de fonction. Une **interface** est un outil très pratique et très largement répandu, car il fournit une séparation parfaite entre l'interface et l'implémentation. De plus, on peut combiner plusieurs interfaces, alors qu'hériter de multiples classes normales ou abstraites est impossible.

Environnement et durée de vie des objets

Techniquement, les spécificités de la programmation orientée objet se résument au typage abstrait des données, à l'héritage et au polymorphisme, mais d'autres particularités peuvent se révéler aussi importantes. Le reste de cette section traite de ces particularités.

L'une des particularités les plus importantes est la façon dont les objets sont créés et détruits. Où se trouvent les données d'un objet et comment sa durée de vie est-elle contrôlée ? Différentes philosophies existent. En C++, qui prône que l'efficacité est le facteur le plus important, le programmeur a le choix. Pour une vitesse optimum à l'exécution, le stockage et la durée de vie peuvent être déterminé quand le programme est écrit, en plaçant les objets sur la pile (ces variables sont parfois appelées *automatiques* ou *de portée*) ou dans l'espace de stockage statique. La vitesse d'allocation et de libération est dans ce cas prioritaire et leur contrôle peut être vraiment appréciable dans certaines situations. Cependant, cela se fait aux dépends de la flexibilité car il faut connaître la quantité exacte, la durée de vie et le type des objets pendant qu'on écrit le programme. Si le problème à résoudre est plus général, tel que de la modélisation assistée par ordinateur, de la gestion d'entrepôts ou du contrôle de trafic aérien, cela se révèle beaucoup trop restrictif.

La deuxième approche consiste à créer les objets dynamiquement dans un pool de mémoire appelé le segment. Dans cette approche, le nombre d'objets nécessaire n'est pas connu avant l'exécution, de même que leur durée de vie ou leur type exact. Ces paramètres sont déterminés sur le coup, au moment où le programme s'exécute. Si on a besoin d'un nouvel objet, il est simplement créé dans le segment au moment où on en a besoin. Comme le stockage est géré de manière dynamique lors de l'exécution, le temps de traitement requis pour allouer de la place dans le segment est plus important que le temps mis pour stocker sur la pile (stocker sur la pile se résume souvent à une instruction assembleur pour déplacer le pointeur de pile vers le bas, et une autre pour le redéplacer vers le haut). L'approche dynamique fait la supposition généralement justifiée que les objets ont tendance à être compliqués, et que le surcoût de temps dû à la recherche d'une place de stockage et à sa libération n'aura pas d'impact significatif sur la création d'un objet. De plus, la plus grande flexibilité qui en résulte est essentielle pour résoudre le problème modélisé par le programme.

Une autre particularité importante est la durée de vie d'un objet. Avec les langages qui autorisent la création d'objets dans la pile, le compilateur détermine combien de temps l'objet est amené à vivre et peut le détruire automatiquement. Mais si l'objet est créé dans le segment, le compilateur n'a aucune idée de sa durée de vie. Dans un langage comme le C++, il faut déterminer dans le programme quand détruire l'objet, ce qui peut mener à des fuites de mémoire si cela n'est pas fait correctement (et c'est un problème courant en C++). Java propose une fonctionnalité appelée ramasse-miettes (garbage collector) qui découvre automatiquement quand un objet n'est plus utilisé et le détruit. Java propose donc un niveau plus élevé d'assurance contre les fuites de mémoire. Disposer d'un ramasse-miettes est pratique car cela réduit le code à écrire et, plus important, le nombre de

problèmes liés à la gestion de la mémoire (qui ont mené à l'abandon de plus d'un projet C++).

Le reste de cette section s'attarde sur des facteurs additionnels concernant l'environnement et la durée de vie des objets.

Collections et itérateurs

Si le nombre d'objets nécessaires à la résolution d'un problème est inconnu, ou combien de temps on va en avoir besoin, on ne peut pas non plus savoir comment les stocker. Comment déterminer l'espace nécessaire pour créer ces objets ? C'est impossible car cette information n'est connue que lors de l'exécution.

La solution à la plupart des problèmes en conception orientée objet est simple : il suffit de créer un nouveau type d'objet. Le nouveau type d'objets qui résout ce problème particulier contient des références aux autres objets. Bien sûr, un tableau ferait aussi bien l'affaire. Mais il y a plus. Ce nouvel objet, appelé *conteneur* (ou *collection*, mais la bibliothèque Java utilise ce terme dans un autre sens ; nous utiliserons donc le terme « conteneur » dans la suite de ce livre), grandira automatiquement pour accepter tout ce qu'on place dedans. Connaître le nombre d'objets qu'on désire stocker dans un conteneur n'est donc plus nécessaire. Il suffit de créer un objet conteneur et le laisser s'occuper des détails.

Heureusement, les langages orientés objet décents fournissent ces conteneurs. En C++, ils font partie de la bibliothèque standard (STL, Standard Template Library). Le Pascal Objet dispose des conteneurs dans sa Bibliothèque de Composants Visuels (VCL, Visual Component Library). Smalltalk propose un ensemble vraiment complet de conteneurs. Java aussi propose des conteneurs dans sa bibliothèque standard. Dans certaines bibliothèques, un conteneur générique est jugé suffisant pour tous les besoins, et dans d'autres (Java par exemple), la bibliothèque dispose de différents types de conteneurs suivant les besoins : des vecteurs (appelé **ArrayList** en Java) pour un accès pratique à tous les éléments, des listes chaînées pour faciliter l'insertion, par exemple, on peut donc choisir le type particulier qui convient le mieux. Les bibliothèques de conteneurs peuvent aussi inclure les ensembles, les files, les dictionnaires, les arbres, les piles, etc...

Tous les conteneurs disposent de moyens pour y stocker des choses et les récupérer ; ce sont habituellement des fonctions pour ajouter des éléments dans un conteneur et d'autres pour les y retrouver. Mais retrouver des éléments peut être problématique, car une fonction de sélection unique peut se révéler trop restrictive. Comment manipuler ou comparer un ensemble d'éléments dans le conteneur ?

La réponse à cette question prend la forme d'un itérateur, qui est un objet dont le travail est de choisir les éléments d'un conteneur et de les présenter à l'utilisateur de l'itérateur. En tant que classe, il fournit de plus un niveau d'abstraction supplémentaire. Cette abstraction peut être utilisée pour séparer les détails du conteneur du code qui utilise ce conteneur. Le conteneur, via l'itérateur, est perçu comme une séquence. L'itérateur permet de parcourir cette séquence sans se préoccuper de sa structure sous-jacente - qu'il s'agisse d'une **ArrayList** (vecteur), une **LinkedList** (liste chaînée), une **Stack** (pile) ou autre. Cela permet de changer facilement la structure de données sous-jacente sans perturber le code du programme. Java commença (dans les versions 1.0 et 1.1) avec un itérateur standard, appelé **Enumeration**, pour toutes ses classes conteneurs. Java 2 est accompagné d'une bibliothèque de conteneurs beaucoup plus complète qui contient entre autres un itérateur appelé **Iterator** bien plus puissant que l'ancienne **Enumeration**.

Du point de vue du design, tout ce dont on a besoin est une séquence qui peut être manipulée pour résoudre le problème. Si un seul type de séquence satisfaisait tous les besoins, il n'y aurait pas

de raison d'en avoir de types différents. Il y a deux raisons qui font qu'on a besoin d'un choix de conteneurs. Tout d'abord, les conteneurs fournissent différents types d'interfaces et de comportements. Une pile a une interface et un comportement différents de ceux d'une file, qui sont différents de ceux fournis par un ensemble ou une liste. L'un de ces conteneurs peut se révéler plus flexible qu'un autre pour la résolution du problème considéré. Deuxièmement, les conteneurs ne sont pas d'une même efficacité pour les mêmes opérations. Prenons le cas d'une **ArrayList** et d'une **LinkedList**. Les deux sont de simples séquences qui peuvent avoir la même interface et comportement. Mais certaines opérations ont des coûts radicalement différents. Accéder à des éléments au hasard dans une **ArrayList** est une opération qui demande toujours le même temps, quel que soit l'élément auquel on souhaite accéder. Mais dans une **LinkedList**, il est coûteux de se déplacer dans la liste pour rechercher un élément, et cela prend plus de temps pour trouver un élément qui se situe plus loin dans la liste. Par contre, si on souhaite insérer un élément au milieu d'une séquence, c'est bien plus efficace dans une **LinkedList** que dans une **ArrayList**. Ces opérations et d'autres ont des efficacités différentes suivant la structure sous-jacente de la séquence. Dans la phase de conception, on peut débuter avec une **LinkedList** et lorsqu'on se penche sur l'optimisation, changer pour une **ArrayList**. Grâce à l'abstraction fournie par les itérateurs, on peut passer de l'une à l'autre avec un impact minime sur le code.

En définitive, un conteneur n'est qu'un espace de stockage où placer des objets. Si ce conteneur couvre tous nos besoins, son implémentation réelle n'a pas grande importance (un concept de base pour la plupart des objets). Mais il arrive que la différence de coûts entre une **ArrayList** et une **LinkedList** ne soit pas à négliger, suivant l'environnement du problème et d'autres facteurs. On peut n'avoir besoin que d'un seul type de séquence. On peut même imaginer le conteneur « parfait », qui changerait automatiquement son implémentation selon la manière dont on l'utilise.

La hiérarchie de classes unique

L'une des controverses en POO devenue proéminente depuis le C++ demande si toutes les classes doivent être finalement dérivées d'une classe de base unique. En Java (et comme dans pratiquement tous les autres langages OO) la réponse est « oui » et le nom de cette classe de base ultime est tout simplement **Object**. Les bénéfices d'une hiérarchie de classes unique sont multiples.

Tous les objets dans une hiérarchie unique ont une interface commune, ils sont donc tous du même type fondamental. L'alternative (proposée par le C++) est qu'on ne sait pas que tout est du même type fondamental. Du point de vue de la compatibilité ascendante, cela épouse plus le modèle du C et peut se révéler moins restrictif, mais lorsqu'on veut programmer en tout objet il faut reconstruire sa propre hiérarchie de classes pour bénéficier des mêmes avantages fournis par défaut par les autres langages OO. Et dans chaque nouvelle bibliothèque de classes qu'on récupère, une interface différente et incompatible sera utilisée. Cela demande des efforts (et éventuellement l'utilisation de l'héritage multiple) pour intégrer la nouvelle interface dans la conception. Est-ce que la « flexibilité » que le C++ fournit en vaut réellement le coup ? Si on en a besoin - par exemple si on dispose d'un gros investissement en C - alors oui. Mais si on démarre de zéro, d'autres alternatives telles que Java se révèlent beaucoup plus productives.

Tous les objets dans une hiérarchie de classes unique (comme celle que propose Java) sont garantis d'avoir certaines fonctionnalités. Un certain nombre d'opérations élémentaires peuvent être effectuées sur tous les objets du système. Une hiérarchie de classes unique, accompagnée de la création des objets dans le segment, simplifie considérablement le passage d'arguments (l'un des sujets les plus complexes en C++).

Une hiérarchie de classes unique facilite aussi l'implémentation d'un ramasse-miettes (qui est

fourni en standard en Java). Le support nécessaire est implanté dans la classe de base, et le ramasse-miettes peut donc envoyer le message idoine à tout objet du système. Sans une hiérarchie de classe unique et un système permettant de manipuler un objet via une référence, il est difficile d'implémenter un ramasse-miettes.

Comme tout objet dispose en lui d'informations dynamiques, on ne peut se retrouver avec un objet dont on ne peut déterminer le type. Ceci est particulièrement important avec les opérations du niveau système, telles que le traitement des exceptions, et cela permet une plus grande flexibilité dans la programmation.

Bibliothèques de collections et support pour l'utilisation aisée des collections

Parce qu'un conteneur est un outil qu'on utilise fréquemment, il est logique d'avoir une bibliothèque de conteneurs conçus de manière à être réutilisables, afin de pouvoir en prendre un et l'insérer dans le programme. Java fournit une telle bibliothèque, qui devrait satisfaire tous les besoins.

Transtypages descendants vs. patrons génériques

Pour rendre ces conteneurs réutilisables, ils stockent le type universel en Java précédemment mentionné : **Object**. La hiérarchie de classe unique implique que tout est un **Object**, un conteneur stockant des **Object**s peut donc stocker n'importe quoi. Cela rend les conteneurs aisément réutilisables.

Pour utiliser ces conteneurs, il suffit d'y ajouter des références à des objets, et les redemander plus tard. Mais comme le conteneur ne stocke que des **Object**s, quand une référence à un objet est ajoutée dans le conteneur, il subit un transtypage ascendant en **Object**, perdant alors son identité. Quand il est recherché par la suite, on récupère une référence à un **Object**, et non une référence au type qu'on a inséré. Comment le récupérer et retrouver l'interface de l'objet qu'on a stocké dans le conteneur ?

On assiste ici aussi à un transtypage, mais cette fois-ci il ne remonte pas dans la hiérarchie de classe à un type plus général, mais descend dans la hiérarchie jusqu'à un type plus spécifique, c'est un transtypage descendant ou spécialisation, ou soustypage. Avec la généralisation, on sait par exemple qu'un **Cercle** est un type de **Forme**, et que le transtypage est donc sans danger ; mais on ne sait pas qu'un **Object** est aussi un **Cercle** ou une **Forme**, il est donc rarement sur d'appliquer une spécialisation à moins de savoir exactement à quoi on a affaire.

Ce n'est pas trop dangereux cependant, car si une spécialisation est tentée jusqu'à un type incompatible le système d'exécution générera une erreur appelée *exception*, qui sera décrite plus loin. Quand une référence d'objet est rapatriée d'un conteneur, il faut donc un moyen de se rappeler exactement son type afin de pouvoir le spécialiser correctement.

La spécialisation et les contrôles à l'exécution génèrent un surcoût de temps pour le programme, et des efforts supplémentaires de la part du programmeur. Il semblerait plus logique de créer le conteneur de façon à ce qu'il connaisse le type de l'objet stocké, éliminant du coup la spécialisation et la possibilité d'erreur. La solution est fournie par les types paramétrés, qui sont des classes que le compilateur peut personnaliser pour les faire fonctionner avec des types particuliers. Par exemple, avec un conteneur paramétré, le compilateur peut personnaliser ce conteneur de façon à ce qu'il n'accepte que des **Forme**s et ne renvoie que des **Forme**s.

Les types paramétrés sont importants en C++, en particulier parce que le C++ ne dispose pas d'une hiérarchie de classe unique. En C++, le mot clef qui implémente les types paramétrés est

« template ». Java ne propose pas actuellement de types paramétrés car c'est possible de les simuler - bien que difficilement - via la hiérarchie de classes unique. Une solution de types paramétrés basée sur la syntaxe des templates C++ est actuellement en cours de proposition.

Transtypages descendants vs. patrons génériques

Pour rendre ces conteneurs réutilisables, ils stockent le type universel en Java précédemment mentionné : **Object**. La hiérarchie de classe unique implique que tout est un **Object**, un conteneur stockant des **Object**s peut donc stocker n'importe quoi. Cela rend les conteneurs aisément réutilisables.

Pour utiliser ces conteneurs, il suffit d'y ajouter des références à des objets, et les redemander plus tard. Mais comme le conteneur ne stocke que des **Object**s, quand une référence à un objet est ajoutée dans le conteneur, il subit un transtypage ascendant en **Object**, perdant alors son identité. Quand il est recherché par la suite, on récupère une référence à un **Object**, et non une référence au type qu'on a inséré. Comment le récupérer et retrouver l'interface de l'objet qu'on a stocké dans le conteneur ?

On assiste ici aussi à un transtypage, mais cette fois-ci il ne remonte pas dans la hiérarchie de classe à un type plus général, mais descend dans la hiérarchie jusqu'à un type plus spécifique, c'est un transtypage descendant ou spécialisation, ou soustypage. Avec la généralisation, on sait par exemple qu'un **Cercle** est un type de **Forme**, et que le transtypage est donc sans danger ; mais on ne sait pas qu'un **Object** est aussi un **Cercle** ou une **Forme**, il est donc rarement sur d'appliquer une spécialisation à moins de savoir exactement à quoi on a affaire.

Ce n'est pas trop dangereux cependant, car si une spécialisation est tentée jusqu'à un type incompatible le système d'exécution générera une erreur appelée *exception*, qui sera décrite plus loin. Quand une référence d'objet est rapatriée d'un conteneur, il faut donc un moyen de se rappeler exactement son type afin de pouvoir le spécialiser correctement.

La spécialisation et les contrôles à l'exécution génèrent un surcoût de temps pour le programme, et des efforts supplémentaires de la part du programmeur. Il semblerait plus logique de créer le conteneur de façon à ce qu'il connaisse le type de l'objet stocké, éliminant du coup la spécialisation et la possibilité d'erreur. La solution est fournie par les types paramétrés, qui sont des classes que le compilateur peut personnaliser pour les faire fonctionner avec des types particuliers. Par exemple, avec un conteneur paramétré, le compilateur peut personnaliser ce conteneur de façon à ce qu'il n'accepte que des **Forme**s et ne renvoie que des **Forme**s.

Les types paramétrés sont importants en C++, en particulier parce que le C++ ne dispose pas d'une hiérarchie de classe unique. En C++, le mot clef qui implémente les types paramétrés est « template ». Java ne propose pas actuellement de types paramétrés car c'est possible de les simuler - bien que difficilement - via la hiérarchie de classes unique. Une solution de types paramétrés basée sur la syntaxe des templates C++ est actuellement en cours de proposition.

Le dilemme du nettoyage : qui en est responsable ?

Chaque objet requiert des ressources, en particulier de la mémoire. Quand un objet n'est plus utilisé il doit être nettoyé afin de rendre ces ressources pour les réutiliser. Dans la programmation de situations simples, la question de savoir comment un objet est libéré n'est pas trop compliquée : il suffit de créer l'objet, l'utiliser aussi longtemps que désiré, et ensuite le détruire. Il n'est pas rare par contre de se trouver dans des situations beaucoup plus complexes.

Supposons qu'on veuille concevoir un système pour gérer le trafic aérien d'un aéroport (ou pour gérer des caisses dans un entrepôt, ou un système de location de cassettes, ou un chenil pour animaux). Cela semble simple de prime abord : créer un conteneur pour stocker les avions, puis créer un nouvel avion et le placer dans le conteneur pour chaque avion qui entre dans la zone de contrôle du trafic aérien. Pour le nettoyage, il suffit de détruire l'objet avion correspondant lorsqu'un avion quitte la zone.

Mais supposons qu'une autre partie du système s'occupe d'enregistrer des informations à propos des avions, ces données ne requérant pas autant d'attention que la fonction principale de contrôle. Il s'agit peut-être d'enregistrer les plans de vol de tous les petits avions quittant l'aéroport. On dispose donc d'un second conteneur des petits avions, et quand on crée un objet avion on doit aussi le stocker dans le deuxième conteneur si c'est un petit avion. Une tâche de fond s'occupe de traiter les objets de ce conteneur durant les moments d'inactivité du système.

Le problème est maintenant plus compliqué : comment savoir quand détruire les objets ? Quand on en a fini avec un objet, une autre partie du système peut ne pas en avoir terminé avec. Ce genre de problème arrive dans un grand nombre de situations, et dans les systèmes de programmation (comme le C++) où les objets doivent être explicitement détruits cela peut devenir relativement complexe.

Avec Java, le ramasse-miettes est conçu pour s'occuper du problème de la libération de la mémoire (bien que cela n'inclut pas les autres aspects du nettoyage de l'objet). Le ramasse-miettes « sait » quand un objet n'est plus utilisé, et il libère automatiquement la mémoire utilisée par cet objet. Ceci (associé avec le fait que tous les objets sont dérivés de la classe de base fondamentale **Object** et que les objets sont créés dans le segment) rend la programmation Java plus simple que la programmation C++. Il y a beaucoup moins de décisions à prendre et d'obstacles à surmonter.

Ramasse-miettes vs. efficacité et flexibilité

Si cette idée est si bonne, pourquoi le C++ n'intègre-t-il pas ce mécanisme ? Bien sûr car il y a un prix à payer pour cette facilité de programmation, et ce surcoût se traduit par du temps système. Comme on l'a vu, en C++ on peut créer des objets dans la pile et dans ce cas ils sont automatiquement nettoyés (mais dans ce cas on n'a pas la flexibilité de créer autant d'objets que voulu lors de l'exécution). Créer des objets dans la pile est la façon la plus efficace d'allouer de l'espace pour des objets et de libérer cet espace. Créer des objets dans le segment est bien plus coûteux. Hériter de la même classe de base et rendre tous les appels de fonctions polymorphes prélèvent aussi un tribut. Mais le ramasse-miettes est un problème à part car on ne sait pas quand il va démarrer ou combien de temps il va prendre. Cela veut dire qu'il y a une inconsistance dans le temps d'exécution de programmes Java ; on ne peut donc l'utiliser dans certaines situations, comme celles où le temps d'exécution d'un programme est critique (appelés programmes en temps réel, bien que tous les problèmes de programmation en temps réel ne soient pas aussi astreignants).

Les concepteurs du langage C++, en voulant amadouer les programmeurs C, ne voulurent pas ajouter de nouvelles fonctionnalités au langage qui puissent impacter la vitesse ou défavoriser l'utilisation du C++ dans des situations où le C se serait révélé acceptable. Cet objectif a été atteint, mais au prix d'une plus grande complexité lorsqu'on programme en C++. Java est plus simple que le C++, mais la contrepartie en est l'efficacité et quelquefois son champ d'applications. Pour un grand nombre de problèmes de programmation cependant, Java constitue le meilleur choix.

Traitement des exceptions : gérer les erreurs

Depuis les débuts des langages de programmation, le traitement des erreurs s'est révélé l'un des problèmes les plus ardus. Parce qu'il est difficile de concevoir un bon mécanisme de gestion des erreurs, beaucoup de langages ignorent ce problème et le délèguent aux concepteurs de bibliothèques qui fournissent des mécanismes qui fonctionnent dans beaucoup de situations mais peuvent être facilement contournés, généralement en les ignorant. L'une des faiblesses de la plupart des mécanismes d'erreur est qu'ils reposent sur la vigilance du programmeur à suivre des conventions non imposées par le langage. Si le programmeur n'est pas assez vigilant - ce qui est souvent le cas s'il est pressé - ces mécanismes peuvent facilement être oubliés.

Le système des exceptions pour gérer les erreurs se situe au niveau du langage de programmation et parfois même au niveau du système d'exploitation. Une exception est un objet qui est « émis » depuis l'endroit où l'erreur est apparue et peut être intercepté par un gestionnaire d'exception conçu pour gérer ce type particulier d'erreur. C'est comme si la gestion des exceptions était un chemin d'exécution parallèle à suivre quand les choses se gâtent. Et parce qu'elle utilise un chemin d'exécution séparé, elle n'interfère pas avec le code s'exécutant normalement. Cela rend le code plus simple à écrire car on n'a pas à vérifier constamment si des erreurs sont survenues. De plus, une exception émise n'est pas comme une valeur de retour d'une fonction signalant une erreur ou un drapeau positionné par une fonction pour indiquer une erreur - ils peuvent être ignorés. Une exception ne peut pas être ignorée, on a donc l'assurance qu'elle sera traitée quelque part. Enfin, les exceptions permettent de revenir d'une mauvaise situation assez facilement. Plutôt que terminer un programme, il est souvent possible de remettre les choses en place et de restaurer son exécution, ce qui produit des programmes plus robustes.

Le traitement des exceptions de Java se distingue parmi les langages de programmes, car en Java le traitement des exceptions a été intégré depuis le début et on est forcé de l'utiliser. Si le code produit ne gère pas correctement les exceptions, le compilateur générera des messages d'erreur. Cette consistance rend la gestion des erreurs bien plus aisée.

Il est bon de noter que le traitement des exceptions n'est pas une caractéristique orientée objet, bien que dans les langages OO une exception soit normalement représentée par un objet. Le traitement des exceptions existait avant les langages orientés objet.

Multithreading

L'un des concepts fondamentaux dans la programmation des ordinateurs est l'idée de traiter plus d'une tâche à la fois. Beaucoup de problèmes requièrent que le programme soit capable de stopper ce qu'il est en train de faire, traite un autre problème puis retourne à sa tâche principale. Le problème a été abordé de beaucoup de manières différentes. Au début, les programmeurs ayant des connaissances sur le fonctionnement de bas niveau de la machine écrivaient des routines d'interruption de service et l'interruption de la tâche principale était initiée par une interruption matérielle. Bien que cela fonctionne correctement, c'était difficile et non portable, et cela rendait le portage d'un programme sur un nouveau type de machine lent et cher.

Quelquefois les interruptions sont nécessaires pour gérer les tâches critiques, mais il existe une large classe de problèmes dans lesquels on tente juste de partitionner le problème en parties séparées et indépendantes afin que le programme soit plus réactif. Dans un programme, ces parties séparées sont appelés threads et le concept général est appelé *multithreading*. Un exemple classique de multithreading est l'interface utilisateur. En utilisant les threads, un utilisateur peur appuyer sur un bouton et obtenir une réponse plus rapide que s'il devait attendre que le programme termine sa

tâche courante.

Généralement, les threads sont juste une façon d'allouer le temps d'un seul processeur. Mais si le système d'exploitation supporte les multi-processeurs, chaque thread peut être assigné à un processeur différent et ils peuvent réellement s'exécuter en parallèle. L'une des caractéristiques intéressantes du multithreading au niveau du langage est que le programmeur n'a pas besoin de se préoccuper du nombre de processeurs. Le programme est divisé logiquement en threads et si la machine dispose de plusieurs processeurs, le programme tourne plus vite, sans aucun ajustement.

Tout ceci pourrait faire croire que le multithreading est simple. Il y a toutefois un point d'achoppement : les ressources partagées. Un problème se pose si plus d'un thread s'exécutant veulent accéder à la même ressource. Par exemple, deux tâches ne peuvent envoyer simultanément de l'information à une imprimante. Pour résoudre ce problème, les ressources pouvant être partagées, comme l'imprimante, doivent être verrouillées avant d'être utilisées. Un thread verrouille donc une ressource, accomplit ce qu'il a à faire, et ensuite relâche le verrou posé afin que quelqu'un d'autre puisse utiliser cette ressource.

Persistance

Quand un objet est créé, il existe aussi longtemps qu'on en a besoin, mais en aucun cas il continue d'exister après que le programme se termine. Bien que cela semble logique de prime abord, il existe des situations où il serait très pratique si un objet pouvait continuer d'exister et garder son information même quand le programme ne tourne plus. A l'exécution suivante du programme, l'objet serait toujours là et il aurait la même information dont il disposait dans la session précédente. Bien sûr, on peut obtenir ce comportement en écrivant l'information dans un fichier ou une base de données, mais dans l'esprit du tout objet, il serait pratique d'être capable de déclarer un objet persistant et que tous les détails soient pris en charge.

Java fournit un support pour la « persistance légère », qui signifie qu'on peut facilement stocker des objets sur disque et les récupérer plus tard. La raison pour laquelle on parle de « persistance légère » est qu'on est toujours obligé de faire des appels explicites pour le stockage et la récupération. De plus, JavaSpaces (décrit au Chapitre 15) fournit un type de stockage persistant des objets. Un support plus complet de la persistance peut être amené à apparaître dans de futures versions.

Java et l'Internet

Java n'étant qu'un autre langage de programmation, on peut se demander pourquoi il est si important et pourquoi il est présenté comme une étape révolutionnaire de la programmation. La réponse n'est pas immédiate si on se place du point de vue de la programmation classique. Bien que Java soit très pratique pour résoudre des problèmes traditionnels, il est aussi important car il permet de résoudre des problèmes liés au web.

Qu'est-ce que le Web ?

Le Web peut sembler un peu mystérieux au début, avec tout ce vocabulaire de « surf », « page personnelles », etc... Il y a même eu une réaction importante contre cette « Internet-mania », se questionnant sur la valeur économique et les revenus d'un tel engouement. Il peut être utile de revenir en arrière et voir de quoi il retourne, mais auparavant il faut comprendre le modèle client/serveur, un autre aspect de l'informatique relativement confus.

Le concept Client/Serveur

L'idée primitive d'un système client/serveur est qu'on dispose d'un dépôt centralisé de l'-information - souvent dans une base de données - qu'on veut distribuer à un ensemble de personnes ou de machines. L'un des concepts majeurs du client/serveur est que le dépôt d'informations est centralisé afin qu'elles puissent être changées facilement et que ces changements se propagent jusqu'aux consommateurs de ces informations. Pris ensemble, le dépôt d'informations, le programme qui distribue l'information et la (les) machine(s) où résident informations et programme sont appelés le serveur. Le programme qui réside sur la machine distante, communique avec le serveur, récupère l'information, la traite et l'affiche sur la machine distante est appelé le *client*.

Le concept de base du client/serveur n'est donc pas si compliqué. Les problèmes surviennent car un seul serveur doit servir de nombreux clients à la fois. Généralement, un système de gestion de base de données est impliqué afin que le concepteur répartisse les informations dans des tables pour une utilisation optimale. De plus, les systèmes permettent généralement aux utilisateurs d'ajouter de nouvelles informations au serveur. Cela veut dire qu'ils doivent s'assurer que les nouvelles données d'un client ne contredisent pas les nouvelles données d'un autre client, ou que ces nouvelles données ne soient pas perdues pendant leur inscription dans la base de données (cela fait appel aux transactions). Comme les programmes d'application client changent, ils doivent être créés, débogués, et installés sur les machines clientes, ce qui se révèle plus compliqué et onéreux que ce à quoi on pourrait s'attendre. C'est particulièrement problématique dans les environnements hétérogènes comprenant différents types de matériels et de systèmes d'exploitation. Enfin, se pose toujours le problème des performances : on peut se retrouver avec des centaines de clients exécutant des requêtes en même temps, et donc un petit délai multiplié par le nombre de clients peut être particulièrement pénalisant. Pour minimiser les temps de latence, les programmeurs travaillent dur pour réduire la charge de travail des tâches incriminées, parfois jusque sur les machines clientes, mais aussi sur d'autres machines du site serveur, utilisant ce qu'on appelle le *middleware* (le middleware est aussi utilisé pour améliorer la maintenabilité).

L'idée relativement simple de distribuer de l'information aux gens a de si nombreux niveaux de complexité dans son implémentation que le problème peut sembler désespérément insoluble. Et pourtant il est crucial : le client/serveur compte pour environ la moitié des activités de programmation. On le retrouve pour tout ce qui va des transactions de cartes de crédit à la distribution de n'importe quel type de données - économique, scientifique, gouvernementale, il suffit de choisir. Dans le passé, on en est arrivé à des solutions particulières aux problèmes particuliers, obligeant à réinventer une solution à chaque fois. Elles étaient difficiles à créer et à utiliser, et l'utilisateur devait apprendre une nouvelle interface pour chacune. Le problème devait être repris dans son ensemble.

Le Web en tant que serveur géant

Le Web est en fait un système client/serveur géant. C'est encore pire que ça, puisque tous les serveurs et les clients coexistent en même temps sur un seul réseau. Vous n'avez pas besoin de le savoir d'ailleurs, car tout ce dont vous vous souciez est de vous connecter et d'interagir avec un serveur à la fois (même si vous devez parcourir le monde pour trouver le bon serveur).

Initialement, c'était un processus à une étape. On faisait une requête à un serveur qui renvoyait un fichier, que le logiciel (ie, le client) interprétait en le formatant sur la machine locale. Mais les gens en voulurent plus : afficher des pages d'un serveur ne leur suffisait plus. Ils voulaient bénéficier de toutes les fonctionnalités du client/serveur afin que le client puisse renvoyer des

informations au serveur, par exemple pour faire des requêtes précises dans la base de données, ajouter de nouvelles informations au serveur, ou passer des commandes (ce qui nécessitait plus de sécurité que ce que le système primitif offrait). Nous assistons à ces changements dans le développement du Web.

Le browser Web fut un grand bond en avant en avançant le concept qu'une information pouvait être affichée indifféremment sur n'importe quel ordinateur. Cependant, les browsers étaient relativement primitifs et ne remplissaient pas toutes les demandes des utilisateurs. Ils n'étaient pas particulièrement interactifs, et avaient tendance à saturer le serveur et l'Internet car à chaque fois qu'on avait besoin de faire quelque chose qui requérait un traitement il fallait renvoyer les informations au serveur pour que celui-ci les traite. Cela pouvait prendre plusieurs secondes ou minutes pour finalement se rendre compte qu'on avait fait une faute de frappe dans la requête. Comme le browser n'était qu'un afficheur, il ne pouvait pas effectuer le moindre traitement (d'un autre côté, cela était beaucoup plus sûr, car il ne pouvait pas de ce fait exécuter sur la machine locale de programmes contenant des bugs ou des virus).

Pour résoudre ce problème, différentes solutions ont été approchées. En commençant par les standards graphiques qui ont été améliorés pour mettre des animations et de la vidéo dans les browsers. Le reste du problème ne peut être solutionné qu'en incorporant la possibilité d'exécuter des programmes sur le client, dans le browser. C'est ce qu'on appelle la programmation côté client.

La programmation côté client

Le concept initial du Web (serveur-browser) fournissait un contenu interactif, mais l'interactivité était uniquement le fait du serveur. Le serveur produisait des pages statiques pour le browser client, qui ne faisait que les interpréter et les afficher. Le HTML de base contient des mécanismes simples pour récupérer de l'information : des boîtes de texte, des cases à cocher, des listes à options et des listes de choix, ainsi qu'un bouton permettant de réinitialiser le contenu du formulaire ou d'envoyer les données du formulaire au serveur. Cette soumission de données passe par CGI (Common Gateway Interface), un mécanisme fourni par tous les serveurs web. Le texte de la soumission indique à CGI quoi faire avec ces données. L'action la plus commune consiste à exécuter un programme situé sur le serveur dans un répertoire typiquement appelé « cgi-bin » (si vous regardez l'adresse en haut de votre browser quand vous appuyez sur un bouton dans une page web, vous verrez certainement « cgi-bin » quelque part au milieu de tous les caractères). Ces programmes peuvent être écrits dans la plupart des langages. Perl est souvent préféré car il a été conçu pour la manipulation de texte et est interprété, et peut donc être installé sur n'importe quel serveur sans tenir compte du matériel ou du système d'exploitation.

Beaucoup de sites Web populaires sont aujourd'hui bâtis strictement sur CGI, et de fait on peut faire ce qu'on veut avec. Cependant, les sites web bâtis sur des programmes CGI peuvent rapidement devenir ingérables et trop compliqués à maintenir, et se pose de plus le problème du temps de réponse. La réponse d'un programme CGI dépend de la quantité de données à envoyer, ainsi que de la charge du serveur et de l'Internet (de plus, le démarrage d'un programme CGI a tendance à être long). Les concepteurs du Web n'avaient pas prévu que la bande passante serait trop petite pour le type d'applications que les gens développèrent. Par exemple, tout type de graphisme dynamique est impossible à réaliser correctement car un fichier GIF doit être créé et transféré du serveur au client pour chaque version de ce graphe. Et vous avez certainement déjà fait l'expérience de quelque chose d'aussi simple que la validation de données sur un formulaire. Vous appuyez sur le bouton « submit » d'une page, les données sont envoyées sur le serveur, le serveur démarre le programme CGI qui y découvre une erreur, formate une page HTML vous informant de votre erreur et vous la ren-

voie ; vous devez alors revenir en arrière d'une page et réessayer. Non seulement c'est lent et frustrant, mais c'est inélégant.

La solution est la programmation côté client. La plupart des machines qui font tourner des browsers Web sont des ordinateurs puissants capables de beaucoup de choses, et avec l'approche originale du HTML statique, elles ne font qu'attendre que le serveur leur envoie parcimonieusement la page suivante. La programmation côté client implique que le browser Web prenne en charge la partie du travail qu'il est capable de traiter, avec comme résultat pour l'utilisateur une interactivité et une vitesse inégalées.

Les problèmes de la programmation côté client ne sont guère différents des discussions de la programmation en général. Les paramètres sont quasiment les mêmes, mais la plateforme est différente : un browser Web est comme un système d'exploitation limité. Au final, on est toujours obligé de programmer, et ceci explique le nombre de problèmes rencontrés dans la programmation côté client. Le reste de cette section présente les différentes approches de la programmation côté client.

Les plug-ins

L'une des plus importantes avancées en terme de programmation orientée client a été le développement du plug-in. C'est une façon pour le programmeur d'ajouter de nouvelles fonctionnalités au browser en téléchargeant une partie de logiciel qui s'enfiche à l'endroit adéquat dans le browser. Il indique au browser : « à partir de maintenant tu peux réaliser cette nouvelle opération » (on n'a besoin de télécharger le plug-in qu'une seule fois). De nouveaux comportements puissants sont donc ajoutés au browser via les plug-ins, mais écrire un plug-in n'est pas une tâche facile, et ce n'est pas quelque chose qu'on souhaiterait faire juste pour bâtir un site Web particulier. La valeur d'un plug-in pour la programmation côté client est qu'il permet à un programmeur expérimenté de développer un nouveau langage et d'intégrer ce langage au browser sans la permission du distributeur du browser. Ainsi, les plug-ins fournissent une « porte dérobée » qui permet la création de nouveaux langages de programmation côté client (bien que tous les langages ne soient pas implémentés en tant que plug-ins).

Les langages de script

Les plug-ins ont déclenché une explosion de langages de script. Avec un langage de script, du code source est intégré directement dans la page HTML, et le plug-in qui interprète ce langage est automatiquement activé quand le page HTML est affichée. Les langages de script tendent à être relativement aisés à comprendre ; et parce qu'ils sont du texte faisant partie de la page HTML, ils se chargent très rapidement dans la même requête nécessaire pour se procurer la page auprès du serveur. La contrepartie en est que le code est visible pour tout le monde (qui peut donc le voler). Mais généralement on ne fait pas des choses extraordinairement sophistiquées avec les langages de script et ce n'est donc pas trop pénalisant.

Ceci montre que les langages de script utilisés dans les browsers Web sont vraiment spécifiques à certains types de problèmes, principalement la création d'interfaces utilisateurs plus riches et attrayantes. Cependant, un langage de script permet de résoudre 80 pour cent des problèmes rencontrés dans la programmation côté client. La plupart de vos problèmes devraient se trouver parmi ces 80 pour cent, et puisque les langages de script sont plus faciles à mettre en oeuvre et permettent un développement plus rapide, vous devriez d'abord vérifier si un langage de script ne vous suffirait pas avant de vous lancer dans une solution plus complexe telle que la programmation Java ou ActiveX.

Les langages de script les plus répandus parmi les browsers sont JavaScript (qui n'a rien à voir avec Java ; le nom a été choisi uniquement pour bénéficier de l'engouement marketing pour Java du moment), VBScript (qui ressemble à Visual Basic), et Tcl/Tk, qui vient du fameux langage de construction d'interfaces graphiques. Il y en a d'autres bien sûr, et sans doute encore plus en développement.

JavaScript est certainement le plus commun d'entre eux. Il est présent dès l'origine dans Netscape Navigator et Microsoft Internet Explorer (IE). De plus, il y a certainement plus de livres disponibles sur JavaScript que sur les autres langages, et certains outils créent automatiquement des pages utilisant JavaScript. Cependant, si vous parlez couramment Visual Basic ou Tcl/Tk, vous serez certainement plus productif en utilisant ces langages qu'en en apprenant un nouveau (vous aurez déjà largement de quoi faire avec les problèmes soulevés par le Web).

Java

Si un langage de script peut résoudre 80 pour cent des problèmes de la programmation côté client, qu'en est-il des 20 pour cent restants ? La solution la plus populaire aujourd'hui est Java. C'est non seulement un langage de programmation puissant conçu pour être sur, interplateformes et international, mais Java est continuellement étendu pour fournir un langage proposant des caractéristiques et des bibliothèques permettant de gérer de manière élégante des problèmes traditionnellement complexes dans les langages de programmation classiques, tels que le multithreading, les accès aux bases de données, la programmation réseau, l'informatique répartie Java permet la programmation côté client via les *applets*.

Une applet est un mini-programme qui ne tourne que dans un browser Web. L'applet est téléchargée automatiquement en temps que partie d'une page Web (comme, par exemple, une image est automatiquement téléchargée). Quand l'applet est activée elle exécute un programme. Là se trouve la beauté de la chose : cela vous permet de distribuer le logiciel client à partir du serveur au moment où l'utilisateur en a besoin et pas avant. L'utilisateur récupère toujours la dernière version en date du logiciel et sans avoir besoin de le réinstaller. De par la conception même de Java, le programmeur n'a besoin de créer qu'un seul programme, et ce programme tournera automatiquement sur tous les browsers disposant d'un interpréteur interne (ce qui inclut la grande majorité des machines). Puisque Java est un langage de programmation complet, on peut déporter une grande partie des traitements sur le client avant et après avoir envoyé les requêtes au serveur. Par exemple, une requête comportant une erreur dans une date ou utilisant un mauvais paramètre n'a plus besoin d'être envoyée sur Internet pour se voir refusée par le serveur ; un client peut très bien aussi s'occuper de gérer l'affichage d'un nouveau point sur un graphe plutôt que d'attendre que le serveur ne s'en occupe, crée une nouvelle image et l'envoie au client. On gagne ainsi non seulement en vitesse et confort d'utilisation, mais le trafic engendré sur le réseau et la charge résultante sur les serveurs peut être réduite, ce qui est bénéfique pour l'Internet dans son ensemble.

L'un des avantages qu'une applet Java a sur un programme écrit dans un langage de script est qu'il est fourni sous une forme précompilée, et donc le code source n'est pas disponible pour le client. D'un autre côté, une applet Java peut être rétroingéniérée sans trop de problèmes, mais cacher son code n'est pas souvent une priorité absolue. Deux autres facteurs peuvent être importants. Comme vous le verrez plus tard dans ce livre, une applet Java compilée peut comprendre plusieurs modules et nécessiter plusieurs accès au serveur pour être téléchargée (à partir de Java 1.1 cela est minimisé par les archives Java ou fichiers JAR, qui permettent de paqueter tous les modules requis ensemble dans un format compressé pour un téléchargement unique). Un programme scripté sera juste inséré dans le texte de la page Web (et sera généralement plus petit et réduira le nombre d'ac-

cès au serveur). Cela peut être important pour votre site Web. Un autre facteur à prendre en compte est la courbe d'apprentissage. Indépendamment de tout ce que vous avez pu entendre, Java n'est pas un langage trivial et facile à apprendre. Si vous avez l'habitude de programmer en Visual Basic, passer à VBScript sera beaucoup plus rapide et comme cela permet de résoudre la majorité des problèmes typiques du client/serveur, il pourrait être difficile de justifier l'investissement de l'apprentissage de Java. Si vous êtes familier avec un langage de script, vous serez certainement gagnant en vous tournant vers JavaScript ou VBScript avant Java, car ils peuvent certainement combler vos besoins et vous serez plus productif plus tôt.

ActiveX

A un certain degré, le compétiteur de Java est ActiveX de Microsoft, bien qu'il s'appuie sur une approche radicalement différente. ActiveX était au départ une solution disponible uniquement sur Windows, bien qu'il soit actuellement développé par un consortium indépendant pour devenir interplateformes. Le concept d'ActiveX clame : « si votre programme se connecte à son environnement de cette façon, il peut être inséré dans une page Web et exécuté par un browser qui supporte ActiveX » (IE supporte ActiveX en natif et Netscape utilise un plug-in). Ainsi, ActiveX ne vous restreint pas à un langage particulier. Si par exemple vous êtes un programmeur Windows expérimenté utilisant un langage tel que le C++, Visual Basic ou Delphi de Borland, vous pouvez créer des composants ActiveX sans avoir à tout réapprendre. ActiveX fournit aussi un mécanisme pour la réutilisation de code dans les pages Web.

La sécurité

Le téléchargement automatique et l'exécution de programmes sur Internet ressemble au rêve d'un concepteur de virus. ActiveX en particulier pose la question épineuse de la sécurité dans la programmation côté client. Un simple click sur un site Web peut déclencher le téléchargement d'un certain nombre de fichiers en même temps que la page HTML : des images, du code scripté, du code Java compilé, et des composants ActiveX. Certains d'entre eux sont sans importance : les images ne peuvent causer de dommages, et les langages de script sont généralement limités dans les opérations qu'ils sont capables de faire. Java a été conçu pour exécuter ses applets à l'intérieur d'un périmètre de sécurité (appelé bac à sable), qui l'empêche d'aller écrire sur le disque ou d'accéder à la mémoire en dehors de ce périmètre de sécurité.

ActiveX se trouve à l'opposé du spectre. Programmer avec ActiveX est comme programmer sous Windows - on peut faire ce qu'on veut. Donc un composant ActiveX téléchargé en même temps qu'une page Web peut causer bien des dégâts aux fichiers du disque. Bien sûr, les programmes téléchargés et exécutés en dehors du browser présentent les mêmes risques. Les virus téléchargés depuis les BBS ont pendant longtemps été un problème, mais la vitesse obtenue sur Internet accroît encore les difficultés.

La solution semble être les « signatures digitales », où le code est vérifié pour montrer qui est l'auteur. L'idée est basée sur le fait qu'un virus se propage parce que son concepteur peut rester anonyme, et si cet anonymat lui est retiré, l'individu devient responsable de ses actions. Mais si le programme contient un bug fatal bien que non intentionnel cela pose toujours problème.

L'approche de Java est de prévenir ces problèmes via le périmètre de sécurité. L'interpréteur Java du browser Web examine l'applet pendant son chargement pour y détecter les instructions interdites. En particulier, les applets ne peuvent écrire sur le disque ou effacer des fichiers (ce que font la plupart des virus). Les applets sont généralement considérées comme sûres ; et comme ceci

est essentiel pour bénéficier d'un système client/serveur fiable, les bugs découverts dans Java permettant la création de virus sont très rapidement corrigés (il est bon de noter que le browser renforce ces restrictions de sécurité, et que certains d'entre eux permettent de sélectionner différents niveaux de sécurité afin de permettre différents niveaux d'accès au système).

On pourrait s'interroger sur cette restriction draconienne contre les accès en écriture sur le disque local. Par exemple, on pourrait vouloir construire une base de données locale ou sauvegarder des données pour un usage futur en mode déconnecté. La vision initiale obligeait tout le monde à se connecter pour réaliser la moindre opération, mais on s'est vite rendu compte que cela était impraticable. La solution a été trouvée avec les « applets signées » qui utilisent le chiffrement avec une clef publique pour vérifier qu'une applet provient bien de là où elle dit venir. Une applet signée peut toujours endommager vos données et votre ordinateur, mais on en revient à la théorie selon laquelle la perte de l'anonymat rend le créateur de l'applet responsable de ses actes, il ne fera donc rien de préjudiciable. Java fournit un environnement pour les signatures digitales afin de permettre à une applet de sortir du périmètre de sécurité si besoin est.

Les signatures digitales ont toutefois oublié un paramètre important, qui est la vitesse à laquelle les gens bougent sur Internet. Si on télécharge un programme buggué et qu'il accomplisse ses méfaits, combien de temps se passera-t-il avant que les dommages ne soient découverts ? Cela peut se compter en jours ou en semaines. Et alors, comment retrouver le programme qui les a causés ? Et en quoi cela vous aidera à ce point ?

Le Web est la solution la plus générique au problème du client/serveur, il est donc logique de vouloir appliquer la même technologie pour résoudre ceux liés aux client/serveur *à l'intérieur* d'une entreprise. Avec l'approche traditionnelle du client/serveur se pose le problème de l'hétérogénéité du parc de clients, de même que les difficultés pour installer les nouveaux logiciels clients. Ces problèmes sont résolus par les browsers Web et la programmation côté client. Quand la technologie du Web est utilisée dans le système d'information d'une entreprise, on s'y réfère en tant qu'intranet. Les intranets fournissent une plus grande sécurité qu'Internet puisqu'on peut contrôler les accès physiques aux serveurs à l'intérieur de l'entreprise. En terme d'apprentissage, il semblerait qu'une fois que les utilisateurs se sont familiarisés avec le concept d'un browser il leur est plus facile de gérer les différences de conception entre les pages et applets, et le coût d'apprentissage de nouveaux systèmes semble se réduire.

Le problème de la sécurité nous mène à l'une des divisions qui semble se former automatiquement dans le monde de la programmation côté client. Si un programme est distribué sur Internet, on ne sait pas sur quelle plateforme il va tourner et il faut être particulièrement vigilant pour ne pas diffuser du code buggué. Il faut une solution multiplateforme et sûre, comme un langage de script ou Java.

Dans le cas d'un intranet, les contraintes peuvent être complètement différentes. Le temps passé à installer les nouvelles versions est la raison principale qui pousse à passer aux browsers, car les mises à jours sont invisibles et automatiques. Il n'est pas rare que tous les clients soient des plateformes Wintel (Intel / Windows). Dans un intranet, on est responsable de la qualité du code produit et on peut corriger les bugs quand ils sont découverts. De plus, on dispose du code utilisé dans une approche plus traditionnelle du client/serveur où il fallait installer physiquement le client à chaque mise à jour du logiciel. Dans le cas d'un tel intranet, l'approche la plus raisonnable consiste à choisir la solution qui permettra de réutiliser le code existant plutôt que de repartir de zéro dans un nouveau langage.

Quand on est confronté au problème de la programmation côté client, la meilleure chose à faire est une analyse coûts-bénéfices. Il faut prendre en compte les contraintes du problème, et la so-

lution qui serait la plus rapide à mettre en oeuvre. En effet, comme la programmation côté client reste de la programmation, c'est toujours une bonne idée de choisir l'approche permettant un développement rapide.

La programmation côté serveur

Toute cette discussion a passé sous silence la programmation côté serveur. Que se passe-t-il lorsqu'un serveur reçoit une requête ? La plupart du temps cette requête est simplement « envoie-moi ce fichier ». Le browser interprète alors ce fichier d'une manière particulière : comme une page HTML, une image graphique, une applet Java, un programme script, etc... Une requête plus compliquée à un serveur implique généralement une transaction vers une base de données. Un scénario classique consiste en une requête complexe de recherche d'enregistrements dans une base de données que le serveur formate dans une page HTML et renvoie comme résultat (bien sûr, si le client est « intelligent » via Java ou un langage de script, la phase de formatage peut être réalisée par le client et le serveur n'aura qu'a envoyer les données brutes, ce qui allégera la charge et le trafic engendré). Ou alors un client peut vouloir s'enregistrer dans une base de données quand il joint un groupe ou envoie une commande, ce qui implique des changements dans la base de données. Ces requêtes doivent être traitées par du code s'exécutant sur le serveur, c'est ce qu'on appelle la programmation côté serveur. Traditionnellement, la programmation côté serveur s'est appuyée sur Perl et les scripts CGI, mais des systèmes plus sophistiqués sont en train de voir le jour. Cela inclut des serveurs Web écrits en Java qui permettent de réaliser toute la programmation côté serveur en Java en écrivant ce qu'on appelle des servlets. Les servlets et leur progéniture, les JSPs, sont les deux raisons principales qui poussent les entreprises qui développent un site Web à se tourner vers Java, en particulier parce qu'ils éliminent les problèmes liés aux différences de capacité des browsers.

Une scène séparée : les applications

La plupart de la publicité faite autour de Java se référait aux applets. Mais Java est aussi un langage de programmation à vocation plus générale qui peut résoudre n'importe quel type de problème - du moins en théorie. Et comme précisé plus haut, il peut y avoir des moyens plus efficaces de résoudre la plupart des problèmes client/serveur. Quand on quitte la scène des applets (et les restrictions qui y sont liées telles que les accès en écriture sur le disque), on entre dans le monde des applications qui s'exécutent sans le soutien d'un browser, comme n'importe quel programme. Les atouts de Java sont là encore non seulement sa portabilité, mais aussi sa facilité de programmation. Comme vous allez le voir tout au long de ce livre, Java possède beaucoup de fonctionnalités qui permettent de créer des programmes robustes dans un laps de temps plus court qu'avec d'autres langages de programmation.

Mais il faut bien être conscient qu'il s'agit d'un compromis. Le prix à payer pour ces améliorations est une vitesse d'exécution réduite (bien que des efforts significatifs soient mis en oeuvre dans ce domaine - JDK 1.3, en particulier, introduit le concept d'amélioration de performances « hotspot »). Comme tout langage de programmation, Java a des limitations internes qui peuvent le rendre inadéquat pour résoudre certains types de problèmes. Java évolue rapidement cependant, et à chaque nouvelle version il permet d'adresser un spectre de plus en plus large de problèmes.

Analyse et conception

Le paradigme de la POO constitue une approche nouvelle et différente de la programmation. Beaucoup de personnes rencontrent des difficultés pour appréhender leur premier projet orienté ob-

jet. Une fois compris que tout est supposé être un objet, et au fur et à mesure qu'on se met à penser dans un style plus orienté objet, on commence à créer de « bonnes » conceptions qui s'appuient sur tous les avantages que la POO offre.

Une *méthode* (ou méthodologie) est un ensemble de processus et d'heuristiques utilisés pour réduire la complexité d'un problème. Beaucoup de méthodes orientées objet ont été formulées depuis l'apparition de la POO. Cette section vous donne un aperçu de ce que vous essayez d'accomplir en utilisant une méthode.

Une méthodologie s'appuie sur un certain nombre d'expériences, il est donc important de comprendre quel problème la méthode tente de résoudre avant d'en adopter une. Ceci est particulièrement vrai avec Java, qui a été conçu pour réduire la complexité (comparé au C) dans l'expression d'un programme. Cette philosophie supprime le besoin de méthodologies toujours plus complexes. Des méthodologies simples peuvent se révéler tout à fait suffisantes avec Java pour une classe de problèmes plus large que ce qu'elles ne pourraient traiter avec des langages procéduraux.

Il est important de réaliser que le terme «> méthodologie » est trompeur et promet trop de choses. Tout ce qui est mis en oeuvre quand on conçoit et réalise un programme est une méthode. Ca peut être une méthode personnelle, et on peut ne pas en être conscient, mais c'est une démarche qu'on suit au fur et à mesure de l'avancement du projet. Si cette méthode est efficace, elle ne nécessitera sans doute que quelques petites adaptations pour fonctionner avec Java. Si vous n'êtes pas satisfait de votre productivité ou du résultat obtenu, vous serez peut-être tenté d'adopter une méthode plus formelle, ou d'en composer une à partir de plusieurs méthodes formelles.

Au fur et à mesure que le projet avance, le plus important est de ne pas se perdre, ce qui est malheureusement très facile. La plupart des méthodes d'analyse et de conception sont conçues pour résoudre même les problèmes les plus gros. Il faut donc bien être conscient que la plupart des projets ne rentrant pas dans cette catégorie, on peut arriver à une bonne analyse et conception avec juste une petite partie de ce qu'une méthode recommande [8]. Une méthode de conception, même limitée, met sur la voie bien mieux que si on commence à coder directement.

Il est aussi facile de rester coincé et tomber dans la « paralysie analytique » où on se dit qu'on ne peut passer à la phase suivante car on n'a pas traqué le moindre petit détail de la phase courante. Il faut bien se dire que quelle que soit la profondeur de l'analyse, certains aspects d'un problème ne se révéleront qu'en phase de conception, et d'autres en phase de réalisation, voire même pas avant que le programme ne soit achevé et exécuté. A cause de ceci, il est crucial d'avancer relativement rapidement dans l'analyse et la conception, et d'implémenter un test du système proposé.

Il est bon de développer un peu ce point. A cause des déboires rencontrés avec les langages procéduraux, il est louable qu'une équipe veuille avancer avec précautions et comprendre tous les détails avant de passer à la conception et l'implémentation. Il est certain que lors de la création d'une base de données, il est capital de comprendre à fond les besoins du client. Mais la conception d'une base de données fait partie d'une classe de problèmes bien définie et bien comprise ; dans ce genre de programmes, la structure de la base de données *est* le problème à résoudre. Les problèmes traités dans ce chapitre font partie de la classe de problèmes « joker » (invention personnelle), dans laquelle la solution n'est pas une simple reformulation d'une solution déjà éprouvée de nombreuses fois, mais implique un ou plusieurs « facteurs joker » - des éléments pour lesquels il n'existe aucune solution préétablie connue, et qui nécessitent de pousser les recherches [9]. Tenter d'analyser à fond un problème joker avant de passer à la conception et l'implémentation mène à la paralysie analytique parce qu'on ne dispose pas d'assez d'informations pour résoudre ce type de problèmes durant la phase d'analyse. Résoudre ce genre de problèmes requiert de répéter le cycle complet, et cela demande de prendre certains risques (ce qui est sensé, car on essaie de faire quelque chose de nouveau

et les revenus potentiels en sont plus élevés). On pourrait croire que le risque est augmenté par cette ruée vers une première implémentation, mais elle peut réduire le risque dans un projet joker car on peut tout de suite se rendre compte si telle approche du problème est viable ou non. Le développement d'un produit s'apparente à de la gestion de risque.

Souvent cela se traduit par « construire un prototype qu'il va falloir jeter ». Avec la POO, on peut encore avoir à en jeter *une partie*, mais comme le code est encapsulé dans des classes, on aura inévitablement produit durant la première passe quelques classes qui valent la peine d'être conservées et développé des idées sur la conception du système. Ainsi, une première passe rapide sur un problème non seulement fournit des informations critiques pour la prochaine passe d'analyse, de conception et d'implémentation, mais elle produit aussi une base du code.

Ceci dit, si on cherche une méthode qui contient de nombreux détails et suggère de nombreuses étapes et documents, il est toujours difficile de savoir où s'arrêter. Il faut garder à l'esprit ce qu'on essaye de découvrir :

1. Quels sont les objets ? (Comment partitionner le projet en ses composants élémentaires ?)
2. Quelles en sont leur interface ? (Quels sont les messages qu'on a besoin d'envoyer à chaque objet ?)

Si on arrive à trouver quels sont les objets et leur interface, alors on peut commencer à coder. On pourra avoir besoin d'autres descriptions et documents, mais on ne peut pas faire avec moins que ça.

Le développement peut être décomposé en cinq phases, et une Phase 0 qui est juste l'engagement initial à respecter une structure de base.

Phase 0 : Faire un plan

Il faut d'abord décider quelles étapes on va suivre dans le développement. Cela semble simple (en fait, *tout* semble simple), et malgré tout les gens ne prennent cette décision qu'après avoir commencé à coder. Si le plan se résume à « retroussons nos manches et codons », alors ça ne pose pas de problèmes (quelquefois c'est une approche valable quand on a affaire à un problème bien connu). Mais il faut néanmoins accepter que c'est le plan.

On peut aussi décider dans cette phase qu'une structure additionnelle est nécessaire. Certains programmeurs aiment travailler en « mode vacances », sans structure imposée sur le processus de développement de leur travail : « Ce sera fait lorsque ce sera fait ». Cela peut être séduisant un moment, mais disposer de quelques jalons aide à se concentrer et focalise les efforts sur ces jalons au lieu d'être obnubilé par le but unique de « finir le projet ». De plus, cela divise le projet en parties plus petites, ce qui le rend moins redoutable (sans compter que les jalons offrent des opportunités de fête).

Quand j'ai commencé à étudier la structure des histoires (afin de pouvoir un jour écrire un roman), j'étais réticent au début à l'idée de structure, trouvant que j'écrivais mieux quand je laissais juste la plume courir sur le papier. Mais j'ai réalisé plus tard que quand j'écris à propos des ordinateurs, la structure est suffisamment claire pour moi pour que je n'ai pas besoin de trop y réfléchir. Mais je structure tout de même mon travail, bien que ce soit inconsciemment dans ma tête. Même si on pense que le plan est juste de commencer à coder, on passe tout de même par les phases successives en se posant certaines questions et en y répondant.

L'exposé de la mission

Tout système qu'on construit, quelle que soit sa complexité, a un but, un besoin fondamental qu'il satisfait. Si on peut voir au delà de l'interface utilisateur, des détails spécifiques au matériel - ou au système -, des algorithmes de codage et des problèmes d'efficacité, on arrive finalement au coeur du problème - simple et nu. Comme le soi-disant *concept fondamental* d'un film hollywoodien, on peut le décrire en une ou deux phrases. Cette description pure est le point de départ.

Le concept fondamental est assez important car il donne le ton du projet ; c'est l'exposé de la mission. Ce ne sera pas nécessairement le premier jet qui sera le bon (on peut être dans une phase ultérieure du projet avant qu'il ne soit complètement clair), mais il faut continuer d'essayer jusqu'à ce que ça sonne bien. Par exemple, dans un système de contrôle de trafic aérien, on peut commencer avec un concept fondamental basé sur le système qu'on construit : « Le programme tour de contrôle garde la trace d'un avion ». Mais cela n'est plus valable quand le système se réduit à un petit aérodrome, avec un seul contrôleur, ou même aucun. Un modèle plus utile ne décrira pas tant la solution qu'on crée que le problème : « Des avions arrivent, déchargent, partent en révision, rechargent et repartent ».

Phase 1 : Que construit-on ?

Dans la génération précédente de conception de programmes (*conception procédurale*), cela s'appelait « l'*analyse des besoins* et les *spécifications du système* ». C'étaient des endroits où on se perdait facilement, avec des documents au nom intimidant qui pouvaient occulter le projet. Leurs intentions étaient bonnes, pourtant. L'analyse des besoins consiste à « faire une liste des indicateurs qu'on utilisera pour savoir quand le travail sera terminé et le client satisfait ». Les spécifications du système consistent en « une description de ce que le programme fera (sans ce préoccuper du *comment*) pour satisfaire les besoins ». L'analyse des besoins est un contrat entre le développeur et le client (même si le client travaille dans la même entreprise, ou se trouve être un objet ou un autre système). Les spécifications du système sont une exploration générale du problème, et en un sens permettent de savoir s'il peut être résolu et en combien de temps. Comme ils requièrent des consensus entre les intervenants sur le projet (et parce qu'ils changent au cours du temps), il vaut mieux les garder aussi bruts que possible - idéalement en tant que listes et diagrammes - pour ne pas perdre de temps. Il peut y avoir d'autres contraintes qui demandent de produire de gros documents, mais en gardant les documents initiaux petits et concis, cela permet de les créer en quelques sessions de brainstorming avec un leader qui affine la description dynamiquement. Cela permet d'impliquer tous les acteurs du projet, et encourage la participation de toute l'équipe. Plus important encore, cela permet de lancer un projet dans l'enthousiasme.

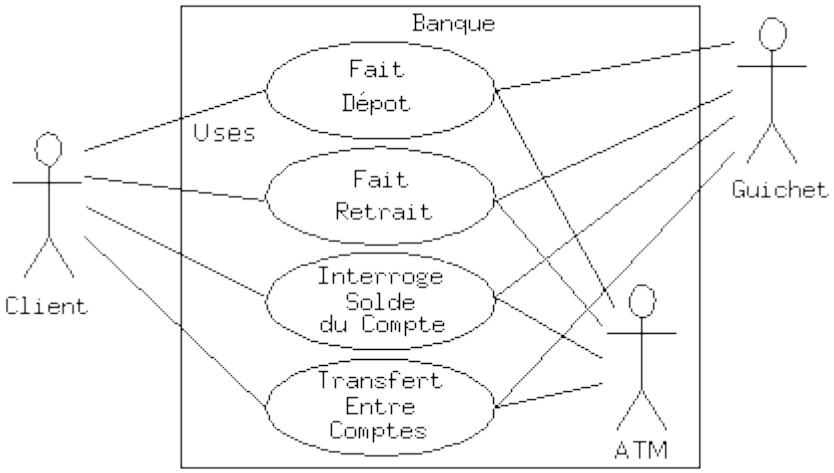
Il est nécessaire de rester concentré sur ce qu'on essaye d'accomplir dans cette phase : déterminer ce que le système est supposé faire. L'outil le plus utile pour cela est une collection de ce qu'on appelle « cas d'utilisation ». Les cas d'utilisation identifient les caractéristiques clefs du système qui vont révéler certaines des classes fondamentales qu'on utilisera. Ce sont essentiellement des réponses descriptives à des questions comme[10] :

- « Qui utilisera le système ? »
- « Que peuvent faire ces personnes avec le système ? »
- « Comment *tel* acteur fait-il *cela* avec le système ? »
- « Comment cela pourrait-il fonctionner si quelqu'un d'autre faisait cela, ou si le même acteur avait un objectif différent ? » (pour trouver les variations)
- « Quels problèmes peuvent apparaître quand on fait cela avec le système ? » (pour

trouver les exceptions)

Si on conçoit un guichet automatique, par exemple, le cas d'utilisation pour un aspect particulier des fonctionnalités du système est capable de décrire ce que le guichet fait dans chaque situation possible. Chacune de ces situations est appelée un *scénario*, et un cas d'utilisation peut être considéré comme une collection de scénarios. On peut penser à un scénario comme à une question qui commence par « Qu'est-ce que le système fait si... ? ». Par exemple, « Qu'est que le guichet fait si un client vient de déposer un chèque dans les dernières 24 heures, et qu'il n'y a pas assez dans le compte sans que le chèque soit encaissé pour fournir le retrait demandé ? ».

Les diagrammes de cas d'utilisations sont voulus simples pour ne pas se perdre prématurément dans les détails de l'implémentation du système :



Chaque bonhomme représente un « acteur », typiquement une personne ou une autre sorte d'agent (cela peut même être un autre système informatique, comme c'est le cas avec « ATM »). La boîte représente les limites du système. Les ellipses représentent les cas d'utilisation, qui sont les descriptions des actions qui peuvent être réalisées avec le système. Les lignes entre les acteurs et les cas d'utilisation représentent les interactions.

Tant que le système est perçu ainsi par l'utilisateur, son implémentation n'est pas importante.

Un cas d'utilisation n'a pas besoin d'être complexe, même si le système sous-jacent l'est. Il est seulement destiné à montrer le système tel qu'il apparaît à l'utilisateur. Par exemple :

Les cas d'utilisation produisent les spécifications des besoins en déterminant toutes les interactions que l'utilisateur peut avoir avec le système. Il faut trouver un ensemble complet de cas d'utilisations du système, et cela terminé on se retrouve avec le coeur de ce que le système est censé faire. La beauté des cas d'utilisation est qu'ils ramènent toujours aux points essentiels et empêchent

de se disperser dans des discussions non essentielles à la réalisation du travail à faire. Autrement dit, si on dispose d'un ensemble complet de cas d'utilisation, on peut décrire le système et passer à la phase suivante. Tout ne sera pas parfaitement clair dès le premier jet, mais ça ne fait rien. Tout se décantera avec le temps, et si on cherche à obtenir des spécifications du système parfaites à ce point on se retrouvera coincé.

Si on est bloqué, on peut lancer cette phase en utilisant un outil d'approximation grossier : décrire le système en quelques paragraphes et chercher les noms et les verbes. Les noms suggèrent les acteurs, le contexte des cas d'utilisation ou les objets manipulés dans les cas d'utilisation. Les verbes suggèrent les interactions entre les acteurs et les cas d'utilisation, et spécifient les étapes à l'intérieur des cas d'utilisation. On verra aussi que les noms et les verbes produisent des objets et des messages durant la phase de design (on peut noter que les cas d'utilisation décrivent les interactions entre les sous-systèmes, donc la technique « des noms et des verbes » ne peut être utilisée qu'en tant qu'outil de brainstorming car il ne fournit pas les cas d'utilisation) [11].

Bien que cela tienne plus de l'art obscur, à ce point un calendrier de base est important. On dispose maintenant d'une vue d'ensemble de ce qu'on construit, on peut donc se faire une idée du temps nécessaire à sa réalisation. Un grand nombre de facteurs entre en jeu ici. Si on estime un calendrier trop important, l'entreprise peut décider d'abandonner le projet (et utiliser leurs ressources sur quelque chose de plus raisonnable - ce qui est une *bonne* chose). Ou un directeur peut avoir déjà décidé du temps que le projet devrait prendre et voudra influencer les estimations. Mais il vaut mieux proposer un calendrier honnête et prendre les décisions importantes au début. Beaucoup de techniques pour obtenir des calendriers précis ont été proposées (de même que pour prédire l'évolution de la bourse), mais la meilleure approche est probablement de se baser sur son expérience et son intuition. Proposer une estimation du temps nécessaire pour réaliser le système, puis doubler cette estimation et ajouter 10 pour cent. L'estimation initiale est probablement correcte, on *peut* obtenir un système fonctionnel avec ce temps. Le doublement transforme le délai en quelque chose de décent, et les 10 pour cent permettront de poser le vernis final et de traiter les détails [12]. Peu importe comment on l'explique et les gémissements obtenus quand on révèle un tel planning, il semble juste que ça fonctionne de cette façon.

Phase 2 : Comment allons-nous le construire ?

Dans cette phase on doit fournir une conception qui décrive ce à quoi les classes ressemblent et comment elles interagissent. Un bon outil pour déterminer les classes et les interactions est la méthode des cartes *Classes-Responsabilités-Collaboration* (CRC). L'un des avantages de cet outil est sa simplicité : on prend des cartes vierges et on écrit dessus au fur et à mesure. Chaque carte représente une classe, et sur la carte on écrit :

1. Le nom de la classe. Il est important que le nom de cette classe reflète l'essence de ce que la classe fait, afin que sa compréhension soit immédiate.
2. Les « responsabilités » de la classe : ce qu'elle doit faire. Typiquement, cela peut être résumé par le nom des fonctions membres (puisque ces noms doivent être explicites dans une bonne conception), mais cela n'empêche pas de compléter par d'autres notes. Pour s'aider, on peut se placer du point de vue d'un programmeur fainéant : quels objets voudrait-on voir apparaître pour résoudre le problème ?
3. Les « collaborations » de la classe : avec quelles classes interagit-elle ? « Interagir » est intentionnellement évasif, il peut se référer à une agrégation ou indiquer qu'un autre objet existant va travailler pour le compte d'un objet de la classe. Les collaborations doivent aussi prendre en compte l'audience de cette classe. Par exemple, si on crée une classe **Pétard**, qui

va l'observer, un **Chimiste** ou un **Spectateur** ? Le premier voudra connaître la composition chimique, tandis que le deuxième sera préoccupé par les couleurs et le bruit produits quand il explose.

On pourrait se dire que les cartes devraient être plus grandes à cause de toutes les informations qu'on aimerait mettre dessus, mais il vaut mieux les garder les plus petites possibles, non seulement pour concevoir de petites classes, mais aussi pour éviter de plonger trop tôt dans les détails. Si on ne peut pas mettre toutes les informations nécessaires à propos d'une classe sur une petite carte, la classe est trop complexe (soit le niveau de détails est trop élevé, soit il faut créer plus d'une classe). La classe idéale doit être comprise en un coup d'oeil. L'objectif des cartes CRC est de fournir un premier jet de la conception afin de saisir le plan général pour pouvoir ensuite affiner cette conception.

L'un des avantages des cartes CRC réside dans la communication. Il vaut mieux les réaliser en groupe, sans ordinateur. Chacun prend le rôle d'une ou plusieurs classes (qui au début n'ont pas de nom ni d'information associée). Il suffit alors de dérouler une simulation impliquant un scénario à la fois, et décider quels messages sont envoyés aux différents objets pour satisfaire chaque scénario. Au fur et à mesure du processus, on découvre quelles sont les classes nécessaires, leurs responsabilités et collaborations, et on peut remplir les cartes. Quand tous les scénarios ont été couverts, on devrait disposer d'une bonne approximation de la conception.

Avant d'utiliser les cartes CRC, la meilleure conception initiale que j'ai fourni sur un projet fut obtenue en dessinant des objets sur un tableau devant une équipe - qui n'avait jamais participé à un projet de POO auparavant. Nous avons discuté de la communication entre ces objets, effacé et remplacé certains d'entre eux par d'autres objets. De fait, je recréais la méthode des cartes CRC au tableau. L'équipe (qui connaissait ce que le projet était censé faire) a effectivement créé la conception ; et de ce fait ils la contrôlaient. Je me contentais de guider le processus en posant les bonnes questions, proposant quelques hypothèses et utilisant les réponses de l'équipe pour modifier ces hypothèses. La beauté de la chose fut que l'équipe a appris à bâtir une conception orientée objet non en potassant des exemples abstraits, mais en travaillant sur la conception qui les intéressait au moment présent : celle de leur projet.

Une fois qu'on dispose d'un ensemble de cartes CRC, on peut vouloir une description plus formelle de la conception en utilisant UML [13]. L'utilisation d'UML n'est pas une obligation, mais cela peut être utile, surtout si on veut afficher au mur un diagramme auquel tout le monde puisse se référer, ce qui est une bonne idée. Une alternative à UML est une description textuelle des objets et de leur interface, ou suivant le langage de programmation, le code lui-même [14].

UML fournit aussi une notation pour décrire le modèle dynamique du système. Cela est pratique dans les cas où les états de transition d'un système ou d'un sous-système sont suffisamment importants pour nécessiter leurs propres diagrammes (dans un système de contrôle par exemple). On peut aussi décrire les structures de données, pour les systèmes ou sous-systèmes dans lesquels les données sont le facteur dominant (comme une base de données).

On sait que la Phase 2 est terminée quand on dispose de la description des objets et de leur interface. Ou du moins de la majorité d'entre eux - il y en a toujours quelques-uns qu'on ne découvre qu'en Phase 3 , mais cela ne fait rien. La préoccupation principale est de découvrir tous les objets. Il est plus agréable de les découvrir le plus tôt possible, mais la POO est assez souple pour pouvoir s'adapter si on en découvre de nouveaux par la suite. En fait, la conception d'un objet se fait en cinq étapes.

Les cinq étapes de la conception d'un objet

La conception d'un objet n'est pas limitée à la phase de codage du programme. En fait, la conception d'un objet passe par une suite d'étapes. Garder cela à l'esprit permet d'éviter de prétendre à la perfection immédiate. On réalise que la compréhension de ce que fait un objet et de ce à quoi il doit ressembler se fait progressivement. Ceci s'applique d'ailleurs aussi à la conception de nombreux types de programmes ; le modèle d'un type de programme n'émerge qu'après s'être confronté encore et encore au problème (se référer au livre *Thinking in Patterns with Java*, téléchargeable sur *www.BruceEckel.com*). Les objets aussi ne se révèlent à la compréhension qu'après un long processus.

1. Découverte de l'objet. Cette étape se situe durant l'analyse initiale du programme. Les objets peuvent être découvert en cherchant les facteurs extérieurs et les frontières, la duplication d'éléments dans le système, et les plus petites unités conceptuelles. Certains objets sont évidents si on dispose d'un ensemble de bibliothèques de classes. La ressemblance entre les classes peut suggérer des classes de base et l'héritage peut en être déduit immédiatement, ou plus tard dans la phase de conception.

2. Assemblage des objets. Lors de la construction d'un objet, on peut découvrir le besoin de nouveaux membres qui n'était pas apparu durant l'étape de découverte. Les besoins internes d'un objet peuvent requérir d'autres classes pour les supporter.

3. Construction du système. Une fois de plus, un objet peut révéler des besoins supplémentaires durant cette étape. Au fur et à mesure de l'avancement du projet, les objets évoluent. Les besoins de la communication et de l'interconnexion avec les autres objets du système peuvent changer les besoins des classes ou demander de nouvelles classes. Par exemple, on peut découvrir le besoin de classes d'utilitaires, telles que des listes chaînées, qui contiennent peu ou pas d'information et sont juste là pour aider les autres classes.

4. Extension du système. Si on ajoute de nouvelles fonctionnalités au système, on peut se rendre compte que sa conception ne facilite pas l'extension du système. Avec cette nouvelle information, on peut restructurer certaines parties du système, éventuellement en ajoutant de nouvelles classes ou de nouvelles hiérarchies de classes.

5. Réutilisation des objets. Ceci est le test final pour une classe. Si quelqu'un tente de réutiliser une classe dans une situation entièrement différente, il y découvrira certainement des imperfections. La modification de la classe pour s'adapter à de nouveaux programmes va en révéler les principes généraux, jusqu'à l'obtention d'un type vraiment réutilisable. Cependant, il ne faut pas s'attendre à ce que tous les objets d'un système soient réutilisables - il est tout à fait légitime que la majorité des objets soient spécifiques au système. Les classes réutilisables sont moins fréquentes, et doivent traiter de problèmes plus génériques pour être réutilisables.

Indications quant au développement des objets

Ces étapes suggèrent quelques règles de base concernant le développement des classes :

1. Quand un problème spécifique génère une classe, la laisser grandir et mûrir durant la résolution d'autres problèmes.

2. Se rappeler que la conception du système consiste principalement à découvrir les classes dont on a besoin (et leurs interfaces). Si on dispose déjà de ces classes, le projet ne devrait pas être compliqué.

3. Ne pas vouloir tout savoir dès le début ; compléter ses connaissances au fur et à mesure de l'avancement du projet. La connaissance viendra de toutes façons tôt ou tard.

4. Commencer à programmer ; obtenir un prototype qui marche afin de pouvoir approuver la conception ou au contraire la dénoncer. Ne pas avoir peur de se retrouver avec du code-spaghetti à la procédurale - les classes partitionnent le problème et aident à contrôler l'anarchie. Les mauvaises classes n'affectent pas les classes bien conçues.

5. Toujours rester le plus simple possible. De petits objets propres avec une utilité apparente sont toujours mieux conçus que ceux disposant de grosses interfaces compliquées. Quand une décision doit être prise, utiliser l'approche du rasoir d'Occam : choisir la solution la plus simple, car les classes simples sont presque toujours les meilleures. Commencer petit et simple, et étendre l'interface de la classe quand on la comprend mieux. Il est toujours plus difficile d'enlever des éléments d'une classe.

Phase 3 : Construire le coeur du système

Ceci est la conversion initiale de la conception brute en portion de code compilable et exécutable qui peut être testée, et surtout qui va permettre d'approuver ou d'invalider l'architecture retenue. Ce n'est pas un processus qui se fait en une passe, mais plutôt le début d'une série d'étapes qui vont construire le système au fur et à mesure comme le montre la Phase 4.

Le but ici est de trouver le coeur de l'architecture du système qui a besoin d'être implémenté afin de générer un système fonctionnel, sans se soucier de l'état de complétion du système dans cette passe initiale. Il s'agit ici de créer un cadre sur lequel on va pouvoir s'appuyer pour les itérations suivantes. On réalise aussi la première des nombreuses intégrations et phases de tests, et on donne les premiers retours aux clients sur ce à quoi leur système ressemblera et son état d'avancement. Idéalement, on découvre quelques-uns des risques critiques. Des changements ou des améliorations sur l'architecture originelle seront probablement découverts - des choses qu'on n'aurait pas découvert avant l'implémentation du système.

Une partie de la construction du système consiste à confronter le système avec l'analyse des besoins et les spécifications du système (quelle que soit la forme sous laquelle ils existent). Il faut s'assurer en effet que les tests vérifient les besoins et les cas d'utilisations. Quand le coeur du système est stable, on peut passer à la suite et ajouter des fonctionnalités supplémentaires.

Phase 4 : Itérer sur les cas d'utilisation

Une fois que le cadre de base fonctionne, chaque fonctionnalité ajoutée est un petit projet en elle-même. On ajoute une fonctionnalité durant une *itération*, période relativement courte du développement.

Combien de temps dure une itération ? Idéalement, chaque itération dure entre une et trois semaines (ceci peut varier suivant le langage d'implémentation choisi). A la fin de cette période, on dispose d'un système intégré et testé avec plus de fonctionnalités que celles dont il disposait auparavant. Mais ce qu'il est intéressant de noter, c'est qu'un simple cas d'utilisation constitue la base d'une itération. Chaque cas d'utilisation est un ensemble de fonctionnalités qu'on ajoute au système toutes en même temps, durant une itération. Non seulement cela permet de se faire une meilleure idée de ce que recouvre ce cas d'utilisation, mais cela permet de le valider, puisqu'il n'est pas abandonné après l'analyse et la conception, mais sert au contraire tout au long du processus de création.

Les itérations s'arrêtent quand on dispose d'un système comportant toutes les fonctionnalités

souhaitées ou qu'une date limite arrive et que le client se contente de la version courante (se rappeler que les commanditaires dirigent l'industrie du logiciel). Puisque le processus est itératif, on dispose de nombreuses opportunités pour délivrer une version intermédiaire plutôt qu'un produit final ; les projets open-source travaillent uniquement dans un environnement itératif avec de nombreux retours, ce qui précisément les rend si productifs.

Un processus de développement itératif est intéressant pour de nombreuses raisons. Cela permet de révéler et de résoudre des risques critiques très tôt, les clients ont de nombreuses opportunités pour changer d'avis, la satisfaction des programmeurs est plus élevée, et le projet peut être piloté avec plus de précision. Mais un bénéfice additionnel particulièrement important est le retour aux commanditaires du projet, qui peuvent voir grâce à l'état courant du produit où le projet en est. Ceci peut réduire ou éliminer le besoin de réunions soporifiques sur le projet, et améliore la confiance et le support des commanditaires.

Phase 5 : Évolution

Cette phase du cycle de développement a traditionnellement été appelée « maintenance », un terme fourre-tout qui peut tout vouloir dire depuis « faire marcher le produit comme il était supposé le faire dès le début » à « ajouter de nouvelles fonctionnalités que le client a oublié de mentionner » au plus traditionnel « corriger les bugs qui apparaissent » et « ajouter de nouvelles fonctionnalités quand le besoin s'en fait sentir ». Le terme « maintenance » a été la cause de si nombreux malentendus qu'il en est arrivé à prendre un sens péjoratif, en partie parce qu'il suggère qu'on a fourni un programme parfait et que tout ce qu'on a besoin de faire est d'en changer quelques parties, le graisser et l'empêcher de rouiller. Il existe peut-être un meilleur terme pour décrire ce qu'il en est réellement.

J'utiliserai plutôt le terme *évolution*[15]. C'est à dire, « Tout ne sera pas parfait dès le premier jet, il faut se laisser la latitude d'apprendre et de revenir en arrière pour faire des modifications ». De nombreux changements seront peut-être nécessaires au fur et à mesure que l'appréhension et la compréhension du problème augmentent. Si on continue d'évoluer ainsi jusqu'au bout, l'élégance obtenue sera payante, à la fois à court et long terme. L'évolution permet de passer d'un bon à un excellent programme, et clarifie les points restés obscurs durant la première passe. C'est aussi dans cette phase que les classes passent d'un statut d'utilité limitée au système à ressource réutilisable.

Ici, « jusqu'au bout » ne veut pas simplement dire que le programme fonctionne suivant les exigences et les cas d'utilisation. Cela veut aussi dire que la structure interne du code présente une logique d'organisation et semble bien s'assembler, sans abus de syntaxe, d'objets surdimensionnés ou de code inutilement exposé. De plus, il faut s'assurer que la structure du programme puisse s'adapter aux changements qui vont inévitablement arriver pendant sa durée vie, et que ces changements puissent se faire aisément et proprement. Ceci n'est pas une petite caractéristique. Il faut comprendre non seulement ce qu'on construit, mais aussi comment le programme va évoluer (ce que j'appelle le *vecteur changement*). Heureusement, les langages de programmation orientés objet sont particulièrement adaptés à ce genre de modifications continuelles - les frontières créées par les objets sont ce qui empêchent la structure du programme de s'effondrer. Ils permettent aussi de faire des changements - même ceux qui seraient considérés comme sévères dans un programme procédural - sans causer de ravages dans l'ensemble du code. En fait le support de l'évolution pourrait bien être le bénéfice le plus important de la programmation orientée objet.

Avec l'évolution, on crée quelque chose qui approche ce qu'on croit avoir construit, on le compare avec les exigences et on repère les endroits où cela coince. On peut alors revenir en arrière et corriger cela en remodélisant et réimplémentant les portions du programme qui ne fonctionnaient pas correctement [16]. De fait, on peut avoir besoin de résoudre le problème ou un de ses aspects un

certain nombre de fois avant de trouver la bonne solution (une étude de *Design Patterns* s'avère généralement utile ici). On pourra trouver plus d'informations dans *Thinking in Patterns with Java*, téléchargeable à *www.BruceEckel.com*).

Il faut aussi évoluer quand on construit un système, qu'on voit qu'il remplit les exigences et qu'on découvre finalement que ce n'était pas ce qu'on voulait. Quand on se rend compte après avoir vu le système en action qu'on essayait de résoudre un autre problème. Si on pense que ce genre d'évolution est à prendre en considération, alors on se doit de construire une première version aussi rapidement que possible afin de déterminer au plus tôt si c'est réellement ce qu'on veut.

La chose la plus importante à retenir est que par défaut - par définition, plutôt - si on modifie une classe, ses classes parentes et dérivées continueront de fonctionner. Il ne faut pas craindre les modifications (surtout si on dispose d'un ensemble de tests qui permettent de vérifier les modifications apportées). Les modifications ne vont pas nécessairement casser le programme, et tout changement apporté sera limité aux sous-classes et / ou aux collaborateurs spécifiques de la classe qu'on change.

Les plans sont payants

Bien sûr on ne bâtirait pas une maison sans une multitude de plans dessinés avec attention. Si on construit un pont ou une niche, les plans ne seront pas aussi élaborés, mais on démarre avec quelques esquisses pour se guider. Le développement de logiciels a connu les extrêmes. Longtemps les gens ont travaillé sans structure, mais on a commencé à assister à l'effondrement de gros projets. En réaction, on en est arrivé à des méthodologies comprenant un luxe de structure et de détails, destinées justement à ces gros projets. Ces méthodologies étaient trop intimidantes pour qu'on les utilise - on avait l'impression de passer son temps à écrire des documents et aucun moment à coder (ce qui était souvent le cas). J'espère que ce que je vous ai montré ici suggère un juste milieu. Utilisez une approche qui corresponde à vos besoins (et votre personnalité). Même s'il est minimal, la présence d'un plan vous apportera beaucoup dans la gestion de votre projet. Rappelez-vous que selon la plupart des estimations, plus de 50 pour cent des projets échouent (certaines estimations vont jusqu'à 70 pour cent).

En suivant un plan - de préférence un qui soit simple et concis - et en produisant une modélisation de la structure avant de commencer à coder, vous découvrirez que les choses s'arrangent bien mieux que si on se lance comme ça dans l'écriture. Vous en retirerez aussi une plus grande satisfaction. Suivant mon expérience, arriver à une solution élégante procure une satisfaction à un niveau entièrement différent ; cela ressemble plus à de l'art qu'à de la technologie. Et l'élégance est toujours payante, ce n'est pas une vaine poursuite. Non seulement on obtient un programme plus facile à construire et déboguer, mais qui est aussi plus facile à comprendre et maintenir, et c'est là que sa valeur financière réside.

J'ai étudié à différentes reprises les techniques d'analyse et de conception depuis que je suis sorti de l'école. Le concept de *Programmation Extrême* (XP) est le plus radical et divertissant que j'ai vu. Il est rapporté dans Extrême *Programming Explained* de Kent Beck (Addison-Wesley, 2000) et sur le web à *www.xprogramming.com*.

XP est à la fois une philosophie à propos de la programmation et un ensemble de règles de bases. Certaines de ces règles sont reprises dans d'autres méthodologies récentes, mais les deux contributions les plus importantes et novatrices, sont à mon sens « commencer par écrire les tests » et « programmation en binôme ». Bien qu'il soutienne et argumente l'ensemble de la théorie, Beck insiste sur le fait que l'adoption de ces deux pratiques améliore grandement la productivité et la fia-

bilité.

Commencer par écrire les tests

Les tests ont traditionnellement été relégués à la dernière partie d'un projet, une fois que « tout marche, mais c'est juste pour s'en assurer ». Ils ne sont généralement pas prioritaires et les gens qui se spécialisent dedans ne sont pas reconnus à leur juste valeur et se sont souvent vus cantonnés dans un sous-sol, loin des « véritables programmeurs ». Les équipes de test ont réagi en conséquence, allant jusqu'à porter des vêtements de deuil et glousser joyeusement quand ils trouvaient des erreurs (pour être honnête, j'ai eu moi aussi ce genre de sentiments lorsque je mettais des compilateurs en faute).

XP révolutionne complètement le concept du test en lui donnant une priorité aussi importante (ou même plus forte) que le code. En fait, les tests sont écrits avant le code qui sera testé, et les tests restent tout le temps avec le code. Ces tests doivent être exécutés avec succès à chaque nouvelle intégration dans le projet (ce qui peut arriver plus d'une fois par jour).

Écrire les tests d'abord a deux effets extrêmement importants.

Premièrement, cela nécessite une définition claire de l'interface d'une classe. J'ai souvent suggéré que les gens « imaginent la classe parfaite qui résolve un problème particulier » comme outil pour concevoir le système. La stratégie de test de XP va plus loin - elle spécifie exactement ce à quoi la classe doit ressembler pour le client de cette classe, et comment la classe doit se comporter. Dans des termes non ambigus. On peut écrire tout ce qu'on veut, ou créer tous les diagrammes qu'on veut, décrivant comment une classe devrait se comporter et ce à quoi elle ressemble, mais rien n'est aussi réel qu'une batterie de tests. Le premier est une liste de voeux, mais les tests sont un contrat certifié par un compilateur et un programme qui marche. Il est dur d'imaginer une description plus concrète d'une classe que des tests.

En créant les tests, on est forcé de penser complètement la classe et souvent on découvre des fonctionnalités nécessaires qui ont pu être manquées lors de l'utilisation des diagrammes UML, des cartes CRC, des cas d'utilisation, etc...

Le deuxième effet important dans l'écriture des tests en premier vient du fait qu'on peut lancer les tests à chaque nouvelle version du logiciel. Cela permet d'obtenir l'autre moitié des tests réalisés par le compilateur. Si on regarde l'évolution des langages de programmation de ce point de vue, on se rend compte que les améliorations réelles dans la technologie ont en fait tourné autour du test. Les langages assembleur vérifiaient uniquement la syntaxe, puis le C a imposé des restrictions sémantiques, et cela permettait d'éviter certains types d'erreurs. Les langages orientés objet imposent encore plus de restrictions sémantiques, qui sont quand on y pense des formes de test. « Est-ce que ce type de données est utilisé correctement ? » et « Est-ce que cette fonction est appelée correctement ? » sont le genre de tests effectués par le compilateur ou le système d'exécution. On a pu voir le résultat d'avoir ces tests dans le langage même : les gens ont été capables de construire des systèmes plus complexes, et de les faire marcher, et ce en moins de temps et d'efforts. Je me suis souvent demandé pourquoi, mais maintenant je réalise que c'est grâce aux tests : si on fait quelque chose de faux, le filet de sécurité des tests intégré au langage prévient qu'il y a un problème et montre même où il réside.

Mais les tests intégrés permis par la conception du langage ne peuvent aller plus loin. A partir d'un certain point, il est de notre responsabilité de produire une suite complète de tests (en coopération avec le compilateur et le système d'exécution) qui vérifie tout le programme. Et, de même qu'il est agréable d'avoir un compilateur qui vérifie ce qu'on code, ne serait-il pas préférable que ces tests

soient présents depuis le début ? C'est pourquoi on les écrit en premier, et qu'on les exécute automatiquement à chaque nouvelle version du système. Les tests deviennent une extension du filet de sécurité fourni par le langage.

L'utilisation de langages de programmation de plus en plus puissants m'a permis de tenter plus de choses audacieuses, parce que je sais que le langage m'empêchera de perdre mon temps à chasser les bugs. La stratégie de tests de XP réalise la même chose pour l'ensemble du projet. Et parce qu'on sait que les tests vont révéler tous les problèmes introduits (et on ajoute de nouveaux tests au fur et à mesure qu'on les imagine), on peut faire de gros changements sans se soucier de mettre le projet complet en déroute. Ceci est une approche particulièrement puissante.

Programmation en binôme

La programmation en binôme va à l'encontre de l'individualisme farouche endoctriné, depuis l'école (où on réussit ou échoue suivant nos mérites personnels, et où travailler avec ses voisins est considéré comme « tricher »), et jusqu'aux médias, en particulier les films hollywoodiens dans lequel le héros se bat contre la conformité [17]. Les programmeurs aussi sont considérés comme des parangons d'individualisme - des « codeurs cowboys » comme aime à le dire Larry Constantine. Et XP, qui se bat lui-même contre la pensée conventionnelle, soutient que le code devrait être écrit avec deux personnes par station de travail. Et cela devrait être fait dans un endroit regroupant plusieurs stations de travail, sans les barrières dont raffolent les gens des Moyens Généraux. En fait, Beck dit que la première tâche nécessaire pour implémenter XP est de venir avec des tournevis et d'enlever tout ce qui se trouve dans le passage [18](cela nécessite un responsable qui puisse absorber la colère du département des Moyens Généraux).

Dans la programmation en binôme, une personne produit le code tandis que l'autre y réfléchit. Le penseur garde la conception générale à l'esprit - la description du problème, mais aussi les règles de XP à portée de main. Si deux personnes travaillent, il y a moins de chance que l'une d'entre elles s'en aille en disant « Je ne veux pas commencer en écrivant les tests », par exemple. Et si le codeur reste bloqué, ils peuvent changer leurs places. Si les deux restent bloqués, leurs songeries peuvent être remarquées par quelqu'un d'autre dans la zone de travail qui peut venir les aider. Travailler en binôme permet de garder une bonne productivité et de rester sur la bonne pente. Probablement plus important, cela rend la programmation beaucoup plus sociable et amusante.

J'ai commencé à utiliser la programmation en binôme durant les périodes d'exercice dans certains de mes séminaires et il semblerait que cela enrichisse l'expérience personnelle de chacun.

Les raisons du succès de Java

Java est si populaire actuellement car son but est de résoudre beaucoup de problèmes auxquels les programmeurs doivent faire face aujourd'hui. Le but de Java est d'améliorer la productivité. La productivité vient de différents horizons, mais le langage a été conçu pour aider un maximum tout en entravant le moins possible avec des règles arbitraires ou l'obligation d'utiliser un ensemble particulier de caractéristiques. Java a été conçu pour être pratique ; les décisions concernant la conception de Java furent prises afin que le programmeur en retire le maximum de bénéfices.

Les systèmes sont plus faciles à exprimer et comprendre

Les classes conçues pour résoudre le problème sont plus faciles à exprimer. La solution est mise en oeuvre dans le code avec des termes de l'espace problème (« Mets le fichier à la poubelle »)

plutôt qu'en termes machine, l'espace solution (« Positionne le bit à 1 ce qui veut dire que le relais va se fermer »). On traite de concepts de plus haut niveau et une ligne de code est plus porteuse de sens.

L'autre aspect bénéfique de cette facilité d'expression se retrouve dans la maintenance, qui représente (s'il faut en croire les rapports) une grosse part du coût d'un programme. Si un programme est plus facile à comprendre, il est forcément plus facile à maintenir. Cela réduit aussi le coût de création et de maintenance de la documentation.

Puissance maximale grâce aux bibliothèques

La façon la plus rapide de créer un programme est d'utiliser du code déjà écrit : une bibliothèque. Un des buts fondamentaux de Java est de faciliter l'emploi des bibliothèques. Ce but est obtenu en convertissant ces bibliothèques en nouveaux types de données (classes), et utiliser une bibliothèque revient à ajouter de nouveaux types au langage. Comme le compilateur Java s'occupe de l'interfaçage avec la bibliothèque - garantissant une initialisation et un nettoyage propres, et s'assurant que les fonctions sont appelées correctement - on peut se concentrer sur ce qu'on attend de la bibliothèque, et non sur les moyens de le faire.

Traitement des erreurs

L'une des difficultés du C est la gestion des erreurs, problème connu et largement ignoré - le croisement de doigts est souvent impliqué. Si on construit un programme gros et complexe, il n'y a rien de pire que de trouver une erreur enfouie quelque part sans qu'on sache d'où elle vienne. Le *traitement des exceptions* de Java est une façon de garantir qu'une erreur a été remarquée, et que quelque chose est mis en oeuvre pour la traiter.

Mise en oeuvre de gros projets

Beaucoup de langages traditionnels imposent des limitations internes sur la taille des programmes et leur complexité. BASIC, par exemple, peut s'avérer intéressant pour mettre en oeuvre rapidement des solutions pour certains types de problèmes ; mais si le programme dépasse quelques pages de long, ou s'aventure en dehors du domaine du langage, cela revient à tenter de nager dans un liquide encore plus visqueux. Aucune limite ne prévient que le cadre du langage est dépassé, et même s'il en existait, elle serait probablement ignorée. On devrait se dire : « Mon programme BASIC devient trop gros, je vais le réécrire en C ! », mais à la place on tente de glisser quelques lignes supplémentaires pour implémenter cette nouvelle fonctionnalité. Le coût total continue donc à augmenter.

Java a été conçu pour pouvoir mettre en oeuvre toutes les tailles de projets - c'est à dire, pour supprimer ces frontières liées à la complexité croissante d'un gros projet. L'utilisation de la programmation orientée objet n'est certainement pas nécessaire pour écrire un programme utilitaire du style « Hello world ! », mais les fonctionnalités sont présentes quand on en a besoin. Et le compilateur ne relâche pas son attention pour dénicher les bugs - produisant indifféremment des erreurs pour les petits comme pour les gros programmes.

Stratégies de transition

Si on décide d'investir dans la POO, la question suivante est généralement : « Comment convaincre mes directeur / collègues / département / collaborateurs d'utiliser les objets ? ». Il faut se

mettre à leur place et réfléchir comment on ferait s'il fallait apprendre un nouveau langage et un nouveau paradigme de programmation. Ce cas s'est sûrement déjà présenté de par le passé. Il faut commencer par des cours et des exemples ; puis lancer un petit projet d'essai pour inculquer les bases sans les perdre dans les détails. Ensuite passer à un projet « réel » qui réalise quelque chose d'utile. Au cours du premier projet, continuer l'apprentissage par la lecture, poser des questions à des experts et échanger des astuces avec des amis. Cette approche est celle recommandée par de nombreux programmeurs expérimentés pour passer à Java. Si toute l'entreprise décide de passer à Java, cela entraînera bien sûr une dynamique de groupe, mais cela aide de se rappeler à chaque étape comment une personne seule opérerait.

Règles de base

Voici quelques indications à prendre en compte durant la transition vers la programmation orientée objet et Java.

1. Cours

La première étape comprend une formation. Il ne faut pas jeter la confusion pendant six ou neuf mois dans l'entreprise, le temps que tout le monde comprenne comment fonctionnent les interfaces. Il vaut mieux se contenter d'un petit groupe d'endoctrinement, composé de préférence de personnes curieuses, s'entendant bien entre elles et suffisamment indépendantes pour créer leur propre réseau de support concernant l'apprentissage de Java.

Une approche alternative suggérée quelquefois comprend la formation de tous les niveaux de l'entreprise à la fois, en incluant à la fois des cours de présentation pour les responsables et des cours de conception et de programmation pour les développeurs. Ceci est particulièrement valable pour les petites entreprises qui n'hésitent pas à introduire des changements radicaux dans leurs manières de faire, ou au niveau division des grosses entreprises. Cependant, comme le coût résultant est plus important, certains choisiront de commencer avec des cours, de réaliser un projet pilote (si possible avec un mentor extérieur), et de laisser l'équipe du projet devenir les professeurs pour le reste de l'entreprise.

2. Projet à faible risque

Tester sur un projet à faible risque qui autorise les erreurs. Une fois gagnée une certaine expérience, on peut soit répartir les membres de cette première équipe sur les autres projets ou utiliser cette équipe comme support technique de la POO. Le premier projet peut ne pas fonctionner correctement dès le début, il ne faut donc pas qu'il soit critique pour l'entreprise. Il doit être simple, indépendant et instructif : cela veut dire qu'il doit impliquer la création de classes qui soient compréhensibles aux autres programmeurs quand arrivera leur tour d'apprendre Java.

3. S'inspirer de bonnes conceptions

Rechercher des exemples de bonnes conceptions orientées objet avant de partir de zéro. Il y a de fortes chances que quelqu'un ait déjà résolu le problème, et s'ils ne l'ont pas résolu exactement modifier le système existant (grâce à l'abstraction, cf plus haut) pour qu'il remplisse nos besoins. C'est le concept général des *patrons génériques*, couverts dans *Thinking in Patterns with Java*, téléchargeable à *www.BruceEckel.com*.

4. Utiliser les bibliothèques de classes existantes

La motivation économique première pour passer à la POO est la facilité de réutilisation de code existant sous la forme de bibliothèques de classes (en particulier, les Bibliothèques Standard Java, couvertes tout au long de ce livre). On arrive au cycle de développement le plus court quand on crée et utilise des objets directement tirés des bibliothèques. Cependant, de nouveaux programmeurs ne saisissent pas ce point, ne connaissent pas l'existence de telles bibliothèques ou, fascinés par le langage, veulent réécrire des classes qui existent déjà. Le succès de la transition vers Java et la POO passe par des efforts pour encourager les gens à réutiliser le plus possible le code des autres.

5. Ne pas traduire du code existant en Java

Ce n'est généralement pas le meilleur usage de son temps qu'on puisse trouver que de prendre du code existant et fonctionnel et de le traduire en Java (si on doit le transformer en objets, on peut s'interfacer avec du code C ou C++ en utilisant l'Interface Native Java - JNI - décrite dans l'annexe B). Il y a bien sûr des avantages à le faire, particulièrement si ce code est destiné à être réutilisé. Mais il y a de fortes chances pour qu'on passe à côté de la spectaculaire amélioration de productivité espérée sauf si ce premier projet en est un nouveau. Java et la POO sont mis en valeur quand on suit un projet de la conception à la mise en oeuvre.

Les obstacles au niveau du management

Si on se situe du côté des responsables, le travail consiste à acquérir des ressources pour l'équipe, surmonter les obstacles pouvant gêner l'équipe, et en général tenter de fournir l'environnement le plus productif et agréable afin que l'équipe puisse accomplir ces miracles qu'on vous demande toujours de réaliser. Le passage à Java entre dans ces trois catégories, et ce serait merveilleux si de plus cela ne coûtait rien. Bien que le passage à Java puisse être moins onéreux - suivant les contraintes - que d'autres alternatives de POO pour une équipe de programmeurs C (et probablement pour les développeurs dans d'autres langages), ce coût n'est pas nul, et il y a des obstacles dont il faut être conscient avant de promouvoir le passage à Java à l'intérieur de l'entreprise et de s'embarquer dans la transition.

Coûts de mise en oeuvre

Le coût du passage à Java recouvre plus que l'acquisition d'un compilateur Java (le compilateur Java fourni par Sun est gratuit, cela n'est donc pas un obstacle). Les coûts à moyen et long terme seront minimisés si on investit dans la formation (et éventuellement dans la participation d'un consultant pour le premier projet), de même que si on identifie et achète des bibliothèques de classes qui résolvent les problèmes plutôt que de tenter de réécrire ces bibliothèques soi-même. Ce sont des investissements lourds qui doivent être relativisés dans une proposition raisonnable. De plus, il existe des coûts cachés dans la baisse de productivité liée à l'apprentissage d'un nouveau langage et éventuellement d'un nouvel environnement de développement. La formation et l'encadrement peuvent certainement les réduire, mais les membres de l'équipe devront mener leur propre combat pour maîtriser la nouvelle technologie. Durant cette phase ils feront plus d'erreurs (c'est prouvé, et les erreurs comprises constituent le meilleur moyen de progresser) et seront moins productifs. Cependant, avec certains types de problèmes, les bonnes classes et le bon environnement de développement, il est possible d'être plus productif pendant l'apprentissage de Java (même en considérant qu'on fait plus d'erreurs et qu'on écrit moins de lignes de code par jour) que si on en était res-

té au C.

Problèmes de performances

Une question qui revient souvent est : « Est-ce que la POO ne rend pas obligatoirement mes programmes plus gros et plus lents ? ». La réponse est « Ca dépend. ». Les fonctionnalités de vérification introduites dans Java ont prélevé leur tribut sur la performance comparé à un langage comme le C++. Des technologies telles que le « hotspot » et les techniques de compilation ont significativement amélioré la vitesse dans la plupart des cas, et les efforts pour des performances accrues continuent.

Quand on se concentre sur le prototypage rapide, on assemble des composants aussi vite que possible en ignorant les problèmes liés à l'efficacité. Si on utilise des bibliothèques extérieures, celles-ci ont généralement été optimisées par leurs distributeurs ; de toutes façons ce n'est pas la préoccupation première quand on est en phase de développement rapide. Quand on dispose d'un système qui nous satisfait et s'il est suffisamment petit et rapide, alors le tour est joué. Sinon, on tente de mettre au point avec un outil de profilage, en cherchant en premier des améliorations qui peuvent être faites en réécrivant de petites portions de code. Si cela ne suffit pas, il faut chercher si on peut apporter quelques changements à l'implémentation sous-jacente afin qu'aucune classe particulière n'ait besoin de modifier son code. Il ne faut toucher à la conception que si rien d'autre ne résout le problème. Le fait que les performances soient si critiques dans cette phase de la conception est un indicateur que ce doit être un des critères essentiels de la conception. On a le bénéfice de s'en rendre compte relativement tôt en utilisant le prototypage rapide.

Si on trouve une fonction qui soit un goulot d'étranglement, on peut la réécrire en C ou en C++ en utilisant les *méthodes natives* de Java, sujet de l'annexe B.

Erreurs classiques de conception

Quand l'équipe commencera la programmation orientée objet et Java, les programmeurs vont typiquement passer par une série d'erreurs classiques de conception. Ceci est souvent du à des retours insuffisants de la part d'experts durant la conception et l'implémentation des premiers projets, puisqu'aucun expert n'existe encore au sein de l'entreprise et qu'il peut y avoir des réticences à engager des consultants. Il est facile de se dire trop tôt qu'on maîtrise la POO et partir sur de mauvaises bases. Quelque chose d'évident pour une personne expérimentée peut être le sujet d'un débat interne intense pour un novice. La plupart de ces difficultés peuvent être évitées en utilisant un expert extérieur pour la formation.

Java ressemble beaucoup au C++, et il semblerait naturel que le C++ soit remplacé par Java. Mais je commence à m'interroger sur cette logique. D'une part, C++ dispose toujours de fonctionnalités que Java n'a pas, et bien que de nombreuses promesses aient été faites sur le fait que Java soit un jour aussi rapide, voire même plus, que le C++, on a vu de grosses améliorations mais pas de révolutions spectaculaires. De plus, il semblerait que le C++ intéresse une large communauté passionnée, et je ne pense donc pas que ce langage puisse disparaître prochainement (les langages semblent résister au cours du temps. Allen Hollub a affirmé durant l'un de mes « Séminaires Java Intermédiaire/Avancé » que les deux langages les plus utilisés étaient Rexx et COBOL, dans cet ordre).

Je commence à croire que la force de Java réside dans une optique différente de celle du C++. C++ est un langage qui n'essaie pas de se fondre dans un moule. Il a déjà été adapté un certain nombre de fois pour résoudre des problèmes particuliers. Certains des outils du C++ combinent des bibliothèques, des modèles de composants et des outils de génération de code pour résoudre les

problèmes concernant le développement d'applications fenêtrées (pour Microsoft Windows). Et pourtant, la vaste majorité des développeurs Windows utilisent Microsoft Visual Basic (VB). Et ceci malgré le fait que VB produise le genre de code qui devient ingérable quand le programme fait plus de quelques pages de long (sans compter que la syntaxe peut être profondément mystérieuse). Aussi populaire que soit VB, ce n'est pas un très bon exemple de conception de langage. Il serait agréable de pouvoir disposer des facilités et de la puissance fournies par VB sans se retrouver avec ce code ingérable. Et c'est là où je pense que Java va pouvoir briller : comme le « VB du futur ». On peut frissonner en entendant ceci, mais Java est conçu pour aider le développeur à résoudre des problèmes comme les applications réseaux ou interfaces utilisateurs multiplateformes, et la conception du langage permet la création de portions de code très importantes mais néanmoins flexibles. Ajoutons à ceci le fait que Java dispose des systèmes de vérifications de types et de gestion des erreurs les plus robustes que j'ai jamais rencontré dans un langage et on se retrouve avec les éléments constitutifs d'un bond significatif dans l'amélioration de la productivité dans la programmation.

Faut-il utiliser Java en lieu et place du C++ dans les projets ? En dehors des applets Web, il y a deux points à considérer. Premièrement, si on veut réutiliser un certain nombre de bibliothèques C ++ (et on y gagnera certainement en productivité), ou si on dispose d'une base existante en C ou C+ +, Java peut ralentir le développement plutôt que l'accélérer.

Si on développe tout le code en partant de zéro, alors la simplicité de Java comparée au C++ réduira significativement le temps de développement - des anecdotes (selon des équipes C++ à qui j'ai parlé après qu'ils eurent changé pour Java) suggèrent un doublement de la vitesse de développement comparé au C++. Si les performances moindres de Java ne rentrent pas en ligne de compte ou qu'on peut les compenser, les contraintes de temps font qu'il est difficile de choisir le C++ aux détriments de Java.

Le point le plus important est la performance. Java interprété est lent, environ 20 à 50 fois plus lent que le C dans les interpréteurs Java originels. Ceci a été grandement amélioré avec le temps, mais il restera toujours un important facteur de différence. Les ordinateurs existent de par leur rapidité ; si ce n'était pas considérablement plus rapide de réaliser une tâche sur ordinateur, on la ferait à la main. J'ai même entendu suggérer de démarrer avec Java, pour gagner sur le temps de développement plus court, et ensuite utiliser un outil et des bibliothèques de support pour traduire le code en C++ si on a un besoin de vitesse d'exécution plus rapide.

La clef pour rendre Java viable dans la plupart des projets consiste en des améliorations de vitesse d'exécution, grâce à des compilateurs « juste à temps » (« just in time », JIT), la technologie « hotspot » de Sun, et même des compilateurs de code natif. Bien sûr, les compilateurs de code natif éliminent les possibilités d'exécution interplateformes du programme compilé, mais la vitesse des exécutables produits se rapprochera de celle du C et du C++. Et réaliser un programme multiplateformes en Java devrait être beaucoup plus facile qu'en C ou C++ (en théorie, il suffit de recompiler, mais cette promesse a déjà été faite pour les autres langages).

Vous trouverez des comparaisons entre Java et C++ et des observations sur Java dans les annexes de la première édition de ce livre (disponible sur le CD ROM accompagnant ce livre, et à *www.BruceEckel.com*).

Résumé

Ce chapitre tente de vous donner un aperçu des sujets couverts par la programmation orientée objet et Java (les raisons qui font que la POO est particulière, de même que Java), les concepts des méthodologies de la POO, et finalement le genre de problèmes que vous rencontrerez quand vous

migrerez dans votre entreprise à la programmation orientée objet et Java.

La POO et Java ne sont pas forcément destinés à tout le monde. Il est important d'évaluer ses besoins et décider si Java satisfera au mieux ces besoins, ou si un autre système de programmation ne conviendrait pas mieux (celui qu'on utilise actuellement y compris). Si on connaît ses besoins futurs et qu'ils impliquent des contraintes spécifiques non satisfaites par Java, alors on se doit d'étudier les alternatives existantes [19]. Et même si finalement Java est retenu, on saura au moins quelles étaient les options et les raisons de ce choix.

On sait à quoi ressemble un programme procédural : des définitions de données et des appels de fonctions. Pour trouver le sens d'un tel programme il faut se plonger dans la chaîne des appels de fonctions et des concepts de bas niveau pour se représenter le modèle du programme. C'est la raison pour laquelle on a besoin de représentations intermédiaires quand on conçoit des programmes procéduraux - par nature, ces programmes tendent à être confus car le code utilise des termes plus orientés vers la machine que vers le problème qu'on tente de résoudre.

Parce que Java introduit de nombreux nouveaux concepts par rapport à ceux qu'on trouve dans un langage procédural, on pourrait se dire que la fonction **main()** dans un programme Java sera bien plus compliquée que son équivalent dans un programme C. On sera agréablement surpris de constater qu'un programme Java bien écrit est généralement beaucoup plus simple et facile à comprendre que son équivalent en C. On n'y voit que les définitions des objets qui représentent les concepts de l'espace problème (plutôt que leur représentation dans l'espace machine) et les messages envoyés à ces objets pour représenter les activités dans cet espace. L'un des avantages de la POO est qu'avec un programme bien conçu, il est facile de comprendre le code en le lisant. De plus, il y a généralement moins de code, car beaucoup de problèmes sont résolus en réutilisant du code existant dans des bibliothèques.

[2]Voir *Multiparadigm Programming in Leda* de Timothy Budd (Addison-Wesley 1995).

[3]Certaines personnes établissent une distinction, en disant qu'un type détermine une interface tandis qu'une classe est une implémentation particulière de cette interface.

[4]Je suis reconnaissant envers mon ami Scott Meyers pour cette expression.

[5]Cela suffit généralement pour la plupart des diagrammes, et on n'a pas besoin de préciser si on utilise un agrégat ou une composition.

[6]Invention personnelle.

[7]Les types primitifs, que vous rencontrerez plus loin, sont un cas spécial.

[8]Un très bon exemple en est *UML Distilled*, 2nd edition, de Martin Fowler (Addison-Wesley 2000), qui réduit la méthode UML - parfois écrasante - à un sous-ensemble facilement gérable.

[9]Ma règle pour estimer de tels projets : s'il y a plus d'un facteur joker, n'essayez même pas de planifier combien de temps cela va prendre ou d'estimer le coût avant d'avoir créé un prototype fonctionnel. Il y a trop de degrés de liberté.

[10]Merci à James H Jarrett pour son aide.

[11]D'autres informations sur les cas d'utilisation peuvent être trouvées dans *Applying Use Cases* de Schneider & Winters (Addison-Wesley 1998) et *Use Case Driven Object Modeling with UML* de Rosenberg (Addison-Wesley 1999).

[12]Mon avis personnel sur tout cela a changé récemment. Doubler et ajouter 10 pour cent donnera une estimation raisonnablement précise (en assumant qu'il n'y ait pas trop de facteurs

joker), mais il faudra tout de même ne pas traîner en cours de route pour respecter ce délai. Si on veut réellement du temps pour rendre le système élégant et ne pas être continuellement sous pression, le multiplicateur correct serait plus trois ou quatre fois le temps prévu, je pense..

[13]Pour les débutants, je recommande *UML Distilled*, 2nd edition, déjà mentionné plus haut.

[14]Python (www.python.org) est souvent utilisé en tant que « pseudo code exécutable ».

[15]Au moins un aspect de l'évolution est couvert dans le livre de Martin Fowler *Refactoring : improving the design of existing code (Addison-Wesley 1999)*, qui utilise exclusivement des exemples Java.

[16]Cela peut ressembler au « prototypage rapide » où on est supposé fournir une version rapide-et-sale afin de pouvoir appréhender correctement le problème, puis jeter ce prototype et construire une version finale acceptable. L'inconvénient du prototypage rapide est que les gens ne jettent pas le prototype, mais s'en servent de base pour le développement. Combiné au manque de structure de la programmation procédurale, cela mène à des systèmes compliqués et embrouillés, difficiles et onéreux à maintenir.

[17]Bien que ceci soit plus une perception américaine, les productions hollywoodiennes touchent tout le monde.

[18]En particulier le système d'annonces sonores. J'ai travaillé une fois dans une entreprise qui insistait pour diffuser tous les appels téléphoniques à tous les responsables, et cela interrompait constamment notre productivité (mais les responsables n'imaginaient même pas qu'il soit concevable de vouloir couper un service aussi important que le système d'annonces sonores). Finalement, j'ai coupé les fils des haut-parleurs quand personne ne regardait.

[19]En particulier, je recommande de regarder dans la direction de Python (www.Python.org).

Chapitre 2 - Tout est « objet »

Bien qu'il soit basé sur C++, Java est un langage orienté objet plus « pur ».

C++ et Java sont tous les deux des langages hybrides, mais dans Java, les concepteurs ont pensé que l'hybridation est moins importante qu'elle ne l'est en C++. Un langage hybride autorise plusieurs styles de programmation : C++ est hybride pour assurer la compatibilité avec le langage C. Comme C++ est une extension du langage C, il contient un grand nombre des particularités indésirables de ce langage, ce qui peut rendre certains aspects du C++ particulièrement embrouillés.

Le langage Java suppose qu'on ne veut faire que de la programmation orientée objet (POO). Ceci signifie qu'avant de pouvoir commencer il faut tourner sa vision des choses vers un monde orienté objets (à moins qu'elle ne le soit déjà). L'avantage de cet effort préliminaire est la capacité à programmer dans un langage qui est plus simple à apprendre et à utiliser que beaucoup d'autres langages de POO. Dans ce chapitre nous verrons les composantes de base d'un programme Java et nous apprendrons que tout dans Java est objet, même un programme Java.

Les objets sont manipulés avec des références

Chaque langage de programmation a ses propres façons de manipuler les données. Parfois le programmeur doit être constamment conscient du type des manipulations en cours. Manipulez-vous l'objet directement, ou avez-vous affaire à une sorte de représentation indirecte (un pointeur en C ou C++) qui doit être traité avec une syntaxe particulière ?

Tout ceci est simplifié en Java. On considère tout comme des objets, ainsi il n'y a qu'une seule syntaxe cohérente qui est utilisée partout. Bien qu'on *traite* tout comme des objets, les identificateurs qui sont manipulés sont en réalité des « références » vers des objets [21]. On pourrait imaginer cette situation comme une télévision (l'objet) avec une télécommande (la référence). Tant qu'on conserve cette référence, on a une liaison vers la télévision, mais quand quelqu'un dit « change de chaîne » ou « baisse le volume », ce qu'on manipule est la référence, qui en retour modifie l'objet. Si on veut se déplacer dans la pièce tout en contrôlant la télévision, on emporte la télécommande/référence, pas la télévision.

De plus, la télécommande peut exister par elle même sans télévision. C'est à dire que le fait d'avoir une référence ne signifie pas nécessairement qu'un objet y soit associé. Ainsi, si on veut avoir avoir un mot ou une phrase, on crée une référence sur une **String** :

```
String s;
```

Mais on a *seulement* créé la référence, pas un objet. À ce point, si on décidait d'envoyer un message à **s**, on aurait une erreur (lors de l'exécution) parce que **s** n'est pas rattachée à quoi que ce soit (il n'y a pas de télévision). Une pratique plus sûre est donc de toujours initialiser une référence quand on la crée :

```
String s = "asdf";
```

Toutefois, ceci utilise une caractéristique spéciale de Java : les chaînes de caractères peuvent être initialisées avec du texte entre guillemets. Normalement, on doit utiliser un type d'initialisation plus général pour les objets.

Vous devez créer tous les objets

Quand on crée une référence, on veut la connecter à un nouvel objet. Ceci se fait, en général, avec le mot-clef **new**. **new** veut dire « fabrique moi un de ces objets ». Ainsi, dans l'exemple précédent, on peut dire :

```
String s = new String("asdf");
```

Ceci ne veut pas seulement dire « fabrique moi un nouvel objet **String** », mais cela donne aussi une information sur *comment* fabriquer l'objet **String** en fournissant une chaîne de caractères initiale.

Bien sûr, **String** n'est pas le seul type qui existe. Java propose une pléthore de types tout prêts. Le plus important est qu'on puisse créer ses propres types. En fait, c'est l'activité fondamentale en programmation Java et c'est ce qu'on apprendra à faire dans la suite de ce livre.

Où réside la mémoire ?

Il est utile de visualiser certains aspects de comment les choses sont arrangées lorsque le programme tourne, en particulier comment la mémoire est organisée. Il y a six endroits différents pour stocker les données :

1. **Les registres**. C'est le stockage le plus rapide car il se trouve à un endroit différent des autres zones de stockage : dans le processeur. Toutefois, le nombre de registres est sévèrement limité, donc les registres sont alloués par le compilateur en fonction de ses besoins. On n'a aucun contrôle direct et il n'y a même aucune trace de l'existence des registres dans les programmes.

2. **La pile**. Elle se trouve dans la RAM (random access memory) mais elle est prise en compte directement par le processeur via son *pointeur de pile*. Le pointeur de pile est déplacé vers le bas pour créer plus d'espace mémoire et déplacé vers le haut pour libérer cet espace. C'est un moyen extrêmement efficace et rapide d'allouer de la mémoire, supplanté seulement par les registres. Le compilateur Java doit connaître, lorsqu'il crée le programme, la taille et la durée de vie exacte de toutes les données qui sont rangées sur la pile, parce qu'il doit générer le code pour déplacer le pointeur de pile vers le haut et vers le bas. Cette contrainte met des limites à la flexibilité des programmes, donc, bien qu'il y ait du stockage Java sur la pile -- en particulier les références aux objets -- les objets Java eux même ne sont pas placés sur la pile.

3. **Le segment**. C'est une réserve de mémoire d'usage général (aussi en RAM) où résident tous les objets java. La bonne chose à propos du segment est que, contrairement à la pile, le compilateur n'a pas besoin de savoir de combien de place il a besoin d'allouer sur le segment ni combien de temps cette place doit rester sur le segment.Ainsi, il y a une grande flexibilité à utiliser la mémoire sur le segment. Lorsqu'on a besoin de créer un objet, il suffit d'écrire le code pour le créer en utilisant **new** et la mémoire est allouée sur le segment lorsque le programme s'exécute. Bien entendu il y a un prix à payer pour cette flexibilité : il faut plus de temps pour allouer de la mémoire sur le segment qu'il n'en faut pour allouer de la mémoire sur la pile (c'est à dire si on *avait* la possibilité de créer des objets sur la pile en Java, comme on peut le faire en C++).

4. **La mémoire statique**. « Statique » est utilisé ici dans le sens « à un endroit fixe » (bien que ce soit aussi dans la RAM). La mémoire statique contient les données qui sont disponibles pendant tout le temps d'exécution du programme. On peut utiliser le mot-clef **static** pour

spécifier qu'un élément particulier d'un objet est statique, mais les objets Java par eux-mêmes ne sont jamais placés dans la mémoire statique.

5. **Les constantes**. Les valeurs des constantes sont souvent placées directement dans le code du programme, ce qui est sûr puisqu'elles ne peuvent jamais changer. Parfois les constantes sont isolées de façon à pouvoir être optionnellement placées dans une mémoire accessible en lecture seulement (ROM).

6. **Stockage hors RAM**. Si les données résident entièrement hors du programme, elles peuvent exister même quand le programme ne tourne pas, en dehors du contrôle du programme. Les deux exemples de base sont les *flots de données*, pour lesquels les données sont transformées en flots d'octets, généralement pour être transmises vers une autre machine, et les *objets persistants*, pour lesquels les objets sont placés sur disque de façon à ce qu'ils conservent leur état même après que le programme soit terminé. L'astuce avec ces types de stockage est de transformer les objets en quelque chose qui peut exister sur l'autre support, tout en pouvant être ressuscité en un objet normal en mémoire, lorsque c'est nécessaire. Java fournit des outils pour la *persistance légère*, et les versions futures pourraient fournir des solutions plus complètes pour la persistance.

Cas particulier : les types primitifs

Il y a un ensemble de types qui sont soumis à un traitement particulier ; ils peuvent être considérés comme les types « primitifs » fréquemment utilisés en programmation. La raison de ce traitement particulier est que la création d'un objet avec **new** -- en particulier une simple variable -- n'est pas très efficace parce que **new** place les objets sur le segment. Pour ces types, Java a recours à l'approche retenue en C et en C++. Au lieu de créer la variable en utilisant **new**, une variable « automatique », qui n'est pas une référence, est créée. La variable contient la valeur et elle est placée sur la pile, ce qui est beaucoup plus efficace.

Java fixe la taille de chacun des types primitifs. Ces tailles ne changent pas d'une architecture de machine à une autre, comme c'est le cas dans la plupart des langages. L'invariance de la taille de ces types est l'une des raisons pour lesquelles Java est si portable.

Type primitif	**Taille**	**Intervalles**	**Type wrapper**
boolean	Selon jvm	**True** ou **false**	**Boolean**
char	16-bit	0 à 65535	**Character**
byte	8-bit	-128 à 127	**Byte**
short	16-bit	-32 768 à 32 767	**Short**
int	32-bit	-2 147 483 648 à 2 147 483 647	**Integer**
long	64-bit	-2^64 à (2^64)-1	**Long**
float	32-bit	IEEE754	**Float**
double	64-bit	IEEE754	**Double**
void	-	-	**Void**

Tous les types numériques sont signés, il est donc inutile d'aller chercher après des types non signés.

Les types de données primitifs sont aussi associés à des classes « wrapper ». Ceci signifie que pour faire un objet non primitif sur le segment pour représenter ce type primitif il faut utiliser le wrapper associé. Par exemple :

```
char c = 'x';Character C =
  new Character(c);
```

On peut aussi utiliser :

```
Character C = new
  Character('x');
```

Les raisons pour lesquelles on fait ceci seront indiquées dans un prochain chapitre.

Nombres de grande précision

Java contient deux classes pour effectuer des opérations arithmétiques de grande précision : **BigInteger** et **BigDecimal**. Bien que ceux-ci soient dans la même catégorie que les classes « wrapper », aucun d'eux n'a d'analogue primitif.

Chacune de ces classes a des méthodes qui fournissent des opérations analogues à celles qu'on peut faire sur les types primitifs. C'est à dire qu'avec un **BigInteger** ou un **BigDecimal** on peut faire tout ce qu'on peut faire avec un **int** ou un **float**, seulement il faut utiliser des appels de méthodes au lieu des opérateurs. Par ailleurs, comme elles en font plus, les opérations sont plus lentes. On échange la vitesse contre la précision.

BigInteger sert aux entiers de précision arbitraire. C'est à dire qu'ils permettent de représenter des valeurs entières de n'importe quelle taille sans perdre aucune information au cours des opérations.

BigDecimal sert aux nombres à virgule fixe de précision arbitraire ; par exemple, on peut les utiliser pour des calculs monétaires précis.

Il faut se reporter à la documentation en ligne pour obtenir des détails sur les constructeurs et méthodes utilisables avec ces deux classes.

Tableaux en Java

Pratiquement tous les langages de programmation gèrent les tableaux. Utiliser des tableaux en C ou C++ est dangereux car ces tableaux ne sont que des blocs de mémoire. Si un programme accède à un tableau en dehors de son bloc mémoire, ou s'il utilise la mémoire avant initialisation (erreurs de programmation fréquentes) les résultats seront imprévisibles.

Un des principaux objectifs de Java est la sécurité, aussi, un grand nombre des problèmes dont souffrent C et C++ ne sont pas répétés en Java. On est assuré qu'un tableau Java est initialisé et qu'il ne peut pas être accédé en dehors de ses bornes. La vérification des bornes se fait au prix d'un petit excédent de mémoire pour chaque tableau ainsi que de la vérification de l'index lors de l'exécution, mais on suppose que le gain en sécurité et en productivité vaut la dépense.

Quand on crée un tableau d'objets, on crée en réalité un tableau de références, et chacune de

ces références est automatiquement initialisée à une valeur particulière avec son propre mot clé : **null**. Quand Java voit **null**, il reconnaît que la référence en question ne pointe pas vers un objet. Il faut affecter un objet à chaque référence avant de l'utiliser et si on essaye d'utiliser une référence encore à **null**, le problème sera signalé lors de l'exécution. Ainsi, les erreurs typiques sur les tableaux sont évitées en Java.

On peut aussi créer des tableaux de variables de type primitif. À nouveau, le compilateur garantit l'initialisation car il met à zéro la mémoire utilisée par ces tableaux.

Les tableaux seront traités plus en détails dans d'autres chapitres.

Vous n'avez jamais besoin de détruire un objet

Dans la plupart des langages de programmation, le concept de durée de vie d'une variable monopolise une part significative des efforts de programmation. Combien de temps une variable existe-t-elle ? S'il faut la détruire, quand faut-il le faire ? Des erreurs sur la durée de vie des variables peuvent être la source de nombreux bugs et cette partie montre comment Java simplifie énormément ce problème en faisant le ménage tout seul.

Notion de portée

La plupart des langages procéduraux ont le concept de *portée*. Il fixe simultanément la visibilité et la durée de vie des noms définis dans cette portée. En C, C++ et Java, la portée est fixée par l'emplacement des accolades {}. Ainsi, par exemple :

```
{
 int x = 12;
 /* seul x est accessible */
 {
  int q = 96;
  /* x & q sont tous les deux accessibles */
 }
 /* seul x est accessible */
 /* q est « hors de portée » */
}
```

Une variable définie dans une portée n'est accessible que jusqu'à la fin de cette portée.

L'indentation rend le code Java plus facile à lire. Étant donné que Java est un langage indépendant de la mise en page, les espaces, tabulations et retours chariots supplémentaires ne changent pas le programme.

Il faut remarquer qu'on ne peut pas faire la chose suivante, bien que cela soit autorisé en C et C++ :

```
{
 int x = 12;
 {
  int x = 96; /* illegal */
 }
}
```

Le compilateur annoncera que la variable **x** a déjà été définie. Ainsi, la faculté du C et du C++ à « cacher » une variable d'une portée plus étendue n'est pas autorisée parce que les concepteurs de Java ont pensé que ceci mène à des programmes confus.

Portée des objets

Les objets Java n'ont pas la même durée de vie que les variables primitives. Quand on crée un objet Java avec **new**, il existe toujours après la fin de la portée. Ainsi, si on fait :

```
{
  String s = new String("a string");
} /* fin de portée */
```

la référence **s** disparaît à la fin de la portée. Par contre l'objet **String** sur lequel **s** pointait occupe toujours la mémoire. Dans ce bout de code, il n'y a aucun moyen d'accéder à l'objet parce que son unique référence est hors de portée. Dans d'autres chapitres on verra comment la référence à un objet peut être transmise et dupliquée dans un programme.

Il s'avère que du simple fait qu'un objet créé avec new reste disponible tant qu'on le veut, tout un tas de problèmes de programmation du C++ disparaissent tout simplement en Java. Il semble que les problèmes les plus durs surviennent en C++ parce que le langage ne fournit aucune aide pour s'assurer que les objets sont disponibles quand on en a besoin. Et, encore plus important, en C++ on doit s'assurer qu'on détruit bien les objets quand on en a terminé avec eux.

Ceci amène une question intéressante. Si Java laisse les objets traîner, qu'est-ce qui les empêche de complètement remplir la mémoire et d'arrêter le programme ? C'est exactement le problème qui surviendrait dans un programme C++. C'est là qu'un peu de magie apparaît. Java a un ramasse-miettes qui surveille tous les objets qui ont été créés avec **new** et qui arrive à deviner lesquels ne sont plus référencés. Ensuite il libère la mémoire de ces objets de façon à ce que cette mémoire puisse être utilisée pour de nouveaux objets. Ceci signifie qu'il ne faut jamais s'embêter à récupérer la mémoire soi-même. On crée simplement les objets, et quand on n'en a plus besoin, ils disparaissent d'eux même. Ceci élimine toute une classe de problèmes de programmation : les soi-disant « fuites de mémoire » qui arrivent quand un programmeur oublie de libérer la mémoire.

Créer de nouveaux types de données : class

Si tout est objet, qu'est-ce qui définit à quoi ressemble une classe particulière d'objets et comment elle se comporte ? Autrement dit, qu'est-ce qui constitue le *type* d'un objet ? On pourrait s'attendre à avoir un mot-clef appelé « type », et cela serait parfaitement sensé. Historiquement, toutefois, la plupart des langages orientés objet ont utilisé le mot-clef **class** qui signifie « je vais décrire à quoi ressemble un nouveau type d'objet ». Le mot-clef **class** (qui est si commun qu'il ne sera plus mis en gras dans la suite de ce livre) est suivi par le nom du nouveau type. Par exemple :

```
class ATypeName { /* le corps de la classe vient ici */ }
```

Ceci introduit un nouveau type, on peut alors créer un objet de ce type en utilisant **new** :

```
ATypeName a = new ATypeName();
```

Dans **ATypeName**, le corps de la classe ne consiste qu'en un commentaire (les étoiles et barres obliques et ce qu'il y a à l'intérieur, ce qui sera décrit ultérieurement dans ce chapitre), donc il

n'y pas grand chose à faire avec. En fait, on ne peut pas lui dire de faire quoi que ce soit (c'est à dire qu'on ne peut pas lui transmettre de message intéressant) tant qu'on n'y définit pas de méthodes.

Lorsqu'on définit une classe (et tout ce que l'on fait en Java consiste à définir des classes, fabriquer des objets à partir de ces classes et envoyer des messages à ces objets) on peut mettre deux types d'éléments dans ces classes : des données membres de la classe (aussi appelées champs) et des fonctions membres de la classe (habituellement appelées méthodes). Une donnée membre est un objet de n'importe quel type avec lequel on peut communiquer via sa référence. Il peut aussi s'agir d'un des types primitifs (dans ce cas, ce n'est pas une référence). S'il s'agit d'une référence à un objet, il faut initialiser cette référence pour la connecter à un objet réel (en utilisant **new** comme indiqué précédemment) grâce à une fonction particulière appelée un *constructeur* (entièrement décrit dans le chapitre 4). S'il s'agit d'un type primitif il est possible de l'initialiser directement lors de sa définition dans la classe (comme on le verra plus tard, les références peuvent aussi être initialisées lors de la définition).

Chaque objet met ses données membres dans sa zone de mémoire propre, les données membres ne sont pas partagées entre les objets. Voici un exemple de classe avec des données membres :

```
class DataOnly {
  int i;
  float f;
  boolean b;
}
```

Cette classe ne fait rien mais on peut créer un objet :

```
DataOnly d = new DataOnly();
```

On peut affecter des valeurs aux données membres mais il faut d'abord savoir comment faire référence à un membre d'un objet. Ceci s'effectue en indiquant le nom de la référence à l'objet, suivi par un point, suivi par le nom du membre dans l'objet :

```
objectReference.member
```

Par exemple :

```
d.i = 47;
d.f = 1.1f;
d.b = false
```

Il est aussi possible que l'objet puisse contenir d'autres objets qui contiennent des données que l'on souhaite modifier. Pour cela il suffit de continuer à « associer les points ». Par exemple :

```
myPlane.leftTank.capacity = 100;
```

La classe **DataOnly** ne peut pas faire grand chose à part contenir des données car elle n'a pas de fonctions membres (méthodes). Pour comprendre comment celles-ci fonctionnent il faut d'abord comprendre les notions de *paramètres* et de *valeurs de retour*, qui seront brièvement décrites.

Valeurs par défaut des membres primitifs

Quand une donnée d'un type primitif est membre d'une classe on est assuré qu'elle a une valeur par défaut si on ne l'initialise pas :

Type primitif	Valeur par défaut
boolean	**false**
char	`\u0000' (**null**)
byte	**(byte)0**
short	**(short)0**
int	**0**
long	**0L**
float	**0.0f**
double	**0.0d**

Par prudence, il faut remarquer que les valeurs par défaut sont celles que Java garantit quand la variable est utilisée *comme un membre d'une classe*. Ceci assure que les variables membres de type primitif sont toujours initialisées (parfois C++ ne le fait pas), ce qui supprime une source de bugs. Toutefois, cette valeur initiale peut ne pas être correcte ou même légale pour le programme qui est écrit. Il est préférable de toujours initialiser explicitement les variables.

Cette garantie ne s'applique pas aux variables « locales » -- celles qui ne sont pas des champs d'une classe. Ainsi, si dans la définition d'une fonction, on a :

```
int x;
```

Alors **x** aura une valeur arbitraire (comme en C et C++), il ne sera pas initialisé automatiquement à zéro. On a la responsabilité d'affecter une valeur appropriée avant d'utiliser **x**. Si on oublie de le faire, Java est sans aucun doute mieux conçu que C++ sur ce point : on obtient une erreur de compilation qui dit que la variable pourrait ne pas être initialisée (avec beaucoup de compilateurs C ++ on a des avertissements concernant les variables non initialisées, mais avec Java ce sont des erreurs).

Méthodes, paramètres et valeurs de retour

Jusqu'à présent le terme *fonction* a été employé pour désigner une sous-routine nommée. Le terme qui est plus généralement employé en Java est *méthode*, en tant que « moyen de faire quelque chose ». Il est possible, si on le souhaite, de continuer à raisonner en terme de fonctions. Il s'agit simplement d'une différence de syntaxe, mais à partir de maintenant on utilisera « méthode » plutôt que fonction, dans ce livre.

Les méthodes en Java définissent les messages qu'un objet peut recevoir. Dans cette partie on verra à quel point il est simple de définir une méthode.

Les éléments fondamentaux d'une méthode sont le nom, les paramètres, le type de retour et le corps. Voici la forme de base :

```
returnType methodName( /* liste de paramètres */ ) {
  /* corps de la méthode */
}
```

Le type de retour est le type de la valeur qui est retournée par la méthode après son appel. La liste de paramètres donne le type et le nom des informations qu'on souhaite passer à la méthode. L'association du nom de la méthode et de la liste de paramètres identifie de façon unique la méthode.

En Java, les méthodes ne peuvent être créées que comme une composante d'une classe. Une méthode ne peut être appelée que pour un objet [22] et cet objet doit être capable de réaliser cet appel de méthode. Si on essaye d'appeler une mauvaise méthode pour un objet, on obtient un message d'erreur lors de la compilation. On appelle une méthode pour un objet en nommant l'objet suivi d'un point, suivi du nom de la méthode et de sa liste d'arguments, comme ça : **objectName.methodName(arg1, arg2, arg3)**. Par exemple, si on suppose qu'on a une méthode **f( )** qui ne prend aucun paramètre et qui retourne une valeur de type **int**. Alors, si on a un objet appelé **a** pour lequel **f( )** peut être appelé, on peut écrire :

```
int x = a.f();
```

Le type de la valeur de retour doit être compatible avec le type de **x**.

On appelle généralement *envoyer un message à un objet* cet acte d'appeler une méthode. Dans l'exemple précédent le message est **f()** et l'objet est **a**. La programmation orientée objet est souvent simplement ramenée à « envoyer des messages à des objets ».

La liste de paramètres

La liste de paramètres de la méthode spécifie quelles informations on passe à la méthode. Comme on peut le supposer, ces informations -- comme tout le reste en Java -- sont sous la forme d'objets. On doit donc indiquer dans la liste de paramètres les types des objets à transmettre et les noms à employer pour chacun. Comme dans toutes les situation où on a l'impression de manipuler des objets, en Java on passe effectivement des références [23]. Toutefois, le type de la référence doit être correct. Si le paramètre est censé être un objet de type **String**, ce qu'on transmet doit être de ce type.

Considérons une méthode qui prend un objet de classe **String** en paramètre. Voici la définition qui doit être mise à l'intérieur de la définition d'une classe pour qu'elle soit compilée :

```
int storage(String s) {
  return s.length() * 2;
}
```

Cette méthode indique combien d'octets sont nécessaires pour contenir une **String** donnée (chaque **char** dans une **String** fait 16 bits, ou deux octets, pour permettre les caractères Unicode). Le paramètre est de type **String** et il est appelé **s**. Une fois que s est passé à une méthode, il peut être traité comme n'importe quel autre objet (on peut lui envoyer des messages). Ici, on appelle la méthode **length( )** qui est une des méthodes de la classe **String** ; elle retourne le nombre de caractères que contient la chaîne.

On peut aussi voir l'utilisation du mot-clef **return** qui fait deux choses. D'abord, il signifie

« quitte la méthode, j'ai terminé ». Ensuite, si la méthode retourne une valeur, cette valeur est placée juste après la déclaration du **return**. Dans le cas présent, la valeur de retour est produite en évaluant l'expression **s.length( ) * 2**.

On peut retourner des valeurs de tous les types qu'on souhaite, mais si on souhaite ne rien retourner du tout on peut le faire en indiquant que la méthode retourne **void**. Voici quelques exemples :

```
boolean flag() { return true; }
float naturalLogBase() { return 2.718f; }
void nothing() { return; }
void nothing2() {}
```

Quand le type de retour est **void**, alors le mot-clef **return** n'est utilisé que pour sortir de la méthode, il n'est donc pas nécessaire quand on atteint la fin de la méthode. On peut retourner d'une méthode à n'importe quel endroit mais si on a indiqué un type de retour qui n'est pas **void** alors le compilateur imposera (avec des messages d'erreur) un retour avec une valeur d'un type approprié sans tenir compte de l'endroit auquel le retour se produit.

À ce point, on peut penser qu'un programme n'est qu'un paquet d'objets avec des méthodes qui prennent d'autres objets en paramètres pour transmettre des messages à ces autres objets. C'est effectivement l'essentiel de ce qui se passe mais dans les chapitres suivants on verra comment faire le travail de bas niveau en prenant des décisions au sein d'une méthode. Pour ce chapitre, envoyer des messages est suffisant.

Construction d'un programme Java

Il y a plusieurs autres éléments à comprendre avant de voir le premier programme Java.

Visibilité des noms

Un problème commun à tous les langages de programmation est le contrôle des noms. Si on utilise un nom dans un module du programme et si un autre programmeur utilise le même nom dans un autre module, comment distingue-t-on un nom d'un autre et comment empêche-t-on les « collisions » de noms ? En C c'est un problème particulier car un programme est souvent un océan de noms incontrôlable. Les classes C++ (sur lesquelles les classes Java sont basées) imbriquent les fonctions dans les classes de telle sorte qu'elles ne peuvent pas entrer en collision avec les noms de fonctions imbriqués dans d'autres classes. Toutefois, C++ autorise toujours les données et les fonctions globales, donc les collisions sont toujours possibles. Pour résoudre ce problème, C++ a introduit les *domaines de noms* (namespace) en utilisant des mots-clefs supplémentaires.

Java a pu éviter tout cela en employant une approche originale. Pour générer sans ambiguïté un nom pour une bibliothèque, le spécificateur utilisé n'est pas très différent d'un nom de domaine Internet. En fait, les créateurs de Java veulent qu'on utilise son propre nom de domaine Internet inversé, puisqu'on est assuré que ceux-ci sont uniques. Puisque mon nom de domaine est **BruceEckel.com**, ma bibliothèque d'utilitaires (utility) pour mes marottes (foibles) devrait être appelée **com.bruceeckel.utility.foibles**. Après avoir inversé le nom de domaine, les points sont destinés à représenter des sous-répertoires.

Dans Java 1.0 et Java 1.1 les extensions de domaines com, edu, org, net, etc. étaient mises en lettres capitales par convention, ainsi la bibliothèque serait : **COM.bruceeckel.utility.foibles**. Tou-

tefois, au cours du développement de Java 2, on s'est rendu compte que cela causait des problèmes et par conséquent les noms de packages sont entièrement en lettres minuscules.

Ce mécanisme signifie que tous les fichiers existent automatiquement dans leur propre domaine de nom et toutes les classe contenues dans un fichier donné doivent avoir un identificateur unique. Ainsi, on n'a pas besoin d'apprendre de particularités spécifiques au langage pour résoudre ce problème -- le langage s'en occupe à votre place.

Utilisation d'autres composantes

Lorsqu'on souhaite utiliser une classe prédéfinie dans un programme, le compilateur doit savoir comment la localiser. Bien entendu, la classe pourrait déjà exister dans le même fichier source que celui d'où elle est appelée. Dans ce cas on utilise simplement la classe -- même si la classe n'est définie que plus tard dans le fichier. Java élimine le problème des « référence anticipées », il n'y a donc pas à s'en préoccuper.

Qu'en est-il des classes qui existent dans un autre fichier ? On pourrait penser que le compilateur devrait être suffisamment intelligent pour aller simplement la chercher lui même, mais il y a un problème. Imaginons que l'on veuille utiliser une classe ayant un nom spécifique mais qu'il existe plus d'une classe ayant cette définition (il s'agit probablement de définitions différentes). Ou pire, imaginons que l'on écrive un programme et qu'en le créant on ajoute à sa bibliothèque une nouvelle classe qui entre en conflit avec le nom d'une classe déjà existante.

Pour résoudre ce problème il faut éliminer les ambiguïtés potentielles. Ceci est réalisé en disant exactement au compilateur Java quelles classes on souhaite, en utilisant le mot-clef **import**. **import** dit au compilateur d'introduire un *package* qui est une bibliothèque de classes (dans d'autres langages, une bibliothèque pourrait comporter des fonctions et des données au même titre que des classes mais il faut se rappeler que tout le code Java doit être écrit dans des classes).

La plupart du temps on utilise des composantes des bibliothèques Java standard qui sont fournies avec le compilateur. Avec celles-ci il n'y a pas à se tracasser à propos des longs noms de domaines inversés ; il suffit de dire, par exemple :

```
import java.util.ArrayList;
```

pour dire au compilateur que l'on veut utiliser la classe Java **ArrayList**. Toutefois, **util** contient de nombreuses classes et on pourrait vouloir utiliser plusieurs d'entre elles sans les déclarer explicitement. Ceci est facilement réalisé en utilisant '*' pour indiquer un joker :

```
import java.util.*;
```

Il est plus courant d'importer une collection de classes de cette manière que d'importer les classes individuellement.

Le mot-clef static

Normalement, quand on crée une classe, on décrit ce à quoi ressemblent les objets de cette classe et comment ils se comportent. Rien n'existe réellement avant de créer un objet de cette classe avec **new** ; à ce moment la zone de données est créée et les méthodes deviennent disponibles.

Mais il y a deux situation pour lesquelles cette approche n'est pas suffisante. L'une, si on veut avoir une zone de stockage pour des données spécifiques, sans tenir compte du nombre d'objets

créés, ou même si aucun objet n'a été créé. L'autre, si on a besoin d'une méthode qui n'est associée à aucun objet particulier de la classe. C'est à dire si on a besoin d'une méthode qui puisse être appelée même si aucun objet n'a été créé. On peut obtenir ces deux effets avec le mot-clef **static**. Dire que quelque chose est **static** signifie que la donnée ou la méthode n'est pas spécifiquement rattachée à un objet instance de cette classe. Donc, même si aucun objet de cette classe n'a jamais été créé il est possible d'appeler une méthode **static** ou d'accéder à une donnée **static**. Avec des données et des méthodes non **static** ordinaires il faut connaître l'objet spécifique avec lequel elles fonctionnent. Bien entendu, étant donné que les méthodes **static** n'ont pas besoin qu'un objet soit créé avant d'être utilisées, elles ne peuvent pas accéder *directement* à des membres ou des méthodes non **static** en appelant ces autres membres sans faire référence à un objet nommé (puisque les membres et méthodes non **static** doivent être rattachés à un objet spécifique).

Certains langages orientés objet emploient les expressions *données de classe* et *méthodes de classe*, ce qui signifie que les données et les méthodes n'existent que pour la classe en tant que tout et pas pour des objets particuliers de la classe. Parfois la littérature Java utilise aussi ces expressions.

Pour rendre statique une méthode ou une donnée membre il suffit de mettre le mot-clef **static** avant la définition. Par exemple, le code suivant crée une donnée membre **static** et l'initialise :

```
class StaticTest {
   static int i = 47;
}
```

Maintenant, même en créant deux objet **StaticTest**, il n'y aura qu'une seule zone de stockage pour **StaticTest.i**. Tous les objets partageront le même i. Considérons :

```
StaticTest st1 = new StaticTest();
StaticTest st2 = new StaticTest();
```

A ce point, st1.i et st2.i ont la même valeur 47 puisqu'elles font référence à la même zone mémoire.

Il y a deux façons de faire référence à une variable **static**. Comme indiqué ci-dessus, il est possible de la nommer via un objet, en disant par exemple st2.i. Il est aussi possible d'y faire référence directement par le nom de la classe, ce qui ne peut pas être fait avec un membre non **static** (c'est le moyen de prédilection pour faire référence à une variable **static** puisque cela met en évidence la nature **static** de la variable).

```
StaticTest.i++;
```

L'opérateur ++ incrémente la variable. À ce point, st1.i et st2.i auront tous deux la valeur 48.

Une logique similaire s'applique aux méthodes statiques. On peut faire référence à une méthode statique soit par l'intermédiaire d'un objet, comme on peut le faire avec n'importe quelle méthode, ou avec la syntaxe spécifique supplémentaire **ClassName.method( )**. Une méthode statique est définie de façon similaire :

```
class StaticFun {
  static void incr() { StaticTest.i++; }
}
```

On peut voir que la méthode **incr( )** de **StaticFun** incrémente la donnée **static i**. On peut appeler **incr( )** de façon classique, par le biais d'un objet :

```
StaticFun sf = new StaticFun();
sf.incr();
```

Ou, parce que **incr( )** est une méthode statique, il est possible de l'appeler directement par sa classe :

```
StaticFun.incr();
```

Alors que **static**, lorsqu'il est appliqué à une donnée membre, change sans aucun doute la façon dont la donnée est créée (une pour chaque classe par opposition à une pour chaque objet dans le cas des données non statiques), lorsqu'il est appliqué à une méthode, le changement est moins significatif. Un cas important d'utilisation des méthodes **static** est de permettre d'appeler cette méthode sans créer d'objet. C'est essentiel, comme nous le verrons, dans la définition de la méthode **main( )** qui est le point d'entrée pour exécuter une application.

Comme pour toute méthode, une méthode statique peut créer ou utiliser des objets nommés de son type, ainsi les méthodes statiques sont souvent utilisées comme « berger » pour un troupeau d'instances de son propre type.

Votre premier programme Java

Voici enfin notre premier programme . Il commence par écrire une chaîne de caractères, puis il écrit la date en utilisant la classe **Date** de la bibliothèque standard de Java. Il faut remarquer qu'un style de commentaire supplémentaire est introduit ici : le '//' qui est un commentaire jusqu'à la fin de la ligne.

```
// HelloDate.java
import java.util.*;
public class HelloDate {
  public static void main(String[] args) {
    System.out.println("Hello, it's: ");
    System.out.println(new Date());
  }
}
```

Si on retourne au début pour sélectionner **java.lang** puis **System**, on voit que la classe **System** a plusieurs champs et si on sélectionne **out** on découvre que c'est un objet **static PrintStream**. Puisqu'il est statique, on n'a pas besoin de créer quoique ce soit. L'objet **out** est toujours là et il suffit de l'utiliser. Ce qu'on peut faire avec **out** est défini par son type : **PrintStream**. D'une façon très pratique, **PrintStream** est affiché dans la description comme un hyper lien, ainsi, en cliquant dessus on voit la liste de toutes les méthodes qu'on peut appeler pour **PrintStream**. Il y en a un certain nombre et elles seront décrites ultérieurement dans ce livre. Pour l'instant nous ne nous intéressons qu'à **println( )**, qui veut dire « écrit ce que je te donne sur la console et passe à la ligne ». Ainsi, dans tout programme Java on peut dire **System.out.println("quelque chose")** chaque fois qu'on souhaite écrire quelque chose sur la console.

Le nom de la classe est le même que celui du fichier. Quand on crée un programme autonome comme celui-là une des classes du fichier doit avoir le même nom que le fichier (le compilateur se

plaint si on ne le fait pas). Cette classe doit contenir une méthode appelée **main( )** avec la signature suivante :

```
public static void main(String[] args);
```

Le mot-clef **public** signifie que la méthode est accessible au monde extérieur (décrit en détail dans le chapitre 5). Le paramètre de **main( )** est un tableau d'objets de type **String**. Le paramètre **args** n'est pas utilisé dans ce programme mais le compilateur Java insiste pour qu'il soit là car il contient les paramètres invoqués sur la ligne de commande.

La ligne qui écrit la date est assez intéressante :

```
System.out.println(new Date());
```

Considérons le paramètre : un objet **Date** est créé juste pour transmettre sa valeur à **println( )**. Dès que cette instruction est terminée, cette date est inutile et le ramasse-miettes peut venir le récupérer n'importe quand. On n'a pas à s'occuper de s'en débarrasser.

Compilation et exécution

Pour compiler et exécuter ce programme, et tous les autres programmes de ce livre, il faut d'abord avoir un environnement de programmation Java. Il y a un grand nombre d'environnements de développement mais dans ce livre nous supposerons que vous utilisez le JDK de Sun qui est gratuit. Si vous utilisez un autre système de développement, vous devrez vous reporter à la documentation de ce système pour savoir comment compiler et exécuter les programmes.

Connectez vous à Internet et allez sur *http://java.sun.com*. Là, vous trouverez des informations et des liens qui vous guideront pour télécharger et installer le JDK pour votre plate-forme.

Une fois que le JDK est installé et que vous avez configuré les informations relatives au chemin sur votre ordinateur afin qu'il puisse trouver **javac** et **java**, téléchargez et décompressez le code source de ce livre (on peut le trouver sur le CD-ROM qui est livré avec le livre ou sur *www.BruceEckel.com*). Ceci créera un sous répertoire pour chacun des chapitres de ce livre. Allez dans le sous-répertoire **c02** et tapez :

```
javac HelloDate.java
```

Cette commande ne devrait produire aucune réponse. Si vous obtenez un message d'erreur de quelque sorte que ce soit cela signifie que vous n'avez pas installé le JDK correctement et que vous devez corriger le problème.

Par contre, si vous obtenez simplement votre invite de commande vous pouvez taper :

```
java HelloDate
```

et vous obtiendrez en sortie le message ainsi que la date.

C'est le procédé que vous pouvez employer pour compiler et exécuter chacun des programmes de ce livre. Toutefois, vous verrez que le code source de ce livre a aussi un fichier appelé **makefile** dans chaque chapitre, et celui-ci contient les commandes « make » pour construire automatiquement les fichiers de ce chapitre. Reportez-vous à la page Web de ce livre sur www.BruceEckel.com pour plus de détails sur la manière d'utiliser ces *makefiles*.

Commentaires et documentation intégrée

Il y a deux types de commentaires en Java. Le premier est le commentaire traditionnel, style C, dont a hérité C++. Ces commentaires commencent par /* et continuent, éventuellement sur plusieurs lignes, jusqu'à un */. Il faut remarquer que de nombreux programmeurs commencent chaque ligne de continuation de commentaire avec *, on voit donc souvent :

```
/* Ceci est un commentaire
* qui continue
* sur plusieurs lignes
*/
```

Il faut toutefois se rappeler que tout ce qui se trouve entre /* et */ est ignoré, il n'y a donc aucune différence avec :

```
/* Ceci est un commentaire qui
  continue sur plusieurs lignes */
```

La seconde forme de commentaires vient du C++. C'est le commentaire sur une seule ligne qui commence avec // et continue jusqu'à la fin de la ligne. Ce type de commentaire est pratique et souvent rencontré car il est simple. Il n'y a pas à se démener sur le clavier pour trouver / puis * (à la place il suffit d'appuyer deux fois sur la même touche) et il n'est pas nécessaire de fermer le commentaire. On trouve donc fréquemment :

```
// Ceci est un commentaire sur une seule ligne
```

Commentaires de documentation

Un des plus solides éléments de Java est que les concepteurs n'ont pas considéré que l'écriture du code est la seule activité importante -- ils ont aussi pensé à sa documentation. Le plus gros problème avec la documentation de code est probablement la maintenance de cette documentation. Si la documentation et le code sont séparés, ça devient embêtant de changer la documentation chaque fois que le code change. La solution a l'air simple : relier le code et la documentation. Le moyen le plus simple de le faire est de tout mettre dans le même fichier. Toutefois, pour compléter le tableau il faut une syntaxe de commentaire particulière pour marquer la documentation particulière et un outil pour extraire ces commentaires et les mettre sous une forme exploitable. C'est ce que Java a fait.

L'outil pour extraire les commentaires est appelé *javadoc*. Il utilise certaines technologies du compilateur Java pour rechercher les marqueurs spécifiques des commentaires qui ont été mis dans les programmes. Il ne se contente pas d'extraire les informations repérées par ces marqueurs, mais il extrait aussi le nom de classe ou le nom de méthode adjoint au commentaire. Ainsi on parvient avec un travail minimal à générer une documentation de programme convenable.

La sortie de javadoc est un fichier HTML qui peut être visualisé avec un browser Web. Cet outil permet de créer et maintenir un unique fichier source et à générer automatiquement une documentation utile. Grâce à javadoc on a un standard pour créer la documentation et il est suffisamment simple pour qu'on puisse espérer ou même exiger une documentation avec toute bibliothèque Java.

Syntaxe

Toutes les commandes javadoc n'apparaissent que dans les commentaires /**. Les commentaires finissent avec */ comme d'habitude. Il y a deux principales façons d'utiliser javadoc : du HTML intégré ou des « onglets doc ». Les onglets doc sont des commandes qui commencent avec un '@' et qui sont placées au début d'une ligne de commentaire (toutefois, un '*' en tête est ignoré).

Il y a trois types de commentaires de documentation qui correspondent aux éléments suivant le commentaire : classe, variable ou méthode. C'est à dire qu'un commentaire de classe apparaît juste avant la définition de la classe, un commentaire de variable apparaît juste avant la définition de la variable et un commentaire de méthode apparaît juste avant la définition de la méthode. Voici un exemple simple :

```
/** Un commentaire de classe */
public class docTest {
  /** Un commentaire de variable */
  public int i;
  /** Un commentaire de méthode */
  public void f() {}
}
```

Il faut noter que javadoc ne traite les commentaires de documentation que pour les membres **public** et **protected**. Les commentaires pour les membres **private** et « amis » (voir Chapitre 5) sont ignorés et on ne verra aucune sortie (toutefois on peut utiliser le flag **private** pour inclure aussi les membres **private**). Ceci est sensé puisque seuls les membres **public** et **protected** sont accessibles en dehors du fichier, ce qui est la perspective du client du programmeur. Toutefois, tous les commentaires de classe sont inclus dans le fichier de sortie.

La sortie du code précédent est un fichier HTML qui a le même format standard que tout le reste de la documentation Java, ainsi les utilisateurs seront à l'aise avec le format et pourront facilement naviguer dans les classes. Ça vaut la peine de taper le code précédent, de le passer dans javadoc et d'étudier le fichier HTML résultant pour voir le résultat.

HTML intégré

Javadoc passe les commandes HTML dans les documents HTML générés. Ceci permet une utilisation complète de HTML, la motivation principale étant de permettre le formatage du code comme suit :

```
/**
* <pre>
* System.out.println(new Date());
* </pre>
*/
```

On peut aussi utiliser HTML comme on pourrait le faire dans n'importe quel autre document Web pour formater du texte courant dans les descriptions :

```
/**
* On peut <em>même</em> insérer une liste :
* <ol>
```

```
* <li> élément un
* <li> élément deux
* <li> élément trois
* </ol>
*/
```

Il faut noter que dans les commentaires de documentation, les astérisques en début de ligne sont éliminés par javadoc, ainsi que les espaces en tête de ligne. Javadoc reformate tout pour assurer la conformité avec l'apparence des documentations standard. Il ne faut pas utiliser des titres tels que **<h1>** ou **<hr>** dans le HTML intégré car javadoc insère ses propres titres et il y aurait des interférences.

Tous les types de commentaires de documentation -- classes, variables et méthodes -- acceptent l'intégration de HTML.

@see : faire référence aux autres classes

Les trois types de commentaires de documentation (classes, variables et méthodes) peuvent contenir des onglets **@see**, qui permettent de faire référence à de la documentation dans d'autres classes. Javadoc génèrera du HTML où les onglets **@see** seront des hyper liens vers d'autres documentations. Les différentes formes sont :

```
@see classname
@see fully-qualified-classname
@see fully-qualified-classname#method-name
```

Chacune d'entre elles ajoute un hyper lien de type « Voir aussi » à la documentation générée. Javadoc ne vérifie pas si l'hyper lien indiqué est valide.

Class documentation tags

En plus du HTML intégré et des références **@see**, les documentations de classe peuvent inclure des onglets pour les informations de version et pour le nom de l'auteur. Les documentations de classe peuvent aussi être utilisées pour les *interfaces* (voir Chapitre 8).

@version

Voici le format :

```
@version version-information
```

dans lequel **version-information** est n'importe quelle information significative que l'on souhaite inclure. Quand le flag **-version** est mis sur la ligne de commande de javadoc, les informations de version seront exploitées, particulièrement dans la documentation HTML.

@author

Voici le format :

```
@author author-information
```

dans lequel **author-information** est, a priori, votre nom mais il peut aussi contenir une adresse email ou toute autre information appropriée. Quand le flag **-author** est mis sur la ligne de commande javadoc, les informations sur l'auteur seront exploitées, particulièrement dans la documentation HTML.

On peut avoir plusieurs onglets d'auteur pour une liste d'auteurs mais ils doivent être placés consécutivement. Toutes les informations d'auteurs seront regroupées dans un unique paragraphe dans le code HTML généré.

@since

Cet onglet permet d'indiquer la version du code qui a commencé à utiliser une caractéristique particulière. On la verra apparaître dans la documentation HTML de Java pour indiquer quelle version de JDK est utilisée.

Les onglets de documentation de variables

Les documentations de variables ne peuvent contenir que du HTML intégré et des références **@see**.

Les onglets de documentation de méthodes

En plus du HTML intégré et des références **@see**, les méthodes acceptent des onglets de documentation pour les paramètres, les valeurs de retour et les exceptions.

@param

Voici le format :

```
@param parameter-name description
```

dans lequel **parameter-name** est l'identificateur dans la liste de paramètres et **description** est du texte qui peut se prolonger sur les lignes suivantes. La description est considérée comme terminée quand un nouvel onglet de documentation est trouvé. On peut en avoir autant qu'on veut, a priori un pour chaque paramètre.

@return

Voici le format :

```
@return description
```

dans lequel **description** indique la signification de la valeur de retour. Le texte peut se prolonger sur les lignes suivantes.

@throws

Les exceptions seront introduites dans le Chapitre 10 mais, brièvement, ce sont des objets qui peuvent être émis (throw) par une méthode si cette méthode échoue. Bien qu'une seule exception puisse surgir lors de l'appel d'une méthode, une méthode donnée est susceptible de produire

n'importe quel nombre d'exceptions de types différents, chacune d'entre elles nécessitant une description. Ainsi, le format de l'onglet d'exception est :

```
@throws fully-qualified-class-name description
```

dans lequel **fully-qualified-class-name** indique sans ambiguïté un nom de classe d'exception qui est définie quelque part, et **description** (qui peut se prolonger sur les lignes suivantes) précise pourquoi ce type particulier d'exception peut survenir lors de l'appel de la méthode.

@deprecated

Ceci est utilisé pour marquer des fonctionnalités qui ont été remplacées par d'autres qui sont meilleures. L'onglet **deprecated** suggère de ne plus utiliser cette fonctionnalité particulière étant donné qu'elle sera probablement supprimée ultérieurement. Une méthode marquée **@deprecated** fait produire un warning par le compilateur si elle est utilisée.

Exemple de documentation

Voici à nouveau le premier programme Java, cette fois après avoir ajouté les commentaires de documentation :

```
//: c02:HelloDate.java
import java.util.*;
/** Le premier exemple de programme de Thinking in Java.
* Affiche une chaîne de caractères et la date du jour.
* @author Bruce Eckel
* @author http://www.BruceEckel.com
* @version 2.0
*/
public class HelloDate {
  /** Unique point d'entrée de la classe et de l'application
   * @param args tableau de paramètres sous forme de chaînes de caractères
   * @return Pas de valeur de retour
   * @exception exceptions Pas d'exceptions émises
  */
  public static void main(String[] args) {
    System.out.println("Hello, it's: ");
    System.out.println(new Date());
  }
} ///:~
```

La première ligne du fichier utilise une technique personnelle qui consiste à mettre un ':' comme marqueur spécifique pour la ligne de commentaire contenant le nom du fichier source. Cette ligne contient le chemin du fichier (dans ce cas, c02 indique le Chapitre 2) suivi du nom de fichier [25]. La dernière ligne finit aussi avec un commentaire et celui-ci indique la fin du listing du code source, ce qui permet de l'extraire automatiquement du texte de ce livre et de le contrôler avec un compilateur.

Style de programmation

Le standard non officiel de Java consiste à mettre en majuscule la première lettre des noms de classes. Si le nom de classe est composé de plusieurs mots, ils sont accolés (c'est à dire qu'on ne sépare pas les noms avec un trait bas) et la première lettre de chaque mot est mise en majuscule ainsi :

```
class AllTheColorsOfTheRainbow { // ...
```

Pour pratiquement tout les reste : méthodes, champs (variables membres) et les noms des références d'objets, le style retenu est comme pour les classes *sauf* que la première lettre de l'identificateur est une minuscule. Par exemple :

```
class AllTheColorsOfTheRainbow {
  int anIntegerRepresentingColors;
  void changeTheHueOfTheColor(int newHue) {
    // ...
  }
  // ...
}
```

Bien entendu il faut se rappeler que l'utilisateur doit aussi taper tous ces longs noms, donc soyez clément.

Le code Java qu'on voit dans les bibliothèques de Sun respecte aussi le placement des accolades ouvrantes et fermantes qui est utilisé dans ce livre.

Dans ce chapitre on en a vu suffisamment sur la programmation Java pour comprendre comment écrire un programme simple et on a eu une vue d'ensemble du langage et de certaines des idées de base. Toutefois, les exemples vus jusqu'à présent ont tous été du type « faire ceci, puis faire cela, puis faire autre chose ». Qu'en advient-il si on veut faire un programme pour réaliser des choix, comme dans « si le résultat de ceci est rouge, faire cela, sinon faire autre chose » ? Les outils disponibles en Java pour cette activité de programmation fondamentale seront vus dans le prochain chapitre.

Exercices

1. En suivant l'exemple **HelloDate.java** de ce chapitre, créez un programme « hello, world » qui affiche simplement cette phrase. Vous n'avez besoin que d'une seule méthode dans la classe (la méthode « main » qui est exécutée quand le programme démarre). Pensez à la rendre **static** et à indiquer la liste de paramètres, même si la liste de paramètres n'est pas utilisée. Compilez ce programme avec **javac**. Si vous utilisez un environnement de développement autre que JDK, apprenez à compiler et exécuter les programmes dans cet environnement.

2. Trouver le morceau de code concernant **ATypeName** et faites-en un programme qui compile et s'exécute.

3. Transformez le morceau de code DataOnly en un programme qui compile et qui s'exécute.

4. Modifiez l'exercice 3 de telle sorte que les valeurs des données dans **DataOnly** soient affectées et affichées dans **main( )**.

5. Écrivez un programme qui inclus et appelle la méthode **storage( )** définie comme morceau de code dans ce chapitre.

6. Transformez le morceau de code **StaticFun** en un programme qui fonctionne.

7. Écrivez un programme qui imprime trois paramètres saisis sur la ligne de commande. Pour faire ceci il faut indexer le tableau de **String** représentant la ligne de commande.

8. Transformez l'exemple **AllTheColorsOfTheRainbow** en un programme qui compile et s'exécute.

9. Trouvez le code de la seconde version de **HelloDate.java** qui est le petit exemple de commentaire de documentation. Exécutez javadoc sur le fichier et visualisez le résultat avec votre browser Web.

10. Transformez docTest en un fichier qui compile et exécutez javadoc dessus. Contrôlez la documentation qui en résulte avec votre browser Web.

11. Ajoutez une liste d'éléments en HTML, à la documentation de l'exercice 10.

12. Prenez le programme de l'exercice 1 et ajoutez lui une documentation. Sortez cette documentation dans un fichier HTML à l'aide de javadoc et visualisez la avec votre browser Web.

[21] Ceci peut être une poudrière. Il y a ceux qui disent « c'est clairement un pointeur », mais ceci suppose une implémentation sous-jacente. De plus, les références Java sont plutôt apparentées aux références C++ qu'aux pointeurs dans leur syntaxe. Dans la première édition de ce livre, j'ai choisi d'inventer un nouveau terme, « manipulateur » (handle), car les références C++ et les références Java ont des différences importantes. Je venais du C++ et je ne voulais pas embrouiller les programmeurs C++ que je supposais être le principal public de Java. Dans la seconde édition, j'ai décidé que « référence » était le terme le plus généralement utilisé et que tous ceux qui venaient du C++ auraient à s'occuper de bien d'autres choses que de la terminologie de référence, donc qu'ils pourraient aussi bien y plonger les deux pieds en avant. Toutefois, il y a des gens qui ne sont pas d'accord, même avec le terme « référence ». J'ai lu dans un livre qu'il était « complètement faux de dire que Java supportait les passages par références », parce que les identificateurs des objets Java (selon cet auteur) sont *en fait* des « références à des objets ». Et (continue-t-il) tout est *en fait* passé par valeur. Donc on ne passe pas une référence, on « passe une référence à un objet, par valeur ». On pourrait discuter de la précision d'une explication si alambiquée mais je pense que mon approche simplifie la compréhension du concept sans blesser qui que ce soit (d'accord, les avocats du langage pourraient prétendre que je mens, mais je répondrai que je fournis une abstraction appropriée).

[22] Les méthodes **static**, que nous découvrirons ultérieurement, peuvent être appelées *pour la classe*, sans passer par un objet.

[23] Avec les exceptions habituelles pour les types de données précités **boolean, char**, **byte**, **short**, **int**, **long**, **float,** et **double**. En général, toutefois, on passe des objets, ce qui veut dire qu'en réalité on passe des références à des objets.

[24] Certains environnements de programmations feront apparaître brièvement le programme à l'écran et le fermeront avant d'avoir eu une chance de voir le résultat. On peut mettre le morceau de code suivant à la fin du **main( )** pour obtenir une pause sur la sortie :

```
try {
```

```
            System.in.read();
        } catch(Exception e) {}
```

Cette pause durera jusqu'à ce qu'on appuie sur « Entrée » (ou toute autre touche). Ce code met en jeu des concepts qui ne seront introduits que bien plus tard dans ce livre, donc vous ne le comprendrez pas d'ici là, mais il fera l'affaire.

[25] Un outil que j'ai créé avec Python (voir www.Python.org) utilise cette information pour extraire les fichiers sources, les mettre dans les sous-répertoires appropriés et créer les makefiles.

Chapitre 3 - Contrôle du flux du programme

Tout comme une créature sensible, un programme doit agir sur son environnement et faire des choix durant sa vie.

En Java les objets et les données sont manipulés au moyen d'opérateurs, et les choix sont faits au moyen des instructions de contrôle d'exécution. Java a hérité de C++, c'est pourquoi beaucoup d'instructions seront familières aux programmeurs C et C++. Java a également amené quelques améliorations et aussi quelques simplifications.

Si vous avez l'impression de patauger quelque peu dans ce chapitre, voyez le CD ROM multimédia fourni avec le livre : *Thinking in C : Foundations for Java and C++*. Il contient des cours audio, des diapositives, des exercices, et des solutions, le tout spécialement conçu pour vous familiariser rapidement avec la syntaxe C nécessaire pour apprendre Java.

Utilisation des opérateurs Java

Un opérateur agit sur un ou plusieurs arguments pour produire une nouvelle valeur. Les arguments se présentent sous une forme différente de celle d'un appel de méthode standard, mais le résultat est le même. Votre expérience de programmation antérieure a dû vous familiariser avec les concepts généraux des opérateurs. L'addition (+), la soustraction et le moins unaire (-), la multiplication (*), la division (/), et l'affectation (=) fonctionnent de la même manière dans tous les langages de programmation.

Tous les opérateurs produisent une valeur à partir de leurs opérandes. En outre, un opérateur peut changer la valeur de l'un de ses opérandes. C'est ce qu'on appelle un *effet de bord*. L'utilisation la plus fréquente des opérateurs modifiant leurs opérandes est justement de générer un effet de bord, mais dans ce cas il faut garder à l'esprit que la valeur produite est disponible tout comme si on avait utilisé l'opérateur sans chercher à utiliser son effet de bord.

Presque tous les opérateurs travaillent uniquement avec les types primitifs. Les exceptions sont « = », « == » et « **!=** », qui fonctionnent avec tous les objets (ce qui est parfois déroutant lorsqu'on traite des objets). De plus, la classe **String** admet les opérateurs « + » et « += ».

Priorité

La priorité des opérateurs régit la manière d'évaluer une expression comportant plusieurs opérateurs. Java a des règles spécifiques qui déterminent l'ordre d'évaluation. La règle la plus simple est que la multiplication et la division passent avant l'addition et la soustraction. Souvent les programmeurs oublient les autres règles de priorité, aussi vaut-il mieux utiliser les parenthèses afin que l'ordre d'évaluation soit explicite. Par exemple :

```
A = X + Y - 2/2 + Z;
```

a une signification différente de la même instruction dans laquelle certains termes sont groupés entre parenthèses :

```
A = X + (Y - 2)/(2 + Z);
```

L'affectation

L'affectation est réalisée au moyen de l'opérateur « = ». Elle signifie « prendre la valeur se trouvant du côté droit (souvent appelée *rvalue*) et la copier du côté gauche (souvent appelée *lvalue*) ». Une *rvalue* représente toute constante, variable ou expression capable de produire une valeur, mais une *lvalue* doit être une variable distincte et nommée (autrement dit, il existe un emplacement physique pour ranger le résultat). Par exemple, on peut affecter une valeur constante à une variable (**A = 4;**), mais on ne peut pas affecter quoi que ce soit à une valeur constante - elle ne peut pas être une *lvalue* (on ne peut pas écrire **4 =A;**).

L'affectation des types primitifs est très simple. Puisque les données de type primitif contiennent une valeur réelle et non une référence à un objet, en affectant une valeur à une variable de type primitif on copie le contenu d'un endroit à un autre. Par exemple, si on écrit **A = B** pour des types primitifs, alors le contenu de **B** est copié dans **A**. Si alors on modifie **A**, bien entendu **B** n'est pas affecté par cette modification. C'est ce qu'on rencontre généralement en programmation.

Toutefois, les choses se passent différemment lorsqu'on affecte des objets. Quand on manipule un objet, on manipule en fait sa référence, ce qui fait que lorsqu'on effectue une affectation « depuis un objet vers un autre », en réalité on copie une référence d'un endroit à un autre. En d'autres termes, si on écrit **C = D** pour des objets, après l'exécution **C** et **D** pointeront tous deux vers l'objet qui, à l'origine, était pointé uniquement par **D**. L'exemple suivant démontre cela.

Voici l'exemple :

```
//: c03:Assignment.java
// l'affectation avec des objets n'est pas triviale.

class Number {
  int i;
}

public class Assignment {
  public static void main(String[] args) {
    Number n1 = new Number();
    Number n2 = new Number();
    n1.i = 9;
    n2.i = 47;
    System.out.println("1: n1.i: " + n1.i +
      ", n2.i: " + n2.i);
    n1 = n2;
    System.out.println("2: n1.i: " + n1.i +
      ", n2.i: " + n2.i);
    n1.i = 27;
    System.out.println("3: n1.i: " + n1.i +
      ", n2.i: " + n2.i);
  }
} ///:~
```

La classe **Number** est simple, **main( )** en crée deux instances (**n1** Etant **n2**). La valeur **i** de chaque **Number** est initialisée différemment, puis **n2** est affecté à **n1**. Dans beaucoup de langages de programmation on s'attendrait à ce que **n1** et **n2** restent toujours indépendants, mais voici le résultat de ce programme, dû au fait qu'on a affecté une référence :

```
1: n1.i: 9, n2.i: 47
2: n1.i: 47, n2.i: 47
3: n1.i: 27, n2.i: 27
```

Si on modifie l'objet **n1**, l'objet **n2** est lui aussi modifié ! Ceci parce que **n1** et **n2** contiennent une même référence pointant vers le même objet. (la référence originale qui se trouvait dans **n1** et qui pointait sur un objet contenant la valeur 9 a été écrasée lors de l'affectation et a été perdue ; l'objet sur lequel elle pointait sera nettoyé par le ramasse-miettes).

Ce phénomène est souvent appelé *aliasing* (fausse désignation) et c'est la manière fondamentale de gérer les objets en Java. Bien. Et si on ne veut pas de l'aliasing ? Alors il ne faut pas utiliser l'affectation directe **n1 = n2**, il faut écrire :

```
n1.i = n2.i;
```

Les deux objets restent indépendants plutôt que d'en perdre un et de faire pointer **n1**et **n2** vers le même objet ; mais on s'aperçoit très vite que manipuler les champs des objets ne donne pas un code lisible et va à l'encontre des bons principes de la conception orientée objet. C'est un sujet non trivial, je le laisserai de côté et je le traiterai dans l'annexe A, consacrée à l'aliasing. En attendant, gardons à l'esprit que l'affectation des objets peut entraîner des surprises.

L'aliasing pendant l'appel des méthodes

L'aliasing peut également se produire en passant un objet à une méthode :

```
//: c03:PassObject.java
// Le passage d'objets à une méthodes peut avoir
// un effet différent de celui qu'on espère


class Letter {
  char c;
}

public class PassObject {
  static void f(Letter y) {
    y.c = 'z';
  }
  public static void main(String[] args) {
    Letter x = new Letter();
    x.c = 'a';
    System.out.println("1: x.c: " + x.c);
    f(x);
    System.out.println("2: x.c: " + x.c);
  }
```

```
} ///:~
```

Dans beaucoup de langages de programmation, la méthode **f( )** est censée faire une copie de son argument **Letter y** dans la zone de visibilité de la méthode. Mais, encore une fois, c'est une référence qui est passée et donc la ligne :

```
y.c = 'z';
```

modifie en réalité l'objet se trouvant au-dehors de **f( )**. Voici la sortie :

```
1: x.c: a
2: x.c: z
```

L'*aliasing* et ses conséquences sont un sujet complexe, toutes les réponses à ces questions seront données dans l'Annexe A, mais il vous faut dès maintenant prendre conscience de son existence afin d'en éviter les pièges.

Les opérateurs mathématiques

Les opérateurs mathématiques de base sont les mêmes que ceux qu'on trouve dans beaucoup de langages de programmation : l'addition (+), la soustraction (-), la division (/), la multiplication (*) et le modulo (**%**, le reste de la division entière). La division entière tronque le résultat sans l'arrondir.

Java utilise également une notation abrégée pour effectuer en un seul temps une opération et une affectation. Ceci est compatible avec tous les opérateurs du langage (lorsque cela a un sens), on le note au moyen d'un opérateur suivi d'un signe égal. Par exemple, pour ajouter 4 à la variable **x** et affecter le résultat à **x**, on écrit : **x += 4**.

Cet exemple montre l'utilisation des opérateurs mathématiques :

```
//: c03:MathOps.java
// Démonstration des opérateurs mathématiques.
import java.util.*;

public class MathOps {
  // raccourci pour éviter des frappes de caractères :
  static void prt(String s) {
    System.out.println(s);
  }
  // raccourci pour imprimer une chaîne et un entier :
  static void pInt(String s, int i) {
    prt(s + " = " + i);
  }
  // raccourci pour imprimer une chaîne et un nombre en virgule flottante :
  static void pFlt(String s, float f) {
    prt(s + " = " + f);
  }
  public static void main(String[] args) {
    // Crée un générateur de nombres aléatoires,
```

```
        // initialisé par défaut avec l'heure actuelle :
        Random rand = new Random();
        int i, j, k;
        // '%' limite la valeur maximale à 99 :
        j = rand.nextInt() % 100;
        k = rand.nextInt() % 100;
        pInt("j",j);  pInt("k",k);
        i = j + k; pInt("j + k", i);
        i = j - k; pInt("j - k", i);
        i = k / j; pInt("k / j", i);
        i = k * j; pInt("k * j", i);
        i = k % j; pInt("k % j", i);
        j %= k; pInt("j %= k", j);
        // tests sur les nombres en virgule flottante :
        float u,v,w;  // s'applique aussi aux nombres en double précision
        v = rand.nextFloat();
        w = rand.nextFloat();
        pFlt("v", v); pFlt("w", w);
        u = v + w; pFlt("v + w", u);
        u = v - w; pFlt("v - w", u);
        u = v * w; pFlt("v * w", u);
        u = v / w; pFlt("v / w", u);
        // ce qui suit fonctionne également avec les types
        // char, byte, short, int, long et double :
        u += v; pFlt("u += v", u);
        u -= v; pFlt("u -= v", u);
        u *= v; pFlt("u *= v", u);
        u /= v; pFlt("u /= v", u);
      }
    } ///:~
```

Tout d'abord on remarque quelques méthodes servant de raccourcis pour imprimer : la méthode **prt( )** imprime une **String**, **pInt( )** imprime une **String** suivie d'un **int** et **pFlt( )** imprime une **String** suivie d'un **float**. Bien entendu toutes se terminent par un appel à **System.out.println( )**.

Pour générer des nombres, le programme crée un objet de type **Random**. Aucun argument n'étant passé à la création, Java utilise l'heure courante comme semence d'initialisation pour le générateur de nombres aléatoires. Pour générer des nombres de différents types, le programme appelle tout simplement différentes méthodes de l'objet **Random** : **nextInt( )**, **nextLong( )**, **nextFloat( )** ou **nextDouble( )**.

L'opérateur modulo, appliqué au résultat du générateur de nombres aléatoires, limite le résultat à un maximum correspondant à la valeur de l'opérande moins un (dans ce cas, 99).

Les opérateurs unaires (à un opérande) moins et plus

Le moins unaire (-) et le plus unaire (+) sont identiques au moins binaire et au plus binaire. Le compilateur les reconnaît par le contexte de l'expression. Par exemple, l'instruction :

```
x = -a;
```

a une signification évidente. Le compilateur est capable d'interpréter correctement :

```
x = a * -b;
```

mais le lecteur pourrait être déconcerté, aussi est-il plus clair d'écrire :

```
x = a * (-b);
```

Le moins unaire a pour résultat la négation de la valeur. Le plus unaire existe pour des raisons de symétrie, toutefois il n'a aucun effet.

Incrémentation et décrémentation automatique

Java, tout comme C, possède beaucoup de raccourcis. Les raccourcis autorisent un code plus concis, ce qui le rend plus facile ou difficile à lire suivant les cas.

Les deux raccourcis les plus agréables sont les opérateurs d'incrémentation et de décrémentation (souvent cités en tant qu'opérateur d'auto-incrémentation et d'auto-décrémentation). L'opérateur de décrémentation est « **--** » et signifie « diminuer d'une unité ». L'opérateur d'incrémentation est « ++ » et signifie « augmenter d'une unité ». Si **a** est un **int**, par exemple, l'expression **++a** est équivalente à (**a = a + 1**). Le résultat des opérateurs d'incrémentation et de décrémentation est la variable elle-même.

Il existe deux versions de chaque type d'opérateur, souvent nommées version préfixée et version postfixée. Pour les deux opérateurs, incrémentation et décrémentation, préfixée signifie que l'opérateur (« ++ » ou « -- ») se trouve juste avant la variable ou l'expression, postfixée que l'opérateur se trouve après la variable ou l'expression. Pour la pré-incrémentation et la pré-décrémentation, (c'est à dire, **++a** ou **--a**), l'opération est réalisée en premier, puis la valeur est produite. Pour la post-incrémentation et la post-décrémentation (c'est à dire **a++** ou **a--**), la valeur est produite, puis l'opération est réalisée. En voici un exemple :

```
//: c03:AutoInc.java
// Démonstration des opérateurs ++ et --.

public class AutoInc {
  public static void main(String[] args) {
    int i = 1;
    prt("i : " + i);
    prt("++i : " + ++i); // Pré-incrémentation
    prt("i++ : " + i++); // Post-incrémentation
    prt("i : " + i);
    prt("--i : " + --i); // Pré-décrémentation
    prt("i-- : " + i--); // Post-décrémentation
    prt("i : " + i);
  }
  static void prt(String s) {
    System.out.println(s);
  }
```

```
} ///:~
```

Voici le résultat de ce programme :

```
i : 1
++i : 2
i++ : 2
i : 3
--i : 2
i-- : 2
i : 1
```

On constate qu'avec la forme préfixée on obtient la valeur de la variable après que l'opération ait été exécutée, et qu'avec la forme postfixée on obtient la valeur de la variable avant que l'opération ne soit réalisée. Ce sont les seuls opérateurs ayant des effets de bord, mis à part ceux qui impliquent l'affectation : en d'autres termes, ils modifient l'opérande au lieu de simplement utiliser sa valeur.

L'opérateur d'incrémentation explique le nom C++, qui voudrait signifier « un pas de plus au-delà de C ». Dans un ancien discours sur Java, Bill Joy (l'un des créateurs), disait que « Java = C++-- » (C plus plus moins moins), suggérant que Java serait C++ auquel on aurait ôté les parties difficiles et non nécessaires, et qu'il serait donc un langage bien plus simple. Au fil de votre lecture, vous découvrirez ici que beaucoup de choses sont plus simples, bien que Java ne soit pas *tellement* plus simple que C++.

Les opérateurs relationnels

Les opérateurs relationnels créent un résultat de type **boolean**. Ils évaluent les rapports entre les valeurs des opérandes. Une expression relationnelle renvoie **true** si le rapport est vrai, **false** dans le cas opposé. Les opérateurs relationnels sont : plus petit que (<), plus grand que (>), plus petit que ou égal à (<=), plus grand que ou égal à (>=), équivalent (==) et non équivalent (!=). Le type **boolean** n'accepte comme opérateur relationnel que les opérateurs d'équivalence (==) et de non équivalence (!=), lesquels peuvent être utilisés avec tous les types de données disponibles dans le langage.

Tester l'équivalence des objets

Les opérateurs relationnels == et **!=** fonctionnent avec tous les objets, mais leur utilisation déroute souvent le programmeur Java novice. Voici un exemple :

```
//: c03:Equivalence.java

public class Equivalence {
  public static void main(String[] args) {
    Integer n1 = new Integer(47);
    Integer n2 = new Integer(47);
    System.out.println(n1 == n2);
    System.out.println(n1 != n2);
  }
```

```
} ///:~
```

L'expression **System.out.println(n1 == n2)** imprimera le résultat de la comparaison de type **boolean**. Il semble à priori évident que la sortie sera **true** puis **false**, puisque les deux objets de type **Integer** sont identiques. Mais, bien que le *contenu* des objets soit le même, les *références* sont différentes, et il se trouve que les opérateurs == and **!**= comparent des références d'objet. En réalité la sortie sera **false** puis **true**. Naturellement, cela en surprendra plus d'un.

Que faire si on veut comparer le contenu réel d'un objet ? Il faut utiliser la méthode spéciale **equals( )** qui existe pour tous les objets (mais non pour les types primitifs, qui s'accommodent mieux de == et **!**=). Voici comment l'utiliser :

```
//: c03:EqualsMethod.java

public class EqualsMethod {
  public static void main(String[] args) {
    Integer n1 = new Integer(47);
    Integer n2 = new Integer(47);
    System.out.println(n1.equals(n2));
  }
} ///:~
```

```
//: c03:EqualsMethod2.java

class Value {
  int i;
}

public class EqualsMethod2 {
  public static void main(String[] args) {
    Value v1 = new Value();
    Value v2 = new Value();
    v1.i = v2.i = 100;
    System.out.println(v1.equals(v2));
  }
} ///:~
```

nous voici revenus à la case départ : le résultat est **false**. Ceci parce que, par défaut, **equals( )** compare des *références*. Aussi, faute de redéfinir **equals( )** dans la nouvelle classe, nous n'obtiendrons pas le résultat désiré. Mais la redéfinition des méthodes ne sera exposée que dans le Chapitre 7, aussi d'ici là il nous faudra garder à l'esprit que l'utilisation de **equals( )** peut poser des problèmes.

Beaucoup de classes des bibliothèques Java implémentent la méthode **equals( )** afin de comparer le contenu des objets plutôt que leurs références.

Les opérateurs logiques

Les opérateurs logiques AND (**&&**), OR (||) et NOT (**!**) produisent une valeur **boolean** qui prend la valeur **true** ou **false** en fonction des arguments. Cet exemple utilise les opérateurs relationnels et logiques :

```
//: c03:Bool.java
// Opérateurs relationnels et logiques.
import java.util.*;

public class Bool {
  public static void main(String[] args) {
    Random rand = new Random();
    int i = rand.nextInt() % 100;
    int j = rand.nextInt() % 100;
    prt("i = " + i);
    prt("j = " + j);
    prt("i > j is " + (i > j));
    prt("i < j is " + (i < j));
    prt("i >= j is " + (i >= j));
    prt("i <= j is " + (i <= j));
    prt("i == j is " + (i == j));
    prt("i != j is " + (i != j));

// Traiter un int comme un boolean
// n'est pas légal en Java
//! prt("i && j is " + (i && j));
//! prt("i || j is " + (i || j));
//! prt("!i is " + !i);

    prt("(i < 10) && (j < 10) is "
      + ((i < 10) && (j < 10)) );
    prt("(i < 10) || (j < 10) is "
      + ((i < 10) || (j < 10)) );
  }
  static void prt(String s) {
    System.out.println(s);
  }
} ///:~
```

On ne peut appliquer AND, OR, et NOT qu'aux valeurs **boolean**. On ne peut pas utiliser une variable non booléenne comme si elle était booléenne, comme on le fait en C et C++. Les tentatives (erronées) de le faire ont été mises en commentaires avec le marqueur **//!**. Les expressions qui suivent, toutefois, produisent des valeurs **boolean** en utilisant les opérateurs de comparaisons relationnels, puis appliquent des opérations logiques sur les résultats.

Exemple de listing de sortie :

```
i = 85
j = 4
```

```
i > j is true
i < j is false
i >= j is true
i <= j is false
i == j is false
i != j is true
(i < 10) && (j < 10) is false
(i < 10) || (j < 10) is true
```

Notez qu'une valeur **boolean** est automatiquement convertie en texte approprié lorsqu'elle est utilisée dans un contexte où on attend une **String**.

Dans le programme précédent, on peut remplacer la définition **int** par n'importe quelle autre donnée de type primitif excepté **boolean**. Toutefois il faut rester attentif au fait que la comparaison de nombres en virgule flottante est très stricte. Un nombre qui diffère très légèrement d'un autre est toujours « différent ». Un nombre représenté par le plus petit bit significatif au-dessus de zéro est différent de zéro.

« Court-circuit »

En travaillant avec les opérateurs logiques on rencontre un comportement appelé « court-circuit ». Cela signifie que l'évaluation de l'expression sera poursuivie *jusqu'à ce que* la vérité ou la fausseté de l'expression soit déterminée sans ambiguïté. En conséquence, certaines parties d'une expression logique peuvent ne pas être évaluées. Voici un exemple montrant une évaluation « court-circuitée » :

```
//: c03:ShortCircuit.java
// Démonstration du fonctionnement du "court-circuit"
// avec les opérateurs logiques.

public class ShortCircuit {
  static boolean test1(int val) {
    System.out.println("test1(" + val + ")");
    System.out.println("result: " + (val < 1));
    return val < 1;
  }
  static boolean test2(int val) {
    System.out.println("test2(" + val + ")");
    System.out.println("result: " + (val < 2));
    return val < 2;
  }
  static boolean test3(int val) {
    System.out.println("test3(" + val + ")");
    System.out.println("result: " + (val < 3));
    return val < 3;
  }
  public static void main(String[] args) {
    if(test1(0) && test2(2) && test3(2))
```

```
      System.out.println("expression is true");
    else
      System.out.println("expression is false");
  }
} ///:~
```

Chaque fonction test effectue une comparaison sur l'argument et renvoie **true** ou **false**. De plus elle imprime cette valeur pour montrer qu'elle est appelée. Les tests sont utilisés dans l'expression :

```
if(test1(0) && test2(2) && test3(2))
```

On pourrait naturellement penser que les trois tests sont exécutés, mais la sortie montre le contraire :

```
test1(0)
result: true
test2(2)
result: false
expression is false
```

Le premier test produit un résultat **true**, et l'évaluation de l'expression se poursuit. Toutefois, le second test produit un résultat **false**. Puisque cela signifie que l'expression complète sera **false**, pourquoi poursuivre l'évaluation du reste de l'expression ? Cela pourrait avoir un coût. C'est de fait la justification du « court-circuit » : gagner potentiellement en performance s'il n'est pas nécessaire d'évaluer complètement l'expression logique.

Les opérateurs bit à bit

Les opérateurs bit à bit permettent de manipuler les bits individuels d'une donnée de type primitif. Les opérateurs bit à bit effectuent des opérations d'algèbre booléenne sur les bits en correspondance dans les deux arguments afin de produire un résultat.

L'origine des opérateurs bit à bit est à rechercher dans l'orientation bas niveau du langage C ; il fallait alors manipuler directement le hardware ainsi que les bits des registres hardware. Java a été conçu à l'origine pour être embarqué dans les décodeurs TV, ce qui explique cette orientation bas niveau. Vous n'utiliserez vraisemblablement pas beaucoup ces opérateurs.

L'opérateur AND (**&**) bit à bit retourne la valeur un si les deux bits correspondants des opérandes d'entrée sont à un ; sinon il retourne la valeur zéro. L'opérateur OR (|) bit à bit retourne la valeur un si l'un des deux bits correspondants des opérandes d'entrée est à un et retourne la valeur zéro dans le cas où les deux bits sont à zéro. L'opérateur EXCLUSIVE OR, ou XOR (^), retourne la valeur un si l'un des deux bits correspondants des opérandes est à un, mais pas les deux. L'opérateur NOT bit à bit (~ , appelé également opérateur de *complément à un*) est un opérateur unaire, il a un seul argument (tous les autres opérateurs bit à bit sont des opérateurs binaires), il renvoie l'opposé de l'argument - un si le bit de l'argument est à zéro, zéro si le bit est à un.

Les opérateurs bit à bit et les opérateurs logiques étant représentés par les mêmes caractères, je vous propose un procédé mnémotechnique pour vous souvenir de leur signification : les bits étant « petits », les opérateurs bit à bit comportent un seul caractère.

Les opérateurs bit à bit peuvent être combinés avec le signe = pour réaliser en une seule fois opération et affectation : **&=**, |= et ^= sont tous légitimes. (~ étant un opérateur unaire, il ne peut être combiné avec le signe =).

Le type **boolean** est traité comme une valeur binaire et il est quelque peu différent. Il est possible de réaliser des opérations AND, OR et XOR « bit à bit », mais il est interdit d'effectuer un NOT « bit à bit » (vraisemblablement pour ne pas faire de confusion avec le NOT logique). Pour le type **boolean** les opérateurs bit à bit ont le même effet que les opérateurs logiques, sauf qu'il n'y a pas de « court-circuit ». De plus, parmi les opérations bit à bit effectuées sur des types **boolean** il existe un opérateur XOR logique qui ne fait pas partie de la liste des opérateurs « logiques ». Enfin, le type **boolean** ne doit pas être utilisé dans les expressions de décalage décrites ci-après.

Les opérateurs de décalage

Les opérateurs de décalage manipulent eux aussi des bits. On ne peut les utiliser qu'avec les types primitifs entiers. L'opérateur de décalage à gauche (<<) a pour résultat la valeur de l'opérande situé à gauche de l'opérateur, décalée vers la gauche du nombre de bits spécifié à droite de l'opérateur (en insérant des zéros dans les bits de poids faible). L'opérateur signé de décalage à droite (>>) a pour résultat la valeur de l'opérande situé à gauche de l'opérateur, décalée vers la droite du nombre de bits spécifié à droite de l'opérateur. L'opérateur signé de décalage à droite >> *étend le signe* : si la valeur est positive, des zéros sont insérés dans les bits de poids fort ; si la valeur est négative, des uns sont insérés dans les bits de poids fort. Java comprend également un opérateur de décalage à droite non signé **>>>,** qui *étend les zéros* : quel que soit le signe, des zéros sont insérés dans les bits de poids fort. Cet opérateur n'existe pas en C ou C++.

Si on décale un **char**, **byte,** ou **short**, il sera promu en **int** avant le décalage, et le résultat sera un **int**. Seuls seront utilisés les cinq bits de poids faible de la valeur de décalage, afin de ne pas décaler plus que le nombre de bits dans un **int**. Si on opère avec un **long**, le résultat sera un **long**. Seuls les six bits de poids faible de la valeur de décalage seront utilisés, on ne peut donc décaler un **long** d'un nombre de bits supérieur à celui qu'il contient.

Les décalages peuvent être combinés avec le signe égal (<<=, >>= ou >>>=). La *lvalue* est remplacée par la *lvalue* décalée de la valeur *rvalue*. Il y a un problème, toutefois, avec le décalage à droite non signé combiné à une affectation. Son utilisation avec un **byte** ou un **short** ne donne pas un résultat correct. En réalité, l'opérande est promu en **int**, décalé à droite, puis tronqué comme s'il devait être affecté dans sa propre variable, et dans ce cas on obtient **-1**. L'exemple suivant démontre cela :

```
//: c03:URShift.java
// Test du décalage à droite non signé.

public class URShift {
  public static void main(String[] args) {
    int i = -1;
    i >>>= 10;
    System.out.println(i);
    long l = -1;
    l >>>= 10;
    System.out.println(l);
    short s = -1;
```

```
    s >>>= 10;
    System.out.println(s);
    byte b = -1;
    b >>>= 10;
    System.out.println(b);
    b = -1;
    System.out.println(b>>>10);
  }
} ///:~
```

Dans la dernière ligne, la valeur résultante n'est pas réaffectée à **b**, mais directement imprimée et dans ce cas le comportement est correct.

Voici un exemple montrant l'utilisation de tous les opérateurs travaillant sur des bits :

```
//: c03:BitManipulation.java
// Utilisation des opérateurs bit à bit.
import java.util.*;

public class BitManipulation {
  public static void main(String[] args) {
    Random rand = new Random();
    int i = rand.nextInt();
    int j = rand.nextInt();
    pBinInt("-1", -1);
    pBinInt("+1", +1);
    int maxpos = 2147483647;
    pBinInt("maxpos", maxpos);
    int maxneg = -2147483648;
    pBinInt("maxneg", maxneg);
    pBinInt("i", i);
    pBinInt("~i", ~i);
    pBinInt("-i", -i);
    pBinInt("j", j);
    pBinInt("i & j", i & j);
    pBinInt("i | j", i | j);
    pBinInt("i ^ j", i ^ j);
    pBinInt("i << 5", i << 5);
    pBinInt("i >> 5", i >> 5);
    pBinInt("(~i) >> 5", (~i) >> 5);
    pBinInt("i >>> 5", i >>> 5);
    pBinInt("(~i) >>> 5", (~i) >>> 5);

    long l = rand.nextLong();
    long m = rand.nextLong();
    pBinLong("-1L", -1L);
    pBinLong("+1L", +1L);
    long ll = 9223372036854775807L;
```

```
    pBinLong("maxpos", ll);
    long lln = -9223372036854775808L;
    pBinLong("maxneg", lln);
    pBinLong("l", l);
    pBinLong("~l", ~l);
    pBinLong("-l", -l);
    pBinLong("m", m);
    pBinLong("l & m", l & m);
    pBinLong("l | m", l | m);
    pBinLong("l ^ m", l ^ m);
    pBinLong("l << 5", l << 5);
    pBinLong("l >> 5", l >> 5);
    pBinLong("(~l) >> 5", (~l) >> 5);
    pBinLong("l >>> 5", l >>> 5);
    pBinLong("(~l) >>> 5", (~l) >>> 5);
  }
  static void pBinInt(String s, int i) {
    System.out.println(
      s + ", int: " + i + ", binary: ");
    System.out.print("   ");
    for(int j = 31; j >=0; j--)
      if(((1 << j) &  i) != 0)
        System.out.print("1");
      else
        System.out.print("0");
    System.out.println();
  }
  static void pBinLong(String s, long l) {
    System.out.println(
      s + ", long: " + l + ", binary: ");
    System.out.print("   ");
    for(int i = 63; i >=0; i--)
      if(((1L << i) & l) != 0)
        System.out.print("1");
      else
        System.out.print("0");
    System.out.println();
  }
} ///:~
```

Les deux dernières méthodes, **pBinInt( )** et **pBinLong( )** sont appelées respectivement avec un **int** et un **long**, et l'impriment en format binaire avec une description. Nous ne parlerons pas de cette implémentation pour le moment.

Remarquez l'utilisation de **System.out.print( )** au lieu de **System.out.println( )**. La méthode **print( )** n'émet pas de retour-chariot, et permet ainsi d'imprimer une ligne en plusieurs fois.

Cet exemple montre l'effet de tous les opérateurs bit à bit pour les **int** et les **long**, mais aussi ce qui se passe avec les valeurs minimale, maximale, +1 et -1, pour un **int** et pour un **long**. Noter

que le bit de poids le plus fort représente le signe : 0 signifie positif, 1 négatif. Voici la sortie pour la partie **int** :

```
-1, int: -1, binary:
  11111111111111111111111111111111
+1, int: 1, binary:
  00000000000000000000000000000001
maxpos, int: 2147483647, binary:
  01111111111111111111111111111111
maxneg, int: -2147483648, binary:
  10000000000000000000000000000000
i, int: 59081716, binary:
  00000011100001011000001111110100
~i, int: -59081717, binary:
  11111100011110100111110000001011
-i, int: -59081716, binary:
  11111100011110100111110000001100
j, int: 198850956, binary:
  00001011110110100011100110001100
i & j, int: 58720644, binary:
  00000011100000000000000110000100
i | j, int: 199212028, binary:
  00001011110111111011101111111100
i ^ j, int: 140491384, binary:
  00001000010111111011101001111000
i << 5, int: 1890614912, binary:
  01110000101100000111111010000000
i >> 5, int: 1846303, binary:
  00000000000111000010110000011111
(~i) >> 5, int: -1846304, binary:
  11111111111000111101001111100000
i >>> 5, int: 1846303, binary:
  00000000000111000010110000011111
(~i) >>> 5, int: 132371424, binary:
  00000111111000111101001111100000
```

La représentation binaire des nombres est dite en *complément à deux signé*.

Cet opérateur est inhabituel parce qu'il a trois opérandes. C'est un véritable opérateur dans la mesure où il produit une valeur, à l'inverse de l'instruction habituelle if-else que nous étudierons dans la prochaine section de ce chapitre. L'expression est de la forme :

```
expression-booléenne ? valeur0 : valeur1
```

Si le résultat de *expression-booléenne* est **true**, l'expression *valeur0* est évaluée et son résultat devient le résultat de l'opérateur. Si *expression-booléenne* est **false**, c'est l'expression *valeur1* qui est évaluée et son résultat devient le résultat de l'opérateur.

Bien entendu, il est possible d'utiliser à la place une instruction **if-else** (qui sera décrite plus loin), mais l'opérateur ternaire est plus concis. Bien que C (d'où est issu cet opérateur) s'enorgueillit

d'être lui-même un langage concis, et que l'opérateur ternaire ait été introduit, entre autres choses, pour des raisons d'efficacité, il faut se garder de l'utiliser à tout bout de champ car on aboutit très facilement à un code illisible.

Cet opérateur conditionnel peut être utilisé, soit pour ses effets de bord, soit pour la valeur produite, mais en général on recherche la valeur puisque c'est elle qui rend cet opérateur distinct du **if-else**. En voici un exemple :

```
static int ternary(int i) {
  return i < 10 ? i * 100 : i * 10;
}
```

Ce code est plus compact que celui qu'on aurait écrit sans l'opérateur ternaire :

```
static int alternative(int i) {
  if (i < 10)
    return i * 100;
  else
    return i * 10;
}
```

La deuxième forme est plus compréhensible, et ne nécessite pas de commentaire. Il est donc nécessaire de bien peser tous les arguments avant d'opter pour l'opérateur ternaire.

L'opérateur virgule

La virgule est utilisée en C et C++ non seulement comme séparateur dans la liste des argument des fonctions, mais aussi en tant qu'opérateur pour une évaluation séquentielle. L'*opérateur* virgule est utilisé en Java uniquement dans les boucles **for**, qui seront étudiées plus loin dans ce chapitre.

L'opérateur + pour les String

Un des opérateurs a une utilisation spéciale en Java : l'opérateur + peut être utilisé pour concaténer des chaînes de caractères, comme on l'a déjà vu. Il semble que ce soit une utilisation naturelle de l'opérateur **+,** même si cela ne correspond pas à son utilisation traditionnelle. Étendre cette possibilité semblait une bonne idée en C++, aussi la *surcharge d'opérateurs* fut ajoutée au C++ afin de permettre au programmeur d'ajouter des significations différentes à presque tous les opérateurs. En fait, la surcharge d'opérateurs, combinée à d'autres restrictions du C++, s'est trouvée être très compliquée à mettre en oeuvre par les programmeurs pour la conception de leurs classes. En Java, la surcharge d'opérateurs aurait été plus simple à implémenter qu'elle ne l'a été en C++ ; mais cette fonctionnalité a été jugée trop complexe, et les programmeurs Java, à la différence des programmeurs C++, ne peuvent implémenter leurs propres surcharges d'opérateurs.

L'utilisation de l'opérateur + pour les **String** présente quelques caractéristiques intéressantes. Si une expression commence par une **String**, alors tous les opérandes qui suivent doivent être des **String** (souvenez-vous que le compilateur remplace une séquence de caractères entre guillemets par une **String**) :

```
int x = 0, y = 1, z = 2;
```

```
String sString = "x, y, z ";
System.out.println(sString + x + y + z);
```

Ici, le compilateur Java convertit **x**, **y**, et **z** dans leurs représentations **String** au lieu d'ajouter d'abord leurs valeurs. Et si on écrit :

```
System.out.println(x + sString);
```

Java remplacera **x** par une **String**.

Les pièges classiques dans l'utilisation des opérateurs

L'un des pièges dûs aux opérateurs est de vouloir se passer des parenthèses alors qu'on n'est pas tout à fait certain de la manière dont sera évaluée l'opération. Ceci reste vrai en Java.

Une erreur très classique en C et C++ ressemble à celle-ci :

```
while(x = y) {
   // ....
}
```

Le programmeur voulait tester l'équivalence (==) et non effectuer une affectation. En C et C++ le résultat de cette affectation est toujours **true** si **y** est différent de zéro, et on a toutes les chances de partir dans une boucle infinie. En Java, le résultat de cette expression n'est pas un **boolean** ; le compilateur attendant un **boolean**, et ne transtypant pas l'**int**, générera une erreur de compilation avant même l'exécution du programme. Par suite cette erreur n'apparaîtra jamais en Java. Il n'y aura aucune erreur de compilation *que* dans le *seul* cas où **x** et **y** sont des **boolean**, pour lequel **x = y** est une expression légale ; mais il s'agirait probablement d'une erreur dans l'exemple ci-dessus.

Un problème similaire en C et C++ consiste à utiliser les AND et OR bit à bit au lieu de leurs versions logiques. Les AND et OR bit à bit utilisent un des caractères (**&** ou |) alors que les AND et OR logique en utilisent deux (**&&** et ||). Tout comme avec = et ==, il est facile de ne frapper qu'un caractère au lieu de deux. En Java, le compilateur interdit cela et ne vous laissera pas utiliser cavalièrement un type là où il n'a pas lieu d'être.

Les opérateurs de transtypage

Le mot transtypage est utilisé dans le sens de « couler dans un moule ». Java transforme automatiquement un type de données dans un autre lorsqu'il le faut. Par exemple, si on affecte une valeur entière à une variable en virgule flottante, le compilateur convertira automatiquement l'**int** en **float**. Le transtypage permet d'effectuer cette conversion explicitement, ou bien de la forcer lorsqu'elle ne serait pas effectuée implicitement.

Pour effectuer un transtypage, il suffit de mettre le type de données voulu (ainsi que tous ses modificateurs) entre parenthèses à gauche de n'importe quelle valeur. Voici un exemple :

```
void casts() {
  int i = 200;
  long l = (long)i;
  long l2 = (long)200;
}
```

Comme on peut le voir, il est possible de transtyper une valeur numérique aussi bien qu'une variable. Toutefois, dans les deux exemples présentés ici, le transtypage est superflu puisque le compilateur promouvra une valeur **int** en **long** si nécessaire. Il est tout de même possible d'effectuer un tel transtypage, soit pour le souligner, soit pour rendre le code plus clair. Dans d'autres situations, un transtypage pourrait être utilisé afin d'obtenir un code compilable.

En C et C++, le transtypage est parfois la source de quelques migraines. En Java, le transtypage est sûr, avec l'exception suivante : lorsqu'on fait ce qu'on appelle une conversion rétrécissante (c'est à dire lorsqu'on transtype depuis un type de données vers un autre, le premier pouvant contenir plus d'information que le second) on court le risque de perdre de l'information. Dans ce cas le compilateur demande un transtypage explicite, en émettant un message à peu près formulé ainsi « ceci peut être dangereux - néanmoins, si c'est ce que vous voulez vraiment faire, je vous en laisse la responsabilité ». Avec une conversion élargissante, le transtypage explicite n'est pas obligatoire car il n'y a pas de risque de perte d'information, le nouveau type pouvant contenir plus d'information que l'ancien.

Java permet de transtyper n'importe quel type primitif vers n'importe quel autre type primitif, excepté le type **boolean,** pour lequel il n'existe aucun transtypage. Les types Class ne peuvent être transtypés. Pour convertir une classe en une autre il faut utiliser des méthodes spéciales. (**String** est un cas à part, et on verra plus loin dans ce livre que les objets peuvent être transtypés à l'intérieur d'une *famille* de types ; un **Chêne** peut être transtypé en **Arbre** et vice versa, mais non dans un type étranger tel que **Roche**).

Les littéraux

Habituellement le compilateur sait exactement quel type affecter aux valeurs littérales insérées dans un programme. Quelquefois, cependant, le type est ambigu. Dans de tels cas il faut guider le compilateur en ajoutant une information supplémentaire sous la forme de caractères associés à la valeur littérale. Le code suivant montre l'utilisation de ces caractères :

```
//: c03:Literals.java

class Literals {
  char c = 0xffff; // plus grande valeur char en hexadécimal
  byte b = 0x7f; // plus grande valeur byte en hexadécimal
  short s = 0x7fff; // plus grande valeur short en hexadécimal
  int i1 = 0x2f; // Hexadécimal (minuscules)
  int i2 = 0X2F; // Hexadécimal (majuscules)
  int i3 = 0177; // Octal (avec zéro en tête)
  // Hexadécimal et Octal avec des long.
  long n1 = 200L; // suffixe long
  long n2 = 200l; // suffixe long
  long n3 = 200;
  //! long l6(200); // non autorisé
  float f1 = 1;
  float f2 = 1F; // suffixe float
  float f3 = 1f; // suffixe float
  float f4 = 1e-45f; // 10 puissance
  float f5 = 1e+9f; // suffixe float
```

```
    double d1 = 1d; // suffixe double
    double d2 = 1D; // suffixe double
    double d3 = 47e47d; // 10 puissance
  } ///:~
```

L'hexadécimal (base 16), utilisable avec tous les types entier, est représenté par **0x** ou **0X** suivi de caractères **0-9** et/ou **a-f** en majuscules ou en minuscules. Si on tente d'initialiser une variable avec une valeur plus grande que celle qu'elle peut contenir (indépendamment de la forme numérique de la valeur), le compilateur émettra un message d'erreur. Le code ci-dessus montre entre autres la valeur hexadécimale maximale possible pour les types **char**, **byte,** et **short**. Si on dépasse leur valeur maximale, le compilateur crée automatiquement une valeur **int** et émet un message nous demandant d'utiliser un transtypage rétrécissant afin de réaliser l'affectation : Java nous avertit lorsqu'on franchit la ligne.

L'octal (base 8) est représenté par un nombre dont le premier digit est **0** (zéro) et les autres **0-7**. Il n'existe pas de représentation littérale des nombres binaires en C, C++ ou Java.

Un caractère suivant immédiatement une valeur littérale établit son type : **L**, majuscule ou minuscule, signifie **long** ; **F**, majuscule ou minuscule, signifie **float**, et **D**, majuscule ou minuscule, **double**.

Les exposants utilisent une notation qui m'a toujours passablement consterné : **1.39 e-47f**. En science et en ingénierie, « **e** » représente la base des logarithmes naturels, approximativement 2.718 (une valeur **double** plus précise existe en Java, c'est **Math.E**). **e** est utilisé dans les expressions d'exponentielle comme 1.39 x e-47, qui signifie 1.39 x 2.718-47. Toutefois, lorsque FORTRAN vit le jour il fut décidé que **e** signifierait naturellement « dix puissance », décision bizarre puisque FORTRAN a été conçu pour résoudre des problèmes scientifiques et d'ingénierie, et on aurait pu penser que ses concepteurs auraient fait preuve de plus de bons sens avant d'introduire une telle ambiguïté. [25] Quoi qu'il en soit, cette habitude a continué avec C, C++ et maintenant Java. Ceux d'entre vous qui ont utilisé **e** en tant que base des logarithmes naturels doivent effectuer une translation mentale en rencontrant une expression telle que **1.39 e-47f** en Java ; elle signifie 1.39 x 10-47.

Noter que le caractère de fin n'est pas obligatoire lorsque le compilateur est capable de trouver le type approprié. Avec :

```
long n3 = 200;
```

il n'y a pas d'ambiguïté, un **L** suivant le 200 serait superflu. Toutefois, avec :

```
float f4 = 1e-47f; // 10 puissance
```

le compilateur traite normalement les nombres en notation scientifique en tant que **double**, et en l'absence du **f** final générerait un message d'erreur disant qu'on doit effectuer un transtypage explicite afin de convertir un **double** en **float**.

La promotion

Lorsqu'on effectue une opération mathématique ou bit à bit sur des types de données primitifs plus petits qu'un **int** (c'est à dire **char**, **byte,** ou **short**), on découvre que ces valeurs sont promues en **int** avant que les opérations ne soient effectuées, et que le résultat est du type **int**. Par suite, si on affecte ce résultat à une variable d'un type plus petit, il faut effectuer un transtypage explicite qui peut

d'ailleurs entraîner une perte d'information. En général, dans une expression, la donnée du type le plus grand est celle qui détermine le type du résultat de cette expression ; en multipliant un **float** et un **double**, le résultat sera un **double** ; en ajoutant un **int** et un **long**, le résultat sera un **long**.

Java n'a pas de « sizeof »

En C and C++, l'opérateur **sizeof( )** satisfait un besoin spécifique : il renseigne sur le nombre d'octets alloués pour les données individuelles. En C et C++, la principale raison d'être de l'opérateur **sizeof( )** est la portabilité. Plusieurs types de données peuvent avoir des tailles différentes sur des machines différentes, et le programmeur doit connaître la taille allouée pour ces types lorsqu'il effectue des opérations sensibles à la taille des données. Par exemple, un ordinateur peut traiter les entiers sur 32 bits, alors qu'un autre les traitera sur 16 bits : les programmes peuvent ranger de plus grandes valeurs sur la première machine. Comme on peut l'imaginer, la portabilité est un énorme casse-tête pour les programmeurs C et C++.

Java n'a pas besoin d'un opérateur **sizeof( )** car tous les types de données ont la même taille sur toutes les machines. Il n'est absolument pas besoin de parler de portabilité à ce niveau - celle-ci est déjà intégrée au langage.

Retour sur la priorité des opérateurs

Lors d'un séminaire, entendant mes jérémiades au sujet de la difficulté de se souvenir de la priorité des opérateurs, un étudiant suggéra un procédé mnémotechnique qui est également un commentaire : « Ulcer Addicts Really Like C A lot ». (« Les Accrocs de l'Ulcère Adorent Réellement C »)

Mnémonique	**Type d'opérateur**	**Opérateurs**
Ulcer	Unaire	**+ - ++--**
Addicts	Arithmétique (et décalage)	*** / % + - << >>**
Really	Relationnel	**> < >= <= == !=**
Like	Logique (et bit à bit)	**&& || & | ^**
C	Conditionnel (ternaire)	**A > B ? X : Y**
A Lot	Affectation	= (et affectation composée comme*****=)

Bien entendu, ce n'est pas un moyen mnémotechnique parfait puisque les opérateurs de décalage et les opérateurs bit à bit sont quelque peu éparpillés dans le tableau, mais il fonctionne pour les autres opérateurs.

Résumé sur les opérateurs

L'exemple suivant montre les types de données primitifs qu'on peut associer avec certains opérateurs. Il s'agit du même exemple de base répété plusieurs fois, en utilisant des types de données primitifs différents. Le fichier ne doit pas générer d'erreur de compilation dans la mesure où les lignes pouvant générer de telles erreurs ont été mises en commentaires avec **//!** :

Noter que le type **boolean** est totalement limité. On peut lui assigner les valeurs **true** et **false**,

on peut tester sa vérité ou sa fausseté, mais on ne peut ni additionner des booléens ni effectuer un autre type d'opération avec eux.

On peut observer l'effet de la promotion des types **char**, **byte**, et **short** avec l'application d'un opérateur arithmétique sur l'un d'entre eux. Le résultat est un **int**, qui doit être explicitement transtypé vers le type original (c'est à dire une conversion rétrécissante qui peut entraîner une perte d'information) afin de l'affecter à une variable de ce type. Avec les valeurs **int**, toutefois, il n'est pas besoin de transtyper, puisque tout ce qu'on obtient est toujours un **int**. N'allez cependant pas croire qu'on peut faire n'importe quoi en toute impunité. Le résultat de la multiplication de deux **int**s un peu trop grands peut entraîner un débordement. L'exemple suivant en fait la démonstration :

```
//: c03:Overflow.java
// Surprise! Java ne contrôle pas vos débordements.

public class Overflow {
  public static void main(String[] args) {
    int big = 0x7fffffff; // plus grande valeur int
    prt("big = " + big);
    int bigger = big * 4;
    prt("bigger = " + bigger);
  }
  static void prt(String s) {
    System.out.println(s);
  }
} ///:~
```

Voici le résultat :

```
big = 2147483647
bigger = -4
```

La compilation se déroule sans erreur ni avertissement ; il n'y a pas d'exception lors de l'exécution : Java est puissant, mais tout de même *pas* à ce point-là.

Les affectations composées ne nécessitent *pas* de transtypage pour les types **char**, **byte,** ou **short**, bien qu'elles entraînent des promotions qui ont le même résultat que les opérations arithmétiques directes. Cette absence de transtypage simplifie certainement le code.

On remarque qu'à l'exception du type **boolean**, tous les types primitifs peuvent être transtypés entre eux. Il faut rester attentif à l'effet des conversions rétrécissantes en transtypant vers un type plus petit, sous peine de perdre de l'information sans en être averti.

Le Contrôle d'exécution

Java utilisant toutes les instructions de contrôle de C, bien des choses seront familières au programmeur C ou C++. La plupart des langages de programmation procéduraux possèdent le même type d'instructions de contrôle, et on retrouve beaucoup de choses d'un langage à un autre. En Java, les mots clefs sont **if-else**, **while**, **do-while**, **for**, et une instruction de sélection appelée **switch**. Toutefois Java ne possède pas le **goto** très pernicieux (lequel reste le moyen le plus expéditif pour traiter certains types de problèmes). Il est possible d'effectuer un saut ressemblant au **goto**, mais bien plus contraignant qu'un **goto** classique.

true et false

Toutes les instructions conditionnelles utilisent la vérité ou la fausseté d'une expression conditionnelle pour déterminer le chemin d'exécution. Exemple d'une expression conditionnelle : **A == B**. Elle utilise l'opérateur conditionnel == pour déterminer si la valeur **A** est équivalente à la valeur **B**. L'expression renvoie la valeur **true** ou **false**. Tous les opérateurs relationnels vus plus haut dans ce chapitre peuvent être utilisés pour créer une instruction conditionnelle. Il faut garder à l'esprit que Java ne permet pas d'utiliser un nombre à la place d'un **boolean**, même si c'est autorisé en C et C++ (pour lesquels la vérité est « différent de zéro » et la fausseté « zéro »). Pour utiliser un non-**boolean** dans un test **boolean**, tel que **if(a)**, il faut d'abord le convertir en une valeur **boolean** en utilisant une expression booléenne, par exemple **if(a != 0)**.

if-else

L'instruction **if-else** est sans doute le moyen le plus simple de contrôler le déroulement du programme. La clause **else** est optionnelle, et par suite l'instruction **if** possède deux formes :

```
if(expression booléenne)
  instruction
```

ou bien :

```
if(expression booléenne)
  instruction
else
  instruction
```

L'expression conditionnelle *expression booléenne* doit fournir un résultat de type **boolean**. *instruction* désigne soit une instruction simple terminée par un point-virgule, soit une instruction composée, c'est à dire un groupe d'instructions simples placées entre deux accolades. Par la suite, chaque utilisation du mot *« instruction »* sous-entendra que l'instruction peut être simple ou composée.

Voici un exemple d'instruction **if-else** : la méthode **test( )** détermine si une estimation est supérieure, inférieure ou équivalente à une valeur donnée :

```
//: c03:IfElse.java
public class IfElse {
  static int test(int testval, int target) {
    int result = 0;
    if(testval > target)
      result = +1;
    else if(testval < target)
      result = -1;
    else
      result = 0; // Identique
    return result;
  }
  public static void main(String[] args) {
    System.out.println(test(10, 5));
```

```
    System.out.println(test(5, 10));
    System.out.println(test(5, 5));
  }
} ///:~
```

Par convention, on indente le corps d'une instruction de contrôle de flux, afin que le lecteur détermine plus facilement son début et sa fin.

return

Le mot clef **return** a deux buts : il spécifie la valeur que la méthode doit retourner (si elle n'a pas un type de retour **void**) et provoque le renvoi immédiat de cette valeur. Voici la méthode **test( )** ci-dessus, réécrite en utilisant cette propriété :

```
//: c03:IfElse2.java
public class IfElse2 {
  static int test(int testval, int target) {
    int result = 0;
    if(testval > target)
      return +1;
    else if(testval < target)
      return -1;
    else
      return 0; // Identique
  }
  public static void main(String[] args) {
    System.out.println(test(10, 5));
    System.out.println(test(5, 10));
    System.out.println(test(5, 5));
  }
} ///:~
```

Itération

Les instructions de contrôle de boucle **while**, **do-while** et **for** sont souvent appelés *instructions d'itération*. Une *instruction* est répétée jusqu'à ce que *l'expression booléenne* de contrôle devienne fausse. Voici la forme d'une boucle **while** :

```
while(expression booléenne)
  instruction
```

expression booléenne est évaluée à l'entrée de la boucle, puis après chaque itération ultérieure d'*instruction*.

Voici un exemple simple qui génère des nombres aléatoires jusqu'à l'arrivée d'une condition particulière :

```
//: c03:WhileTest.java
// Démonstration de la boucle while.
```

```
public class WhileTest {
  public static void main(String[] args) {
    double r = 0;
    while(r < 0.99d) {
      r = Math.random();
      System.out.println(r);
    }
  }
} ///:~
```

Ce test utilise la méthode **static random( )** de la bibliothèque **Math**, qui génère une valeur **double** comprise entre 0 et 1. (0 inclus, 1 exclu). L'expression conditionnelle de la boucle **while** signifie « continuer l'exécution de cette boucle jusqu'à ce que le nombre généré soit égal ou supérieur à 0.99 ». Le résultat est une liste de nombre, et la taille de cette liste est différente à chaque exécution du programme.

do-while

Voici la forme de la boucle **do-while** :

```
do
  instruction
while(expression booléenne);
```

La seule différence entre **while** et **do-while** est que dans une boucle **do-while**, *instruction* est exécutée au moins une fois, même si l'expression est fausse la première fois. Dans une boucle **while**, si l'expression conditionnelle est fausse la première fois, l'instruction n'est jamais exécutée. En pratique, la boucle **do-while** est moins utilisée que **while**.

for

La boucle **for** effectue une initialisation avant la première itération. Puis elle effectue un test conditionnel et, à la fin de chaque itération, une « instruction d'itération ». Voici la forme d'une boucle **for** :

```
for(instruction d'initialisation; expression booléenne; instruction d'itération)
  instruction
```

Chacune des expressions *instruction d'initialisation*, *expression booléenne* et *instruction d'itération* peut être vide. *expression booléenne* est testée avant chaque itération, et dès qu'elle est évaluée à **false** l'exécution continue à la ligne suivant l'instruction **for**. À la fin de chaque boucle, *instruction d'itération* est exécutée.

Les boucles **for** sont généralement utilisées pour les tâches impliquant un décompte :

```
//: c03:ListCharacters.java
// Démonstration de la boucle "for" listant
// tous les caractères ASCII.
```

```
public class ListCharacters {
  public static void main(String[] args) {
  for( char c = 0; c < 128; c++)
    if (c != 26 )  // Effacement de l'écran ANSI
      System.out.println(
        "value: " + (int)c +
        " character: " + c);
  }
} ///:~
```

Remarquons que la variable **c** est définie à l'endroit où elle est utilisée, à l'intérieur de l'expression de contrôle de la boucle **for**, plutôt qu'au début du bloc commençant à l'accolade ouvrante. La portée de **c** est l'expression contrôlée par la boucle **for**.

Les langages procéduraux traditionnels tels que C exigent que toutes les variables soient définies au début d'un bloc afin que le compilateur leur alloue de l'espace mémoire lorsqu'il crée un bloc. En Java et C++ les déclarations de variables peuvent apparaître n'importe où dans le bloc, et être ainsi définies au moment où on en a besoin. Ceci permet un style de code plus naturel et rend le code plus facile à comprendre.

Il est possible de définir plusieurs variables dans une instruction **for**, à condition qu'elles aient le même type :

```
for(int i = 0, j = 1;
   i < 10 && j != 11;
   i++, j++)
 /* corps de la boucle for */;
```

La définition **int** de l'instruction **for** s'applique à la fois à **i** et à **j**. La possibilité de définir des variables dans une expression de contrôle est limitée à la boucle **for**. On ne peut l'utiliser dans aucune autre instruction de sélection ou d'itération.

L'opérateur virgule

Au début de ce chapitre j'ai déclaré que l'*opérateur* virgule (à distinguer du *séparateur* virgule, utilisé pour séparer les définitions et les arguments de fonctions) avait un seul usage en Java, à savoir dans l'expression de contrôle d'une boucle **for**. Aussi bien la partie initialisation que la partie itération de l'expression de contrôle peuvent être formées de plusieurs instructions séparées par des virgules, et ces instructions seront évaluées séquentiellement. L'exemple précédent utilisait cette possibilité. Voici un autre exemple :

```
//: c03:CommaOperator.java
public class CommaOperator {
  public static void main(String[] args) {
    for(int i = 1, j = i + 10; i < 5;
        i++, j = i * 2) {
      System.out.println("i= " + i + " j= " + j);
    }
  }
```

```
} ///:~
```

En voici le résultat :

```
i= 1 j= 11
i= 2 j= 4
i= 3 j= 6
i= 4 j= 8
```

remarquez que la partie initialisation ainsi que la partie itération sont évaluées en ordre séquentiel. La partie initialisation peut comporter n'importe quel nombre de définitions *du même type*.

break et continue

Le déroulement de toutes les instructions d'itération peut être contrôlé de l'intérieur du corps de la boucle au moyen des instructions **break** et **continue**. L'instruction **break** sort de la boucle sans exécuter la suite des instructions. L'instruction **continue** arrête l'exécution de l'itération courante, et l'exécution reprend en début de boucle avec l'itération suivante.

Ce programme montre des exemples d'utilisation des instructions **break** et **continue** dans des boucles **for** et **while** :

```
//: c03:BreakAndContinue.java
// Démonstration des mots clefs break et continue.

public class BreakAndContinue {
  public static void main(String[] args) {
    for(int i = 0; i < 100; i++) {
      if(i == 74) break; // Sortie définitive de la boucle for
      if(i % 9 != 0) continue; // Continue avec l'itération suivante
      System.out.println(i);
    }
    int i = 0;
    // une "boucle infinie" :
    while(true) {
      i++;
      int j = i * 27;
      if(j == 1269) break; // Sortie de boucle
      if(i % 10 != 0) continue; // Début de boucle
      System.out.println(i);
    }
  }
} ///:~
```

Dans la boucle **for** la valeur de **i** n'atteint jamais 100 car l'instruction **break** termine la boucle lorsque **i** prend la valeur 74. En principe, il ne faudrait pas utiliser **break** de cette manière, à moins que l'on ne connaisse pas le moment où la condition de fin arrivera. L'instruction **continue** provoque un branchement au début de la boucle d'itération (donc en incrémentant **i**) chaque fois que **i**

n'est pas divisible par 9. Lorsqu'il l'est, la valeur est imprimée.

La seconde partie montre une « boucle infinie » qui, théoriquement, ne devrait jamais s'arrêter. Toutefois, elle contient une instruction **break** lui permettant de le faire. De plus, l'instruction **continue** retourne au début de la boucle au lieu d'exécuter la fin, ainsi la seconde boucle n'imprime que lorsque la valeur de **i** est divisible par 10. La sortie est :

```
0
9
18
27
36
45
54
63
72
10
20
30
40
```

La valeur 0 est imprimée car 0 % 9 a pour résultat 0.

Il existe une seconde forme de boucle infinie, c'est **for(;;)**. Le compilateur traite **while(true)** et **for(;;)** de la même manière, le choix entre l'une et l'autre est donc affaire de goût.

L'infâme « goto »

Le mot clef **goto** est aussi ancien que les langages de programmation. En effet, **goto** a été le premier moyen de contrôle des programmes dans les langages assembleur : « si la condition A est satisfaite, alors sauter ici, sinon sauter là ». Lorsqu'on lit le code assembleur finalement généré par n'importe quel compilateur, on voit qu'il comporte beaucoup de sauts. Toutefois, un **goto** au niveau du code source est un saut, et c'est ce qui lui a donné mauvaise réputation. Un programme n'arrêtant pas de sauter d'un point à un autre ne peut-il être réorganisé afin que le flux du programme soit plus séquentiel ? **goto** tomba en disgrâce après la publication du fameux article « Le goto considéré comme nuisible » écrit par Edsger Dijkstra, et depuis lors les guerres de religion à propos de son utilisation sont devenues monnaie courante, les partisans du mot clef honni recherchant une nouvelle audience.

Comme il est habituel en de semblables situations, la voie du milieu est la plus indiquée. Le problème n'est pas d'utiliser le **goto**, mais de trop l'utiliser - dans quelques rares situations le **goto** est réellement la meilleure façon de structurer le flux de programme.

Bien que **goto** soit un mot réservé de Java, on ne le trouve pas dans le langage ; Java n'a pas de **goto**. Cependant, il existe quelque chose qui ressemble à un saut, lié aux mots clefs **break** et **continue**. Ce n'est pas vraiment un saut, mais plutôt une manière de sortir d'une instruction d'itération. On le présente dans les discussions sur le **goto** parce qu'il utilise le même mécanisme : une étiquette.

Une étiquette est un identificateur suivi du caractère deux points, comme ceci :

```
label1:
```

En Java, une étiquette ne peut se trouver qu'en un *unique* endroit : juste avant une instruction d'itération. Et, j'insiste, *juste* avant - il n'est pas bon de mettre une autre instruction entre l'étiquette et la boucle d'itération. De plus, il n'y a qu'une seule bonne raison de mettre une étiquette avant une itération, c'est lorsqu'on a l'intention d'y nicher une autre itération ou un switch. Ceci parce que les mots clefs **break** et **continue** ont pour fonction naturelle d'interrompre la boucle en cours, alors qu'utilisés avec une étiquette ils interrompent la boucle pour se brancher à l'étiquette :

```
label1:
outer-iteration {
 inner-iteration {
  //...
  break; // 1
  //...
  continue;  // 2
  //...
  continue label1; // 3
  //...
  break label1;  // 4
 }
}
```

Cas 1 : **break** interrompt l'itération intérieure, on se retrouve dans l'itération extérieure. Cas 2 : **continue** branche à l'itération intérieure. Mais, cas 3 : **continue label1** interrompt l'itération intérieure *et* l'itération extérieure, et branche dans tous les cas à **label1**. En fait, l'itération continue, mais en redémarrant à partir de l'itération extérieure. Cas 4 : **break label1** interrompt lui aussi dans tous les cas et branche à **label1**, mais ne rentre pas à nouveau dans l'itération. En fait il sort des deux itérations.

Voici un exemple utilisant les boucles **for** :

```
//: c03:LabeledFor.java
// la boucle for "étiquetée" en Java.

public class LabeledFor {
 public static void main(String[] args) {
  int i = 0;
  outer: // Il ne peut y avoir d'instruction ici
  for(; true ;) { // boucle infinie
   inner: // Il ne peut y avoir d'instruction ici
   for(; i < 10; i++) {
    prt("i = " + i);
    if(i == 2) {
     prt("continue");
     continue;
    }
    if(i == 3) {
     prt("break");
     i++; // Sinon i ne sera
          // jamais incrémenté.
```

```
          break;
        }
        if(i == 7) {
          prt("continue outer");
          i++; // Sinon i ne sera
               // jamais incrémenté.
          continue outer;
        }
        if(i == 8) {
          prt("break outer");
          break outer;
        }
        for(int k = 0; k < 5; k++) {
          if(k == 3) {
            prt("continue inner");
            continue inner;
          }
        }
      }
    }
    // On ne peut pas utiliser un break ou
    // un continue vers une étiquette ici
  }
  static void prt(String s) {
    System.out.println(s);
  }
} ///:~
```

Ce programme utilise la méthode **prt( )** déjà définie dans d'autres exemples.

Noter que **break** sort de la boucle **for**, et l'expression d'incrémentation ne sera jamais exécutée avant le passage en fin de boucle **for**. Puisque **break** saute l'instruction d'incrémentation, l'incrémentation est réalisée directement dans le cas où **i == 3**. Dans le cas **i == 7**, l'instruction **continue outer** saute elle aussi en tête de la boucle, donc saute l'incrémentation, qui est, ici aussi, réalisée directement.

Voici le résultat :

```
i = 0
continue inner
i = 1
continue inner
i = 2
continue
i = 3
break
i = 4
continue inner
i = 5
```

```
continue inner
i = 6
continue inner
i = 7
continue outer
i = 8
break outer
```

Si l'instruction **break outer** n'était pas là, il n'y aurait aucun moyen de se brancher à l'extérieur de la boucle extérieure depuis la boucle intérieure, puisque **break** utilisé seul ne permet de sortir que de la boucle la plus interne (il en est de même pour **continue**).

Évidemment, lorsque **break** a pour effet de sortir de la boucle *et* de la méthode, il est plus simple d'utiliser **return**.

Voici une démonstration des instructions étiquetées **break** et **continue** avec des boucles **while** :

```
//: c03:LabeledWhile.java
// La boucle while "étiquetée" en Java.

public class LabeledWhile {
  public static void main(String[] args) {
    int i = 0;
    outer:
    while(true) {
      prt("Outer while loop");
      while(true) {
        i++;
        prt("i = " + i);
        if(i == 1) {
          prt("continue");
          continue;
        }
        if(i == 3) {
          prt("continue outer");
          continue outer;
        }
        if(i == 5) {
          prt("break");
          break;
        }
        if(i == 7) {
          prt("break outer");
          break outer;
        }
      }
    }
  }
```

```
  static void prt(String s) {
    System.out.println(s);
  }
} ///:~
```

Les mêmes règles s'appliquent au **while** :

1. Un **continue** génère un saut en tête de la boucle courante et poursuit l'exécution ;
2. Un **continue** étiqueté génère un saut à l'étiquette puis entre à nouveau dans la boucle juste après cette étiquette ;
3. Un **break** « sort de la boucle par le bas » ;
4. Un **break** étiqueté sort de la boucle par le bas, à la fin de la boucle repérée par l'étiquette.

Le résultat de cette méthode éclaircit cela :

```
Outer while loop
i = 1
continue
i = 2
i = 3
continue outer
Outer while loop
i = 4
i = 5
break
Outer while loop
i = 6
i = 7
break outer
```

Il est important de se souvenir qu'il n'y a *qu'une seule* raison d'utiliser une étiquette en Java, c'est lorsqu'il existe plusieurs boucles imbriquées et que l'on veut utiliser **break** ou **continue** pour traverser plus d'un niveau d'itération.

Dijkstra, dans son article « Le goto considéré comme nuisible », critiquait les étiquettes, mais pas le goto lui-même. Il avait observé que le nombre de bugs semblait augmenter avec le nombre d'étiquettes d'un programme. Les étiquettes et les goto rendent difficile l'analyse statique d'un programme, car ils introduisent des cycles dans le graphe d'exécution de celui-ci. Remarquons que les étiquettes Java ne sont pas concernées par ce problème, dans la mesure où leur emplacement est contraint, et qu'elles ne peuvent pas être utilisées pour réaliser un transfert de contrôle à la demande. Il est intéressant de noter que nous nous trouvons dans un cas où une fonctionnalité du langage est rendue plus utile en en diminuant la puissance.

switch

On parle parfois de **switch** comme d'une *instruction de sélection*. L'instruction **switch** sélectionne un morceau de code parmi d'autres en se basant sur la valeur d'une expression entière. Voici sa forme :

```
switch(sélecteur-entier) {
  case valeur-entière1 : instruction; break;
  case valeur-entière2 : instruction; break;
  case valeur-entière3 : instruction; break;
  case valeur-entière4 : instruction; break;
  case valeur-entière5 : instruction; break;
        // ...
  default : instruction;
}
```

Notons dans la définition précédente que chaque instruction **case** se termine par une instruction **break**, qui provoque un saut à la fin du corps de l'instruction **switch**. Ceci est la manière habituelle de construire une instruction **switch**. Cependant **break** est optionnel. S'il n'est pas là, le code associé à l'instruction **case** suivante est exécuté, et ainsi de suite jusqu'à la rencontre d'une instruction **break**. Bien qu'on n'ait généralement pas besoin d'utiliser cette possibilité, elle peut se montrer très utile pour un programmeur expérimenté. La dernière instruction, qui suit **default**, n'a pas besoin de **break** car en fait l'exécution continue juste à l'endroit où une instruction **break** l'aurait amenée de toute façon. Il n'y a cependant aucun problème à terminer l'instruction **default** par une instruction **break** si on pense que le style de programmation est une question importante.

L'instruction **switch** est une manière propre d'implémenter une sélection multiple (c'est à dire sélectionner un certain nombre de possibilités d'exécution), mais elle requiert une expression de sélection dont le résultat soit entier, comme **int** ou **char**. Par exemple on ne peut pas utiliser une chaîne de caractères ou un flottant comme sélecteur dans une instruction **switch**. Pour les types non entiers, il faut mettre en oeuvre une série d'instructions **if**.

Voici un exemple qui crée des lettres aléatoirement, et détermine s'il s'agit d'une voyelle ou d'une consonne :

```
//: c03:VowelsAndConsonants.java
// Démonstration de l'instruction switch.

public class VowelsAndConsonants {
  public static void main(String[] args) {
    for(int i = 0; i < 100; i++) {
      char c = (char)(Math.random() * 26 + 'a');
      System.out.print(c + ": ");
      switch(c) {
      case 'a':
      case 'e':
      case 'i':
      case 'o':
      case 'u':
            System.out.println("vowel");
            break;
      case 'y':
      case 'w':
            System.out.println(
              "Sometimes a vowel");
```

```
            break;
        default:
            System.out.println("consonant");
        }
      }
    }
  } ///:~
```

Math.random( ) générant une valeur entre 0 et 1, il faut la multiplier par la borne supérieure de l'intervalle des nombres qu'on veut produire (26 pour les lettres de l'alphabet) puis ajouter un décalage correspondant à la borne inférieure.

Bien qu'il semble que la sélection s'opère ici sur un type caractère, l'instruction **switch** utilise en réalité la valeur entière du caractère. Les caractères entre guillemets simples des instructions **case** sont également interprétés comme des entiers avant la comparaison.

Remarquez la manière d'empiler les instructions **case** pour affecter un même code d'exécution à plusieurs cas d'égalité. Il faut également faire attention à terminer chaque instruction par une instruction **break**, faute de quoi le contrôle serait transféré à l'instruction correspondant au **case** suivant.

Détails de calcul :

L'instruction :

```
char c = (char)(Math.random() * 26 + 'a');
```

mérite une attention particulière. **Math.random( )** a pour résultat un **double**, par suite la valeur 26 est convertie en **double** avant que la multiplication ne soit effectuée ; cette dernière a aussi pour résultat un **double**. Ce qui signifie que '**a**' doit lui aussi être converti en **double** avant d'exécuter l'addition. Le résultat **double** est finalement transtypé en **char**.

Que fait exactement ce transtypage ? En d'autres termes, si on obtient une valeur de 29.7 et qu'on la transtype en **char**, le résultat est-il 30 ou 29 ? La réponse est dans l'exemple suivant :

```
//: c03:CastingNumbers.java
// Qu'arrive-t-il lorsqu'on transtype un flottant
// ou un double vers une valeur entière ?

public class CastingNumbers {
  public static void main(String[] args) {
    double
      above = 0.7,
      below = 0.4;
    System.out.println("above: " + above);
    System.out.println("below: " + below);
    System.out.println(
      "(int)above: " + (int)above);
    System.out.println(
      "(int)below: " + (int)below);
```

```
    System.out.println(
      "(char)('a' + above): " +
      (char)('a' + above));
    System.out.println(
      "(char)('a' + below): " +
      (char)('a' + below));
  }
} ///:~
```

Le résultat :

```
above: 0.7
below: 0.4
(int)above: 0
(int)below: 0
(char)('a' + above): a
(char)('a' + below): a
```

La réponse est donc : le transtypage d'un type **float** ou **double** vers une valeur entière entraîne une troncature dans tous les cas.

Une seconde question à propos de **Math.random( )** : cette méthode produit des nombres entre zéro et un, mais : les valeurs 0 et 1 sont-elles inclues ou exclues ? En jargon mathématique, obtient-on]0,1[, [0,1],]0,1] ou [0,1[? (les crochets « tournés vers le nombre » signifient « ce nombre est inclus », et ceux tournés vers l'extérieur « ce nombre est exclu »). À nouveau, un programme de test devrait nous donner la réponse :

```
//: c03:RandomBounds.java
// Math.random() produit-il les valeurs 0.0 et 1.0 ?

public class RandomBounds {
  static void usage() {
    System.out.println("Usage: \n\t" +
      "RandomBounds lower\n\t" +
      "RandomBounds upper");
    System.exit(1);
  }
  public static void main(String[] args) {
    if(args.length != 1) usage();
    if(args[0].equals("lower")) {
      while(Math.random() != 0.0)
        ; // Essayer encore
      System.out.println("Produced 0.0!");
    }
    else if(args[0].equals("upper")) {
      while(Math.random() != 1.0)
        ; // Essayer encore
      System.out.println("Produced 1.0!");
    }
```

```
    else
      usage();
  }
} ///:~
```

Pour lancer le programme, frapper en ligne de commande :

```
java RandomBounds lower
```

ou bien :

```
java RandomBounds upper
```

Dans les deux cas nous sommes obligés d'arrêter le programme manuellement, et il *semble* donc que **Math.random( )** ne produise jamais les valeurs 0.0 ou 1.0. Une telle expérience est décevante. Si vous remarquez [26] qu'il existe environ 262 fractions différentes en double-précision entre 0 et 1, alors la probabilité d'atteindre expérimentalement l'une ou l'autre des valeurs pourrait dépasser la vie d'un ordinateur, voire celle de l'expérimentateur. Cela ne permet pas de montrer que 0.0 *est* inclus dans les résultats de **Math.random( )**. En réalité, en jargon mathématique, le résultat est [0,1[.

Résumé

Ce chapitre termine l'étude des fonctionnalités fondamentales qu'on retrouve dans la plupart des langages de programmation : calcul, priorité des opérateurs, transtypage, sélection et itération. Vous êtes désormais prêts à aborder le monde de la programmation orientée objet. Le prochain chapitre traite de la question importante de l'initialisation et du nettoyage des objets, et le suivant du concept essentiel consistant à cacher l'implémentation.

Exercices

Les solutions des exercices sélectionnés sont dans le document électronique *The Thinking in Java Annotated Solution Guide*, disponible pour un faible coût à *www.BruceEckel.com*.

1. Dans la section « priorité » au début de ce chapitre, il y a deux expressions. Utiliser ces expressions dans un programme qui montre qu'elles produisent des résultats différents.
2. Utiliser les méthodes ternary( ) et alternative( ) dans un programme qui fonctionne.
3. À partir des sections « if-else » et « return », utiliser les méthodes test( ) et test2( ) dans un programme qui fonctionne.
4. Écrire un programme qui imprime les valeurs de un à 100.
5. Modifier l'exercice 4 de manière que le programme termine avec la valeur 47, en utilisant le mot clef **break**. Même exercice en utilisant **return**.
6. Écrire une fonction prenant pour arguments deux **String**, utiliser les comparaisons booléennes pour comparer les deux chaînes et imprimer le résultat. En plus de == et !=, tester aussi equals( ). Dans la méthode main( ), appeler la fonction avec différents objets de type **String**.
7. Écrire un programme qui génère aléatoirement 25 valeurs entières. Pour chaque va-

leur, utiliser une instruction if-then-else pour la classer (plus grande, plus petite, ou égale) par rapport à une deuxième valeur générée aléatoirement.

8. Modifier l'exercice 7 en englobant le code dans une boucle while infinie. Il devrait alors fonctionner tant qu'on ne l'interrompt pas au moyen du clavier (classiquement avec Ctrl-C).

9. Écrire un programme utilisant deux boucles **for** imbriquées ainsi que l'opérateur modulo (**%**) pour détecter et imprimer les nombres premiers (les nombres entiers qui ne sont divisibles que par eux-mêmes et l'unité).

10. Écrire une instruction **switch** qui imprime un message pour chaque **case**, la mettre dans une boucle **for** qui teste chaque **case**. Mettre un **break** après chaque **case**, tester, puis enlever les **break** et voir ce qui se passe..

[25] John Kirkham écrivait, « J'ai fait mes débuts en informatique en 1962 avec FORTRAN II sur un IBM 1620. À cette époque, pendant les années 60 et jusqu'au début des années 70, FORTRAN était un langage entièrement écrit en majuscules. Sans doute parce que beaucoup de périphériques d'entrée anciens étaient de vieux téléscripteurs utilisant le code Baudot à cinq moments (à cinq bits), sans possibilité de minuscules. La lettre 'E' dans la notation scientifique était en majuscule et ne pouvait jamais être confondue avec la base 'e' des logarithmes naturels, toujours écrite en minuscule. 'E' signifiait simplement puissance, et, naturellement, puissance de la base de numération habituellement utilisée c'est à dire puissance de 10. À cette époque également, l'octal était largement utilisé par les programmeurs. Bien que je ne l'aie jamais vu utiliser, si j'avais vu un nombre octal en notation scientifique j'aurais pensé qu'il était en base 8. Mon plus ancien souvenir d'une notation scientifique utilisant un 'e' minuscule remonte à la fin des années 70 et je trouvais immédiatement cela déroutant. Le problème apparut lorsqu'on introduisit les minuscules en FORTRAN, et non à ses débuts. En réalité il existait des fonctions qui manipulaient la base des logarithmes naturels, et elles étaient toutes en majuscules. »

[26] Chuck Allison écrivait : Le nombre total de nombres dans un système à virgule flottante est **2(M-m+1)b^(p-1) + 1** où **b** est la base (généralement 2), **p** la précision (le nombre de chiffres dans la mantisse), **M** le plus grand exposant, et **m** le plus petit. Dans la norme IEEE 754, on a : **M = 1023, m = -1022, p = 53, b = 2** d'où le nombre total de nombres est **2(1023+1022+1)2^52 = 2((2^10-1) + (2^10-1))2^52 = (2^10-1)2^54 = 2^64 - 2^54** La moitié de ces nombres (correspondant à un exposant dans l'intervalle[-1022, 0]) sont inférieurs à un en valeur absolue, et donc 1/4 de cette expression, soit 2^62 - 2^52 + 1 (approximativement 2^62) est dans l'intervalle [0,1[. Voir mon article à http://www.freshsources.com/1995006a.htm (suite de l'article).

Chapitre 4 - Initialisation & nettoyage

Depuis le début de la révolution informatique, la programmation «sans garde-fou» est la principale cause des coûts de développement excessifs.

L'*initialisation* et la *libération* d'éléments sont deux problèmes majeurs. De nombreux bogues en C surviennent lorsque le programmeur oublie d'initialiser une variable. L'utilisation de bibliothèques augmente ce risque car les utilisateurs ne savent pas toujours comment initialiser certains composants, ni même qu'ils le doivent. La phase de nettoyage ou libération pose problème dans la mesure où il est très facile d'oublier l'existence d'un élément dont on n'a plus besoin, car justement il ne nous intéresse plus. Dans ce cas, certaines ressources utilisées par un élément oublié sont conservées. Ce phénomène peut entraîner un manque de ressources (dans la majorité des cas, un manque de mémoire).

C++ a introduit la notion de *constructeur*, une méthode appelée automatiquement à la création d'un objet. Java utilise aussi les constructeurs, associé à un ramasse-miettes qui libère les ressources mémoire lorsqu'elles ne sont plus utilisées. Ce chapitre décrit les concepts d'initialisation et de libération, ainsi que leur support dans Java.

Garantie d'initialisation grâce au constructeur

Pour chaque classe il serait possible de créer une méthode **initialise( )**. Ce nom inciterait à exécuter la méthode avant d'utiliser l'objet. Malheureusement ce serait à l'utilisateur de se souvenir d'appeler cette méthode pour chaque instance. En Java, le concepteur d'une classe peut garantir son initialisation grâce à une méthode spéciale que l'on dénomme *constructeur*. Quand une classe possède un constructeur, Java l'appelle automatiquement à toute création d'objets, avant qu'ils ne puissent être utilisés. L'initialisation est donc bien garantie.

Le premier problème consiste à trouver un nom pour cette méthode ce qui entraîne deux nouveaux problèmes. Tout d'abord il pourrait y avoir un conflit avec le nom d'un attribut. Ensuite c'est à la compilation que l'appel du constructeur est vérifié. Il faut donc que le compilateur puisse décider du nom du constructeur. La solution de C++ paraît la plus simple et la plus logique, elle est donc aussi utilisée en Java. Il faut donc donner au constructeur le nom de sa classe. Il semble naturel qu'une telle méthode soit en charge de l'initialisation de la classe.

Voici une classe avec un constructeur :

```
//: c04:SimpleConstrutor.java
// Démonstration d'un constructeur.

class Rock {
  Rock() { // Ceci est un constructeur
    System.out.println("Creating Rock");
  }
}

public class SimpleConstructor {
  public static void main(String[] args) {
    for(int i = 0; i < 10; i++)
```

```
      new Rock();
  }
} ///:~
```

Quand un objet est créé :

```
new Rock();
```

de l'espace mémoire est alloué et le constructeur est appelé. L'objet sera obligatoirement initialisé avant qu'il ne puisse être manipulé.

Notez que la convention de nommage qui impose une minuscule pour la première lettre des noms de méthode ne s'applique pas aux constructeurs, leur nom devant *exactement* coïncider avec celui de la classe.

Comme les autres méthodes, un constructeur peut prendre des paramètres. Cela permet de préciser *comment* l'objet va être créé. Notre premier exemple peut facilement être modifié pour que le constructeur prenne un unique paramètre :

```
//: c04:SimpleConstructor2.java
// Les constructeurs peuvent prendre des paramètres.

class Rock2 {
  Rock2(int i) {
    System.out.println(
      "Creating Rock number " + i);
  }
}

public class SimpleConstructor2 {
  public static void main(String[] args) {
    for(int i = 0; i < 10; i++)
      new Rock2(i);
  }
} ///:~
```

Les paramètres des constructeurs permettent de personnaliser la création des objets. Par exemple, si la classe **Tree** (arbre) a un constructeur avec un paramètre de type **int** qui détermine la hauteur de l'arbre, un objet **Tree** se crée de la façon suivante :

```
Tree t = new Tree(12);  // arbre de 12 pieds
```

De plus, si **Tree(int)** est le seul constructeur, le compilateur ne permettra pas de créer un objet **Tree** d'une autre façon.

La notion de constructeur élimine toute une catégorie d'erreurs et rend plus aisée la lecture du code. Dans le fragment de code précédent, par exemple, il n'y a pas d'appel explicite à une certaine méthode **initialise( )** qui serait conceptuellement séparée de la définition. En Java, définition et initialisation sont des concepts unifiés - il est impossible d'avoir l'un sans l'autre.

Un constructeur est une méthode très spéciale de par le fait qu'elle n'a pas de valeur de retour. Cela n'a absolument rien à voir avec le type de retour **void**, qui signifie qu'une méthode ne renvoie

rien mais qu'il aurait tout à fait été possible de lui faire renvoyer autre chose. Les constructeurs ne retournent rien et on n'a pas le choix. S'il y avait une valeur de retour, et si l'on pouvait choisir son type, le compilateur devrait trouver une utilisation à cette valeur.

Surcharge de méthodes

L'un des points les plus importants de tout langage de programmation est le nommage. Créer un objet revient à donner un nom à un emplacement mémoire. Une méthode est un nom d'action. En utilisant des noms pour décrire un système, on simplifie la lecture et la modification des programmes. Cela s'apparente à l'écriture en prose dont le but est de communiquer avec le lecteur.

On se réfère à tous les objets et méthodes en utilisant leurs noms. Des noms biens choisis rendent la compréhension du code plus aisée, tant pour le développeur que pour les relecteurs.

Les difficultés commencent lorsque l'on essaie d'exprimer les nuances subtiles du langage humain dans un langage de programmation. Très souvent, un même mot a plusieurs sens, on parle de *surcharge*. Cette notion est très pratique pour exprimer les différences triviales de sens. On dit « laver la chemise » , « laver la voiture » et « laver le chien » . Cela paraîtrait absurde d'être obligé de dire « laverChemise la chemise » , « laverVoiture la voiture » et « laverChien le chien » pour que l'auditoire puisse faire la distinction entre ces actions. La plupart des langages humains sont redondants à tel point que même sans entendre tous les mots, il est toujours possible de comprendre le sens d'une phrase. Nous n'avons aucunement besoin d'identifiants uniques, le sens peut être déduit du contexte.

La plupart des langages de programmation (C en particulier) imposent un nom unique pour chaque fonction. Ils ne permettent pas d'appeler une fonction **affiche( )** pour afficher des entiers et une autre appelée **affiche( )** pour afficher des flottants, chaque fonction doit avoir un nom unique.

En Java (et en C++), un autre facteur impose la surcharge de noms de méthodes : les constructeurs. Comme le nom d'un constructeur est déterminé par le nom de la classe, il ne peut y avoir qu'un seul nom de constructeur. Mais que se passe-t-il quand on veut créer un objet de différentes façons ? Par exemple, supposons que l'on construise une classe qui peut s'initialiser de façon standard ou en lisant des informations depuis un fichier. Nous avons alors besoin de deux constructeurs, l'un ne prenant pas de paramètre (le constructeur *par défaut*, aussi appelé le constructeur *sans paramètre / no-arg*), et un autre prenant une **Chaîne / String** comme paramètre, qui représente le nom du fichier depuis lequel on souhaite initialiser l'objet. Tous les deux sont des constructeurs, ils doivent donc avoir le même nom, le nom de la classe. Cela montre que la *surcharge de méthode* est essentielle pour utiliser le même nom de méthode pour des utilisations sur différents types de paramètres. Et si la surcharge de méthode est obligatoire pour les constructeurs, elle est aussi très pratique pour les méthodes ordinaires.

L'exemple suivant montre à la fois une surcharge de constructeur et une surcharge de méthode ordinaire :

```
//: c04:Overloading.java
// Exemple de surcharge de constructeur
// et de méthode ordinaire.
import java.util.*;

class Tree {
  int height;
```

```
  Tree() {
    prt("Planting a seedling");  // Planter une jeune pousse
    height = 0;
  }
  Tree(int i) {
    prt("Creating new Tree that is " // Création d'un Arbre
        + i + " feet tall"); // de i pieds de haut
    height = i;
  }
  void info() {
    prt("Tree is " + height // L'arbre mesure x pieds
        + " feet tall");
  }
  void info(String s) {
    prt(s + ": Tree is "      // valeur de s :  L'arbre mesure x pieds
        + height + " feet tall");
  }
  static void prt(String s) {
    System.out.println(s);
  }
}

public class Overloading {
  public static void main(String[] args) {
    for(int i = 0; i < 5; i++) {
      Tree t = new Tree(i);
      t.info();
      t.info("overloaded method");
    }
    // constructeur surchargé :
    new Tree();
  }
} ///:~
```

Un objet **Tree** peut être créé soit en tant que jeune pousse, sans fournir de paramètre, soit en tant que plante poussée en pépinière, en donnant une hauteur initiale. Pour permettre ceci, il y a deux constructeurs, l'un ne prend pas de paramètre (on appelle les constructeurs sans paramètre des *constructeurs par défaut* [27]) et un deuxième qui prend la hauteur initiale de l'arbre.

Il est aussi possible d'appeler la méthode **info( )** de plusieurs façons. Par exemple, avec un paramètre **String** si un message supplémentaire est désiré, ou sans paramètre lorsqu'il n'y a rien d'autre à dire. Cela paraîtrait étrange de donner deux noms distincts à ce qui est manifestement le même concept. Heureusement, la surcharge de méthode permet l'utilisation du même nom pour les deux.

Différencier les méthodes surchargées

Quand deux méthodes ont le même nom, comment Java peut-il décider quelle méthode est

demandée ? Il y a une règle toute simple : chaque méthode surchargée doit prendre une liste unique de types de paramètres.

Lorsqu'on y pense, cela paraît tout à fait sensé : comment le développeur lui-même pourrait-il choisir entre deux méthodes du même nom, autrement que par le type des paramètres ?

Une différence dans l'ordre des paramètres est suffisante pour distinguer deux méthodes (cette approche n'est généralement pas utilisée car elle donne du code difficile à maintenir.) :

```
//: c04:OverloadingOrder.java
// Surcharge basée sur l'ordre
// des paramètres.

public class OverloadingOrder {
  static void print(String s, int i) {
    System.out.println(
      "String: " + s +
      ", int: " + i);
  }
  static void print(int i, String s) {
    System.out.println(
      "int: " + i +
      ", String: " + s);
  }
  public static void main(String[] args) {
    print("String first", 11);
    print(99, "Int first");
  }
} ///:~
```

Les deux méthodes **print( )** ont les mêmes paramètres, mais dans un ordre différent, et c'est ce qui les différencie.

Surcharge avec types de base

Un type de base peut être promu automatiquement depuis un type plus petit vers un plus grand ; ceci peut devenir déconcertant dans certains cas de surcharge. L'exemple suivant montre ce qui se passe lorsqu'un type de base est passé à une méthode surchargée :

```
//: c04:PrimitiveOverloading.java
// Promotion des types de base et surcharge.

public class PrimitiveOverloading {
  // boolean ne peut pas être converti automatiquement
  static void prt(String s) {
    System.out.println(s);
  }

  void f1(char x) { prt("f1(char)"); }
  void f1(byte x) { prt("f1(byte)"); }
```

```
void f1(short x) { prt("f1(short)"); }
void f1(int x) { prt("f1(int)"); }
void f1(long x) { prt("f1(long)"); }
void f1(float x) { prt("f1(float)"); }
void f1(double x) { prt("f1(double)"); }

void f2(byte x) { prt("f2(byte)"); }
void f2(short x) { prt("f2(short)"); }
void f2(int x) { prt("f2(int)"); }
void f2(long x) { prt("f2(long)"); }
void f2(float x) { prt("f2(float)"); }
void f2(double x) { prt("f2(double)"); }

void f3(short x) { prt("f3(short)"); }
void f3(int x) { prt("f3(int)"); }
void f3(long x) { prt("f3(long)"); }
void f3(float x) { prt("f3(float)"); }
void f3(double x) { prt("f3(double)"); }

void f4(int x) { prt("f4(int)"); }
void f4(long x) { prt("f4(long)"); }
void f4(float x) { prt("f4(float)"); }
void f4(double x) { prt("f4(double)"); }

void f5(long x) { prt("f5(long)"); }
void f5(float x) { prt("f5(float)"); }
void f5(double x) { prt("f5(double)"); }

void f6(float x) { prt("f6(float)"); }
void f6(double x) { prt("f6(double)"); }

void f7(double x) { prt("f7(double)"); }

void testConstVal() {
  prt("Testing with 5");
  f1(5);f2(5);f3(5);f4(5);f5(5);f6(5);f7(5);
}
void testChar() {
  char x = 'x';
  prt("char argument:");
  f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
}
void testByte() {
  byte x = 0;
  prt("byte argument:");
  f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
}
```

```
  void testShort() {
    short x = 0;
    prt("short argument:");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
  }
  void testInt() {
    int x = 0;
    prt("int argument:");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
  }
  void testLong() {
    long x = 0;
    prt("long argument:");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
  }
  void testFloat() {
    float x = 0;
    prt("float argument:");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
  }
  void testDouble() {
    double x = 0;
    prt("double argument:");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
  }
  public static void main(String[] args) {
    PrimitiveOverloading p =
      new PrimitiveOverloading();
    p.testConstVal();
    p.testChar();
    p.testByte();
    p.testShort();
    p.testInt();
    p.testLong();
    p.testFloat();
    p.testDouble();
  }
} ///:~
```

En regardant la sortie du programme, on voit que la constante 5 est considérée comme un **int**. Lorsqu'une méthode surchargée utilisant un **int** est disponible, elle est utilisée. Dans tous les autres cas, si un type de données est plus petit que l'argument de la méthode, le type est promu. **char** est légèrement différent, comme il ne trouve pas une correspondance exacte, il est promu vers un **int**.

```
//: c04:Demotion.java
// Types de base déchus et surcharge.

public class Demotion {
```

```
  static void prt(String s) {
    System.out.println(s);
  }

  void f1(char x) { prt("f1(char)"); }
  void f1(byte x) { prt("f1(byte)"); }
  void f1(short x) { prt("f1(short)"); }
  void f1(int x) { prt("f1(int)"); }
  void f1(long x) { prt("f1(long)"); }
  void f1(float x) { prt("f1(float)"); }
  void f1(double x) { prt("f1(double)"); }

  void f2(char x) { prt("f2(char)"); }
  void f2(byte x) { prt("f2(byte)"); }
  void f2(short x) { prt("f2(short)"); }
  void f2(int x) { prt("f2(int)"); }
  void f2(long x) { prt("f2(long)"); }
  void f2(float x) { prt("f2(float)"); }

  void f3(char x) { prt("f3(char)"); }
  void f3(byte x) { prt("f3(byte)"); }
  void f3(short x) { prt("f3(short)"); }
  void f3(int x) { prt("f3(int)"); }
  void f3(long x) { prt("f3(long)"); }

  void f4(char x) { prt("f4(char)"); }
  void f4(byte x) { prt("f4(byte)"); }
  void f4(short x) { prt("f4(short)"); }
  void f4(int x) { prt("f4(int)"); }

  void f5(char x) { prt("f5(char)"); }
  void f5(byte x) { prt("f5(byte)"); }
  void f5(short x) { prt("f5(short)"); }

  void f6(char x) { prt("f6(char)"); }
  void f6(byte x) { prt("f6(byte)"); }

  void f7(char x) { prt("f7(char)"); }

  void testDouble() {
    double x = 0;
    prt("double argument:");
    f1(x);f2((float)x);f3((long)x);f4((int)x);
    f5((short)x);f6((byte)x);f7((char)x);
  }
  public static void main(String[] args) {
    Demotion p = new Demotion();
```

```
    p.testDouble();
  }
} ///:~
```

Ici, les méthodes prennent des types de base plus restreints. Si les paramètres sont d'un type plus grand, if faut les *caster (convertir)* vers le type requis en utilisant le nom du type entre parenthèses. Sinon, le compilateur donnera un message d'erreur.

Il est important de noter qu'il s'agit d'une *conversion vers un type plus petit,* ce qui signifie que des informations peuvent être perdues pendant la conversion. C'est d'ailleurs pour cette raison que le compilateur force une conversion explicite.

Surcharge sur la valeur de retour

Il est fréquent de se demander «Pourquoi seulement les noms de classes et la liste des paramètres des méthodes ? Pourquoi ne pas aussi distinguer entre deux méthodes en se basant sur leur type de retour ?» Par exemple, ces deux méthodes, qui ont le même nom et les mêmes arguments, peuvent facilement être distinguées l'une de l'autre :

```
void f() {}
int f() {}
```

Cela fonctionne bien lorsque le compilateur peut déterminer le sens sans équivoque depuis le contexte, comme dans **int x = f( )**. Par contre, on peut utiliser une méthode et ignorer sa valeur de retour. On se réfère souvent à cette action comme *appeler une méthode pour ses effets de bord* puisqu'on ne s'intéresse pas à la valeur de retour mais aux autres effets que cet appel de méthode génère. Donc, si on appelle la méthode comme suit :

```
f();
```

Comment Java peut-il déterminer quelle méthode **f( )** doit être exécutée ? Et comment quelqu'un lisant ce code pourrait-il le savoir ? A cause de ce genre de difficultés, il est impossible d'utiliser la valeur de retour pour différencier deux méthodes Java surchargées.

Constructeurs par défaut

Comme mentionné précédemment, un constructeur par défaut (c.a.d un constructeur «no-arg») est un constructeur sans argument, utilisé pour créer des « objets de base ». Si une classe est créée sans constructeur, le compilateur créé automatiquement un constructeur par défaut. Par exemple :

```
//: c04:DefaultConstructor.java

class Bird {
  int i;
}

public class DefaultConstructor {
  public static void main(String[] args) {
    Bird nc = new Bird(); // défaut !
  }
```

```
} ///:~
```

La ligne

```
new Bird();
```

crée un nouvel objet et appelle le constructeur par défaut, même s'il n'était pas défini explicitement. Sans lui, il n'y aurait pas de méthode à appeler pour créer cet objet. Par contre, si au moins un constructeur est défini (avec ou sans argument), le compilateur n'en synthétisera *pas* un :

```
class Bush {
  Bush(int i) {}
  Bush(double d) {}
}
```

Maintenant si on écrit :

```
new Bush();
```

le compilateur donnera une erreur indiquant qu'aucun constructeur ne correspond. C'est comme si lorsqu'aucun constructeur n'est fourni, le compilateur dit «Il faut un constructeur, je vais en créer un.» Alors que s'il existe un constructeur, le compilateur dit «Il y a un constructeur donc le développeur sait se qu'il fait; s'il n'a pas défini de constructeur par défaut c'est qu'il ne désirait pas qu'il y en ait un.»

Le mot-clé this

Lorsqu'il existe deux objets **a** et **b** du même type , il est intéressant de se demander comment on peut appeler une méthode **f( )** sur ces deux objets :

```
class Banana { void f(int i) { /* ... */ } }
Banana a = new Banana(), b = new Banana();
a.f(1);
b.f(2);
```

S'il y a une unique méthode **f( )**, comment cette méthode peut-elle savoir si elle a été appelée sur l'objet **a** ou **b** ?

Pour permettre au développeur d'écrire le code dans une syntaxe pratique et orienté objet dans laquelle on «envoie un message vers un objet,» le compilateur effectue un travail secret pour le développeur. Il y a un premier paramètre caché passé à la méthode **f( )**, et ce paramètre est une référence vers l'objet en train d'être manipulé. Les deux appels de méthode précédents correspondent donc à ceci :

```
Banana.f(a,1);
Banana.f(b,2);
```

Ce travail est interne et il est impossible d'écrire des expressions de ce type directement en espérant que le compilateur les acceptera, mais cela donne une idée de ce qui se passe.

Supposons maintenant que l'on est à l'intérieur d'une méthode et que l'on désire obtenir une référence sur l'objet courant. Comme cette référence est passée en tant que paramètre *caché* par le

compilateur, il n'y a pas d'identificateur pour elle. Cette pour cette raison que le mot clé **this** existe. **this** - qui ne peut être utilisé qu'à l'intérieur d'une méthode - est une référence sur l'objet pour lequel cette méthode à été appelée. On peut utiliser cette référence comme tout autre référence vers un objet. Il n'est toutefois pas nécessaire d'utiliser **this** pour appeler une méthode de la classe courante depuis une autre méthode de la classe courante ; il suffit d'appeler cette méthode. La référence **this** est automatiquement utilisée pour l'autre méthode. On peut écrire :

```
class Apricot {
  void pick() { /* ... */ }
  void pit() { pick(); /* ... */ }
}
```

A l'intérieur de **pit( )**, on *pourrait* écrire **this.pick( )** mais ce n'est pas nécessaire. Le compilateur le fait automatiquement pour le développeur. Le mot-clé **this** est uniquement utilisé pour les cas spéciaux dans lesquels on doit utiliser explicitement une référence sur l'objet courant. Par exemple, il est couramment utilisé en association avec **return** quand on désire renvoyer une référence sur l'objet courant :

```
//: c04:Leaf.java
// Utilisation simple du mot-clé "this".

public class Leaf {
  int i = 0;
  Leaf increment() {
    i++;
    return this;
  }
  void print() {
    System.out.println("i = " + i);
  }
  public static void main(String[] args) {
    Leaf x = new Leaf();
    x.increment().increment().increment().print();
  }
} ///:~
```

Puisque **increment( )** renvoie une référence vers l'objet courant par le biais du mot-clé **this**, on peut facilement appeler plusieurs opérations successivement sur le même objet.

Appeler un constructeur depuis un autre constructeur

Quand une classe possède plusieurs constructeurs, il peut être utile d'appeler un constructeur depuis un autre pour éviter de la duplication de code. C'est possible grâce au mot-clé **this**.

En temps normal, **this** signifie «cet objet» ou «l'objet courant,» et renvoie une référence sur l'objet courant. Dans un constructeur, le mot-clé **this** prend un sens différent quand on lui passe une liste de paramètres : il signifie un appel explicite au constructeur qui correspond à cette liste de paramètres. Cela donne un moyen très simple d'appeler d'autres constructeurs :

```
//: c04:Flower.java
```

```
// Appel de constructeurs avec "this."

public class Flower {
  int petalCount = 0;
  String s = new String("null");
  Flower(int petals) {
    petalCount = petals;
    // Constructeur avec un unique paramètre int
    System.out.println(
      "Constructor w/ int arg only, petalCount= "
      + petalCount);
  }
  Flower(String ss) {
    // Constructeur avec un unique paramètre String
    System.out.println(
      "Constructor w/ String arg only, s=" + ss);
    s = ss;
  }
  Flower(String s, int petals) {
    this(petals);
//!    this(s); // Impossible d'en appeler deux !
    this.s = s; // Autre usage de "this"
    System.out.println("String & int args");
  }

  // Constructeur par défaut
  Flower() {
    this("hi", 47);
    System.out.println(
      "default constructor (no args)");
  }
  void print() {
//!    this(11); // Pas à l'intérieur d'une méthode normale !
    System.out.println(
      "petalCount = " + petalCount + " s = "+ s);
  }
  public static void main(String[] args) {
    Flower x = new Flower();
    x.print();
  }
} ///:~
```

Le constructeur **Flower(String s, int petals)** montre qu'on peut appeler un constructeur en utilisant **this**, mais pas deux. De plus, l'appel au constructeur doit absolument être la première instruction sinon le compilateur donnera un message d'erreur.

Cet exemple montre aussi un usage différent du mot-clé **this**. Les noms du paramètre **s** et du membre de données **s** étant les mêmes, il y a ambiguïté. On la résoud en utilisant **this.s** pour se ré-

férer au membre de donnés. Cette forme est très courante en Java et utilisée fréquemment dans ce livre.

Dans la méthode **print( )** on peut voir que le compilateur ne permet pas l'appel d'un constructeur depuis toute autre méthode qu'un constructeur.

La signification de static

En pensant au mot-clé **this**, on comprend mieux le sens de rendre une méthode **static**s. Cela signifie qu'il n'y a pas de **this** pour cette méthode. Il est impossible d'appeler une méthode non-**static** depuis une méthode **static** [28] (par contre, l'inverse est possible), et il est possible d'appeler une méthode **static** sur la classe elle-même, sans aucun objet. En fait, c'est principalement la raison de l'existence des méthodes **static**. C'est l'équivalent d'une fonction globale en C. Sauf que les fonctions globales sont interdites en Java, et ajouter une méthode **static** dans une classe lui permet d'accéder à d'autres méthodes **static** ainsi qu'aux membres **static**.

Certaines personnes argumentent que les méthodes **static** ne sont pas orientées objet puisqu'elles ont la sémantique des fonctions globales ; avec une méthode **static** on n'envoie pas un message vers un objet, puisqu'il n'y a pas de **this**. C'est probablement un argument valable, et si vous utilisez *beaucoup* de méthodes statiques vous devriez repenser votre stratégie. Pourtant, les méthodes **static**s sont utiles et il y a des cas où on en a vraiment besoin. On peut donc laisser les théoriciens décider si oui ou non il s'agit de vraie programmation orientée objet. D'ailleurs, même Smalltalk a un équivalent avec ses «méthodes de classe.»

Nettoyage : finalisation et ramasse-miettes

Les programmeurs connaissent l'importance de l'initialisation mais oublient souvent celle du nettoyage. Après tout, qui a besoin de nettoyer un **int** ? Cependant, avec des bibliothèques, simplement oublier un objet après son utilisation n'est pas toujours sûr. Bien entendu, Java a un ramasse-miettes pour récupérer la mémoire prise par des objets qui ne sont plus utilisés. Considérons maintenant un cas très particulier. Supposons que votre objet alloue une zone de mémoire spéciale sans utiliser **new**. Le ramasse-miettes ne sait récupérer que la mémoire allouée *avec* **new**, donc il ne saura pas comment récupérer la zone «spéciale» de mémoire utilisée par l'objet. Pour gérer ce cas, Java fournit une méthode appelée **finalize( )** qui peut être définie dans votre classe. Voici comment c'est *supposé* marcher. Quand le ramasse-miettes est prêt à libérer la mémoire utilisée par votre objet, il va d'abord appeler **finalize( )** et ce n'est qu'à la prochaine passe du ramasse-miettes que la mémoire de l'objet est libérée. En choisissant d'utiliser **finalize( )**, on a la possibilité d'effectuer d'importantes tâches de nettoyage *à l'exécution du ramasse-miettes*.

C'est un piège de programmation parce que certains programmeurs, particulièrement les programmeurs C++, risquent au début de confondre **finalize( )** avec le *destructeur* de C++ qui est une fonction toujours appelée quand un objet est détruit. Cependant il est important ici de faire la différence entre C++ et Java, car en C++ *les objets sont toujours détruits* (dans un programme sans bug), alors qu'en Java les objets ne sont pas toujours récupérés par le ramasse-miettes. Dit autrement :

Le mécanisme de ramasse-miettes n'est pas un mécanisme de destruction.

Si vous vous souvenez de cette règle de base, il n'y aura pas de problème. Cela veut dire que si une opération doit être effectuée avant la disparition d'un objet, celle-ci est à la charge du développeur. Java n'a pas de mécanisme équivalent au destructeur, il est donc nécessaire de créer une

méthode ordinaire pour réaliser ce nettoyage. Par exemple, supposons qu'un objet se dessine à l'écran pendant sa création. Si son image n'est pas effacée explicitement de l'écran, il se peut qu'elle ne le soit jamais. Si l'on ajoute une fonctionnalité d'effacement dans **finalize( )**, alors l'image sera effacée de l'écran si l'objet est récupéré par le ramasse-miette, sinon l'image restera. Il y a donc une deuxième règle à se rappeler :

Les objets peuvent ne pas être récupérés par le ramasse-miettes.

Il se peut que la mémoire prise par un objet ne soit jamais libérée parce que le programme n'approche jamais la limite de mémoire qui lui a été attribuée. Si le programme se termine sans que le ramasse-miettes n'ait jamais libéré la mémoire prise par les objets, celle-ci sera rendue *en masse* (NDT : en français dans le texte) au système d'exploitation au moment où le programme s'arrête. C'est une bonne chose, car le ramasse-miettes implique un coût supplémentaire et s'il n'est jamais appelé, c'est autant d'économisé.

A quoi sert finalize( ) ?

A ce point, on peut croire qu'il ne faudrait pas utiliser **finalize( )** comme méthode générale de nettoyage. A quoi sert-elle alors ?

Une troisième règle stipule :

Le ramasse-miettes ne s'occupe que de la mémoire.

C'est à dire que la seule raison d'exister du ramasse-miettes est de récupérer la mémoire que le programme n'utilise plus. Par conséquent, toute activité associée au ramasse-miettes, la méthode **finalize( )** en particulier, doit se concentrer sur la mémoire et sa libération.

Est-ce que cela veut dire que si un objet contient d'autres objets, **finalize( )** doit libérer ces objets explicitement ? La réponse est... non. Le ramasse-miettes prend soin de libérer tous les objets quelle que soit la façon dont ils ont été créés. Il se trouve que l'on a uniquement besoin de **finalize( )** dans des cas bien précis où un objet peut allouer de la mémoire sans créer un autre objet. Cependant vous devez vous dire que tout est objet en Java, donc comment est-ce possible ?

Il semblerait que **finalize( )** ait été introduit parce qu'il est possible d'allouer de la mémoire à-la-C en utilisant un mécanisme autre que celui proposé normalement par Java. Cela arrive généralement avec des *méthodes natives*, qui sont une façon d'appeler du code non-Java en Java (les méthodes natives sont expliquées en Appendice B). C et C++ sont les seuls langages actuellement supportés par les méthodes natives, mais comme elles peuvent appeler des routines écrites avec d'autres langages, il est en fait possible d'appeler n'importe quoi. Dans ce code non-Java, on peut appeler des fonctions de la famille de **malloc( )** en C pour allouer de la mémoire, et à moins qu'un appel à **free( )** ne soit effectué cette mémoire ne sera pas libérée, provoquant une «fuite». Bien entendu, **free( )** est une fonction C et C++, ce qui veut dire qu'elle doit être appelée dans une méthode native dans le **finalize( )** correspondant.

Le nettoyage est impératif

Pour nettoyer un objet, son utilisateur doit appeler une méthode de nettoyage au moment où celui-ci est nécessaire. Cela semble assez simple, mais se heurte au concept de destructeur de C++. En C++, tous les objets sont, ou plutôt *devraient être*, détruits. Si l'objet C++ est créé localement (c'est à dire sur la pile, ce qui n'est pas possible en Java), alors la destruction se produit à la fermeture de la portée dans laquelle l'objet a été créé. Si l'objet a été créé par **new** (comme en Java) le

destructeur est appelé quand le programmeur appelle l'opérateur C++ **delete** (cet opérateur n'existe pas en Java). Si le programmeur C++ oublie d'appeler **delete**, le destructeur n'est jamais appelé et l'on obtient une fuite mémoire. De plus les membres de l'objet ne sont jamais nettoyé non plus. Ce genre de bogue peut être très difficile à repérer.

Contrairement à C++, Java ne permet pas de créer des objets locaux, **new** doit toujours être utilisé. Cependant Java n'a pas de «delete» pour libérer l'objet car le ramasse-miettes se charge automatiquement de récupérer la mémoire. Donc d'un point de vue simplistique, on pourrait dire qu'à cause du ramasse-miettes, Java n'a pas de destructeur. Cependant à mesure que la lecture de ce livre progresse, on s'aperçoit que la présence d'un ramasse-miettes ne change ni le besoin ni l'utilité des destructeurs (de plus, **finalize( )** ne devrait jamais être appelé directement, ce n'est donc pas une bonne solution pour ce problème). Si l'on a besoin d'effectuer des opérations de nettoyage autre que libérer la mémoire, il est *toujours* nécessaire d'appeler explicitement la méthode correspondante en Java, ce qui correspondra à un destructeur C++ sans être aussi pratique.

Une des utilisations possibles de **finalize( )** est l'observation du ramasse-miettes. L'exemple suivant montre ce qui se passe et résume les descriptions précédentes du ramasse-miettes :

```
//: c04:Garbage.java
// Démonstration du ramasse-miettes
// et de la finalisation

class Chair {
  static boolean gcrun = false;
  static boolean f = false;
  static int created = 0;
  static int finalized = 0;
  int i;
  Chair() {
    i = ++created;
    if(created == 47)
      System.out.println("Created 47");
  }
  public void finalize() {
    if(!gcrun) {
      // Premier appel de finalize() :
      gcrun = true;
      System.out.println(
        "Beginning to finalize after " +
        created + " Chairs have been created");
    }
    if(i == 47) {
      System.out.println(
        "Finalizing Chair #47, " +
        "Setting flag to stop Chair creation");
      f = true;
    }
    finalized++;
    if(finalized >= created)
```

```
        System.out.println(
          "All " + finalized + " finalized");
      }
    }

    public class Garbage {
      public static void main(String[] args) {
        // Tant que le flag n'a pas été levé,
        // construire des objets Chair et String:
        while(!Chair.f) {
          new Chair();
          new String("To take up space");
        }
        System.out.println(
          "After all Chairs have been created:\n" +
          "total created = " + Chair.created +
          ", total finalized = " + Chair.finalized);
        // Arguments optionnels pour forcer
        // la finalisation et l'exécution du ramasse-miettes :
        if(args.length > 0) {
          if(args[0].equals("gc") ||
             args[0].equals("all")) {
            System.out.println("gc():");
            System.gc();
          }
          if(args[0].equals("finalize") ||
             args[0].equals("all")) {
            System.out.println("runFinalization():");
            System.runFinalization();
          }
        }
        System.out.println("bye!");
      }
    } ///:~
```

Le programme ci-dessus crée un grand nombre d'objets **Chair** et, à un certain point après que le ramasse-miettes ait commencé à s'exécuter, le programme arrête de créer des **Chair**s. Comme le ramasse-miettes peut s'exécuter n'importe quand, on ne sait pas exactement à quel moment il se lance, il existe donc un *flag* appelé **gcrun** qui indique si le ramasse-miettes a commencé son exécution. Un deuxième *flag* **f** est le moyen pour **Chair** de prévenir la boucle **main( )** qu'elle devrait arrêter de fabriquer des objets. On lève ces deux *flag*s dans **finalize( )**, qui est appelé pendant l'exécution du ramasse-miettes.

Deux autres variables **statiques**, **created** and **finalized**, enregistre le nombre d'objets **Chair** créés par rapport au nombre réclamé par le ramasse-miettes. Enfin, chaque objet **Chair** contient sa propre version (non statique) de l'**int i** pour savoir quel est son numéro. Quand l'objet **Chair** numéro 47 est réclamé, le flag est mis à **true** pour arrêter la création des objets **Chair**.

Tout ceci se passe dans le **main( )**, dans la boucle

```
while(!Chair.f) {
    new Chair();
    new String("To take up space");
  }
```

On peut se demander comment cette boucle va se terminer puisque rien dans la boucle ne change la valeur de **Chair.f**. Cependant, **finalize( )** le fera au moment de la réclamation du numéro 47.

La création d'un objet **String** à chaque itération représente simplement de l'espace mémoire supplémentaire pour inciter le ramasse-miettes à s'exécuter, ce qu'il fera dès qu'il se sentira inquiet pour le montant de mémoire disponible.

A l'exécution du programme, l'utilisateur fournit une option sur la ligne de commande : «gc,» «finalize,» ou «all». Le paramètre «gc» permet l'appel de la méthode **System.gc( )** (pour forcer l'exécution du ramasse-miettes). «finalize» permet d'appeler **System.runFinalization( )** ce qui, en théorie, fait que tout objet non finalisé soit finalisé. Enfin, «all» exécute les deux méthodes.

Le comportement de ce programme et celui de la version de la première édition de cet ouvrage montrent que la question du ramasse-miettes et de la finalisation a évolué et qu'une grosse part de cette évolution s'est passée en coulisse. En fait, il est possible que le comportement du programme soit tout à fait différent lorsque vous lirez ces lignes.

Si **System.gc( )** est appelé, alors la finalisation concerne tous les objets. Ce n'était pas forcément le cas avec les implémentations précédentes du JDK bien que la documentation dise le contraire. De plus, il semble qu'appeler **System.runFinalization( )** n'ait aucun effet.

Cependant, on voit que toutes les méthodes de finalisation sont exécutées seulement dans le cas où **System.gc( )** est appelé après que tous les objets aient été créés et mis à l'écart. Si **System.gc( )** n'est pas appelé, seulement certains objets seront finalisés. En Java 1.1, la méthode **System.runFinalizersOnExit( )** fut introduite pour que les programmes puissent exécuter toutes les méthodes de finalisation lorsqu'ils se terminent, mais la conception était boguée et la méthode a été classée *deprecated*. C'est un indice supplémentaire qui montre que les concepteurs de Java ont eu de nombreux démêlés avec le problème du ramasse-miettes et de la finalisation. Il est à espérer que ces questions ont été réglées dans Java 2.

Le programme ci-dessus montre que les méthodes de finalisation sont toujours exécutées mais seulement si le programmeur force lui-même l'appel. Si on ne force pas l'appel de **System.gc( )**, le résultat ressemblera à ceci :

```
Created 47
Beginning to finalize after 3486 Chairs have been created
Finalizing Chair #47, Setting flag to stop Chair creation
After all Chairs have been created:
total created = 3881, total finalized = 2684
bye!
```

Toutes les méthodes de finalisation ne sont donc pas appelées à la fin du programme. Ce n'est que quand **System.gc( )** est appelé que tous les objets qui ne sont plus utilisés seront finalisés et détruits.

Il est important de se souvenir que ni le ramasse-miettes, ni la finalisation ne sont garantis. Si

la machine virtuelle Java (JVM) ne risque pas de manquer de mémoire, elle ne perdra (légitimement) pas de temps à en récupérer grâce au ramasse-miettes.

La «death condition»

En général, on ne peut pas compter sur un appel à **finalize( )**, et il est nécessaire de créer des fonctions spéciales de nettoyage et de les appeler explicitement. Il semblerait donc que **finalize( )** ne soit utile que pour effectuer des tâches de nettoyage mémoire très spécifiques dont la plupart des programmeurs n'aura jamais besoin. Cependant, il existe une très intéressante utilisation de **finalize( )** qui ne nécessite pas que son appel soit garanti. Il s'agit de la vérification de la *death condition* [29] d'un objet (état d'un objet à sa destruction).

Au moment où un objet n'est plus intéressant, c'est à dire lorsqu'il est prêt à être réclamé par le ramasse-miettes, cet objet doit être dans un état où sa mémoire peut être libérée sans problème. Par exemple, si l'objet représente un fichier ouvert, celui-ci doit être fermé par le programmeur avant que la mémoire prise par l'objet ne soit réclamée. Si certaines parties de cet objet n'ont pas été nettoyées comme il se doit, il s'agit d'un bogue du programme qui peut être très difficile à localiser. L'intérêt de **finalize( )** est qu'il est possible de l'utiliser pour découvrir cet état de l'objet, même si cette méthode n'est pas toujours appelée. Si une des finalisations trouve le bogue, alors le problème est découvert et c'est ce qui compte vraiment après tout.

Voici un petit exemple pour montrer comment on peut l'utiliser :

```
//: c04:DeathCondition.java
// Comment utiliser finalize() pour détecter les objets qui
// n'ont pas été nettoyés correctement.

class Book {
  boolean checkedOut = false;
  Book(boolean checkOut) {
    checkedOut = checkOut;
  }
  void checkIn() {
    checkedOut = false;
  }
  public void finalize() {
    if(checkedOut)
      System.out.println("Error: checked out");
  }
}

public class DeathCondition {
  public static void main(String[] args) {
    Book novel = new Book(true);
    // Nettoyage correct :
    novel.checkIn();
    // Perd la référence et oublie le nettoyage :
    new Book(true);
    // Force l'exécution du ramasse-miettes et de la finalisation :
```

```
    System.gc();
  }
} ///:~
```

Ici, la «death condition» est le fait que tous les objets de type **Book** doivent être «rendus» (checked in) avant d'être récupéré par le ramasse-miettes, mais dans la fonction **main( )** une erreur de programmation fait qu'un de ces livres n'est pas rendu. Sans **finalize( )** pour vérifier la «death condition», cela pourrait s'avérer un bogue difficile à trouver.

Il est important de noter l'utilisation de **System.gc( )** pour forcer l'exécution de la finalisation (en fait, il est utile de le faire pendant le développement du programme pour accélérer le débogage). Cependant même si **System.gc()** n'est pas appelé, il est très probable que le livre (**Book**) perdu soit découvert par plusieurs exécutions successives du programme (en supposant que suffisamment de mémoire soit alloué pour que le ramasse-miettes se déclenche).

Comment fonctionne un ramasse-miettes ?

Les utilisateurs de langages où l'allocation d'objets sur le tas coûte cher peuvent supposer que la façon qu'a Java de tout allouer sur le tas (à l'exception des types de base) coûte également cher. Cependant, il se trouve que l'utilisation d'un ramasse-miettes peut *accélérer* de manière importante la création d'objets. Ceci peut sembler un peu bizarre à première vue : la réclamation d'objets aurait un effet sur la création d'objets. Mais c'est comme ça que certaines JVMs fonctionnent et cela veut dire, qu'en Java, l'allocation d'objets sur le tas peut être presque aussi rapide que l'allocation sur la pile dans d'autres langages.

Un exemple serait de considérer le tas en C++ comme une pelouse où chaque objet prend et délimite son morceau de gazon. Cet espace peut être abandonné un peu plus tard et doit être réutilisé. Avec certaines JVMs, le tas de Java est assez différent ; il ressemble plus à une chaîne de montage qui avancerait à chaque fois qu'un objet est alloué. Ce qui fait que l'allocation est remarquablement rapide. Le «pointeur du tas» progresse simplement dans l'espace vide, ce qui correspond donc à l'allocation sur la pile en C++ (il y a bien sûr une petite pénalité supplémentaire pour le fonctionnement interne mais ce n'est pas comparable à la recherche de mémoire libre).

On peut remarquer que le tas n'est en fait pas vraiment une chaîne de montage, et s'il est traité de cette manière, la mémoire finira par avoir un taux de «paging» (utiliser toute la mémoire virtuelle incluant la partie sur disque dur) important (ce qui représente un gros problème de performance) et finira par manquer de mémoire. Le ramasse-miettes apporte la solution en s'interposant et, alors qu'il collecte les miettes (les objets inutilisables), il compacte tous les objets du tas. Ceci représente l'action de déplacer le «pointeur du tas» un peu plus vers le début et donc plus loin du «page fault» (interruption pour demander au système d'exploitation des pages de mémoire supplémentaire situées dans la partie de la mémoire virtuelle qui se trouve sur disque dur). Le ramasse-miettes réarrange tout pour permettre l'utilisation de ce modèle d'allocation très rapide et utilisant une sorte de «tas infini».

Pour comprendre comment tout cela fonctionne, il serait bon de donner maintenant une meilleure description de la façon dont un ramasse-miettes fonctionne. Nous utiliserons l'acronyme GC (en anglais, un ramasse-miette est appelé Garbage Collector) dans les paragraphes suivants. Une technique de GC relativement simple mais lente est le compteur de référence. L'idée est que chaque objet contient un compteur de référence et à chaque fois qu'une nouvelle référence sur un objet est créée le compteur est incrémenté. A chaque fois qu'une référence est hors de portée ou que

la valeur **null** lui est assignée, le compteur de références est décrémenté. Par conséquent, la gestion des compteurs de références représente un coût faible mais constant tout au long du programme. Le ramasse-miettes se déplace à travers toute la liste d'objets et quand il en trouve un avec un compteur à zéro, il libère la mémoire. L'inconvénient principal est que si des objets se référencent de façon circulaire, ils ne peuvent jamais avoir un compteur à zéro tout en étant inaccessible. Pour localiser ces objets qui se référencent mutuellement, le ramasse-miettes doit faire un important travail supplémentaire. Les compteurs de références sont généralement utilisés pour expliquer les ramasses-miettes mais ils ne semblent pas être utilisés dans les implémentations de la JVM.

D'autres techniques, plus performantes, n'utilisent pas de compteur de références. Elles sont plutôt basées sur l'idée que l'on est capable de remonter la chaîne de références de tout objet «non-mort» (i.e encore en utilisation) jusqu'à une référence vivant sur la pile ou dans la zone statique. Cette chaîne peut très bien passer par plusieurs niveaux d'objets. Par conséquent, si l'on part de la pile et de la zone statique et que l'on trace toutes les références, on trouvera tous les objets encore en utilisation. Pour chaque référence que l'on trouve, il faut aller jusqu'à l'objet référencé et ensuite suivre toutes les références contenues dans *cet* objet, aller jusqu'aux objets référencés, etc. jusqu'à ce que l'on ait visité tous les objets que l'on peut atteindre depuis la référence sur la pile ou dans la zone statique. Chaque objet visité doit être encore vivant. Notez qu'il n'y a aucun problème avec les groupes qui s'auto-référencent : ils ne sont tout simplement pas trouvés et sont donc automatiquement morts.

Avec cette approche, la JVM utilise un ramasse-miettes *adaptatif*. Le sort des objets vivants trouvés dépend de la variante du ramasse-miettes utilisée à ce moment-là. Une de ces variantes est le *stop-and-copy*. L'idée est d'arrêter le programme dans un premier temps (ce n'est pas un ramasse-miettes qui s'exécute en arrière-plan). Puis, chaque objet vivant que l'on trouve est copié d'un tas à un autre, délaissant les objets morts. De plus, au moment où les objets sont copiés, ils sont rassemblés les uns à côté des autres, compactant de ce fait le nouveau tas (et permettant d'allouer de la mémoire en la récupérant à l'extrémité du tas comme cela a été expliqué auparavant).

Bien entendu, quand un objet est déplacé d'un endroit à un autre, toutes les références qui pointent (i.e. qui *référencent*) l'objet doivent être mis à jour. La référence qui part du tas ou de la zone statique vers l'objet peut être modifiée sur le champ, mais il y a d'autres références pointant sur cet objet qui seront trouvées «sur le chemin». Elles seront corrigées dès qu'elles seront trouvées (on peut s'imaginer une table associant les anciennes adresses aux nouvelles).

Il existe deux problèmes qui rendent ces «ramasse-miettes par copie» inefficaces. Le premier est l'utilisation de deux tas et le déplacement des objets d'un tas à l'autre, utilisant ainsi deux fois plus de mémoire que nécessaire. Certaines JVMs s'en sortent en allouant la mémoire par morceau et en copiant simplement les objets d'un morceau à un autre.

Le deuxième problème est la copie. Une fois que le programme atteint un état stable, il se peut qu'il ne génère pratiquement plus de miettes (i.e. d'objets morts). Malgré ça, le ramasse-miettes par copie va quand même copier toute la mémoire d'une zone à une autre, ce qui est du gaspillage put et simple. Pour éviter cela, certaines JVMs détectent que peu d'objets meurent et choisissent alors une autre technique (c'est la partie d'«adaptation»). Cette autre technique est appelée *mark and sweep* (NDT : littéralement *marque et balaye*), et c'est ce que les versions précédentes de la JVM de Sun utilisaient en permanence. En général, le «mark and sweep» est assez lent, mais quand on sait que l'on génère peu ou pas de miettes, la technique est rapide.

La technique de «mark and sweep» suit la même logique de partir de la pile et de la zone de mémoire statique et de suivre toutes les références pour trouver les objets encore en utilisation. Cependant, à chaque fois qu'un objet vivant est trouvé, il est marqué avec un flag, mais rien n'est en-

core collecté. C'est seulement lorsque la phase de «mark» est terminée que le «sweep» commence. Pendant ce balayage, les objets morts sont libérés. Aucune copie n'est effectuée, donc si le ramasse-miettes décide de compacter la mémoire, il le fait en réarrangeant les objets.

Le «stop-and-copy» correspond à l'idée que ce type de ramasse-miettes ne s'exécute *pas* en tâche de fond, le programme est en fait arrêté pendant l'exécution du ramasse-miettes. La littérature de Sun mentionne assez souvent le ramasse-miettes comme une tâche de fond de basse priorité, mais il se trouve que le ramasse-miettes n'a pas été implémenté de cette manière, tout au moins dans les premières versions de la JVM de Sun. Le ramasse-miettes était plutôt exécuté quand il restait peu de mémoire libre. De plus, le «mark-and-sweep» nécessite l'arrêt du programme.

Comme il a été dit précédemment, la JVM décrite ici alloue la mémoire par blocs. Si un gros objet est alloué, un bloc complet lui est réservé. Le «stop-and-copy» strictement appliqué nécessite la copie de chaque objet vivant du tas d'origine vers un nouveau tas avant de pouvoir libérer le vieux tas, ce qui se traduit par la manipulation de beaucoup de mémoire. Avec des blocs, le ramasse-miettes peut simplement utiliser les blocs vides (et/ou contenant uniquement des objets morts) pour y copier les objets. Chaque bloc possède un *compteur de génération* pour savoir s'il est « mort » (vide) ou non. Dans le cas normal, seuls les blocs créés depuis le ramasse-miettes sont compactés ; les compteurs de générations de tous les autres blocs sont mis à jour s'ils ont été référencés. Cela prend en compte le cas courant des nombreux objets ayant une durée de vie très courte. Régulièrement, un balayage complet est effectué, les gros objets ne sont toujours pas copiés (leurs compteurs de génération sont simplement mis à jour) et les blocs contenant des petits objets sont copiés et compactés. La JVM évalue constamment l'efficacité du ramasse-miettes et si cette technique devient une pénalité plutôt qu'un avantage, elle la change pour un « mark-and-sweep ». De même, la JVM évalue l'efficacité du mark-and-sweep et si le tas se fragmente, le stop-and-copy est réutilisé. C'est là où l'« adaptation » vient en place et finalement on peut utiliser ce terme anglophone à rallonge : « adaptive generational stop-and-copy mark-and-sweep » qui correspondrait à « adaptatif entre marque-et-balaye et stoppe-et-copie de façon générationnelle ».

Initialisation de membre

Java prend en charge l'initialisation des variables avant leur utilisation. Dans le cas des variables locales à une méthode, cette garantie prend la forme d'une erreur à la compilation. Donc le code suivant :

```
void f() {
  int i;
  i++;
 }
```

générera un message d'erreur disant que la variable **i** peut ne pas avoir été initialisée. Bien entendu, le compilateur aurait pu donner à **i** une valeur par défaut, mais il est plus probable qu'il s'agit d'une erreur de programmation et une valeur par défaut aurait masqué ce problème. En forçant le programmeur à donner une valeur par défaut, il y a plus de chances de repérer un bogue.

Cependant, si une valeur primitive est un membre de données d'une classe, les choses sont un peu différentes. Comme n'importe quelle méthode peut initialiser ou utiliser cette donnée, il ne serait pas très pratique ou faisable de forcer l'utilisateur à l'initialiser correctement avant son utilisation. Cependant, il n'est pas correct de la laisser avec n'importe quoi comme valeur, Java garantit donc de donner une valeur initiale à chaque membre de données avec un type primitif. On peut voir

ces valeurs ici :

```
//: c04:InitialValues.java
// Imprime les valeurs initiales par défaut.

class Measurement {
  boolean t;
  char c;
  byte b;
  short s;
  int i;
  long l;
  float f;
  double d;
  void print() {
    System.out.println(
      "Data type      Initial value\n" +
      "boolean        " + t + "\n" +
      "char          [" + c + "] "+ (int)c +"\n"+
      "byte           " + b + "\n" +
      "short          " + s + "\n" +
      "int           " + i + "\n" +
      "long           " + l + "\n" +
      "float          " + f + "\n" +
      "double         " + d);
  }
}

public class InitialValues {
  public static void main(String[] args) {
    Measurement d = new Measurement();
    d.print();
    /* Dans ce cas, il est également possible d'écrire :
    new Measurement().print();
    */
  }
} ///:~
```

Voici la sortie de ce programme :

```
Data type      Initial value
boolean        false
char          [ ] 0
byte           0
short          0
int           0
long           0
float         0.0
double         0.0
```

La valeur pour **char** est zéro, ce qui se traduit par un espace dans la sortie-écran.

Nous verrons plus tard que quand on définit une référence sur un objet dans une classe sans l'initialiser avec un nouvel objet, la valeur spéciale **null** (mot-clé Java) est donnée à cette référence.

On peut voir que même si des valeurs ne sont pas spécifiées, les données sont initialisées automatiquement. Il n'y a donc pas de risque de travailler par inattention avec des variables non-initialisées.

Spécifier une initialisation

Comment peut-on donner une valeur initiale à une variable ? Une manière directe de le faire est la simple affectation au moment de la définition de la variable dans la classe (note : il n'est pas possible de le faire en C++ bien que tous les débutants s'y essayent). Les définitions des champs de la classe **Measurement** sont modifiées ici pour fournir des valeurs initiales :

```
class Measurement {
  boolean b = true;
  char c = 'x';
  byte B = 47;
  short s = 0xff;
  int i = 999;
  long l = 1;
  float f = 3.14f;
  double d = 3.14159;
  //. . .
```

On peut initialiser des objets de type non-primitif de la même manière. Si **Depth** (NDT : « profondeur ») est une classe, on peut ajouter une variable et l'initialiser de cette façon :

```
class Measurement {
  Depth o = new Depth();
  boolean b = true;
  // . . .
```

Si **o** ne reçoit pas de valeur initiale et que l'on essaye de l'utiliser malgré tout, on obtient une erreur à l'exécution appelée *exception* (explications au chapitre 10).

Il est même possible d'appeler une méthode pour fournir une valeur d'initialisation :

```
class CInit {
  int i = f();
  //...
}
```

Bien sûr cette méthode peut avoir des arguments, mais ceux-ci ne peuvent pas être d'autres membres non encore initialisés, de la classe. Par conséquent ce code est valide :

```
class CInit {
  int i = f();
  int j = g(i);
```

```
  //...
}
```

Mais pas celui-ci :

```
class CInit {
  int j = g(i);
  int i = f();
  //...
}
```

C'est un des endroits où le compilateur *se plaint* avec raison du forward referencing (référence à un objet déclaré plus loin dans le code), car il s'agit d'une question d'ordre d'initialisation et non pas de la façon dont le programme est compilé.

Cette approche par rapport à l'initialisation est très simple. Elle est également limitée dans le sens où *chaque* objet de type **Measurement** aura les mêmes valeurs d'initialisation. Quelquefois c'est exactement ce dont on a besoin, mais d'autres fois un peu plus de flexibilité serait nécessaire.

Initialisation par constructeur

On peut utiliser le constructeur pour effectuer les initialisations. Cela apporte plus de flexibilité pour le programmeur car il est possible d'appeler des méthodes et effectuer des actions à l'exécution pour déterminer les valeurs initiales. Cependant il y a une chose à se rappeler : cela ne remplace pas l'initialisation automatique qui est faite avant l'exécution du constructeur. Donc par exemple :

```
class Counter {
  int i;
  Counter() { i = 7; }
  // . . .
```

Dans ce cas, **i** sera d'abord initialisé à 0 puis à 7. C'est ce qui ce passe pour tous les types primitifs et les références sur objet, même pour ceux qui ont été initialisés explicitement au moment de leur définition. Pour cette raison, le compilateur ne force pas l'utilisateur à initialiser les éléments dans le constructeur à un endroit donné, ni avant leur utilisation : l'initialisation est toujours garantie [30].

Ordre d'initialisation

Dans une classe, l'ordre d'initialisation est déterminé par l'ordre dans lequel les variables sont definies. Les définitions de variables peuvent être disséminées n'importe où et même entre les définitions des méthodes, mais elles sont initialisées avant tout appel à une méthode, même le constructeur. Par exemple :

```
//: c04:OrderOfInitialization.java
// Montre l'ordre d'initialisation.

// Quand le constructeur est appelé pour créer
// un objet Tag, un message s'affichera :
```

```
class Tag {
  Tag(int marker) {
    System.out.println("Tag(" + marker + ")");
  }
}

class Card {
  Tag t1 = new Tag(1); // Avant le constructeur
  Card() {
    // Montre que l'on est dans le constructeur :
    System.out.println("Card()");
    t3 = new Tag(33); // Réinitialisation de t3
  }
  Tag t2 = new Tag(2); // Après le constructeur
  void f() {
    System.out.println("f()");
  }
  Tag t3 = new Tag(3); // la fin
}

public class OrderOfInitialization {
  public static void main(String[] args) {
    Card t = new Card();
    t.f(); // Montre que la construction a été effectuée
  }
} ///:~
```

Dans la classe **Card**, les définitions des objets **Tag** sont intentionnellement dispersées pour prouver que ces objets seront tous initialisés avant toute action (y compris l'appel du constructeur). De plus, **t3** est réinitialisé dans le constructeur. La sortie-écran est la suivante :

```
Tag(1)
Tag(2)
Tag(3)
Card()
Tag(33)
f()
```

La référence sur **t3** est donc initialisée deux fois, une fois avant et une fois pendant l'appel au constructeur (on jette le premier objet pour qu'il soit récupéré par le ramasse-miettes plus tard). A première vue, cela ne semble pas très efficace, mais cela garantit une initialisation correcte ; que se passerait-il si l'on surchargeait le constructeur avec un autre constructeur qui *n'initialiserait pas* **t3** et qu'il n'y avait pas d'initialisation « par défaut » dans la définition de **t3** ?

Initialisation de données statiques

Quand les données sont statiques (**static**) la même chose se passe ; s'il s'agit d'une donnée de type primitif et qu'elle n'est pas initialisée, la variable reçoit une valeur initiale standard. Si c'est une

référence sur un objet, c'est la valeur **null** qui est utilisée à moins qu'un nouvel objet ne soit créé et sa référence donnée comme valeur à la variable.

Pour une initialisation à l'endroit de la définition, les mêmes règles que pour les variables non-statiques sont appliquées. Il n'y a qu'une seule version (une seule zone mémoire) pour une variable statique quel que soit le nombre d'objets créés. Mais une question se pose lorsque cette zone statique est initialisée. Un exemple va rendre cette question claire :

```
//: c04:StaticInitialization.java
// Préciser des valeurs initiales dans une
// définition de classe.

class Bowl {
  Bowl(int marker) {
    System.out.println("Bowl(" + marker + ")");
  }
  void f(int marker) {
    System.out.println("f(" + marker + ")");
  }
}

class Table {
  static Bowl b1 = new Bowl(1);
  Table() {
    System.out.println("Table()");
    b2.f(1);
  }
  void f2(int marker) {
    System.out.println("f2(" + marker + ")");
  }
  static Bowl b2 = new Bowl(2);
}

class Cupboard {
  Bowl b3 = new Bowl(3);
  static Bowl b4 = new Bowl(4);
  Cupboard() {
    System.out.println("Cupboard()");
    b4.f(2);
  }
  void f3(int marker) {
    System.out.println("f3(" + marker + ")");
  }
  static Bowl b5 = new Bowl(5);
}

public class StaticInitialization {
  public static void main(String[] args) {
    System.out.println(
```

```
      "Creating new Cupboard() in main");
    new Cupboard();
    System.out.println(
      "Creating new Cupboard() in main");
    new Cupboard();
    t2.f2(1);
    t3.f3(1);
  }
  static Table t2 = new Table();
  static Cupboard t3 = new Cupboard();
} ///:~
```

Bowl permet de visionner la création d'une classe. **Table**, ainsi que **Cupboard**, créent des membres **static** de **Bowl** partout au travers de leur définition de classe. Il est à noter que **Cupboard** crée un **Bowl b3** non-**statique** avant les définitions **statiques**. La sortie montre ce qui se passe :

```
Bowl(1)
Bowl(2)
Table()
f(1)
Bowl(4)
Bowl(5)
Bowl(3)
Cupboard()
f(2)
Creating new Cupboard() in main
Bowl(3)
Cupboard()
f(2)
Creating new Cupboard() in main
Bowl(3)
Cupboard()
f(2)
f2(1)
f3(1)
```

L'initialisation **statique** intervient seulement si c'est nécessaire. Si on ne crée jamais d'objets **Table** et que **Table.b1** ou **Table.b2** ne sont jamais référencés, les membres **statiques Bowl b1** et **b2** ne seront jamais créés. Cependant, ils ne sont initialisés que lorsque le *premier* objet **Table** est créé (ou le premier accès **statique** est effectué). Après cela, les objets **statiques** ne sont pas réinitialisés.

Dans l'ordre d'initialisation, les membres **static** viennent en premier, s'ils n'avaient pas déjà été initialisés par une précédente création d'objet, les objets non **static** sont traités. On peut le voir clairement dans la sortie du programme.

Il peut être utile de résumer le processus de création d'un objet. Considérons une classe appelée **Dog** :

1. La première fois qu'un objet de type **Dog** est créé, *ou* la première fois qu'on utilise une méthode déclarée **static** ou un champ **static** de la classe **Dog**, l'interpréteur Java doit locali-

ser **Dog.class**, ce qu'il fait en cherchant dans le classpath ;

2. Au moment où **Dog.class** est chargée (créant un objet **Class**, que nous verrons plus tard), toutes les fonctions d'initialisation **statiques** sont exécutées. Par conséquent, l'initialisation **statique** n'arrive qu'une fois, au premier chargement de l'objet **Class** ;

3. Lorsque l'on exécute **new Dog( )** pour créer un nouvel objet de type **Dog**, le processus de construction commence par allouer suffisamment d'espace mémoire sur le tas pour contenir un objet **Dog** ;

4. Cet espace est mis à zéro, donnant automatiquement à tous les membres de type primitif dans cet objet **Dog** leurs valeurs par défaut (zéro pour les nombres et l'équivalent pour les **boolean** et les **char**) et aux références la valeur **null** ;

5. Toute initialisation effectuée au moment de la définition des champs est exécutée ;

6. Les constructeurs sont exécutés. Comme nous le verrons au chapitre 6, ceci peut en fait déclencher beaucoup d'activité, surtout lorsqu'il y a de l'héritage.

Initialisation statique explicite

Java permet au programmeur de grouper toute autre initialisation statique dans une « clause de construction » **static** (quelquefois appelé *bloc statique*) dans une classe. Cela ressemble à ceci :

```
class Spoon {
  static int i;
  static {
    i = 47;
  }
  // . . .
```

On dirait une méthode, mais il s'agit simplement du mot-clé **static** suivi d'un corps de méthode. Ce code, comme les autres initialisations statiques, est exécuté une seule fois, à la création du premier objet de cette classe *ou* au premier accès à un membre déclaré **static** de cette classe (même si on ne crée jamais d'objet de cette classe). Par exemple :

```
//: c04:ExplicitStatic.java
// Initialisation statique explicite
// avec l'instruction "static".

class Cup {
  Cup(int marker) {
    System.out.println("Cup(" + marker + ")");
  }
  void f(int marker) {
    System.out.println("f(" + marker + ")");
  }
}

class Cups {
  static Cup c1;
  static Cup c2;
```

```
  static {
    c1 = new Cup(1);
    c2 = new Cup(2);
  }
  Cups() {
    System.out.println("Cups()");
  }
}

public class ExplicitStatic {
  public static void main(String[] args) {
    System.out.println("Inside main()");
    Cups.c1.f(99);  // (1)
  }
  // static Cups x = new Cups();  // (2)
  // static Cups y = new Cups();  // (2)
} ///:~
```

Les instructions statiques d'initialisation pour **Cups** sont exécutées soit quand l'accès à l'objet **static c1** intervient à la ligne (1), soit si la ligne (1) est mise en commentaire et les lignes (2) ne le sont pas. Si (1) et (2) sont en commentaire, l'initialisation **static** pour **Cups** n'intervient jamais. De plus, que l'on enlève les commentaires pour les deux lignes (2) ou pour une seule n'a aucune importance : l'initialisation statique n'est effectuée qu'une seule fois.

Initialisation d'instance non statique

Java offre une syntaxe similaire pour initialiser les variables non **static** pour chaque objet. Voici un exemple :

```
//: c04:Mugs.java
// Java "Instance Initialization." (Initialisation d'instance de Java)

class Mug {
  Mug(int marker) {
    System.out.println("Mug(" + marker + ")");
  }
  void f(int marker) {
    System.out.println("f(" + marker + ")");
  }
}

public class Mugs {
  Mug c1;
  Mug c2;
  {
    c1 = new Mug(1);
    c2 = new Mug(2);
    System.out.println("c1 & c2 initialized");
```

```
  }
  Mugs() {
    System.out.println("Mugs()");
  }
  public static void main(String[] args) {
    System.out.println("Inside main()");
    Mugs x = new Mugs();
  }
} ///:~
```

```
  {
    c1 = new Mug(1);
    c2 = new Mug(2);
    System.out.println("c1 & c2 initialized");
  }
```

ressemble exactement à la clause d'initialisation statique moins le mot-clé **static**. Cette syntaxe est nécessaire pour permettre l'initialisation de *classes internes anonymes* (voir Chapitre 8).

Initialisation des tableaux

L'initialisation des tableaux en C est laborieuse et source d'erreurs. C++ utilise *l'initialisation d'aggregats* pour rendre cette opération plus sûre [31]. Java n'a pas d'« aggregats » comme C++, puisque tout est objet en Java. Java possède pourtant des tableaux avec initialisation.

Un tableau est simplement une suite d'objets ou de types de base, tous du même type et réunis ensemble sous un même nom. Les tableaux sont définis et utilisés avec l'*opérateur d'indexation* **[ ]** (crochets ouvrant et fermant). Pour définir un tableau il suffit d'ajouter des crochets vides après le nom du type :

```
int[] a1;
```

Les crochets peuvent également être placé après le nom de la variable :

```
int a1[];
```

Cela correspond aux attentes des programmeurs C et C++. Toutefois, la première syntaxe est probablement plus sensée car elle annonce le type comme un « tableau de **int**. » Ce livre utilise cette syntaxe.

Le compilateur ne permet pas de spécifier la taille du tableau à sa définition. Cela nous ramène à ce problème de « référence. » A ce point on ne dispose que d'une référence sur un tableau, et aucune place n'a été allouée pour ce tableau. Pour créer cet espace de stockage pour le tableau, il faut écrire une expression d'initialisation. Pour les tableaux, l'initialisation peut apparaître à tout moment dans le code, mais on peut également utiliser un type spécial d'initialisation qui doit alors apparaître à la déclaration du tableau. Cette initalisation spéciale est un ensemble de valeurs entre accolades. L'allocation de l'espace de stockage pour le tableau (l'équivalent de **new**) est prise en charge par le compilateur dans ce cas. Par exemple :

```
int[] a1 = { 1, 2, 3, 4, 5 };
```

Mais pourquoi voudrait-on définir une référence sur tableau sans tableau ?

```
int[] a2;
```

Il est possible d'affecter un tableau à un autre en Java, on peut donc écrire :

```
a2 = a1;
```

Cette expression effectue en fait une copie de référence, comme le montre la suite :

```
//: c04:Arrays.java
// Tableau de types primitifs.

public class Arrays {
  public static void main(String[] args) {
    int[] a1 = { 1, 2, 3, 4, 5 };
    int[] a2;
    a2 = a1;
    for(int i = 0; i < a2.length; i++)
      a2[i]++;
    for(int i = 0; i < a1.length; i++)
      System.out.println(
        "a1[" + i + "] = " + a1[i]);
  }
} ///:~
```

On peut voir que **a1** a une valeur initiale tandis qu' **a2** n'en a pas ; **a2** prend une valeur plus tard— dans ce cas, vers un autre tableau.

Maintenant voyons quelque chose de nouveau : tous les tableaux ont un membre intrinsèque (qu'ils soient tableaux d'objets ou de types de base) que l'on peut interroger — mais pas changer — ; il donne le nombre d'éléments dans le tableau. Ce membre s'appelle **length** (longueur). Comme les tableaux en Java , comme C et C++, commencent à la case zero, le plus grand nombre d'éléments que l'on peut indexer est **length - 1**. Lorsqu'on dépasse ces bornes, C et C++ acceptent cela tranquillement et la mémoire peut être corrompue ; ceci est la cause de bogues infâmes. Par contre, Java empêche ce genre de problèmes en générant une erreur d'exécution (une *exception*, le sujet du Chapitre 10) lorsque le programme essaye d'accéder à une valeur en dehors des limites. Bien sûr, vérifier ainsi chaque accès coûte du temps et du code ; comme il n'y a aucun moyen de désactiver ces vérifications, les accès tableaux peuvent être une source de lenteur dans un programme s'ils sont placés à certains points critiques de l'exécution. Les concepteurs de Java ont pensé que cette vitesse légèrement réduite était largement contrebalancée par les aspects de sécurité sur Internet et la meilleure productivité des programmeurs.

Que faire quand on ne sait pas au moment où le programme est écrit, combien d'éléments vont être requis à l'exécution ? Il suffit d'utiliser **new** pour créer les éléments du tableau. Dans ce cas, **new** fonctionne même pour la création d'un tableau de types de base (**new** ne peut pas créer un type de base) :

```
//: c04:ArrayNew.java
// Créer des tableaux avec new.
import java.util.*;

public class ArrayNew {
  static Random rand = new Random();
  static int pRand(int mod) {
    return Math.abs(rand.nextInt()) % mod + 1;
  }
  public static void main(String[] args) {
    int[] a;
    a = new int[pRand(20)];
    System.out.println(
      "length of a = " + a.length);
    for(int i = 0; i < a.length; i++)
      System.out.println(
        "a[" + i + "] = " + a[i]);
  }
} ///:~
```

Comme la taille du tableau est choisie aléatoirement (en utilisant la méthode **pRand( )**), il est clair que la création du tableau se passe effectivement à l'exécution. De plus, on peut voir en exécutant le programme que les tableaux de types primitifs sont automatiquement initialisés avec des valeurs “vides” (pour les nombres et les **char**, cette valeur est zéro, pour les **boolean**, cette valeur est **false**).

Bien sûr le tableau pourrait aussi avoir été défini et initialisé sur la même ligne :

```
int[] a = new int[pRand(20)];
```

Lorsque l'on travaille avec un tableau d'objets non primitifs, il faut toujours utiliser **new**. Encore une fois, le problème des références revient car ce que l'on crée est un tableau de références. Considérons le type englobant **Integer,** qui est une classe et non un type de base :

```
//: c04:ArrayClassObj.java
// Création d'un tableau d'objets (types de base exclus).
import java.util.*;

public class ArrayClassObj {
  static Random rand = new Random();
  static int pRand(int mod) {
    return Math.abs(rand.nextInt()) % mod + 1;
  }
  public static void main(String[] args) {
    Integer[] a = new Integer[pRand(20)];
    System.out.println(
      "length of a = " + a.length);
    for(int i = 0; i < a.length; i++) {
      a[i] = new Integer(pRand(500));
```

```
      System.out.println(
        "a[" + i + "] = " + a[i]);
    }
  }
} ///:~
```

Ici, même après que **new** ait été appelé pour créer le tableau :

```
Integer[] a = new Integer[pRand(20)];
```

c'est uniquement un tableau de références, et l'initialisation n'est pas complète tant que cette référence n'a pas elle-même été initialisée en créant un nouvel objet **Integer** :

```
a[i] = new Integer(pRand(500));
```

Oublier de créer l'objet produira une exception d'exécution dès que l'on accédera à l'emplacement.

Regardons la formation de l'objet **String** à l'intérieur de print. On peut voir que la référence vers l'objet **Integer** est automatiquement convertie pour produire une **String** représentant la valeur à l'intérieur de l'objet.

Il est également possible d'initialiser des tableaux d'objets en utilisant la liste délimitée par des accolades. Il y a deux formes :

```
//: c04:ArrayInit.java
// Initialisation de tableaux.

public class ArrayInit {
  public static void main(String[] args) {
    Integer[] a = {
      new Integer(1),
      new Integer(2),
      new Integer(3),
    };

    Integer[] b = new Integer[] {
      new Integer(1),
      new Integer(2),
      new Integer(3),
    };
  }
} ///:~
```

C'est parfois utile, mais d'un usage plus limité car la taille du tableau est déterminée à la compilation. La virgule finale dans la liste est optionnelle. (Cette fonctionnalité permet une gestion plus facile des listes longues.)

La deuxième forme d'initialisation de tableaux offre une syntaxe pratique pour créer et appeler des méthodes qui permet de donner le même effet que *les listes à nombre d'arguments variable* de C (“varargs” en C). Ces dernières permettent le passage d'un nombre quelconque de paramètres,

chacun de type inconnu. Comme toutes les classes héritent d'une classe racine **Object** (un sujet qui sera couvert en détail tout au long du livre), on peut créer une méthode qui prend un tableau d'**Object** et l'appeler ainsi :

```
//: c04:VarArgs.java
// Utilisation de la syntaxe des tableaux pour créer
// des listes à nombre d'argument variable.

class A { int i; }

public class VarArgs {
  static void f(Object[] x) {
    for(int i = 0; i < x.length; i++)
      System.out.println(x[i]);
  }
  public static void main(String[] args) {
    f(new Object[] {
        new Integer(47), new VarArgs(),
        new Float(3.14), new Double(11.11) });
    f(new Object[] {"one", "two", "three" });
    f(new Object[] {new A(), new A(), new A()});
  }
} ///:~
```

A ce niveau, il n'y a pas grand chose que l'on peut faire avec ces objets inconnus, et ce programme utilise la conversion automatique vers **String** afin de faire quelque chose d'utile avec chacun de ces **Object**s. Au chapitre 12, qui explique l'*identification dynamique de types* (RTTI), nous verrons comment découvrir le type exact de tels objets afin de les utiliser à des fins plus intéressantes.

Tableaux multidimensionnels

Java permet de créer facilement des tableaux multidimensionnels :

```
//: c04:MultiDimArray.java
// Création de tableaux multidimensionnels.
import java.util.*;

public class MultiDimArray {
  static Random rand = new Random();
  static int pRand(int mod) {
    return Math.abs(rand.nextInt()) % mod + 1;
  }
  static void prt(String s) {
    System.out.println(s);
  }
  public static void main(String[] args) {
    int[][] a1 = {
      { 1, 2, 3, },
```

```
    { 4, 5, 6, },
  };
  for(int i = 0; i < a1.length; i++)
    for(int j = 0; j < a1[i].length; j++)
      prt("a1[" + i + "][" + j +
          "] = " + a1[i][j]);
  // tableau 3-D avec taille fixe :
  int[][][] a2 = new int[2][2][4];
  for(int i = 0; i < a2.length; i++)
    for(int j = 0; j < a2[i].length; j++)
      for(int k = 0; k < a2[i][j].length;
          k++)
        prt("a2[" + i + "][" +
            j + "][" + k +
            "] = " + a2[i][j][k]);
  // tableau 3-D avec vecteurs de taille variable :
  int[][][] a3 = new int[pRand(7)][][];
  for(int i = 0; i < a3.length; i++) {
    a3[i] = new int[pRand(5)][];
    for(int j = 0; j < a3[i].length; j++)
      a3[i][j] = new int[pRand(5)];
  }
  for(int i = 0; i < a3.length; i++)
    for(int j = 0; j < a3[i].length; j++)
      for(int k = 0; k < a3[i][j].length;
          k++)
        prt("a3[" + i + "][" +
            j + "][" + k +
            "] = " + a3[i][j][k]);
  // Tableau d'objets non primitifs :
  Integer[][] a4 = {
    { new Integer(1), new Integer(2)},
    { new Integer(3), new Integer(4)},
    { new Integer(5), new Integer(6)},
  };
  for(int i = 0; i < a4.length; i++)
    for(int j = 0; j < a4[i].length; j++)
      prt("a4[" + i + "][" + j +
          "] = " + a4[i][j]);
  Integer[][] a5;
  a5 = new Integer[3][];
  for(int i = 0; i < a5.length; i++) {
    a5[i] = new Integer[3];
    for(int j = 0; j < a5[i].length; j++)
      a5[i][j] = new Integer(i*j);
  }
  for(int i = 0; i < a5.length; i++)
```

```
        for(int j = 0; j < a5[i].length; j++)
          prt("a5[" + i + "][" + j +
              "] = " + a5[i][j]);
    }
} ///:~
```

Le code d'affichage utilise **length** ; de cette façon il ne force pas une taille de tableau fixe.

Le premier exemple montre un tableau multidimensionnel de type primitifs. Chaque vecteur du tableau est délimité par des accolades :

```
int[ ][ ] a1 = {
    { 1, 2, 3, },
    { 4, 5, 6, },
  };
```

Chaque paire de crochets donne accès à la dimension suivante du tableau.

Le deuxième exemple montre un tableau à trois dimensions alloué par **new**. Ici le tableau entier est alloué en une seule fois :

```
int[ ][ ][ ] a2 = new int[2][2][4];
```

Par contre, le troisième exemple montre que les vecteurs dans les tableaux qui forment la matrice peuvent être de longueurs différentes :

```
int[ ][ ][ ] a3 = new int[pRand(7)][][];
  for(int i = 0; i < a3.length; i++) {
    a3[i] = new int[pRand(5)][];
    for(int j = 0; j < a3[i].length; j++)
      a3[i][j] = new int[pRand(5)];
  }
```

Le premier **new** crée un tableau avec un longueur aléatoire pour le premier élément et le reste de longueur indéterminée. Le deuxième **new** à l'intérieur de la boucle **for** remplit les éléments mais laisse le troisième index indéterminé jusqu'au troisième **new**.

On peut voir à l'exécution que les valeurs des tableaux sont automatiquement initialisées à zéro si on ne leur donne pas explicitement de valeur initiale.

Les tableaux d'objets non primitifs fonctionnent exactement de la même manière, comme le montre le quatrième exemple, qui présente la possibilité d'utiliser **new** dans les accolades d'initialisation :

```
Integer[][] a4 = {
    { new Integer(1), new Integer(2)},
    { new Integer(3), new Integer(4)},
    { new Integer(5), new Integer(6)},
  };
```

Le cinquième exemple montre comment un tableau d'objets non primitifs peut être construit pièce par pièce :

```
Integer[ ][ ] a5;
  a5 = new Integer[3][ ];
  for(int i = 0; i < a5.length; i++) {
    a5[i] = new Integer[3];
    for(int j = 0; j < a5[i].length; j++)
      a5[i][j] = new Integer(i*j);
  }
```

L'expression **i*j** est là uniquement pour donner une valeur intéressante à l' **Integer**.

Résumé

Le mécanisme apparemment sophistiqué d'initialisation que l'on appelle constructeur souligne l'importance donnée à l'initialisation dans ce langage. Quand Stroustrup était en train de créer C++, une des premières observations qu'il fit à propos de la productivité en C était qu'une initialisation inappropriée des variables cause de nombreux problèmes de programmation. Ce genre de bogues est difficile à trouver. Des problèmes similaires se retrouvent avec un mauvais nettoyage. Parce que les constructeurs permettent de *garantir* une initialisation et un nettoyage correct (le compilateur n'autorisera pas la création d'un objet sans un appel valide du constructeur), le programmeur a un contrôle complet en toute sécurité.

En C++, la destruction est importante parce que les objets créés avec **new** doivent être détruits explicitement .En Java, le ramasse-miettes libère automatiquement la mémoire pour tous les objets, donc la méthode de nettoyage équivalente en Java n'est pratiquement jamais nécessaire. Dans les cas où un comportement du style destructeur n'est pas nécessaire, le ramasse-miettes de Java simplifie grandement la programmation et ajoute une sécurité bien nécessaire à la gestion mémoire. Certains ramasse-miettes peuvent même s'occuper du nettoyage d'autres ressources telles que les graphiques et les fichiers. Cependant, le prix du ramasse-miettes est payé par une augmentation du temps d'exécution, qu'il est toutefois difficile d'évaluer à cause de la lenteur globale des interpréteurs Java au moment de l'écriture de cet ouvrage. Lorsque cela changera, il sera possible de savoir si le coût du ramasse-miettes posera des barrières à l'utilisation de Java pour certains types de programmes (un des problèmes est que le ramasse-miettes est imprévisible).

Parce que Java garantit la construction de tous les objets, le constructeur est, en fait, plus conséquent que ce qui est expliqué ici. En particulier, quand on crée de nouvelles classes en utilisant soit *la composition*, soit *l'héritage* la garantie de construction est maintenue et une syntaxe supplémentaire est nécessaire. La composition, l'héritage et leurs effets sur les constructeurs sont expliqués un peu plus loin dans cet ouvrage.

Exercices

Les solutions aux exercices choisis peuvent être trouvées dans le document électronique *The Thinking in Java Annotated Solution Guide*, disponible pour une modeste somme à l'adresse *www.BruceEckel.com*.

1. Créez une classe avec un constructeur par défaut (c'est à dire sans argument) qui imprime un message. Créez un objet de cette classe.

2. Ajoutez à la classe de l'exercice 1 un constructeur surchargé qui prend une **String** en argument et qui l'imprime avec votre message.

3. Créez un tableau de références sur des objets de la classe que vous avez créée à l'exercice 2. Mais ne créez pas les objets eux-même. Quand le programme s'exécute, voyez si les messages d'initialisation du constructeur sont imprimés.

4. Terminez l'exercice 3 en créant les objets pour remplir le tableau de références.

5. Créez un tableau d'objets **String** et affectez une chaîne de caractères à chaque élément. Imprimez le tableau en utilisant une boucle **for**.

6. Créez une classe **Dog** avec une méthode **bark( )** (NDT: to bark = aboyer) surchargée. Cette méthode sera surchargée en utilisant divers types primitifs de données et devra imprimer différent types d'aboiement, hurlement, ... suivant la version surchargée qui est appelée. Écrivez également une méthode **main( )** qui appellera toutes les versions.

7. Modifiez l'exercice 6 pour que deux des méthodes surchargées aient deux paramètres (de deux types différents), mais dans l'ordre inverse l'une par rapport à l'autre. Vérifiez que cela fonctionne.

8. Créez une classe sans constructeur et créez ensuite un objet de cette classe dans **main( )** pour vérifier que le constructeur par défaut est construit automatiquement.

9. Créez une classe avec deux méthodes. Dans la première méthode, appelez la seconde méthode deux fois : la première fois sans utiliser **this** et la seconde fois en l'utilisant.

10. Créez une classe avec deux constructeurs (surchargés). En utilisant **this**, appelez le second constructeur dans le premier.

11. Créez une classe avec une méthode **finalize( )** qui imprime un message. Dans **main( )**, créez un objet de cette classe. Expliquez le comportement de ce programme.

12. Modifiez l'exercice 11 pour que votre **finalize( )** soit toujours appelé.

13. Créez une classe **Tank** (NDT: citerne) qui peut être remplie et vidée et qui a une *death condition* qui est que la citerne doit être vide quand l'objet est nettoyé. Écrivez une méthode **finalize( )** qui vérifie cette death condition. Dans **main( )**, testez tous les scénario possibles d'utilisation de **Tank**.

14. Créez une classe contenant un **int** et un **char** non initialisés et imprimez leurs valeurs pour vérifier que Java effectue leurs initialisations par défaut.

15. Créez une classe contenant une référence non-initialisée à une **String**. Montrez que cette référence est initialisée à **null** par Java.

16. Créez une classe avec un champ **String** qui est initialisé à l'endroit de sa définition et un autre qui est initialisé par le constructeur. Quelle est la différence entre les deux approches ?

17. Créez une classe avec un champ **static String** qui est initialisé à l'endroit de la définition et un autre qui est initialisé par un bloc **static**. Ajoutez une méthode statique qui imprime les deux champs et montre qu'ils sont initialisés avant d'être utilisés.

18. Créez une classe avec un champ **String** qui est initialisé par une « initialisation d'instance ». Décrire une utilisation de cette fonctionnalité (autre que celle spécifiée dans cet ouvrage).

19. Écrivez une méthode qui crée et initialise un tableau de **double** à deux dimensions. La taille de ce tableau est déterminée par les arguments de la méthode. Les valeurs d'initialisation sont un intervalle déterminé par des valeurs de début et de fin également donné en paramètres de la méthode. Créez une deuxième méthode qui imprimera le tableau généré par la première. Dans **main( )**, testez les méthodes en créant et en imprimant plusieurs tableaux de différentes tailles.

20. Recommencez l'exercice 19 pour un tableau à trois dimensions.

21. Mettez en commentaire la ligne marquée (1) dans **ExplicitStatic.java** et vérifiez que la clause d'initialisation statique n'est pas appelée. Maintenant décommentez une des lignes marquées (2) et vérifiez que la clause d'initialisation statique *est* appelée. Décommentez maintenant l'autre ligne marquée (2) et vérifiez que l'initialisation statique n'est effectuée qu'une fois.

22. Faites des expériences avec **Garbage.java** en exécutant le programme avec les arguments « gc », « finalize, » ou « all ». Recommencez le processus et voyez si vous détectez des motifs répétitifs dans la sortie écran. Modifiez le code pour que **System.runFinalization( )** soit appelé *avant* **System.gc( )** et regardez les résultats.

[27] Dans certains articles écrits par Sun relatifs à Java, il est plutôt fait référence au terme maladroit bien que descriptif « no-arg constructors ». Le terme « constructeur par défaut » est utilisé depuis des années et c'est donc celui que j'utiliserai.

[28] Le seul cas dans lequel cela est possible, est si l'on passe une référence à un objet dans la méthode **statique**. Ensuite, en utilisant la référence (qui est en fait **this** maintenant), on peut appeler des méthodes non-**statiques** et accéder à des champs non-**statiques**. Mais, en général, lorsque l'on veut faire quelque chose comme cela, on crée tout simplement un méthode non-statique.

[29] Un terme créé par Bill Venners (www.artima.com) pendant le séminaire que lui et moi avons donné ensemble.

[30] En comparaison, C++ possède la *liste d'initialisation du constructeur* qui déclenche l'initialisation avant d'entrer dans le corps du constructeur. Voir *Thinking in C++, 2nde édition* (disponible sur le CDROM de cet ouvrage et à *www.BruceEckel.com*).

[31] Voir *Thinking in C++, 2nde édition* pour une description complète de l'initialisation par agrégat en C++.

Chapitre 5 - Cacher l'implémentation

Une règle principale en conception orientée objet est de « séparer les choses qui changent des choses qui ne changent pas ».

Ceci est particulièrement important pour les bibliothèques *[libraries]*. Les utilisateurs (*programmeurs clients*) de cette bibliothèque doivent pouvoir s'appuyer sur la partie qu'ils utilisent, et savoir qu'ils n'auront pas à réécrire du code si une nouvelle version de la bibliothèque sort. Inversement, le créateur de la bibliothèque doit avoir la liberté de faire des modifications et améliorations avec la certitude que le code du programmeur client ne sera pas affecté par ces modifications.

On peut y parvenir à l'aide de conventions. Par exemple le programmeur de bibliothèque doit admettre de ne pas enlever des méthodes existantes quand il modifie une classe dans la bibliothèque, puisque cela casserait le code du programmeur. La situation inverse est plus délicate. Dans le cas d'un membre de données, comment le créateur de bibliothèque peut-il savoir quels membres de données ont été accédés par les programmeurs clients ? C'est également vrai avec les méthodes qui ne sont qu'une partie de l'implémentation d'une classe et qui ne sont pas destinées à être utilisées directement par le programmeur client. Mais que se passe-t-il si le créateur de bibliothèque veut supprimer une ancienne implémentation et en mettre une nouvelle ? Changer un de ces membres pourrait casser le code d'un programmeur client. De ce fait la marge de manoeuvre du créateur de bibliothèque est plutôt étroite et il ne peut plus rien changer.

Pour corriger ce problème, Java fournit des *spécificateurs d'accès* pour permettre au créateur de bibliothèque de dire au programmeur client ce qui est disponible et ce qui ne l'est pas. Les niveaux de contrôles d'accès, depuis le « plus accessible » jusqu'au « moins accessible » sont **public**, **protected** (protégé), « friendly » (amical, qui n'a pas de mot-clé), et **private** (privé). Le paragraphe précédent semble montrer que, en tant que concepteur de bibliothèque, on devrait tout garder aussi « privé » *[private]* que possible, et n'exposer que les méthodes qu'on veut que le programmeur client utilise. C'est bien ce qu'il faut faire, même si c'est souvent non intuitif pour des gens qui programment dans d'autres langages (particulièrement en C) et qui ont l'habitude d'accéder à tout sans restrictions. Avant la fin de ce chapitre vous devriez être convaincus de l'importance du contrôle d'accès en Java.

Le concept d'une bibliothèque de composants et le contrôle de qui peut accéder aux composants de cette bibliothèque n'est cependant pas complet. Reste la question de savoir comment les composants sont liés entre eux dans une unité de bibliothèque cohérente. Ceci est contrôlé par le mot-clé **package** en Java, et les spécificateurs d'accès varient selon que la classe est dans le même package ou dans un autre package. Donc, pour commencer ce chapitre, nous allons apprendre comment les composants de bibliothèques sont placés dans des packages. Nous serons alors capables de comprendre la signification complète des spécificateurs d'accès.

package : l'unité de bibliothèque

Un package est ce qu'on obtient lorsqu'on utilise le mot-clé **import** pour apporter une bibliothèque complète, tel que

```
import java.util.*;
```

Cette instruction apporte la bibliothèque complète d'utilitaires qui font partie de la distribution Java standard. Comme par exemple la classe **ArrayList** est dans **java.util**, on peut maintenant soit spécifier le nom complet **java.util.ArrayList** (ce qui peut se faire sans l'instruction **import**), ou on peut simplement dire **ArrayList** (grâce à l'instruction **import**).

Si on veut importer une seule classe, on peut la nommer dans l'instruction **import** :

```
import java.util.ArrayList;
```

Maintenant on peut utiliser ArrayList sans précision. Cependant, aucune des autres classes de **java.util** n'est disponible.

La raison de tous ces imports est de fournir un mécanisme pour gérer les « espaces de nommage » *[name spaces]*. Les noms de tous les membres de classe sont isolés les uns des autres. Une méthode **f()** dans une classe **A** ne sera pas en conflit avec une **f()** qui a la même signature (liste d'arguments) dans une classe **B**. Mais qu'en est-il des noms des classes ? Que se passe-t-il si on crée une classe **stack** qui est installée sur une machine qui a déjà une classe **stack** écrite par quelqu'un d'autre ? Avec Java sur Internet, ceci peut se passer sans que l'utilisateur le sache, puisque les classes peuvent être téléchargées automatiquement au cours de l'exécution d'un programme Java.

Ce conflit de noms potentiel est la raison pour laquelle il est important d'avoir un contrôle complet sur les espaces de nommage en Java, et d'être capable de créer des noms complètement uniques indépendamment des contraintes d'Internet.

Jusqu'ici, la plupart des exemples de ce livre étaient dans un seul fichier et ont été conçus pour un usage en local, et ne se sont pas occupés de noms de packages (dans ce cas le nom de la classe est placé dans le « package par défaut »). C'est certainement une possibilité, et dans un but de simplicité cette approche sera utilisée autant que possible dans le reste de ce livre. Cependant, si on envisage de créer des bibliothèques ou des programmes amicaux *[friendly]* vis-à-vis d'autres programmes sur la même machine, il faut penser à se prémunir des conflits de noms de classes.

Quand on crée un fichier source pour Java, il est couramment appelé une *unité de compilation [compilation unit]*(parfois une *unité de traduction [translation unit]*). Chaque unité de compilation doit avoir un nom se terminant par **.java**, et dans l'unité de compilation il peut y avoir une classe **public** qui doit avoir le même nom que le fichier (y compris les majuscules et minuscules, mais sans l'extension .java). Il ne peut y avoir qu'une seule classe **public** dans chaque unité de compilation, sinon le compilateur sortira une erreur. Le reste des classes de cette unité de compilation, s'il y en a, sont cachées du monde extérieur parce qu'elles ne sont *pas* **public,** elles sont des classes « support » pour la classe **public** principale.

Quand on compile un fichier **.java**, on obtient un fichier de sortie avec exactement le même nom mais avec une extension **.class** *pour chaque classe* du fichier .java. De ce fait on peut obtenir un nombre important de fichiers **.class** à partir d'un petit nombre de fichiers **.java**. Si vous avez programmé avec un langage compilé, vous avez sans doute remarqué que le compilateur génère un fichier de forme intermédiaire (généralement un fichier « obj ») qui est ensuite assemblé avec d'autres fichiers de ce type à l'aide d'un éditeur de liens *[linker]* (pour créer un fichier exécutable) ou un « gestionnaire de bibliothèque » *[librarian]*(pour créer une bibliothèque). Ce n'est pas comme cela que Java travaille ; un programme exécutable est un ensemble de fichiers **.class**, qui peuvent être empaquetés *[packaged]* et compressés dans un fichier JAR (en utilisant l'archiveur Java **jar**). L'interpréteur Java est responsable de la recherche, du chargement et de l'interprétation de ces fichiers [32].

Une bibliothèque est aussi un ensemble de ces fichiers classes. Chaque fichier possède une

classe qui est **public** (on n'est pas obligé d'avoir une classe **public**, mais c'est ce qu'on fait classiquement), de sorte qu'il y a un composant pour chaque fichier. Si on veut dire que tous ces composants (qui sont dans leurs propres fichiers **.java** et **.class** séparés) sont reliés entre eux, c'est là que le mot-clé **package** intervient.

Quand on dit :

```
package mypackage;
```

au début d'un fichier (si on utilise l'instruction **package**, elle *doit* apparaître à la première ligne du fichier, commentaires mis à part), on déclare que cette unité de compilation fait partie d'une bibliothèque appelée **mypackage**. Ou, dit autrement, cela signifie que le nom de la classe **public** dans cette unité de compilation est sous la couverture du nom **mypackage**, et si quelqu'un veut utiliser ce nom, il doit soit spécifier le nom complet, soit utiliser le mot-clé import en combinaison avec **mypackage** (utilisation des choix donnés précédemment). Remarquez que la convention pour les noms de packages Java est de n'utiliser que des lettres minuscules, même pour les mots intermédiaires.

Par exemple, supposons que le nom du fichier est **MyClass.java**. Ceci signifie qu'il peut y avoir une et une seule classe **public** dans ce fichier, et que le nom de cette classe doit être **MyClass** (y compris les majuscules et minuscules) :

```
package mypackage;
public class MyClass {
  // . . .
```

Maintenant, si quelqu'un veut utiliser **MyClass** ou, aussi bien, une des autres classes **public** de **mypackage**, il doit utiliser le mot-clé **import** pour avoir à sa disposition le ou les noms définis dans **mypackage**. L'autre solution est de donner le nom complet :

```
mypackage.MyClass m = new mypackage.MyClass();
```

Le mot-clé **import** peut rendre ceci beaucoup plus propre :

```
import mypackage.*;
// . . .
MyClass m = new MyClass();
```

Il faut garder en tête que ce que permettent les mots-clés **package** et **import**, en tant que concepteur de bibliothèque, c'est de diviser l'espace de nommage global afin d'éviter le conflit de nommages, indépendamment de combien il y a de personnes qui vont sur Internet écrire des classes en Java.

Créer des noms de packages uniques

On pourrait faire remarquer que, comme un package n'est jamais réellement « empaqueté » *[packaged]* dans un seul fichier, un package pourrait être fait de nombreux fichiers .class et les choses pourraient devenir un peu désordonnées. Pour éviter ceci, une chose logique à faire est de placer tous les fichiers **.class** d'un package donné dans un même répertoire ; c'est-à-dire utiliser à son avantage la structure hiérarchique des fichiers définie par le système d'exploitation. C'est un des moyens par lequel Java traite le problème du désordre ; on verra l'autre moyen plus tard lorsque l'u-

tilitaire **jar** sera présenté.

Réunir les fichiers d'un package dans un même répertoire résoud deux autres problèmes : créer des noms de packages uniques, et trouver ces classes qui pourraient être enfouies quelque part dans la structure d'un répertoire. Ceci est réalisé, comme présenté dans le chapitre 2, en codant le chemin où se trouvent les fichiers **.class** dans le nom du **package**. Le compilateur force cela ; aussi, par convention, la première partie du nom de **package** est le nom de domaine Internet du créateur de la classe, renversé. Comme les noms de domaines Internet sont garantis uniques, *si* on suit cette convention, il est garanti que notre nom de **package** sera unique et donc qu'il n'y aura jamais de conflit de noms (c'est-à-dire, jusqu'à ce qu'on perde son nom de domaine au profit de quelqu'un qui commencerait à écrire du code Java avec les mêmes noms de répertoires). Bien entendu, si vous n'avez pas votre propre nom de domaine vous devez fabriquer une combinaison improbable (telle que votre prénom et votre nom) pour créer des noms de packages uniques. Si vous avez décidé de publier du code Java, cela vaut la peine d'effectuer l'effort relativement faible d'obtenir un nom de domaine.

La deuxième partie de cette astuce consiste à résoudre le nom de package à l'intérieur d'un répertoire de sa machine, de manière à ce que lorsque le programme Java s'exécute et qu'il a besoin de charger le fichier **.class** (ce qu'il fait dynamiquement, à l'endroit du programme où il doit créer un objet de cette classe particulière, ou la première fois qu'on accède à un membre static de la classe), il puisse localiser le répertoire où se trouve le fichier **.class**.

L'interpréteur Java procède de la manière suivante. D'abord il trouve la variable d'environnement CLASSPATH (positionnée à l'aide du système d'exploitation, parfois par le programme d'installation qui installe Java ou un outil Java sur votre machine). CLASSPATH contient un ou plusieurs répertoires qui sont utilisés comme racines de recherche pour les fichiers **.class**. En commençant à cette racine, l'interpréteur va prendre le nom de package et remplacer chaque point par un « slash » pour construire un nom de chemin depuis la racine CLASSPATH (de sorte que **package foo.bar.baz** devient **foo\bar\baz** ou **foo/bar/baz** ou peut-être quelque chose d'autre, selon votre système d'exploitation). Ceci est ensuite concaténé avec les diverses entrées du CLASSPATH. C'est là qu'il recherche le fichier .class portant le nom correspondant à la classe qu'on est en train d'essayer de créer (il recherche aussi certains répertoires standards relativement à l'endroit où se trouve l'interpréteur Java).

Pour comprendre ceci, prenons mon nom de domaine, qui est **bruceeckel.com**. En l'inversant, **com.bruceeckel** établit le nom global unique pour mes classes (les extensions com, edu, org, etc... étaient auparavant en majuscules dans les packages Java, mais ceci a été changé en Java 2 de sorte que le nom de package est entièrement en minuscules). Je peux ensuite subdiviser ceci en décidant de créer un répertoire simplement nommé **simple**, de façon à obtenir un nom de package :

```
package com.bruceeckel.simple;
```

Maintenant ce nom de package peut être utilisé comme un espace de nommage de couverture pour les deux fichiers suivants :

```
//: com:bruceeckel:simple:Vector.java
// Création d'un package.
package com.bruceeckel.simple;
public class Vector {
  public Vector() {
    System.out.println(
```

```
        "com.bruceeckel.util.Vector");
  }
} ///:~
```

Lorsque vous créerez vos propres packages, vous allez découvrir que l'instruction package doit être le premier code non-commentaire du fichier. Le deuxième fichier ressemble assez au premier :

```
//: com:bruceeckel:simple:List.java
// Création d'un package.
package com.bruceeckel.simple;
public class List {
  public List() {
    System.out.println(
      "com.bruceeckel.util.List");
  }
} ///:~
```

Chacun de ces fichiers est placé dans le sous-répertoire suivant dans mon système :

```
C:\DOC\JavaT\com\bruceeckel\simple
```

Dans ce nom de chemin, on peut voir le nom de package **com.bruceeckel.simple**, mais qu'en est-il de la première partie du chemin ? Elle est prise en compte dans la variable d'environnement CLASSPATH, qui est, sur ma machine :

```
CLASSPATH=.;D:\JAVA\LIB;C:\DOC\JavaT
```

On voit que le CLASSPATH peut contenir plusieurs chemins de recherche.

Il y a toutefois une variante lorsqu'on utilise des fichiers JAR. Il faut mettre également le nom du fichier JAR dans le classpath, et pas seulement le chemin où il se trouve. Donc pour un JAR nommé **grape.jar** le classpath doit contenir :

```
CLASSPATH=.;D:\JAVA\LIB;C:\flavors\grape.jar
```

Une fois le classpath défini correctement, le fichier suivant peut être placé dans n'importe quel répertoire :

```
//: c05:LibTest.java
// Utilise la bibliothèque.
import com.bruceeckel.simple.*;
public class LibTest {
  public static void main(String[] args) {
    Vector v = new Vector();
    List l = new List();
  }
} ///:~
```

Lorsque le compilateur rencontre l'instruction **import**, il commence à rechercher dans les répertoires spécifiés par CLASSPATH, recherchant le sous-répertoire com\bruceeckel\simple, puis re-

cherchant les fichiers compilés de noms appropriés (Vector.class pour Vector et List.class pour List). Remarquez que chacune des classes et méthodes utilisées de **Vector** et **List** doivent être **public**.

Positionner le CLASSPATH a posé tellement de problèmes aux utilisateurs débutants de Java (ça l'a été pour moi quand j'ai démarré) que Sun a rendu le JDK un peu plus intelligent dans Java 2. Vous verrez, quand vous l'installerez, que vous pourrez compiler et exécuter des programmes Java de base même si vous ne positionnez pas de CLASSPATH. Pour compiler et exécuter le package des sources de ce livre (disponible sur le CD ROM livré avec ce livre, ou sur *www.BruceEckel.com)*, vous devrez cependant faire quelques modifications de votre CLASSPATH (celles-ci sont expliquées dans le package de sources).

Collisions

Que se passe-t-il si deux bibliothèques sont importées à l'aide de * et qu'elles contiennent les mêmes noms ? Par exemple, supposons qu'un programme fasse ceci :

```
import com.bruceeckel.simple.*;
import java.util.*;
```

Comme **java.util.*** contient également une classe **Vector**, ceci crée une collision potentielle. Cependant, tant qu'on n'écrit pas le code qui cause effectivement la collision, tout va bien ; c'est une chance, sinon on pourrait se retrouver à écrire une quantité de code importante pour éviter des collisions qui n'arriveraient jamais.

La collision se produit si maintenant on essaye de créer un **Vector** :

```
Vector v = new Vector();
```

A quelle classe **Vector** ceci se réfère-t-il ? Le compilateur ne peut pas le savoir, et le lecteur non plus. Le compilateur se plaint et nous oblige à être explicite. Si je veux le **Vector** Java standard, par exemple, je dois dire :

```
java.util.Vector v =new java.util.Vector();
```

Comme ceci (avec le CLASSPATH) spécifie complètement l'emplacement de ce Vector, il n'y a pas besoin d'instruction **import java.util.***, à moins que j'utilise autre chose dans **java.util**.

Une bibliothèque d'outils personnalisée

Avec ces connaissances, vous pouvez maintenant créer vos propres bibliothèques d'outils pour réduire ou éliminer les duplications de code. On peut par exemple créer un alias pour **System.out.println( )** pour réduire la frappe. Ceci peut faire partie d'un package appelé **tools** :

```
//: com:bruceeckel:tools:P.java
// Les raccourcis P.rint & P.rintln
package com.bruceeckel.tools;
public class P {
  public static void print(String s) {
    System.out.print(s);
  }
  public static void println(String s) {
```

```
    System.out.println(s);
  }
} ///:~
```

On peut utiliser ce raccourci pour imprimer une **String** , soit avec une nouvelle ligne (**P.-rintln( )**) , ou sans (**P.rint( )**).

Vous pouvez deviner que l'emplacement de ce fichier doit être dans un répertoire qui commence à un des répertoire du CLASSPATH, puis continue dans **com/bruceeckel/tools**. Après l'avoir compilé, le fichier **P.class** peut être utilisé partout sur votre système avec une instruction **import** :

```
//: c05:ToolTest.java
// Utilise la bibliothèque tools
import com.bruceeckel.tools.*;
public class ToolTest {
  public static void main(String[] args) {
    P.println("Available from now on!");
    P.println("" + 100); // Le force à être une String
    P.println("" + 100L);
    P.println("" + 3.14159);
  }
} ///:~
```

Remarquez que chacun des objets peut facilement être forcé dans une représentation **String** en les plaçant dans une expression **String** ; dans le cas ci-dessus, le fait de commencer l'expression avec une **String** vide fait l'affaire. Mais ceci amène une observation intéressante. Si on appelle **System.out.println(100)**, cela fonctionne sans le transformer *[cast]* en String. Avec un peu de surcharge *[overloading]*, on peut amener la classe P à faire ceci également (c'est un exercice à la fin de ce chapitre).

A partir de maintenant, chaque fois que vous avez un nouvel utilitaire intéressant, vous pouvez l'ajouter au répertoire **tools** (ou à votre propre répertoire **util** ou **tools**).

Une caractéristique manquant à Java est la *compilation conditionelle* du C, qui permet de changer un commutateur pour obtenir un comportement différent sans rien changer d'autre au code. La raison pour laquelle cette caractéristique n'a pas été retenue en Java est probablement qu'elle est surtout utilisée en C pour résoudre des problèmes de portabilité : des parties du code sont compilées différemment selon la plateforme pour laquelle le code est compilé. Comme le but de Java est d'être automatiquement portable, une telle caractéristique ne devrait pas être nécessaire.

Il y a cependant d'autres utilisations valables de la compilation conditionnelle. Un usage très courant est le debug de code. Les possibilités de debug sont validées pendant le développement, et invalidées dans le produit livré. Allen Holub (*www.holub.com*) a eu l'idée d'utiliser les packages pour imiter la compilation conditionnelle. Il l'a utilisée pour créer une version du très utile *mécanisme d'affirmation [assertion mechanism]* du C, dans lequel on peut dire « ceci devrait être vrai » ou « ceci devrait être faux » et si l'instruction n'est pas d'accord avec cette affirmation, on en sera averti. Un tel outil est très utile pendant la phase de debuggage.

Voici une classe qu'on utilisera pour debugger :

```
//: com:bruceeckel:tools:debug:Assert.java
// Outil d'affirmation pour debugger
package com.bruceeckel.tools.debug;
public class Assert {
  private static void perr(String msg) {
    System.err.println(msg);
  }
  public final static void is_true(boolean exp) {
    if(!exp) perr("Assertion failed");
  }
  public final static void is_false(boolean exp){
    if(exp) perr("Assertion failed");
  }
  public final static void
  is_true(boolean exp, String msg) {
    if(!exp) perr("Assertion failed: " + msg);
  }
  public final static void
  is_false(boolean exp, String msg) {
    if(exp) perr("Assertion failed: " + msg);
  }
} ///:~
```

Cette classe encapsule simplement des tests booléens, qui impriment des messages d'erreur en cas d'échec. Dans le chapitre 10, on apprendra à utiliser un outil plus sophistiqué pour traiter les erreurs, appelé *traitement d'exceptions [exception handling]*, mais la méthode **perr( )** conviendra bien en attendant.

La sortie est affichée sur la console dans le flux d'*erreur standard [standard error stream]*en écrivant avec **System.err**.

Pour utiliser cette classe, ajoutez cette ligne à votre programme :

```
import com.bruceeckel.tools.debug.*;
```

Pour supprimer les affirmations afin de pouvoir livrer le code, une deuxième classe **Assert** est créée, mais dans un package différent :

```
//: com:bruceeckel:tools:Assert.java
// Suppression de la sortie de l'affirmation
// pour pouvoir livrer le programme.
package com.bruceeckel.tools;
public class Assert {
  public final static void is_true(boolean exp){}
  public final static void is_false(boolean exp){}
  public final static void
  is_true(boolean exp, String msg) {}
  public final static void
  is_false(boolean exp, String msg) {}
```

```
} ///:~
```

Maintenant si on change l'instruction **import** précédente en :

```
import com.bruceeckel.tools.*;
```

le programme n'affichera plus d'affirmations. Voici un exemple :

```
//: c05:TestAssert.java
// Démonstration de l'outil d'affirmation.
// Mettre en commentaires ce qui suit, et enlever
// le commentaire de la ligne suivante
// pour modifier le comportement de l'affirmation :
import com.bruceeckel.tools.debug.*;
// import com.bruceeckel.tools.*;
public class TestAssert {
  public static void main(String[] args) {
    Assert.is_true((2 + 2) == 5);
    Assert.is_false((1 + 1) == 2);
    Assert.is_true((2 + 2) == 5, "2 + 2 == 5");
    Assert.is_false((1 + 1) == 2, "1 +1 != 2");
  }
} ///:~
```

En changeant le package qui est importé, on change le code de la version debug à la version de production. Cette technique peut être utilisée pour toutes sortes de code conditionnel.

Avertissement sur les packages

Il faut se rappeler que chaque fois qu'on crée un package, on spécifie implicitement une structure de répertoire en donnant un nom à un package. Le package *doit* se trouver dans le répertoire indiqué par son nom, qui doit être un répertoire qui peut être trouvé en partant du CLASSPATH. L'utilisation du mot-clé **package** peut être un peu frustrante au début, car à moins d'adhérer à la règle nom-de-package - chemin-de-répertoire, on obtient un tas de messages mystérieux à l'exécution signalant l'impossibilité de trouver une classe donnée, même si la classe est là dans le même répertoire. Si vous obtenez un message de ce type, essayez de mettre en commentaire l'instruction **package**, et si ça tourne vous saurez où se trouve le problème.

Les spécificateurs d'accès Java

Quand on les utilise, les spécificateurs d'accès Java **public**, **protected,** et **private** sont placés devant la définition de chaque membre de votre classe, qu'il s'agisse d'un champ ou d'une méthode. Chaque spécificateur d'accès contrôle l'accès pour uniquement cette définition particulière. Ceci est différent du C++, où un spécificateur d'accès contrôle toutes les définitions le suivant jusqu'à ce qu'un autre spécificateur d'accès soit rencontré.

D'une façon ou d'une autre, toute chose a un type d'accès spécifié. Dans les sections suivantes, vous apprendrez à utiliser les différents types d'accès, à commencer par l'accès par défaut.

« Friendly »

Que se passe-t-il si on ne précise aucun spécificateur d'accès, comme dans tous les exemples avant ce chapitre ? L'accès par défaut n'a pas de mot-clé, mais on l'appelle couramment « friendly » (amical). Cela veut dire que toutes les autres classes du package courant ont accès au membre amical, mais pour toutes les classes hors du package le membre apparaît **private**. Comme une unité de compilation -un fichier- ne peut appartenir qu'à un seul package, toutes les classes d'une unité de compilation sont automatiquement amicales entre elles. De ce fait, on dit aussi que les éléments amicaux ont un *accès de package[package access]*.

L'accès amical permet de grouper des classes ayant des points communs dans un package, afin qu'elles puissent facilement interagir entre elles. Quand on met des classes ensemble dans un package (leur accordant de ce fait un accès mutuel à leurs membres amicaux, c'est à dire en les rendant « amicaux ») on « possède » le code de ce package. Il est logique que seul le code qu'on possède ait un accès amical à un autre code qu'on possède. On pourrait dire que l'accès amical donne une signification ou une raison pour regrouper des classes dans un package. Dans beaucoup de langages l'organisation des définitions dans des fichiers peut être faite tant bien que mal, mais en Java on est astreint à les organiser d'une manière logique. De plus, on exclura probablement les classes qui ne devraient pas avoir accès aux classes définies dans le package courant.

La classe contrôle quel code a accès à ses membres. Il n'y a pas de moyen magique pour « entrer par effraction ». Le code d'un autre package ne peut pas dire « Bonjour, je suis un ami de **Bob** ! » et voir les membres **protected**, friendly, et **private** de **Bob.** La seule façon d'accorder l'accès à un membre est de :

1. Rendre le membre **public**. Ainsi tout le monde, partout, peut y accéder.
2. Rendre le membre amical en n'utilisant aucun spécificateur d'accès, et mettre les autres classes dans le même package. Ainsi les autres classes peuvent accéder à ce membre.
3. Comme on le verra au Chapitre 6, lorsque l'héritage sera présenté, une classe héritée peut avoir accès à un membre **protected** ou **public** (mais pas à des membres **private**). Elle peut accéder à des membres amicaux seulement si les deux classes sont dans le même package. Mais vous n'avez pas besoin de vous occuper de cela maintenant.
4. Fournir des méthodes « accessor/mutator » (connues aussi sous le nom de méthodes « get/set ») qui lisent et changent la valeur. Ceci est l'approche la plus civilisée en termes de programmation orientée objet, et elle est fondamentale pour les JavaBeans, comme on le verra dans le Chapitre 13.

public : accès d'interface

Lorsqu'on utilise le mot-clé **public**, cela signifie que la déclaration du membre qui suit immédiatement **public** est disponible pour tout le monde, en particulier pour le programmeur client qui utilise la bibliothèque. Supposons qu'on définisse un package **dessert** contenant l'unité de compilation suivante :

```
//: c05:dessert:Cookie.java
// Création d'une bibliothèque.
package c05.dessert;
public class Cookie {
```

```
  public Cookie() {
   System.out.println("Cookie constructor");
  }
  void bite() { System.out.println("bite"); }
} ///:~
```

Souvenez-vous, **Cookie.java** doit se trouver dans un sous-répertoire appelé **dessert**, en dessous de **c05** (qui signifie le Chapitre 5 de ce livre) qui doit être en-dessous d'un des répertoire du CLASSPATH. Ne faites pas l'erreur de croire que Java va toujours considérer le répertoire courant comme l'un des points de départ de la recherche. Si vous n'avez pas un « **.** » comme un des chemins dans votre CLASSPATH, Java n'y regardera pas.

Maintenant si on crée un programme qui utilise **Cookie** :

```
//: c05:Dinner.java
// Utilise la bibliothèque.
import c05.dessert.*;

public class Dinner {
  public Dinner() {
   System.out.println("Dinner constructor");
  }
  public static void main(String[] args) {
    Cookie x = new Cookie();
    //! x.bite(); // Ne peut y accéder
  }
} ///:~
```

on peut créer un objet **Cookie**, puisque son constructeur est **public** et que la classe est **public**. (Nous allons regarder plus en détail le concept d'une classe **public** plus tard.) Cependant, le membre **bite()** est inaccessible depuis **Dinner.java** car **bite()** est amical uniquement à l'intérieur du package **dessert**.

Le package par défaut

Vous pourriez être surpris de découvrir que le code suivant compile, bien qu'il semble ne pas suivre les règles :

```
//: c05:Cake.java
// Accède à une classe dans
// une unité de compilation séparée.
class Cake {
  public static void main(String[] args) {
    Pie x = new Pie();
    x.f();
  }
} ///:~
```

Dans un deuxième fichier, dans le même répertoire :

```
//: c05:Pie.java
// L'autre classe.
class Pie {
  void f() { System.out.println("Pie.f()"); }
} ///:~
```

On pourrait à première vue considérée ces fichiers comme complètement étrangers l'un à l'autre, et cependant **Cake** est capable de créer l'objet **Pie** et d'appeler sa méthode **f()** ! (Remarquez qu'il faut avoir « . » dans le CLASSPATH pour que ces fichiers puissent être compilés.) On penserait normalement que **Pie** et **f()** sont amicaux et donc non accessibles par **Cake**. Ils *sont* amicaux, cette partie-là est exacte. La raison pour laquelle ils sont accessible dans **Cake.java** est qu'ils sont dans le même répertoire et qu'ils n'ont pas de nom de package explicite. Java traite de tels fichiers comme faisant partie du « package par défaut » pour ce répertoire, et donc amical pour tous les autres fichiers du répertoire.

private : ne pas toucher !

Le mot-clé **private** signifie que personne ne peut accéder à ce membre à part la classe en question, dans les méthodes de cette classe. Les autres classes du package ne peuvent accéder à des membres **private**, et c'est donc un peu comme si on protégeait la classe contre soi-même. D'un autre côté il n'est pas impossible qu'un package soit codé par plusieurs personnes, et dans ce cas **private** permet de modifier librement ce membre sans que cela affecte une autre classe dans le même package.

L'accès « amical » par défaut dans un package cache souvent suffisamment les choses ; souvenez-vous, un membre « amical » est inaccessible à l'utilisateur du package. C'est parfait puisque l'accès par défaut est celui qu'on utilise normalement (et c'est celui qu'on obtient si on oublie d'ajouter un contrôle d'accès). De ce fait, on ne pensera normalement qu'aux accès aux membres qu'on veut explicitement rendre **public** pour le programmeur client, et donc on pourrait penser qu'on n'utilise pas souvent le mot-clé **private** puisqu'on peut s'en passer (ceci est une différence par rapport au C++). Cependant, il se fait que l'utilisation cohérente de **private** est très importante, particulièrement lors du multithreading (comme on le verra au Chapitre 14).

Voici un exemple de l'utilisation de **private** :

```
//: c05:IceCream.java
// Démontre le mot-clé "private"
class Sundae {
  private Sundae() {}
  static Sundae makeASundae() {
    return new Sundae();
  }
}

public class IceCream {
  public static void main(String[] args) {
    //! Sundae x = new Sundae();
    Sundae x = Sundae.makeASundae();
  }
```

```
} ///:~
```

Ceci montre un cas où **private** vient à propos : on peut vouloir contrôler comment un objet est créé et empêcher quelqu'un d'accéder à un constructeur en particulier (ou à tous). Dans l'exemple ci-dessus, on ne peut pas créer un objet **Sundae** à l'aide de son constructeur ; il faut plutôt utiliser la méthode **makeASundae()** qui le fera pour nous [33].

Toute méthode dont on est certain qu'elle n'est utile qu'à cette classe peut être rendue **private**, pour s'assurer qu'on ne l'utilisera pas ailleurs dans le package, nous interdisant ainsi de la modifier ou de la supprimer. Rendre une méthode **private** garantit cela.

Ceci est également vrai pour un champ **private** dans une classe. A moins qu'on ne doive exposer une implémentation sous-jacente (ce qui est beaucoup plus rare qu'on ne pourrait penser), on devrait déclarer tous les membres private. Cependant, le fait que la référence à un objet est **private** dans une classe ne signifie pas qu'un autre objet ne puisse avoir une référence **public** à cet objet (voir l'annexe A pour les problèmes au sujet de l'aliasing).

protected : « sorte d'amical »

Le spécificateur d'accès **protected** demande un effort de compréhension . Tout d'abord, il faut savoir que vous n'avez pas besoin de comprendre cette partie pour continuer ce livre jusqu'à l'héritage (Chapitre 6). Mais pour être complet, voici une brève description et un exemple d'utilisation de **protected**.

Le mot-clé **protected** traite un concept appelé *héritage*, qui prend une classe existante et ajoute des membres à cette classe sans modifier la classe existante, que nous appellerons la *classe de base*. On peut également changer le comportement des membres d'une classe existants. Pour hériter d'une classe existante, on dit que le nouvelle classe **extends** (étend) une classe existante, comme ceci :

```
class Foo extends Bar {
```

Le reste de la définition de la classe est inchangé.

Si on crée un nouveau package et qu'on hérite d'une classe d'un autre package, les seuls membres accessibles sont les membres **public** du package d'origine. (Bien sûr, si on effectue l'héritage dans le *même* package, on a l'accès normal à tous les membres « amicaux » du package.) Parfois le créateur de la classe de base veut, pour un membre particulier, en accorder l'accès dans les classes dérivées mais pas dans le monde entier en général. C'est ce que **protected** fait. Si on reprend le fichier **Cookie.java**, la classe suivante ne peut pas accéder au membre « amical » :

```
//: c05:ChocolateChip.java
// Nepeut pas accéder à un membre amical
// dans une autre classe.
import c05.dessert.*;

public class ChocolateChip extends Cookie {
  public ChocolateChip() {
   System.out.println(
     "ChocolateChip constructor");
  }
```

```
  public static void main(String[] args) {
    ChocolateChip x = new ChocolateChip();
    //! x.bite(); // Ne peut pas accéder à bite
  }
} ///:~
```

Une des particularités intéressantes de l'héritage est que si la méthode **bite()** existe dans la classe Cookie, alors elle existe aussi dans toute classe héritée de Cookie. Mais comme **bite()** est « amical » dans un autre package, il nous est inaccessible dans celui-ci. Bien sûr, on pourrait le rendre **public**, mais alors tout le monde y aurait accès, ce qui ne serait peut-être pas ce qu'on veut. Si on modifie la classe **Cookie** comme ceci :

```
public class Cookie {
  public Cookie() {
    System.out.println("Cookie constructor");
  }
  protected void bite() {
    System.out.println("bite");
  }
}
```

alors **bite()** est toujours d'accès « amical » dans le package **dessert**, mais il est aussi accessible à tous ceux qui héritent de **Cookie**. Cependant, il n'est pas **public**.

Le contrôle d'accès est souvent appelé *cacher l'implémentation [implementation hiding]* . L'enveloppement *[wrapping]* des données et méthodes à l'intérieur des classes combiné au masquage de l'implémentation est souvent appelé *encapsulation* [34]. Le résultat est un type de données avec des caractéristiques et des comportements.

Le contrôle d'accès pose des limites sur un type de données pour deux raisons importantes. La première est de déterminer ce que les programmeurs clients peuvent et ne peuvent pas utiliser. On peut construire les mécanismes internes dans la structure sans se préoccuper du risque que les programmeurs clients prennent ces mécanismes internes comme faisant partie de l'interface qu'ils doivent utiliser.

Ceci amène directement à la deuxième raison, qui est de séparer l'interface de son implémentation. Si la structure est utilisée dans un ensemble de programmes, mais que les programmeurs clients ne peuvent qu'envoyer des messages à l'interface **public**, alors on peut modifier tout ce qui *n'est pas* **public** (c'est à dire « amical », **protected**, ou **private**) sans que cela nécessite des modifications du code client.

Nous sommes maintenant dans le monde de la programmation orientée objet, dans lequel une **class** est en fait la description d' « une classe d'objets », comme on décrirait une classe des poissons ou une classe des oiseaux. Tout objet appartenant à cette classe partage ces caractéristiques et comportements. La classe est une description de la façon dont les objets de ce type vont nous apparaître et se comporter.

Dans le langage de POO d'origine, Simula-67 , le mot-clé **class** était utilisé pour décrire un nouveau type de données. Le même mot-clé a été repris dans la plupart des langages orientés objet. Ceci est le point central de tout le langage : la création de nouveaux types de données qui sont plus que simplement des boîtes contenant des données et des méthodes.

La classe est le concept de POO fondamental en Java. C'est l'un des mots-clés qui *ne sera pas* mis en gras dans ce livre, ça devient lourd pour un mot aussi souvent répété que « class ».

Pour plus de clarté, il est préférable d'utiliser un style de création des classes qui place les membres **public** au début, suivi par les membres **protected**, amicaux et **private**. L'avantage de ceci est que l'utilisateur de la classe peut voir ce qui est important pour lui (les membres **public**, parce qu'on peut y accéder de l'extérieur du fichier), en lisant depuis le début et en s'arrêtant lorsqu'il rencontre les membres non-**public**, qui font partie de l'implémentation interne :

```
public class X {
  public void pub1( ) { /* . . . */ }
  public void pub2( ) { /* . . . */ }
  public void pub3( ) { /* . . . */ }
  private void priv1( ) { /* . . . */ }
  private void priv2( ) {/* . . . */ }
  private void priv3( ) { /* . . . */ }
  private int i;
  // . . .
}
```

Ceci ne la rendra que partiellement plus lisible parce que l'interface et l'implémentation sont encore mélangés. C'est-à-dire qu'on voit toujours le code source (l'implémentation) parce qu'il est là dans la classe. Cependant, la documentation sous forme de commentaires supportée par javadoc (décrite au Chapitre 2) diminue l'importance de la lisibilité du code par le programmeur client. Afficher l'interface au consommateur d'une classe est normalement le travail du *class browser*, un outil dont le travail consiste à inspecter toutes les classes disponibles et de montrer ce qu'on peut en faire (c'est à dire quels membres sont disponibles) de façon pratique. Au moment où vous lisez ceci, les browsers devraient faire partie de tout bon outil de développement Java.

L'accès aux classes

En Java, les spécificateurs d'accès peuvent aussi être utilisés pour déterminer quelles classes d'une bibliothèque seront accessibles aux utilisateurs de cette bibliothèque. Si on désire qu'une classe soit disponible pour un programmeur client, on place le mot-clé **public** quelque part devant l'accolade ouvrante du corps de la classe. Ceci permet de contrôler le fait même qu'un programmeur client puisse créer un objet de cette classe.

Pour contrôler l'accès à la classe, le spécificateur doit apparaître avant le mot-clé **class**. Donc on peut dire :

```
public class Widget {
```

Maintenant, si le nom de la bibliothèque est **mylib**, tout programmeur client peut accéder à **Widget** en disant

```
import mylib.Widget;
```

ou

```
import mylib.*;
```

Il y a cependant un ensemble de contraintes supplémentaires :

1. Il ne peut y avoir qu'une seule classe **public** par unité de compilation (fichier). L'idée est que chaque unité de compilation a une seule interface publique représentée par cette classe **public** . Elle peut avoir autant de classes « amicales » de support qu'on veut. Si on a plus d'une classe **public** dans une unité de compilation, le compilateur générera un message d'erreur.

2. Le nom de la classe **public** doit correspondre exactement au nom du fichier contenant l'unité de compilation, y compris les majuscules et minuscules. Par exemple pour **Widget**, le nom du fichier doit être **Widget.java**, et pas **widget.java** ou **WIDGET.java**. Là aussi on obtient des erreurs de compilation s'ils ne correspondent pas.

3. Il est possible, bien que non habituel, d'avoir une unité de compilation sans aucune classe **public**. Dans ce cas, on peut appeler le fichier comme on veut.

Que se passe-t-il si on a une classe dans **mylib** qu'on utilise uniquement pour accomplir les tâches effectuées par **Widget** ou une autre classe public de **mylib** ? On ne veut pas créer de documentation pour un programmeur client, et on pense que peut-être plus tard on modifiera tout et qu'on refera toute la classe en lui en substituant une nouvelle. Pour garder cette possibilité, il faut s'assurer qu'aucun programmeur client ne devienne dépendant des détails d'implémentation cachés dans **mylib**. Pour réaliser ceci il suffit d'enlever le mot-clé **public** de la classe, qui devient dans ce cas amicale. (Cette classe ne peut être utilisée que dans ce package.)

Remarquez qu'une classe ne peut pas être **private** (cela ne la rendrait accessible à personne d'autre que cette classe), ou **protected** [35]. Il n'y a donc que deux choix pour l'accès aux classes : « amical » ou **public**. Si on ne veut pas que quelqu'un d'autre accède à cette classe, on peut rendre tous les constructeurs **private**, ce qui empêche tout le monde de créer un objet de cette classe, à part soi-même dans un membre static de la classe [36]. Voici un exemple :

```
//: c05:Lunch.java
// Démontre les spécificateurs d'accès de classes.
// Faire une classe effectivement private
// avec des constructeurs private :
class Soup {
  private Soup() {}
  // (1) Permettre la création à l'aide d'une méthode static :
  public static Soup makeSoup() {
    return new Soup();
  }
  // (2) Créer un objet static et
  // retourner une référence à la demande.
  // (le patron "Singleton"):
  private static Soup ps1 = new Soup();
  public static Soup access() {
    return ps1;
  }
  public void f() {}
}

class Sandwich { // Utilise Lunch
```

```
  void f() { new Lunch(); }
}

// Une seule classe public autorisée par fichier :
public class Lunch {
  void test() {
// Ne peut pas faire ceci ! Constructeur privé :
    //! Soup priv1 = new Soup();
    Soup priv2 = Soup.makeSoup();
    Sandwich f1 = new Sandwich();
    Soup.access().f();
  }
} ///:~
```

Jusqu'ici, la plupart des méthodes retournaient soit **void** soit un type primitif, ce qui fait que la définition :

```
public static Soup access() {
    return ps1;
  }
```

pourrait paraître un peu confuse. Le mot devant le nom de la méthode (**access**) dit ce que retourne la méthode. Jusqu'ici cela a été le plus souvent **void**, ce qui signifie qu'elle ne retourne rien. Mais on peut aussi retourner la référence à un objet, comme c'est le cas ici. Cette méthode retourne une référence à un objet de la classe **Soup**.

La classe **Soup** montre comment empêcher la création directe d'une classe en rendant tous les constructeurs **private**. Souvenez-vous que si vous ne créez pas explicitement au moins un constructeur, le constructeur par défaut (un constructeur sans arguments) sera créé pour vous. En écrivant ce constructeur par défaut, il ne sera pas créé automatiquement. En le rendant private, personne ne pourra créer un objet de cette classe. Mais alors comment utilise-t-on cette classe ? L'exemple ci-dessus montre deux possibilités. Premièrement, une méthode **static** est créée, elle créee un nouveau **Soup** et en retourne la référence. Ceci peut être utile si on veut faire des opérations supplémentaires sur **Soup** avant de le retourner, ou si on veut garder un compteur du nombre d'objets **Soup** créés (peut-être pour restreindre leur population).

La seconde possibilité utilise ce qu'on appelle un *patron de conception [design pattern],* qui est expliqué dans *Thinking in Patterns with Java*, téléchargeable sur *www.BruceEckel.com*. Ce patron particulier est appelé un « singleton » parce qu'il n'autorise la création que d'un seul objet. L'objet de classe Soup est créé comme un membre **static private** de **Soup**, ce qui fait qu'il n'y en a qu'un seul, et on ne peut y accéder qu'à travers la méthode **public access()**.

Comme mentionné précédemment, si on ne met pas de spécificateur d'accès il est « amical » par défaut. Ceci signifie qu'un objet de cette classe peut être créé par toute autre classe du package, mais pas en dehors du package (souvenez-vous que tous les fichiers dans le même répertoire qui n'ont pas de déclaration **package** explicite font implicitement partie du package par défaut pour ce répertoire). Cependant, si un membre **static** de cette classe est **public**, le programmeur client peut encore accéder à ce membre **static** même s'il ne peut pas créer un objet de cette classe.

Résumé

Dans toute relation il est important d'avoir des limites respectées par toutes les parties concernées. Lorsqu'on crée une bibliothèque, on établit une relation avec l'utilisateur de cette bibliothèque (le programmeur client) qui est un autre programmeur, mais qui construit une application ou qui utilise la bibliothèque pour créer une bibliothèque plus grande.

Sans règles, les programmeurs clients peuvent faire tout ce qu'ils veulent des membres de la classe, même si vous préféreriez qu'ils ne manipulent pas directement certains des membres. Tout est à découvert dans le monde entier.

Ce chapitre a décrit comment les classes sont construites pour former des bibliothèques ; d'abord, comment un groupe de classes est empaqueté *[packaged]*dans une bibliothèque, et ensuite comment la classe contrôle l'accès à ses membres.

On estime qu'un projet programmé en C commence à s'écrouler losqu'il atteint 50K à 100K de lignes de code, parce que le C a un seul « espace de nommage », et donc les noms commencent à entrer en conflit, provoquant un surcroît de gestion. En Java, le mot-clé **package**, le principe de nommage des packages et le mot-clé **import** donnent un contrôle complet sur les noms, et le problème de conflit de nommage est facilement évité.

Il y a deux raisons pour contrôler l'accès aux membres. La première est d'écarter les utilisateurs des outils qu'ils ne doivent pas utiliser ; outils qui sont nécessaires pour les traitements internes des types de données, mais qui ne font pas partie de l'interface dont les utilisateurs ont besoin pour résoudre leurs problèmes particuliers. Donc créer des méthodes et des champs **private** est un service rendu aux utilisateurs parce qu'ils peuvent facilement voir ce qui est important pour eux et ce qu'ils peuvent ignorer. Cela leur simplifie la compréhension de la classe.

La deuxième raison, et la plus importante, pour le contrôle d'accès, est de permettre au concepteur de bibliothèque de modifier les fonctionnements internes de la classe sans se soucier de la façon dont cela peut affecter le programmeur client. On peut construire une classe d'une façon, et ensuite découvrir qu'une restructuration du code améliorera grandement sa vitesse. Si l'interface et l'implémentation sont clairement séparées et protégées, on peut y arriver sans forcer l'utilisateur à réécrire son code.

Les spécificateurs d'accès en Java donnent un contrôle précieux au créateur d'une **class**. Les utilisateurs de la **class** peuvent voir clairement et exactement ce qu'ils peuvent utiliser et ce qu'ils peuvent ignorer. Encore plus important, on a la possiblité de garantir qu'aucun utilisateur ne deviendra dépendant d'une quelconque partie de l'implémentation d'une **class**. Sachant cela en tant que créateur d'une **class**, on peut en modifier l'implémentation en sachant qu'aucun programmeur client ne sera affecté par les modifications car il ne peut accéder à cette partie de la **class**.

Lorsqu'on a la possibilité de modifier l'implémentation, on peut non seulement améliorer la conception plus tard, mais on a aussi le droit de faire des erreurs . Quelles que soient les soins que vous apportez à votre planning et à votre conception, vous ferez des erreurs. Savoir qu'il est relativement peu dangereux de faire ces erreurs veut dire que vous ferez plus d'expériences, vous apprendrez plus vite, et vous finirez plus vite votre projet.

L'interface publique d'une classe est ce que l'utilisateur *voit*, donc c'est la partie qui doit être « correcte » lors de l'analyse et la conception. Et même ici vous avez encore quelques possibilités de modifications. Si vous n'avez pas la bonne interface du premier coup, vous pouvez *ajouter* des méthodes , pour autant que vous ne supprimiez pas celles que les programmeurs clients ont déjà utilisé dans leur code.

Exercices

Les solutions des exercices sélectionnés sont disponibles dans le document électronique *The Thinking in Java Annotated Solution Guide*, disponible à un prix raisonnable sur *www.BruceEckel.com*.

1. Ecrivez un programme qui crée un objet ArrayList sans importer explicitement **java.util.***.

2. Dans la section nommée « package : l'unité de bibliothèque », transformez les fragments de code concernant **mypackage** en un ensemble de fichiers Java qui peuvent être compilés et qui tournent.

3. Dans la section appelée « Collisions », prenez les fragments de code et transformez les en programme, et vérifiez qu'en fait les collisions arrivent.

4. Généralisez la classe P définie dans ce chapitre en ajoutant toutes les versions surchargées de **rint( )** et **rintln( )** nécessaires pour gérer tous les types Java de base.

5. Modifiez l'instruction import de **TestAssert.java** pour autoriser ou inhiber le mécanisme d'assertion.

6. Créez une classe avec **public**, **private**, **protected,** et des membres de données et des méthodes « amicaux ». Faites attention au fait que des classes dans un même répertoire font partie d'un package « par défaut ».

7. Créez une classe avec des données **protected**. Créez une deuxième classe dans le même fichier, qui a une méthode qui manipule les données **protected** de la première classe.

8. Modifiez la classe Cookie comme spécifié dans la section "**protected** : **sorte d'amical** ». Vérifiez que **bite()** n'est pas **public**.

9. Dans la section nommée « Accès aux classes » vous trouverez des fragments de code décrivant **mylib** et**Widget**. Créez cette bibliothèque, et ensuite créez un **Widget** dans une classe qui ne fait pas partie du package **mylib**.

10. Créez un nouveau répertoire et modifiez votre CLASSPATH pour inclure ce nouveau répertoire. Copiez le fichier **P.class** (produit par la compilation de **com.bruceeckel.tools.P.java**) dans votre nouveau répertoire et changez ensuite le nom du fichier, la classe **P** à l'intérieur, et les noms de méthodes (vous pouvez aussi ajouter des sorties supplémentaires pour observer comment cela fonctionne). Créez un autre programme dans un autre répertoire, qui utilise votre nouvelle classe.

11. En suivant la forme de l'exemple **Lunch.java**, créez une classe appelée **ConnectionManager** qui gère un tableau fixe d'objets **Connection**. Le programmeur client ne doit pas pouvoir créer explicitement des objets **Connection**, mais doit seulement pouvoir les obtenir à l'aide d'une méthode static dans **ConnectionManager**. Lorsque le **ConnectionManager** tombe à court d'objets, il retourne une référence **null**. Testez la classe dans **main()**.

12. Créez le fichier suivant dans le répertoire c05/local (supposé être dans votre CLASSPATH) :

```
///: c05:local:PackagedClass.java
package c05.local;
class PackagedClass {
```

```
  public PackagedClass() {
    System.out.println(
      "Creating a packaged class");
  }
}///:~
```

13. Créez enuite le fichier suivant dans un répertoire autre que c05 :

```
///: c05:foreign:Foreign.java
package c05.foreign;
import c05.local.*;
public class Foreign {
  public static void main (String[] args) {
    PackagedClass pc = new PackagedClass();
  }
} ///:~
```

Expliquez pourquoi le compilateur génère une erreur. Le fait de mettre la classe **Foreign** dans le package **c05.local** changerait-il quelque chose ?

[32] Rien en Java n'oblige à utiliser un interpréteur. Il existe des compilateurs Java de code natif qui génèrent un seul fichier exécutable.

[33] Il y a un autre effet dans ce cas. Comme le constructeur par défaut est le seul défini, et qu'il est **private**, il empêchera l'héritage de cette classe. (Un sujet qui sera présenté dans le Chapitre 6.)

[34] Cependant, on parle souvent aussi d'encapsulation pour le seul fait de cacher l'implémentation.

[35] En fait, une *classe interne[inner class]* peut être private ou protected, mais il s'agit d'un cas particulier. Ceux-ci seront présentés au Chapitre 7.

[36] On peut aussi le faire en héritant (Chapitre 6) de cette classe.

Chapitre 6 - Réutiliser les classes

Une des caractéristiques les plus excitantes de Java est la réutilisation du code. Mais pour être vraiment révolutionnaire, il faut faire plus que copier du code et le changer.

C'est l'approche utilisée dans les langages procéduraux comme C, et ça n'a pas très bien fonctionné. Comme tout en Java, la solution réside dans les classes. On réutilise du code en créant de nouvelles classes, mais au lieu de les créer depuis zéro, on utilise les classes que quelqu'un a construit et testé.

L'astuce est d'utiliser les classes sans détériorer le code existant. Dans ce chapitre nous verrons deux manières de faire. La première est plutôt directe : On crée simplement des objets de nos classes existantes à l'intérieur de la nouvelle classe. Ça s'appelle *la composition*, parce que la nouvelle classe se compose d'objets de classes existantes. On réutilise simplement les fonctionnalités du code et non sa forme.

La seconde approche est plus subtile. On crée une nouvelle classe comme un *type* d'une classe existante. On prend littéralement la forme d'une classe existante et on lui ajoute du code sans modifier la classe existante. Cette magie s'appelle l'*héritage*, et le compilateur fait le plus gros du travail. L'héritage est une des pierres angulaires de la programmation par objets, et a bien d'autres implications qui seront explorées au chapitre 7.

Il s'avère que beaucoup de la syntaxe et du comportement sont identiques pour la composition et l' héritage (cela se comprend parce qu'ils sont tous deux des moyens de construire des nouveaux types à partir de types existants). Dans ce chapitre, nous apprendrons ces mécanismes de réutilisation de code.

Syntaxe de composition

Jusqu'à maintenant, la composition a été utilisée assez fréquemment. On utilise simplement des références sur des objets dans de nouvelles classes. Par exemple, supposons que l'on souhaite un objet qui contient plusieurs objets de type **String**, quelques types primitifs et un objet d'une autre classe. Pour les objets, on met des références à l'intérieur de notre nouvelle classe, mais on définit directement les types primitifs:

```
// ! c06:SprinklerSystem.java
// La composition pour réutiliser du code.

class WaterSource {
  private String s;
  WaterSource() {
    System.out.println("WaterSource()");
    s = new String("Constructed");
  }
  public String toString() { return s; }
}

public class SprinklerSystem {
  private String valve1, valve2, valve3, valve4;
```

```
  WaterSource source;
  int i;
  float f;
  void print() {
    System.out.println("valve1 = " + valve1);
    System.out.println("valve2 = " + valve2);
    System.out.println("valve3 = " + valve3);
    System.out.println("valve4 = " + valve4);
    System.out.println("i = " + i);
    System.out.println("f = " + f);
    System.out.println("source = " + source);
  }
  public static void main(String[] args) {
    SprinklerSystem x = new SprinklerSystem();
    x.print();
  }
} ///:~
```

Une des méthodes définies dans **WaterSource** est spéciale : **toString( )**. Vous apprendrez plus tard que chaque type non primitif a une méthode **toString( )**, et elle est appelée dans des situations spéciales lorsque le compilateur attend une **String** alors qu'il ne trouve qu'un objet. Donc dans une expression:

```
System.out.println("source = " + source);
```

le compilateur voit que vous essayez d'ajouter un objet **String** ("**source** = ") à un **WaterSource**. Ceci n'a pas de sens parce qu'on peut seulement ajouter une **String** à une autre **String**, donc il se dit qu'il va convertir **source** en une **String** en appelant **toString( )** ! Après avoir fait cela il combine les deux **String**s et passe la **String** résultante à **System.out.println( )**. Dès qu'on veut permettre ce comportement avec une classe qu'on crée, il suffit simplement de définir une méthode **toString( )**.

Au premier regard, on pourrait supposer — Java étant sûr et prudent comme il l'est — que le compilateur construirait automatiquement des objets pour chaque référence dans le code ci-dessus ; par exemple, en appelant le constructeur par défaut pour **WaterSource** pour initialiser **source**. Le résultat de l'instruction d'impression affiché est en fait :

```
valve1 = null
valve2 = null
valve3 = null
valve4 = null
i = 0
f = 0.0
source = null
```

Les types primitifs qui sont des champs d'une classe sont automatiquement initialisés à zéro, comme précisé dans le chapitre 2. Mais les références objet sont initialisées à **null**, et si on essaye d'appeler des méthodes pour l'un d'entre eux, on obtient une exception. En fait il est bon (et utile) qu'on puisse les afficher sans lancer d'exception.

On comprend bien que le compilateur ne crée pas un objet par défaut pour chaque référence parce que cela induirait souvent une surcharge inutile. Si on veut initialiser les références, on peut faire :

1. Au moment où les objets sont définis. Cela signifie qu'ils seront toujours initialisés avant que le constructeur ne soit appelé ;
2. Dans le constructeur pour la classe ;
3. Juste avant d'utiliser l'objet, ce qui est souvent appelé *initialisation paresseuse.*

Cela peut réduire la surcharge dans les situations où l'objet n'a pas besoin d'être créé à chaque fois.

Les trois approches sont montrées ici:

```
// ! c06:Bath.java
// Initialisation dans le constructeur avec composition.

class Soap {
  private String s;
  Soap() {
    System.out.println("Soap()");
    s = new String("Constructed");
  }
  public String toString() { return s; }
}

public class Bath {
  private String
    // Initialisation au moment de la définition:
    s1 = new String("Happy"),
    s2 = "Happy",
    s3, s4;
  Soap castille;
  int i;
  float toy;
  Bath() {
    System.out.println("Inside Bath()");
    s3 = new String("Joy");
    i = 47;
    toy = 3.14f;
    castille = new Soap();
  }
  void print() {
    // Initialisation différée:
    if(s4 == null)
      s4 = new String("Joy");
    System.out.println("s1 = " + s1);
    System.out.println("s2 = " + s2);
    System.out.println("s3 = " + s3);
    System.out.println("s4 = " + s4);
```

```
    System.out.println("i = " + i);
    System.out.println("toy = " + toy);
    System.out.println("castille = " + castille);
  }
  public static void main(String[] args) {
    Bath b = new Bath();
    b.print();
  }
} ///:~
```

Notez que dans le constructeur de **Bath** une instruction est exécutée avant que toute initialisation ait lieu. Quand on n'initialise pas au moment de la définition, il n'est pas encore garanti qu'on exécutera une initialisation avant qu'on envoie un message à un objet — sauf l'inévitable exception à l'exécution.

Ici la sortie pour le programme est :

```
Inside Bath()
Soap()
s1 = Happy
s2 = Happy
s3 = Joy
s4 = Joy
i = 47
toy = 3.14
castille = Constructed
```

Quand **print( )** est appelé, il remplit **s4** donc tout les champs sont proprement initialisés au moment où ils sont utilisés.

La syntaxe de l'héritage

L'héritage est une partie primordiale de Java (et des langages de programmation par objets en général). Il s'avère qu'on utilise toujours l'héritage quand on veut créer une classe, parce qu'à moins d'hériter explicitement d'une autre classe, on hérite implicitement de la classe racine standard **Object**.

La syntaxe de composition est évidente, mais pour réaliser l'héritage il y a une forme distinctement différente. Quand on hérite, on dit « Cette nouvelle classe est comme l'ancienne classe ». On stipule ceci dans le code en donnant le nom de la classe comme d'habitude, mais avant l'accolade ouvrante du corps de la classe, on met le mot clé **extends** suivi par le nom de *la classe de base*. Quand on fait cela, on récupère automatiquement toutes les données membres et méthodes de la classe de base. Voici un exemple:

```
// ! c06:Detergent.java
// Syntaxe d'héritage & propriétés.

class Cleanser {
  private String s = new String("Cleanser");
```

```
  public void append(String a) { s += a; }
  public void dilute() { append(" dilute()"); }
  public void apply() { append(" apply()"); }
  public void scrub() { append(" scrub()"); }
  public void print() { System.out.println(s); }
  public static void main(String[] args) {
    Cleanser x = new Cleanser();
    x.dilute(); x.apply(); x.scrub();
    x.print();
  }
}

public class Detergent extends Cleanser {
  // Change une méthode:
  public void scrub() {
    append(" Detergent.scrub()");
    super.scrub(); // Appel de la version de la classe de base
  }
  // Ajoute une méthode à l'interface:
  public void foam() { append(" foam()"); }
  // Test de la nouvelle classe:
  public static void main(String[] args) {
    Detergent x = new Detergent();
    x.dilute();
    x.apply();
    x.scrub();
    x.foam();
    x.print();
    System.out.println("Testing base class:");
    Cleanser.main(args);
  }
} ///:~
```

Ceci montre un certain nombre de caractéristiques. Premièrement, dans **Cleanser** la méthode **append( )**, les **String**s sont concaténées dans **s** en utilisant l'opérateur +=, qui est l'un des opérateurs (avec « + ») que les créateurs de Java « ont surchargé » pour travailler avec les **String**s.

Deuxièmement, tant **Cleanser** que **Detergent** contiennent une méthode **main( )**. On peut créer une **main( )** pour chacune de nos classes, et il est souvent recommandé de coder de cette manière afin de garder le code de test dans la classe. Même si on a beaucoup de classes dans un programme, seulement la méthode **main( )** pour une classe invoquée sur la ligne de commande sera appelée. Aussi longtemps que **main( )** est **public**, il importe peu que la classe dont elle fait partie soit **public** ou non. Donc dans ce cas, quand on écrit **java Detergent**, **Detergent.main( )** sera appelée. Mais on peut également écrire **java Cleanser** pour invoquer **Cleanser.main( )**, même si **Cleanser** n'est pas une classe **public**. Cette technique de mettre une **main( )** dans chaque classe permet de tester facilement chaque classe. Et on n'a pas besoin d'enlever la méthode **main( )** quand on a finit de tester ; on peut la laisser pour tester plus tard.

Ici, on peut voir que **Detergent.main( )** appelle **Cleanser.main( )** explicitement, en passant

les même arguments depuis la ligne de commande (quoiqu'il en soit, on peut passer n'importe quel tableau de **String**).

Il est important que toutes les méthodes de **Cleanser** soient **public**. Il faut se souvenir que si on néglige tout modifieur d'accès, par défaut l'accès sera « friendly », lequel permet d'accéder seulement aux membres du même package. Donc, *au sein d'un même package*, n'importe qui peut utiliser ces méthodes s'il n'y a pas de spécificateur d'accès. **Detergent** n'aurait aucun problème, par exemple. Quoiqu'il en soit, si une classe d'un autre package devait hériter de **Cleanser** il pourrait accéder seulement aux membres **public**. Donc pour planifier l'héritage, en règle générale mettre tous les champs **private** et toutes les méthodes **public** (les membres **protected** permettent également d'accéder depuis une classe dérivée ; nous verrons cela plus tard). Bien sûr, dans des cas particuliers on devra faire des ajustements, mais cela est une règle utile.

Notez que **Cleanser** contient un ensemble de méthodes dans son interface : **append( )**, **dilute( )**, **apply( )**, **scrub( )**, et **print( )**. Parce que **Detergent** est *dérivé de* **Cleanser** (à l'aide du mot-clé **extends**) il récupère automatiquement toutes les méthodes de son interface , même si elles ne sont pas toutes définies explicitement dans **Detergent**. On peut penser à l'héritage *comme à une réutilisation de l'interface* (l'implémentation vient également avec elle, mais ceci n'est pas le point principal).

Comme vu dans **scrub( )**, il est possible de prendre une méthode qui a été définie dans la classe de base et la modifier. Dans ce cas, on pourrait vouloir appeler la méthode de la classe de base dans la nouvelle version. Mais à l'intérieur de **scrub( )** on ne peut pas simplement appeler **scrub( )**, car cela produirait un appel récursif, ce qui n'est pas ce que l'on veut. Pour résoudre ce problème, Java a le mot-clé **super** qui réfère à la super classe de la classe courante. Donc l'expression **super.scrub( )** appelle la version de la classe de base de la méthode **scrub( )**.

Quand on hérite, on n'est pas tenu de n'utiliser que les méthodes de la classe de base. On peut également ajouter de nouvelles méthodes à la classe dérivée exactement de la manière dont on met une méthode dans une classe : il suffit de la définir. La méthode **foam( )** en est un exemple.

Dans **Detergent.main( )** on peut voir que pour un objet **Detergent** on peut appeler toutes les méthodes disponible dans **Cleanser** aussi bien que dans **Detergent** (e.g., **foam( )**).

Initialiser la classe de base

Depuis qu'il y a deux classes concernées - la classe de base et la classe dérivée - au lieu d'une seule, il peut être un peu troublant d'essayer d'imaginer l'objet résultant produit par la classe dérivée. De l'extérieur, il semble que la nouvelle classe a la même interface que la classe de base et peut-être quelques méthodes et champs additionnels. Mais l'héritage ne se contente pas simplement de copier l'interface de la classe de base. Quand on créée un objet de la classe dérivée, il contient en lui un *sous-objet* de la classe de base. Ce sous-objet est le même que si on crée un objet de la classe de base elle-même. C'est simplement que, depuis l'extérieur, le sous-objet de la classe de base est enrobé au sein de l'objet de la classe dérivée.

Bien sûr, il est essentiel que le sous-objet de la classe de base soit correctement initialisé et il y a un seul moyen de garantir cela: exécuter l'initialisation dans le constructeur, en appelant la constructeur de la classe de base, lequel a tous les connaissances et les privilèges appropriés pour exécuter l'initialisation de la classe de base. Java insère automatiquement les appels au constructeur de la classe de base au sein du constructeur de la classe dérivée. L'exemple suivant montre comment cela fonctionne avec 3 niveaux d'héritage :

```
// ! c06:Cartoon.java
// Appels de constructeur durant l'initialisation

class Art {
  Art() {
    System.out.println("Art constructor");
  }
}

class Drawing extends Art {
  Drawing() {
    System.out.println("Drawing constructor");
  }
}

public class Cartoon extends Drawing {
  Cartoon() {
    System.out.println("Cartoon constructor");
  }
  public static void main(String[] args) {
    Cartoon x = new Cartoon();
  }
} ///:~
```

La sortie de ce programme montre les appels automatiques:

```
Art constructor
Drawing constructor
Cartoon constructor
```

On peut voir que la construction commence par la classe la plus haute dans la hiérarchie, donc la classe de base est initialisée avant que les constructeurs de la classe dérivée puisse y accéder.

Même si on ne crée pas de constructeur pour **Cartoon( )**, le compilateur fournira un constructeur.

Constructeurs avec paramètres

L'exemple ci-dessus a des constructeurs par défaut ; ils n'ont pas de paramètres. C'est facile pour le compilateur d'appeler ceux-ci parce qu'il n'y a pas de questions à se poser au sujet des arguments à passer. Si notre classe n'a pas de paramètres par défaut, ou si on veut appeler le constructeur d'une classe de base avec paramètre, on doit explicitement écrire les appels au contructeur de la classe de base en utilisant le mot clé super ainsi que la liste de paramètres appropriée : **super** et la liste de paramètres appropriée:

```
// ! c06:Chess.java
// Héritage, constructeurs et paramètres.
```

```
class Game {
  Game(int i) {
    System.out.println("Game constructor");
  }
}

class BoardGame extends Game {
  BoardGame(int i) {
    super(i);
    System.out.println("BoardGame constructor");
  }
}

public class Chess extends BoardGame {
  Chess() {
    super(11);
    System.out.println("Chess constructor");
  }
  public static void main(String[] args) {
    Chess x = new Chess();
  }
} ///:~
```

Si on n'appelle pas le constructeur de la classe de base dans **BoardGame( )**, le compilateur va se plaindre qu'il ne peut pas trouver le constructeur de la forme **Game( )**. De plus, l'appel du constructeur de la classe de base *doit* être la première chose que l'on fait dans le constructeur de la classe dérivée. Le compilateur va le rappeler si on se trompe.

Comme nous venons de le préciser, le compilateur nous force à placer l'appel du constructeur de la classe de base en premier dans le constructeur de la classe dérivée. Cela signifie que rien ne peut être placé avant lui. Comme vous le verrez dans le chapitre 10, cela empêche également le constructeur de la classe dérivée d'attraper une exception qui provient de la classe de base. Ceci peut être un inconvénient de temps en temps.

Combiner composition et héritage.

Il est très classique d'utiliser ensemble la composition et l'héritage. L'exemple suivant montre la création d'une classe plus complexe, utilisant à la fois l'héritage et la composition, avec la nécessaire initialisation des contructeurs:

```
// ! c06:PlaceSetting.java
// Mélanger composition & héritage.

class Plate {
  Plate(int i) {
    System.out.println("Plate constructor");
  }
}
```

```
class DinnerPlate extends Plate {
  DinnerPlate(int i) {
    super(i);
    System.out.println(
      "DinnerPlate constructor");
  }
}

class Utensil {
  Utensil(int i) {
    System.out.println("Utensil constructor");
  }
}

class Spoon extends Utensil {
  Spoon(int i) {
    super(i);
    System.out.println("Spoon constructor");
  }
}

class Fork extends Utensil {
  Fork(int i) {
    super(i);
    System.out.println("Fork constructor");
  }
}

class Knife extends Utensil {
  Knife(int i) {
    super(i);
    System.out.println("Knife constructor");
  }
}

// Une manière culturelle de faire quelque chose:
class Custom {
  Custom(int i) {
    System.out.println("Custom constructor");
  }
}

public class PlaceSetting extends Custom {
  Spoon sp;
  Fork frk;
  Knife kn;
```

```
  DinnerPlate pl;
  PlaceSetting(int i) {
    super(i + 1);
    sp = new Spoon(i + 2);
    frk = new Fork(i + 3);
    kn = new Knife(i + 4);
    pl = new DinnerPlate(i + 5);
    System.out.println(
      "PlaceSetting constructor");
  }
  public static void main(String[] args) {
    PlaceSetting x = new PlaceSetting(9);
  }
} ///:~
```

Tant que le compilateur nous force à initialiser les classes de base, et requiert que nous le fassions directement au début du constructeur, il ne vérifie pas que nous initialisons les objets membres, donc nous devons nous souvenir de faire attention à cela.

Garantir un nettoyage propre

Java ne possède pas le concept C++ de *destructeur*, une méthode qui est automatiquement appelée quand un objet est détruit. La raison est probablement qu'en Java la pratique est simplement d'oublier les objets plutôt que les détruire, laissant le ramasse-miette réclamer la mémoire selon les besoins.

Souvent cela convient, mais il existe des cas où votre classe pourrait, durant son existence, exécuter des tâches nécessitant un nettoyage. Comme mentionné dans le chapitre 4, on ne peut pas savoir quand le ramasse-miettes sera exécuté, ou s'il sera appelé. Donc si on veut nettoyer quelque chose pour une classe, on doit écrire une méthode particulière, et être sûr que l'usager sait qu'il doit appeler cette méthode. Par dessus tout, comme décrit dans le chapitre 10 (« Gestion d'erreurs avec les exceptions ») - on doit se protéger contre une exception en mettant un tel nettoyage dans une clause **finally**.

Considérons un exemple d'un système de conception assisté par ordinateur qui dessine des images sur l'écran:

```
// ! c06:CADSystem.java
// Assurer un nettoyage propre.
import java.util.*;

class Shape {
  Shape(int i) {
    System.out.println("Shape constructor");
  }
  void cleanup() {
    System.out.println("Shape cleanup");
  }
```

```
}

class Circle extends Shape {
  Circle(int i) {
    super(i);
    System.out.println("Drawing a Circle");
  }
  void cleanup() {
    System.out.println("Erasing a Circle");
    super.cleanup();
  }
}

class Triangle extends Shape {
  Triangle(int i) {
    super(i);
    System.out.println("Drawing a Triangle");
  }
  void cleanup() {
    System.out.println("Erasing a Triangle");
    super.cleanup();
  }
}

class Line extends Shape {
  private int start, end;
  Line(int start, int end) {
    super(start);
    this.start = start;
    this.end = end;
    System.out.println("Drawing a Line : " +
        start + ", " + end);
  }
  void cleanup() {
    System.out.println("Erasing a Line : " +
        start + ", " + end);
    super.cleanup();
  }
}

public class CADSystem extends Shape {
  private Circle c;
  private Triangle t;
  private Line[] lines = new Line[10];
  CADSystem(int i) {
    super(i + 1);
    for(int j = 0; j < 10; j++)
```

```
      lines[j] = new Line(j, j*j);
    c = new Circle(1);
    t = new Triangle(1);
    System.out.println("Combined constructor");
  }
  void cleanup() {
    System.out.println("CADSystem.cleanup()");
    // L'ordre de nettoyage est l'inverse
    // de l'ordre d'initialisation
    t.cleanup();
    c.cleanup();
    for(int i = lines.length - 1; i >= 0; i--)
      lines[i].cleanup();
    super.cleanup();
  }
  public static void main(String[] args) {
    CADSystem x = new CADSystem(47);
    try {
      // Code et gestion des exceptions...
    } finally {
      x.cleanup();
    }
  }
} ///:~
```

Tout dans ce système est une sorte de **Shape** (lequel est une sorte d'**Object** puisqu'il hérite implicitement de la classe racine). Chaque classe redéfinit la méthode **cleanup( )** de **Shape** en plus d'appeler la méthode de la classe de base en utilisant **super**. Les classes **Shape** spécifiques — **Circle**, **Triangle** et **Line**— ont toutes des constructeurs qui « dessinent », bien que n'importe quelle méthode appelée durant la durée de vie d'un objet pourrait être responsable de faire quelque chose qui nécessite un nettoyage. Chaque classe possède sa propre méthode **cleanup( )** pour restaurer les choses de la manière dont elles étaient avant que l'objet n'existe.

main( ), on peut noter deux nouveaux mot-clés qui ne seront pas officiellement introduits avant le chapitre 10 : **try** et **finally**. Le mot-clé **try** indique que le bloc qui suit (délimité par les accolades) est une *région gardée*, ce qui signifie qu'elle a un traitement particulier. Un de ces traitements particuliers est que le code dans la clause **finally** suivant la région gardée est *toujours* exécuté, quelle que soit la manière dont le bloc **try** termine. Avec la gestion des exceptions, il est possible de quitter le bloc **try** de manière non ordinaire. Ici la clause finally dit de toujours appeler **cleanup( )** pour x, quoiqu'il arrive. Ces mot-clés seront expliqués plus en profondeur dans le chapitre 10.

Notez que dans votre méthode cleanup vous devez faire attention à l'ordre d'appel pour les méthodes cleanup de la classe de base et celle des objet-membres au cas où un des sous-objets dépend des autres. En général, vous devriez suivre la même forme qui est imposée pour le compilateur C++ sur ses destructeurs : premièrement exécuter tout le nettoyage spécifique à votre classe, dans l'ordre inverse de la création. En général, cela nécessite que les éléments de la classe de base soient encore viable. Ensuite appeler la méthode cleanup de la classe de base, comme démontré ici.

Il y a beaucoup de cas pour lesquels le problème de nettoyage n'est pas un problème ; on

laisse le ramasse-miettes faire le travail. Mais quand on doit le faire explicitement, diligence et attention sont requis.

L'ordre du ramasse-miettes

Il n'y a pas grand chose sur quoi on puisse se fier quand il s'agit de ramasse-miettes. Le ramasse-miette peut ne jamais être appelé. S'il l'est, il peut réclamer les objets dans n'importe quel ordre. Il est préférable de ne pas se fier au ramasse-miette pour autre chose que libérer la mémoire. Si on veut que le nettoyage ait lieu, faites vos propre méthodes de nettoyage et ne vous fiez pas à **finalize( )**. Comme mentionné dans le chapitre 4, Java peut être forcé d'appeler tous les finalizers.

Cacher les noms

Seuls les programmeurs C++ pourraient être surpris par le masquage de noms, étant donné que le fonctionnement est différent dans ce langage. Si une classe de base Java a un nom de méthode qui est surchargé plusieurs fois, redéfinir ce nom de méthode dans une sous-classe ne cachera aucune des versions de la classe de base. Donc la surcharge fonctionne sans savoir si la méthode était définie à ce niveau ou dans une classe de base:

```
// ! c06:Hide.java
// Surchage le nom d'une méthode de la classe de base
// dans une classe dérivée ne cache pas
// les versions de la classe de base.

class Homer {
  char doh(char c) {
    System.out.println("doh(char)");
    return 'd';
  }
  float doh(float f) {
    System.out.println("doh(float)");
    return 1.0f;
  }
}

class Milhouse {}

class Bart extends Homer {
  void doh(Milhouse m) {}
}

class Hide {
  public static void main(String[] args) {
    Bart b = new Bart();
    b.doh(1); // doh(float) utilisé
    b.doh('x');
    b.doh(1.0f);
    b.doh(new Milhouse());
```

```
  }
} ///:~
```

Comme nous le verrons dans le prochain chapitre, il est beaucoup plus courant de surcharger les méthodes de même nom et utilisant exactement la même signature et le type retour que dans la classe de base. Sinon cela peut être source de confusion. C'est pourquoi le C++ ne le permet pas, pour empêcher de faire ce qui est probablement une erreur.

Choisir la composition à la place de l'héritage

La composition et l'héritage permettent tous les deux de placer des sous-objets à l'intérieur de votre nouvelle classe. Vous devriez vous demander quelle est la différence entre les deux et quand choisir l'une plutôt que l'autre.

La composition est généralement utilisée quand on a besoin des caractéristiques d'une classe existante dans une nouvelle classe, mais pas son interface. On inclut un objet donc on peut l'utiliser pour implémenter une fonctionnalité dans la nouvelle classe, mais l'utilisateur de la nouvelle classe voit l'interface qu'on a défini et non celle de l'objet inclus. Pour ce faire, il suffit d'inclure des objets **private** de classes existantes dans la nouvelle classe.

Parfois il est sensé de permettre à l'utilisateur d'une classe d'accéder directement à la composition de notre nouvelle classe ; pour ce faire on déclare les objets membres **public**. Les objets membres utilisent l'implémentation en se cachant les uns des autres, ce qui est une bonne chose. Quand l'utilisateur sait qu'on assemble un ensemble de parties, cela rend l'interface plus facile à comprendre. Un objet **car** (voiture en anglais) est un bon exemple:

```
// ! c06:Car.java
// Composition avec des objets publics.

class Engine {
  public void start() {}
  public void rev() {}
  public void stop() {}
}

class Wheel {
  public void inflate(int psi) {}
}

class Window {
  public void rollup() {}
  public void rolldown() {}
}

class Door {
  public Window window = new Window();
  public void open() {}
  public void close() {}
}
```

```
public class Car {
  public Engine engine = new Engine();
  public Wheel[] wheel = new Wheel[4];
  public Door left = new Door(),
      right = new Door(); // 2-door
  public Car() {
    for(int i = 0; i < 4; i++)
      wheel[i] = new Wheel();
  }
  public static void main(String[] args) {
    Car car = new Car();
    car.left.window.rollup();
    car.wheel[0].inflate(72);
  }
} ///:~
```

Du fait que la composition de la voiture fait partie de l'analyse du problème (et non pas simplement de la conception sous-jacente), rendre les membre **public**s aide le programmeur client à comprendre comment utiliser la classe et nécessite moins de complexité de code pour le créateur de la classe. Quoiqu'il en soit, gardez à l'esprit que c'est un cas spécial et qu'en général on devrait définir les champs **privés**.

Quand on hérite, on prend la classe existante et on en fait une version spéciale. En général, cela signifie qu'on prend une classe d'usage général et on l'adapte à un cas particulier. Avec un peu de bon sens, vous verrez que ça n'a pas de sens de composer une voiture en utilisant un objet véhicule. Une voiture ne contient pas un véhicule, c'*est* un véhicule. La relation *est-un* s'exprime avec l'héritage et la relation *a-un* s'exprime avec la composition.

protected

Maintenant que nous avons introduit l'héritage, le mot clé **protected** prend finalement un sens. Dans un monde idéal, les membres **private** devraient toujours être des membres **private** purs et durs, mais dans les projets réels il arrive souvent qu'on veuille cacher quelque chose au monde au sens large et qu'on veuille permettre l'accès pour les membres des classes dérivées. Le mot clé **protected** est un moyen pragmatique de faire. Il dit « Ceci est **private** en ce qui concerne la classe utilisatrice, mais c'est disponible pour quiconque hérite de cette classe ou appartient au même package ». C'est pourquoi **protected** en Java est automatiquement « friendly ».

La meilleure approche est de laisser les membres de données **private**. Vous devriez toujours préserver le droit de changer l'implémentation sous jacente. Ensuite vous pouvez permettre l'accès contrôlé pour les héritiers de votre classe à travers les méthodes **protected** :

```
// ! c06:Orc.java
// Le mot clé protected .
import java.util.*;

class Villain {
  private int i;
```

```
  protected int read() { return i; }
  protected void set(int ii) { i = ii; }
  public Villain(int ii) { i = ii; }
  public int value(int m) { return m*i; }
}

public class Orc extends Villain {
  private int j;
  public Orc(int jj) { super(jj); j = jj; }
  public void change(int x) { set(x); }
} ///:~
```

On peut voir que **change( )** a accès à **set( )** parce qu'il est **protected**.

Développement incrémental

Un des avantages de l'héritage est qu'il supporte le *développement incrémental* en permettant d'ajouter du code sans créer de bogues dans le code existant. Ceci permet également d'isoler les nouveaux bogues dans le nouveau code. En héritant d'une classe existante et fonctionnelle et en ajoutant des données membres et des méthodes et en redéfinissant des méthodes existantes, on laisse le code existant - que quelqu'un d'autre peut encore utiliser - inchangé et non bogué. Si un bogue survient, on sait alors qu'il est dans le nouveau code, lequel est beaucoup plus rapide et facile à lire que si on avait modifié le code existant.

Il est plutôt surprenant de voir que les classes sont séparées proprement. Nous n'avons même pas besoin du code source des méthodes afin de pouvoir les utiliser. Au pire, on importe simplement un package. Ceci est vrai à la fois pour l'héritage et la composition.

Il est important de réaliser que le développement d'un programme est un processus incrémental, exactement comme l'apprentissage humain. On peut analyser autant que l'on veut, mais on ne connaîtra pas toutes les réponses au démarrage d'un projet. Vous aurez beaucoup plus de succès et de retour immédiat si vous commencez par faire « grandir » votre projet comme un organisme organique et évolutif plutôt que de le construire comme un gratte-ciel en verre.

Bien que l'héritage pour l'expérimentation puisse être une technique utile, à un moment donné les choses se stabilisent et vous devez jeter un regard neuf à votre hiérarchie de classes pour la réduire en une structure plus logique. Souvenons nous qu'en dessous de tout, l'héritage est utilisé pour exprimer une relation qui dit : « Cette nouvelle classe est *du type de* l'ancienne classe ». Votre programme ne devrait pas être concerné par la manipulation de bits ici ou là, mais par la création et la manipulation d'objets de différents types afin d'exprimer un modèle dans les termes propres à l'espace du problème.

Transtypage ascendant

L'aspect le plus important de l'héritage n'est pas qu'il fournisse des méthodes pour les nouvelles classes. C'est la relation exprimée entre la nouvelle classe et la classe de base. Cette relation peut être résumée en disant : « La nouvelle classe est *un type de* la classe existante ».

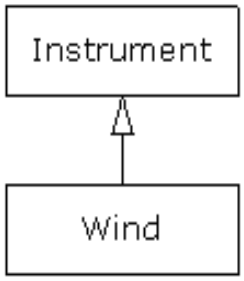
Cette description n'est pas simplement une manière amusante d'expliquer l'héritage - c'est supporté directement par le langage. Comme exemple, considérons une classe de base appelée

Instrument qui représente les instruments de musique et une classe dérivée appelée **Wind**. Puisque l'héritage signifie que toutes les méthodes de la classe de base sont également disponibles pour la classe dérivée, n'importe quel message envoyé à la classe de base peut également être envoyé à la classe dérivée. Si la classe **Instrument** a une méthode **play( )**, les instruments **Wind** également. Cela signifie qu'il est exact de dire qu'un objet **Wind** est également un type de **Instrument**. L'exemple suivant montre comment le compilateur implémente cette notion :

```
// ! c06:Wind.java
// Héritage & transtypage ascendant.
import java.util.*;

class Instrument {
  public void play() {}
  static void tune(Instrument i) {
    // ...
    i.play();
  }
}

// Les objets Wind sont des instruments
// parce qu'ils ont la même interface:
class Wind extends Instrument {
  public static void main(String[] args) {
    Wind flute = new Wind();
    Instrument.tune(flute); // Transtypage ascendant
  }
} ///:~
```

Pourquoi le transtypage ascendant ?

La raison de ce terme est historique et basée sur la manière dont les diagrammes d'héritage ont été traditionnellement dessinés : avec la racine au sommet de la page, et grandissant vers le bas. Bien sûr vous pouvez dessiner vos diagrammes de la manière que vous trouvez le plus pratique. Le diagramme d'héritage pour **Wind.java** est :



Transtyper depuis une classe dérivée vers la classe de base nous déplace **vers le haut** dans le diagramme, on fait donc communément référence à un *transtypage ascendant*. Le transtypage ascendant est toujours sans danger parce qu'on va d'un type plus spécifique vers un type plus général. La classe dérivée est un sur-ensemble de la classe de base. Elle peut contenir plus de méthodes que

la classe de base, mais elle contient *au moins* les méthodes de la classe de base. La seule chose qui puisse arriver à une classe pendant le transtypage ascendant est de perdre des méthodes et non en gagner. C'est pourquoi le compilateur permet le transtypage ascendant sans transtypage explicite ou une notation spéciale.

On peut également faire l'inverse du transtypage ascendant, appelé *transtypage descendant*, mais cela génère un dilemme qui est le sujet du chapitre 12.

Composition à la place de l'héritage revisité

En programmation orienté objet, la manière la plus probable pour créer et utiliser du code est simplement de mettre des méthodes et des données ensemble dans une classe puis d'utiliser les objets de cette classe. On utilisera également les classes existantes pour construire les nouvelles classes avec la composition. Moins fréquemment on utilisera l'héritage. Donc bien qu'on insiste beaucoup sur l'héritage en apprenant la programmation orientée objet, cela ne signifie pas qu'on doive l'utiliser partout où l'on peut. Au contraire, on devrait l'utiliser avec parcimonie, seulement quand il est clair que l'héritage est utile. Un des moyens les plus clairs pour déterminer si on doit utiliser la composition ou l'héritage est de se demander si on aura jamais besoin de faire un transtypage ascendant de la nouvelle classe vers la classe de base. Si on doit faire un transtypage ascendant, alors l'héritage est nécessaire, mais si on n'a pas besoin de faire un transtypage ascendant, alors il faut regarder avec attention pour savoir si on a besoin de l'héritage. Le prochain chapitre (polymorphisme) fournit une des plus excitantes raisons pour le transtypage ascendant, mais si vous vous rappelez de vous demander « Ai-je besoin de transtypage ascendant ? », vous aurez un bon outil pour décider entre composition et héritage.

Le mot clé final

Le mot clé Java **final** a des sens légèrement différents suivant le contexte, mais en général il signifie « Cela ne peut pas changer ». Vous pourriez vouloir empêcher les changements pour deux raisons : conception ou efficacité. Parce que ces deux raisons sont quelque peu différentes, il est possible de mal utiliser le mot clé **final**.

Les sections suivantes parlent des trois endroits où le mot clé **final** peut être utilisé : données, méthodes et classes.

Données finales

Beaucoup de langages de programmation ont un moyen de dire au compilateur que cette donnée est constante. Une constante est utile pour deux raisons:

1. Elle peut être une *constante lors de la compilation* qui ne changera jamais ;
2. Elle peut être une valeur initialisée à l'exécution qu'on ne veut pas changer.

Dans le cas d'une constante à la compilation, le compilateur inclut « en dur » la valeur de la constante pour tous les calculs où elle intervient ; dans ce cas, le calcul peut être effectué à la compilation, éliminant ainsi un surcoût à l'exécution. En Java, ces sortes de constantes doivent être des primitives et sont exprimées en utilisant le mot-clé **final**. Une valeur doit être donnée au moment de la définition d'une telle constante.

Un champ qui est à la fois **static** et **final** a un emplacement de stockage fixe qui ne peut pas être changé.

Quand on utilise **final** avec des objets références plutôt qu'avec des types primitifs la signification devient un peu confuse. Avec un type primitif, **final** fait de la *valeur* une constante, mais avec un objet référence, **final** fait de la « référence » une constante. Une fois la référence liée à un objet, elle ne peut jamais changer pour pointer vers un autre objet. Quoiqu'il en soit, l'objet lui même peut être modifié ; Java ne fournit pas de moyen de rendre un objet arbitraire une constante. On peut quoiqu'il en soit écrire notre classe de manière que les objets paraissent constants. Cette restriction inclut les tableaux, qui sont également des objets.

Voici un exemple qui montre les champs **final**:

```
// ! c06:FinalData.java
// L'effet de final sur les champs.

class Value {
  int i = 1;
}

public class FinalData {
  // Peut être des constantes à la compilation
  final int i1 = 9;
  static final int VAL_TWO = 99;
  // Constantes publiques typiques:
  public static final int VAL_THREE = 39;
  // Ne peuvent pas être des constantes à la compilation:
  final int i4 = (int)(Math.random()*20);
  static final int i5 = (int)(Math.random()*20);

  Value v1 = new Value();
  final Value v2 = new Value();
  static final Value v3 = new Value();
  // Tableaux:
  final int[] a = { 1, 2, 3, 4, 5, 6 };

  public void print(String id) {
    System.out.println(
      id + " : " + "i4 = " + i4 +
      ", i5 = " + i5);
  }
  public static void main(String[] args) {
    FinalData fd1 = new FinalData();
    // ! fd1.i1++; // Erreur : on ne peut pas changer la valeur
    fd1.v2.i++; // L'objet n'est pas une constante!
    fd1.v1 = new Value(); // OK -- non final
    for(int i = 0; i < fd1.a.length; i++)
      fd1.a[i]++; // L'objet n'est pas une constante!
    // ! fd1.v2 = new Value(); // Erreur : Ne peut pas
    // ! fd1.v3 = new Value(); // changer la référence
    // ! fd1.a = new int[3];
```

```
    fd1.print("fd1");
    System.out.println("Creating new FinalData");
    FinalData fd2 = new FinalData();
    fd1.print("fd1");
    fd2.print("fd2");
  }
} ///:~
```

Etant donné que **i1** et **VAL_TWO** sont des primitives **final** ayant une valeur à la compilation, elles peuvent être toutes les deux utilisées comme constantes à la compilation et ne sont pas vraiment différentes. **VAL_THREE** nous montre la manière la plus typique de définir ces constantes : **public** afin qu'elles puissent être utilisées en dehors du package, **static** pour souligner qu'il ne peut y en avoir qu'une seulement, et **final** pour dire que c'est une constante. Notez que les primitives **final static** avec des valeurs initiales constantes (ce sont des constantes à la compilation) sont nommées avec des lettre capitales par convention, avec des mots séparés par des underscores. Ce sont comme des constantes C, d'où cette convention est originaire. Notons également que **i5** ne peut pas être connu à la compilation, donc elle n'est pas en lettres capitales.

Le fait que quelque chose soit **final** ne signifie pas que sa valeur est connue à la compilation. Ceci est montré par l'initialisation de **i4** et **i5** à l'exécution en utilisant des nombres générés aléatoirement. La portion de cet exemple montre également la différence entre mettre une valeur **final static** ou non **static**. Cette différence n'est visible que quand les valeurs sont initialisées à l'exécution, tandis que les valeurs à la compilation sont traitées de même par le compilateur. Et vraisemblablement optimisées à la compilation. La différence est montrée par la sortie d'une exécution:

```
fd1 : i4 = 15, i5 = 9
Creating new FinalData
fd1 : i4 = 15, i5 = 9
fd2 : i4 = 10, i5 = 9
```

Notez que les valeurs de **i4** pour **fd1** et **fd2** sont uniques, mais la valeur de **i5** n'est pas changée en créant un second objet **FinalData**. C'est parce qu'elle est **static** et initialisée une fois pour toutes lors du chargement et non à chaque fois qu'un nouvel objet est créé.

Les variables **v1** jusqu'à **v4** montre le sens de références **final**. Comme on peut le voir dans **main()**, le fait que **v2** soit **final** ne signifie pas qu'on ne peut pas changer sa valeur. Quoiqu'il en soit, on ne peut pas réaffecter un nouvel objet à **v2**, précisément parce qu'il est **final**. C'est ce que **final** signifie pour une référence. On peut également voir que ce sens reste vrai pour un tableau, qui est une autre sorte de référence. Il n'y a aucun moyen de savoir comment rendre les références du tableau elle-mêmes **final**. Mettre les références **final** semble moins utile que mettre les primitives **final**.

Finals sans initialisation

Java permet la création de *finals sans initialisation*, qui sont des champs déclarés **final**, mais n'ont pas de valeur d'initialisation. Dans tous les cas, un final sans initialisation *doit* être initialisé avant d'être utilisé, et le compilateur doit s'en assurer. Quoiqu'il en soit, les finals sans initialisation fournissent bien plus de flexibilité dans l'usage du mot-clé **final** depuis que, par exemple, un champ **final** à l'intérieur d'une classe peut maintenant être différent pour chaque objet tout en gardant son caractère immuable. Voici un exemple:

```
// ! c06:BlankFinal.java
// Les membres des données final sans initialisation

class Poppet { }

class BlankFinal {
  final int i = 0; // Final initialisé
  final int j; // Final sans initialisation
  final Poppet p; // Référence final sans initialisation
  // Les finals doivent être initialisés
  // dans le constructeur:
  BlankFinal() {
    j = 1; // Initialise le final sans valeur initiale
    p = new Poppet();
  }
  BlankFinal(int x) {
    j = x; // Initialise le final sans valeur initiale
    p = new Poppet();
  }
  public static void main(String[] args) {
    BlankFinal bf = new BlankFinal();
  }
} ///:~
```

Vous êtes forcés d'initialiser un **final** soit avec une expression au moment de la définition, soit dans chaque constructeur. De cette manière il est garanti que le champ **final** sera toujours initialisé avant son utilisation.

Arguments final

Java permet de définir les arguments **final** en les déclarant comme tels dans la liste des arguments. Cela signifie qu'à l'intérieur de la méthode on ne peut pas changer ce vers quoi pointe l'argument:

```
// ! c06:FinalArguments.java
// Utilisation de « final » dans les arguments d'une méthode.

class Gizmo {
  public void spin() {}
}

public class FinalArguments {
  void with(final Gizmo g) {
    // ! g = new Gizmo(); // Illégal -- g est final
  }
  void without(Gizmo g) {
    g = new Gizmo(); // OK -- g n'est pas final
    g.spin();
```

```
  }
  // void f(final int i) { i++; } // Ne peut pas changer
  // On peut seulement lire depuis une primitive final:
  int g(final int i) { return i + 1; }
  public static void main(String[] args) {
    FinalArguments bf = new FinalArguments();
    bf.without(null);
    bf.with(null);
  }
} ///:~
```

A noter qu'on peut encore affecter une référence **null** à un argument qui est final sans que le compilateur ne l'empêche, comme on pourrait le faire pour un argument non-**final**.

Les méthodes **f( )** et **g( )** montre ce qui arrive quand les arguments primitifs sont **final**: on peut lire l'argument, mais on ne peut pas le changer.

Méthodes final

Les méthodes **final** ont deux raisons d'être. La première est de mettre un « verrou » sur la méthode pour empêcher toute sous-classe de la redéfinir. Ceci est fait pour des raisons de conception quand on veut être sûr que le comportement d'une méthode est préservé durant l'héritage et ne peut pas être redéfini.

La deuxième raison est l'efficacité. Si on met une méthode **final**, on permet au compilateur de convertir tout appel à cette méthode en un appel *incorporé*. Quand le compilateur voit un appel à une méthode **final**, il peut à sa discrétion éviter l'approche normale d'insérer du code pour exécuter l'appel de la méthode (mettre les arguments sur la pile, sauter au code de la méthode et l' exécuter, revenir au code courant et nettoyer les arguments de la pile, s'occuper de la valeur de retour) et à la place remplacer l'appel de méthode avec une copie du code de cette méthode dans le corps de la méthode courante. Ceci élimine le surcoût de l'appel de méthode. Bien entendu, si une méthode est importante, votre code commencera alors à grossir et vous ne verrez plus le gain de performance dû au code « incorporé », parce que toute amélioration sera cachée par le temps passé à l'intérieur de la méthode. Ceci implique que le compilateur Java est capable de détecter ces situations et de choisir sagement si oui ou non il faut « incorporer » une méthode **final**. Quoiqu'il en soit, il est mieux de ne pas faire confiance à ce que peut faire le compilateur et de mettre une méthode **final** seulement si elle est plutôt petite ou si on veut explicitement empêcher la surcharge.

final et private

Toutes les méthodes **private** sont implicitement **final**. Parce qu'on ne peut pas accéder à une méthode **private**, on ne peut pas la surcharger (même si le compilateur ne donne pas de messages d'erreur si on essaye de la redéfinir, on ne redéfinit pas la méthode, on a simplement créé une nouvelle méthode). On peut ajouter le mot-clé **final** à une méthode **private**, mais ça n'apporte rien de plus.

Ce problème peut rendre les choses un peu confuses, parce que si on essaye de surcharger une méthode **private** qui est implicitement **final** ça semble fonctionner:

```
// ! c06:FinalOverridingIllusion.java
```

```
// C'est seulement une impression qu'on peut
// surcharger une méthode private ou private final.

class WithFinals {
  // Identique à « private » tout seul:
  private final void f() {
    System.out.println("WithFinals.f()");
  }
  // Également automatiquement « final »:
  private void g() {
    System.out.println("WithFinals.g()");
  }
}

class OverridingPrivate extends WithFinals {
  private final void f() {
    System.out.println("OverridingPrivate.f()");
  }
  private void g() {
    System.out.println("OverridingPrivate.g()");
  }
}

class OverridingPrivate2
  extends OverridingPrivate {
  public final void f() {
    System.out.println("OverridingPrivate2.f()");
  }
  public void g() {
    System.out.println("OverridingPrivate2.g()");
  }
}

public class FinalOverridingIllusion {
  public static void main(String[] args) {
    OverridingPrivate2 op2 =
      new OverridingPrivate2();
    op2.f();
    op2.g();
    // On peut faire un transtypage ascendant:
    OverridingPrivate op = op2;
    // Mais on ne peut pas appeler les méthodes:
    // ! op.f();
    // ! op.g();
    // Idem ici:
    WithFinals wf = op2;
    // ! wf.f();
```

```
    // ! wf.g();
  }
} ///:~
```

« Surcharger » peut seulement arriver si quelque chose fait partie de l'interface de la classe de base. On doit être capable de faire un transtypage ascendant vers la classe de base et d'appeler la même méthode. Ce point deviendra clair dans le prochain chapitre. Si une méthode est **private**, elle ne fait pas partie de l'interface de la classe de base. C'est simplement du code qui est caché à l'intérieur de la classe, et il arrive simplement qu'elle a ce nom, mais si on définit une méthode **public**, **protected** ou « amies » dans la classe dérivée, il n'y a aucune connexion avec la méthode de même nom dans la classe de base. Étant donné qu'une méthode **private** est inatteignable et effectivement invisible, elle ne sert à rien d'autre qu'à l'organisation du code dans la classe dans laquelle elle est définie.

Classes final

Quand on dit qu'une classe entière est **final** (en faisant précéder sa définition par le mot-clé **final**) on stipule qu'on ne veut pas hériter de cette classe ou permettre à qui que ce soit de le faire. En d'autres mots, soit la conception de cette classe est telle qu'on n'aura jamais besoin de la modifier, soit pour des raisons de sûreté ou de sécurité on ne veut pas qu'elle soit sous-classée. Ou alors, on peut avoir affaire à un problème d'efficacité, et on veut s'assurer que toute activité impliquant des objets de cette classe sera aussi efficace que possible.

Notons que les données membres peuvent ou non être **final**, comme on le choisit. Les mêmes règles s'appliquent à **final** pour les données membres sans tenir compte du fait que la classe est ou non **final**. Définir une classe comme **final** empêche simplement l'héritage - rien de plus. Quoiqu'il en soit, parce que cela empêche l'héritage, toutes les méthodes d'une classe **final** sont implicitement **final**, étant donné qu'il n'y a aucun moyen de les surcharger. Donc le compilateur à les mêmes options d'efficacité que si on définissait explicitement une méthode **final**.

On peut ajouter le modificateur **final** à une méthode dans une classe **final**, mais ça ne rajoute aucune signification.

Attention finale

Il peut paraître raisonnable de déclarer une méthode **final** alors qu'on conçoit une classe. On peut décider que l'efficacité est très importante quand on utilise la classe et que personne ne pourrait vouloir surcharger les méthodes de toute manière. Cela est parfois vrai.

Mais il faut être prudent avec ces hypothèses. En général, il est difficile d' anticiper comment une classe va être réutilisée, surtout une classe générique. Si on définit une méthode comme **final**, on devrait empêcher la possibilité de réutiliser cette classe par héritage dans les projets d'autres programmeurs simplement parce qu'on ne pourrait pas l'imaginer être utilisée de cette manière.

La bibliothèque standard de Java en est un bon exemple. En particulier, la classe **Vector** en Java 1.0/1.1 était communément utilisée et pourrait avoir encore été plus utile si, au nom de l'efficacité, toutes les méthodes n'avaient pas été **final**. Ça parait facilement concevable que l'on puisse vouloir hériter et surcharger une telle classe fondamentale, mais les concepteurs d'une manière ou d'une autre ont décidé que ce n'était pas approprié. C'est ironique pour deux raisons. Premièrement, **Stack** hérite de **Vector**, ce qui dit qu'une **Stack** est un **Vector**, ce qui n'est pas vraiment vrai d'un

point de vue logique. Deuxièmement, beaucoup des méthodes importantes de **Vector**, telles que **addElement( )** et **elementAt( )** sont **synchronized**. Comme nous verrons au chapitre 14, ceci implique un surcoût significatif au niveau des performances qui rend caduque tout gain fourni par **final**. Ceci donne de la crédibilité à la théorie que les programmeurs sont constamment mauvais pour deviner où les optimisations devraient être faites.Il est vraiment dommage qu'une conception aussi maladroite ait fait son chemin dans la bibliothèque standard que nous utilisons tous. Heureusement, la bibliothèque de collections Java 2 remplace **Vector** avec **ArrayList**, laquelle se comporte bien mieux. Malheureusement , il y a encore beaucoup de code nouveau écrit qui utilise encore l'ancienne bibliothèques de collections.

Il est intéressant de noter également que **Hashtable**, une autre classe importante de la bibliothèque standard, **n'a pas** une seule méthode **final**. Comme mentionné à plusieurs endroits dans ce livre, il est plutôt évident que certaines classes ont été conçues par des personnes totalement différentes. Vous verrez que les noms de méthodes dans **Hashtable** sont beaucoup plus court comparés à ceux de **Vector**, une autre preuve. C'est précisément ce genre de choses qui ne devrait pas être évident aux utilisateurs d'une bibliothèque de classes. Quand plusieurs choses sont inconsistantes, cela fait simplement plus de travail pour l'utilisateur. Encore un autre grief à la valeur de la conception et de la qualité du code. À noter que la bibliothèque de collection de Java 2 remplaces **Hashtable** avec **HashMap**.

Initialisation et chargement de classes

Dans des langages plus traditionnels, les programmes sont chargés tout d'un coup au moment du démarrage. Ceci est suivi par l'initialisation et ensuite le programme commence. Le processus d'initialisation dans ces langages doit être contrôlé avec beaucoup d'attention afin que l'ordre d'initialisation des **static**s ne pose pas de problème. C++, par exemple, a des problèmes si une **static** attend qu'une autre **static** soit valide avant que la seconde ne soit initialisée.

Java n'a pas ce problème parce qu'il a une autre approche du chargement. Parce que tout en Java est un objet, beaucoup d'actions deviennent plus facile, et ceci en est un exemple. Comme vous l'apprendrez plus complètement dans le prochain chapitre, le code compilé de chaque classe existe dans son propre fichier séparé. Ce fichier n'est pas chargé tant que ce n'est pas nécessaire. En général, on peut dire que « le code d'une classe est chargé au moment de la première utilisation ». C'est souvent au moment où le premier objet de cette classe est construit, mais le chargement se produit également lorsqu'on accède à un champ **static** ou une méthode **static**.

Le point de première utilisation est également là où l'initialisation des **static**s a lieu. Tous les objets **static** et le bloc de code **static** sera initialisé dans l'ordre textuel (l'ordre dans lequel ils sont définis dans la définition de la classe) au moment du chargement. Les **static**s, bien sûr, ne sont initialisés qu'une seule fois.

Initialisation avec héritage

Il est utile de regarder l'ensemble du processus d'initialisation, incluant l'héritage pour obtenir une compréhension globale de ce qui se passe. Considérons le code suivant:

```
// ! c06:Beetle.java
// Le processus complet d'initialisation.

class Insect {
```

```
  int i = 9;
  int j;
  Insect() {
    prt("i = " + i + ", j = " + j);
    j = 39;
  }
  static int x1 =
    prt("static Insect.x1 initialisé");
  static int prt(String s) {
    System.out.println(s);
    return 47;
  }
}

public class Beetle extends Insect {
  int k = prt("Beetle.k initialisé");
  Beetle() {
    prt("k = " + k);
    prt("j = " + j);
  }
  static int x2 =    prt("static Beetle.x2 initialisé");
  public static void main(String[] args) {
    prt("Constructeur Beetle");
    Beetle b = new Beetle();
  }
} ///:~
```

La sortie de ce programme est:

```
static Insect.x1 initialisé
static Beetle.x2 initialisé
Constructeur Beetle
i = 9, j = 0
Beetle.k initialisé
k = 47
j = 39
```

La première chose qui se passe quand on exécute **Beetle** en Java est qu'on essaye d'accéder à **Beetle.main( )** (une méthode **static**), donc le chargeur cherche et trouve le code compilé pour la classe **Beetle** (en général dans le fichier appelé **Beetle.class**). Dans le processus de son chargement, le chargeur remarque qu'elle a une classe de base (c'est ce que le mot-clé **extends** veut dire), laquelle est alors chargée. Ceci se produit qu'on construise ou non un objet de la classe de base. Essayez de commenter la création de l'objet pour vous le prouver.

Si la classe de base a une classe de base, cette seconde classe de base sera à son tour chargée, etc. Ensuite, l'initialisation **static** dans la classe de base racine (dans ce cas, **Insect**) est effectuée, ensuite la prochaine classe dérivée, etc. C'est important parce que l'initialisation static de la classe dérivée pourrait dépendre de l'initialisation correcte d'un membre de la classe de base.

À ce point, les classes nécessaires ont été chargées, donc l'objet peut être créé. Premièrement, toutes les primitives dans l'objet sont initialisées à leurs valeurs par défaut et les références objets sont initialisées à **null** - ceci se produit en une seule passe en mettant la mémoire dans l'objet au zéro binaire. Ensuite le constructeur de la classe de base est appelé. Dans ce cas, l'appel est automatique, mais on peut également spécifier le constructeur de la classe de base (comme la première opération dans le constructeur **Beetle( )**) en utilisant **super**. La constructeur de la classe de base suit le même processus dans le même ordre que le constructeur de la classe dérivée. Lorsque le constructeur de la classe de base a terminé, les variables d'instance sont initialisées dans l'ordre textuel. Finalement le reste du corps du constructeur est exécuté.

Résumé

L'héritage et la composition permettent tous les deux de créer de nouveaux types depuis des types existants. Typiquement, quoiqu'il en soit, on utilise la composition pour réutiliser des types existants comme partie de l'implémentation sous-jacente du nouveau type, et l'héritage quand on veut réutiliser l'interface. Étant donné que la classe dérivée possède l'interface de la classe de base, on peut faire un **transtypage ascendant** vers la classe de base, lequel est critique pour le polymorphisme, comme vous le verrez dans le prochain chapitre.

En dépit de l'importance particulièrement forte de l'héritage dans la programmation orienté objet, quand on commence une conception on devrait généralement préférer la composition durant la première passe et utiliser l'héritage seulement quand c'est clairement nécessaire. La composition tend à être plus flexible. De plus, par le biais de l'héritage, vous pouvez changer le type exact de vos objets, et donc, le comportement, de ces objets membres à l'exécution. Par conséquent, on peut changer le comportement d'objets composés à l'exécution.

Bien que la réutilisation du code à travers la composition et l'héritage soit utile pour un développement rapide, on voudra généralement concevoir à nouveau la hiérarchie de classes avant de permettre aux autres programmeurs d'en devenir dépendant. Votre but est une hiérarchie dans laquelle chaque classe a un usage spécifique et n'est ni trop grosse (englobant tellement de fonctionnalités qu'elle en est difficile à manier pour être réutilisée) ni trop ennuyeusement petite (on ne peut pas l'utiliser par elle-même ou sans ajouter de nouvelles fonctionnalités).

Exercices

Les solutions aux exercices sélectionnés peuvent être trouvées dans le document électronique *The Thinking in Java Annotated Solution Guide*, disponible pour un faible coût depuis *www.BruceEckel.com*.

1. Créer deux classes, **A** et **B**, avec des constructeurs par défaut (liste d'arguments vide) qui s'annoncent eux-même. Faire hériter une nouvelle classe **C** de **A**, et créer une classe membre **B** à l'intérieur de **C**. Ne pas créer un constructeur pour **C**. Créer un objet d'une classe **C** et observer les résultats.

2. Modifier l'exercice 1 afin que **A** et **B** aient des constructeurs avec arguments au lieu de constructeurs par défaut. Écrire un constructeur pour **C** et effectuer toutes les initialisations à l'intérieur du constructeur de **C**.

3. Créer une simple classe. À l'intérieur d'une seconde classe, définir un champ pour un objet de la première classe. Utiliser l'initialisation paresseuse pour instancier cet objet.

4. Hériter une nouvelle classe de la classe **Detergent**. Redéfinir **scrub( )** et ajouter une nouvelle méthode appelée **sterilize( )**.

5. Prendre le fichier **Cartoon.java** et enlever le commentaire autour du constructeur de la classe **Cartoon**. Expliquer ce qui arrive.

6. Prendre le fichier **Chess.java** et enlever le commentaire autour du constructeur de la classe **Chess**. Expliquer ce qui se passe.

7. Prouver que des constructeurs par défaut sont créés pour vous par le compilateur.

8. Prouver que les constructeurs de la classe de base sont (a) toujours appelés et (b) appelés avant les constructeurs des classes dérivées.

9. Créer une classe de base avec seulement un constructeur qui ne soit pas un constructeur par défaut, et une classe dérivée avec à la fois une constructeur par défaut et un deuxième constructeur. Dans les constructeurs de la classe dérivée, appeler le constructeur de la classe de base.

10. Créer une classe appelée **Root** qui contient une instance de chaque classe (que vous aurez également créé) appelées **Component1**, **Component2**, et **Component3**. Dériver une classe **Stem** de **Root** qui contienne également une instance de chaque « component ». Toutes les classes devraient avoir un constructeur par défaut qui affiche un message au sujet de cette classe.

11. Modifier l'exercice 10 de manière à ce que chaque classe ait des constructeurs qui ne soient pas des constructeurs par défaut.

12. Ajouter une hiérarchie propre de méthodes **cleanup( )** à toutes les classes dans l'exercice 11.

13. Créer une classe avec une méthode surchargée trois fois. Hériter une nouvelle classe, ajouter une nouvelle surcharge de la méthode et montrer que les quatre méthodes sont disponibles dans la classe dérivée.

14. Dans **Car.java** ajouter une méthode **service( )** à **Engine** et appeler cette méthode dans **main( )**.

15. Créer une classe à l'intérieur d'un package. Cette classe doit contenir une méthode **protected**. À l'extérieur du package, essayer d'appeler la méthode **protected** et expliquer les résultats. Maintenant hériter de cette classe et appeler la méthode **protected** depuis l'intérieur d'une méthode de la classe dérivée.

16. Créer une classe appelée **Amphibian**. De celle-ci, hériter une classe appelée **Frog**. Mettre les méthodes appropriées dans la classe de base. Dans **main( )**, créer une **Frog** et faire un transtypage ascendant **Amphibian**, et démontrer que toutes les méthodes fonctionnent encore.

17. Modifier l'exercice 16 de manière que **Frog** redéfinisse les définitions de méthodes de la classe de base (fournir de nouvelles définitions utilisant les mêmes signatures des méthodes). Noter ce qui se passe dans **main( )**.

18. Créer une classe avec un champ **static final** et un champ **final** et démontrer la différence entre les deux.

19. Créer une classe avec une référence **final** sans initialisation vers un objet. Exécuter l'initialisation de cette **final** sans initialisation à l'intérieur d'une méthode (pas un constructeur)

juste avant de l'utiliser. Démontrer la garantie que le **final** doit être initialisé avant d'être utilisé, et ne peut pas être changé une fois initialisé.

20. Créer une classe avec une méthode **final**. Hériter de cette classe et tenter de redéfinir cette méthode.

21. Créer une classe **final** et tenter d'en hériter.

22. Prouver que le chargement d'une classe n'a lieu qu'une fois. Prouver que le chargement peut être causé soit par la création de la première instance de cette classe, soit par l'accès à un membre **static**.

23. Dans **Beetle.java**, hériter un type spécifique de coccinelle de la classe **Beetle**, suivant le même format des classes existantes. Regarder et expliquer le flux de sortie du programme.

Chapitre 7 - Polymorphisme

Le polymorphisme est la troisième caractéristique essentielle d'un langage de programmation orienté objet, après l'abstraction et l'héritage.

Le polymorphisme fournit une autre dimension séparant la partie interface de l'implémentation qui permet de découpler le *quoi* du *comment*. Le polymorphisme améliore l'organisation du code et sa lisibilité de même qu'il permet la création de programmes *extensible* qui peuvent évoluer non seulement pendant la création initiale du projet mais également quand des fonctions nouvelles sont désirées.

L'encapsulation crée de nouveaux types de données en combinant les caractéristiques et les comportements. Cacher la mise en œuvre permet de séparer l'interface de l'implémentation en mettant les détails privés [**private**]. Cette sorte d'organisation mécanique est bien comprise par ceux viennent de la programmation procédurale. Mais le polymorphisme s'occupe de découpler au niveau des *types*. Dans le chapitre précédant, nous avons vu comment l'héritage permet le traitement d'un objet comme son propre type *ou* son type de base. Cette capacité est critique car elle permet à beaucoup de types (dérivé d'un même type de base) d'être traités comme s'ils n'étaient qu'un type, et permet a un seul morceau de code de traiter sans distinction tous ces types différents. L'appel de méthode polymorphe permet à un type d'exprimer sa distinction par rapport à un autre, un type semblable, tant qu'ils dérivent tous les deux d'un même type de base. Cette distinction est exprimée à travers des différences de comportement des méthodes que vous pouvez appeler par la classe de base.

Dans ce chapitre, vous allez comprendre le polymorphisme [également appelé en anglais *dynamic binding* ou *late binding* ou encore *run-time binding*] en commençant par les notions de base, avec des exemples simples qui évacuent tout ce qui ne concerne pas le comportement polymorphe du programme.

Upcasting

Dans le chapitre 6 nous avons vu qu'un objet peut être manipulé avec son propre type ou bien comme un objet de son type de base. Prendre la référence d'un objet et l'utiliser comme une référence sur le type de base est appelé *upcasting* (*transtypage* en français) , en raison du mode de représentation des arbres d'héritages avec la classe de base en haut.

On avait vu le problème reprit ci-dessous apparaître:

```
//: c07:music:Music.java
// Héritage & upcasting.
class Note {
  private int value;
  private Note(int val) { value = val; }
  public static final Note
    MIDDLE_C = new Note(0),
    C_SHARP  = new Note(1),
    B_FLAT   = new Note(2);
} // Etc.
class Instrument {
```

```
  public void play(Note n) {
    System.out.println("Instrument.play()");
  }
}

// Les objets Wind sont des instruments
// car ils ont la même interface:
class Wind extends Instrument {
  // Redéfinition de la méthode de l'interface:
  public void play(Note n) {
    System.out.println("Wind.play()");
  }
}

public class Music {
  public static void tune(Instrument i) {
    // ...
    i.play(Note.MIDDLE_C);
  }
  public static void main(String[] args) {
    Wind flute = new Wind();
    tune(flute); // Upcasting
  }
} ///:~
```

La méthode **Music.tune()** accepte une référence sur un **Instrument**, mais également sur tout ce qui dérive de **Instrument**. Dans le **main()**, ceci se matérialise par une référence sur un objet **Wind** qui est passée à **tune()**, sans qu'un changement de type (un cast) soit nécessaire. Ceci est correct; l'interface dans **Instrument** doit exister dans **Wind**, car **Wind** hérite de la classe **Instrument**. Utiliser l'upcast de **Wind** vers **Instrument** peut « rétrécir » cette interface, mais au pire elle est réduite à l'interface complète d'**Instrument**.

Pourquoi utiliser l'upcast?

Ce programme pourrait vous sembler étrange. Pourquoi donc *oublier* intentionnellement le type d'un objet? C'est ce qui arrive quand on fait un upcast, et il semble beaucoup plus naturel que **tune()** prenne tout simplement une référence sur **Wind** comme argument. Ceci introduit un point essentiel: en faisant ça, il faudrait écrire une nouvelle méthode **tune()** pour chaque type de **Instrument** du système. Supposons que l'on suive ce raisonnement et que l'on ajoute les instruments à cordes [**Stringed**] et les cuivres [**Brass**]:

```
//: c07:music2:Music2.java
// Surcharger plutôt que d'utiliser l'upcast.

class Note {
  private int value;
  private Note(int val) { value = val; }
```

```
  public static final Note
    MIDDLE_C = new Note(0),
    C_SHARP = new Note(1),
    B_FLAT = new Note(2);
} // Etc.

class Instrument {
  public void play(Note n) {
    System.out.println("Instrument.play()");
  }
}

class Wind extends Instrument {
  public void play(Note n) {
    System.out.println("Wind.play()");
  }
}

class Stringed extends Instrument {
  public void play(Note n) {
    System.out.println("Stringed.play()");
  }
}

class Brass extends Instrument {
  public void play(Note n) {
    System.out.println("Brass.play()");
  }
}

public class Music2 {
  public static void tune(Wind i) {
    i.play(Note.MIDDLE_C);
  }
  public static void tune(Stringed i) {
    i.play(Note.MIDDLE_C);
  }
  public static void tune(Brass i) {
    i.play(Note.MIDDLE_C);
  }
  public static void main(String[] args) {
    Wind flute = new Wind();
    Stringed violin = new Stringed();
    Brass frenchHorn = new Brass();
    tune(flute); // Pas d' upcast
    tune(violin);
    tune(frenchHorn);
```

```
  }
} ///:~
```

Ceci fonctionne, mais avec un inconvénient majeur: il vous faut écrire des classes spécifique à chaque ajout d'une classe **Instrument**. Ceci implique davantage de programmation dans un premier temps, mais également beaucoup de travail à venir si l'on désire ajouter une nouvelle méthode comme **tune()** ou un nouveau type d' **Instrument**. Sans parler du compilateur qui est incapable de signaler l'oubli de surcharge de l'une de vos méthodes qui fait que toute cette construction utilisant les types devient assez compliquée.

Ne serait-il pas plus commode d'écrire une seule méthode qui prenne la name="Index673">classe de base en argument plutôt que toutes les classes dérivées spécifiques? Ou encore, ne serait-il pas agréable d'oublier qu'il y a des classes dérivées et d'écrire votre code en ne s'adressant qu'à la classe de base?

C'est exactement ce que le polymorphisme vous permet de faire. Souvent, ceux qui viennent de la programmation procédurale sont déroutés par le mode de fonctionnement du polymorphisme.

The twist

L'ennui avec **Music.java** peut être visualisé en exécutant le programme. L'output est **Wind.play()**. C'est bien sûr le résultat attendu, mais il n'est pas évident de comprendre le fonctionnement.. Examinons la méthode **tune()**:

```
public static void tune(Instrument i) {
  // ...
  i.play(Note.MIDDLE_C);
}
```

Elle prend une référence sur un **Instrument** en argument. Comment le compilateur peut-il donc deviner que cette référence sur un **Instrument** pointe dans le cas présent sur un objet **Wind** et non pas un objet **Brass** ou un objet **Stringed**? Hé bien il ne peut pas. Mieux vaut examiner le mécanisme d'*association [binding]* pour bien comprendre la question soulevée.name="_Toc312374042">

Liaison de l'appel de méthode

Raccorder un appel de méthode avec le corps de cette méthode est appelé *association*. Quand cette association est réalisée avant l'exécution du programme (par le compilateur et l'éditeur de lien, s'il y en a un), c'est de l'*association prédéfinie*. Vous ne devriez pas avoir déjà entendu ce terme auparavant car avec les langages procéduraux, c'est imposé . Les compilateurs C n'ont qu'une sorte d'appel de méthode, l'association prédéfinie.

Ce qui déroute dans le programme ci-dessus tourne autour de l'association prédéfinie car le compilateur ne peut pas connaître la bonne méthode a appeler lorsqu'il ne dispose que d'une référence sur **Instrument**.

La solution s'appelle l' *association tardive*, ce qui signifie que l'association est effectuée à l'exécution en se basant sur le type de l'objet. L'association tardive est également appelée *associa-*

tion dynamique [dynamic binding ou *run-time binding]*. Quand un langage implémente l'association dynamique, un mécanisme doit être prévu pour déterminer le type de l'objet lors de l'exécution et pour appeler ainsi la méthode appropriée. Ce qui veut dire que le compilateur ne connaît toujours pas le type de l'objet, mais le mécanisme d'appel de méthode trouve et effectue l'appel vers le bon corps de méthode. Les mécanismes d'association tardive varient selon les langages, mais vous pouvez deviner que des informations relatives au type doivent être implantées dans les objets.

Toutes les associations de méthode en Java utilisent l'association tardive à moins que l'on ait déclaré une méthode **final**. Cela signifie que d'habitude vous n'avez pas à vous préoccuper du déclenchement de l'association tardive, cela arrive automatiquement.

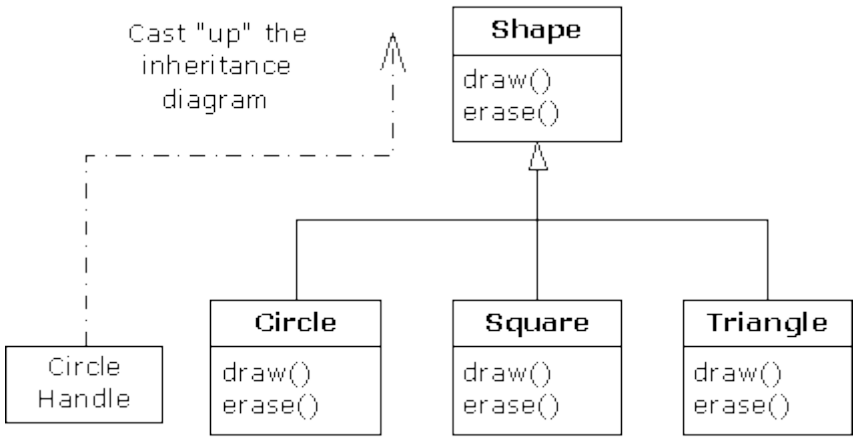
Pourquoi déclarer une méthode avec **final**? On a vu dans le chapitre précédant que cela empêche quelqu'un de redéfinir cette méthode. Peut-être plus important, cela « coupe » effectivement l'association dynamique, ou plutôt cela indique au compilateur que l'association dynamique n'est pas nécessaire. Le compilateur génère du code légèrement plus efficace pour les appels de méthodes spécifiés **final**. Cependant, dans la plupart des cas cela ne changera pas la performance globale de votre programme; mieux vaut utiliser **final** à la suite d'une décision de conception, et non pas comme tentative d'amélioration des performances.

Produire le bon comportement

Quand vous savez qu'en Java toute association de méthode se fait de manière polymorphe par l'association tardive, vous pouvez écrire votre code en vous adressant à la classe de base en sachant que tous les cas des classes dérivées fonctionneront correctement avec le même code. Dit autrement, vous « envoyez un message à un objet et laissez l'objet trouver le comportement adéquat. »

L'exemple classique utilisée en POO est celui de la forme [shape]. Cet exemple est généralement utilisé car il est facile à visualiser, mais peut malheureusement sous-entendre que la POO est cantonnée au domaine graphique, ce qui n'est bien sûr pas le cas.

Dans cet exemple il y a une classe de base appelée **Shape** et plusieurs types dérivés: **Circle**, **Square**, **Triangle**, etc. Cet exemple marche très bien car il est naturel de dire qu'un cercle est « une sorte de forme. » Le diagramme d'héritage montre les relations :



l'upcast pourrait se produire dans une instruction aussi simple que :

```
Shape s = new Circle();
```

On créé un objet **Circle** et la nouvelle référence est assignée à un **Shape**, ce qui semblerait être une erreur (assigner un type à un autre), mais qui est valide ici car un Cercle [**Circle**] *est* par héritage une sorte de forme [**Shape**]. Le compilateur vérifie la légalité de cette instruction et n'émet pas de message d'erreur.

Supposons que vous appeliez une des méthode de la classe de base qui a été redéfinie dans les classes dérivées :

```
s.draw();
```

De nouveau, vous pourriez vous attendre à ce que la méthode **draw**() de **Shape** soit appelée parce que c'est après tout une référence sur **Shape.** Alors comment le compilateur peut-il faire une autre liaison? Et malgré tout le bon **Circle.draw()** est appelé grâce à la liaison tardive (polymorphisme).

L'exemple suivant le montre de manière légèrement différente :

```
//: c07:Shapes.java
// Polymorphisme en Java.

class Shape {
  void draw() {}
  void erase() {}
}

class Circle extends Shape {
  void draw() {
    System.out.println("Circle.draw()");
  }
  void erase() {
    System.out.println("Circle.erase()");
  }
}

class Square extends Shape {
  void draw() {
    System.out.println("Square.draw()");
  }
  void erase() {
    System.out.println("Square.erase()");
  }
}

class Triangle extends Shape {
  void draw() {
    System.out.println("Triangle.draw()");
  }
  void erase() {
    System.out.println("Triangle.erase()");
  }
```

```
}

public class Shapes {
  public static Shape randShape() {
    switch((int)(Math.random() * 3)) {
      default:
      case 0: return new Circle();
      case 1: return new Square();
      case 2: return new Triangle();
    }
  }
  public static void main(String[] args) {
    Shape[] s = new Shape[9];
    // Remplissage du tableau avec des formes [shapes]:
    forint i = 0; i < s.length; i++)
      s[i] = randShape();
    // Appel polymorphe des méthodes:
    forint i = 0; i < s.length; i++)
      s[i].draw();
  }
} ///:~
```

La classe de base **Shape** établit l'interface commune pour tout ce qui hérite de **Shape** — C'est à dire, toutes les formes (shapes en anglais) peuvent être dessinées [draw] et effacées [erase]. Les classes dérivées redéfinissent ces méthodes pour fournir un comportement unique pour chaque type de forme spécifique.

La classe principale **Shapes** contient une méthode statique **randShape ()** qui rend une référence sur un objet sélectionné de manière aléatoire à chaque appel. Remarquez que la généralisation se produit sur chaque instruction **return**, qui prend une référence sur un cercle [circle], un carré [square], ou un triangle et la retourne comme le type de retour de la méthode, en l'occurrence **Shape**. Ainsi à chaque appel de cette méthode vous ne pouvez pas voir quel type spécifique vous obtenez, puisque vous récupérez toujours une simple référence sur **Shape**.

Le **main()** a un tableau de références sur **Shape** remplies par des appels a **randShape()**. Tout ce que l'on sait dans la première boucle c'est que l'on a des objets formes [**Shape**s], mais on ne sait rien de plus (pareil pour le compilateur). Cependant, quand vous parcourez ce tableau en appelant **draw()** pour chaque référence dans la seconde boucle, le bon comportement correspondant au type spécifique se produit comme par magie, comme vous pouvez le constater sur l'output de l'exemple :

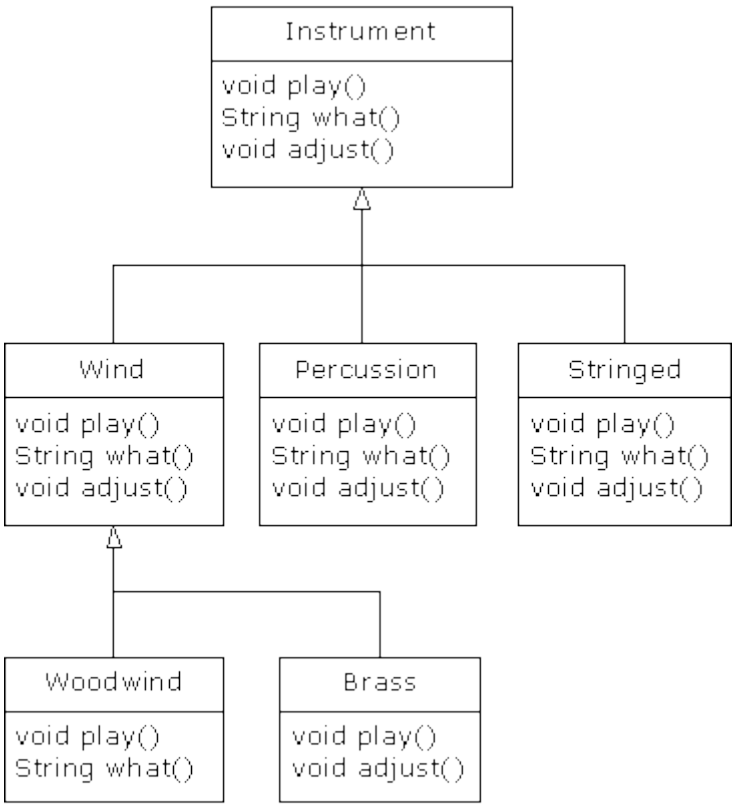
```
Circle.draw()
Triangle.draw()
Circle.draw()
Circle.draw()
Circle.draw()
Square.draw()
Triangle.draw()
Square.draw()
Square.draw()
```

Comme toutes les formes sont choisies aléatoirement à chaque fois, vous obtiendrez bien sûr des résultats différents. L'intérêt de choisir les formes aléatoirement est d'illustrer le fait que le compilateur ne peut avoir aucune connaissance spéciale lui permettant de générer les appels corrects au moment de la compilation. Tous les appels à **draw()** sont réalisés par liaison dynamique.

Extensibilité

Revenons maintenant à l'exemple sur l'instrument de musique. En raison du polymorphisme, vous pouvez ajouter autant de nouveaux types que vous voulez dans le système sans changer la méthode **tune()**. Dans un programme orienté objet bien conçu, la plupart ou même toutes vos méthodes suivront le modèle de **tune()** et communiqueront seulement avec l'interface de la classe de base. Un tel programme est *extensible* parce que vous pouvez ajouter de nouvelles fonctionnalités en héritant de nouveaux types de données de la classe de base commune. Les méthodes qui utilisent l'interface de la classe de base n'auront pas besoin d'être retouchées pour intégrer de nouvelles classes.

Regardez ce qui se produit dans l'exemple de l'instrument si vous ajoutez des méthodes dans la classe de base et un certain nombre de nouvelles classes. Voici le schéma :



Toutes ces nouvelles classes fonctionnent correctement avec la vieille méthode **tune()**, sans modification. Même si **tune()** est dans un fichier séparé et que de nouvelles méthodes sont ajoutées à l'interface de **Instrument**, **tune()** fonctionne correctement sans recompilation. Voici l'implémentation du diagramme présenté ci-dessus :

```
//: c07:music3:Music3.java
// Un programme extensible.
import java.util.*;

class Instrument {
  public void play() {
    System.out.println("Instrument.play()");
  }
  public String what() {
    return "Instrument";
  }
  public void adjust() {}
}

class Wind extends Instrument {
  public void play() {
    System.out.println("Wind.play()");
  }
  public String what() { return "Wind"; }
  public void adjust() {}
}

class Percussion extends Instrument {
  public void play() {
    System.out.println("Percussion.play()");
  }
  public String what() { return "Percussion"; }
  public void adjust() {}
}

class Stringed extends Instrument {
  public void play() {
    System.out.println("Stringed.play()");
  }
  public String what() { return "Stringed"; }
  public void adjust() {}
}

class Brass extends Wind {
  public void play() {
    System.out.println("Brass.play()");
  }
  public void adjust() {
    System.out.println("Brass.adjust()");
  }
}
```

```
class Woodwind extends Wind {
  public void play() {
    System.out.println("Woodwind.play()");
  }
  public String what() { return "Woodwind"; }
}

public class Music3 {
// Indépendants des types, ainsi les nouveaux types
  // ajoutés au système marchent toujours bien:
  static void tune(Instrument i) {
    // ...
    i.play();
  }
  static void tuneAll(Instrument[] e) {
    forint i = 0; i < e.length; i++)
      tune(e[i]);
  }
  public static void main(String[] args) {
    Instrument[] orchestra = new Instrument[5];
    int i = 0;
    // Upcasting pendant l'ajout au tableau:
    orchestra[i++] = new Wind();
    orchestra[i++] = new Percussion();
    orchestra[i++] = new Stringed();
    orchestra[i++] = new Brass();
    orchestra[i++] = new Woodwind();
    tuneAll(orchestra);
  }
} ///:~
```

Dans le **main()**, quand on met quelque chose dans le tableau d' **Instrument,** on upcast automatiquement en **Instrument**.

Vous pouvez constater que la méthode **tune()** ignore fort heureusement tous les changements qui sont intervenus autour d'elle, et pourtant cela marche correctement. C'est exactement ce que le polymorphisme est censé fournir. Vos modifications ne peuvent abîmer les parties du programme qui ne devraient pas être affectées. Dit autrement, le polymorphisme est une des techniques majeures permettant au programmeur de « séparer les choses qui changent des choses qui restent les mêmes. »

Redéfinition et Surcharge

Regardons sous un angle différent le premier exemple de ce chapitre. Dans le programme suivant, l'interface de la méthode **play()** est changée dans le but de la redéfinir, ce qui signifie que vous n'avez pas *redéfinie* la méthode, mais plutôt *surchargée*. Le compilateur vous permet de surcharger des méthodes, il ne proteste donc pas. Mais le comportement n'est probablement pas celui

que vous vouliez. Voici l'exemple :

```
//: c07:WindError.java
// Changement accidentel de l'interface.
class NoteX {
  public static final int
    MIDDLE_C = 0, C_SHARP = 1, C_FLAT = 2;
}

class InstrumentX {
  public void play(int NoteX) {
    System.out.println("InstrumentX.play()");
  }
}

class WindX extends InstrumentX {
  // OUPS! L'interface de la méthode change:
  public void play(NoteX n) {
    System.out.println("WindX.play(NoteX n)");
  }
}

public class WindError {
  public static void tune(InstrumentX i) {
    // ...
    i.play(NoteX.MIDDLE_C);
  }
  public static void main(String[] args) {
    WindX flute = new WindX();
    tune(flute); // Ce n'est pas le comportement souhaité!
  }
} ///:~
```

Il y a un autre aspect déroutant ici. Dans **InstrumentX**, la méthode **play()** a pour argument un **int** identifié par **NoteX**. Bien que **NoteX** soit un nom de classe, il peut également être utilisé comme identificateur sans erreur. Mais dans **WindX**, **play()** prend une référence de **NoteX** qui a pour identificateur **n** (bien que vous puissiez même écrire **play(NoteX NoteX)** sans erreur). En fait, il s'avère que le programmeur a désiré redéfinir **play()** mais s'est trompé de type. Du coup le compilateur a supposé qu'une surcharge était souhaitée et non pas une redéfinition. Remarquez que si vous respectez la convention standard de nommage Java, l'identificateur d'argument serait **noteX** ('n' minuscule), ce qui le distinguerait du nom de la classe.

Dans **tune**, le message **play()** est envoyé à l'**InstrumentX i**, avec comme argument un de membres de **NoteX** (**MIDDLE_C**). Puisque **NoteX** contient des définitions d'**int**, ceci signifie que c'est la version avec **int** de la méthode **play(),** dorénavant surchargée, qui est appelée. Comme elle *n'a pas* été redéfinie, c'est donc la méthode de la classe de base qui est utilisée.

L'output est le suivant :

```
InstrumentX.play()
```

Ceci n'est pas un appel polymorphe de méthode. Dès que vous comprenez ce qui se passe, vous pouvez corriger le problème assez facilement, mais imaginez la difficulté pour trouver l'anomalie si elle est enterrée dans un gros programme.

Classes et méthodes abstraites

Dans tous ces exemples sur l'instrument de musique, les méthodes de la classe de base **Instrument** étaient toujours factices. Si jamais ces méthodes sont appelées, c'est que vous avez fait quelque chose de travers. C'est parce que le rôle de la classe **Instrument** est de créer une interface *commune* pour toutes les classes dérivées d'elle.

La seule raison d'avoir cette interface commune est qu'elle peut être exprimée différemment pour chaque sous-type différent. Elle établit une forme de base, ainsi vous pouvez dire ce qui est commun avec toutes les classes dérivées. Une autre manière d'exprimer cette factorisation du code est d'appeler **Instrument** une classe de base abstraite (ou simplement une classe abstraite). Vous créez une classe abstraite quand vous voulez manipuler un ensemble de classes à travers cette interface commune. Toutes les méthodes des classes dérivées qui correspondent à la signature de la déclaration de classe de base seront appelées employant le mécanisme de liaison dynamique. (Cependant, comme on l'a vu dans la dernière section, si le nom de la méthode est le même comme la classe de base mais les arguments sont différents, vous avez une surcharge, ce qui n'est pas probablement que vous voulez.)

Si vous avez une classe abstraite comme **Instrument**, les objets de cette classe n'ont pratiquement aucune signification. Le rôle d'**Instrument** est uniquement d'exprimer une interface et non pas une implémentation particulière, ainsi la création d'un objet **Instrument** n'a pas de sens et vous voudrez probablement dissuader l'utilisateur de le faire. Une implémentation possible est d'afficher un message d'erreur dans toutes les méthodes d'**Instrument**, mais cela retarde le diagnostic à l'exécution et exige un code fiable et exhaustif. Il est toujours préférable de traiter les problèmes au moment de la compilation.

Java fournit un mécanisme qui implémente cette fonctionnalité: c'est la méthode abstraite [37]. C'est une méthode qui est incomplète; elle a seulement une déclaration et aucun corps de méthode. Voici la syntaxe pour une déclaration de méthode abstraite [abstract] :

```
abstract void f();
```

Une classe contenant des méthodes abstraites est appelée une *classe abstraite*. Si une classe contient une ou plusieurs méthodes abstraites, la classe doit être qualifiée comme **abstract**. (Autrement, le compilateur signale une erreur.)

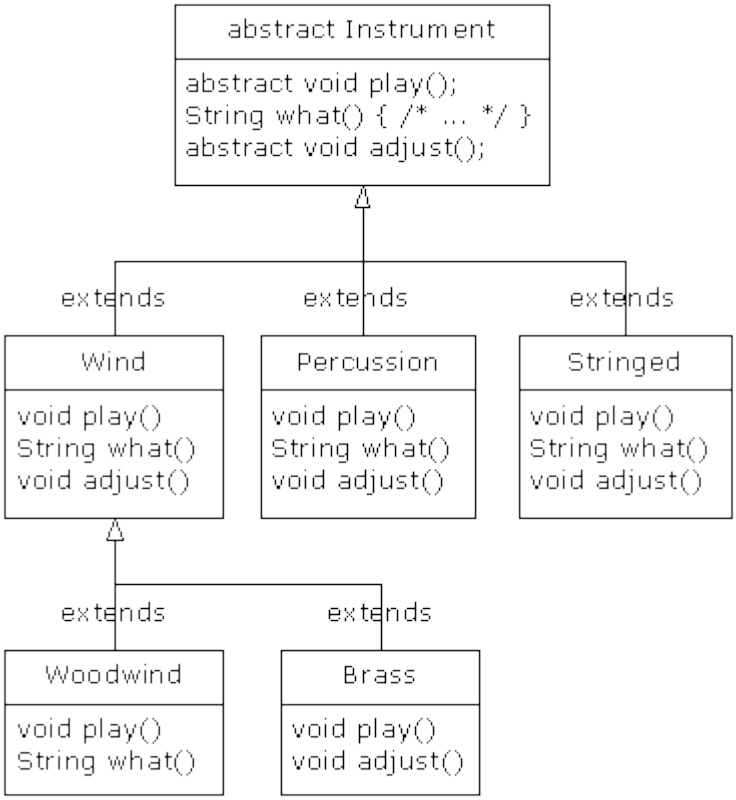
Si une classe abstraite est incomplète, comment doit réagir le compilateur si quelqu'un essaye de créer un objet de cette classe? Il ne peut pas créer sans risque un objet d'une classe abstraite, donc vous obtenez un message d'erreur du compilateur. C'est ainsi que le compilateur assure la pureté de la classe abstraite et ainsi vous n'avez plus a vous soucier d'un usage impropre de la classe.

Si vous héritez d'une classe abstraite et que vous voulez fabriquer des objets du nouveau type, vous devez fournir des définitions de méthode correspondant à toutes les méthodes abstraites de la classe de base. Si vous ne le faites pas (cela peut être votre choix), alors la classe dérivée est aussi abstraite et le compilateur vous forcera à qualifier cette classe avec le mot clé **abstract**.

Il est possible de créer une classe abstraite sans qu'elle contienne des méthodes abstraites. C'est utile quand vous avez une classe pour laquelle avoir des méthodes abstraites n'a pas de sens et

que vous voulez empêcher la création d'instance de cette classe.

La classe **Instrument** peut facilement être changée en une classe abstraite. Seules certaines des méthodes seront abstraites, puisque créer une classe abstraite ne vous oblige pas a avoir que des méthodes abstraites. Voici à quoi cela ressemble :



Voici l'exemple de l'orchestre modifié en utilisant des classes et des méthodes abstraites :

```
//: c07:music4:Music4.java
// Classes et méthodes abstraites.
import java.util.*;

abstract class Instrument {
  int i; // Alloué à chaque fois
  public abstract void play();
  public String what() {
    return "Instrument"size="4">;
  }
  public abstract void adjust();
}

class Wind extends Instrument {
  public void play() {
    System.out.println("Wind.play()");
  }
  public String what() { return "Wind"; }
```

```
  public void adjust() {}
}

class Percussion extends Instrument {
  public void play() {
    System.out.println("Percussion.play()");
  }
  public String what() { return "Percussion"; }
  public void adjust() {}
}

class Stringed extends Instrument {
  public void play() {
    System.out.println("Stringed.play()");
  }
  public String what() { return "Stringed"; }
  public void adjust() {}
}

class Brass extends Wind {
  public void play() {
    System.out.println("Brass.play()");
  }
  public void adjust() {
    System.out.println("Brass.adjust()");
  }
}

class Woodwind extends Wind {
  public void play() {
    System.out.println("Woodwind.play()");
  }
  public String what() { return "Woodwind"; }
}

public class Music4 {
  // Ne se préoccupe pas des types: des nouveaux
  // ajoutés au système marcheront très bien:
  static void tune(Instrument i) {
    // ...
    i.play();
  }
  static void tuneAll(Instrument[] e) {
    forint i = 0; i < e.length; i++)
      tune(e[i]);
  }
  public static void main(String[] args) {
```

```
    Instrument[] orchestra = new Instrument[5];
    int i = 0;
    // Upcast lors de l'ajout au tableau:
    orchestra[i++] = new Wind();
    orchestra[i++] = new Percussion();
    orchestra[i++] = new Stringed();
    orchestra[i++] = new Brass();
    orchestra[i++] = new Woodwind();
    tuneAll(orchestra);
  }
} ///:~
```

Vous pouvez voir qu'il n'y a vraiment aucun changement excepté dans la classe de base.

Il est utile de créer des classes et des méthodes abstraites parce qu'elles forment l'abstraction d'une classe explicite et indique autant à utilisateur qu'au compilateur comment elles doivent être utilisées.

Constructeurs et polymorphisme

Comme d'habitude, les constructeurs se comportent différemment des autres sortes de méthodes. C'est encore vrai pour le polymorphisme. Quoique les constructeurs ne soient pas polymorphes (bien que vous puissiez avoir un genre de "constructeur virtuel", comme vous le verrez dans le chapitre 12), il est important de comprendre comment les constructeurs se comportent dans des hiérarchies complexes combiné avec le polymorphisme. Cette compréhension vous aidera a éviter de désagréables plats de nouilles.

Ordre d'appel des constructeurs

L'ordre d'appel des constructeurs a été brièvement discuté dans le chapitre 4 et également dans le chapitre 6, mais c'était avant l'introduction du polymorphisme.

Un constructeur de la classe de base est toujours appelé dans le constructeur d'une classe dérivée, en remontant la hiérarchie d'héritage de sorte qu'un constructeur pour chaque classe de base est appelé. Ceci semble normal car le travail du constructeur est précisément de construire correctement l'objet. Une classe dérivée a seulement accès à ses propres membres, et pas à ceux de la classe de base (dont les membres sont en général **private**). Seul le constructeur de la classe de base a la connaissance et l'accès appropriés pour initialiser ses propres éléments. Par conséquent, il est essentiel que tous les constructeurs soient appelés, sinon l'objet ne serait pas entièrement construit. C'est pourquoi le compilateur impose un appel de constructeur pour chaque partie d'une classe dérivée. Il appellera silencieusement le constructeur par défaut si vous n'appelez pas explicitement un constructeur de la classe de base dans le corps du constructeur de la classe dérivée. S' il n'y a aucun constructeur de défaut, le compilateur le réclamera (dans le cas où une classe n'a aucun constructeur, le compilateur générera automatiquement un constructeur par défaut).

Prenons un exemple qui montre les effets de la composition, de l'héritage, et du polymorphisme sur l'ordre de construction :

```
//: c07:Sandwich.java
```

```
// Ordre d'appel des constructeurs.
class Meal {
  Meal() { System.out.println("Meal()"); }
}

class Bread {
  Bread() { System.out.println("Bread()"); }
}

class Cheese {
  Cheese() { System.out.println("Cheese()"); }
}

class Lettuce {
  Lettuce() { System.out.println("Lettuce()"); }
}

class Lunch extends Meal {
  Lunch() { System.out.println("Lunch()");}
}

class PortableLunch extends Lunch {
  PortableLunch() {
    System.out.println("PortableLunch()");
  }
}

class Sandwich extends PortableLunch {
  Bread b = new Bread();
  Cheese c = new Cheese();
  Lettuce l = new Lettuce();
  Sandwich() {
    System.out.println("Sandwich()");
  }
  public static void main(String[] args) {
    new Sandwich();
  }
} ///:~
```

Cet exemple utilise une classe complexe et d'autres classes, chaque classe a un constructeur qui s'annonce lui-même. La classe importante est **Sandwich**, qui est au troisième niveau d'héritage (quatre, si vous comptez l'héritage implicite de **Object**) et qui a trois objets membres. Quand un objet **Sandwich** est créé dans le **main()**, l'output est :

```
Meal()
Lunch()
PortableLunch()
```

```
Bread()
Cheese()
Lettuce()
Sandwich()
```

Ceci signifie que l'ordre d'appel des constructeurs pour un objet complexe est le suivant :

1. Le constructeur de la classe de base est appelé. Cette étape est répétée récursivement jusqu'à ce que la racine de la hiérarchie soit construite d'abord, suivie par la classe dérivée suivante, etc, jusqu'à atteindre la classe la plus dérivée.
2. Les initialiseurs des membres sont appelés dans l'ordre de déclaration
3. Le corps du constructeur de la classe dérivée est appelée.

L'ordre d'appel des constructeurs est important. Quand vous héritez, vous savez tout au sujet de la classe de base et pouvez accéder à tous les membres **public** et **protected** de la classe de base. Ceci signifie que vous devez pouvoir présumer que tous les membres de la classe de base sont valides quand vous êtes dans la classe dérivée. Dans une méthode normale, la construction a déjà eu lieu, ainsi tous les membres de toutes les parties de l'objet ont été construits. Dans le constructeur, cependant, vous devez pouvoir supposer que tous les membres que vous utilisez ont été construits. La seule manière de le garantir est d'appeler d'abord le constructeur de la classe de base. Ainsi, quand êtes dans le constructeur de la classe dérivée, tous les membres que vous pouvez accéder dans la classe de base ont été initialisés. Savoir que tous les membres sont valides à l'intérieur du constructeur est également la raison pour laquelle, autant que possible, vous devriez initialiser tous les objets membres (c'est à dire les objets mis dans la classe par composition) à leur point de définition dans la classe (par exemple, **b**, **c**, et **l** dans l'exemple ci-dessus). Si vous suivez cette recommandation, vous contribuerez à vous assurer que tous les membres de la classe de base *et* les objets membres de l'objet actuel aient été initialisés. Malheureusement, cela ne couvre pas tous les cas comme vous allez le voir dans le paragraphe suivant.

La méthode finalize() et l'héritage

Quand vous utilisez la composition pour créer une nouvelle classe, vous ne vous préoccupez pas de l'achèvement des objets membres de cette classe. Chaque membre est un objet indépendant et traité par le garbage collector indépendamment du fait qu'il soit un membre de votre classe. Avec l'héritage, cependant, vous devez redéfinir **finalize()** dans la classe dérivée si un nettoyage spécial doit être effectué pendant la phase de garbage collection. Quand vous redéfinissez **finalize()** dans une classe fille, il est important de ne pas oublier d'appeler la version de **finalize()** de la classe de base**,** sinon l'achèvement de la classe de base ne se produira pas. L'exemple suivant le prouve :

```
//: c07:Frog.java
// Test de la méthode finalize avec l'héritage.

class DoBaseFinalization {
  public static boolean flag = false;
}

class Characteristic {
  String s;
```

```
  Characteristic(String c) {
    s = c;
    System.out.println(
      "Creating Characteristic " + s);
  }
  protected void finalize() {
    System.out.println(
      "finalizing Characteristic " + s);
  }
}

class LivingCreature {
  Characteristic p =
    new Characteristic("is alive");
  LivingCreature() {
    System.out.println("LivingCreature()");
  }
  protected void finalize() {
    System.out.println(
      "LivingCreature finalize");
    // Appel de la version de la classe de base, à la fin!
    if(DoBaseFinalization.flag)
      try {
        super.finalize();
      } catch(Throwable t) {}
  }
}

class Animal extends LivingCreature {
  Characteristic p =
    new Characteristic("has heart");
  Animal() {
    System.out.println("Animal()");
  }
  protected void finalize() {
    System.out.println("Animal finalize");
    if(DoBaseFinalization.flag)
      try {
        super.finalize();
      } catch(Throwable t) {}
  }
}

class Amphibian extends Animal {
  Characteristic p =
    new Characteristic("can live in water");
  Amphibian() {
```

```
    System.out.println("Amphibian()");
  }
  protected void finalize() {
    System.out.println("Amphibian finalize");
    if(DoBaseFinalization.flag)
      try {
        super.finalize();
      } catch(Throwable t) {}
  }
}

public class Frog extends Amphibian {
  Frog() {
    System.out.println("Frog()");
  }
  protected void finalize() {
    System.out.println("Frog finalize");
    if(DoBaseFinalization.flag)
      try {
        super.finalize();
      } catch(Throwable t) {}
  }
  public static void main(String[] args) {
    if(args.length != 0 &&
       args[0].equals("finalize"))
      DoBaseFinalization.flag = true;
    else
        System.out.println("not finalizing bases");
    new Frog(); // Devient instantanément récupérable par le garbage collector
    System.out.println("bye!");
    // Force l'appel des finalisations:
    System.gc();
  }
} ///:~
```

Chaque classe dans la hiérarchie contient également un objet de la classe **Characteristic**. Vous constaterez que les objets de **Characteristic** sont toujours finalisés indépendamment de l'appel conditionné des finaliseurs de la classe de base.

Chaque méthode **finalize()** redéfinie doit au moins avoir accès aux membres **protected** puisque la méthode **finalize()** de la classe **Object** est **protected** et le que compilateur ne vous permettra pas de réduire l'accès pendant l'héritage (« Friendly » est moins accessible que **protected**).

Dans **Frog.main()**, l'indicateur **DoBaseFinalization** est configuré et un seul objet **Frog** est créé. Rappelez-vous que la phase de garbage collection, et en particulier la finalisation, ne peut pas avoir lieu pour un objet particulier, ainsi pour la forcer, l'appel à **System.gc()** déclenche le garbage collector, et ainsi la finalisation. Sans finalisation de la classe de base, l'output est le suivant :

```
not finalizing bases
Creating Characteristic is alive
LivingCreature()
Creating Characteristic has heart
Animal()
Creating Characteristic can live in water
Amphibian()
Frog()
bye!
Frog finalize
finalizing Characteristic is alive
finalizing Characteristic has heart
finalizing Characteristic can live in water
```

Vous pouvez constater qu'aucune finalisation n'est appelée pour les classes de base de **Frog** (les objets membres, eux, sont achevés, comme on s'y attendait). Mais si vous ajoutez l'argument « finalize » sur la ligne de commande, on obtient ce qui suit :

```
Creating Characteristic is alive
LivingCreature()
Creating Characteristic has heart
Animal()
Creating Characteristic can live in water
Amphibian()
Frog()
bye!
Frog finalize
Amphibian finalize
Animal finalize
LivingCreature finalize
finalizing Characteristic is alive
finalizing Characteristic has heart
finalizing Characteristic can live in water
```

Bien que l'ordre de finalisation des objets membres soit le même que l'ordre de création, l'ordre de finalisation des objets est techniquement non spécifié. Cependant, vous avez le contrôle sur cet ordre pour les classes de base. Le meilleur ordre à suivre est celui qui est montré ici, et qui est l'ordre inverse de l'initialisation. Selon le modèle qui est utilisé pour des destructeurs en C++, vous devez d'abord exécuter la finalisation des classes dérivées, puis la finalisation de la classe de base. La raison est que la finalisation des classes dérivées pourrait appeler des méthodes de la classe de base qui exigent que les composants de la classe de base soient toujours vivants, donc vous ne devez pas les détruire prématurément.

Comportement des méthodes polymorphes dans les constructeurs

La hiérarchie d'appel des constructeurs pose un dilemme intéressant. Qu'arrive t-il si à l'intérieur d'un constructeur vous appelez une méthode dynamiquement attachée de l'objet en cours de construction? À l'intérieur d'une méthode ordinaire vous pouvez imaginer ce qui arriverait: l'appel dynamiquement attaché est résolu à l'exécution parce que l'objet ne peut pas savoir s'il appartient à la classe dans laquelle se trouve la méthode ou bien dans une classe dérivée. Par cohérence, vous pourriez penser que c'est ce qui doit arriver dans les constructeurs.

Ce n'est pas ce qui se passe. Si vous appelez une méthode dynamiquement attachée à l'intérieur d'un constructeur, c'est la définition redéfinie de cette méthode est appelée. Cependant, *l'effet* peut être plutôt surprenant et peut cacher des bugs difficiles à trouver.

Le travail du constructeur est conceptuellement d'amener l'objet à l'existence (qui est à peine un prouesse ordinaire). À l'intérieur de n'importe quel constructeur, l'objet entier pourrait être seulement partiellement formé - vous pouvez savoir seulement que les objets de la classe de base ont été initialisés, mais vous ne pouvez pas connaître les classes filles qui hérite de vous. Cependant, un appel de méthode dynamiquement attaché, atteint « extérieurement » la hiérarchie d'héritage. Il appelle une méthode dans une classe dérivée. Si vous faites ça à l'intérieur d'un constructeur, vous appelez une méthode qui pourrait manipuler des membres non encore initialisés - une recette très sûre pour le désastre.

Vous pouvez voir le problème dans l'exemple suivant :

```
//: c07:PolyConstructors.java
// Constructeurs et polymorphisme ne conduisent
// pas ce à quoi que vous pourriez vous attendre.
abstract class Glyph {
  abstract void draw();
  Glyph() {
    System.out.println("Glyph() before draw()");
    draw();
    System.out.println("Glyph() after draw()");
  }
}

class RoundGlyph extends Glyph {
  int radius = 1;
  RoundGlyph(int r) {
    radius = r;
    System.out.println(
      "RoundGlyph.RoundGlyph(), radius = "
      + radius);
  }
  void draw() {
    System.out.println(
      "RoundGlyph.draw(), radius = " + radius);
  }
}
```

```
public class PolyConstructors {
  public static void main(String[] args) {
    new RoundGlyph(5);
  }
} ///:~
```

Dans **Glyph**, la méthode dessiner [**draw()**] est abstraite; elle a donc été conçue pour être redéfinie. En effet, vous êtes forcés de la redéfinir dans **RoundGlyph**. Mais le constructeur de **Glyph** appelle cette méthode et l'appel aboutit à **RoundGlyph.draw()**, ce qui semble être l'intention. Mais regardez l'output :

```
Glyph() before draw()
RoundGlyph.draw(), radius = 0
Glyph() after draw()
RoundGlyph.RoundGlyph(), radius = 5
```

Quand le constructeur de **Glyph** appelle **draw()**, le rayon [**radius**] n'a même pas encore la valeur initiale de 1, il vaut zéro. Le résultat serait probablement réduit à l'affichage d'un point ou même à rien du tout, avec vous, fixant un écran désespérément vide essayant de comprendre pourquoi le programme ne marche pas.

L'ordre de l'initialisation décrit dans la section précédente n'est pas complètement exhaustif, et c'est la clé qui va résoudre le mystère. La procédure d'initialisation est la suivante :

1. La zone allouée à l'objet est initialisée à zéro binaire avant tout.
2. Les constructeurs des classes de base sont appelés comme décrit précédemment. Puis, la méthode **draw()** redéfinie est appelée (et oui, *avant* l'appel du constructeur de **RoundGlyph**), et utilise radius qui vaut zéro à cause de la première étape.
3. Les initialiseurs des membres sont appelés dans l'ordre de déclaration.
4. Le corps du constructeur de la classe dérivée est appelé

Le bon coté est que tout est au moins initialisé au zéro (selon la signification de zéro pour un type de donnée particulier) et non laissé avec n'importe quelles valeurs. Cela inclut les références d'objet qui sont incorporés à l'intérieur d'une classe par composition, et qui passent à **null**. Ainsi si vous oubliez d'initialiser une référence vous obtiendrez une exception à l'exécution. Tout le reste est à zéro, qui est habituellement une valeur que l'on repère en examinant l'output.

D'autre part, vous devez être assez horrifiés du résultat de ce programme. Vous avez fait une chose parfaitement logique et pourtant le comportement est mystérieusement faux, sans aucune manifestation du compilateur (C ++ a un comportement plus correct dans la même situation). Les bugs dans ce goût là peuvent facilement rester cachés et nécessiter pas mal de temps d'investigation.

Il en résulte la recommandation suivante pour les constructeurs: « Faire le minimum pour mettre l'objet dans un bon état et si possible, ne pas appeler de méthodes. » Les seules méthodes qui sont appelables en toute sécurité à l'intérieur d'un constructeur sont celles qui sont finales dans la classe de base (même chose pour les méthodes privées, qui sont automatiquement finales.). Celles-ci ne peuvent être redéfinies et ne réservent donc pas de surprise.

Concevoir avec l'héritage

Après avoir vu le polymorphisme, c'est un instrument tellement astucieux qu'on dirait que tout doit être hérité. Ceci peut alourdir votre conception; en fait si vous faites le choix d'utiliser l'héritage d'entrée lorsque vous créez une nouvelle classe à partir d'une classe existante, cela peut devenir inutilement compliqué.

Une meilleure approche est de choisir d'abord la composition, quand il ne vous semble pas évident de choisir entre les deux. La composition n'oblige pas à concevoir une hiérarchie d'héritage, mais elle est également plus flexible car il est alors possible de choisir dynamiquement un type (et son comportement), alors que l'héritage requiert un type exact déterminé au moment de la compilation. L'exemple suivant l'illustre :

```
//: c07:Transmogrify.java
// Changer dynamiquement le comportement
// d'un objet par la composition.
abstract class Actor {
  abstract void act();
}

class HappyActor extends Actor {
  public void act() {
    System.out.println("HappyActor");
  }
}

class SadActor extends Actor {
  public void act() {
    System.out.println("SadActor");
  }
}

class Stage {
  Actor a = new HappyActor();
  void change() { a = new SadActor(); }
  void go() { a.act(); }
}

public class Transmogrify {
  public static void main(String[] args) {
    Stage s = new Stage();
    s.go(); // Imprime "HappyActor"
    s.change();
    s.go(); // Imprime "SadActor"
  }
} ///:~
```

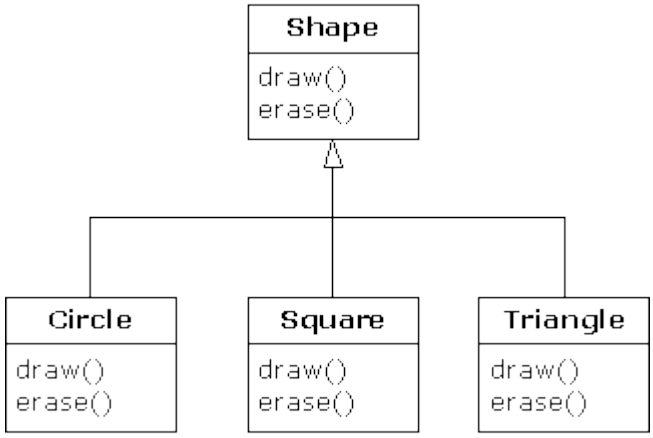
Un objet **Stage** contient une référence vers un **Actor**, qui est initialisé par un objet **HappyActor**. Cela signifie que **go()** produit un comportement particulier. Mais puisqu'une référence peut être

reliée à un objet différent à l'exécution, une référence à un objet **SadActor** peut être substituée dans **a** et alors le comportement produit par **go()** change. Ainsi vous gagnez en flexibilité dynamique à l'exécution (également appelé le *State Pattern.* Voir *Thinking in Patterns with Java*, téléchargeable sur *www.BruceEckel.com.*). Par contre, vous ne pouvez pas décider d'hériter différemment à l'exécution; cela doit être complètement déterminé à la compilation.

Voici une recommandation générale: « Utilisez l'héritage pour exprimer les différences de comportement, et les champs pour exprimer les variations d'état. » Dans l'exemple ci-dessus, les deux sont utilisés: deux classes différentes héritent pour exprimer la différence dans la méthode **act()**, et **Stage** utilise la composition pour permettre à son état d'être changé. Dans ce cas, ce changement d'état provoque un changement de comportement.

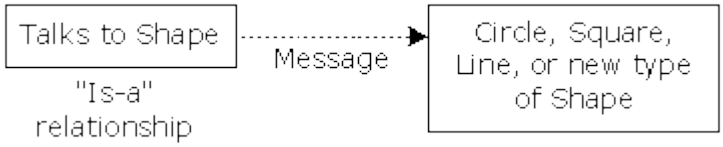
Héritage pur contre extensionname="Index739">

Lorsque l'on étudie l'héritage, il semblerait que la façon la plus propre de créer une hiérarchie d'héritage est de suivre l'approche « pure. » A savoir que seules les méthodes qui ont été établies dans la classe de base ou l'**interface** sont surchargeables dans la classe dérivée, comme le montre ce diagramme :



Ceci peut se nommer une relation « est-un » pure car l'interface d'une classe établie ce qu'elle est. L'héritage garantie que toute classe dérivée aura l'interface de la classe de base et rien de moins. Si vous suivez le diagramme ci-dessus, les classes dérivées auront également *pas plus* que l'interface de la classe de base.

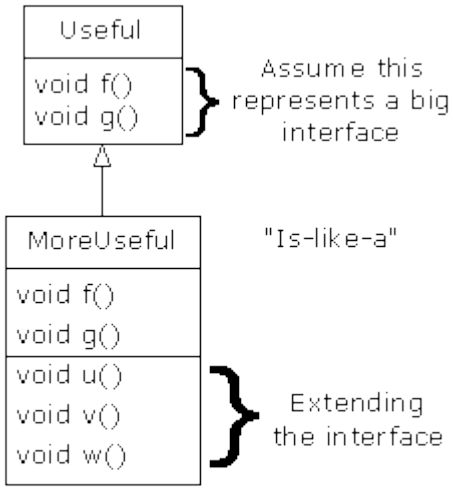
Ceci peut être considéré comme une *substitution pure*, car les objets de classe dérivée peuvent être parfaitement substitués par la classe de base, et vous n'avez jamais besoin de connaître d' information supplémentaire sur les sous-classes quand vous les utilisez :



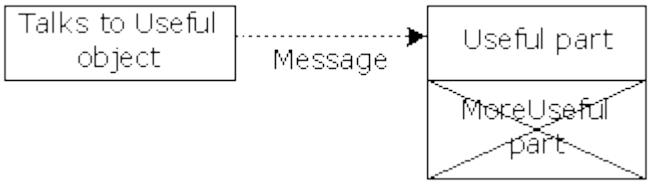
Cela étant, la classe de base peut recevoir tout message que vous pouvez envoyer à la classe dérivée car les deux ont exactement la même interface. Tout ce que vous avez besoin de faire est d'utiliser l'upcast à partir de la classe dérivée et de ne jamais regarder en arrière pour voir quel type

exact d'objet vous manipulez.

En la considérant de cette manière, une relation pure « est-un » semble la seule façon sensée de pratiquer, et toute autre conception dénote une réflexion embrouillée et est par définition hachée. Ceci aussi est un piège. Dès que vous commencez à penser de cette manière, vous allez tourner en rond et découvrir qu'étendre l'interface (ce que, malencontreusement, le mot clé **extends** semble encourager) est la solution parfaite à un problème particulier. Ceci pourrait être qualifié de relation « est-comme-un » car la classe dérivée est *comme* la classe de base, elle a la même interface fondamentale mais elle a d'autres éléments qui nécessitent d'implémenter des méthodes additionnelles :



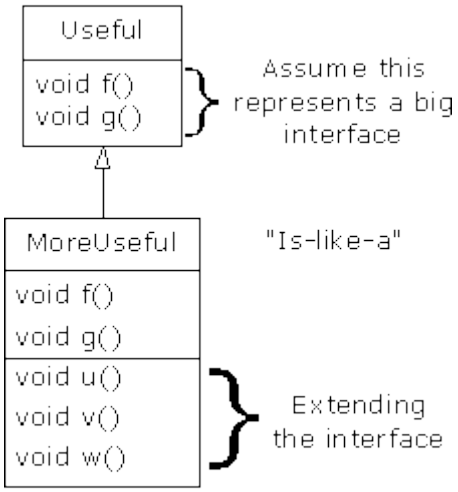
Mais si cette approche est aussi utile et sensée (selon la situation) elle a un inconvénient. La partie étendue de l'interface de la classe dérivée n'est pas accessible à partir de la classe de base, donc une fois que vous avez utilisé l'upcast vous ne pouvez pas invoquer les nouvelles méthodes :



Si vous n'upcastez pas dans ce cas, cela ne va pas vous incommoder, mais vous serez souvent dans une situation où vous aurez besoin de retrouver le type exact de l'objet afin de pouvoir accéder aux méthodes étendues de ce type. La section suivante montre comment cela se passe.

Downcasting et identification du type à l'exécution

Puisque vous avez perdu l'information du type spécifique par un *upcast* (en remontant la hiérarchie d'héritage), il est logique de retrouver le type en redescendant la hiérarchie d'héritage par un *downcast*. Cependant, vous savez qu'un upcast est toujours sûr; la classe de base ne pouvant pas avoir une interface plus grande que la classe dérivée, ainsi tout message que vous envoyez par l'interface de la classe de base a la garantie d'être accepté. Mais avec un downcast, vous ne savez pas vraiment qu'une forme (par exemple) est en réalité un cercle. Cela pourrait plutôt être un triangle ou un carré ou quelque chose d'un autre type.



Pour résoudre ce problème il doit y avoir un moyen de garantir qu'un downcast est correct, ainsi vous n'allez pas effectuer un cast accidentel vers le mauvais type et ensuite envoyer un message que l'objet ne pourrait accepter. Ce serait assez imprudent.

Dans certains langages (comme C++) vous devez effectuer une opération spéciale afin d'avoir un cast ascendant sûr, mais en Java *tout cast* est vérifié! Donc même si il semble que vous faites juste un cast explicite ordinaire, lors de l'exécution ce cast est vérifié pour assurer qu'en fait il s'agit bien du type auquel vous vous attendez. Si il ne l'est pas, vous récupérez une **ClassCastException**. Cette action de vérifier les types au moment de l'exécution est appelé *run-time type identification* (RTTI). L'exemple suivant montre le comportement de la RTTI :

```
//: c07:RTTI.java
// Downcasting & Run-time Type
// Identification (RTTI).
import java.util.*;

class Useful {
  public void f() {}
  public void g() {}
}

class MoreUseful extends Useful {
  public void f() {}
  public void g() {}
  public void u() {}
  public void v() {}
  public void w() {}
}

public class RTTI {
  public static void main(String[] args) {
    Useful[] x = {
      new Useful(),
```

```
        new MoreUseful()
      };
      x[0].f();
      x[1].g();
      // Compilation: méthode non trouvée dans Useful:
      //! x[1].u();
      ((MoreUseful)x[1]).u(); // Downcast/RTTI
      ((MoreUseful)x[0]).u(); // Exception envoyée
    }
} ///:~
```

Si vous voulez accéder à l'interface étendue d'un objet **MoreUseful**, vous pouvez essayer un downcast. Si c'est le type correct, cela fonctionnera. Autrement, vous allez recevoir une **ClassCastException**. Vous n'avez pas besoin d'écrire un code spécial pour cette exception, car elle indique une erreur du programmeur qui pourrait arriver n'importe où dans un programme.

La RTTI est plus riche qu'un simple cast. Par exemple, il y a une façon de connaître le type que vous manipulez *avant* d'essayer de le downcaster. Tout le Chapitre 12 est consacré à l'étude de différents aspects du « run-time type identification » Java.

Résumé

Polymorphisme signifie « différentes formes. » Dans la programmation orientée objet, vous avez la même physionomie (l'interface commune dans la classe de base) et différentes formes qui utilisent cette physionomie: les différentes versions des méthodes dynamiquement attachées.

Vous avez vu dans ce chapitre qu'il est impossible de comprendre, ou même créer, un exemple de polymorphisme sans utiliser l'abstraction et l'héritage. Le polymorphisme est une notion qui ne peut pas être présenté séparément (comme on peut le faire par exemple avec un **switch**), mais qui fonctionne plutôt en conjonction avec le schéma global #big picture# des relation entre classes. Les gens sont souvent troublés par d'autres dispositifs non-orientés objet de Java, comme la surcharge de méthode, qui sont parfois présentés comme étant orientés objet. Ne soyez pas dupe: si ce n'est pas de la liaison tardive, ce n'est pas du polymorphisme.

Pour utiliser le polymorphisme, et par conséquent les techniques orientées objet, pertinemment dans vos programmes vous devez élargir votre vision de la programmation pour y inclure non seulement les membres et les messages d'une classe individuelle, mais également ce qui est partagé entre les classes et leurs rapports entre elles. Bien que ceci exige un effort significatif, ça vaut vraiment le coup car il en résulte un développement plus rapide, un code mieux organisé, des programmes extensibles et une maintenance plus facile.

Exercices

1. Ajouter une nouvelle méthode à la classe de base de **Shapes.java** qui affiche un message, mais sans la redéfinir dans les classes dérivées. Expliquer ce qui se passe. Maintenant la redéfinir dans une des classes dérivées mais pas dans les autres, et voir ce qui se passe. Finalement, la redéfinir dans toutes les classes dérivées.
2. Ajouter un nouveau type de **Shape** à **Shapes.java** et vérifier dans **main()** que le polymorphisme fonctionne pour votre nouveau type comme il le fait pour les anciens types.

3. Changer **Music3.java** pour que **what()** devienne une méthode **toString()** de la classe racine **Object** . Essayer d'afficher les objets **Instrument** en utilisant **System.out.println()** (sans aucun cast).

4. Ajouter un nouveau type d'**Instrument** à **Music3.java** et vérifier que le polymorphisme fonctionne pour votre nouveau type.

5. Modifier **Music3.java** pour qu'il crée de manière aléatoire des objets **Instrument** de la même façon que **Shapes.java** le fait.

6. Créer une hiérarchie d'héritage de **Rongeur**: **Souris**, **Gerbille**, **Hamster**, etc. Dans la classe de base, fournir des méthodes qui sont communes à tous les **Rongeur**s, et les redéfinir dans les classes dérivées pour exécuter des comportements différents dépendant du type spécifique du **Rongeur**. Créer un tableau de **Rongeur**, le remplir avec différent types spécifiques de **Rongeur**s, et appeler vos méthodes de la classe de base pour voir ce qui arrive.

7. Modifier l'Exercice 6 pour que **Rongeur** soit une classe **abstract**. Rendre les méthodes de **Rongeur** abstraites dès que possible.

8. Créer une classe comme étant **abstract** sans inclure aucune méthode **abstract**, et vérifier que vous ne pouvez créer aucune instance de cette classe.

9. Ajouter la classe **Pickle** à **Sandwich.java**.

10. Modifier l'Exercice 6 afin qu'il démontre l'ordre des initialisations des classes de base et des classes dérivées. Maintenant ajouter des objets membres à la fois aux classes de base et dérivées, et montrer dans quel ordre leurs initialisations se produisent durant la construction.

11. Créer une hiérarchie d'héritage à 3 niveaux. Chaque classe dans la hiérarchie devra avoir une méthode **finalize()**, et devra invoquer correctement la version de la classe de base de **finalize()**. Démontrer que votre hiérarchie fonctionne de manière appropriée.

12. Créer une classe de base avec deux méthodes. Dans la première méthode, appeler la seconde méthode. Faire hériter une classe et redéfinir la seconde méthode. Créer un objet de la classe dérivée, upcaster le vers le type de base, et appeler la première méthode. Expliquer ce qui se passe.

13. Créer une classe de base avec une méthode **abstract print()** qui est redéfinie dans une classe dérivée. La version redéfinie de la méthode affiche la valeur d'une variable **int** définie dans la classe dérivée. Au point de définition de cette variable, lui donner une valeur non nulle. Dans le constructeur de la classe de base, appeler cette méthode. Dans **main()**, créer un objet du type dérivé, et ensuite appeler sa méthode **print()**. Expliquer les résultats.

14. Suivant l'exemple de **Transmogrify.java**, créer une classe **Starship** contenant une référence **AlertStatus** qui peut indiquer trois états différents. Inclure des méthodes pour changer les états.

15. Créer une classe **abstract** sans méthodes. Dériver une classe et ajouter une méthode. Créer une méthode **static** qui prend une référence vers la classe de base, effectue un downcast vers la classe dérivée, et appelle la méthode. Dans **main()**, démontrer que cela fonctionne. Maintenant mettre la déclaration **abstract** pour la méthode dans la classe de base, éliminant ainsi le besoin du downcast.

[37] Pour les programmeurs C++, ceci est analogue aux *fonctions virtuelles pures* du C++.

Chapitre 8 - Interfaces et classes internes

Les interfaces et les classes internes sont des manières plus sophistiquées d'organiser et de contrôler les objets du système construit.

C++, par exemple, ne propose pas ces mécanismes, bien que le programmeur expérimenté puisse les simuler. Le fait qu'ils soient présents dans Java indique qu'ils furent considérés comme assez importants pour être intégrés directement grâce à des mots-clefs.

Dans le chapitre 7, on a vu le mot-clef **abstract**, qui permet de créer une ou plusieurs méthodes dans une classe qui n'ont pas de définition - on fournit une partie de l'interface sans l'implémentation correspondante, qui est créée par ses héritiers. Le mot-clef **interface** produit une classe complètement abstraite, qui ne fournit absolument aucune implémentation. Nous verrons qu'une **interface** est un peu plus qu'une classe abstraite poussée à l'extrême, puisqu'elle permet d'implémenter « l'héritage multiple » du C++ en créant une classe qui peut être transtypée en plus d'un type de base.

Les classes internes ressemblent au premier abord à un simple mécanisme de dissimulation de code : on crée une classe à l'intérieur d'autres classes. Cependant, les classes internes font plus que cela - elles connaissent et peuvent communiquer avec la classe principale - ; sans compter que le code produit en utilisant les classes internes est plus élégant et compréhensible, bien que ce soit un concept nouveau pour beaucoup. Cela prend un certain temps avant d'intégrer les classes internes dans la conception.

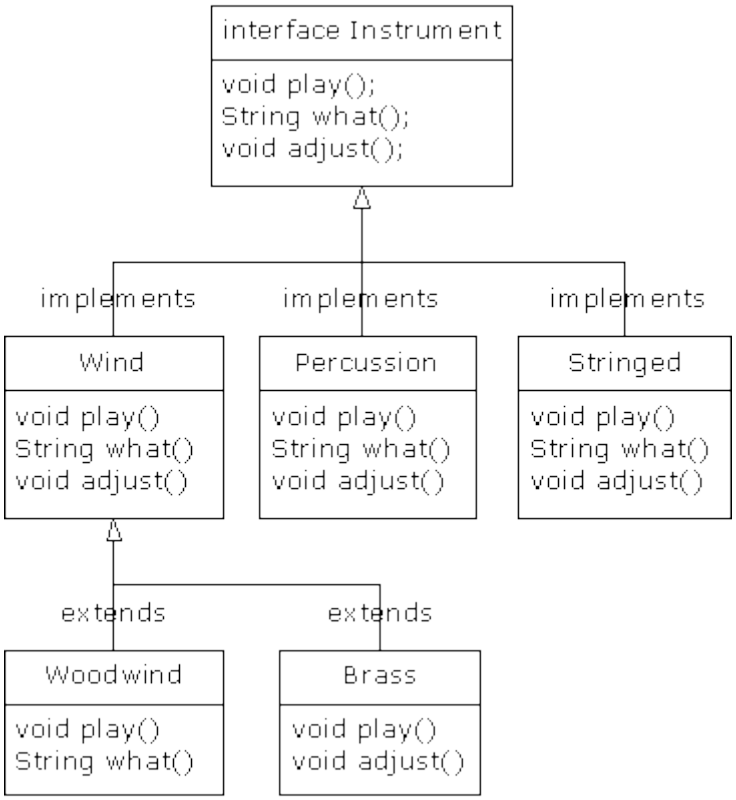
Interfaces

Le mot-clef **interface** pousse le concept **abstract** un cran plus loin. On peut y penser comme à une classe « purement » **abstract**. Il permet au créateur d'établir la forme qu'aura la classe : les noms des méthodes, les listes d'arguments et les types de retour, mais pas les corps des méthodes. Une **interface** peut aussi contenir des données membres, mais elles seront implicitement **static** et **final**. Une interface fournit un patron pour la classe, mais aucune implémentation.

Une **interface** déclare : « Voici ce à quoi ressemblera toutes les classes qui *implémenteront* cette interface ». Ainsi, tout code utilisant une **interface** particulière sait quelles méthodes peuvent être appelées pour cette **interface**, et c'est tout. Une **interface** est donc utilisée pour établir un « protocole » entre les classes (certains langages de programmation orientés objets ont un mot-clef *protocol* pour réaliser la même chose).

Pour créer une **interface**, il faut utiliser le mot-clef **interface** à la place du mot-clef **class**. Comme pour une classe, on peut ajouter le mot-clef **public** devant le mot-clef **interface** (mais seulement si l'**interface** est définie dans un fichier du même nom) ou ne rien mettre pour lui donner le statut « amical » afin qu'elle ne soit utilisable que dans le même package.

Le mot-clef **implements** permet de rendre une classe conforme à une **interface** particulière (ou à un groupe d'**interface**s). Il dit en gros : « L'**interface** spécifie ce à quoi la classe ressemble, mais maintenant on va spécifier comment cela *fonctionne* ». Sinon, cela s'apparente à de l'héritage. Le diagramme des instruments de musique suivant le montre :



Une fois une **interface** implémentée, cette implémentation devient une classe ordinaire qui peut être étendue d'une façon tout à fait classique.

On peut choisir de déclarer explicitement les méthodes d'une **interface** comme **public**. Mais elles sont **public** même sans le préciser. C'est pourquoi il faut définir les méthodes d'une **interface** comme **public** quand on implémente une **interface**. Autrement elles sont « amicales » par défaut, impliquant une réduction de l'accessibilité d'une méthode durant l'héritage, ce qui est interdit par le compilateur Java.

On peut le voir dans cette version modifiée de l'exemple **Instrument**. Notons que chaque méthode de l'**interface** n'est strictement qu'une déclaration, la seule chose que le compilateur permette. De plus, aucune des méthodes d'**Instrument** n'est déclarée comme **public**, mais elles le sont automatiquement.

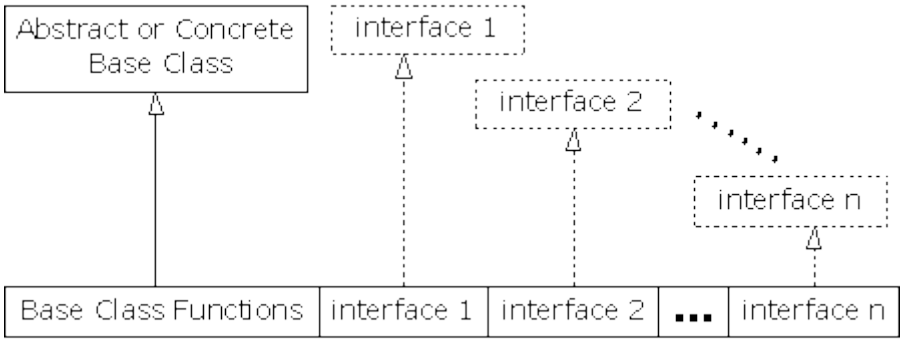
```
//: c08:music5:Music5.java
// Interfaces.
import java.util.*;

interface Instrument {
  // Constante compilée :
  int i = 5; // static & final
  // Définitions de méthodes interdites :
  void play(); // Automatiquement public
  String what();
  void adjust();
}
```

```
class Wind implements Instrument {
  public void play() {
    System.out.println("Wind.play()");
  }
  public String what() { return "Wind"; }
  public void adjust() {}
}

class Percussion implements Instrument {
  public void play() {
    System.out.println("Percussion.play()");
  }
  public String what() { return "Percussion"; }
  public void adjust() {}
}

class Stringed implements Instrument {
  public void play() {
    System.out.println("Stringed.play()");
  }
  public String what() { return "Stringed"; }
  public void adjust() {}
}

class Brass extends Wind {
  public void play() {
    System.out.println("Brass.play()");
  }
  public void adjust() {
    System.out.println("Brass.adjust()");
  }
}

class Woodwind extends Wind {
  public void play() {
    System.out.println("Woodwind.play()");
  }
  public String what() { return "Woodwind"; }
}

public class Music5 {
  // Le type n'est pas important, donc les nouveaux
  // types ajoutés au système marchent sans problème :
  static void tune(Instrument i) {
    // ...
    i.play();
```

```
  }
  static void tuneAll(Instrument[] e) {
    for(int i = 0; i < e.length; i++)
      tune(e[i]);
  }
  public static void main(String[] args) {
    Instrument[] orchestra = new Instrument[5];
    int i = 0;
    // Transtypage ascendant durant le stockage dans le tableau :
    orchestra[i++] = new Wind();
    orchestra[i++] = new Percussion();
    orchestra[i++] = new Stringed();
    orchestra[i++] = new Brass();
    orchestra[i++] = new Woodwind();
    tuneAll(orchestra);
  }
} ///:~
```

« Héritage multiple » en Java

Une **interface** n'est pas simplement une forme « plus pure » d'une classe **abstract**. Elle a un but plus important que cela. Puisqu'une **interface** ne dispose d'aucune implémentation - autrement dit, aucun stockage n'est associé à une **interface** -, rien n'empêche de combiner plusieurs **interface**s. Ceci est intéressant car certaines fois on a la relation "Un **x** est un **a** *et* un **b** *et* un **c**". En C++, le fait de combiner les interfaces de plusieurs classes est appelé *héritage multiple*, et entraîne une lourde charge du fait que chaque classe peut avoir sa propre implémentation. En Java, on peut réaliser la même chose, mais une seule classe peut avoir une implémentation, donc les problèmes rencontrés en C++ n'apparaissent pas en Java lorsqu'on combine les interfaces multiples :



Dans une classe dérivée, on n'est pas forcé d'avoir une classe de base qui soit **abstract** ou « concrète » (i.e. sans méthode **abstract**). Si une classe hérite d'une classe qui n'est pas une **interface**, elle ne peut dériver que de cette seule classe. Tous les autres types de base doivent être des **interface**s. On place les noms des interfaces après le mot-clef **implements** en les séparant par des virgules. On peut spécifier autant d'**interface**s qu'on veut - chacune devient un type indépendant vers lequel on peut transtyper. L'exemple suivant montre une classe concrète combinée à plusieurs **interface**s pour produire une nouvelle classe :

```
//: c08:Adventure.java
// Interfaces multiples.
import java.util.*;

interface CanFight {
  void fight();
}

interface CanSwim {
  void swim();
}

interface CanFly {
  void fly();
}

class ActionCharacter {
  public void fight() {}
}

class Hero extends ActionCharacter
    implements CanFight, CanSwim, CanFly {
  public void swim() {}
  public void fly() {}
}

public class Adventure {
  static void t(CanFight x) { x.fight(); }
  static void u(CanSwim x) { x.swim(); }
  static void v(CanFly x) { x.fly(); }
  static void w(ActionCharacter x) { x.fight(); }
  public static void main(String[] args) {
    Hero h = new Hero();
    t(h); // Le traite comme un CanFight
    u(h); // Le traite comme un CanSwim
    v(h); // Le traite comme un CanFly
    w(h); // Le traite comme un ActionCharacter
  }
} ///:~
```

Ici, **Hero** combine la classe concrète **ActionCharacter** avec les interfaces **CanFight**, **CanSwim** et **CanFly**. Quand on combine une classe concrète avec des interfaces de cette manière, la classe concrète doit être spécifiée en premier, avant les interfaces (autrement le compilateur génère une erreur).

Notons que la signature de **fight()** est la même dans l'**interface CanFight** et dans la classe **ActionCharacter**, et que **Hero** ne fournit *pas* de définition pour **fight()**. On peut hériter d'une **interface** (comme on va le voir bientôt), mais dans ce cas on a une autre **interface**. Si on veut créer un

objet de ce nouveau type, ce doit être une classe implémentant toutes les définitions. Bien que la classe **Hero** ne fournisse pas explicitement une définition pour **fight()**, la définition est fournie par **ActionCharacter**, donc héritée par **Hero** et il est ainsi possible de créer des objets **Hero**.

Dans la classe **Adventure**, on peut voir quatre méthodes prenant les diverses interfaces et la classe concrète en argument. Quand un objet **Hero** est créé, il peut être utilisé dans chacune de ces méthodes, ce qui veut dire qu'il est transtypé tour à tour dans chaque **interface**. De la façon dont cela est conçu en Java, cela fonctionne sans problème et sans effort supplémentaire de la part du programmeur.

L'intérêt principal des interfaces est démontré dans l'exemple précédent : être capable de transtyper vers plus d'un type de base. Cependant, une seconde raison, la même que pour les classes de base **abstract**, plaide pour l'utilisation des interfaces : empêcher le programmeur client de créer un objet de cette classe et spécifier qu'il ne s'agit que d'une interface. Cela soulève une question : faut-il utiliser une **interface** ou une classe **abstract** ? Une **interface** apporte les bénéfices d'une classe **abstract** *et* les bénéfices d'une **interface**, donc s'il est possible de créer la classe de base sans définir de méthodes ou de données membres, il faut toujours préférer les **interface**s aux classes **abstract**. En fait, si on sait qu'un type sera amené à être dérivé, il faut le créer d'emblée comme une **interface**, et ne le changer en classe **abstract**, voire en classe concrète, que si on est forcé d'y placer des définitions de méthodes ou des données membres.

Combinaison d'interfaces et collisions de noms

On peut renconter un problème lorsqu'on implémente plusieurs interfaces. Dans l'exemple précédent, **CanFight** et **ActionCharacter** ont tous les deux une méthode **void fight()** identique. Cela ne pose pas de problèmes parce que la méthode est identique dans les deux cas, mais que se passe-t-il lorsque ce n'est pas le cas ? Voici un exemple :

```
//: c08:InterfaceCollision.java

interface I1 { void f(); }
interface I2 { int f(int i); }
interface I3 { int f(); }
class C { public int f() { return 1; } }

class C2 implements I1, I2 {
  public void f() {}
  public int f(int i) { return 1; } // surchargée
}

class C3 extends C implements I2 {
  public int f(int i) { return 1; } // surchargée
}

class C4 extends C implements I3 {
  // Identique, pas de problème :
  public int f() { return 1; }
}
```

```
// Les méthodes diffèrent seulement par le type de retour :
//! class C5 extends C implements I1 {}
//! interface I4 extends I1, I3 {} ///:~
```

Les difficultés surviennent parce que la redéfinition, l'implémentation et la surcharge sont toutes les trois utilisées ensemble, et que les fonctions surchargées ne peuvent différer seulement par leur type de retour. Quand les deux dernières lignes sont décommentées, le message d'erreur est explicite :

```
InterfaceCollision.java:23: f() in C cannot
implement f() in I1; attempting to use
incompatible return type
found   : int
required: void
InterfaceCollision.java:24: interfaces I3 and I1 are incompatible; both define f
(), but with different return type
```

De toutes façons, utiliser les mêmes noms de méthode dans différentes interfaces destinées à être combinées affecte la compréhension du code. Il faut donc l'éviter autant que faire se peut.

Etendre une interface avec l'héritage

On peut facilement ajouter de nouvelles déclarations de méthodes à une **interface** en la dérivant, de même qu'on peut combiner plusieurs **interface**s dans une nouvelle **interface** grâce à l'héritage. Dans les deux cas on a une nouvelle **interface**, comme dans l'exemple suivant :

```
//: c08:HorrorShow.java
// Extension d'une interface grâce à l'héritage.

interface Monster {
  void menace();
}

interface DangerousMonster extends Monster {
  void destroy();
}

interface Lethal {
  void kill();
}

class DragonZilla implements DangerousMonster {
  public void menace() {}
  public void destroy() {}
}

interface Vampire
    extends DangerousMonster, Lethal {
```

```
  void drinkBlood();
}

class HorrorShow {
  static void u(Monster b) { b.menace(); }
  static void v(DangerousMonster d) {
    d.menace();
    d.destroy();
  }
  public static void main(String[] args) {
    DragonZilla if2 = new DragonZilla();
    u(if2);
    v(if2);
  }
} ///:~
```

DangerousMonster est une simple extension de **Monster** qui fournit une nouvelle **interface**. Elle est implémentée dans **DragonZilla**.

La syntaxe utilisée dans **Vampire** n'est valide *que* lorsqu'on dérive des interfaces. Normalement, on ne peut utiliser **extends** qu'avec une seule classe, mais comme une **interface** peut être constituée de plusieurs autres interfaces, **extends** peut se référer à plusieurs interfaces de base lorsqu'on construit une nouvelle **interface**. Comme on peut le voir, les noms d'**interface** sont simplement séparées par des virgules.

Groupes de constantes

Puisque toutes les données membres d'une **interface** sont automatiquement **static** et **final**, une **interface** est un outil pratique pour créer des groupes de constantes, un peu comme avec le **enum** du C ou du C++. Par exemple :

```
//: c08:Months.java
// Utiliser les interfaces pour créer des groupes de constantes.
package c08;

public interface Months {
  int
    JANUARY = 1, FEBRUARY = 2, MARCH = 3,
    APRIL = 4, MAY = 5, JUNE = 6, JULY = 7,
    AUGUST = 8, SEPTEMBER = 9, OCTOBER = 10,
    NOVEMBER = 11, DECEMBER = 12;
} ///:~
```

Notons au passage l'utilisation des conventions de style Java pour les champs **static final**s initialisés par des constantes : rien que des majuscules (avec des underscores pour séparer les mots à l'intérieur d'un identifiant).

Les données membres d'une **interface** sont automatiquement **public**, il n'est donc pas nécessaire de le spécifier.

Maintenant on peut utiliser les constantes à l'extérieur du package en important **c08.*** ou **c08.Months** de la même manière qu'on le ferait avec n'importe quel autre package, et référencer les valeurs avec des expressions comme **Months.JANUARY**. Bien sûr, on ne récupère qu'un **int**, il n'y a donc pas de vérification additionnelle de type comme celle dont dispose l'**enum** du C++, mais cette technique (couramment utilisée) reste tout de même une grosse amélioration comparée aux nombres codés en dur dans les programmes (appelés « nombres magiques » et produisant un code pour le moins difficile à maintenir).

Si on veut une vérification additionnelle de type, on peut construire une classe de la manière suivante [38]:

```
//: c08:Month2.java
// Un système d'énumération plus robuste.
package c08;

public final class Month2 {
  private String name;
  private Month2(String nm) { name = nm; }
  public String toString() { return name; }
  public final static Month2
    JAN = new Month2("January"),
    FEB = new Month2("February"),
    MAR = new Month2("March"),
    APR = new Month2("April"),
    MAY = new Month2("May"),
    JUN = new Month2("June"),
    JUL = new Month2("July"),
    AUG = new Month2("August"),
    SEP = new Month2("September"),
    OCT = new Month2("October"),
    NOV = new Month2("November"),
    DEC = new Month2("December");
  public final static Month2[] month =  {
    JAN, JAN, FEB, MAR, APR, MAY, JUN,
    JUL, AUG, SEP, OCT, NOV, DEC
  };
  public static void main(String[] args) {
    Month2 m = Month2.JAN;
    System.out.println(m);
    m = Month2.month[12];
    System.out.println(m);
    System.out.println(m == Month2.DEC);
    System.out.println(m.equals(Month2.DEC));
  }
} ///:~
```

Cette classe est appelée **Month2**, puisque **Month** existe déjà dans la bibliothèque Java standard. C'est une classe **final** avec un constructeur **private** afin que personne ne puisse la dériver ou en faire une instance. Les seules instances sont celles **static final** créées dans la classe elle-même

: **JAN**, **FEB**, **MAR**, etc. Ces objets sont aussi utilisés dans le tableau **month**, qui permet de choisir les mois par leur index au lieu de leur nom (notez le premier **JAN** dans le tableau pour introduire un déplacement supplémentaire de un, afin que Décembre soit le mois numéro 12). Dans **main()** on dispose de la vérification additionnelle de type : **m** est un objet **Month2** et ne peut donc se voir assigné qu'un **Month2**. L'exemple précédent (**Months.java**) ne fournissait que des valeurs **int**, et donc une variable **int** destinée à représenter un mois pouvait en fait recevoir n'importe quelle valeur entière, ce qui n'était pas très sûr.

Cette approche nous permet aussi d'utiliser indifféremment == ou **equals()**, ainsi que le montre la fin de **main()**.

Initialisation des données membres des interfaces

Les champs définis dans les interfaces sont automatiquement **static** et **final**. Ils ne peuvent être des « finals vides », mais peuvent être initialisés avec des expressions non constantes. Par exemple :

```
//: c08:RandVals.java
// Initialisation de champs d'interface
// avec des valeurs non-constantes.
import java.util.*;

public interface RandVals {
  int rint = (int)(Math.random() * 10);
  long rlong = (long)(Math.random() * 10);
  float rfloat = (float)(Math.random() * 10);
  double rdouble = Math.random() * 10;
} ///:~
```

Comme les champs sont **static**, ils sont initialisés quand la classe est chargée pour la première fois, ce qui arrive quand n'importe lequel des champs est accédé pour la première fois. Voici un simple test :

```
//: c08:TestRandVals.java

public class TestRandVals {
  public static void main(String[] args) {
    System.out.println(RandVals.rint);
    System.out.println(RandVals.rlong);
    System.out.println(RandVals.rfloat);
    System.out.println(RandVals.rdouble);
  }
} ///:~
```

Les données membres, bien sûr, ne font pas partie de l'interface mais sont stockées dans la zone de stockage **static** de cette interface.

Interfaces imbriquées

[39]Les interfaces peuvent être imbriquées dans des classes ou à l'intérieur d'autres interfaces.

Ceci révèle nombre de fonctionnalités intéressantes :

```
//: c08:NestingInterfaces.java

class A {
  interface B {
    void f();
  }
  public class BImp implements B {
    public void f() {}
  }
  private class BImp2 implements B {
    public void f() {}
  }
  public interface C {
    void f();
  }
  class CImp implements C {
    public void f() {}
  }
  private class CImp2 implements C {
    public void f() {}
  }
  private interface D {
    void f();
  }
  private class DImp implements D {
    public void f() {}
  }
  public class DImp2 implements D {
    public void f() {}
  }
  public D getD() { return new DImp2(); }
  private D dRef;
  public void receiveD(D d) {
    dRef = d;
    dRef.f();
  }
}

interface E {
  interface G {
    void f();
  }
  // « public » est redondant :
  public interface H {
    void f();
  }
```

```
  void g();
  // Ne peut pas être private dans une interface :
  //! private interface I {}
}

public class NestingInterfaces {
  public class BImp implements A.B {
    public void f() {}
  }
  class CImp implements A.C {
    public void f() {}
  }
  // Ne peut pas implémenter une interface private sauf
  // à l'intérieur de la classe définissant cette interface :
  //! class DImp implements A.D {
  //!  public void f() {}
  //! }
  class EImp implements E {
    public void g() {}
  }
  class EGImp implements E.G {
    public void f() {}
  }
  class EImp2 implements E {
    public void g() {}
    class EG implements E.G {
      public void f() {}
    }
  }
  public static void main(String[] args) {
    A a = new A();
    // Ne peut accéder à A.D :
    //! A.D ad = a.getD();
    // Ne renvoie qu'un A.D :
    //! A.DImp2 di2 = a.getD();
    // Ne peut accéder à un membre de l'interface :
    //! a.getD().f();
    // Seul un autre A peut faire quelque chose avec getD() :
    A a2 = new A();
    a2.receiveD(a.getD());
  }
} ///:~
```

La syntaxe permettant d'imbriquer une interface à l'intérieur d'une classe est relativement évidente ; et comme les interfaces non imbriquées, elles peuvent avoir une visibilité **public** ou « amicale ». On peut aussi constater que les interfaces **public** et « amicales » peuvent être implémentées dans des classes imbriquées **public**, « amicales » ou **private**.

Une nouvelle astuce consiste à rendre les interfaces **private** comme **A.D** (la même syntaxe est utilisée pour la qualification des interfaces imbriquées et pour les classes imbriquées). A quoi sert une interface imbriquée **private** ? On pourrait penser qu'elle ne peut être implémentée que comme une classe **private** imbriquée comme **DImp**, mais **A.DImp2** montre qu'elle peut aussi être implémentée dans une classe **public**. Cependant, **A.DImp2** ne peut être utilisée que comme elle-même : on ne peut mentionner le fait qu'elle implémente l'interface **private**, et donc implémenter une interface **private** est une manière de forcer la définition des méthodes de cette interface sans ajouter aucune information de type (c'est à dire, sans autoriser de transtypage ascendant).

La méthode **getD()** se trouve quant à elle dans une impasse du fait de l'interface **private** : c'est une méthode **public** qui renvoie une référence à une interface **private**. Que peut-on faire avec la valeur de retour de cette méthode ? Dans **main()**, on peut voir plusieurs tentatives pour utiliser cette valeur de retour, qui échouent toutes. La seule solution possible est lorsque la valeur de retour est gérée par un objet qui a la permission de l'utiliser - dans ce cas, un objet **A**, via la méthode **receiveD()**.

L'interface **E** montre que les interfaces peuvent être imbriquées les unes dans les autres. Cependant, les règles portant sur les interfaces - en particulier celle stipulant que tous les éléments doivent être **public** - sont strictement appliquées, donc une interface imbriquée à l'intérieur d'une autre interface est automatiquement **public** et ne peut être déclarée **private**.

NestingInterfaces montre les différentes manières dont les interfaces imbriquées peuvent être implémentées. En particulier, il est bon de noter que lorsqu'on implémente une interface, on n'est pas obligé d'en implémenter les interfaces imbriquées. De plus, les interfaces **private** ne peuvent être implémentées en dehors de leur classe de définition.

On peut penser que ces fonctionnalités n'ont été introduites que pour assurer une cohérence syntaxique, mais j'ai remarqué qu'une fois qu'une fonctionnalité est connue, on découvre souvent des endroits où elle se révèle utile.

Classes internes

Il est possible de placer la définition d'une classe à l'intérieur de la définition d'une autre classe. C'est ce qu'on appelle une classe interne. Les classes internes sont une fonctionnalité importante du langage car elles permettent de grouper les classes qui sont logiquement rattachées entre elles, et de contrôler la visibilité de l'une à partir de l'autre. Cependant, il est important de comprendre que le mécanisme des classes internes est complètement différent de celui de la composition.

Souvent, lorsqu'on en entend parler pour la première fois, l'intérêt des classes internes n'est pas immédiatement évident. A la fin de cette section, après avoir discuté de la syntaxe et de la sémantique des classes internes, vous trouverez des exemples qui devraient clairement montrer les bénéfices des classes internes.

Une classe interne est créée comme on pouvait s'y attendre - en plaçant la définition de la classe à l'intérieur d'une autre classe :

```
//: c08:Parcel1.java
// Création de classes internes.

public class Parcel1 {
```

```
  class Contents {
    private int i = 11;
    public int value() { return i; }
  }
  class Destination {
    private String label;
    Destination(String whereTo) {
      label = whereTo;
    }
    String readLabel() { return label; }
  }
  // L'utilisation d'une classe interne ressemble à
  // l'utilisation de n'importe quelle autre classe depuis Parcell :
  public void ship(String dest) {
    Contents c = new Contents();
    Destination d = new Destination(dest);
    System.out.println(d.readLabel());
  }
  public static void main(String[] args) {
    Parcel1 p = new Parcel1();
    p.ship("Tanzania");
  }
} ///:~
```

Les classes internes, quand elles sont utilisées dans **ship()**, ressemblent à n'importe quelle autre classe. La seule différence en est que les noms sont imbriqués dans **Parcel1**. Mais nous allons voir dans un moment que ce n'est pas la seule différence.

Plus généralement, une classe externe peut définir une méthode qui renvoie une référence à une classe interne, comme ceci :

```
//: c08:Parcel2.java
// Renvoyer une référence à une classe interne.

public class Parcel2 {
  class Contents {
    private int i = 11;
    public int value() { return i; }
  }
  class Destination {
    private String label;
    Destination(String whereTo) {
      label = whereTo;
    }
    String readLabel() { return label; }
  }
  public Destination to(String s) {
    return new Destination(s);
```

```
  }
  public Contents cont() {
    return new Contents();
  }
  public void ship(String dest) {
    Contents c = cont();
    Destination d = to(dest);
    System.out.println(d.readLabel());
  }
  public static void main(String[] args) {
    Parcel2 p = new Parcel2();
    p.ship("Tanzania");
    Parcel2 q = new Parcel2();
    // Définition de références sur des classes internes :
    Parcel2.Contents c = q.cont();
    Parcel2.Destination d = q.to("Borneo");
  }
} ///:~
```

Si on veut créer un objet de la classe interne ailleurs que dans une méthode non-**static** de la classe externe, il faut spécifier le type de cet objet comme *NomDeClasseExterne.NomDeClasseInterne*, comme on peut le voir dans **main()**.

Classes internes et transtypage ascendant

Jusqu'à présent, les classes internes ne semblent pas tellement intéressantes. Après tout, si le but recherché est le camouflage, Java propose déjà un très bon mécanisme pour cela - il suffit de rendre la classe « amicale » (visible seulement depuis un certain package) plutôt que de la déclarer comme une classe interne.

Cependant, les classes internes prennent de l'intérêt lorsqu'on transtype vers une classe de base, et en particulier vers une **interface** (produire une référence vers une interface depuis un objet l'implémentant revient à transtyper vers une classe de base). En effet la classe interne - l'implémentation de l'**interface** - est complètement masquée et indisponible pour tout le monde, ce qui est pratique pour cacher l'implémentation. La seule chose qu'on récupère est une référence sur la classe de base ou l'**interface**.

Tout d'abord, les interfaces sont définies dans leurs propres fichiers afin de pouvoir être utilisées dans tous les exemples :

```
//: c08:Destination.java
public interface Destination {
  String readLabel();
} ///:~
```

```
//: c08:Contents.java
public interface Contents {
```

```
  int value();
} ///:~
```

Maintenant **Contents** et **Destination** sont des interfaces disponibles pour le programmeur client (une **interface** déclare automatiquement tous ses membres comme **public**).

Quand on récupère une référence sur la classe de base ou l'**interface**, il est possible qu'on ne puisse même pas en découvrir le type exact, comme on peut le voir dans le code suivant :

```
//: c08:Parcel3.java
// Renvoyer une référence sur une classe interne.

public class Parcel3 {
  private class PContents implements Contents {
    private int i = 11;
    public int value() { return i; }
  }
  protected class PDestination
      implements Destination {
    private String label;
    private PDestination(String whereTo) {
      label = whereTo;
    }
    public String readLabel() { return label; }
  }
  public Destination dest(String s) {
    return new PDestination(s);
  }
  public Contents cont() {
    return new PContents();
  }
}

class Test {
  public static void main(String[] args) {
    Parcel3 p = new Parcel3();
    Contents c = p.cont();
    Destination d = p.dest("Tanzania");
    // Illégal -- ne peut accéder à une classe private :
    //! Parcel3.PContents pc = p.new PContents();
  }
} ///:~
```

Notez que puisque **main()** se trouve dans **Test**, pour lancer ce programme il ne faut pas exécuter **Parcel3**, mais :

```
java Test
```

Dans **Parcel3**, de nouvelles particularités ont été ajoutées : la classe interne **PContents** est

private afin que seule **Parcel3** puisse y accéder. **PDestination** est **protected**, afin que seules **Parcel3**, les classes du packages de **Parcel3** (puisque **protected** fournit aussi un accès package - c'est à dire que **protected** est « amical »), et les héritiers de **Parcel3** puissent accéder à **PDestination**. Cela signifie que le programmeur client n'a qu'une connaissance et des accès restreints à ces membres. En fait, on ne peut faire de transtypage descendant vers une classe interne **private** (ou une classe interne **protected** à moins d'être un héritier), parce qu'on ne peut accéder à son nom, comme on peut le voir dans la classe **Test**. La classe interne **private** fournit donc un moyen pour le concepteur de la classe d'interdire tout code testant le type et de cacher complètement les détails de l'implémentation. De plus, l'extension d'une **interface** est inutile du point de vue du programmeur client puisqu'il ne peut accéder à aucune méthode additionnelle ne faisant pas partie de l'**interface public** de la classe. Cela permet aussi au compilateur Java de générer du code plus efficace.

Les classes normales (non internes) ne peuvent être déclarées **private** ou **protected** - uniquement **public** ou « amicales ».

Classes internes définies dans des méthodes et autres portées

Ce qu'on a pu voir jusqu'à présent constitue l'utilisation typique des classes internes. En général, le code impliquant des classes internes que vous serez amené à lire et à écrire ne mettra en oeuvre que des classes internes « régulières », et sera simple à comprendre. Cependant, le support des classes internes est relativement complet et il existe de nombreuses autres manières, plus obscures, de les utiliser si on le souhaite : les classes internes peuvent être créées à l'intérieur d'une méthode ou même d'une portée arbitraire. Deux raisons possibles à cela :

1. Comme montré précédemment, on implémente une interface d'un certain type afin de pouvoir créer et renvoyer une référence.
2. On résoud un problème compliqué pour lequel la création d'une classe aiderait grandement, mais on ne veut pas la rendre publiquement accessible.

Dans les exemples suivants, le code précédent est modifié afin d'utiliser :

1. Une classe définie dans une méthode
2. Une classe définie dans une portée à l'intérieur d'une méthode
3. Une classe anonyme implémentant une interface
4. Une classe anonyme étendant une classe qui dispose d'un constructeur autre que le constructeur par défaut
5. Une classe anonyme réalisant des initialisations de champs
6. Une classe anonyme qui se construit en initialisant des instances (les classes internes anonymes ne peuvent avoir de constructeurs)

Bien que ce soit une classe ordinaire avec une implémentation, **Wrapping** est aussi utilisée comme une « interface » commune pour ses classes dérivées :

```
//: c08:Wrapping.java
public class Wrapping {
  private int i;
  public Wrapping(int x) { i = x; }
  public int value() { return i; }
```

```
} ///:~
```

Notez que **Wrapping** dispose d'un constructeur requérant un argument, afin de rendre les choses un peu plus intéressantes.

Le premier exemple montre la création d'une classe entière dans la portée d'une méthode (au lieu de la portée d'une autre classe) :

```
//: c08:Parcel4.java
// Définition d'une classe à l'intérieur d'une méthode.

public class Parcel4 {
  public Destination dest(String s) {
    class PDestination
       implements Destination {
      private String label;
      private PDestination(String whereTo) {
        label = whereTo;
      }
      public String readLabel() { return label; }
    }
    return new PDestination(s);
  }
  public static void main(String[] args) {
    Parcel4 p = new Parcel4();
    Destination d = p.dest("Tanzania");
  }
} ///:~
```

La classe **PDestination** appartient à **dest()** plutôt qu'à **Parcel4** (notez aussi qu'on peut utiliser l'identifiant **PDestination** pour une classe interne à l'intérieur de chaque classe du même sous-répertoire sans collision de nom). Cependant, **PDestination** ne peut être accédée en dehors de **dest()**. Notez le transtypage ascendant réalisé par l'instruction de retour - **dest()** ne renvoie qu'une référence à **Destination**, la classe de base. Bien sûr, le fait que le nom de la classe **PDestination** soit placé à l'intérieur de **dest()** ne veut pas dire que **PDestination** n'est pas un objet valide une fois sorti de **dest()**.

L'exemple suivant montre comment on peut imbriquer une classe interne à l'intérieur de n'importe quelle portée :

```
//: c08:Parcel5.java
// Définition d'une classe à l'intérieur d'une portée quelconque.

public class Parcel5 {
  private void internalTracking(boolean b) {
    if(b) {
      class TrackingSlip {
        private String id;
        TrackingSlip(String s) {
```

```
        id = s;
      }
      String getSlip() { return id; }
    }
    TrackingSlip ts = new TrackingSlip("slip");
    String s = ts.getSlip();
  }
  // Utilisation impossible ici ! En dehors de la portée :
  //! TrackingSlip ts = new TrackingSlip("x");
 }
 public void track() { internalTracking(true); }
 public static void main(String[] args) {
  Parcel5 p = new Parcel5();
  p.track();
 }
} ///:~
```

La classe **TrackingSlip** est définie dans la portée de l'instruction **if**. Cela ne veut pas dire que la classe est créée conditionnellement - elle est compilée avec tout le reste. Cependant, elle n'est pas accessible en dehors de la portée dans laquelle elle est définie. Mis à part cette restriction, elle ressemble à n'importe quelle autre classe ordinaire.

Classes internes anonymes

L'exemple suivant semble relativement bizarre :

```
//: c08:Parcel6.java
// Une méthode qui renvoie une classe interne anonyme.

public class Parcel6 {
 public Contents cont() {
  return new Contents() {
   private int i = 11;
   public int value() { return i; }
  }; // Point-virgule requis dans ce cas
 }
 public static void main(String[] args) {
  Parcel6 p = new Parcel6();
  Contents c = p.cont();
 }
} ///:~
```

La méthode **cont()** combine la création d'une valeur de retour avec la définition de la classe de cette valeur de retour ! De plus, la classe est anonyme - elle n'a pas de nom. Pour compliquer le tout, il semble qu'on commence par créer un objet **Contents** :

```
return new Contents()
```

Mais alors, avant de terminer l'instruction par un point-virgule, on dit : « Eh, je crois que je

vais insérer une définition de classe » :

```
return new Contents() {

  private int i = 11;
  public int value() { return i; }
};
```

Cette étrange syntaxe veut dire : « Crée un objet d'une classe anonyme dérivée de **Contents** ». La référence renvoyée par l'expression **new** est automatiquement transtypée vers une référence **Contents**. La syntaxe d'une classe interne anonyme est un raccourci pour :

```
class MyContents implements Contents {
  private int i = 11;
  public int value() { return i; }
}
return new MyContents();
```

Dans la classe interne anonyme, **Contents** est créée avec un constructeur par défaut. Le code suivant montre ce qu'il faut faire dans le cas où la classe de base dispose d'un constructeur requérant un argument :

```
//: c08:Parcel7.java
// Une classe interne anonyme qui appelle
// le constructeur de la classe de base.

public class Parcel7 {
  public Wrapping wrap(int x) {
    // Appel du constructeur de la classe de base :
    return new Wrapping(x) {
      public int value() {
        return super.value() * 47;
      }
    }; // Point-virgule requis
  }
  public static void main(String[] args) {
    Parcel7 p = new Parcel7();
    Wrapping w = p.wrap(10);
  }
} ///:~
```

Autrement dit, on passe simplement l'argument approprié au constructeur de la classe de base, vu ici comme le **x** utilisé dans **new Wrapping(x)**. Une classe anonyme ne peut avoir de constructeur dans lequel on appellerait normalement **super()**.

Dans les deux exemples précédents, le point-virgule ne marque pas la fin du corps de la classe (comme il le fait en C++). Il marque la fin de l'expression qui se trouve contenir la définition de la classe anonyme. Son utilisation est donc similaire à celle que l'on retrouve partout ailleurs.

Que se passe-t-il lorsque certaines initialisations sont nécessaires pour un objet d'une classe interne anonyme ? Puisqu'elle est anonyme, on ne peut donner de nom au constructeur - et on ne

peut donc avoir de constructeur. On peut néanmoins réaliser des initialisations lors de la définition des données membres :

```
//: c08:Parcel8.java
// Une classe interne anonyme qui réalise
// des initialisations. Une version plus courte
// de Parcel5.java.

public class Parcel8 {
  // L'argument doit être final pour être utilisé
  // la classe interne anonyme :
  public Destination dest(final String dest) {
    return new Destination() {
      private String label = dest;
      public String readLabel() { return label; }
    };
  }
  public static void main(String[] args) {
    Parcel8 p = new Parcel8();
    Destination d = p.dest("Tanzania");
  }
} ///:~
```

Si on définit une classe interne anonyme et qu'on veut utiliser un objet défini en dehors de la classe interne anonyme, le compilateur requiert que l'objet extérieur soit **final**. C'est pourquoi l'argument de **dest()** est **final**. Si le mot-clef est omis, le compilateur générera un message d'erreur.

Tant qu'on se contente d'assigner un champ, l'approche précédente est suffisante. Mais comment faire si on a besoin de réaliser plusieurs actions comme un constructeur peut être amené à le faire ? Avec l'*initialisation d'instances*, on peut, dans la pratique, créer un constructeur pour une classe interne anonyme :

```
//: c08:Parcel9.java
// Utilisation des « initialisations d'instances » pour
// réaliser la construction d'une classe interne anonyme.

public class Parcel9 {
  public Destination
  dest(final String dest, final float price) {
    return new Destination() {
      private int cost;
      // Initialisation d'instance pour chaque objet :
      {
        cost = Math.round(price);
        if(cost > 100)
          System.out.println("Over budget!");
      }
      private String label = dest;
      public String readLabel() { return label; }
```

```
    };
  }
  public static void main(String[] args) {
    Parcel9 p = new Parcel9();
    Destination d = p.dest("Tanzania", 101.395F);
  }
} ///:~
```

A l'intérieur de l'initialisateur d'instance on peut voir du code pouvant ne pas être exécuté comme partie d'un initialisateur de champ (l'instruction **if**). Dans la pratique, donc, un initialisateur d'instance est un constructeur pour une classe interne anonyme. Bien sûr, ce mécanisme est limité ; on ne peut surcharger les initialisateurs d'instance et donc on ne peut avoir qu'un seul de ces constructeurs.

Lien vers la classe externe

Jusqu'à présent, les classes internes apparaissent juste comme un mécanisme de camouflage de nom et d'organisation du code, ce qui est intéressant mais pas vraiment indispensable. Cependant, elles proposent un autre intérêt. Quand on crée une classe interne, un objet de cette classe interne possède un lien vers l'objet extérieur qui l'a créé, il peut donc accéder aux membres de cet objet externe - *sans* aucune qualification spéciale. De plus, les classes internes ont accès à tous les éléments de la classe externe [40]. L'exemple suivant le démontre :

```
//: c08:Sequence.java
// Contient une séquence d'Objects.

interface Selector {
  boolean end();
  Object current();
  void next();
}

public class Sequence {
  private Object[] obs;
  private int next = 0;
  public Sequence(int size) {
    obs = new Object[size];
  }
  public void add(Object x) {
    if(next < obs.length) {
      obs[next] = x;
      next++;
    }
  }
  private class SSelector implements Selector {
    int i = 0;
    public boolean end() {
      return i == obs.length;
```

```
    }
    public Object current() {
      return obs[i];
    }
    public void next() {
      if(i < obs.length) i++;
    }
  }
  public Selector getSelector() {
    return new SSelector();
  }
  public static void main(String[] args) {
    Sequence s = new Sequence(10);
    for(int i = 0; i < 10; i++)
      s.add(Integer.toString(i));
    Selector sl = s.getSelector();
    while(!sl.end()) {
      System.out.println(sl.current());
      sl.next();
    }
  }
} ///:~
```

La **Sequence** est simplement un tableau d'**Object**s de taille fixe paqueté dans une classe. On peut appeler **add()** pour ajouter un nouvel **Object** à la fin de la séquence (s'il reste de la place). Pour retrouver chacun des objets dans une **Sequence**, il existe une interface appelée **Selector**, qui permet de vérifier si on se trouve à la fin (**end()**), de récupérer l'**Object** courant (**current()**), et de se déplacer vers l'**Object** suivant (**next()**) dans la **Sequence**. Comme **Selector** est une interface, beaucoup d'autres classes peuvent implémenter l'**interface** comme elles le veulent, et de nombreuses méthodes peuvent prendre l'**interface** comme un argument, afin de créer du code générique.

Ici, **SSelector** est une classe **private** qui fournit les fonctionnalités de **Selector**. Dans **main()**, on peut voir la création d'une **Sequence**, suivie par l'addition d'un certain nombre d'objets **String**. Un **Selector** est alors produit grâce à un appel à **getSelector()** et celui-ci est alors utilisé pour se déplacer dans la **Sequence** et sélectionner chaque item.

Au premier abord, **SSelector** ressemble à n'importe quelle autre classe interne. Mais regardez-la attentivement. Notez que chacune des méthodes **end()**, **current()** et **next()** utilisent **obs**, qui est une référence n'appartenant pas à **SSelector**, un champ **private** de la classe externe. Cependant, la classe interne peut accéder aux méthodes et aux champs de la classe externe comme si elle les possédait. Ceci est très pratique, comme on peut le voir dans cet exemple.

Une classe interne a donc automatiquement accès aux membres de la classe externe. Comment cela est-il possible ? La classe interne doit garder une référence de l'objet de la classe externe responsable de sa création. Et quand on accède à un membre de la classe externe, cette référence (cachée) est utilisée pour sélectionner ce membre. Heureusement, le compilateur gère tous ces détails pour nous, mais vous pouvez maintenant comprendre qu'un objet d'une classe interne ne peut être créé qu'en association avec un objet de la classe externe. La construction d'un objet d'une classe interne requiert une référence sur l'objet de la classe externe, et le compilateur se plaindra s'il ne peut accéder à cette référence. La plupart du temps cela se fait sans aucune intervention de la part

du programmeur.

Classes internes static

Si on n'a pas besoin du lien entre l'objet de la classe interne et l'objet de la classe externe, on peut rendre la classe interne **static**. Pour comprendre le sens de **static** quand il est appliqué aux classes internes, il faut se rappeler que l'objet d'une classe interne ordinaire garde implicitement une référence sur l'objet externe qui l'a créé. Ceci n'est pas vrai cependant lorsque la classe interne est **static**. Une classe interne **static** implique que :

Les classes internes **static** diffèrent aussi des classes internes non **static** d'une autre manière. Les champs et les méthodes des classes internes non **static** ne peuvent être qu'au niveau externe de la classe, les classes internes non **static** ne peuvent donc avoir de données **static**, de champs **static** ou de classes internes **static**. Par contre, les classes internes **static** peuvent avoir tout cela :

```
//: c08:Parcel10.java
// Classes internes static.

public class Parcel10 {
  private static class PContents
  implements Contents {
    private int i = 11;
    public int value() { return i; }
  }
  protected static class PDestination
      implements Destination {
    private String label;
    private PDestination(String whereTo) {
      label = whereTo;
    }
    public String readLabel() { return label; }
    // Les classes internes static peuvent
    // contenir d'autres éléments static :
    public static void f() {}
    static int x = 10;
    static class AnotherLevel {
      public static void f() {}
      static int x = 10;
    }
  }
  public static Destination dest(String s) {
    return new PDestination(s);
  }
  public static Contents cont() {
    return new PContents();
  }
  public static void main(String[] args) {
    Contents c = cont();
    Destination d = dest("Tanzania");
```

```
  }
} ///:~
```

Dans **main()**, aucun objet **Parcel10** n'est nécessaire ; on utilise à la place la syntaxe habituelle pour sélectionner un membre **static** pour appeler les méthodes qui renvoient des références sur **Contents** et **Destination**.

Comme on va le voir bientôt, dans une classe interne ordinaire (non **static**), le lien avec la classe externe est réalisé avec une référence spéciale **this**. Une classe interne **static** ne dispose pas de cette référence spéciale **this**, ce qui la rend analogue à une méthode **static**.

Normalement, on ne peut placer du code à l'intérieur d'une **interface**, mais une classe interne **static** peut faire partie d'une **interface**. Comme la classe est **static**, cela ne viole pas les règles des interfaces - la classe interne **static** est simplement placée dans l'espace de noms de l'interface :

```
//: c08:IInterface.java
// Classes internes static à l'intérieur d'interfaces.

interface IInterface {
  static class Inner {
    int i, j, k;
    public Inner() {}
    void f() {}
  }
} ///:~
```

Plus tôt dans ce livre je suggérais de placer un **main()** dans chaque classe se comportant comme un environnement de tests pour cette classe. Un inconvénient de cette approche est le volume supplémentaire de code compilé qu'on doit supporter. Si cela constitue un problème, on peut utiliser une classe interne **static** destinée à contenir le code de test :

```
//: c08:TestBed.java
// Code de test placé dans une classe interne static.

class TestBed {
  TestBed() {}
  void f() { System.out.println("f()"); }
  public static class Tester {
    public static void main(String[] args) {
      TestBed t = new TestBed();
      t.f();
    }
  }
} ///:~
```

Ceci génère une classe séparée appelée **TestBed$Tester** (pour lancer le programme, il faut utiliser la commande **java TestBed$Tester**). On peut utiliser cette classe lors des tests, mais on n'a pas besoin de l'inclure dans le produit final.

Se référer à l'objet de la classe externe

Si on a besoin de produire la référence à l'objet de la classe externe, il faut utiliser le nom de la classe externe suivi par un point et **this**. Par exemple, dans la classe **Sequence.SSelector**, chacune des méthodes peut accéder à la référence à la classe externe **Sequence** stockée en utilisant **Sequence.this**. Le type de la référence obtenue est automatiquement correct (il est connu et vérifié lors de la compilation, il n'y a donc aucune pénalité sur les performances lors de l'exécution).

On peut demander à un autre objet de créer un objet de l'une de ses classes internes. Pour cela il faut fournir une référence à l'autre objet de la classe externe dans l'expression **new**, comme ceci :

```
//: c08:Parcel11.java
// Création d'instances de classes internes.

public class Parcel11 {
  class Contents {
    private int i = 11;
    public int value() { return i; }
  }
  class Destination {
    private String label;
    Destination(String whereTo) {
      label = whereTo;
    }
    String readLabel() { return label; }
  }
  public static void main(String[] args) {
    Parcel11 p = new Parcel11();
    // On doit utiliser une instance de la classe externe
    // pour créer une instance de la classe interne :
    Parcel11.Contents c = p.new Contents();
    Parcel11.Destination d =      p.new Destination("Tanzania");
  }
} ///:~
```

Pour créer un objet de la classe interne directement, il ne faut pas utiliser la même syntaxe et se référer au nom de la classe externe **Parcel11** comme on pourrait s'y attendre ; à la place il faut utiliser un *objet* de la classe externe pour créer un objet de la classe interne :

```
Parcel11.Contents c = p.new Contents();
```

Il n'est donc pas possible de créer un objet de la classe interne sans disposer déjà d'un objet de la classe externe, parce qu'un objet de la classe interne est toujours connecté avec l'objet de la classe externe qui l'a créé. Cependant, si la classe interne est **static**, elle n'a pas besoin d'une référence sur un objet de la classe externe.

Classe interne à plusieurs niveaux d'imbrication

[41]Une classe interne peut se situer à n'importe quel niveau d'imbrication - elle pourra toujours accéder de manière transparente à tous les membres de toutes les classes l'entourant, comme

on peut le voir :

```
//: c08:MultiNestingAccess.java
// Les classes imbriquées peuvent accéder à tous les membres de tous
// les niveaux des classes dans lesquelles elles sont imbriquées.

class MNA {
  private void f() {}
  class A {
    private void g() {}
    public class B {
      void h() {
        g();
        f();
      }
    }
  }
}

public class MultiNestingAccess {
  public static void main(String[] args) {
    MNA mna = new MNA();
    MNA.A mnaa = mna.new A();
    MNA.A.B mnaab = mnaa.new B();
    mnaab.h();
  }
} ///:~
```

On peut voir que dans **MNA.A.B**, les méthodes **g()** et **f()** sont appelées sans qualification (malgré le fait qu'elles soient **private**). Cet exemple présente aussi la syntaxe utilisée pour créer des objets de classes internes imbriquées quand on crée ces objets depuis une autre classe. La syntaxe « **.new** » fournit la portée correcte et on n'a donc pas besoin de qualifier le nom de la classe dans l'appel du constructeur.

Dériver une classe interne

Comme le constructeur d'une classe interne doit stocker une référence à l'objet de la classe externe, les choses sont un peu plus compliquées lorsqu'on dérive une classe interne. Le problème est que la référence « secrète » sur l'objet de la classe externe *doit* être initialisée, et dans la classe dérivée il n'y a plus d'objet sur lequel se rattacher par défaut. Il faut donc utiliser une syntaxe qui rende cette association explicite :

```
//: c08:InheritInner.java
// Inheriting an inner class.

class WithInner {
  class Inner {}
}
```

```
public class InheritInner
    extends WithInner.Inner {
  //! InheritInner() {} // Ne compilera pas.
  InheritInner(WithInner wi) {
    wi.super();
  }
  public static void main(String[] args) {
    WithInner wi = new WithInner();
    InheritInner ii = new InheritInner(wi);
  }
} ///:~
```

On peut voir que **InheritInner** étend juste la classe interne, et non la classe externe. Mais lorsqu'on en arrive au constructeur, celui fourni par défaut n'est pas suffisant et on ne peut se contenter de passer une référence à un objet externe. De plus, on doit utiliser la syntaxe :

```
enclosingClassReference.super();
```

à l'intérieur du constructeur. Ceci fournit la référence nécessaire et le programme pourra alors être compilé.

Les classes internes peuvent-elles redéfinies ?

Que se passe-t-il quand on crée une classe interne, qu'on dérive la classe externe et qu'on redéfinit la classe interne ? Autrement dit, est-il possible de rédéfinir une classe interne ? Ce concept semble particulièrement puissant, mais « redéfinir » une classe interne comme si c'était une méthode de la classe externe ne fait rien de spécial :

```
//: c08:BigEgg.java
// Une classe interne ne peut être
// redéfinie comme une méthode.

class Egg {
  protected class Yolk {
    public Yolk() {
      System.out.println("Egg.Yolk()");
    }
  }
  private Yolk y;
  public Egg() {
    System.out.println("New Egg()");
    y = new Yolk();
  }
}

public class BigEgg extends Egg {
  public class Yolk {
    public Yolk() {
```

```
      System.out.println("BigEgg.Yolk()");
    }
  }
  public static void main(String[] args) {
    new BigEgg();
  }
} ///:~
```

Le constructeur par défaut est généré automatiquement par le compilateur, et il appelle le constructeur par défaut de la classe de base. On pourrait penser que puisqu'on crée un **BigEgg**, la version « redéfinie » de **Yolk** sera utilisée, mais ce n'est pas le cas. La sortie produite est :

```
New Egg()
Egg.Yolk()
```

Cet exemple montre simplement qu'il n'y a aucune magie spéciale associée aux classes internes quand on hérite d'une classe externe. Les deux classes internes sont des entités complètement séparées, chacune dans leur propre espace de noms. Cependant, il est toujours possible de dériver explicitement la classe interne :

```
//: c08:BigEgg2.java
// Dérivation d'une classe interne.

class Egg2 {
  protected class Yolk {
    public Yolk() {
      System.out.println("Egg2.Yolk()");
    }
    public void f() {
      System.out.println("Egg2.Yolk.f()");
    }
  }
  private Yolk y = new Yolk();
  public Egg2() {
    System.out.println("New Egg2()");
  }
  public void insertYolk(Yolk yy) { y = yy; }
  public void g() { y.f(); }
}

public class BigEgg2 extends Egg2 {
  public class Yolk extends Egg2.Yolk {
    public Yolk() {
      System.out.println("BigEgg2.Yolk()");
    }
    public void f() {
      System.out.println("BigEgg2.Yolk.f()");
    }
```

```
  }
  public BigEgg2() { insertYolk(new Yolk()); }
  public static void main(String[] args) {
    Egg2 e2 = new BigEgg2();
    e2.g();
  }
} ///:~
```

Maintenant **BiggEgg2.Yolk** étend explicitement **Egg2.Yolk** et redéfinit ses méthodes. La méthode **insertYolk()** permet à **BiggEgg2** de transtyper un de ses propres objets **Yolk** dans la référence **y** de **Egg2**, donc quand **g()** appelle **y.f()**, la version redéfinie de **f()** est utilisée. La sortie du programme est :

```
Egg2.Yolk()
New Egg2()
Egg2.Yolk()
BigEgg2.Yolk()
BigEgg2.Yolk.f()
```

Le second appel à **Egg2.Yolk()** est l'appel du constructeur de la classe de base depuis le constructeur de **BigEgg2.Yolk**. On peut voir que la version redéfinie de **f()** est utilisée lorsque **g()** est appelée.

Identifiants des classes internes

Puisque chaque classe produit un fichier **.class** qui contient toutes les informations concernant la création d'objets de ce type (ces informations produisent une « méta-classe » dans un objet **Class**), il est aisé de deviner que les classes internes produisent aussi des fichiers **.class** qui contiennent des informations pour *leurs* objets **Class**. La nomenclature de ces fichiers / classes est stricte : le nom de la classe externe suivie par un **$**, suivi du nom de la classe interne. Par exemple, les fichiers **.class** créés par **InheritInner.java** incluent :

```
InheritInner.class
WithInner$Inner.class
WithInner.class
```

Si les classes internes sont anonymes, le compilateur génère simplement des nombres comme identifiants de classe interne. Si des classes internes sont imbriquées dans d'autres classes internes, leur nom est simplement ajouté après un **$** et le nom des identifiants des classes externes.

Bien que cette gestion interne des noms soit simple et directe, elle est robuste et gère la plupart des situations [42]. Et comme cette notation est la notation standard pour Java, les fichiers générés sont automatiquement indépendants de la plateforme (Notez que le compilateur Java modifie les classes internes d'un tas d'autres manières afin de les faire fonctionner).

Raison d'être des classes internes

Jusqu'à présent, on a vu la syntaxe et la sémantique décrivant la façon dont les classes internes fonctionnent, mais cela ne répond pas à la question du pourquoi de leur existence. Pourquoi Sun s'est-il donné tant de mal pour ajouter au langage cette fonctionnalité fondamentale ?

Typiquement, la classe interne hérite d'une classe ou implémente une **interface**, et le code de la classe interne manipule l'objet de la classe externe l'ayant créé. On peut donc dire qu'une classe interne est une sorte de fenêtre dans la classe externe.

Mais si on a juste besoin d'une référence sur une **interface**, pourquoi ne pas implémenter cette **interface** directement dans la classe externe ? La réponse à cette question allant au coeur des classes internes est simple : « Si c'est tout ce dont on a besoin, alors c'est ainsi qu'il faut procéder ». Alors qu'est-ce qui distingue une classe interne implémentant une **interface** d'une classe externe implémentant cette même **interface** ? C'est tout simplement qu'on ne dispose pas toujours des facilités fournies par les **interface**s - quelquefois on est obligé de travailler avec des implémentations. Voici donc la raison principale d'utiliser des classes internes :

Chaque classe interne peut hériter indépendamment d'une implémentation. La classe interne n'est pas limitée par le fait que la classe externe hérite déjà d'une implémentation.

Sans cette capacité que fournissent les classes internes d'hériter - dans la pratique - de plus d'une classe concrète ou **abstract**, certaines conceptions ou problèmes seraient impossibles à résoudre. Les classes internes peuvent donc être considérées comme la suite de la solution au problème de l'héritage multiple. Les interfaces résolvent une partie du problème, mais les classes internes permettent réellement « l'héritage multiple d'implémentations ». Les classes internes permettent effectivement de dériver plusieurs non **interface**s.

Pour voir ceci plus en détails, imaginons une situation dans laquelle une classe doit implémenter deux interfaces. Du fait de la flexibilité des interfaces, on a le choix entre avoir une classe unique ou s'aider d'une classe interne :

```
//: c08:MultiInterfaces.java
// Deux façons pour une classe
// d'implémenter des interfaces multiples.

interface A {}
interface B {}

class X implements A, B {}

class Y implements A {
  B makeB() {
    // Classe interne anonyme :
    return new B() {};
  }
}

public class MultiInterfaces {
  static void takesA(A a) {}
  static void takesB(B b) {}
  public static void main(String[] args) {
    X x = new X();
    Y y = new Y();
    takesA(x);
    takesA(y);
```

```
    takesB(x);
    takesB(y.makeB());
  }
} ///:~
```

Bien sûr, la structure du code peut impliquer une logique pouvant imposer l'une ou l'autre des solutions. La nature du problème fournit généralement aussi des indices pour choisir entre une classe unique ou une classe interne. Mais en l'absence d'aucune autre contrainte, l'approche choisie dans l'exemple précédent ne fait aucune différence du point de vue implémentation. Les deux fonctionnent.

Cependant, si on a des classes **abstract** ou concrètes à la place des **interface**s, on est obligé de recourir aux classes internes si la classe doit implémenter les deux :

Si le problème de « l'héritage multiple d'implémentations » ne se pose pas, on peut tout à fait se passer des classes internes. Mais les classes internes fournissent toutefois des fonctionnalités intéressantes :

1. Les classes internes peuvent avoir plusieurs instances, chacune avec ses propres informations indépendantes des informations de l'objet de la classe externe.
2. Dans une classe externe on peut avoir plusieurs classes internes, chacune implémentant la même **interface** ou dérivant la même classe d'une façon différente. Nous allons en voir un exemple bientôt.
3. Le point de création d'un objet de la classe interne n'est pas lié à la création de l'objet de la classe externe.
4. Aucune confusion concernant la relation « est-un » n'est possible avec la classe interne ; c'est une entité séparée.

Par exemple, si **Sequence.java** n'utilisait pas de classes internes, il aurait fallu dire « une **Sequence** est un **Selector** », et on n'aurait pu avoir qu'un seul **Selector** pour une **Sequence** particulière. De plus, on peut avoir une autre méthode, **getRSelector()**, qui produise un **Selector** parcourant la **Sequence** dans l'ordre inverse. Cette flexibilité n'est possible qu'avec les classes internes.

Fermetures & callbacks

Une *fermeture* est un objet qui retient des informations de la portée dans laquelle il a été créé. A partir de cette définition, il est clair qu'une classe interne est une fermeture orientée objet, parce qu'elle ne contient pas seulement chaque élément d'information de l'objet de la classe externe (« la portée dans laquelle il a été créé »), mais elle contient aussi automatiquement une référence sur l'objet de la classe externe, avec la permission d'en manipuler tous les membres, y compris les **private**.

L'un des arguments les plus percutants mis en avant pour inclure certains mécanismes de pointeur dans Java était de permettre les *callbacks*. Avec un callback, on donne des informations à un objet lui permettant de revenir plus tard dans l'objet originel. Ceci est un concept particulièrement puissant, comme nous le verrons dans les chapitres 13 et 16. Cependant, si les callbacks étaient implémentés avec des pointeurs, le programmeur serait responsable de la gestion de ce pointeur et devrait faire attention afin de ne pas l'utiliser de manière incontrôlée. Mais comme on l'a déjà vu, Java n'aime pas ce genre de solutions reposant sur le programmeur, et les pointeurs ne furent pas inclus dans le langage.

Les classes internes fournissent une solution parfaite pour les fermetures, bien plus flexible et de loin plus sûre qu'un pointeur. Voici un exemple simple :

```
//: c08:Callbacks.java
// Utilisation des classes internes pour les callbacks

interface Incrementable {
  void increment();
}

// Il est très facile d'implémenter juste l'interface :
class Callee1 implements Incrementable {
  private int i = 0;
  public void increment() {
    i++;
    System.out.println(i);
  }
}

class MyIncrement {
  public void increment() {
    System.out.println("Other operation");
  }
  public static void f(MyIncrement mi) {
    mi.increment();
  }
}

// Si la classe doit aussi implémenter increment() d'une
// autre façon, il faut utiliser une classe interne :
class Callee2 extends MyIncrement {
  private int i = 0;
  private void incr() {
    i++;
    System.out.println(i);
  }
  private class Closure implements Incrementable {

    public void increment() { incr(); }
  }
  Incrementable getCallbackReference() {
    return new Closure();
  }
}

class Caller {
  private Incrementable callbackReference;
  Caller(Incrementable cbh) {
```

```
    callbackReference = cbh;
  }
  void go() {
    callbackReference.increment();
  }
}

public class Callbacks {
  public static void main(String[] args) {
    Callee1 c1 = new Callee1();
    Callee2 c2 = new Callee2();
    MyIncrement.f(c2);
    Caller caller1 = new Caller(c1);
    Caller caller2 =
      new Caller(c2.getCallbackReference());
    caller1.go();
    caller1.go();
    caller2.go();
    caller2.go();
  }
} ///:~
```

Cet exemple est un exemple supplémentaire montrant les différences entre l'implémentation d'une interface dans une classe externe ou une classe interne. **Callee1** est sans conteste la solution la plus simple en terme de code. **Callee2** hérite de **MyIncrement** qui dispose déjà d'une méthode **increment()** faisant quelque chose de complètement différent que ce qui est attendu par l'interface **Incrementable**. Quand **MyIncrement** est dérivée dans **Callee2**, **increment()** ne peut être redéfinie pour être utilisée par **Incrementable**, on est donc forcé d'utiliser une implémentation séparée avec une classe interne. Notez également que lorsqu'on crée une classe interne, on n'étend pas ni ne modifie l'interface de la classe externe.

Remarquez bien que tout dans **Callee2** à l'exception de **getCallbackReference()** est **private**. L'**interface Incrementable** est essentielle pour permettre *toute* interaction avec le monde extérieur. Les **interface**s permettent donc une séparation complète entre l'interface et l'implémentation.

La classe interne **Closure** implémente **Incrementable** uniquement pour fournir un point de retour dans **Callee2** - mais un point de retour sûr. Quiconque récupère la référence sur **Incrementable** ne peut appeler qu'**increment()** (contrairement à un pointeur, qui aurait permis de faire tout ce qu'on veut).

Caller prend une référence **Incrementable** dans son constructeur (bien qu'on puisse fournir cette référence - ce callback - n'importe quand), et s'en sert par la suite, parfois bien plus tard, pour « revenir » dans la classe **Callee**.

La valeur des callbacks réside dans leur flexibilité - on peut décider dynamiquement quelles fonctions vont être appelées lors de l'exécution. Les avantages des callbacks apparaîtront dans le chapitre 13, où ils sont utilisés immodérément pour implémenter les interfaces graphiques utilisateurs (GUI).

Classes internes & structures de contrôle

Un exemple plus concret d'utilisation des classes internes est ce que j'appelle les *structures de contrôle*.

Une *structure d'application* est une classe ou un ensemble de classes conçues pour résoudre un type particulier de problème. Pour utiliser une structure d'application, il suffit de dériver une ou plusieurs de ces classes et de redéfinir certaines des méthodes. Le code écrit dans les méthodes redéfinies particularise la solution générale fournie par la structure d'application, afin de résoudre le problème considéré. Les structures de contrôle sont un type particulier des structures d'application dominées par la nécessité de répondre à des événements ; un système qui répond à des événements est appelé un *système à programmation événementielle*. L'un des problèmes les plus ardus en programmation est l'interface graphique utilisateur (GUI), qui est quasiment entièrement événementielle. Comme nous le verrons dans le Chapitre 13, la bibliothèque Java Swing est une structure de contrôle qui résoud élégamment le problème des interfaces utilisateurs en utilisant extensivement les classes internes.

Pour voir comment les classes internes permettent une mise en oeuvre aisée des structures de contrôle, considérons le cas d'une structure de contrôle dont le rôle consiste à exécuter des événements dès lors que ces événements sont « prêts ». Bien que « prêt » puisse vouloir dire n'importe quoi, dans notre cas nous allons nous baser sur un temps d'horloge. Ce qui suit est une structure de contrôle qui ne contient aucune information spécifique sur ce qu'elle contrôle. Voici tout d'abord l'interface qui décrit tout événement. C'est une classe **abstract** plutôt qu'une **interface** parce que le comportement par défaut est de réaliser le contrôle sur le temps, donc une partie de l'implémentation peut y être incluse :

```
//: c08:controller:Event.java
// Les méthodes communes pour n'importe quel événement.
package c08.controller;

abstract public class Event {
  private long evtTime;
  public Event(long eventTime) {
    evtTime = eventTime;
  }
  public boolean ready() {
    return System.currentTimeMillis() >= evtTime;
  }
  abstract public void action();
  abstract public String description();
} ///:~
```

Le constructeur stocke simplement l'heure à laquelle on veut que l'**Event** soit exécuté, tandis que **ready()** indique si c'est le moment de le lancer. Bien sûr, **ready()** peut être redéfini dans une classe dérivée pour baser les **Event** sur autre chose que le temps.

action() est la méthode appelée lorsque l'**Event** est **ready()**, et **description()** donne des informations (du texte) à propos de l'**Event**.

Le fichier suivant contient la structure de contrôle proprement dite qui gère et déclenche les événements. La première classe est simplement une classe « d'aide » dont le rôle consiste à stocker

des objets **Event**. On peut la remplacer avec n'importe quel conteneur plus approprié, et dans le Chapitre 9 nous verrons d'autres conteneurs qui ne requerront pas ce code supplémentaire :

```
//: c08:controller:Controller.java
// Avec Event, la structure générique
// pour tous les systèmes de contrôle :
package c08.controller;

// Ceci est jsute une manière de stocker les objets Event.
class EventSet {
  private Event[] events = new Event[100];
  private int index = 0;
  private int next = 0;
  public void add(Event e) {
    if(index >= events.length)
      return; // (Normalement, générer une exception)
    events[index++] = e;
  }
  public Event getNext() {
    boolean looped = false;
    int start = next;
    do {
      next = (next + 1) % events.length;
      // Vérifie si on a fait le tour :
      if(start == next) looped = true;
      // Si on a fait le tour, c'est que la
      // liste est vide :
      if((next == (start + 1) % events.length)
         && looped)
        return null;
    } while(events[next] == null);
    return events[next];
  }
  public void removeCurrent() {
    events[next] = null;
  }
}

public class Controller {
  private EventSet es = new EventSet();
  public void addEvent(Event c) { es.add(c); }
  public void run() {
    Event e;
    while((e = es.getNext()) != null) {
      if(e.ready()) {
        e.action();
        System.out.println(e.description());
        es.removeCurrent();
```

```
        }
      }
    }
  } ///:~
```

EventSet stocke arbitrairement 100 **Event**s (si un « vrai » conteneur du Chapitre 9 était utilisé ici, on n'aurait pas à se soucier à propos de sa taille maximum, puisqu'il se redimensionnerait de lui-même). L'**index** est utilisé lorsqu'on veut récupérer le prochain **Event** de la liste, pour voir si on a fait le tour. Ceci est important pendant un appel à **getNext()**, parce que les objets **Event** sont enlevés de la liste (avec **removeCurrent()**) une fois exécutés, donc **getNext()** rencontrera des trous dans la liste lorsqu'il la parcourra.

Notez que **removeCurrent()** ne positionne pas simplement un flag indiquant que l'objet n'est plus utilisé. A la place, il positionne la référence à **null**. C'est important car si le ramasse-miettes rencontre une référence qui est encore utilisée il ne pourra pas nettoyer l'objet correspondant. Si l'objet n'a plus de raison d'être (comme c'est le cas ici), il faut alors mettre leur référence à **null** afin d'autoriser le ramasse-miettes à les nettoyer.

C'est dans **Controller** que tout le travail est effectué. Il utiliser un **EventSet** pour stocker ses objets **Event**, et **addEvent()** permet d'ajouter de nouveaux éléments à la liste. Mais la méthode principale est **run()**. Cette méthode parcourt l'**EventSet**, recherchant un objet **Event** qui soit **ready()**. Il appelle alors la méthode **action()** pour cet objet, affiche sa **description()** et supprime l'**Event** de la liste.

Notez que jusqu'à présent dans la conception on ne sait rien sur *ce que fait* exactement un **Event**. Et c'est le point fondamental de la conception : comment elle « sépare les choses qui changent des choses qui ne bougent pas ». Ou, comme je l'appelle, le « vecteur de changement » est constitué des différentes actions des différents types d'objets **Event**, actions différentes réalisées en créant différentes sous-classes d'**Event**.

C'est là que les classes internes interviennent. Elles permettent deux choses :

1. Réaliser l'implémentation complète d'une application de structure de contrôle dans une seule classe, encapsulant du même coup tout ce qui est unique dans cette implémentation. Les classes internes sont utilisées pour décrire les différents types d'**action()** nécessaires pour résoudre le problème. De plus, l'exemple suivant utilise des classes internes **private** afin que l'implémentation soit complètement cachée et puisse être changée en toute impunité.

2. Empêcher que l'implémentation ne devienne trop lourde, puisqu'on est capable d'accéder facilement à chacun des membres de la classe externe. Sans cette facilité, le code deviendrait rapidement tellement confus qu'il faudrait chercher une autre solution.

Considérons une implémentation particulière de la structure de contrôle conçue pour contrôler les fonctions d'une serre [43]. Chaque action est complètement différente : contrôler les lumières, l'arrosage et la température, faire retentir des sonneries et relancer le système. Mais la structure de contrôle est conçue pour isoler facilement ce code différent. Les classes internes permettent d'avoir de multiples versions dérivées de la même classe de base (ici, **Event**) à l'intérieur d'une seule et même classe. Pour chaque type d'action on crée une nouvelle classe interne dérivée d'**Event**, et on écrit le code de contrôle dans la méthode **action()**.

Typiquement, la classe **GreenhouseControls** hérite de **Controller** :

```
//: c08:GreenhouseControls.java
// Ceci est une application spécifique du
// système de contrôle, le tout dans une seule classe.
// Les classes internes permettent d'encapsuler des
// fonctionnalités différentes pour chaque type d'Event.
import c08.controller.*;

public class GreenhouseControls
    extends Controller {
  private boolean light = false;
  private boolean water = false;
  private String thermostat = "Day";
  private class LightOn extends Event {
    public LightOn(long eventTime) {
      super(eventTime);
    }
    public void action() {
      // Placer ici du code de contrôle hardware pour
      // réellement allumer la lumière.
      light = true;
    }
    public String description() {
      return "Light is on";
    }
  }
  private class LightOff extends Event {
    public LightOff(long eventTime) {
      super(eventTime);
    }
    public void action() {
      // Put hardware control code here to
      // physically turn off the light.
      light = false;
    }
    public String description() {
      return "Light is off";
    }
  }
  private class WaterOn extends Event {
    public WaterOn(long eventTime) {
      super(eventTime);
    }
    public void action() {
      // Placer ici du code de contrôle hardware.
      water = true;
    }
    public String description() {
```

```
      return "Greenhouse water is on";
    }
  }
  private class WaterOff extends Event {
    public WaterOff(long eventTime) {
      super(eventTime);
    }
    public void action() {
      // Placer ici du code de contrôle hardware.
      water = false;
    }
    public String description() {
      return "Greenhouse water is off";
    }
  }
  private class ThermostatNight extends Event {
    public ThermostatNight(long eventTime) {
      super(eventTime);
    }
    public void action() {
      // Placer ici du code de contrôle hardware.
      thermostat = "Night";
    }
    public String description() {
      return "Thermostat on night setting";
    }
  }
  private class ThermostatDay extends Event {
    public ThermostatDay(long eventTime) {
      super(eventTime);
    }
    public void action() {
      // Placer ici du code de contrôle hardware.
      thermostat = "Day";
    }
    public String description() {
      return "Thermostat on day setting";
    }
  }
  // Un exemple d'une action() qui insère une nouvelle
  // instance de son type dans la liste d'Event :
  private int rings;
  private class Bell extends Event {
    public Bell(long eventTime) {
      super(eventTime);
    }
    public void action() {
```

```
        // Sonne toutes les 2 secondes, 'rings' fois :
        System.out.println("Bing!");
        if(--rings > 0)
          addEvent(new Bell(
            System.currentTimeMillis() + 2000));
      }
      public String description() {
        return "Ring bell";
      }
    }
    private class Restart extends Event {
      public Restart(long eventTime) {
        super(eventTime);
      }
      public void action() {
        long tm = System.currentTimeMillis();
        // Au lieu d'un codage en dur, on pourrait
        // récupérer les informations en parsant un
        // fichier de configuration :
        rings = 5;
        addEvent(new ThermostatNight(tm));
        addEvent(new LightOn(tm + 1000));
        addEvent(new LightOff(tm + 2000));
        addEvent(new WaterOn(tm + 3000));
        addEvent(new WaterOff(tm + 8000));
        addEvent(new Bell(tm + 9000));
        addEvent(new ThermostatDay(tm + 10000));
        // On peut même ajouter un objet Restart !
        addEvent(new Restart(tm + 20000));
      }
      public String description() {
        return "Restarting system";
      }
    }

  } public static void main(String[] args) {
    GreenhouseControls gc =
      new GreenhouseControls();
    long tm = System.currentTimeMillis();
    gc.addEvent(gc.new Restart(tm));
    gc.run();
  } //:~
```

La plupart des classes **Event** sont similaires, mais **Bell** et **Restart** sont spéciales. **Bell** sonne, et si elle n'a pas sonné un nombre suffisant de fois, elle ajoute un nouvel objet **Bell** à la liste des événements afin de sonner à nouveau plus tard. Notez comme les classes internes *semblent* bénéficier de l'héritage multiple : **Bell** possède toutes les méthodes d'**Event** mais elle semble disposer également de toutes les méthodes de la classe externe **GreenhouseControls**.

Restart est responsable de l'initialisation du système, il ajoute donc tous les événements appropriés. Bien sûr, une manière plus flexible de réaliser ceci serait d'éviter le codage en dur des événements et de les extraire d'un fichier à la place (c'est précisément ce qu'un exercice du Chapitre 11 demande de faire). Puisque **Restart()** n'est qu'un objet **Event** comme un autre, on peut aussi ajouter un objet **Restart** depuis **Restart.action()** afin que le système se relance de lui-même régulièrement. Et tout ce qu'on a besoin de faire dans **main()** est de créer un objet **GreenhouseControls** et ajouter un objet **Restart** pour lancer le processus.

Cet exemple devrait vous avoir convaincu de l'intérêt des classes internes, spécialement dans le cas des structures de contrôle. Si ce n'est pas le cas, dans le Chapitre 13, vous verrez comment les classes internes sont utilisées pour décrire élégamment les actions d'une interface graphique utilisateur. A la fin de ce chapitre vous devriez être complètement convaincu.

Résumé

Les interfaces et les classes internes sont des concepts plus sophistiqués que ce que vous pourrez trouver dans beaucoup de langages de programmation orientés objets. Par exemple, rien de comparable n'existe en C++. Ensemble, elles résolvent le même problème que celui que le C++ tente de résoudre avec les fonctionnalités de l'héritage multiple. Cependant, l'héritage multiple en C ++ se révèle relativement ardu à mettre en oeuvre, tandis que les interfaces et les classes internes en Java sont, en comparaison, d'un abord nettement plus facile.

Bien que les fonctionnalités en elles-mêmes soient relativement simples, leur utilisation relève de la conception, de même que le polymorphisme. Avec le temps, vous reconnaîtrez plus facilement les situations dans lesquelles utiliser une interface, ou une classe interne, ou les deux. Mais à ce point du livre vous devriez à tout le moins vous sentir à l'aise avec leur syntaxe et leur sémantique. Vous intègrerez ces techniques au fur et à mesure que vous les verrez utilisées.

Exercices

Les solutions d'exercices sélectionnés peuvent être trouvées dans le document électronique *The Thinking in Java Annotated Solution Guide*, disponible pour un faible coût sur *www.BruceEckel.com*.

1. Prouvez que les champs d'une **interface** sont implicitement **static** et **final**.
2. Créez une **interface** contenant trois méthodes, dans son propre **package**. Implémentez cette interface dans un **package** différent.
3. Prouvez que toutes les méthodes d'une **interface** sont automatiquement **public**.
4. Dans **c07:Sandwich.java**, créez une interface appelée **FastFood** (avec les méthodes appropriées) et changez **Sandwich** afin qu'il implémente **FastFood**.
5. Créez trois **interface**s, chacune avec deux méthodes. Créez une nouvelle **interface** héritant des trois, en ajoutant une nouvelle méthode. Créez une classe implémentant la nouvelle interface et héritant déjà d'une classe concrète. Ecrivez maintenant quatre méthodes, chacune d'entre elles prenant l'une des quatre **interface**s en argument. Dans **main()**, créez un objet de votre classe et passez-le à chacune des méthodes.
6. Modifiez l'exercice 5 en créant une classe **abstract** et en la dérivant dans la dernière classe.

7. Modifiez **Music5.java** en ajoutant une **interfacePlayable**. Enlevez la déclaration de **play()** d'**Instrument**. Ajoutez **Playable** aux classes dérivées en l'incluant dans la liste **implements**. Changez **tune()** afin qu'il accepte un **Playable** au lieu d'un **Instrument**.

8. Changez l'exercice 6 du Chapitre 7 afin que **Rodent** soit une **interface**.

9. Dans **Adventure.java**, ajoutez une **interface** appelée **CanClimb** respectant la forme des autres interfaces.

10. Ecrivez un programme qui importe et utilise **Month2.java**.

11. En suivant l'exemple donné dans **Month2.java**, créez une énumération des jours de la semaine.

12. Créez une **interface** dans son propre package contenant au moins une méthode. Créez une classe dans un package séparé. Ajoutez une classe interne **protected** qui implémente l'**interface**. Dans un troisième package, dérivez votre classe, et dans une méthode renvoyez un objet de la classe interne **protected**, en le transtypant en **interface** durant le retour.

13. Créez une **interface** contenant au moins une méthode, et implémentez cette **interface** en définissant une classe interne à l'intérieur d'une méthode, qui renvoie une référence sur votre **interface**.

14. Répétez l'exercice 13 mais définissez la classe interne à l'intérieur d'une portée à l'intérieur de la méthode.

15. Répétez l'exercice 13 en utilisant une classe interne anonyme.

16. Créez une classe interne **private** qui implémente une **interface public**. Ecrivez une méthode qui renvoie une référence sur une instance de la classe interne **private**, transtypée en **interface**. Montrez que la classe interne est complètement cachée en essayant de la transtyper à nouveau.

17. Créez une classe avec un constructeur autre que celui par défaut et sans constructeur par défaut. Créez une seconde classe disposant d'une méthode qui renvoie une référence à la première classe. Créez un objet à renvoyer en créant une classe interne anonyme dérivée de la première classe.

18. Créez une classe avec un champ **private** et une méthode **private**. Créez une classe interne avec une méthode qui modifie le champ de la classe externe et appelle la méthode de la classe externe. Dans une seconde méthode de la classe externe, créez un objet de la classe interne et appelez sa méthode ; montrez alors l'effet sur l'objet de la classe externe.

19. Répétez l'exercice 18 en utilisant une classe interne anonyme.

20. Créez une classe contenant une classe interne **static**. Dans **main()**, créez une instance de la classe interne.

21. Créez une **interface** contenant une classe interne **static**. Implémentez cette **interface** et créez une instance de la classe interne.

22. Créez une classe contenant une classe interne contenant elle-même une classe interne. Répétez ce schéma en utilisant des classes internes **static**. Notez les noms des fichiers **.class** produits par le compilateur.

23. Créez une classe avec une classe interne. Dans une classe séparée, créez une instance de la classe interne.

24. Créez une classe avec une classe interne disposant d'un constructeur autre que celui par défaut. Créez une seconde classe avec une classe interne héritant de la première classe interne.

25. Corrigez le problème dans **WindError.java**.

26. Modifiez **Sequence.java** en ajoutant une méthode **getRSelector()** qui produise une implémentation différente de l'**interface Selector** afin de parcourir la séquence en ordre inverse, de la fin vers le début.

27. Créez une **interface U** contenant trois méthodes. Créez une classe **A** avec une méthode qui produise une référence sur un **U** en construisant une classe interne anonyme. Créez une seconde classe **B** qui contienne un tableau de **U**. **B** doit avoir une méthode qui accepte et stocke une référence sur un **U** dans le tableau, une deuxième méthode qui positionne une référence (spécifiée par un argument de la méthode) dans le tableau à **null**, et une troisième méthode qui se déplace dans le tableau et appelle les méthodes de l'objet **U**. Dans **main()**, créez un groupe d'objets **A** et un objet **B**. Remplissez l'objet **B** avec les références **U** produites par les objets **A**. Utilisez **B** pour revenir dans les objets **A**. Enlevez certaines des références **U** de **B**.

28. Dans **GreenhouseControls.java**, ajoutez des classes internes **Event** qui contrôlent des ventilateurs.

29. Montrez qu'une classe interne peut accéder aux éléments **private** de sa classe externe. Déterminez si l'inverse est vrai.

[38]Cette approche m'a été inspirée par un e-mail de Rich Hoffarth.

[39]Merci à Martin Danner pour avoir posé cette question lors d'un séminaire.

[40]Ceci est très différent du concept des *classes imbriquées* en C++, qui est simplement un mécanisme de camouflage de noms. Il n'y a aucun lien avec l'objet externe et aucune permission implicite en C++.

[41]Merci encore à Martin Danner.

[42]Attention cependant, '$' est un méta-caractère pour les shells unix et vous pourrez parfois rencontrer des difficultés en listant les fichiers **.class**. Ceci peut paraître bizarre de la part de Sun, une entreprise résolument tournée vers unix. Je pense qu'ils n'ont pas pris en compte ce problème car ils pensaient que l'attention se porterait surtout sur les fichiers sources.

[43]Pour je ne sais quelle raison, ce problème m'a toujours semblé plaisant ; il vient de mon livre précédent *C++ Inside & Out*, mais Java offre une solution bien plus élégante.

Chapitre 9 - Stockage des objets

C'est un programme relativement simple que celui qui ne manipule que des objets dont le nombre et la durée de vie sont connus à l'avance.

Mais en général, vos programmes créeront de nouveaux objets basés sur des informations qui ne seront pas connues avant le lancement du programme. Le nombre voire même le type des objets nécessaires ne seront pas connus avant la phase d'exécution du programme. Pour résoudre le problème considéré, il faut donc être capable de créer un certain nombre d'objets, n'importe quand, n'importe où. Ce qui implique qu'on ne peut se contenter d'une référence nommée pour stocker chacun des objets du programme :

```
MyObject myReference;
```

puisqu'on ne connaît pas le nombre exact de références qui seront manipulées.

Pour résoudre ce problème fondamental, Java dispose de plusieurs manières de stocker les objets (ou plus exactement les références sur les objets). Le type interne est le tableau, dont nous avons déjà parlé auparavant. De plus, la bibliothèque des utilitaires de Java propose un ensemble relativement complet de *classes conteneurs* (aussi connues sous le nom de *classes collections*, mais comme les bibliothèques de Java 2 utilisent le nom **Collection** pour un sous-ensemble particulier de cette bibliothèque, j'utiliserai ici le terme plus générique « conteneur »). Les conteneurs fournissent des moyens sophistiqués pour stocker et même manipuler les objets d'un programme.

Les tableaux

Les tableaux ont déjà été introduits dans la dernière section du Chapitre 4, qui montrait comment définir et initialiser un tableau. Ce chapitre traite du stockage des objets, et un tableau n'est ni plus ni moins qu'un moyen de stocker des objets. Mais il existe de nombreuses autres manières de stocker des objets : qu'est-ce qui rend donc les tableaux si spécial ?

Les tableaux se distinguent des autres types de conteneurs sur deux points : l'efficacité et le type. Un tableau constitue la manière la plus efficace que propose Java pour stocker et accéder aléatoirement à une séquence d'objets (en fait, de références sur des objets). Un tableau est une simple séquence linéaire, ce qui rend l'accès aux éléments extrêmement rapide ; mais cette rapidité se paye : la taille d'un tableau est fixée lors de sa création et ne peut être plus être changée pendant toute la durée de sa vie. Une solution est de créer un tableau d'une taille donnée, et, lorsque celui-ci est saturé, en créer un nouveau et déplacer toutes les références de l'ancien tableau dans le nouveau. C'est précisément ce que fait la classe **ArrayList**, qui sera étudiée plus loin dans ce chapitre. Cependant, du fait du surcoût engendré par la flexibilité apportée au niveau de la taille, une **ArrayList** est beaucoup moins efficace qu'un tableau.

La classe conteneur **vector** en C++ *connaît* le type des objets qu'il stocke, mais il a un inconvénient comparé aux tableaux de Java : l'**opérateur []** des **vector** C++ ne réalise pas de contrôle sur les indices, on peut donc tenter d'accéder à un élément au-delà de la taille du **vector** [44]. En Java, un contrôle d'indices est automatiquement effectué, qu'on utilise un tableau ou un conteneur - une exception **RuntimeException** est générée si les frontières sont dépassées. Comme vous le verrez dans le Chapitre 10, ce type d'exception indique une erreur due au programmeur, et comme telle il ne faut pas la prendre en considération dans le code. Bien entendu, le **vector** C++ n'effectue pas de

vérifications à chaque accès pour des raisons d'efficacité - en Java, la vérification continuelle des frontières implique une dégradation des performances pour les tableaux comme pour les conteneurs.

Les autres classes de conteneurs génériques qui seront étudiés dans ce chapitre, les **List**s, les **Set**s et les **Map**s, traitent les objets comme s'ils n'avaient pas de type spécifique. C'est à dire qu'ils les traitent comme s'ils étaient des **Object**s, la classe de base de toutes les classes en Java. Ceci est très intéressant d'un certain point de vue : un seul conteneur est nécessaire pour stocker tous les objets Java (excepté les types scalaires - ils peuvent toutefois être stockés dans les conteneurs sous forme de constantes en utilisant les classes Java d'encapsulation des types primitifs, ou sous forme de valeurs modifiables en les encapsulant dans des classes personnelles). C'est le deuxième point où les tableaux se distinguent des conteneurs génériques : lorsqu'un tableau est créé, il faut spécifier le type d'objets qu'il est destiné à stocker. Ce qui implique qu'on va bénéficier d'un contrôle de type lors de la phase compilation, nous empêchant de stocker des objets d'un mauvais type ou de se tromper sur le type de l'objet qu'on extrait. Bien sûr, Java empêchera tout envoi de message inapproprié à un objet, soit lors de la compilation soit lors de l'exécution du programme. Aucune des deux approches n'est donc plus risquée que l'autre, mais c'est tout de même mieux si c'est le compilateur qui signale l'erreur, plus rapide à l'exécution et il y a moins de chances que l'utilisateur final ne soit surpris par une exception.

Du fait de l'efficacité et du contrôle de type, il est toujours préférable d'utiliser un tableau si c'est possible. Cependant, les tableaux peuvent se révéler trop restrictifs pour résoudre certains problèmes. Après un examen des tableaux, le reste de ce chapitre sera consacré aux classes conteneurs proposées par Java.

Les tableaux sont des objets

Indépendamment du type de tableau qu'on utilise, un identifiant de tableau est en fait une référence sur un vrai objet créé dans le segment. C'est l'objet qui stocke les références sur les autres objets, et il peut être créé soit implicitement grâce à la syntaxe d'initialisation de tableau, soit explicitement avec une expression **new**. Une partie de l'objet tableau (en fait, la seule méthode ou champ auquel on peut accéder) est le membre en lecture seule **length** qui indique combien d'éléments peuvent être stockés dans l'objet. La syntaxe « **[]** » est le seul autre accès disponible pour les objets tableaux.

L'exemple suivant montre les différentes façons d'initialiser un tableau, et comment les références sur un tableau peuvent être assignées à différents objets tableau. Il montre aussi que les tableaux d'objets et les tableaux de scalaires sont quasi identiques dans leur utilisation. La seule différence est qu'un tableau d'objets stocke des références, alors qu'un tableau de scalaires stocke les valeurs directement.

```
//: c09:ArraySize.java
// Initialisation & ré-assignation des tableaux.

class Weeble {} // Une petite créature mythique

public class ArraySize {
  public static void main(String[] args) {
    // Tableaux d'objets :
    Weeble[] a; // Référence Null.
    Weeble[] b = new Weeble[5]; // Références Null
```

```
    Weeble[] c = new Weeble[4];
    for(int i = 0; i < c.length; i++)
      c[i] = new Weeble();
    // Initialisation par agrégat :
    Weeble[] d = {
      new Weeble(), new Weeble(), new Weeble()
    };
    // Initialisation dynamique par agrégat :
    a = new Weeble[] {
      new Weeble(), new Weeble()
    };
    System.out.println("a.length=" + a.length);
    System.out.println("b.length = " + b.length);
    // Les références à l'intérieur du tableau sont
    // automatiquement initialisées à null :
    for(int i = 0; i < b.length; i++)
      System.out.println("b[" + i + "]=" + b[i]);
    System.out.println("c.length = " + c.length);
    System.out.println("d.length = " + d.length);
    a = d;
    System.out.println("a.length = " + a.length);

    // Tableaux de scalaires :
    int[] e; // Référence Null
    int[] f = new int[5];
    int[] g = new int[4];
    for(int i = 0; i < g.length; i++)
      g[i] = i*i;
    int[] h = { 11, 47, 93 };
    // Erreur de compilation : variable e non initialisée :
    //!System.out.println("e.length=" + e.length);
    System.out.println("f.length = " + f.length);
    // Les scalaires dans le tableau sont
    // automatiquement initialisées à zéro :
    for(int i = 0; i < f.length; i++)
      System.out.println("f[" + i + "]=" + f[i]);
    System.out.println("g.length = " + g.length);
    System.out.println("h.length = " + h.length);
    e = h;
    System.out.println("e.length = " + e.length);
    e = new int[] { 1, 2 };
    System.out.println("e.length = " + e.length);
  }
} ///:~
```

Voici la sortie du programme :

```
a.length = 2
```

```
b.length = 5
b[0]=null
b[1]=null
b[2]=null
b[3]=null
b[4]=null
c.length = 4
d.length = 3
a.length = 3
f.length = 5
f[0]=0
f[1]=0
f[2]=0
f[3]=0
f[4]=0
g.length = 4
h.length = 3
e.length = 3
e.length = 2
```

Le tableau **a** n'est initialement qu'une référence **null**, et le compilateur interdit d'utiliser cette référence tant qu'elle n'est pas correctement initialisée. Le tableau **b** est initialisé afin de pointer sur un tableau de références **Weeble**, même si aucun objet **Weeble** n'est réellement stocké dans le tableau. Cependant, on peut toujours s'enquérir de la taille du tableau, puisque **b** pointe sur un objet valide. Ceci montre un inconvénient des tableaux : on ne peut savoir combien d'éléments sont actuellement stockés *dans* le tableau, puisque **length** renvoie seulement le nombre d'éléments qu'on *peut* stocker dans le tableau, autrement dit la taille de l'objet tableau, et non le nombre d'éléments qu'il contient réellement. Cependant, quand un objet tableau est créé, ses références sont automatiquement initialisées à **null**, on peut donc facilement savoir si une cellule du tableau contient un objet ou pas en testant si son contenu est **null**. De même, un tableau de scalaires est automatiquement initialisé à zéro pour les types numériques, **(char)0** pour les caractères et **false** pour les **boolean**s.

Le tableau **c** montre la création d'un objet tableau suivi par l'assignation d'un objet **Weeble** à chacune des cellules du tableau. Le tableau **d** illustre la syntaxe d'« initialisation par agrégat » qui permet de créer un objet tableau (implicitement sur le segment avec **new**, comme le tableau **c**) *et* de l'initialiser avec des objets **Weeble**, le tout dans une seule instruction.

L'initialisation de tableau suivante peut être qualifiée d'« initialisation dynamique par agrégat ». L'initialisation par agrégat utilisée par **d** doit être utilisée lors de la définition de **d**, mais avec la seconde syntaxe il est possible de créer et d'initialiser un objet tableau n'importe où. Par exemple, supposons que **hide()** soit une méthode qui accepte un tableau d'objets **Weeble** comme argument. On peut l'appeler via :

```
hide(d);
```

mais on peut aussi créer dynamiquement le tableau qu'on veut passer comme argument :

```
hide(new Weeble[] { new Weeble(), new Weeble() });
```

Cette nouvelle syntaxe est bien plus pratique pour certaines parties de code.

L'expression :

```
a = d;
```

montre comment prendre une référence attachée à un tableau d'objets et l'assigner à un autre objet tableau, de la même manière qu'avec n'importe quel type de référence. Maintenant **a** et **d** pointent sur le même tableau d'objets dans le segment.

La seconde partie de **ArraySize.java** montre que les tableaux de scalaires fonctionnent de la même manière que les tableaux d'objets *sauf que* les tableaux de scalaires stockent directement les valeurs des scalaires.

Conteneurs de scalaires

Les classes conteneurs ne peuvent stocker que des références sur des objets. Un tableau, par contre, peut stocker directement des scalaires aussi bien que des références sur des objets. Il *est* possible d'utiliser des classes d'« encapsulation » telles qu'**Integer**, **Double**, etc. pour stocker des valeurs scalaires dans un conteneur, mais les classes d'encapsulation pour les types primitifs se révèlent souvent lourdes à utiliser. De plus, il est bien plus efficace de créer et d'accéder à un tableau de scalaires qu'à un conteneur de scalaires encapsulés.

Bien sûr, si on utilise un type primitif et qu'on a besoin de la flexibilité d'un conteneur qui ajuste sa taille automatiquement, le tableau ne convient plus et il faut se rabattre sur un conteneur de scalaires encapsulés. On pourrait se dire qu'il serait bon d'avoir un type **ArrayList** spécialisé pour chacun des types de base, mais ils n'existent pas dans Java. Un mécanisme de patrons permettra sans doute un jour à Java de mieux gérer ce problème [45].

Renvoyer un tableau

Supposons qu'on veuille écrire une méthode qui ne renvoie pas une seule chose, mais tout un ensemble de choses. Ce n'est pas facile à réaliser dans des langages tels que C ou C++ puisqu'ils ne permettent pas de renvoyer un tableau, mais seulement un pointeur sur un tableau. Cela ouvre la porte à de nombreux problèmes du fait qu'il devient ardu de contrôler la durée de vie du tableau, ce qui mène très rapidement à des fuites de mémoire.

Java utilise une approche similaire, mais permet de « renvoyer un tableau ». Bien sûr, il s'agit en fait d'une référence sur un tableau, mais Java assume de manière transparente la responsabilité de ce tableau - il sera disponible tant qu'on en aura besoin, et le ramasse-miettes le nettoiera lorsqu'on en aura fini avec lui.

Voici un exemple retournant un tableau de **String** :

```
//: c09:IceCream.java
// Renvoyer un tableau depuis des méthodes.

public class IceCream {
  static String[] flav = {
    "Chocolate", "Strawberry",
    "Vanilla Fudge Swirl", "Mint Chip",
    "Mocha Almond Fudge", "Rum Raisin",
    "Praline Cream", "Mud Pie"
```

```
  };
  static String[] flavorSet(int n) {
    // Force l'argument à être positif & à l'intérieur des indices :
    n = Math.abs(n) % (flav.length + 1);
    String[] results = new String[n];
    boolean[] picked =
      new boolean[flav.length];
    for (int i = 0; i < n; i++) {
      int t;
      do
        t = (int)(Math.random() * flav.length);
      while (picked[t]);
      results[i] = flav[t];
      picked[t] = true;
    }
    return results;
  }
  public static
   void main(String[] args) {
    for(int i = 0; i < 20; i++) {
      System.out.println(
        "flavorSet(" + i + ") = ");
      String[] fl = flavorSet(flav.length);
      for(int j = 0; j < fl.length; j++)
        System.out.println("\t" + fl[j]);
    }
  }
} ///:~
```

La méthode **flavorSet()** crée un tableau de **String**s de taille **n** (déterminé par l'argument de la méthode) appelé **results**. Elle choisit alors au hasard des parfums dans le tableau **flav** et les place dans **results**, qu'elle renvoie quand elle en a terminé. Renvoyer un tableau s'apparente à renvoyer n'importe quel autre objet - ce n'est qu'une référence. Le fait que le tableau ait été créé dans **flavorSet()** n'est pas important, il aurait pu être créé n'importe où. Le ramasse-miettes s'occupe de nettoyer le tableau quand on en a fini avec lui, mais le tableau existera tant qu'on en aura besoin.

Notez en passant que quand **flavorSet()** choisit des parfums au hasard, elle s'assure que le parfum n'a pas déjà été choisi auparavant. Ceci est réalisé par une boucle **do** qui continue de tirer un parfum au sort jusqu'à ce qu'elle en trouve un qui ne soit pas dans le tableau **picked** (bien sûr, on aurait pu utiliser une comparaison sur **String** avec les éléments du tableau **results**, mais les comparaisons sur **String** ne sont pas efficaces). Une fois le parfum sélectionné, elle l'ajoute dans le tableau et trouve le parfum suivant (**i** est alors incrémenté).

main() affiche 20 ensembles de parfums, et on peut voir que **flavorSet()** choisit les parfums dans un ordre aléatoire à chaque fois. Il est plus facile de s'en rendre compte si on redirige la sortie dans un fichier. Et lorsque vous examinerez ce fichier, rappelez-vous que vous *voulez* juste la glace, vous n'en avez pas *besoin*.

La classe Arrays

java.util contient la classe **Arrays**, qui propose un ensemble de méthodes **static** réalisant des opérations utiles sur les tableaux. Elle dispose de quatre fonctions de base : **equals()**, qui compare deux tableaux ; **fill()**, pour remplir un tableau avec une valeur ; **sort()**, pour trier un tableau ; et **binarySearch()**, pour trouver un élément dans un tableau trié. Toutes ces méthodes sont surchargées pour tous les types scalaires et les **Object**s. De plus, il existe une méthode **asList()** qui transforme un tableau en un conteneur **List** - que nous rencontrerons plus tard dans ce chapitre.

Bien que pratique, la classe **Arrays** montre vite ses limites. Par exemple, il serait agréable de pouvoir facilement afficher les éléments d'un tableau sans avoir à coder une boucle **for** à chaque fois. Et comme nous allons le voir, la méthode **fill()** n'accepte qu'une seule valeur pour remplir le tableau, ce qui la rend inutile si on voulait - par exemple - remplir le tableau avec des nombres aléatoires.

Nous allons donc compléter la classe **Arrays** avec d'autres utilitaires, qui seront placés dans le **package com.bruceeckel.util**. Ces utilitaires permettront d'afficher un tableau de n'importe quel type, et de remplir un tableau avec des valeurs ou des objets créés par un objet appelé *générateur* qu'il est possible de définir.

Du fait qu'il faille écrire du code pour chaque type scalaire de base aussi bien que pour la classe **Object**, une grande majorité de ce code est dupliqué [46]. Ainsi, par exemple une interface « générateur » est requise pour chaque type parce que le type renvoyé par **next()** doit être différent dans chaque cas :

```
//: com:bruceeckel:util:Generator.java
package com.bruceeckel.util;
public interface Generator {
  Object next();
} ///:~

//: com:bruceeckel:util:BooleanGenerator.java
package com.bruceeckel.util;
public interface BooleanGenerator {
  boolean next();
} ///:~

//: com:bruceeckel:util:ByteGenerator.java
package com.bruceeckel.util;
public interface ByteGenerator {
  byte next();
} ///:~

//: com:bruceeckel:util:CharGenerator.java
package com.bruceeckel.util;
public interface CharGenerator {
  char next();
} ///:~

//: com:bruceeckel:util:ShortGenerator.java
```

```
package com.bruceeckel.util;
public interface ShortGenerator {
  short next();
} ///:~

//: com:bruceeckel:util:IntGenerator.java
package com.bruceeckel.util;
public interface IntGenerator {
  int next();
} ///:~

//: com:bruceeckel:util:LongGenerator.java
package com.bruceeckel.util;
public interface LongGenerator {
  long next();
} ///:~

//: com:bruceeckel:util:FloatGenerator.java
package com.bruceeckel.util;
public interface FloatGenerator {
  float next();
} ///:~

//: com:bruceeckel:util:DoubleGenerator.java
package com.bruceeckel.util;
public interface DoubleGenerator {
  double next();
} ///:~
```

```
//: com:bruceeckel:util:Arrays2.java
// Un complément à java.util.Arrays, pour fournir
// de nouvelles fonctionnalités utiles lorsqu'on
// travaille avec des tableaux. Permet d'afficher
// n'importe quel tableau, et de le remplir via un
// objet « générateur » personnalisable.
package com.bruceeckel.util;
import java.util.*;

public class Arrays2 {
  private static void
  start(int from, int to, int length) {
    if(from != 0 || to != length)
      System.out.print("["+ from +":"+ to +"] ");
    System.out.print("(");
  }
  private static void end() {
```

```
    System.out.println(")");
  }
  public static void print(Object[] a) {
    print(a, 0, a.length);
  }
  public static void
  print(String msg, Object[] a) {
    System.out.print(msg + " ");
    print(a, 0, a.length);
  }
  public static void
  print(Object[] a, int from, int to){
    start(from, to, a.length);
    for(int i = from; i < to; i++) {
      System.out.print(a[i]);
      if(i < to -1)
        System.out.print(", ");
    }
    end();
  }
  public static void print(boolean[] a) {
      print(a, 0, a.length);
  }
  public static void
  print(String msg, boolean[] a) {
    System.out.print(msg + " ");
    print(a, 0, a.length);
  }
  public static void
  print(boolean[] a, int from, int to) {
    start(from, to, a.length);
    for(int i = from; i < to; i++) {
      System.out.print(a[i]);
      if(i < to -1)
        System.out.print(", ");
    }
    end();
  }
  public static void print(byte[] a) {
      print(a, 0, a.length);
  }
  public static void
  print(String msg, byte[] a) {
    System.out.print(msg + " ");
    print(a, 0, a.length);
  }
  public static void
```

```
print(byte[] a, int from, int to) {
  start(from, to, a.length);
  for(int i = from; i < to; i++) {
    System.out.print(a[i]);
    if(i < to -1)
      System.out.print(", ");
  }
  end();
}
public static void print(char[] a) {
    print(a, 0, a.length);
}
public static void
print(String msg, char[] a) {
  System.out.print(msg + " ");
  print(a, 0, a.length);
}
public static void
print(char[] a, int from, int to) {
  start(from, to, a.length);
  for(int i = from; i < to; i++) {
    System.out.print(a[i]);
    if(i < to -1)
      System.out.print(", ");
  }
  end();
}
public static void print(short[] a) {
    print(a, 0, a.length);
}
public static void
print(String msg, short[] a) {
  System.out.print(msg + " ");
  print(a, 0, a.length);
}
public static void
print(short[] a, int from, int to) {
  start(from, to, a.length);
  for(int i = from; i < to; i++) {
    System.out.print(a[i]);
    if(i < to - 1)
      System.out.print(", ");
  }
  end();
}
public static void print(int[] a) {
    print(a, 0, a.length);
```

```
  }
  public static void
  print(String msg, int[] a) {
    System.out.print(msg + " ");
    print(a, 0, a.length);
  }
  public static void
  print(int[] a, int from, int to) {
    start(from, to, a.length);
    for(int i = from; i < to; i++) {
      System.out.print(a[i]);
      if(i < to - 1)
        System.out.print(", ");
    }
    end();
  }
  public static void print(long[] a) {
    print(a, 0, a.length);
  }
  public static void
  print(String msg, long[] a) {
    System.out.print(msg + " ");
    print(a, 0, a.length);
  }
  public static void
  print(long[] a, int from, int to) {
    start(from, to, a.length);
    for(int i = from; i < to; i++) {
      System.out.print(a[i]);
      if(i < to - 1)
        System.out.print(", ");
    }
    end();
  }
  public static void print(float[] a) {
      print(a, 0, a.length);
  }
  public static void
  print(String msg, float[] a) {
    System.out.print(msg + " ");
    print(a, 0, a.length);
  }
  public static void
  print(float[] a, int from, int to) {
    start(from, to, a.length);
    for(int i = from; i < to; i++) {
      System.out.print(a[i]);
```

```
      if(i < to - 1)
        System.out.print(", ");
    }
    end();
  }
  public static void print(double[] a) {
    print(a, 0, a.length);
  }
  public static void
  print(String msg, double[] a) {
    System.out.print(msg + " ");
    print(a, 0, a.length);
  }
  public static void
  print(double[] a, int from, int to){
    start(from, to, a.length);
    for(int i = from; i < to; i++) {
      System.out.print(a[i]);
      if(i < to - 1)
        System.out.print(", ");
    }
    end();
  }
  // Remplit un tableau en utilisant un générateur :
  public static void
  fill(Object[] a, Generator gen) {
    fill(a, 0, a.length, gen);
  }
  public static void
  fill(Object[] a, int from, int to,
       Generator gen){
    for(int i = from; i < to; i++)
      a[i] = gen.next();
  }
  public static void
  fill(boolean[] a, BooleanGenerator gen) {
    fill(a, 0, a.length, gen);
  }
  public static void
  fill(boolean[] a, int from, int to,
       BooleanGenerator gen) {
    for(int i = from; i < to; i++)
      a[i] = gen.next();
  }
  public static void
  fill(byte[] a, ByteGenerator gen) {
    fill(a, 0, a.length, gen);
```

```
  }
  public static void
  fill(byte[] a, int from, int to,
       ByteGenerator gen) {
    for(int i = from; i < to; i++)
      a[i] = gen.next();
  }
  public static void
  fill(char[] a, CharGenerator gen) {
      fill(a, 0, a.length, gen);
  }
  public static void
  fill(char[] a, int from, int to,
       CharGenerator gen) {
    for(int i = from; i < to; i++)
      a[i] = gen.next();
  }
  public static void
  fill(short[] a, ShortGenerator gen) {
      fill(a, 0, a.length, gen);
  }
  public static void
  fill(short[] a, int from, int to,
       ShortGenerator gen) {
    for(int i = from; i < to; i++)
      a[i] = gen.next();
  }
  public static void
  fill(int[] a, IntGenerator gen) {
      fill(a, 0, a.length, gen);
  }
  public static void
  fill(int[] a, int from, int to,
       IntGenerator gen) {
    for(int i = from; i < to; i++)
      a[i] = gen.next();
  }
  public static void
  fill(long[] a, LongGenerator gen) {
      fill(a, 0, a.length, gen);
  }
  public static void
  fill(long[] a, int from, int to,
       LongGenerator gen) {
    for(int i = from; i < to; i++)
      a[i] = gen.next();
  }
```

```
  public static void
  fill(float[] a, FloatGenerator gen) {
    fill(a, 0, a.length, gen);
  }
  public static void
  fill(float[] a, int from, int to,
      FloatGenerator gen) {
   for(int i = from; i < to; i++)
    a[i] = gen.next();
  }
  public static void
  fill(double[] a, DoubleGenerator gen) {
    fill(a, 0, a.length, gen);
  }
  public static void
  fill(double[] a, int from, int to,
      DoubleGenerator gen){
   for(int i = from; i < to; i++)
    a[i] = gen.next();
  }
  private static Random r = new Random();
  public static class RandBooleanGenerator
  implements BooleanGenerator {
   public boolean next() {
    return r.nextBoolean();
   }
  }
  public static class RandByteGenerator
  implements ByteGenerator {
   public byte next() {
    return (byte)r.nextInt();
   }
  }
  static String ssource =
   "ABCDEFGHIJKLMNOPQRSTUVWXYZ" +
   "abcdefghijklmnopqrstuvwxyz";
  static char[] src = ssource.toCharArray();
  public static class RandCharGenerator
  implements CharGenerator {
   public char next() {
    int pos = Math.abs(r.nextInt());
    return src[pos % src.length];
   }
  }
  public static class RandStringGenerator
  implements Generator {
   private int len;
```

```
    private RandCharGenerator cg =
      new RandCharGenerator();
    public RandStringGenerator(int length) {
      len = length;
    }
    public Object next() {
      char[] buf = new char[len];
      for(int i = 0; i < len; i++)
        buf[i] = cg.next();
      return new String(buf);
    }
  }
  public static class RandShortGenerator
  implements ShortGenerator {
    public short next() {
      return (short)r.nextInt();
    }
  }
  public static class RandIntGenerator
  implements IntGenerator {
    private int mod = 10000;
    public RandIntGenerator() {}
    public RandIntGenerator(int modulo) {
      mod = modulo;
    }
    public int next() {
      return r.nextInt() % mod;
    }
  }
  public static class RandLongGenerator
  implements LongGenerator {
    public long next() { return r.nextLong(); }
  }
  public static class RandFloatGenerator
  implements FloatGenerator {
    public float next() { return r.nextFloat(); }
  }
  public static class RandDoubleGenerator
  implements DoubleGenerator {
    public double next() {return r.nextDouble();}
  }
} ///:~
```

Pour remplir un tableau en utilisant un générateur, la méthode **fill()** accepte une référence sur une **interface** générateur, qui dispose d'une méthode **next()** produisant d'une façon ou d'une autre (selon l'implémentation de l'interface) un objet du bon type. La méthode **fill()** se contente d'appeler **next()** jusqu'à ce que la plage désirée du tableau soit remplie. Il est donc maintenant possible de créer un générateur en implémentant l'**interface** appropriée, et d'utiliser ce générateur avec **fill()**.

Les générateurs de données aléatoires sont utiles lors des tests, un ensemble de classes internes a donc été créé pour implémenter toutes les interfaces pour les types scalaires de base, de même qu'un générateur de **String** pour représenter des **Object**s. On peut noter au passage que **RandStringGenerator** utilise **RandCharGenerator** pour remplir un tableau de caractères, qui est ensuite transformé en **String**. La taille du tableau est déterminée par l'argument du constructeur.

Afin de générer des nombres qui ne soient pas trop grands, **RandIntGenerator** utilise un modulus par défaut de 10'000, mais un constructeur surchargé permet de choisir une valeur plus petite.

Voici un programme qui teste la bibliothèque et illustre la manière de l'utiliser :

```
//: c09:TestArrays2.java
// Teste et illustre les utilitaires d'Arrays2
import com.bruceeckel.util.*;

public class TestArrays2 {
  public static void main(String[] args) {
    int size = 6;
    // Ou récupère la taille depuis la ligne de commande :
    if(args.length != 0)
      size = Integer.parseInt(args[0]);
    boolean[] a1 = new boolean[size];
    byte[] a2 = new byte[size];
    char[] a3 = new char[size];
    short[] a4 = new short[size];
    int[] a5 = new int[size];
    long[] a6 = new long[size];
    float[] a7 = new float[size];
    double[] a8 = new double[size];
    String[] a9 = new String[size];
    Arrays2.fill(a1,
      new Arrays2.RandBooleanGenerator());
    Arrays2.print(a1);
    Arrays2.print("a1 = ", a1);
    Arrays2.print(a1, size/3, size/3 + size/3);
    Arrays2.fill(a2,
      new Arrays2.RandByteGenerator());
    Arrays2.print(a2);
    Arrays2.print("a2 = ", a2);
    Arrays2.print(a2, size/3, size/3 + size/3);
    Arrays2.fill(a3,
      new Arrays2.RandCharGenerator());
    Arrays2.print(a3);
    Arrays2.print("a3 = ", a3);
    Arrays2.print(a3, size/3, size/3 + size/3);
    Arrays2.fill(a4,
      new Arrays2.RandShortGenerator());
    Arrays2.print(a4);
```

```
    Arrays2.print("a4 = ", a4);
    Arrays2.print(a4, size/3, size/3 + size/3);
    Arrays2.fill(a5,
      new Arrays2.RandIntGenerator());
    Arrays2.print(a5);
    Arrays2.print("a5 = ", a5);
    Arrays2.print(a5, size/3, size/3 + size/3);
    Arrays2.fill(a6,
      new Arrays2.RandLongGenerator());
    Arrays2.print(a6);
    Arrays2.print("a6 = ", a6);
    Arrays2.print(a6, size/3, size/3 + size/3);
    Arrays2.fill(a7,
      new Arrays2.RandFloatGenerator());
    Arrays2.print(a7);
    Arrays2.print("a7 = ", a7);
    Arrays2.print(a7, size/3, size/3 + size/3);
    Arrays2.fill(a8,
      new Arrays2.RandDoubleGenerator());
    Arrays2.print(a8);
    Arrays2.print("a8 = ", a8);
    Arrays2.print(a8, size/3, size/3 + size/3);
    Arrays2.fill(a9,
      new Arrays2.RandStringGenerator(7));
    Arrays2.print(a9);
    Arrays2.print("a9 = ", a9);
    Arrays2.print(a9, size/3, size/3 + size/3);
  }
} ///:~
```

Remplir un tableau

La bibliothèque standard Java **Arrays** propose aussi une méthode **fill()**, mais celle-ci est relativement triviale : elle ne fait que dupliquer une certaine valeur dans chaque cellule, ou dans le cas d'objets, copier la même référence dans chaque cellule. En utilisant **Arrays2.print()**, les méthodes **Arrays.fill()** peuvent être facilement illustrées :

```
//: c09:FillingArrays.java
// Utilisation de Arrays.fill()
import com.bruceeckel.util.*;
import java.util.*;

public class FillingArrays {
  public static void main(String[] args) {
    int size = 6;
    // Ou récupère la taille depuis la ligne de commande :
    if(args.length != 0)
```

```
      size = Integer.parseInt(args[0]);
    boolean[] a1 = new boolean[size];
    byte[] a2 = new byte[size];
    char[] a3 = new char[size];
    short[] a4 = new short[size];
    int[] a5 = new int[size];
    long[] a6 = new long[size];
    float[] a7 = new float[size];
    double[] a8 = new double[size];
    String[] a9 = new String[size];
    Arrays.fill(a1, true);
    Arrays2.print("a1 = ", a1);
    Arrays.fill(a2, (byte)11);
    Arrays2.print("a2 = ", a2);
    Arrays.fill(a3, 'x');
    Arrays2.print("a3 = ", a3);
    Arrays.fill(a4, (short)17);
    Arrays2.print("a4 = ", a4);
    Arrays.fill(a5, 19);
    Arrays2.print("a5 = ", a5);
    Arrays.fill(a6, 23);
    Arrays2.print("a6 = ", a6);
    Arrays.fill(a7, 29);
    Arrays2.print("a7 = ", a7);
    Arrays.fill(a8, 47);
    Arrays2.print("a8 = ", a8);
    Arrays.fill(a9, "Hello");
    Arrays2.print("a9 = ", a9);
    // Manipulation de plages d'index :
    Arrays.fill(a9, 3, 5, "World");
    Arrays2.print("a9 = ", a9);
  }
} ///:~
```

On peut soit remplir un tableau complètement, soit - comme le montrent les deux dernières instructions - une certaine plage d'indices. Mais comme il n'est possible de ne fournir qu'une seule valeur pour le remplissage dans **Arrays.fill()**, les méthodes **Arrays2.fill()** sont bien plus intéressantes.

Copier un tableau

La bibliothèque standard Java propose une méthode **static**, **System.arraycopy()**, qui réalise des copies de tableau bien plus rapidement qu'une boucle **for**. **System.arraycopy()** est surchargée afin de gérer tous les types. Voici un exemple qui manipule des tableaux d'**int** :

```
//: c09:CopyingArrays.java
// Utilisation de System.arraycopy()
import com.bruceeckel.util.*;
```

```
import java.util.*;

public class CopyingArrays {
  public static void main(String[] args) {
    int[] i = new int[25];
    int[] j = new int[25];
    Arrays.fill(i, 47);
    Arrays.fill(j, 99);
    Arrays2.print("i = ", i);
    Arrays2.print("j = ", j);
    System.arraycopy(i, 0, j, 0, i.length);
    Arrays2.print("j = ", j);
    int[] k = new int[10];
    Arrays.fill(k, 103);
    System.arraycopy(i, 0, k, 0, k.length);
    Arrays2.print("k = ", k);
    Arrays.fill(k, 103);
    System.arraycopy(k, 0, i, 0, k.length);
    Arrays2.print("i = ", i);
    // Objects :
    Integer[] u = new Integer[10];
    Integer[] v = new Integer[5];
    Arrays.fill(u, new Integer(47));
    Arrays.fill(v, new Integer(99));
    Arrays2.print("u = ", u);
    Arrays2.print("v = ", v);
    System.arraycopy(v, 0,
      u, u.length/2, v.length);
    Arrays2.print("u = ", u);
  }
} ///:~
```

arraycopy() accepte comme arguments le tableau source, le déplacement dans le tableau source à partir duquel démarrer la copie, le tableau destination, le déplacement dans le tableau destination à partir duquel démarrer la copie, et le nombre d'éléments à copier. Bien entendu, toute violation des frontières du tableau générera une exception.

L'exemple montre bien qu'on peut copier des tableaux de scalaires comme des tableaux d'objets. Cependant, dans le cas de la copie de tableaux d'objets, seules les références sont copiées - il n'y a pas duplication des objets eux-mêmes. C'est ce qu'on appelle une *copie superficielle* (voir l'Annexe A).

Comparer des tableaux

Arrays fournit la méthode surchargée **equals()** pour comparer des tableaux entiers. Encore une fois, ces méthodes sont surchargées pour chacun des types de base, ainsi que pour les **Object**s. Pour être égaux, les tableaux doivent avoir la même taille et chaque élément doit être équivalent (au sens de la méthode **equals()**) à l'élément correspondant dans l'autre tableau (pour les types scalaires,

la méthode **equals()** de la classe d'encapsulation du type concerné est utilisé ; par exemple, **Integer.equals()** est utilisé pour les **int**). Voici un exemple :

```
//: c09:ComparingArrays.java
// Utilisation de Arrays.equals()
import java.util.*;

public class ComparingArrays {
  public static void main(String[] args) {
    int[] a1 = new int[10];
    int[] a2 = new int[10];
    Arrays.fill(a1, 47);
    Arrays.fill(a2, 47);
    System.out.println(Arrays.equals(a1, a2));
    a2[3] = 11;
    System.out.println(Arrays.equals(a1, a2));
    String[] s1 = new String[5];
    Arrays.fill(s1, "Hi");
    String[] s2 = {"Hi", "Hi", "Hi", "Hi", "Hi"};
    System.out.println(Arrays.equals(s1, s2));
  }
} ///:~
```

Au début du programme, **a1** et **a2** sont identiques, donc le résultat est « true » ; puis l'un des éléments est changé donc la deuxième ligne affichée est « false ». Dans le dernier cas, tous les éléments de **s1** pointent sur le même objet, alors que **s2** contient cinq objets différents. Cependant, l'égalité de tableaux est basée sur le contenu (via **Object.equals()**) et donc le résultat est « true ».

Comparaison d'éléments de tableau

L'une des fonctionnalités manquantes dans les bibliothèques Java 1.0 et 1.1 sont les opérations algorithmiques - y compris les simples tris. Ceci était relativement frustrant pour quiconque s'attendait à une bibliothèque standard conséquente. Heureusement, Java 2 a corrigé cette situation, au moins pour le problème du tri.

Le problème posé par l'écriture d'une méthode de tri générique est que le tri doit réaliser des comparaisons basées sur le type réel de l'objet. Bien sûr, l'une des approches consiste à écrire une méthode de tri différente pour chaque type, mais cela va à l'encontre du principe de réutilisabilité du code pour les nouveaux types.

L'un des buts principaux de la conception est de « séparer les choses qui changent de celles qui ne bougent pas » ; ici, le code qui reste le même est l'algorithme général de tri, alors que la manière de comparer les objets entre eux est ce qui change d'un cas d'utilisation à l'autre. Donc au lieu de coder en dur le code de comparaison dans différentes procédures de tri, on utilise ici la technique des *callbacks*. Avec un callback, la partie du code qui varie d'un cas à l'autre est encapsulé dans sa propre classe, et la partie du code qui ne change pas appellera ce code pour réaliser les comparaisons. De cette manière, il est possible de créer différents objets pour exprimer différentes sortes de comparaisons et de les passer au même code de tri.

Dans Java 2, il existe deux manières de fournir des fonctionnalités de comparaison. La *mé-*

thode naturelle de comparaison constitue la première, elle est annoncée dans une classe en implémentant l'interface **java.lang.Comparable**. C'est une interface très simple ne disposant que d'une seule méthode, **compareTo()**. Cette méthode accepte un autre **Object** comme argument, et renvoie une valeur négative si l'argument est plus grand que l'objet courant, zéro si ils sont égaux, ou une valeur positive si l'argument est plus petit que l'objet courant.

Voici une classe qui implémente **Comparable** et illustre la comparaison en utilisant la méthode **Arrays.sort()** de la bibliothèque standard Java :

```
//: c09:CompType.java
// Implémenter Comparable dans une classe.
import com.bruceeckel.util.*;
import java.util.*;

public class CompType implements Comparable {
  int i;
  int j;
  public CompType(int n1, int n2) {
    i = n1;
    j = n2;
  }
  public String toString() {
    return "[i = " + i + ", j = " + j + "]";
  }
  public int compareTo(Object rv) {
    int rvi = ((CompType)rv).i;
    return (i < rvi ? -1 : (i == rvi ? 0 : 1));
  }
  private static Random r = new Random();
  private static int randInt() {
    return Math.abs(r.nextInt()) % 100;
  }
  public static Generator generator() {
    return new Generator() {
      public Object next() {
        return new CompType(randInt(),randInt());
      }
    };
  }
  public static void main(String[] args) {
    CompType[] a = new CompType[10];
    Arrays2.fill(a, generator());
    Arrays2.print("before sorting, a = ", a);
    Arrays.sort(a);
    Arrays2.print("after sorting, a = ", a);
  }
} ///:~
```

Lorsque la fonction de comparaison est définie, il vous incombe de décider du sens à donner

à la comparaison entre deux objets. Ici, seules les valeurs **i** sont utilisées dans la comparaison, les valeurs **j** sont ignorées.

La méthode **static randInt()** produit des valeurs positives entre zéro et 100, et la méthode **generator()** produit un objet implémentant l'interface **Generator**, en créant une classe interne anonyme (cf. Chapitre 8). Celui-ci génére des objets **CompType** en les initialisant avec des valeurs aléatoires. Dans **main()**, le générateur est utilisé pour remplir un tableau de **CompType**, qui est alors trié. Si **Comparable** n'avait pas été implémentée, une erreur de compilation aurait été générée lors d'un appel à **sort()**.

Dans le cas où une classe n'implémente pas **Comparable**, ou qu'elle l'implémente d'une manière qui ne vous satisfait pas (c'est à dire que vous souhaitez une autre fonction de comparaison pour ce type), il faut utiliser une autre approche pour comparer des objets. Cette approche nécessite de créer une classe séparée qui implémente l'interface **Comparator**, comportant les deux méthodes **compare()** et **equals()**. Cependant, sauf cas particuliers (pour des raisons de performance notamment), il n'est pas nécessaire d'implémenter **equals()** car chaque classe dérive implicitement de **Object**, qui fournit déjà cette méthode. On peut donc se contenter de la méthode **Object.equals()** pour satisfaire au contrat imposé par l'interface.

La classe **Collections** (que nous étudierons plus en détails par la suite) dispose d'un **Comparator** qui inverse l'ordre de tri. Ceci peut facilement être appliqué à **CompType** :

```
//: c09:Reverse.java
// Le Comparator Collecions.reverseOrder().
import com.bruceeckel.util.*;
import java.util.*;

public class Reverse {
  public static void main(String[] args) {
    CompType[] a = new CompType[10];
    Arrays2.fill(a, CompType.generator());
    Arrays2.print("before sorting, a = ", a);
    Arrays.sort(a, Collections.reverseOrder());
    Arrays2.print("after sorting, a = ", a);
  }
} ///:~
```

L'appel à **Collections.reverseOrder()** produit une référence sur le **Comparator**.

Voici un deuxième exemple dans lequel un **Comparator** compare des objets **CompType** en se basant cette fois sur la valeur de leur **j** plutôt que sur celle de **i** :

```
//: c09:ComparatorTest.java
// Implémenter un Comparator pour une classe.
import com.bruceeckel.util.*;
import java.util.*;

class CompTypeComparator implements Comparator {
  public int compare(Object o1, Object o2) {
    int j1 = ((CompType)o1).j;
    int j2 = ((CompType)o2).j;
```

```
    return (j1 < j2 ? -1 : (j1 == j2 ? 0 : 1));
  }
}

public class ComparatorTest {
  public static void main(String[] args) {
    CompType[] a = new CompType[10];
    Arrays2.fill(a, CompType.generator());
    Arrays2.print("before sorting, a = ", a);
    Arrays.sort(a, new CompTypeComparator());
    Arrays2.print("after sorting, a = ", a);
  }
} ///:~
```

La méthode **compare()** doit renvoyer un entier négatif, zéro ou un entier positif selon que le premier argument est respectivement plus petit, égal ou plus grand que le second.

Trier un tableau

Avec les méthodes de tri intégrées, il est maintenant possible de trier n'importe quel tableau de scalaires ou d'objets implémentant **Comparable** ou disposant d'une classe **Comparator** associée. Ceci comble un énorme trou dans les bibliothèques de Java - croyez-le ou non, Java 1.0 ou 1.1 ne fournissait aucun moyen de trier des **String**s ! Voici un exemple qui génére des objets **String** aléatoirement et les trie :

```
//: c09:StringSorting.java
// Trier un tableau de Strings.
import com.bruceeckel.util.*;
import java.util.*;

public class StringSorting {
  public static void main(String[] args) {
    String[] sa = new String[30];
    Arrays2.fill(sa,
      new Arrays2.RandStringGenerator(5));
    Arrays2.print("Before sorting: ", sa);
    Arrays.sort(sa);
    Arrays2.print("After sorting: ", sa);
  }
} ///:~
```

Il est bon de noter que le tri effectué sur les **String**s est *lexicographique*, c'est à dire que les mots commençant par des majuscules apparaissent avant ceux débutant par une minuscule (typiquement les annuaires sont triés de cette façon). Il est toutefois possible de redéfinir ce comportement et d'ignorer la casse en définissant une classe **Comparator**. Cette classe sera placée dans le package « util » à des fins de réutilisation :

```
//: com:bruceeckel:util:AlphabeticComparator.java
```

```
// Garder les lettres majuscules et minuscules ensemble.
package com.bruceeckel.util;
import java.util.*;

public class AlphabeticComparator
implements Comparator{
  public int compare(Object o1, Object o2) {
    String s1 = (String)o1;
    String s2 = (String)o2;
    return s1.toLowerCase().compareTo(
      s2.toLowerCase());
  }
} ///:~
```

Chaque **String** est convertie en minuscules avant la comparaison. La méthode **compareTo()** de **String** fournit ensuite le comparateur désiré.

Voici un exemple d'utilisation d'**AlphabeticComparator** :

```
//: c09:AlphabeticSorting.java
// Garder les lettres majuscules et minuscules ensemble.
import com.bruceeckel.util.*;
import java.util.*;

public class AlphabeticSorting {
  public static void main(String[] args) {
    String[] sa = new String[30];
    Arrays2.fill(sa,
      new Arrays2.RandStringGenerator(5));
    Arrays2.print("Before sorting: ", sa);
    Arrays.sort(sa, new AlphabeticComparator());
    Arrays2.print("After sorting: ", sa);
  }
} ///:~
```

L'algorithme de tri utilisé dans la bibliothèque standard de Java est conçu pour être optimal suivant le type d'objets triés : un Quicksort pour les scalaires, et un tri-fusion stable pour les objets. Vous ne devriez donc pas avoir à vous soucier des performances à moins qu'un outil de profilage ne vous démontre explicitement que le goulot d'étranglement de votre programme soit le processus de tri.

Effectuer une recherche sur un tableau trié

Une fois un tableau trié, il est possible d'effectuer une recherche rapide sur un item en utilisant **Arrays.binarySearch()**. Il est toutefois très important de ne pas utiliser **binarySearch()** sur un tableau non trié ; le résultat en serait imprévisible. L'exemple suivant utilise un **RandIntGenerator** pour remplir un tableau et produire des valeurs à chercher dans ce tableau :

```
//: c09:ArraySearching.java
```

```
// Utilisation de Arrays.binarySearch().
import com.bruceeckel.util.*;
import java.util.*;

public class ArraySearching {
  public static void main(String[] args) {
    int[] a = new int[100];
    Arrays2.RandIntGenerator gen =
      new Arrays2.RandIntGenerator(1000);
    Arrays2.fill(a, gen);
    Arrays.sort(a);
    Arrays2.print("Sorted array: ", a);
    while(true) {
      int r = gen.next();
      int location = Arrays.binarySearch(a, r);
      if(location >= 0) {
        System.out.println("Location of " + r +
          " is " + location + ", a[" +
          location + "] = " + a[location]);
        break; // Sortie de la boucle while
      }
    }
  }
} ///:~
```

Arrays.binarySearch() renvoie une valeur supérieure ou égale à zéro si l'item recherché est trouvé. Dans le cas contraire, elle renvoie une valeur négative représentant l'endroit où insérer l'élément si on désirait maintenir le tableau trié à la main. La valeur retournée est :

```
-(point d'insertion) - 1
```

Le point d'insertion est l'index du premier élément plus grand que la clef, ou **a.size()** si tous les éléments du tableau sont plus petits que la clef spécifiée.

Si le tableau contient des éléments dupliqués, aucune garantie n'est apportée quant à celui qui sera trouvé. L'algorithme n'est donc pas conçu pour les tableaux comportant des doublons, bien qu'il les tolère. Dans le cas où on a besoin d'une liste triée d'éléments sans doublons, mieux vaut se tourner vers un **TreeSet** (qui sera introduit plus loin dans ce chapitre) qui gère tous ces détails automatiquement, plutôt que de maintenir un tableau à la main (à moins que des questions de performance ne se greffent là-dessus).

Il faut fournir à **binarySearch()** le même objet **Comparator** que celui utilisé pour trier le tableau d'objets (les tableaux de scalaires n'autorisent pas les tris avec des **Comparator**), afin qu'elle utilise la version redéfinie de la fonction de comparaison. Ainsi, le programme **AlphabeticSorting.java** peut être modifié pour effectuer une recherche :

```
//: c09:AlphabeticSearch.java
// Rechercher avec un Comparator.
import com.bruceeckel.util.*;
```

```
import java.util.*;

public class AlphabeticSearch {
  public static void main(String[] args) {
    String[] sa = new String[30];
    Arrays2.fill(sa,
      new Arrays2.RandStringGenerator(5));
    AlphabeticComparator comp =      new AlphabeticComparator();
    Arrays.sort(sa, comp);
    int index =      Arrays.binarySearch(sa, sa[10], comp);
    System.out.println("Index = " + index);
  }
} ///:~
```

binarySearch() accepte le **Comparator** en troisième argument. Dans l'exemple précédent, le succès de la recherche est garanti puisque l'item recherché est tiré du tableau lui-même.

Résumé sur les tableaux

Pour résumer ce qu'on a vu jusqu'à présent, un tableau se révèle la manière la plus simple et la plus efficace pour stocker un groupe d'objets, et le seul choix possible dans le cas où on veut stocker un ensemble de scalaires. Dans le reste de ce chapitre nous allons étudier le cas plus général dans lequel on ne sait pas au moment de l'écriture du programme combien d'objets seront requis, ainsi que des moyens plus sophistiqués de stocker les objets. Java propose en effet des *classes conteneurs* qui adressent différents problèmes. Les types de base en sont les **List**s, les **Set**s et les **Map**s. Un nombre surprenant de problèmes peuvent être facilement résolus grâce à ces outils.

Entre autres caractéristiques - les **Set**s, par exemple, ne stockent qu'un objet de chaque valeur, les **Map**s sont des *tableaux associatifs* qui permettent d'associer n'importe quel objet avec n'importe quel autre objet - les classes conteneurs de Java se redimensionnent automatiquement. A l'inverse des tableaux, ils peuvent donc stocker un nombre quelconque d'objets et on n'a pas besoin de se soucier de leur taille lors de l'écriture du programme.

Introduction sur les conteneurs

Les classes conteneurs sont à mon sens l'un des outils les plus puissants disponibles parce qu'ils augmentent de façon significative la productivité du développement. Les conteneurs de Java 2 résultent d'une reconception approfondie [47] des implémentations relativement pauvres disponibles dans Java 1.0 et 1.1. Cette reconception a permis d'unifier et de rationnaliser certains fonctionnements. Elle a aussi comblé certains manques de la bibliothèque des conteneurs tels que les listes chaînées, les files (queues) et les files doubles (queues à double entrée).

La conception d'une bibliothèque de conteneurs est difficile (de même que tous les problèmes de conception des bibliothèques). En C++, les classes conteneurs couvrent les bases grâce à de nombreuses classes différentes. C'est mieux que ce qui était disponible avant (ie, rien), mais le résultat ne se transpose pas facilement dans Java. J'ai aussi rencontré l'approche opposée, où la bibliothèque de conteneurs consistait en une seule classe qui fonctionnait à la fois comme une séquence linéaire et un tableau associatif. La bibliothèque de conteneurs de Java 2 essaie de trouver un juste milieu : les fonctionnalités auxquelles on peut s'attendre de la part d'une bibliothèque de

conteneurs mâture, mais plus facile à appréhender que les classes conteneurs du C++ ou d'autres bibliothèques de conteneurs similaires. Le résultat peut paraître étrange dans certains cas. Mais contrairement à certaines décisions prises dans la conception des premières bibliothèques Java, ces bizarreries ne sont pas des accidents de conception, mais des compromis minutieusement examinés sur la complexité. Il vous faudra peut-être un petit moment avant d'être à l'aise avec certains aspects de la bibliothèque, mais je pense que vous adopterez quand même très rapidement ces nouveaux outils.

Le but de la bibliothèque de conteneurs de Java 2 est de « stocker des objets » et le divise en deux concepts bien distincts :

1. **Collection** : un groupe d'éléments individuels, souvent associé à une règle définissant leur comportement. Une **List** doit garder les éléments dans un ordre précis, et un **Set** ne peut contenir de doublons (les *sacs* [NdT : *bag* en anglais], qui ne sont pas implémentés dans la bibliothèque de conteneurs de Java - les **List**s fournissant des fonctionnalités équivalentes - ne possèdent pas une telle règle).

2. **Map** : un ensemble de paires clef - valeur. A première vue, on pourrait penser qu'il ne s'agit que d'une **Collection** de paires, mais lorsqu'on essaie de l'implémenter de cette manière, le design devient très rapidement bancal et lourd à mettre en oeuvre ; il est donc plus simple d'en faire un concept séparé. D'un autre côté, il est bien pratique d'examiner certaines portions d'une **Map** en créant une **Collection** représentant cette portion. Une **Map** peut donc renvoyer un **Set** de ses clefs, une **Collection** de ses valeurs**,** ou un **Set** de ses paires. Les **Map**s, comme les tableaux, peuvent facilement être étendus dans de multiples dimensions sans ajouter de nouveaux concepts : il suffit de créer une **Map** dont les valeurs sont des **Map**s (les valeurs de ces **Map**s pouvant *elles-mêmes* être des **Map**s, etc.).

Nous allons d'abord examiner les fonctionnalités générales des conteneurs, puis aller dans les spécificités des conteneurs et enfin nous apprendrons pourquoi certains conteneurs sont déclinés en plusieurs versions, et comment choisir entre eux.

Imprimer les conteneurs

A l'inverse des tableaux, les conteneurs s'affichent correctement sans aide. Voici un exemple qui introduit en même temps les conteneurs de base :

```
//: c09:PrintingContainers.java
// Les conteneurs savent comment s'afficher.
import java.util.*;

public class PrintingContainers {
  static Collection fill(Collection c) {
    c.add("dog");
    c.add("dog");
    c.add("cat");
    return c;
  }
  static Map fill(Map m) {
    m.put("dog", "Bosco");
    m.put("dog", "Spot");
```

```
    m.put("cat", "Rags");
    return m;
  }
  public static void main(String[] args) {
    System.out.println(fill(new ArrayList()));
    System.out.println(fill(new HashSet()));
    System.out.println(fill(new HashMap()));
  }
} ///:~
```

Comme mentionné précédemment, il existe deux catégories de base dans la bibliothèque de conteneurs Java. La distinction est basée sur le nombre d'items stockés dans chaque cellule du conteneur. La catégorie **Collection** ne stocke qu'un item dans chaque emplacement (le nom est un peu trompeur puisque les bibliothèques des conteneurs sont souvent appelées des « collections »). Elle inclut la **List**, qui stocke un groupe d'items dans un ordre spécifique, et le **Set**, qui autorise l'addition d'une seule instance pour chaque item. Une **ArrayList** est un type de **List**, et **HashSet** est un type de **Set**. La méthode **add()** permet d'ajouter des éléments dans une **Collection**.

Une **Map** contient des paires clef - valeur, un peu à la manière d'une mini base de données. Le programme précédent utilise un type de **Map**, le **HashMap**. Si on dispose d'une **Map** qui associe les états des USA avec leur capitale et qu'on souhaite connaître la capitale de l'Ohio, il suffit de la rechercher - comme si on indexait un tableau (les **Map**s sont aussi appelés des *tableaux associatifs*). La méthode **put()**, qui accepte deux arguments - la clef et la valeur -, permet de stocker des éléments dans une **Map**. L'exemple précédent se contente d'ajouter des éléments mais ne les récupère pas une fois stockés. Ceci sera illustré plus tard.

Les méthodes surchargées **fill()** remplissent respectivement des **Collection**s et des **Map**s. En examinant la sortie produite par le programme, on peut voir que le comportement par défaut pour l'affichage (fourni par les méthodes **toString()** des différents conteneurs) produit un résultat relativement clair, il n'est donc pas nécessaire d'ajouter du code pour imprimer les conteneurs comme nous avons du le faire avec les tableaux :

```
[dog, dog, cat]
[cat, dog]
{cat=Rags, dog=Spot}
```

Une **Collection** est imprimée entre crochets, chaque élément étant séparé par une virgule. Une **Map** est entourée par des accolades, chaque clef étant associée à sa valeur avec un signe égal (les clefs à gauche, les valeurs à droite).

Le comportement de base des différents conteneurs est évident dans cet exemple. La **List** stocke les objets dans l'ordre exact où ils ont été ajoutés, sans aucun réarrangement ni édition. Le **Set**, lui, n'accepte qu'une seule instance d'un objet et utilise une méthode interne de tri (en général, un **Set** sert à savoir si un élément est un membre d'un **Set** ou non, et non l'ordre dans lequel il apparaît dans ce **Set** - pour cela il faut utiliser une **List**). La **Map** elle aussi n'accepte qu'une seule instance d'un objet pour la clef, possède elle aussi sa propre organisation interne et ne tient pas compte de l'ordre dans lequel les éléments ont été insérés.

Remplir les conteneurs

Bien que le problème d'impression des conteneurs soit géré pour nous, le remplissage des conteneurs souffre des mêmes limitations que **java.util.Arrays**. De même que pour les **Arrays**, il existe une classe compagnon appelée **Collections** contenant des méthodes **static** dont l'une s'appelle **fill()**. Cette méthode **fill()** ne fait que dupliquer une unique référence sur un objet dans le conteneur, et ne fonctionne que sur les objets **List**, pas sur les **Set**s ni les **Map**s :

```
//: c09:FillingLists.java
// La méthode Collections.fill().
import java.util.*;

public class FillingLists {
  public static void main(String[] args) {
    List list = new ArrayList();
    for(int i = 0; i < 10; i++)
      list.add("");
    Collections.fill(list, "Hello");
    System.out.println(list);
  }
} ///:~
```

Cette méthode est encore moins intéressante parce qu'elle ne fait que remplacer les éléments déjà présents dans la **List**, sans ajouter aucun élément.

Pour être capable de créer des exemples intéressants, voici une bibliothèque complémentaire **Collections2** (appartenant par commodité à **com.bruceeckel.util**) disposant d'une méthode **fill()** utilisant un générateur pour ajouter des éléments, et permettant de spécifier le nombre d'éléments qu'on souhaite ajouter. L'**interface Generator** définie précédemment fonctionne pour les **Collection**s, mais les **Map**s requièrent leur propre **interface** générateur puisqu'un appel à **next()** doit produire une paire d'objets (une clef et une valeur). Voici tout d'abord la classe **Pair** :

```
//: com:bruceeckel:util:Pair.java
package com.bruceeckel.util;
public class Pair {
  public Object key, value;
  Pair(Object k, Object v) {
    key = k;
    value = v;
  }
} ///:~
```

Ensuite, l'**interface** générateur qui produit un objet **Pair** :

```
//: com:bruceeckel:util:MapGenerator.java
package com.bruceeckel.util;
public interface MapGenerator {
  Pair next();
} ///:~
```

Avec ces deux objets, un ensemble d'utilitaires pour travailler avec les classes conteneurs

peuvent être développés :

```
//: com:bruceeckel:util:Collections2.java
// Remplir n'importe quel type de conteneur en
// utilisant un objet générateur.
package com.bruceeckel.util;
import java.util.*;

public class Collections2 {
  // Remplit une Collection en utilisant un générateur :
  public static void
  fill(Collection c, Generator gen, int count) {
    for(int i = 0; i < count; i++)
      c.add(gen.next());
  }
  public static void
  fill(Map m, MapGenerator gen, int count) {
    for(int i = 0; i < count; i++) {
      Pair p = gen.next();
      m.put(p.key, p.value);
    }
  }
  public static class RandStringPairGenerator
  implements MapGenerator {
    private Arrays2.RandStringGenerator gen;
    public RandStringPairGenerator(int len) {
      gen = new Arrays2.RandStringGenerator(len);
    }
    public Pair next() {
      return new Pair(gen.next(), gen.next());
    }
  }
  // Objet par défaut afin de ne pas avoir
  // à en créer un de notre cru :
  public static RandStringPairGenerator rsp =    new RandStringPairGenerator(10);
  public static class StringPairGenerator
  implements MapGenerator {
    private int index = -1;
    private String[][] d;
    public StringPairGenerator(String[][] data) {
      d = data;
    }
    public Pair next() {
      // Force l'index dans la plage de valeurs :
      index = (index + 1) % d.length;
      return new Pair(d[index][0], d[index][1]);
    }
    public StringPairGenerator reset() {
```

```
    index = -1;
    return this;
  }
}
// Utilisation d'un ensemble de données prédéfinies :
public static StringPairGenerator geography =   new StringPairGenerator(
  CountryCapitals.pairs);
// Produit une séquence à partir d'un tableau 2D :
public static class StringGenerator
implements Generator {
  private String[][] d;
  private int position;
  private int index = -1;
  public
  StringGenerator(String[][] data, int pos) {
    d = data;
    position = pos;
  }
  public Object next() {
    // Force l'index dans la plage de valeurs :
    index = (index + 1) % d.length;
    return d[index][position];
  }
  public StringGenerator reset() {
    index = -1;
    return this;
  }
}
// Utilisation d'un ensemble de données prédéfinies :
public static StringGenerator countries =   new StringGenerator(CountryCapi-
tals.pairs,0);
public static StringGenerator capitals =   new StringGenerator(CountryCapitals.pairs,
1);
} ///:~
```

Voici l'ensemble de données prédéfinies, qui consiste en noms de pays avec leur capitale. Il est affiché avec une petite fonte afin de réduire l'espace occupé :

```
//: com:bruceeckel:util:CountryCapitals.java
package com.bruceeckel.util;
public class CountryCapitals {
  public static final String[][] pairs = {
    // Afrique
    {"ALGERIA","Algiers"}, {"ANGOLA","Luanda"},
    {"BENIN","Porto-Novo"}, {"BOTSWANA","Gaberone"},
    {"BURKINA FASO","Ouagadougou"}, {"BURUNDI","Bujumbura"},
    {"CAMEROON","Yaounde"}, {"CAPE VERDE","Praia"},
    {"CENTRAL AFRICAN REPUBLIC","Bangui"},
```

```
        {"CHAD","N'djamena"},  {"COMOROS","Moroni"},
        {"CONGO","Brazzaville"}, {"DJIBOUTI","Dijibouti"},
        {"EGYPT","Cairo"}, {"EQUATORIAL GUINEA","Malabo"},
        {"ERITREA","Asmara"}, {"ETHIOPIA","Addis Ababa"},
        {"GABON","Libreville"}, {"THE GAMBIA","Banjul"},
        {"GHANA","Accra"}, {"GUINEA","Conakry"},
        {"GUINEA","-"}, {"BISSAU","Bissau"},
        {"CETE D'IVOIR (IVORY COAST)","Yamoussoukro"},
        {"KENYA","Nairobi"}, {"LESOTHO","Maseru"},
        {"LIBERIA","Monrovia"}, {"LIBYA","Tripoli"},
        {"MADAGASCAR","Antananarivo"}, {"MALAWI","Lilongwe"},
        {"MALI","Bamako"}, {"MAURITANIA","Nouakchott"},
        {"MAURITIUS","Port Louis"}, {"MOROCCO","Rabat"},
        {"MOZAMBIQUE","Maputo"}, {"NAMIBIA","Windhoek"},
        {"NIGER","Niamey"}, {"NIGERIA","Abuja"},
        {"RWANDA","Kigali"}, {"SAO TOME E PRINCIPE","Sao Tome"},
        {"SENEGAL","Dakar"}, {"SEYCHELLES","Victoria"},
        {"SIERRA LEONE","Freetown"}, {"SOMALIA","Mogadishu"},
        {"SOUTH AFRICA","Pretoria/Cape Town"}, {"SUDAN","Khartoum"},
        {"SWAZILAND","Mbabane"}, {"TANZANIA","Dodoma"},
        {"TOGO","Lome"}, {"TUNISIA","Tunis"},
        {"UGANDA","Kampala"},
        {"DEMOCRATIC REPUBLIC OF THE CONGO (ZAIRE)","Kinshasa"},
        {"ZAMBIA","Lusaka"}, {"ZIMBABWE","Harare"},
        // Asie
        {"AFGHANISTAN","Kabul"}, {"BAHRAIN","Manama"},
        {"BANGLADESH","Dhaka"}, {"BHUTAN","Thimphu"},
        {"BRUNEI","Bandar Seri Begawan"}, {"CAMBODIA","Phnom Penh"},
        {"CHINA","Beijing"}, {"CYPRUS","Nicosia"},
        {"INDIA","New Delhi"}, {"INDONESIA","Jakarta"},
        {"IRAN","Tehran"}, {"IRAQ","Baghdad"},
        {"ISRAEL","Jerusalem"}, {"JAPAN","Tokyo"},
        {"JORDAN","Amman"}, {"KUWAIT","Kuwait City"},
        {"LAOS","Vientiane"}, {"LEBANON","Beirut"},
        {"MALAYSIA","Kuala Lumpur"}, {"THE MALDIVES","Male"},
        {"MONGOLIA","Ulan Bator"}, {"MYANMAR (BURMA)","Rangoon"},
        {"NEPAL","Katmandu"}, {"NORTH KOREA","P'yongyang"},
        {"OMAN","Muscat"}, {"PAKISTAN","Islamabad"},
        {"PHILIPPINES","Manila"}, {"QATAR","Doha"},
        {"SAUDI ARABIA","Riyadh"}, {"SINGAPORE","Singapore"},
        {"SOUTH KOREA","Seoul"}, {"SRI LANKA","Colombo"},
        {"SYRIA","Damascus"}, {"TAIWAN (REPUBLIC OF CHINA)","Taipei"},
        {"THAILAND","Bangkok"}, {"TURKEY","Ankara"},
        {"UNITED ARAB EMIRATES","Abu Dhabi"}, {"VIETNAM","Hanoi"},
        {"YEMEN","Sana'a"},
        // Australie et Océanie
        {"AUSTRALIA","Canberra"}, {"FIJI","Suva"},
```

```
{"KIRIBATI","Bairiki"},
{"MARSHALL ISLANDS","Dalap-Uliga-Darrit"},
{"MICRONESIA","Palikir"}, {"NAURU","Yaren"},
{"NEW ZEALAND","Wellington"}, {"PALAU","Koror"},
{"PAPUA NEW GUINEA","Port Moresby"},
{"SOLOMON ISLANDS","Honaira"}, {"TONGA","Nuku'alofa"},
{"TUVALU","Fongafale"}, {"VANUATU","< Port-Vila"},
{"WESTERN SAMOA","Apia"},
// Europe de l'Est et ancienne URSS
{"ARMENIA","Yerevan"}, {"AZERBAIJAN","Baku"},
{"BELARUS (BYELORUSSIA)","Minsk"}, {"GEORGIA","Tbilisi"},
{"KAZAKSTAN","Almaty"}, {"KYRGYZSTAN","Alma-Ata"},
{"MOLDOVA","Chisinau"}, {"RUSSIA","Moscow"},
{"TAJIKISTAN","Dushanbe"}, {"TURKMENISTAN","Ashkabad"},
{"UKRAINE","Kyiv"}, {"UZBEKISTAN","Tashkent"},
// Europe
{"ALBANIA","Tirana"}, {"ANDORRA","Andorra la Vella"},
{"AUSTRIA","Vienna"}, {"BELGIUM","Brussels"},
{"BOSNIA","-"}, {"HERZEGOVINA","Sarajevo"},
{"CROATIA","Zagreb"}, {"CZECH REPUBLIC","Prague"},
{"DENMARK","Copenhagen"}, {"ESTONIA","Tallinn"},
{"FINLAND","Helsinki"}, {"FRANCE","Paris"},
{"GERMANY","Berlin"}, {"GREECE","Athens"},
{"HUNGARY","Budapest"}, {"ICELAND","Reykjavik"},
{"IRELAND","Dublin"}, {"ITALY","Rome"},
{"LATVIA","Riga"}, {"LIECHTENSTEIN","Vaduz"},
{"LITHUANIA","Vilnius"}, {"LUXEMBOURG","Luxembourg"},
{"MACEDONIA","Skopje"}, {"MALTA","Valletta"},
{"MONACO","Monaco"}, {"MONTENEGRO","Podgorica"},
{"THE NETHERLANDS","Amsterdam"}, {"NORWAY","Oslo"},
{"POLAND","Warsaw"}, {"PORTUGAL","Lisbon"},
{"ROMANIA","Bucharest"}, {"SAN MARINO","San Marino"},
{"SERBIA","Belgrade"}, {"SLOVAKIA","Bratislava"},
{"SLOVENIA","Ljujiana"}, {"SPAIN","Madrid"},
{"SWEDEN","Stockholm"}, {"SWITZERLAND","Berne"},
{"UNITED KINGDOM","London"}, {"VATICAN CITY","---"},
// Amérique du Nord et Amérique Centrale
{"ANTIGUA AND BARBUDA","Saint John's"}, {"BAHAMAS","Nassau"},
{"BARBADOS","Bridgetown"}, {"BELIZE","Belmopan"},
{"CANADA","Ottawa"}, {"COSTA RICA","San Jose"},
{"CUBA","Havana"}, {"DOMINICA","Roseau"},
{"DOMINICAN REPUBLIC","Santo Domingo"},
{"EL SALVADOR","San Salvador"}, {"GRENADA","Saint George's"},
{"GUATEMALA","Guatemala City"}, {"HAITI","Port-au-Prince"},
{"HONDURAS","Tegucigalpa"}, {"JAMAICA","Kingston"},
{"MEXICO","Mexico City"}, {"NICARAGUA","Managua"},
{"PANAMA","Panama City"}, {"ST. KITTS","-"},
```

```
    {"NEVIS","Basseterre"}, {"ST. LUCIA","Castries"},
    {"ST. VINCENT AND THE GRENADINES","Kingstown"},
    {"UNITED STATES OF AMERICA","Washington, D.C."},
    // Amérique du Sud
    {"ARGENTINA","Buenos Aires"},
    {"BOLIVIA","Sucre (legal)/La Paz(administrative)"},
    {"BRAZIL","Brasilia"}, {"CHILE","Santiago"},
    {"COLOMBIA","Bogota"}, {"ECUADOR","Quito"},
    {"GUYANA","Georgetown"}, {"PARAGUAY","Asuncion"},
    {"PERU","Lima"}, {"SURINAME","Paramaribo"},
    {"TRINIDAD AND TOBAGO","Port of Spain"},
    {"URUGUAY","Montevideo"}, {"VENEZUELA","Caracas"},
  };
} ///:~
```

Il s'agit juste d'un tableau bidimensionnel de **String**[48]. Voici un simple test illustrant les méthodes **fill()** et les générateurs :

```
//: c09:FillTest.java
import com.bruceeckel.util.*;
import java.util.*;

public class FillTest {
  static Generator sg =
    new Arrays2.RandStringGenerator(7);
  public static void main(String[] args) {
    List list = new ArrayList();
    Collections2.fill(list, sg, 25);
    System.out.println(list + "\n");
    List list2 = new ArrayList();
    Collections2.fill(list2,
      Collections2.capitals, 25);
    System.out.println(list2 + "\n");
    Set set = new HashSet();
    Collections2.fill(set, sg, 25);
    System.out.println(set + "\n");
    Map m = new HashMap();
    Collections2.fill(m, Collections2.rsp, 25);
    System.out.println(m + "\n");
    Map m2 = new HashMap();
    Collections2.fill(m2,
      Collections2.geography, 25);
    System.out.println(m2);
  }
} ///:~
```

Avec ces outils vous pourrez facilement tester les différents conteneurs en les remplissant avec des données intéressantes.

L'inconvénient des conteneurs : le type est inconnu

L'« inconvénient » des conteneurs Java est qu'on perd l'information du type lorsqu'un objet est stocké dedans, ce qui est tout à fait normal puisque le programmeur de la classe conteneur n'a aucune idée du type spécifique qu'on veut stocker dans le conteneur, et que fournir un conteneur qui ne sache stocker qu'un seul type d'objets irait à l'encontre du but de généricité de l'outil conteneur. C'est pourquoi les conteneurs stockent des références sur des **Object**s, la classe de base de toutes les classes, afin de pouvoir stocker n'importe quel type d'objet (à l'exception bien sûr des types scalaires, qui ne dérivent aucune classe). Cette solution est formidable dans sa conception, sauf sur deux points :

1. Puisque l'information de type est ignorée lorsqu'on stocke une référence dans un conteneur, on ne peut placer aucune restriction sur le type de l'objet stocké dans le conteneur, même si on l'a créé pour ne contenir, par exemple, que des chats. Quelqu'un pourrait très bien ajouter un chien dans le conteneur.
2. Puisque l'information de type est perdue, la seule chose que le conteneur sache est qu'il contient une référence sur un objet. Il faut réaliser un transtypage sur le type adéquat avant de l'utiliser.

Du côté des choses positives, Java ne permettra pas une mauvaise utilisation des objets stockés dans un conteneur. Si on stocke un chien dans le conteneur de chats et qu'on essaie ensuite de traiter tous les objets du conteneur comme un chat, Java générera une *run-time* exception lors de la tentative de transtypage en chat de la référence sur le chien.

Voici un exemple utilisant le conteneur à tout faire **ArrayList**. Les débutants peuvent considérer une **ArrayList** comme « un tableau qui se redimensionne de lui-même ». L'utilisation d'une **ArrayList** est aisée : il suffit de la créer, d'y ajouter des éléments avec la méthode **add()**, et d'y accéder par la suite grâce à la méthode **get()** en utilisant un index - comme pour un tableau, mais sans les crochets [49]. **ArrayList** propose aussi une méthode **size()** qui permet de savoir combien d'éléments ont été stockés afin de ne pas dépasser les frontières et causer une exception.

Tout d'abord, nous créons les classes **Cat** et **Dog** :

```
//: c09:Cat.java
public class Cat {
  private int catNumber;
  Cat(int i) { catNumber = i; }
  void print() {
    System.out.println("Cat #" + catNumber);
  }
} ///:~

//: c09:Dog.java
public class Dog {
  private int dogNumber;
  Dog(int i) { dogNumber = i; }
  void print() {
    System.out.println("Dog #" + dogNumber);
  }
} ///:~
```

Des **Cat**s et des **Dog**s sont placés dans le conteneur, puis extraits :

```
//: c09:CatsAndDogs.java
// Exemple simple avec un conteneur.
import java.util.*;

public class CatsAndDogs {
  public static void main(String[] args) {
    ArrayList cats = new ArrayList();
    for(int i = 0; i < 7; i++)
      cats.add(new Cat(i));
    // Ce n'est pas un problème d'ajouter un chien parmi les chats :
    cats.add(new Dog(7));
    for(int i = 0; i < cats.size(); i++)
      ((Cat)cats.get(i)).print();
    // Le chien est détecté seulement lors de l'exécution.
  }
} ///:~
```

Ceci est plus qu'ennuyeux. Cela peut mener de plus à des bugs relativement durs à trouver. Si une partie (ou plusieurs parties) du programme insère des objets dans le conteneur, et qu'on découvre dans une partie complètement différente du programme via une exception qu'un objet du mauvais type a été placé dans le conteneur, il faut alors déterminer où l'insertion coupable s'est produite. Cependant, il est pratique de démarrer avec les classes conteneur standard pour programmer, en dépit de leurs limitations et de leur lourdeur.

Quelquefois ça marche quand même

Dans certains cas les choses semblent fonctionner correctement sans avoir à transtyper vers le type originel. Un cas particulier est constitué par la classe **String** que le compilateur traite de manière particulière pour la faire fonctionner de manière idoine. Quand le compilateur attend un objet **String** et qu'il obtient autre chose, il appellera automatiquement la méthode **toString()** définie dans **Object** et qui peut être redéfinie par chaque classe Java. Cette méthode produit l'objet **String** désiré, qui est ensuite utilisé là où il était attendu.

Il suffit donc de redéfinir la méthode **toString()** pour afficher un objet d'une classe donnée, comme on peut le voir dans l'exemple suivant :

```
//: c09:Mouse.java
// Redéfinition de toString().
public class Mouse {
  private int mouseNumber;
  Mouse(int i) { mouseNumber = i; }
  // Redféinition de Object.toString():
  public String toString() {
    return "This is Mouse #" + mouseNumber;
  }
  public int getNumber() {
    return mouseNumber;
```

```
  }
} ///:~



//: c09:WorksAnyway.java
// Dans certains cas spéciaux, les choses
// semblent fonctionner correctement.
import java.util.*;

class MouseTrap {
  static void caughtYa(Object m) {
    Mouse mouse = (Mouse)m; // Transtypage depuis un Object
    System.out.println("Mouse: " +
      mouse.getNumber());
  }
}

public class WorksAnyway {
  public static void main(String[] args) {
    ArrayList mice = new ArrayList();
    for(int i = 0; i < 3; i++)
      mice.add(new Mouse(i));
    for(int i = 0; i < mice.size(); i++) {
      // Aucun transtypage nécessaire, appel
      // automatique à Object.toString() :
      System.out.println(
        "Free mouse: " + mice.get(i));
      MouseTrap.caughtYa(mice.get(i));
    }
  }
} ///:~
```

La méthode **toString()** est redéfinie dans **Mouse**. Dans la deuxième boucle **for** de **main()** on peut voir l'instruction :

```
System.out.println("Free mouse: " + mice.get(i));
```

Après le signe « + » le compilateur s'attend à trouver un objet **String**. **get()** renvoie un **Object**, le compilateur appelle donc implicitement la méthode **toString()** pour obtenir l'objet **String** désiré. Cependant, ce comportement magique n'est possible qu'avec les **String**, il n'est pas disponible pour les autres types.

Une seconde approche pour cacher le transtypage est de le placer dans la classe **MouseTrap**. La méthode **caughtYa()** n'accepte pas une **Mouse** mais un **Object**, qu'elle transtype alors en **Mouse**. Ceci ne fait que repousser le problème puisqu'en acceptant un **Object** on peut passer un objet de n'importe quel type à la méthode. Cependant, si le transtypage n'est pas valide - si un objet du mauvais type est passé en argument - une exception est générée lors de l'exécution. Ce n'est pas aussi bien qu'un contrôle lors de la compilation, mais l'approche reste robuste. Notez qu'aucun transty-

page n'est nécessaire lors de l'utilisation de cette méthode :

```
MouseTrap.caughtYa(mice.get(i));
```

Créer une ArrayList consciente du type

Pas question toutefois de s'arrêter en si bon chemin. Une solution encore plus robuste consiste à créer une nouvelle classe utilisant une **ArrayList**, n'acceptant et ne produisant que des objets du type voulu :

```
//: c09:MouseList.java
// Une ArrayList consciente du type.
import java.util.*;

public class MouseList {
  private ArrayList list = new ArrayList();
  public void add(Mouse m) {
    list.add(m);
  }
  public Mouse get(int index) {
    return (Mouse)list.get(index);
  }
  public int size() { return list.size(); }
} ///:~
```

Voici un test pour le nouveau conteneur :

```
//: c09:MouseListTest.java
public class MouseListTest {
  public static void main(String[] args) {
    MouseList mice = new MouseList();
    for(int i = 0; i < 3; i++)
      mice.add(new Mouse(i));
    for(int i = 0; i < mice.size(); i++)
      MouseTrap.caughtYa(mice.get(i));
  }
} ///:~
```

Cet exemple est similaire au précédent, sauf que la nouvelle classe **MouseList** dispose d'un membre **private** de type **ArrayList**, et de méthodes identiques à celles fournies par **ArrayList**. Cependant, ces méthodes n'acceptent et ne produisent pas des **Object**s génériques, mais seulement des objets **Mouse**.

Notez que si **MouseList** avait été *dérivée* de **ArrayList**, la méthode **add(Mouse)** aurait simplement surchargé la méthode existante **add(Object)** et aucune restriction sur le type d'objets acceptés n'aurait donc été ajoutée. La **MouseList** est donc un *substitut* à l'**ArrayList**, réalisant certaines opérations avant de déléguer la responsabilité (cf. *Thinking in Patterns with Java*, téléchargeable sur *www.BruceEckel.com*).

Du fait qu'une **MouseList** n'accepte qu'une **Mouse**, l'instruction suivante :

```
mice.add(new Pigeon());
```

provoque un message d'erreur durant la *phase de compilation*. Cette approche, bien que plus fastidieuse du point de vue du code, signalera immédiatement si on utilise un type de façon incorrecte.

Notez aussi qu'aucun transtypage n'est nécessaire lors d'un appel à **get()** - elle renvoie toujours une **Mouse**.

Types paramétrés

Ce type de problème n'est pas isolé - nombreux sont les cas dans lesquels on a besoin de créer de nouveaux types basés sur d'autres types, et dans lesquels il serait bon de récupérer des informations de type lors de la phase de compilation. C'est le concept des *types paramétrés*. En C++, ceci est directement supporté par le langage via les *templates*. Les futures versions de Java supporteront probablement une implémentation des types paramétrés ; actuellement il faut se contenter de créer des classes similaires à **MouseList**.

Itérateurs

Chaque classe conteneur fournit des méthodes pour stocker des objets et pour les extraire - après tout, le but d'un conteneur est de stocker des choses. Dans **ArrayList**, on insère des objets via la méthode **add()**, et **get()** est l'*un* des moyens de récupérer ces objets. **ArrayList** est relativement souple - il est possible de sélectionner n'importe quel élément à n'importe quel moment, ou de sélectionner plusieurs éléments en même temps en utilisant différents index.

Si on se place à un niveau d'abstraction supérieur, on s'aperçoit d'un inconvénient : on a besoin de connaître le type exact du conteneur afin de l'utiliser. Ceci peut sembler bénin à première vue, mais qu'en est-il si on commence à programmer en utilisant une **ArrayList**, et qu'on se rende compte par la suite qu'il serait plus efficace d'utiliser une **LinkedList** à la place ? Ou alors si on veut écrire une portion de code générique qui ne connait pas le type de conteneur avec lequel elle travaille, afin de pouvoir être utilisé avec différents types de conteneurs sans avoir à réécrire ce code ?

Le concept d'*itérateur* peut être utilisé pour réaliser cette abstraction. Un itérateur est un objet dont le travail est de se déplacer dans une séquence d'objets et de sélectionner chaque objet de cette séquence sans que le programmeur client n'ait à se soucier de la structure sous-jacente de cette séquence. De plus, un itérateur est généralement ce qu'il est convenu d'appeler un objet « léger » : un objet bon marché à construire. Pour cette raison, vous trouverez souvent des contraintes étranges sur les itérateurs ; par exemple, certains itérateurs ne peuvent se déplacer que dans un sens.

L'**Iterator** Java est l'exemple type d'un itérateur avec ce genre de contraintes. On ne peut faire grand-chose avec mis à part :

1. Demander à un conteneur de renvoyer un **Iterator** en utilisant une méthode appelée **iterator()**. Cet **Iterator** sera prêt à renvoyer le premier élément dans la séquence au premier appel à sa méthode **next()**.
2. Récupérer l'objet suivant dans la séquence grâce à sa méthode **next()**.
3. Vérifier s'il reste *encore* d'autres objets dans la séquence via la méthode **hasNext()**.

4. Enlever le dernier élément renvoyé par l'itérateur avec la méthode **remove()**.

Et c'est tout. C'est une implémentation simple d'un itérateur, mais néanmoins puissante (et il existe un **ListIterator** plus sophistiqué pour les **List**s). Pour le voir en action, revisitons le programme **CatsAndDogs.java** rencontré précédemment dans ce chapitre. Dans sa version originale, la méthode **get()** était utilisée pour sélectionner chaque élément, mais la version modifiée suivante se sert d'un **Iterator** :

```
//: c09:CatsAndDogs2.java
// Conteneur simple utilisant un Iterator.
import java.util.*;

public class CatsAndDogs2 {
  public static void main(String[] args) {
    ArrayList cats = new ArrayList();
    for(int i = 0; i < 7; i++)
      cats.add(new Cat(i));
    Iterator e = cats.iterator();
    while(e.hasNext())
      ((Cat)e.next()).print();
  }
} ///:~
```

Les dernières lignes utilisent maintenant un **Iterator** pour se déplacer dans la séquence à la place d'une boucle **for**. Avec l'**Iterator**, on n'a pas besoin de se soucier du nombre d'éléments dans le conteneur. Cela est géré via les méthodes **hasNext()** et **next()**.

Comme autre exemple, considérons maintenant la création d'une méthode générique d'impression :

```
//: c09:HamsterMaze.java
// Utilisation d'un Iterator.
import java.util.*;

class Hamster {
  private int hamsterNumber;
  Hamster(int i) { hamsterNumber = i; }
  public String toString() {
    return "This is Hamster #" + hamsterNumber;
  }
}

class Printer {
  static void printAll(Iterator e) {
    while(e.hasNext())
      System.out.println(e.next());
  }
}

public class HamsterMaze {
```

```
  public static void main(String[] args) {
    ArrayList v = new ArrayList();
    for(int i = 0; i < 3; i++)
      v.add(new Hamster(i));
    Printer.printAll(v.iterator());
  }
} ///:~
```

Examinez attentivement la méthode **printAll()** et notez qu'elle ne dispose d'aucune information sur le type de séquence. Tout ce dont elle dispose est un **Iterator**, et c'est la seule chose dont elle a besoin pour utiliser la séquence : elle peut récupérer l'objet suivant, et savoir si elle se trouve à la fin de la séquence. Cette idée de prendre un conteneur d'objets et de le parcourir pour réaliser une opération sur chaque élément est un concept puissant, et on le retrouvera tout au long de ce livre.

Cet exemple est même encore plus générique, puisqu'il utilise implicitement la méthode **Object.toString()**. La méthode **println()** est surchargée pour tous les types scalaires ainsi que dans **Object** ; dans chaque cas une **String** est automatiquement produite en appelant la méthode **toString()** appropriée.

Bien que cela ne soit pas nécessaire, on pourrait être plus explicite en transtypant le résultat, ce qui aurait pour effet d'appeler **toString()** :

```
System.out.println((String)e.next());
```

En général, cependant, on voudra certainement aller au-delà de l'appel des méthodes de la classe **Object**, et on se heurte de nouveau au problème du transtypage. Il faut donc assumer qu'on a récupéré un **Iterator** sur une séquence contenant des objets du type particulier qui nous intéresse, et transtyper les objets dans ce type (et recevoir une exception à l'exécution si on se trompe).

Récursion indésirable

Comme les conteneurs standard Java héritent de la classe **Object** (comme toutes les autres classes), ils contiennent une méthode **toString()**. Celle-ci a été redéfinie afin de produire une représentation **String** d'eux-mêmes, incluant les objets qu'ils contiennent. A l'intérieur d'**ArrayList**, la méthode **toString()** parcourt les éléments de l'**ArrayList** et appelle **toString()** pour chacun d'eux. Supposons qu'on veuille afficher l'adresse de l'instance. Il semble raisonnable de se référer à **this** (en particulier pour les programmeurs C++ qui sont habitués à cette approche) :

```
//: c09:InfiniteRecursion.java
// Récursion accidentelle.
import java.util.*;

public class InfiniteRecursion {
  public String toString() {
    return " InfiniteRecursion address: "
      + this + "\n";
  }
  public static void main(String[] args) {
```

```
    ArrayList v = new ArrayList();
    for(int i = 0; i < 10; i++)
      v.add(new InfiniteRecursion());
    System.out.println(v);
  }
} ///:~
```

Si on crée un objet **InfiniteRecursion** et qu'on veut l'afficher, on se retrouve avec une séquence infinie d'exceptions. C'est également vrai si on place des objets **InfiniteRecursion** dans une **ArrayList** et qu'on imprime cette **ArrayList** comme c'est le cas ici. Ceci est du à la conversion automatique de type sur les **String**s. Quand on écrit :

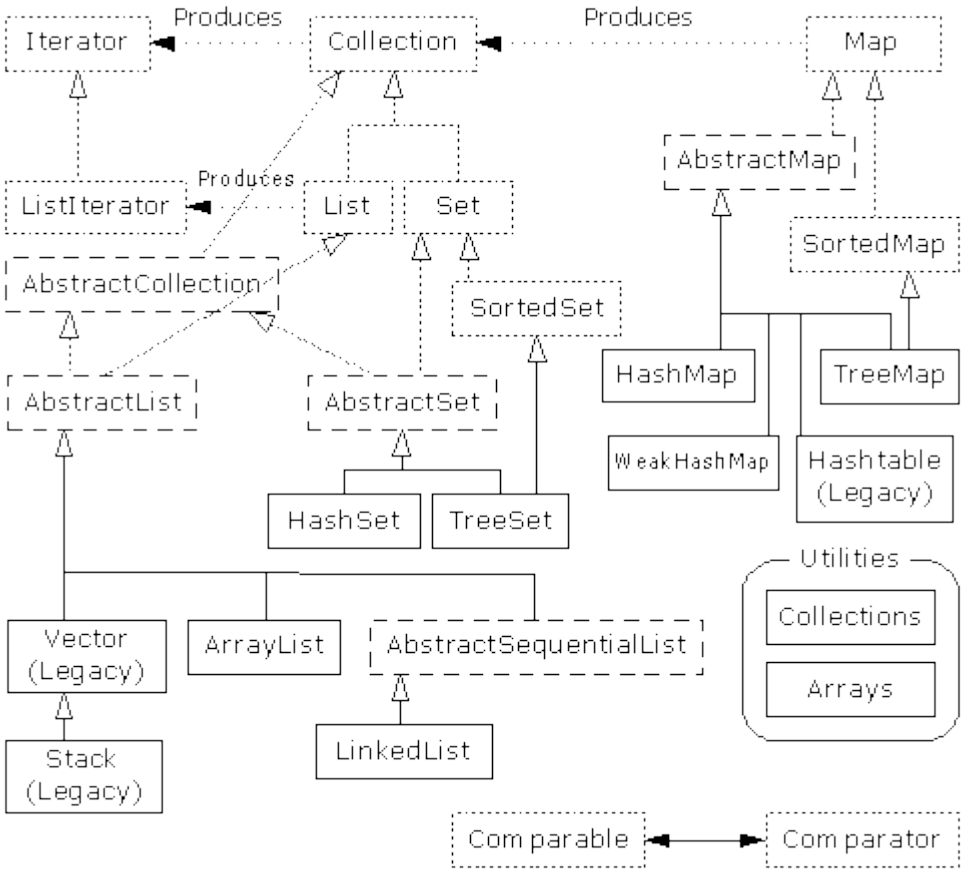
```
"InfiniteRecursion address: " + this
```

Le compilateur voit une **String** suivie par un « + » et quelque chose qui n'est pas une **String**, il essaie donc de convertir **this** en **String**. Il réalise cette conversion en appelant **toString()**, ce qui produit un appel récursif.

Si on veut réellement imprimer l'adresse de l'objet dans ce cas, la solution est d'appeler la méthode **Object.toString()**, qui réalise exactement ceci. Il faut donc utiliser **super.toString()** à la place de **this** (ceci ne fonctionnera que si on hérite directement de la classe **Object**, ou si aucune classe parent n'a redéfini la méthode **toString()**).

Classification des conteneurs

Les **Collection**s et les **Map**s peuvent être implémentés de différentes manières, à vous de choisir la bonne selon vos besoins. Le diagramme suivant peut aider à s'y retrouver parmi les conteneurs Java 2 :



Ce diagramme peut sembler un peu surchargé à première vue, mais il n'y a en fait que trois types conteneurs de base : les **Map**s, les **List**s et les **Set**s, chacun d'entre eux ne proposant que deux ou trois implémentations (avec typiquement une version préférée). Quand on se ramène à cette observation, les conteneurs ne sont plus aussi intimidants.

Les boîtes en pointillé représentent les **interfaces**, les boîtes en tirets représentent des classes **abstract**, et les boîtes pleines sont des classes normales (concrètes). Les lignes pointillées indiquent qu'une classe particulière implémente une **interface** (ou dans le cas d'une classe **abstract**, implémente partiellement cette **interface**). Une flèche pleine indique qu'une classe peut produire des objets de la classe sur laquelle la flèche pointe. Par exemple, une **Collection** peut produire un **Iterator**, tandis qu'une **List** peut produire un **ListIterator** (ainsi qu'un **Iterator** ordinaire, puisque **List** est dérivée de **Collection**).

Les interfaces concernées par le stockage des objets sont **Collection**, **List**, **Set** et **Map**. Idéalement, la majorité du code qu'on écrit sera destinée à ces interfaces, et le seul endroit où on spécifiera le type précis utilisé est lors de la création. On pourra donc créer une **List** de cette manière :

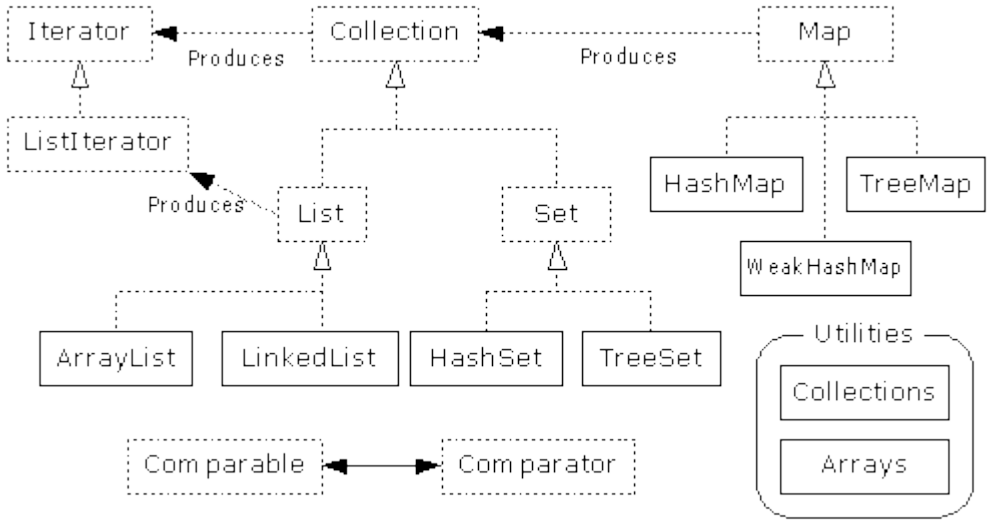
```
List x = new LinkedList();
```

```
List x = new ArrayList();
```

Et on ne touche pas au reste du code (une telle généricité peut aussi être réalisée via des itérateurs).

Dans la hiérarchie de classes, on peut voir un certain nombre de classes dont le nom débute par « **Abstract** », ce qui peut paraître un peu déroutant au premier abord. Ce sont simplement des outils qui implémentent partiellement une interface particulière. Si on voulait réaliser notre propre **Set**, par exemple, il serait plus simple de dériver **AbstractSet** et de réaliser le travail minimum pour créer la nouvelle classe, plutôt que d'implémenter l'interface **Set** et toutes les méthodes qui vont avec. Cependant, la bibliothèque de conteneurs possède assez de fonctionnalités pour satisfaire quasiment tous nos besoins. De notre point de vue, nous pouvons donc ignorer les classes débutant par « **Abstract** ».

Ainsi, lorsqu'on regarde le diagramme, on n'est réellement concerné que par les **interface**s du haut du diagramme et les classes concrètes (celles qui sont entourées par des boîtes solides). Typiquement, on se contentera de créer un objet d'une classe concrète, de la transtyper dans son **interface** correspondante, et ensuite utiliser cette **interface** tout au long du code. De plus, on n'a pas besoin de se préoccuper des éléments pré-existants lorsqu'on produit du nouveau code. Le diagramme peut donc être grandement simplifié pour ressembler à ceci :



Il n'inclut plus maintenant que les classes et les interfaces que vous serez amenés à rencontrer régulièrement, ainsi que les éléments sur lesquels nous allons nous pencher dans ce chapitre.

Voici un exemple simple, qui remplit une **Collection** (représenté ici par une **ArrayList**) avec des objets **String**, et affiche ensuite chaque élément de la **Collection** :

```
//: c09:SimpleCollection.java
// Un exemple simple d'utilisation des Collections Java 2.
import java.util.*;

public class SimpleCollection {
  public static void main(String[] args) {
    // Transtypage ascendant parce qu'on veut juste
    // travailler avec les fonctionnalités d'une Collection
    Collection c = new ArrayList();
    for(int i = 0; i < 10; i++)
      c.add(Integer.toString(i));
    Iterator it = c.iterator();
```

```
    while(it.hasNext())
      System.out.println(it.next());
  }
} ///:~
```

La première ligne de **main()** crée un objet **ArrayList** et le transtype ensuite en une **Collection**. Puisque cet exemple n'utilise que les méthodes de **Collection**, tout objet d'une classe dérivée de **Collection** fonctionnerait, mais l'**ArrayList** est la **Collection** à tout faire typique.

La méthode **add()**, comme son nom le suggère, ajoute un nouvel élément dans la **Collection**. En fait, la documentation précise bien que **add()** « assure que le conteneur contiendra l'élément spécifié ». Cette précision concerne les **Set**s, qui n'ajoutent un élément que s'il n'est pas déjà présent. Avec une **ArrayList**, ou n'importe quel type de **List**, **add()** veut toujours dire « stocker dans », parce qu'une **List** se moque de contenir des doublons.

Toutes les **Collection**s peuvent produire un **Iterator** grâce à leur méthode **iterator()**. Ici, un **Iterator** est créé et utilisé pour traverser la **Collection**, en affichant chaque élément.

Fonctionnalités des Collections

La table suivante contient toutes les opérations définies pour une **Collection** (sans inclure les méthodes directement héritées de la classe **Object**), et donc pour un **Set** ou une **List** (les **List**s possèdent aussi d'autres fonctionnalités). Les **Map**s n'héritant pas de **Collection**, elles seront traitées séparément.

boolean add(Object)	Assure que le conteneur stocke l'argument. Renvoie **false** si elle n'ajoute pas l'argument (c'est une méthode « optionnelle », décrite plus tard dans ce chapitre).
boolean addAll(Collection)	Ajoute tous les éléments de l'argument. Renvoie **true** si un élément a été ajouté (« optionnelle »).
void clear()	Supprime tous les éléments du conteneur (« optionnelle »).
boolean contains(Object)	**true** si le conteneur contient l'argument.
boolean containsAll(Collection)	**true** si le conteneur contient tous les éléments de l'argument.
boolean isEmpty()	**true** si le conteneur ne contient pas d'éléments.
Iterator iterator()	Renvoie un **Iterator** qu'on peut utiliser pour parcourir les éléments du conteneur.
boolean remove(Object)	Si l'argument est dans le conteneur, une instance de cet élément est enlevée. Renvoie **true** si c'est le cas (« optionnelle »).
boolean removeAll(Collection)	Supprime tous les éléments contenus dans l'argument. Renvoie **true** si au moins une suppression a été effectuée (« optionnelle »).
boolean retainAll(Collection)	Ne garde que les éléments contenus dans l'argument (une « intersection » selon la théorie des ensembles). Renvoie **true** s'il y a eu un changement (« optionnelle »).

int size()	Renvoie le nombre d'éléments dans le conteneur.
Object[] toArray()	Renvoie un tableau contenant tous les éléments du conteneur.
Object[] toArray(Object[] a)	Renvoie un tableau contenant tous les éléments du conteneur, dont le type est celui du tableau **a** au lieu d'**Object**s génériques (il faudra toutefois transtyper le tableau dans son type correct).

Notez qu'il n'existe pas de fonction **get()** permettant un accès aléatoire. Ceci parce que les **Collection**s contiennent aussi les **Set**s, qui maintiennent leur propre ordre interne, faisant de toute tentative d'accès aléatoire un non-sens. Il faut donc utiliser un **Iterator** pour parcourir tous les éléments d'une **Collection** ; c'est la seule façon de récupérer les objets stockés.

L'exemple suivant illustre toutes ces méthodes. Encore une fois, cet exemple marcherait avec tout objet héritant de **Collection**, mais nous utilisons ici une **ArrayList** comme « plus petit dénominateur commun » :

```
//: c09:Collection1.java
// Opérations disponibles sur les Collections.
import java.util.*;
import com.bruceeckel.util.*;

public class Collection1 {
  public static void main(String[] args) {
    Collection c = new ArrayList();
    Collections2.fill(c,
      Collections2.countries, 10);
    c.add("ten");
    c.add("eleven");
    System.out.println(c);
    // Crée un tableau à partir de la List :
    Object[] array = c.toArray();
    // Crée un tableau de Strings à partir de la List :
    String[] str =
      (String[])c.toArray(new String[1]);
    // Trouve les éléments mini et maxi ; ceci peut
    // signifier différentes choses suivant la manière
    // dont l'interface Comparable est implémentée :
    System.out.println("Collections.max(c) = " +
      Collections.max(c));
    System.out.println("Collections.min(c) = " +
      Collections.min(c));
    // Ajoute une Collection à une autre Collection
    Collection c2 = new ArrayList();
    Collections2.fill(c2,
      Collections2.countries, 10);
    c.addAll(c2);
    System.out.println(c);
```

```
    c.remove(CountryCapitals.pairs[0][0]);
    System.out.println(c);
    c.remove(CountryCapitals.pairs[1][0]);
    System.out.println(c);
    // Supprime tous les éléments
    // de la Collection argument :
    c.removeAll(c2);
    System.out.println(c);
    c.addAll(c2);
    System.out.println(c);
    // Est-ce qu'un élément est dans la Collection ?
    String val = CountryCapitals.pairs[3][0];
    System.out.println(
      "c.contains(" + val  + ") = "
      + c.contains(val));
    // Est-ce qu'une Collection est contenue dans la Collection ?
    System.out.println(
      "c.containsAll(c2) = "+ c.containsAll(c2));
    Collection c3 = ((List)c).subList(3, 5);
    // Garde les éléments présents à la fois dans
    // c2 et c3 (intersection d'ensembles) :
    c2.retainAll(c3);
    System.out.println(c);
    // Supprime tous les éléments
    // de c2 contenus dans c3 :
    c2.removeAll(c3);
    System.out.println("c.isEmpty() = " +
      c.isEmpty());
    c = new ArrayList();
    Collections2.fill(c,
      Collections2.countries, 10);
    System.out.println(c);
    c.clear(); // Supprime tous les éléments
    System.out.println("after c.clear():");
    System.out.println(c);
  }
} ///:~
```

Les **ArrayList**s sont créées et initialisées avec différents ensembles de données, puis transtypées en objets **Collection** ; il est donc clair que seules les fonctions de l'interface **Collection** sont utilisées. **main()** réalise de simples opérations pour illustrer toutes les méthodes de **Collection**.

Les sections suivantes décrivent les diverses implémentations des **List**s, **Set**s et **Map**s et indiquent dans chaque cas (à l'aide d'une astérisque) laquelle devrait être votre choix par défaut. Vous noterez que les classes pré-existantes **Vector**, **Stack** et **Hashtable** ne sont *pas* incluses car certains conteneurs Java 2 fournissent les mêmes fonctionnalités.

Fonctionnalités des Lists

La **List** de base est relativement simple à utiliser, comme vous avez pu le constater jusqu'à présent avec les **ArrayList**s. Mis à part les méthodes courantes **add()** pour insérer des objets, **get()** pour les retrouver un par un, et **iterator()** pour obtenir un **Iterator** sur la séquence, les listes possèdent par ailleurs tout un ensemble de méthodes qui peuvent se révéler très pratiques.

Les **List**s sont déclinées en deux versions : l'**ArrayList** de base, qui excelle dans les accès aléatoires aux éléments, et la **LinkedList**, bien plus puissante (qui n'a pas été conçue pour un accès aléatoire optimisé, mais dispose d'un ensemble de méthodes bien plus conséquent).

List (interface)	L'ordre est la caractéristique la plus importante d'une **List** ; elle garantit de maintenir les éléments dans un ordre particulier. Les **List**s disposent de méthodes supplémentaires permettant l'insertion et la suppression d'éléments au sein d'une **List** (ceci n'est toutefois recommandé que pour une **LinkedList**). Une **List** produit des **ListIterator**s, qui permettent de parcourir la **List** dans les deux directions, d'insérer et de supprimer des éléments au sein de la **List**.
ArrayList*	Une **List** implémentée avec un tableau. Permet un accès aléatoire instantané aux éléments, mais se révèle inefficace lorsqu'on insère ou supprime un élément au milieu de la liste. Le **ListIterator** ne devrait être utilisé que pour parcourir l'**ArrayList** dans les deux sens, et non pour l'insertion et la suppression d'éléments, opérations coûteuses comparées aux **LinkedList**s.
LinkedList	Fournit un accès séquentiel optimal, avec des coûts d'insertion et de suppression d'éléments au sein de la **List** négligeables. Relativement lente pour l'accès aléatoire (préférer une **ArrayList** pour cela). Fournit aussi les méthodes **addFirst()**, **addLast()**, **getFirst()**, **getLast()**, **removeFirst()** et **removeLast()** (qui ne sont définies dans aucune interface ou classe de base) afin de pouvoir l'utiliser comme une pile, une file (une queue) ou une file double (queue à double entrée).

Les méthodes dans l'exemple suivant couvrent chacune un groupe de fonctionnalités : les opérations disponibles pour toutes les listes (**basicTest()**), le déplacement dans une liste avec un **Iterator** (**iterMotion()**) ainsi que la modification dans une liste avec un **Iterator** (**iterManipulation()**), la visualisation des manipulations sur la **List** (**testVisual()**) et les opérations disponibles uniquement pour les **LinkedList**s.

```
//: c09:List1.java
// Opérations disponibles sur les Lists.
import java.util.*;
import com.bruceeckel.util.*;

public class List1 {
  public static List fill(List a) {
    Collections2.countries.reset();
    Collections2.fill(a,
      Collections2.countries, 10);
    return a;
  }
```

```
  static boolean b;
  static Object o;
  static int i;
  static Iterator it;
  static ListIterator lit;
  public static void basicTest(List a) {
    a.add(1, "x"); // Ajout à l'emplacement 1
    a.add("x"); // Ajout à la fin
    // Ajout d'une Collection :
    a.addAll(fill(new ArrayList()));
    // Ajout d'une Collection à partir du 3ème élément :
    a.addAll(3, fill(new ArrayList()));
    b = a.contains("1"); // L'élément est-il présent ?
    // La Collection entière est-elle présente ?
    b = a.containsAll(fill(new ArrayList()));
    // Les Lists permettent un accès aléatoire aux éléments,
    // bon marché pour les ArrayLists, coûteux pour les LinkedLists :
    o = a.get(1); // Récupère l'objet du premier emplacement
    i = a.indexOf("1"); // Donne l'index de l'objet
    b = a.isEmpty(); // La List contient-elle des éléments ?
    it = a.iterator(); // Iterator de base
    lit = a.listIterator(); // ListIterator
    lit = a.listIterator(3); // Démarre au 3ème élément
    i = a.lastIndexOf("1"); // Dernière concordance
    a.remove(1); // Supprime le premier élément
    a.remove("3"); // Supprime cet objet
    a.set(1, "y"); // Positionne le premier élément à "y"
    // Garde tous les éléments présents dans l'argument
    // (intersection de deux ensembles) :
    a.retainAll(fill(new ArrayList()));
    // Supprime tous les éléments présents dans l'argument :
    a.removeAll(fill(new ArrayList()));
    i = a.size(); // Taille de la List ?
    a.clear(); // Supprime tous les éléments
  }
  public static void iterMotion(List a) {
    ListIterator it = a.listIterator();
    b = it.hasNext();
    b = it.hasPrevious();
    o = it.next();
    i = it.nextIndex();
    o = it.previous();
    i = it.previousIndex();
  }
  public static void iterManipulation(List a) {
    ListIterator it = a.listIterator();
    it.add("47");
```

```
    // Doit aller sur un élément après add() :
    it.next();
    // Supprime l'élément qui vient d'être produit :
    it.remove();
    // Doit aller sur un élément après remove() :
    it.next();
    // Change l'élément qui vient d'être produit :
    it.set("47");
  }
  public static void testVisual(List a) {
    System.out.println(a);
    List b = new ArrayList();
    fill(b);
    System.out.print("b = ");
    System.out.println(b);
    a.addAll(b);
    a.addAll(fill(new ArrayList()));
    System.out.println(a);
    // Insère, supprime et remplace des éléments
    // en utilisant un ListIterator :
    ListIterator x = a.listIterator(a.size()/2);
    x.add("one");
    System.out.println(a);
    System.out.println(x.next());
    x.remove();
    System.out.println(x.next());
    x.set("47");
    System.out.println(a);
    // Traverse la liste à l'envers :
    x = a.listIterator(a.size());
    while(x.hasPrevious())
      System.out.print(x.previous() + " ");
    System.out.println();
    System.out.println("testVisual finished");
  }
  // Certaines opérations ne sont disponibles
  // que pour des LinkedLists :
  public static void testLinkedList() {
    LinkedList ll = new LinkedList();
    fill(ll);
    System.out.println(ll);
    // Utilisation comme une pile, insertion (push) :
    ll.addFirst("one");
    ll.addFirst("two");
    System.out.println(ll);
    // Utilisation comme une pile, récupération de la valeur du premier élément (peek) :
    System.out.println(ll.getFirst());
```

```
    // Utilisation comme une pile, suppression (pop) :
    System.out.println(ll.removeFirst());
    System.out.println(ll.removeFirst());
    // Utilisation comme une file, en retirant les
    // éléments à la fin de la liste :
    System.out.println(ll.removeLast());
    // Avec les opérations ci-dessus, c'est une file double !
    System.out.println(ll);
  }
  public static void main(String[] args) {
    // Crée et remplit une nouvelle List à chaque fois :
    basicTest(fill(new LinkedList()));
    basicTest(fill(new ArrayList()));
    iterMotion(fill(new LinkedList()));
    iterMotion(fill(new ArrayList()));
    iterManipulation(fill(new LinkedList()));
    iterManipulation(fill(new ArrayList()));
    testVisual(fill(new LinkedList()));
    testLinkedList();
  }
} ///:~
```

Réaliser une pile à partir d'une LinkedList

Une pile est un conteneur « dernier arrivé, premier sorti » (LIFO - « Last In, First Out »). C'est à dire que l'objet qu'on « pousse » (« push »)sur la pile en dernier sera le premier accessible lors d'une extraction (« pop »). Comme tous les autres conteneurs de Java, on stocke et récupère des **Object**s, qu'il faudra donc retranstyper après leur extraction, à moins qu'on ne se contente des fonctionnalités de la classe **Object**.

La classe **LinkedList** possède des méthodes qui implémentent directement les fonctionnalités d'une pile, on peut donc utiliser directement une **LinkedList** plutôt que de créer une classe implémentant une pile. Cependant une classe est souvent plus explicite :

```
//: c09:StackL.java
// Réaliser une pile à partir d'une LinkedList.
import java.util.*;
import com.bruceeckel.util.*;

public class StackL {
  private LinkedList list = new LinkedList();
  public void push(Object v) {
    list.addFirst(v);
  }
  public Object top() { return list.getFirst(); }
  public Object pop() {
    return list.removeFirst();
  }
```

```
  public static void main(String[] args) {
    StackL stack = new StackL();
    for(int i = 0; i < 10; i++)
      stack.push(Collections2.countries.next());
    System.out.println(stack.top());
    System.out.println(stack.top());
    System.out.println(stack.pop());
    System.out.println(stack.pop());
    System.out.println(stack.pop());
  }
} ///:~
```

L'héritage n'est pas approprié ici puisqu'il produirait une classe contenant toutes les méthodes d'une **LinkedList** (on verra que cette erreur a déjà été faite par les concepteurs de la bibliothèque Java 1.0 avec la classe **Stack**).

Réaliser une file à partir d'une LinkedList

Une file (ou queue) est un conteneur « premier arrivé, premier sorti » (FIFO - « First In, First Out »). C'est à dire qu'on « pousse » des objets à une extrémité et qu'on les en retire à l'autre extrémité. L'ordre dans lequel on pousse les objets sera donc le même que l'ordre dans lequel on les récupérera. La classe **LinkedList** possède des méthodes qui implémentent directement les fonctionnalités d'une file, on peut donc les utiliser directement dans une classe **Queue** :

```
//: c09:Queue.java
// Réaliser une file à partir d'une LinkedList.
import java.util.*;

public class Queue {
  private LinkedList list = new LinkedList();
  public void put(Object v) { list.addFirst(v); }
  public Object get() {
    return list.removeLast();
  }
  public boolean isEmpty() {
    return list.isEmpty();
  }
  public static void main(String[] args) {
    Queue queue = new Queue();
    for(int i = 0; i < 10; i++)
      queue.put(Integer.toString(i));
    while(!queue.isEmpty())
      System.out.println(queue.get());
  }
} ///:~
```

Il est aussi facile de créer une file double (queue à double entrée) à partir d'une **LinkedList**. Une file double est une file à laquelle on peut ajouter et supprimer des éléments à chacune de ses

extrémités.

Fonctionnalités des Sets

Les **Set**s ont exactement la même interface que les **Collection**s, et à l'inverse des deux différentes **List**s, ils ne proposent aucune fonctionnalité supplémentaire. Les **Set**s sont donc juste une **Collection** ayant un comportement particulier (implémenter un comportement différent constitue l'exemple type où il faut utiliser l'héritage et le polymorphisme). Un **Set** refuse de contenir plus d'une instance de chaque valeur d'un objet (savoir ce qu'est la « valeur » d'un objet est plus compliqué, comme nous allons le voir).

Set (interface)	Chaque élément ajouté au **Set** doit être unique ; sinon le **Set** n'ajoutera pas le doublon. Les **Object**s ajoutés à un **Set** doivent définir la méthode **equals()** pour pouvoir établir l'unicité de l'objet. Un **Set** possède la même interface qu'une **Collection**. L'interface **Set** ne garantit pas qu'il maintiendra les éléments dans un ordre particulier.
HashSet*	Pour les **Set**s où le temps d'accès aux éléments est primordial. Les **Object**s doivent définir la méthode **hashCode()**.
TreeSet	Un **Set** trié stocké dans un arbre. De cette manière, on peut extraire une séquence triée à partir du **Set**.

L'exemple suivant ne montre *pas* tout ce qu'il est possible de faire avec un **Set**, puisque l'interface est la même que pour les **Collection**s, et comme telle a déjà été testée dans l'exemple précédent. Par contre, il illustre les comportements qui rendent un **Set** particulier :

```
//: c09:Set1.java
// Opérations disponibles pour les Sets.
import java.util.*;
import com.bruceeckel.util.*;

public class Set1 {
  static Collections2.StringGenerator gen =
    Collections2.countries;
  public static void testVisual(Set a) {
    Collections2.fill(a, gen.reset(), 10);
    Collections2.fill(a, gen.reset(), 10);
    Collections2.fill(a, gen.reset(), 10);
    System.out.println(a); // Pas de doublons !
    // Ajoute un autre ensemble à celui-ci :
    a.addAll(a);
    a.add("one");
    a.add("one");
    a.add("one");
    System.out.println(a);
    // Extraction d'item :
    System.out.println("a.contains(\"one\"): " +
      a.contains("one"));
```

```
  }
  public static void main(String[] args) {
    System.out.println("HashSet");
    testVisual(new HashSet());
    System.out.println("TreeSet");
    testVisual(new TreeSet());
  }
} ///:~
```

Cet exemple tente d'ajouter des valeurs dupliquées au **Set**, mais lorsqu'on l'imprime on voit que le **Set** n'accepte qu'une instance de chaque valeur.

Lorsqu'on lance ce programme, on voit que l'ordre interne maintenu par le **HashSet** est différent de celui de **TreeSet**, puisque chacune de ces implémentations stocke les éléments d'une manière différente (**TreeSet** garde les éléments triés, tandis que **HashSet** utilise une fonction de hachage, conçue spécialement pour des accès optimisés). Quand on crée un nouveau type, il faut bien se rappeler qu'un **Set** a besoin de maintenir un ordre de stockage, ce qui veut dire qu'il faut implémenter l'interface **Comparable** et définir la méthode **compareTo()**. Voici un exemple :

```
//: c09:Set2.java
// Ajout d'un type particulier dans un Set.
import java.util.*;

class MyType implements Comparable {
  private int i;
  public MyType(int n) { i = n; }
  public boolean equals(Object o) {
    return
      (o instanceof MyType)
      && (i == ((MyType)o).i);
  }
  public int hashCode() { return i; }
  public String toString() { return i + " "; }
  public int compareTo(Object o) {
    int i2 = ((MyType)o).i;
    return (i2 < i ? -1 : (i2 == i ? 0 : 1));
  }
}

public class Set2 {
  public static Set fill(Set a, int size) {
    for(int i = 0; i < size; i++)
      a.add(new MyType(i));
    return a;
  }
  public static void test(Set a) {
    fill(a, 10);
    fill(a, 10); // Tente de créer des doublons
```

```
    fill(a, 10);
    a.addAll(fill(new TreeSet(), 10));
    System.out.println(a);
  }
  public static void main(String[] args) {
    test(new HashSet());
    test(new TreeSet());
  }
} ///:~
```

La forme que doivent avoir les définitions des méthodes **equals()** et **hashCode()** sera décrite plus tard dans ce chapitre. Il faut définir une méthode **equals()** pour les deux implémentations de **Set**, mais **hashCode()** n'est nécessaire que si la classe est placée dans un **hashSet** (ce qui est probable, puisqu'il s'agit de l'implémentation recommandée pour un **Set**). Cependant, c'est une bonne pratique de programmation de redéfinir **hashCode()** lorsqu'on redéfinit **equals()**. Ce processus sera examiné en détails plus loin dans ce chapitre.

Notez que je n'ai *pas* utilisé la forme « simple et évidente » **return i-i2** dans la méthode **compareTo()**. Bien que ce soit une erreur de programmation classique, elle ne fonctionne que si **i** et **i2** sont des **int**s « non signés » (si Java *disposait* d'un mot-clef « **unsigned** », ce qu'il n'a pas). Elle ne marche pas pour les **int**s signés de Java, qui ne sont pas assez grands pour représenter la différence de deux **int**s signés. Si **i** est un grand entier positif et **j** un grand entier négatif, **i-j** débordera et renverra une valeur négative, ce qui n'est pas le résultat attendu.

Sets triés : les SortedSets

Un **SortedSet** (dont **TreeSet** est l'unique représentant) garantit que ses éléments seront stockés triés, ce qui permet de proposer de nouvelles fonctionnalités grâce aux méthodes supplémentaires de l'interface **SortedSet** suivantes :

Comparator comparator() : Renvoie le **Comparator** utilisé pour ce **Set**, ou **null** dans le cas d'un tri naturel.

Object first() : Renvoie le plus petit élément.

Object last() : Renvoie le plus grand élément.

SortedSet subSet(fromElement, toElement) : Renvoie une vue du **Set** contenant les éléments allant de **fromElement** inclus à **toElement** exclu.

SortedSet headSet(toElement) : Renvoie une vue du **Set** contenant les éléments inférieurs à **toElement**.

SortedSet tailSet(fromElement): Renvoie une vue du **Set** contenant les éléments supérieurs ou égaux à **fromElement**.

Fonctionnalités des Maps

Une **ArrayList** permet de sélectionner des éléments dans une séquence d'objets en utilisant un nombre, elle associe donc des nombres à des objets. Mais qu'en est-il si on souhaite sélectionner des éléments d'une séquence en utilisant un autre critère ? Dans l'exemple d'une pile, son critère de sélection est « le dernier objet poussé sur la pile ». Un *tableau associatif*, ou *map*, ou *dictionnaire*

est une alternative particulièrement puissante de cette idée de « sélection dans une séquence ». Conceptuellement, cela ressemble à une **ArrayList**, mais au lieu de sélectionner un objet par un nombre, on le sélectionne en utilisant un *autre objet* ! Ce fonctionnement est d'une valeur inestimable dans un programme.

Le concept est illustré dans Java via l'interface **Map**. La méthode **put(Object key, Object value)** ajoute une valeur (la chose qu'on veut stocker), et l'associe à une clef (la chose grâce à laquelle on va retrouver la valeur). La méthode **get(Object key)** renvoie la valeur associée à la clef correspondante. Il est aussi possible de tester une **Map** pour voir si elle contient une certaine clef ou une certaine valeur avec les méthodes **containsKey()** et **containsValue()**.

La bibliothèque Java standard propose deux types de **Map**s : **HashMap** et **TreeMap**. Les deux implémentations ont la même interface (puisqu'elles implémentent toutes les deux **Map**), mais diffèrent sur un point particulier : les performances. Dans le cas d'un appel à **get()**, il est peu efficace de chercher dans une **ArrayList** (par exemple) pour trouver une clef. C'est là que le **HashMap** intervient. Au lieu d'effectuer une recherche lente sur la clef, il utilise une valeur spéciale appelée *code de hachage* (*hash code*). Le code de hachage est une façon d'extraire une partie de l'information de l'objet en question et de la convertir en un **int** « relativement unique ». Tous les objets Java peuvent produire un code de hachage, et **hashCode()** est une méthode de la classe racine **Object**. Un **HashMap** récupère le **hashCode()** de l'objet et l'utilise pour retrouver rapidement la clef. Le résultat en est une augmentation drastique des performances [50].

Map (interface)	Maintient des associations clef - valeur (des paires), afin de pouvoir accéder à une valeur en utilisant une clef.
HashMap*	Implémentation basée sur une table de hachage (utilisez ceci à la place d'une **Hashtable**). Fournit des performances constantes pour l'insertion et l'extraction de paires. Les performances peuvent être ajustées via des constructeurs qui permettent de positionner la *capacité* et le *facteur de charge* de la table de hachage.
TreeMap	Implémentation basée sur un arbre rouge-noir. L'extraction des clefs ou des paires fournit une séquence triée (selon l'ordre spécifié par **Comparable** ou **Comparator**, comme nous le verrons plus loin). Le point important dans un **TreeMap** est qu'on récupère les résultats dans l'ordre. **TreeMap** est la seule **Map** disposant de la méthode **subMap()**, qui permet de renvoyer une portion de l'arbre.

Nous nous pencherons sur les mécanismes de hachage un peu plus loin. L'exemple suivant utilise la méthode **Collections2.fill()** et les ensembles de données définis précédemment :

```
//: c09:Map1.java
// Opérations disponibles pour les Maps.
import java.util.*;
import com.bruceeckel.util.*;

public class Map1 {
  static Collections2.StringPairGenerator geo =
    Collections2.geography;
  static Collections2.RandStringPairGenerator
```

```
    rsp = Collections2.rsp;
  // Produire un Set de clefs :
  public static void printKeys(Map m) {
    System.out.print("Size = " + m.size() +", ");
    System.out.print("Keys: ");
    System.out.println(m.keySet());
  }
  // Produire une Collection de valeurs :
  public static void printValues(Map m) {
    System.out.print("Values: ");
    System.out.println(m.values());
  }
  public static void test(Map m) {
    Collections2.fill(m, geo, 25);
    // Une Map a un comportement de « Set » pour les clefs :
    Collections2.fill(m, geo.reset(), 25);
    printKeys(m);
    printValues(m);
    System.out.println(m);
    String key = CountryCapitals.pairs[4][0];
    String value = CountryCapitals.pairs[4][1];
    System.out.println("m.containsKey(\"" + key +
      "\"): " + m.containsKey(key));
    System.out.println("m.get(\"" + key + "\"): "
      + m.get(key));
    System.out.println("m.containsValue(\""
      + value + "\"): " +
      m.containsValue(value));
    Map m2 = new TreeMap();
    Collections2.fill(m2, rsp, 25);
    m.putAll(m2);
    printKeys(m);
    key = m.keySet().iterator().next().toString();
    System.out.println("First key in map: "+key);
    m.remove(key);
    printKeys(m);
    m.clear();
    System.out.println("m.isEmpty(): "
      + m.isEmpty());
    Collections2.fill(m, geo.reset(), 25);
    // Les opérations sur le Set changent la Map :
    m.keySet().removeAll(m.keySet());
    System.out.println("m.isEmpty(): "
      + m.isEmpty());
  }
  public static void main(String[] args) {
    System.out.println("Testing HashMap");
```

```
    test(new HashMap());
    System.out.println("Testing TreeMap");
    test(new TreeMap());
  }
} ///:~
```

Les méthodes **printKeys()** et **printValues()** ne sont pas seulement des utilitaires pratiques, elles illustrent aussi comment produire une vue (sous la forme d'une **Collection**) d'une **Map**. La méthode **keySet()** renvoie un **Set** rempli par les clefs de la **Map**. La méthode **values()** renvoie quant à elle une **Collection** contenant toutes les valeurs de la **Map** (notez bien que les clefs doivent être uniques, alors que les valeurs peuvent contenir des doublons). Ces **Collection**s sont liées à la **Map**, tout changement effectué dans une **Collection** sera donc répercuté dans la **Map** associée.

Le reste du programme fournit des exemples simples pour chacune des opérations disponibles pour une **Map**, et teste les deux types de **Map**s.

Comme exemple d'utilisation d'un **HashMap**, considérons un programme vérifiant la nature aléatoire de la méthode **Math.random()** de Java. Idéalement, elle devrait produire une distribution parfaite de nombres aléatoires, mais pour tester cela il faut générer un ensemble de nombres aléatoires et compter ceux qui tombent dans les différentes plages. Un **HashMap** est parfait pour ce genre d'opérations, puisqu'il associe des objets à d'autres objets (dans ce cas, les objets valeurs contiennent le nombre produit par **Math.random()** ainsi que le nombre de fois où ce nombre apparaît) :

```
//: c09:Statistics.java
// Simple démonstration de l'utilisation d'un HashMap.
import java.util.*;

class Counter {
  int i = 1;
  public String toString() {
    return Integer.toString(i);
  }
}

class Statistics {
  public static void main(String[] args) {
    HashMap hm = new HashMap();
    for(int i = 0; i < 10000; i++) {
      // Produit un nombre entre 0 et 20 :
      Integer r =
        new Integer((int)(Math.random() * 20));
      if(hm.containsKey(r))
        ((Counter)hm.get(r)).i++;
      else
        hm.put(r, new Counter());
    }
    System.out.println(hm);
  }
```

```
} ///:~
```

Si la clef n'a pas déjà été stockée, la méthode **put()** insèrera une nouvelle paire clef - valeur dans le **HashMap**. Puisque **Counter** initialise automatiquement sa variable **i** à 1 lorsqu'elle est créée, cela indique une première occurence de ce nombre aléatoire particulier.

Pour afficher le **HashMap**, il est simplement imprimé. La méthode **toString()** de **HashMap** parcourt toutes les paires clef - valeur et appelle **toString()** pour chacune d'entre elles. La méthode **Integer.toString()** est prédéfinie, et on peut voir la méthode **toString()** de la classe **Counter**. La sortie du programme (après l'insertion de quelques retours chariot) ressemble à :

```
{19=526, 18=533, 17=460, 16=513, 15=521, 14=495,
13=512, 12=483, 11=488, 10=487, 9=514, 8=523,
7=497, 6=487, 5=480, 4=489, 3=509, 2=503, 1=475,
0=505}
```

On peut s'interroger sur la nécessité d'une classe **Counter**, qui ne semble même pas avoir les fonctionnalités de la classe d'encapsulation **Integer**. Pourquoi ne pas utiliser un **int** ou un **Integer** ? On ne peut utiliser un **int** puisque les conteneurs ne peuvent stocker que des références d'**Object**. On pourrait alors être tenté de se tourner vers les classes Java d'encapsulation des types primitifs. Cependant, ces classes ne permettent que de stocker une valeur initiale et de lire cette valeur. C'est à dire qu'il n'existe aucun moyen de changer cette valeur une fois qu'un objet d'encapsulation a été créé. Cela rend la classe **Integer** inutile pour résoudre notre problème, et nous force à créer une nouvelle classe qui statisfait l'ensemble de nos besoins.

Maps triées : les SortedMaps

Une **SortedMap** (dont **TreeMap** est l'unique représentant) garantit que ses éléments seront stockés triés selon leur clef, ce qui permet de proposer de nouvelles fonctionnalités grâce aux méthodes supplémentaires de l'interface **SortedMap** suivantes :

Comparator comparator() : Renvoie le **Comparator** utilisé pour cette **Map**, ou **null** dans le cas d'un tri naturel.

Object firstKey() : Renvoie la plus petite clef.

Object lastKey() : Renvoie la plus grande clef.

SortedMap subMap(fromKey, toKey) : Renvoie une vue de la **Map** contenant les paires dont les clefs vont de **fromKey** inclus à **toKey** exclu.

SortedMap headMap(toKey) : Renvoie une vue de la **Map** contenant les paires dont la clef est inférieure à **toKey**.

SortedMap tailMap(fromKey) : Renvoie une vue de la **Map** contenant les paires dont la clef est supérieure ou égale à **fromKey**.

Hachage et codes de hachage

Dans l'exemple précédent, une classe de la bibliothèque standard (**Integer**) était utilisée comme clef pour le **HashMap**. Cela ne pose pas de problèmes car elle dispose de tout ce qu'il faut pour fonctionner correctement comme une clef. Mais il existe un point d'achoppement classique

avec les **HashMap**s lorsqu'on crée une classe destinée à être utilisée comme clef. Considérons par exemple un système de prévision météorologique qui associe des objets **Groundhog** à des objets **Prediction**. Cela semble relativement simple : il suffit de créer deux classes, et d'utiliser **Groundhog** comme clef et **Prediction** comme valeur :

```
//: c09:SpringDetector.java
// Semble plausible, mais ne fonctionne pas.
import java.util.*;

class Groundhog {
  int ghNumber;
  Groundhog(int n) { ghNumber = n; }
}

class Prediction {
  boolean shadow = Math.random() > 0.5;
  public String toString() {
    if(shadow)
      return "Six more weeks of Winter!";
    else
      return "Early Spring!";
  }
}

public class SpringDetector {
  public static void main(String[] args) {
    HashMap hm = new HashMap();
    for(int i = 0; i < 10; i++)
      hm.put(new Groundhog(i), new Prediction());
    System.out.println("hm = " + hm + "\n");
    System.out.println(
      "Looking up prediction for Groundhog #3:");
    Groundhog gh = new Groundhog(3);
    if(hm.containsKey(gh))
      System.out.println((Prediction)hm.get(gh));
    else
      System.out.println("Key not found: " + gh);
  }
} ///:~
```

Chaque **Groundhog** se voit attribuer un numéro d'identité, afin de pouvoir récupérer une **Prediction** dans le **HashMap** en disant « Donne-moi la **Prediction** associée au **Groundhog** numéro 3 ». La classe **Prediction** contient un **boolean** initialisé en utilisant **Math.random()**, et une méthode **toString()** pour en interpréter la valeur. Dans **main()**, un **HashMap** est rempli avec des **Groundhog**s et leurs **Prediction**s associées. Le **HashMap** est affiché afin de voir qu'il a été correctement rempli. Un **Groundhog** avec une identité de 3 est alors utilisé comme clef pour extraire la prédiction du **Groundhog** numéro 3 (qui doit être dans le **HashMap**).

Tout ceci semble très simple, mais ne marche pas. Le problème vient du fait que **Groundhog**

hérite de la classe de base **Object** (ce qui est le comportement par défaut si aucune superclasse n'est précisée ; toutes les classes dérivent donc en fin de compte de la classe **Object**). C'est donc la méthode **hashCode()** de **Object** qui est utilisée pour générer le code de hachage pour chaque objet, et par défaut, celle-ci renvoie juste l'adresse de cet objet. La première instance de **Groundhog(3)** ne renvoie donc *pas* le même code de hachage que celui de la seconde instance de **Groundhog(3)** que nous avons tenté d'utiliser comme clef d'extraction.

On pourrait penser qu'il suffit de redéfinir **hashCode()**. Mais ceci ne fonctionnera toujours pas tant qu'on n'aura pas aussi redéfini la méthode **equals()** qui fait aussi partie de la classe **Object**. Cette méthode est utilisée par le **HashMap** lorsqu'il essaie de déterminer si la clef est égale à l'une des autres clefs de la table. Et la méthode par défaut **Object.equals()** compare simplement les adresses des objets, ce qui fait qu'un objet **Groundhog(3)** est différent d'un autre **Groundhog(3)**.

Pour utiliser un nouveau type comme clef dans un **HashMap**, il faut donc redéfinir les deux méthodes **hashCode()** et **equals()**, comme le montre la solution suivante :

```
//: c09:SpringDetector2.java
// Une classe utilisée comme clef dans un HashMap
// doit redéfinir hashCode() et equals().
import java.util.*;

class Groundhog2 {
  int ghNumber;
  Groundhog2(int n) { ghNumber = n; }
  public int hashCode() { return ghNumber; }
  public boolean equals(Object o) {
    return (o instanceof Groundhog2)
      && (ghNumber == ((Groundhog2)o).ghNumber);
  }
}

public class SpringDetector2 {
  public static void main(String[] args) {
    HashMap hm = new HashMap();
    for(int i = 0; i < 10; i++)
      hm.put(new Groundhog2(i),new Prediction());
    System.out.println("hm = " + hm + "\n");
    System.out.println(
      "Looking up prediction for groundhog #3:");
    Groundhog2 gh = new Groundhog2(3);
    if(hm.containsKey(gh))
      System.out.println((Prediction)hm.get(gh));
  }
} ///:~
```

Notez que cet exemple utilise la classe **Prediction** de l'exemple précédent, donc **SpringDetector.java** doit déjà avoir été compilé ou vous aurez une erreur lorsque vous tenterez de compiler **SpringDetector2.java**.

Groundhog2.hashCode() renvoie le numéro de marmotte comme identifiant. Dans cet

exemple, le programmeur doit s'assurer que deux marmottes ne portent pas le même identifiant. La méthode **hashCode()** n'est pas obligée de renvoyer un identifiant unique (vous comprendrez mieux ceci plus tard dans ce chapitre), mais la méthode **equals()** doit être capable de déterminer si deux objets sont strictement équivalents.

Bien que la méthode **equals()** semble ne vérifier que si l'argument est bien une instance de **Groundhog2** (en utilisant le mot-clef **instanceof**, expliqué plus en détails dans le Chapitre 12), **instanceof** effectue en fait implicitement un deuxième contrôle puisqu'il renvoie **false** si l'argument de gauche est **null**. En assumant que le contrôle de type s'est bien passé, la comparaison est basée sur les **ghNumber**s des instances. Et cette fois, lorsqu'on lance le programme, on peut voir qu'il produit le résultat attendu.

On rencontre les mêmes problèmes quand on crée un nouveau type destiné à être stocké dans un **HashSet** ou utilisé comme clef dans un **HashMap**.

Comprendre hashCode()

L'exemple précédent n'est que le début de la solution complète et correcte de ce problème. Il montre que la structure de données hachée (**HashSet** ou **HashMap**) ne sera pas capable de gérer correctement les objets clef si on ne redéfinit pas les méthodes **hashCode()** et **equals()** pour ces objets. Cependant, pour fournir une solution propre au problème, il faut comprendre ce qui se passe derrière la structure de données hachée.

Pour cela, il faut tout d'abord comprendre le pourquoi du hachage : on veut extraire un objet associé à un autre objet. Mais il est aussi possible d'accomplir ceci avec un **TreeSet** ou un **TreeMap**. Il est même possible d'implémenter sa propre **Map**. Pour cela, il nous faut fournir une méthode **Map.entrySet()** qui renvoie un ensemble d'objets **Map.Entry**. **MPair** sera définie comme le nouveau type de **Map.Entry**. Afin qu'il puisse être placé dans un **TreeSet** il doit implémenter **equals()** et être **Comparable** :

```
//: c09:MPair.java
// Une Map implémentée avec des ArrayLists.
import java.util.*;

public class MPair
implements Map.Entry, Comparable {
  Object key, value;
  MPair(Object k, Object v) {
    key = k;
    value = v;
  }
  public Object getKey() { return key; }
  public Object getValue() { return value; }
  public Object setValue(Object v){
    Object result = value;
    value = v;
    return result;
  }
  public boolean equals(Object o) {
    return key.equals(((MPair)o).key);
```

```
  }
  public int compareTo(Object rv) {
    return ((Comparable)key).compareTo(
      ((MPair)rv).key);
  }
} ///:~
```

Notez que les comparaisons ne s'effectuent que sur les clefs, les valeurs dupliquées sont donc parfaitement légales.

L'exemple suivant impémente une **Map** en utilisant une paire d'**ArrayList**s :

```
//: c09:SlowMap.java
// Une Map implémentée avec des ArrayLists.
import java.util.*;
import com.bruceeckel.util.*;

public class SlowMap extends AbstractMap {
  private ArrayList
    keys = new ArrayList(),
    values = new ArrayList();
  public Object put(Object key, Object value) {
    Object result = get(key);
    if(!keys.contains(key)) {
      keys.add(key);
      values.add(value);
    } else
      values.set(keys.indexOf(key), value);
    return result;
  }
  public Object get(Object key) {
    if(!keys.contains(key))
      return null;
    return values.get(keys.indexOf(key));
  }
  public Set entrySet() {
    Set entries = new HashSet();
    Iterator
      ki = keys.iterator(),
      vi = values.iterator();
    while(ki.hasNext())
      entries.add(new MPair(ki.next(), vi.next()));
    return entries;
  }
  public static void main(String[] args) {
    SlowMap m = new SlowMap();
    Collections2.fill(m,
      Collections2.geography, 25);
```

```
    System.out.println(m);
  }
} ///:~
```

La méthode **put()** stocke simplement les clefs et les valeurs dans les **ArrayList**s correspondantes. Dans **main()**, une **SlowMap** est remplie et imprimée pour montrer qu'elle fonctionne.

Ceci montre qu'il n'est pas difficile de produire un nouveau type de **Map**. Mais comme son nom le suggère, une **SlowMap** n'est pas très rapide, et on ne l'utilisera probablement pas si on dispose d'une autre alternative. Le problème se trouve dans la recherche de la clef : comme elles sont stockées sans aucun ordre, une recherche linéaire est effectuée, ce qui constitue la manière la plus lente de rechercher un item particulier.

Tout l'intérêt du hachage réside dans la vitesse : le hachage permet d'effectuer la recherche rapidement. Puisque le goulot d'étranglement est la recherche de la clef, une des solution du problème serait de garder les clefs triées et d'utiliser ensuite **Collections.binarySearch()** pour réaliser la recherche (un exercice à la fin du chapitre vous mènera le long de ce processus).

Le hachage va encore plus loin en spécifiant que tout ce qu'on a besoin de faire est de stocker la clef *quelque part* afin de pouvoir la retrouver rapidement. Comme on l'a déjà vu dans ce chapitre, la structure la plus efficace pour stocker un ensemble d'éléments est un tableau, c'est donc ce que nous utiliserons pour stocker les informations des clefs (notez bien que j'ai dit : « information des clefs » et non les clefs elles-mêmes). Nous avons aussi vu dans ce chapitre qu'un tableau, une fois alloué, ne peut être redimensionné, nous nous heurtons donc à un autre problème : nous voulons être capable de stocker un nombre quelconque de valeurs dans la **Map**, mais comment cela est-ce possible si le nombre de clefs est fixé par la taille du tableau ?

Tout simplement, le tableau n'est pas destiné à stocker les clefs. Un nombre dérivé de l'objet clef servira d'index dans le tableau. Ce nombre est le *code de hachage*, renvoyé par la méthode **hashCode()** (dans le jargon informatique, on parle de *fonction de hachage*) définie dans la classe **Object** et éventuellement redéfinie dans un nouveau type. Pour résoudre le problème du tableau de taille fixe, plus d'une clef peut produire le même index ; autrement dit, les *collisions* sont autorisées. Et de ce fait, la taille du tableau importe peu puisque chaque objet clef atterira quelque part dans ce tableau.

Le processus de recherche d'une valeur débute donc par le calcul du code de hachage, qu'on utilise pour indexer le tableau. Si on peut garantir qu'il n'y a pas eu de collisions (ce qui est possible si on a un nombre fixé de valeurs), alors on dispose d'une *fonction de hachage parfaite*, mais il s'agit d'un cas spécial. Dans les autres cas, les collisions sont gérées par un *chaînage externe* : le tableau ne pointe pas directement sur une valeur, mais sur une liste de valeurs. Ces valeurs sont alors parcourues de façon linéaire en utilisant la méthode **equals()**. Bien sûr, cet aspect de la recherche est plus lent, mais si la fonction de hachage est correctement écrite, il n'y aura que quelques valeurs au plus dans chaque emplacement. Et donc au lieu de parcourir toute la liste pour trouver une valeur, on saute directement dans une cellule où seules quelques entrées devront être comparées pour trouver la valeur. Cette aproche est bien plus efficace, ce qui explique pourquoi un **HashMap** est si rapide.

Connaissant les bases du hachage, il est possible d'implémenter une **Map** simple hachée :

```
//: c09:SimpleHashMap.java
// Démonstration d'une Map hachée.
```

```
import java.util.*;
import com.bruceeckel.util.*;

public class SimpleHashMap extends AbstractMap {
  // Choisir un nombre premier pour la taille de la table
  // de hachage, afin d'obtenir une distribution uniforme :
  private final static int SZ = 997;
  private LinkedList[] bucket= new LinkedList[SZ];
  public Object put(Object key, Object value) {
    Object result = null;
    int index = key.hashCode() % SZ;
    if(index < 0) index = -index;
    if(bucket[index] == null)
      bucket[index] = new LinkedList();
    LinkedList pairs = bucket[index];
    MPair pair = new MPair(key, value);
    ListIterator it = pairs.listIterator();
    boolean found = false;
    while(it.hasNext()) {
      Object iPair = it.next();
      if(iPair.equals(pair)) {
        result = ((MPair)iPair).getValue();
        it.set(pair); // Remplace l'ancien par le nouveau
        found = true;
        break;
      }
    }
    if(!found)
      bucket[index].add(pair);
    return result;
  }
  public Object get(Object key) {
    int index = key.hashCode() % SZ;
    if(index < 0) index = -index;
    if(bucket[index] == null) return null;
    LinkedList pairs = bucket[index];
    MPair match = new MPair(key, null);
    ListIterator it = pairs.listIterator();
    while(it.hasNext()) {
      Object iPair = it.next();
      if(iPair.equals(match))
        return ((MPair)iPair).getValue();
    }
    return null;
  }
  public Set entrySet() {
    Set entries = new HashSet();
```

```
      for(int i = 0; i < bucket.length; i++) {
        if(bucket[i] == null) continue;
        Iterator it = bucket[i].iterator();
        while(it.hasNext())
          entries.add(it.next());
      }
      return entries;
    }
    public static void main(String[] args) {
      SimpleHashMap m = new SimpleHashMap();
      Collections2.fill(m,
        Collections2.geography, 25);
      System.out.println(m);
    }
  } ///:~
```

Comme on appelle souvent *seaux* les « emplacements » d'une table de hachage, le tableau représentant la table est appelé **bucket**. Pour s'assurer d'une distribution la plus régulière possible, le nombre de seaux est typiquement un nombre premier. Notez qu'il s'agit d'un tableau de **LinkedLists**, qui permet de gérer automatiquement les collisions - chaque nouvel item est simplement ajouté à la fin de la liste.

La valeur de retour de **put()** est **null** ou l'ancienne valeur associée à la clef si la celle-ci était présente dans la liste. La valeur de retour est **result**, qui est initialisée à **null**, mais se voit assigner une clef si celle-ci est découverte dans la liste.

Les deux méthodes **put()** et **get()** commencent par appeler la méthode **hashCode()** de l'objet clef, dont le résultat est forcé à un nombre positif. Il est alors forcé dans la plage du tableau via l'opérateur modulo et la taille du tableau. Si l'emplacement est **null**, cela veut dire qu'aucun élément ne hache à cette localisation, et donc une nouvelle **LinkedList** est créée pour contenir l'objet qui vient de le faire. Sinon, le processus normal est de parcourir la liste pour voir s'il existe un doublon, et si c'est le cas, l'ancienne valeur est stockée dans **result** et la nouvelle valeur remplace l'ancienne. Le flag **found** permet de savoir si une ancienne paire clef - valeur a été trouvée, et dans le cas contraire, une nouvelle paire est ajoutée à la fin de la liste.

Le code de **get()** est similaire à celui de **put()**, en plus simple. L'index dans le tableau **bucket** est calculé, et si une **LinkedList** existe elle est parcourue pour trouver une concordance.

entrySet() doit trouver et parcourir toutes les listes, ajoutant tous les éléments dans le **Set** résultat. Une fois cette méthode fournie, la **Map** peut être testée en la remplissant avec des valeurs et en les imprimant.

Voici tout d'abord une terminologie nécessaire pour comprendre les mécanismes mis en jeu :

Capacité : Le nombre de seaux dans la table.

Capacité initiale : Le nombre de seaux dans la table quand celle-ci est créée. Les **HashMap** et les **HashSet** proposent des constructeurs qui permettent de spécifier la capacité initiale.

Taille : Le nombre courant d'entrées dans la table.

Facteur de charge : taille/capacité. Un facteur de charge de 0 correspond à une table vide, 0.5 correspond à une table à moitié pleine, etc. Une table faiblement chargée aura peu de collisions et

sera donc optimale pour les insertions et les recherches (mais ralentira le processus de parcours avec un itérateur). **HashMap** et **HashSet** proposent des constructeurs qui permettent de spécifier un facteur de charge, ce qui veut dire que lorsque ce facteur de charge est atteint le conteneur augmentera automatiquement sa capacité (le nombre de seaux) en la doublant d'un coup, et redistribuera les objets existants dans le nouvel ensemble de seaux (c'est ce qu'on appelle le *rehachage*).

Le facteur de charge par défaut utilisé par **HashMap** est 0.75 (il ne se rehache pas avant que la table ne soit aux ¾ pleine). Cette valeur est un bon compromis entre les performances et le coût en espace. Un facteur de charge plus élevé réduit l'espace requis par une table mais augmente le coût d'une recherche, ce qui est important parce que les recherches sont les opérations les plus courantes (incluant les appels **get()** et **put()**).

Si un **HashMap** est destiné à recevoir beaucoup d'entrées, le créer avec une grosse capacité initiale permettra d'éviter le surcoût du rehachage automatique.

Redéfinir hashCode()

Maintenant que nous avons vu les processus impliqués dans le fonctionnement d'un **HashMap**, les problèmes rencontrés dans l'écriture d'une méthode **hashCode()** prennent tout leur sens.

Tout d'abord, on ne contrôle pas la valeur réellement utilisée pour indexer le seau dans le tableau. Celle-ci est dépendante de la capacité de l'objet **HashMap**, et cette capacité change suivant la taille et la charge du conteneur. La valeur renvoyée par la méthode **hashCode()** est simplement utilisée pour calculer l'index du seau (dans **SimpleHashMap** le calcul se résume à un modulo de la taille du tableau de seaux).

Le facteur le plus important lors de la création d'une méthode **hashCode()** est qu'elle doit toujours renvoyer la même valeur pour un objet particulier, quel que soit le moment où **hashCode()** est appelée. Si on a un objet dont la méthode **hashCode()** renvoie une valeur lors d'un **put()** dans un **HashMap**, et une autre durant un appel à **get()**, on sera incapable de retrouver cet objet. Si la méthode **hashCode()** s'appuie sur des données modifiables dans l'objet, l'utilisateur doit alors être prévenu que changer ces données produira une clef différente en générant un code de hachage différent.

De plus, on ne veut pas *non plus* générer un code de hachage qui soit basé uniquement sur des informations uniques spécifiques à l'instance de l'objet - en particulier, la valeur de **this** est une mauvaise idée pour un code de hachage, puisqu'on ne peut générer une nouvelle clef identique à celle utilisée pour stocker la paire originale clef-valeur. C'est le problème que nous avons rencontré dans **SpringDetector.java** parce que l'implémentation par défaut de **hashCode()***utilise* l'adresse de l'objet. Il faut donc utiliser des informations de l'objet qui identifient l'objet d'une façon sensée.

Un exemple en est trouvé dans la classe **String**. Les **String**s ont cette caractéristique spéciale : si un programme utilise plusieurs objets **String** contenant la même séquence de caractères, alors ces objets **String** pointent tous vers la même zone de mémoire (ce mécanisme est décrit dans l'annexe A). Il semble donc sensé que le code de hachage produit par deux instances distinctes de **new String("hello")** soit identique. On peut le vérifier avec ce petit programme :

```
//: c09:StringHashCode.java
public class StringHashCode {
  public static void main(String[] args) {
    System.out.println("Hello".hashCode());
    System.out.println("Hello".hashCode());
```

```
  }
} ///:~
```

Pour que ceci fonctionne, le code de hachage de **String** doit être basé sur le contenu de la **String**.

Pour qu'un code de hachage soit efficace, il faut donc qu'il soit rapide et chargé de sens : c'est donc une valeur basée sur le contenu de l'objet. Rappelons que cette valeur n'a pas à être unique - mieux vaut se pencher sur la vitesse que sur l'unicité - mais l'identité d'un objet doit être complètement résolue entre **hashCode()** et **equals()**.

Parce qu'un code de hachage est traité avant de produire un index de seau, la plage de valeurs n'est pas importante ; il suffit de générer un **int**.

Enfin, il existe un autre facteur : une méthode **hashCode()** bien conçue doit renvoyer des valeurs bien distribuées. Si les valeurs tendent à se regrouper, alors les **HashMap**s et les **HashSet**s seront plus chargés dans certaines parties et donc moins rapides que ce qu'ils pourraient être avec une fonction de hachage mieux répartie.

Voici un exemple qui respecte ces règles de base :

```
//: c09:CountedString.java
// Créer une bonne méthode hashCode().
import java.util.*;

public class CountedString {
  private String s;
  private int id = 0;
  private static ArrayList created =
    new ArrayList();
  public CountedString(String str) {
    s = str;
    created.add(s);
    Iterator it = created.iterator();
    // id est le nombre total d'instances de cette
    // chaîne utilisées par CountedString :
    while(it.hasNext())
      if(it.next().equals(s))
        id++;
  }
  public String toString() {
    return "String: " + s + " id: " + id +
      " hashCode(): " + hashCode() + "\n";
  }
  public int hashCode() {
    return s.hashCode() * id;
  }
  public boolean equals(Object o) {
    return (o instanceof CountedString)
      && s.equals(((CountedString)o).s)
```

```
        && id == ((CountedString)o).id;
    }
    public static void main(String[] args) {
      HashMap m = new HashMap();
      CountedString[] cs = new CountedString[10];
      for(int i = 0; i < cs.length; i++) {
        cs[i] = new CountedString("hi");
        m.put(cs[i], new Integer(i));
      }
      System.out.println(m);
      for(int i = 0; i < cs.length; i++) {
        System.out.print("Looking up " + cs[i]);
        System.out.println(m.get(cs[i]));
      }
    }
} ///:~
```

CountedString inclut une **String** et un **id** représentant le nombre d'objets **CountedString** contenant une **String** identique. Le compte est réalisé dans le constructeur en parcourant la **static ArrayList** où toutes les **String**s sont stockées.

Les méthodes **hashCode()** et **equals()** renvoient des résultats basés sur les deux champs ; si elles étaient basées juste sur la **String** ou sur l'**id**, il y aurait eu des doublons pour des valeurs distinctes.

Notez comme la fonction de hachage est simple : le code de hachage de la **String** multiplié par l'**id**. Généralement, la qualité et la rapidité d'une fonction de hachage est inversement proportionnelle à sa taille.

Dans **main()**, un ensemble d'objets **CountedString** est créé, en utilisant la même **String** pour montrer que les doublons créent des valeurs uniques grâce au compteur **id**. Le **HashMap** est affiché afin de voir son organisation interne (aucun ordre n'est discernable) ; chaque clef est alors recherchée individuellement pour démontrer que le mécanisme de recherche fonctionne correctement.

Stocker des références

La bibliothèque **java.lang.ref** contient un ensemble de classes qui permettent une plus grande flexibilité dans le nettoyage des objets, et qui se révèlent particulièrement pratiques lorsqu'on a de gros objets qui peuvent saturer la mémoire. Il y a trois classes dérivées de la classe abstraite **Reference** : **SoftReference**, **WeakReference** et **PhantomReference**. Chacune d'entre elles fournit un niveau différent d'abstraction au ramasse miettes, si l'objet en question n'est accessible *qu'à travers* un de ces objets **Reference**.

Si un objet est *accessible* cela veut dire que l'objet peut être trouvé quelque part dans le programme. Ceci peut vouloir dire qu'on a une référence ordinaire sur la pile qui pointe directement sur l'objet, mais on peut aussi avoir une référence sur un objet qui possède une référence sur l'objet en question ; il peut y avoir de nombreux liens intermédiaires. Si un objet est accessible, le ramasse miettes ne peut pas le nettoyer parce qu'il est toujours utilisé par le programme. Si un objet n'est pas accessible, le programme ne dispose d'aucun moyen pour y accéder et on peut donc nettoyer cet objet tranquillement.

On utilise des objets **Reference** quand on veut continuer à stocker une référence sur cet objet - on veut être capable d'atteindre cet objet - mais on veut aussi permettre au ramasse miettes de nettoyer cet objet. Il s'agit donc d'un moyen permettant de continuer à utiliser l'objet, mais si la saturation de la mémoire est imminente, on permet que cet objet soit nettoyé.

Un objet **Reference** sert donc d'intermédiaire entre le programme et la référence ordinaire, *et* aucune référence ordinaire sur cet objet ne doit exister (mis à part celles encapsulées dans les objets **Reference**). Si le ramasse miette découvre qu'un objet est accessible à travers une référence ordinaire, il ne nettoiera pas cet objet.

Dans l'ordre **SoftReference**, **WeakReference** et **PhantomReference**, chacune d'entre elles est « plus faible » que la précédente, et correspond à un niveau différent d'accessibilité. Les références douces (**SoftReference**s) permettent d'implémenter des caches concernés par les problèmes de mémoire. Les références faibles (**WeakReference**s) sont destinées à implémenter des « mappages canoniques » - où des instances d'objets peuvent être utilisées simultanément dans différents endroits du programme, pour économiser le stockage - qui n'empêchent pas leurs clefs (ou valeurs) d'être nettoyées. Les références fantômes (**PhantomReference**s) permettent d'organiser les actions de nettoyage pre-mortem d'une manière plus flexible que ce qui est possible avec le mécanisme de finalisation de Java.

Pour les **SoftReference**s et les **WeakReference**s, on peut choisir de les stocker dans une **ReferenceQueue** (le dispositif utilisé pour les actions de nettoyage pre-mortem) ou non, mais une **PhantomReference** ne peut être créée que dans une **ReferenceQueue**. En voici la démonstration :

```
//: c09:References.java
// Illustre les objets Reference.
import java.lang.ref.*;

class VeryBig {
  static final int SZ = 10000;
  double[] d = new double[SZ];
  String ident;
  public VeryBig(String id) { ident = id; }
  public String toString() { return ident; }
  public void finalize() {
    System.out.println("Finalizing " + ident);
  }
}

public class References {
  static ReferenceQueue rq= new ReferenceQueue();
  public static void checkQueue() {
    Object inq = rq.poll();
    if(inq != null)
      System.out.println("In queue: " +
        (VeryBig)((Reference)inq).get());
  }
  public static void main(String[] args) {
    int size = 10;
    // La taille peut être choisie via la ligne de commande :
```

```
    if(args.length > 0)
      size = Integer.parseInt(args[0]);
    SoftReference[] sa =      new SoftReference[size];
    for(int i = 0; i < sa.length; i++) {
      sa[i] = new SoftReference(
        new VeryBig("Soft " + i), rq);
      System.out.println("Just created: " +
        (VeryBig)sa[i].get());
      checkQueue();
    }
    WeakReference[] wa =      new WeakReference[size];
    for(int i = 0; i < wa.length; i++) {
      wa[i] = new WeakReference(
        new VeryBig("Weak " + i), rq);
      System.out.println("Just created: " +
        (VeryBig)wa[i].get());
      checkQueue();
    }
    SoftReference s = new SoftReference(
      new VeryBig("Soft"));
    WeakReference w = new WeakReference(
      new VeryBig("Weak"));
    System.gc();
    PhantomReference[] pa =      new PhantomReference[size];
    for(int i = 0; i < pa.length; i++) {
      pa[i] = new PhantomReference(
        new VeryBig("Phantom " + i), rq);
      System.out.println("Just created: " +
        (VeryBig)pa[i].get());
      checkQueue();
    }
  }
} ///:~
```

Quand on lance ce programme (vous voudrez probablement piper la sortie à travers un utilitaire « more » afin de pouvoir l'observer page par page), on verra que les objets sont récupérés par le ramasse miettes, même si on a toujours accès à eux à travers les objets **Reference** (pour obtenir la référence réelle sur l'objet, il faut utilise la méthode **get()**). On notera aussi que **ReferenceQueue** renvoie toujours une **Reference** contenant un objet **null**. Pour utiliser les références, on peut dériver la classe **Reference** particulière qui nous intéresse et ajouter des méthodes au nouveau type de **Reference**.

Le WeakHashMap

La bibliothèque de conteneurs propose une **Map** spéciale pour stocker les références faibles : le **WeakHashMap**. Cette classe est conçue pour faciliter la création de mappages canoniques. Dans de tels mappages, on économise sur le stockage en ne créant qu'une instance d'une valeur particulière. Quand le programme a besoin de cette valeur, il recherche l'objet existant dans le mappage et

l'utilise (plutôt que d'en créer un complètement nouveau). Le mappage peut créer les valeurs comme partie de son initialisation, mais il est plus courant que les valeurs soient créées à la demande.

Puisqu'il s'agit d'une technique permettant d'économiser sur le stockage, il est très pratique que le **WeakHashMap** autorise le ramasse miettes à nettoyer automatiquement les clefs et les valeurs. Aucune opération particulière n'est nécessitée sur les clefs et les valeurs qu'on veut placer dans le **WeakHashMap** ; ils sont automatiquement encapsulés dans des **WeakReference**s par le **WeakHashMap**. Le déclenchement qui autorise le nettoyage survient lorsque la clef n'est plus utilisée, ainsi que démontré dans cet exemple :

```
//: c09:CanonicalMapping.java
// Illustre les WeakHashMaps.
import java.util.*;
import java.lang.ref.*;

class Key {
  String ident;
  public Key(String id) { ident = id; }
  public String toString() { return ident; }
  public int hashCode() {
    return ident.hashCode();
  }
  public boolean equals(Object r) {
    return (r instanceof Key)
      && ident.equals(((Key)r).ident);
  }
  public void finalize() {
    System.out.println("Finalizing Key "+ ident);
  }
}

class Value {
  String ident;
  public Value(String id) { ident = id; }
  public String toString() { return ident; }
  public void finalize() {
    System.out.println("Finalizing Value "+ident);
  }
}

public class CanonicalMapping {
  public static void main(String[] args) {
    int size = 1000;
    // La taille peut être choisie via la ligne de commande :
    if(args.length > 0)
      size = Integer.parseInt(args[0]);
    Key[] keys = new Key[size];
    WeakHashMap whm = new WeakHashMap();
    for(int i = 0; i < size; i++) {
```

```
      Key k = new Key(Integer.toString(i));
      Value v = new Value(Integer.toString(i));
      if(i % 3 == 0)
        keys[i] = k; // Save as "real" references
      whm.put(k, v);
    }
    System.gc();
  }
} ///:~
```

La classe **Key** doit fournir les méthodes **hashCode()** et **equals()** puisqu'elle est utilisée comme clef dans une structure de données hachée, comme décrit précédemment dans ce chapitre.

Quand on lance le programme, on s'aperçoit que le ramasse miettes évite une clef sur trois, parce qu'une référence ordinaire sur cette clef a aussi été placée dans le tableau **keys** et donc ces objets ne peuvent être nettoyés.

Les itérateurs revisités

Nous pouvons maintenant démontrer la vraie puissance d'un **Iterator** : la capacité de séparer l'opération de parcourir une séquence de la structure sous-jacente de cette séquence. Dans l'exemple suivant, la classe **PrintData** utilise un **Iterator** pour se déplacer à travers une séquence et appelle la méthode **toString()** pour chaque objet. Deux types de conteneurs différents sont créés - une **ArrayList** et un **HashMap** - et remplis, respectivement, avec des objets **Mouse** et **Hamster** (ces classes ont été définies précédemment dans ce chapitre). Parce qu'un **Iterator** cache la structure sous-jacente du conteneur associé, **PrintData** ne se soucie pas du type de conteneur dont l'**Iterator** provient :

```
//: c09:Iterators2.java
// Les Iterators revisités.
import java.util.*;

class PrintData {
  static void print(Iterator e) {
    while(e.hasNext())
      System.out.println(e.next());
  }
}

class Iterators2 {
  public static void main(String[] args) {
    ArrayList v = new ArrayList();
    for(int i = 0; i < 5; i++)
      v.add(new Mouse(i));
    HashMap m = new HashMap();
    for(int i = 0; i < 5; i++)
      m.put(new Integer(i), new Hamster(i));
    System.out.println("ArrayList");
```

```
    PrintData.print(v.iterator());
    System.out.println("HashMap");
    PrintData.print(m.entrySet().iterator());
  }
} ///:~
```

Notez que **PrintData.print()** s'appuie sur le fait que les objets dans les conteneurs appartiennent à la classe **Object** et donc l'appel à **toString()** par **System.out.println()** est automatique. Il est toutefois plus courant de devoir assumer qu'un **Iterator** parcourt un conteneur d'un type spécifique. Par exemple, on peut assumer que tous les objets d'un conteneur sont une **Shape** possédant une méthode **draw()**. On doit alors effectuer un transtypage descendant depuis l'**Object** renvoyé par **Iterator.next()** pour produire une **Shape**.

Choisir une implémentation

Vous devriez maintenant être conscient qu'il n'existe que trois types de conteneurs : les **Map**s, les **List**s et les **Set**s, avec seulement deux ou trois implémentations pour chacune de ces interfaces. Mais si on décide d'utiliser les fonctionnalités offertes par une **interface** particulière, comment choisir l'implémentation qui conviendra le mieux ?

Il faut bien voir que chaque implémentation dispose de ses propres fonctionnalités, forces et faiblesses. Par exemple, on peut voir dans le diagramme que les classes **Hashtable**, **Vector** et **Stack** sont des reliquats des versions précédentes de Java, ce vieux code testé et retesté n'est donc pas près d'être pris en défaut. D'un autre côté, il vaut mieux utiliser du code Java 2.

La distinction entre les autres conteneurs se ramène la plupart du temps à leur « support sous-jacent » ; c'est à dire la structure de données qui implémente physiquement l'**interface** désirée. Par exemple, les **ArrayList**s et les **LinkedList**s implémentent toutes les deux l'interface **List**, donc un programme produira les mêmes résultats quelle que soit celle qui est choisie. Cependant, une **ArrayList** est sous-tendue par un tableau, tandis qu'une **LinkedList** est implémentée sous la forme d'une liste doublement chaînée, dont chaque objet individuel contient des données ainsi que des références sur les éléments précédents et suivants dans la liste. De ce fait, une **LinkedList** est le choix approprié si on souhaite effectuer de nombreuses insertions et suppressions au milieu de la liste (les **LinkedList**s proposent aussi des fonctionnalités supplémentaires précisées dans **AbstractSequentialList**). Dans les autres cas, une **ArrayList** est typiquement plus rapide.

De même, un **Set** peut être implémenté soit sous la forme d'un **TreeSet** ou d'un **HashSet**. Un **TreeSet** est supporté par un **TreeMap** et est conçu pour produire un **Set** constamment trié. Cependant, si le **Set** est destiné à stocker de grandes quantités d'objets, les performances en insertion du **TreeSet** vont se dégrader. Quand vous écrirez un programme nécessitant un **Set**, choisissez un **HashSet** par défaut, et changez pour un **TreeSet** s'il est plus important de disposer d'un **Set** constamment trié.

Choisir entre les Lists

Un test de performances constitue la façon la plus flagrante de voir les différences entre les implémentations des **List**s. Le code suivant crée une classe interne de base à utiliser comme structure de test, puis crée un tableau de classes internes anonymes, une pour chaque test différent. Chacune de ces classes internes est appelée par la méthode **test()**. Cette approche permet d'ajouter et de supprimer facilement de nouveaux tests.

```
//: c09:ListPerformance.java
// Illustre les différences de performance entre les Lists.
import java.util.*;
import com.bruceeckel.util.*;

public class ListPerformance {
  private abstract static class Tester {
    String name;
    int size; // Nombre de tests par répétition
    Tester(String name, int size) {
      this.name = name;
      this.size = size;
    }
    abstract void test(List a, int reps);
  }
  private static Tester[] tests = {
    new Tester("get", 300) {
      void test(List a, int reps) {
        for(int i = 0; i < reps; i++) {
          for(int j = 0; j < a.size(); j++)
            a.get(j);
        }
      }
    },
    new Tester("iteration", 300) {
      void test(List a, int reps) {
        for(int i = 0; i < reps; i++) {
          Iterator it = a.iterator();
          while(it.hasNext())
            it.next();
        }
      }
    },
    new Tester("insert", 5000) {
      void test(List a, int reps) {
        int half = a.size()/2;
        String s = "test";
        ListIterator it = a.listIterator(half);
        for(int i = 0; i < size * 10; i++)
          it.add(s);
      }
    },
    new Tester("remove", 5000) {
      void test(List a, int reps) {
        ListIterator it = a.listIterator(3);
        while(it.hasNext()) {
          it.next();
```

```
            it.remove();
          }
        }
      },
    };
    public static void test(List a, int reps) {
      // Une astuce pour imprimer le nom de la classe :
      System.out.println("Testing " +
        a.getClass().getName());
      for(int i = 0; i < tests.length; i++) {
        Collections2.fill(a,
          Collections2.countries.reset(),
          tests[i].size);
        System.out.print(tests[i].name);
        long t1 = System.currentTimeMillis();
        tests[i].test(a, reps);
        long t2 = System.currentTimeMillis();
        System.out.println(": " + (t2 - t1));
      }
    }
    public static void testArray(int reps) {
      System.out.println("Testing array as List");
      // On ne peut effectuer que les deux premiers tests sur un tableau :
      for(int i = 0; i < 2; i++) {
        String[] sa = new String[tests[i].size];
        Arrays2.fill(sa,
          Collections2.countries.reset());
        List a = Arrays.asList(sa);
        System.out.print(tests[i].name);
        long t1 = System.currentTimeMillis();
        tests[i].test(a, reps);
        long t2 = System.currentTimeMillis();
        System.out.println(": " + (t2 - t1));
      }
    }
    public static void main(String[] args) {
      int reps = 50000;
      // Le nombre de répétitions peut être spécifié
      // via la ligne de commande :
      if(args.length > 0)
        reps = Integer.parseInt(args[0]);
      System.out.println(reps + " repetitions");
      testArray(reps);
      test(new ArrayList(), reps);
      test(new LinkedList(), reps);
      test(new Vector(), reps);
    }
```

```
} ///:~
```

La classe interne **Tester** est **abstract**, pour fournir une classe de base aux tests spécifiques. Elle contient une **String** à imprimer quand le test débute, un paramètre **size** destiné à être utilisé par le test comme quantité d'éléments ou nombre de répétitions des tests, un constructeur pour initialiser les champs et un méthode **abstract test()** qui réalise le travail. Tous les types de tests sont regroupés dans le tableau **tests**, initialisé par différentes classes internes anonymes dérivées de **Tester**. Pour ajouter ou supprimer des tests, il suffit d'ajouter ou de supprimer la définition d'une classe interne dans le tableau, et le reste est géré automatiquement.

Pour comparer l'accès aux tableaux avec l'accès aux conteneurs (et particulièrement avec les **ArrayList**s), un test spécial est créé pour les tableaux en en encapsulant un dans une **List** via **Arrays.asList()**. Notez que seuls les deux premiers tests peuvent être réalisés dans ce cas, parce qu'on ne peut insérer ou supprimer des éléments dans un tableau.

La **List** passée à **test()** est d'abord remplie avec des éléments, puis chaque test du tableau **tests** est chronométré. Les résultats dépendent bien entendu de la machine ; ils sont seulement conçus pour donner un ordre de comparaison entre les performances des différents conteneurs. Voici un résumé pour une exécution :

Type	**Get**	**Iteration**	**Insert**	**Remove**
tableau	1430	3850	na	na
ArrayList	3070	12200	500	46850
LinkedList	16320	9110	110	60
Vector	4890	16250	550	46850

Comme prévu, les tableaux sont plus rapides que n'importe quel conteneur pour les accès aléatoires et les itérations. On peut voir que les accès aléatoires (**get()**) sont bon marché pour les **ArrayList**s et coûteux pour les **LinkedList**s (bizarrement, l'itération est *plus rapide* pour une **LinkedList** que pour une **ArrayList**, ce qui est quelque peu contre-intuitif). D'un autre côté, les insertions et les suppressions au milieu d'une liste sont spectaculairement meilleur marché pour une **LinkedList** que pour une **ArrayList** - et *particulièrement* les suppressions. Les **Vector**s ne sont pas aussi rapides que les **ArrayList**s, et doivent être évités ; ils ne sont présents dans la bibliothèque que pour fournir une compatibilité ascendante avec le code existant (la seule raison pour laquelle ils fonctionnent dans ce programme est qu'ils ont été adaptés pour être une **List** dans Java 2). La meilleure approche est de choisir une **ArrayList** par défaut, et de changer pour une **LinkedList** si on découvre des problèmes de performance dûs à de nombreuses insertions et suppressions au milieu de la liste. Bien sûr, si on utilise un ensemble d'éléments de taille fixée, il faut se tourner vers un tableau.

Choisir entre les Sets

Suivant la taille du **Set**, on peut se tourner vers un **TreeSet** ou un **HashSet** (si on a besoin de produire une séquence ordonnée à partir d'un **Set**, il faudra utiliser un **TreeSet**). Le programme de test suivant donne une indication de ce compromis :

```
//: c09:SetPerformance.java
```

```
import java.util.*;
import com.bruceeckel.util.*;

public class SetPerformance {
  private abstract static class Tester {
    String name;
    Tester(String name) { this.name = name; }
    abstract void test(Set s, int size, int reps);
  }
  private static Tester[] tests = {
    new Tester("add") {
      void test(Set s, int size, int reps) {
        for(int i = 0; i < reps; i++) {
          s.clear();
          Collections2.fill(s,
            Collections2.countries.reset(),size);
        }
      }
    },
    new Tester("contains") {
      void test(Set s, int size, int reps) {
        for(int i = 0; i < reps; i++)
          for(int j = 0; j < size; j++)
            s.contains(Integer.toString(j));
      }
    },
    new Tester("iteration") {
      void test(Set s, int size, int reps) {
        for(int i = 0; i < reps * 10; i++) {
          Iterator it = s.iterator();
          while(it.hasNext())
            it.next();
        }
      }
    },
  };
  public static void
  test(Set s, int size, int reps) {
    System.out.println("Testing " +
      s.getClass().getName() + " size " + size);
    Collections2.fill(s,
      Collections2.countries.reset(), size);
    for(int i = 0; i < tests.length; i++) {
      System.out.print(tests[i].name);
      long t1 = System.currentTimeMillis();
      tests[i].test(s, size, reps);
      long t2 = System.currentTimeMillis();
```

```
        System.out.println(": " +
          ((double)(t2 - t1)/(double)size));
      }
    }
    public static void main(String[] args) {
      int reps = 50000;
      // Le nombre de répétitions peut être spécifié
      // via la ligne de commande :
      if(args.length > 0)
        reps = Integer.parseInt(args[0]);
      // Petit :
      test(new TreeSet(), 10, reps);
      test(new HashSet(), 10, reps);
      // Moyen :
      test(new TreeSet(), 100, reps);
      test(new HashSet(), 100, reps);
      // Gros :
      test(new TreeSet(), 1000, reps);
      test(new HashSet(), 1000, reps);
    }
  } ///:~
```

Le tableau suivant montre les résultats d'une exécution (bien sûr, vous obtiendrez des valeurs différentes suivant votre ordinateur et la JVM que vous utilisez ; lancez les tests vous-même pour vous faire une idée) :

Type	Test size	Add	Contains	Iteration
TreeSet	10	138,0	115,0	187,0
	100	189,5	151,1	206,5
	1000	150,6	177,4	40,04
HashSet	10	55,0	82,0	192,0
	100	45,6	90,0	202,2
	1000	36,14	106,5	39,39

Les performances d'un **HashSet** sont généralement supérieures à celles d'un **TreeSet** pour toutes les opérations (et en particulier le stockage et la recherche, les deux opérations les plus fréquentes). La seule raison d'être du **TreeSet** est qu'il maintient ses éléments triés, on ne l'utilisera donc que lorsqu'on aura besoin d'un **Set** trié.

Lorsqu'on doit choisir entre les différentes implémentations d'une **Map**, sa taille est le critère qui affecte le plus les performances, et le programme de test suivant donne une indication des compromis :

```
//: c09:MapPerformance.java
// Illustre les différences de performance entre les Maps.
import java.util.*;
```

```
import com.bruceeckel.util.*;

public class MapPerformance {
  private abstract static class Tester {
    String name;
    Tester(String name) { this.name = name; }
    abstract void test(Map m, int size, int reps);
  }
  private static Tester[] tests = {
    new Tester("put") {
      void test(Map m, int size, int reps) {
        for(int i = 0; i < reps; i++) {
          m.clear();
          Collections2.fill(m,
            Collections2.geography.reset(), size);
        }
      }
    },
    new Tester("get") {
      void test(Map m, int size, int reps) {
        for(int i = 0; i < reps; i++)
          for(int j = 0; j < size; j++)
            m.get(Integer.toString(j));
      }
    },
    new Tester("iteration") {
      void test(Map m, int size, int reps) {
        for(int i = 0; i < reps * 10; i++) {
          Iterator it = m.entrySet().iterator();
          while(it.hasNext())
            it.next();
        }
      }
    },
  };
  public static void
  test(Map m, int size, int reps) {
    System.out.println("Testing " +
      m.getClass().getName() + " size " + size);
    Collections2.fill(m,
      Collections2.geography.reset(), size);
    for(int i = 0; i < tests.length; i++) {
      System.out.print(tests[i].name);
      long t1 = System.currentTimeMillis();
      tests[i].test(m, size, reps);
      long t2 = System.currentTimeMillis();
      System.out.println(": " +
```

```
        ((double)(t2 - t1)/(double)size));
    }
  }
  public static void main(String[] args) {
    int reps = 50000;
    // Le nombre de répétitions peut être spécifié
    // via la ligne de commande :
    if(args.length > 0)
      reps = Integer.parseInt(args[0]);
    // Petit :
    test(new TreeMap(), 10, reps);
    test(new HashMap(), 10, reps);
    test(new Hashtable(), 10, reps);
    // Moyen :
    test(new TreeMap(), 100, reps);
    test(new HashMap(), 100, reps);
    test(new Hashtable(), 100, reps);
    // Gros :
    test(new TreeMap(), 1000, reps);
    test(new HashMap(), 1000, reps);
    test(new Hashtable(), 1000, reps);
  }
} ///:~
```

Parce que la taille du dictionnaire constitue le facteur principal, les tests de chronométrage divisent le temps par la taille du dictionnaire pour normaliser chaque mesure. Voici un ensemble de résultats (les vôtres différeront probablement) :

Type	Test size	Put	Get	Iteration
TreeMap	10	143,0	110,0	186,0
	100	201,1	188,4	280,1
	1000	222,8	205,2	40,7
HashMap	10	66,0	83,0	197,0
	100	80,7	135,7	278,5
	1000	48,2	105,7	41,4
Hashtable	10	61,0	93,0	302,0
	100	90,6	143,3	329,0
	1000	54,1	110,95	47,3

Comme on pouvait s'y attendre, les performances d'une **HashTable** sont à peu près équivalentes à celles d'un **HashMap** (bien que ceux-ci soient généralement un petit peu plus rapides). Les **TreeMap**s étant généralement plus lents que les **HashMap**s, pourquoi voudrait-on les utiliser ? En fait, on les utilise non comme des **Map**s mais comme une façon de créer une liste ordonnée. Le comportement d'un arbre est tel qu'il est toujours ordonné et n'a pas besoin d'être spéci-

fiquement trié. Une fois un **TreeMap** rempli, il est possible d'appeler **keySet()** pour récupérer un **Set** des clefs, puis **toArray()** pour produire un tableau de ces clefs. On peut alors utiliser la méthode **static Arrays.binarySearch()** (que nous étudierons plus loin) pour trouver rapidement des objets dans ce tableau trié. Bien sûr, on ne ferait ceci que si, pour une raison ou une autre, le comportement d'un **HashMap** ne convenait pas, puisqu'un **HashMap** est conçu justement pour retrouver rapidement des objets. De plus, on peut facilement créer un **HashMap** à partir d'un **TreeMap** avec une simple création d'objet. Pour résumer, votre premier réflexe si vous voulez utiliser une **Map** devrait être de se tourner vers un **HashMap**, et n'utiliser un **TreeMap** que si vous avez besoin d'une **Map** constamment triée.

Trier et rechercher dans les Lists

Les fonctions effectuant des tris et des recherches dans les **List**s ont les mêmes noms et signature que celles réalisant ces opérations sur des tableaux d'objets, mais sont des méthodes **static** appartenant à la classe **Collections** au lieu de **Arrays**. En voici un exemple, adapté de **ArraySearching.java** :

```
//: c09:ListSortSearch.java
// Trier et rechercher dans les Lists avec 'Collections.'
import com.bruceeckel.util.*;
import java.util.*;

public class ListSortSearch {
  public static void main(String[] args) {
    List list = new ArrayList();
    Collections2.fill(list,
      Collections2.capitals, 25);
    System.out.println(list + "\n");
    Collections.shuffle(list);
    System.out.println("After shuffling: "+list);
    Collections.sort(list);
    System.out.println(list + "\n");
    Object key = list.get(12);
    int index =
      Collections.binarySearch(list, key);
    System.out.println("Location of " + key +
      " is " + index + ", list.get(" +
      index + ") = " + list.get(index));
    AlphabeticComparator comp =      new AlphabeticComparator();
    Collections.sort(list, comp);
    System.out.println(list + "\n");
    key = list.get(12);
    index =
      Collections.binarySearch(list, key, comp);
    System.out.println("Location of " + key +
      " is " + index + ", list.get(" +
      index + ") = " + list.get(index));
  }
```

```
} ///:~
```

L'utilisation de ces méthodes est identique à celles dans **Arrays**, mais une **List** est utilisée à la place d'un tableau. Comme pour les tableaux, il faut passer le **Comparator** utilisé pour trier la liste lorsqu'on fait un appel à **binarySearch()**.

Ce programme illustre aussi la méthode **shuffle()** de la classe **Collections**, qui modifie aléatoirement l'ordre d'une **List**.

Utilitaires

Il existe un certain nombre d'autres méthodes bien pratiques dans la classe **Collections** :

enumeration(Collection)	Renvoie une **Enumeration** de l'argument.
max(Collection) min(Collection)	Renvoie l'élément maximum ou minimum de l'argument en utilisant la méthode de comparaison naturelle des objets de la **Collection**.
max(Collection, Comparator) min(Collection, Comparator)	Renvoie l'élément maximum ou minimum de la **Collection** en utilisant le **Comparator**.
reverse()	Inverse tous les éléments sur place.
copy(List dest, List src)	Copie les éléments de **src** vers **dest**.
fill(List list, Object o)	Remplace tous les éléments de la liste avec **o**.
nCopies(int n, Object o)	Renvoie une **List** non-modifiable de taille **n** dont les références pointent toutes sur **o**.

Notez que comme **min()** et **max()** fonctionnent avec des objets **Collection**, et non avec des **List**s, il n'est pas nécessaire que celle-ci soit triée (ainsi que nous l'avons déjà vu, il *faut* trier une **List** ou un tableau avant de leur appliquer **binarySearch()**).

Rendre une Collection ou une Map non-modifiable

Il est souvent bien pratique de créer une version en lecture seule d'une **Collection** ou d'une **Map**. La classe **Collections** permet ceci en passant le conteneur original à une méthode qui en renvoie une version non modifiable. Il existe quatre variantes de cette méthode, pour les **Collection**s (si on ne veut pas traiter une **Collection** comme un type plus spécifique), les **List**s, les **Set**s et les **Map**s. Cet exemple montre la bonne manière pour construire des versions en lecture seule des collections :

```
//: c09:ReadOnly.java
// Utilisation des méthodes Collections.unmodifiable.
import java.util.*;
import com.bruceeckel.util.*;

public class ReadOnly {
  static Collections2.StringGenerator gen =
```

```
    Collections2.countries;
  public static void main(String[] args) {
    Collection c = new ArrayList();
    Collections2.fill(c, gen, 25); // Insertion des données
    c = Collections.unmodifiableCollection(c);
    System.out.println(c); // L'accès en lecture est OK
    c.add("one"); // Modification impossible

    List a = new ArrayList();
    Collections2.fill(a, gen.reset(), 25);
    a = Collections.unmodifiableList(a);
    ListIterator lit = a.listIterator();
    System.out.println(lit.next()); // L'accès en lecture est OK
    lit.add("one"); // Modification impossible

    Set s = new HashSet();
    Collections2.fill(s, gen.reset(), 25);
    s = Collections.unmodifiableSet(s);
    System.out.println(s); // L'accès en lecture est OK
    //! s.add("one"); // Modification impossible

    Map m = new HashMap();
    Collections2.fill(m,
      Collections2.geography, 25);
    m = Collections.unmodifiableMap(m);
    System.out.println(m); // L'accès en lecture est OK
    //! m.put("Ralph", "Howdy!");
  }
} ///:~
```

Dans chaque cas, le conteneur doit être rempli de données *avant* de le rendre non-modifiable. Une fois rempli, la meilleure approche consiste à remplacer la référence existante par la référence renvoyée par l'appel à « unmodifiable ». De cette façon, on ne risque pas d'en altérer accidentellement le contenu une fois qu'on l'a rendu non modifiable. D'un autre côté, cet outil permet de garder un conteneur modifiable **private** dans la classe et de renvoyer une référence en lecture seule sur ce conteneur à partir d'une méthode. Ainsi on peut le changer depuis la classe, tandis qu'on ne pourra y accéder qu'en lecture depuis l'extérieur de cette classe.

Appeler la méthode « unmodifiable » pour un type particulier n'implique pas de contrôle lors de la compilation, mais une fois la transformation effectuée, tout appel à une méthode tentant de modifier le contenu du conteneur provoquera une **UnsupportedOperationException**.

Synchroniser une Collection ou une Map

Le mot-clef **synchronized** constitue une partie importante du *multithreading*, un sujet plus compliqué qui ne sera abordé qu'à partir du Chapitre 14. Ici, je noterai juste que la classe **Collections** permet de synchroniser automatiquement un conteneur entier. La syntaxe en est similaire aux méthodes « unmodifiable » :

```
//: c09:Synchronization.java
// Utilisation des méthodes Collections.synchronized.
import java.util.*;

public class Synchronization {
  public static void main(String[] args) {
    Collection c =
      Collections.synchronizedCollection(
        new ArrayList());
    List list = Collections.synchronizedList(
      new ArrayList());
    Set s = Collections.synchronizedSet(
      new HashSet());
    Map m = Collections.synchronizedMap(
      new HashMap());
  }
} ///:~
```

Dans ce cas, on passe directement le nouveau conteneur à la méthode « synchronisée » appropriée ; ainsi la version non synchronisée ne peut être accédée de manière accidentelle.

Echec rapide

Les conteneurs Java possèdent aussi un mécanisme qui permet d'empêcher que le contenu d'un conteneur ne soit modifié par plus d'un processus. Le problème survient si on itère à travers un conteneur alors qu'un autre processus insère, supprime ou change un objet de ce conteneur. Cet objet peut déjà avoir été passé, ou on ne l'a pas encore rencontré, peut-être que la taille du conteneur a diminué depuis qu'on a appelé **size()** - il existe de nombreux scénarios catastrophes. La bibliothèque de conteneurs Java incorpore un mécanisme d'*échec-rapide* qui traque tous les changements effectués sur le conteneur autres que ceux dont notre processus est responsable. S'il détecte que quelqu'un d'autre tente de modifier le conteneur, il produit immédiatement une **ConcurrentModificationException**. C'est l'aspect « échec-rapide » - il ne tente pas de détecter le problème plus tard en utilisant un algorithme plus complexe.

Il est relativement aisé de voir le mécanisme d'échec rapide en action - tout ce qu'on a à faire est de créer un itérateur et d'ajouter alors un item à la collection sur laquelle l'itérateur pointe, comme ceci :

```
//: c09:FailFast.java
// Illustre la notion d'« échec rapide ».
import java.util.*;

public class FailFast {
  public static void main(String[] args) {
    Collection c = new ArrayList();
    Iterator it = c.iterator();
    c.add("An object");
    // Génére une exception :
```

```
    String s = (String)it.next();
  }
} ///:~
```

L'exception survient parce que quelque chose est stocké dans le conteneur *après* que l'itérateur ait été produit par le conteneur. La possibilité que deux parties du programme puissent modifier le même conteneur mène à un état incertain, l'exception notifie donc qu'il faut modifier le code - dans ce cas, récupérer l'itérateur *après* avoir ajouté tous les éléments au conteneur.

Notez qu'on ne peut bénéficier de ce genre de surveillance quand on accède aux éléments d'une **List** en utilisant **get()**.

Il est possible de transformer un tableau en **List** grâce à la méthode **Arrays.asList()** :

```
//: c09:Unsupported.java
// Quielquefois les méthodes définies dans les
// interfaces Collection ne fonctionnent pas!
import java.util.*;

public class Unsupported {
  private static String[] s = {
    "one", "two", "three", "four", "five",
    "six", "seven", "eight", "nine", "ten",
  };
  static List a = Arrays.asList(s);
  static List a2 = a.subList(3, 6);
  public static void main(String[] args) {
    System.out.println(a);
    System.out.println(a2);
    System.out.println(
      "a.contains(" + s[0] + ") = " +
      a.contains(s[0]));
    System.out.println(
      "a.containsAll(a2) = " +
      a.containsAll(a2));
    System.out.println("a.isEmpty() = " +
      a.isEmpty());
    System.out.println(
      "a.indexOf(" + s[5] + ") = " +
      a.indexOf(s[5]));

    // Traversée à reculons :
    ListIterator lit = a.listIterator(a.size());
    while(lit.hasPrevious())
      System.out.print(lit.previous() + " ");
    System.out.println();
    // Modification de la valeur des éléments :
    for(int i = 0; i < a.size(); i++)
      a.set(i, "47");
```

```
        System.out.println(a);
        // Compiles, mais ne marche pas :
        lit.add("X"); // Opération non supportée
        a.clear(); // Non supporté
        a.add("eleven"); // Non supporté
        a.addAll(a2); // Non supporté
        a.retainAll(a2); // Non supporté
        a.remove(s[0]); // Non supporté
        a.removeAll(a2); // Non supporté
      }
    } ///:~
```

En fait, seule une partie des interfaces **Collection** et **List** est implémentée. Le reste des méthodes provoque l'apparition déplaisante d'une chose appelée **UnsupportedOperationException**. Vous en apprendrez plus sur les exceptions dans le chapitre suivant, mais pour résumer l'**interface Collection** - de même que certaines autres **interface**s de la bibliothèque de conteneurs Java - contient des méthodes « optionnelles », qui peuvent ou non être « supportées » dans la classe concrète qui implémente cette **interface**. Appeler une méthode non supportée provoque une **UnsupportedOperationException** pour indiquer une erreur de programmation.

« Comment ?!? » vous exclamez-vous, incrédule. « Tout l'intérêt des **interface**s et des classes de base provient du fait qu'elles promettent que ces méthodes feront quelque chose d'utile ! Cette affirmation rompt cette promesse - non seulement ces méthodes ne réalisent *pas* d'opération intéressante, mais en plus elles arrêtent le programme ! Qu'en est-il du contrôle de type ? »

Ce n'est pas si grave que ça. Avec une **Collection**, une **List**, un **Set** ou une **Map**, le compilateur empêche toujours d'appeler des méthodes autres que celles présentes dans cette **interface**, ce n'est donc pas comme Smalltalk (qui permet d'appeler n'importe quelle méthode sur n'importe objet, appel dont on ne verra la pertinence que lors de l'exécution du programme). De plus, la plupart des méthodes acceptant une **Collection** comme argument ne font que lire cette **Collection** - et les méthodes de « lecture » de **Collection** ne sont *pas* optionnelles.

Cette approche empêche l'explosion du nombre d'interfaces dans la conception. Les autres conceptions de bibliothèques de conteneurs semblent toujours se terminer par une pléthore d'interfaces pour décrire chacune des variations sur le thème principal et sont donc difficiles à appréhender. Il n'est même pas possible de capturer tous les cas spéciaux dans les **interface**s, car n'importe qui peut toujours créer une nouvelle **interface**. L'approche « opération non supportée » réalise l'un des buts fondamentaux de la bibliothèque de conteneurs Java : les conteneurs sont simples à apprendre et à utiliser ; les opérations non supportées sont des cas spéciaux qui peuvent être appris par la suite. Pour que cette approche fonctionne, toutefois :

1. Une **UnsupportedOperationException** doit être un événement rare. C'est à dire que toutes les opérations doivent être supportées dans la majorité des classes, et seuls quelques rares cas spéciaux peuvent ne pas supporter une certaine opération. Ceci est vrai dans la bibliothèque de conteneurs Java, puisque les classes que vous utiliserez dans 99 pour cent des cas - **ArrayList**, **LinkedList**, **HashSet** et **HashMap**, de même que les autres implémentations concrètes - supportent toutes les opérations. Cette conception fournit une « porte dérobée » si on veut créer une nouvelle **Collection** sans fournir de définition sensée pour toutes les méthodes de l'**interface Collection**, et s'intégrer tout de même dans la bibliothèque.
2. Quand une opération n'est *pas* supportée, il faut une probabilité raisonnable qu'une

UnsupportedOperationException apparaisse lors de l'implémentation plutôt qu'une fois le produit livré au client. Après tout, elle indique une erreur de programmation : une implémentation a été utilisée de manière incorrecte. Ce point est moins certain, et c'est là où la nature expérimentale de cette conception entre en jeu. Nous ne nous apercevrons de son intérêt qu'avec le temps.

Dans l'exemple précédent, **Arrays.asList()** produit une **List** supportée par un tableau de taille fixe. Il est donc sensé que les seules opérations supportées soient celles qui ne changent pas la taille du tableau. D'un autre côté, si une nouvelle **interface** était requise pour exprimer ce différent type de comportement (peut-être appelée « **FixedSizeList** »), cela laisserait la porte ouverte à la complexité et bientôt on ne saurait où donner de la tête lorsqu'on voudrait utiliser la bibliothèque.

La documentation d'une méthode qui accepte une **Collection**, une **List**, un **Set** ou une **Map** en argument doit spécifier quelles méthodes optionnelles doivent être implémentées. Par exemple, trier requiert les méthodes **set()** et **Iterator.set()**, mais pas **add()** ou **remove()**.

Les conteneurs Java 1.0 / 1.1

Malheureusement, beaucoup de code a été écrit en utilisant les conteneurs Java 1.0 / 1.1, et ajourd'hui encore ces classes continuent d'être utilisées dans du nouveau code. Bien qu'il faille éviter d'utiliser les anciens conteneurs lorsqu'on produit du code nouveau, il faut toutefois être conscient qu'ils existent. Cependant, les anciens conteneurs étaient plutôt limités, il n'y a donc pas tellement à en dire sur eux (puisqu'ils appartiennent au passé, j'éviterais de trop me pencher sur certaines horribles décisions de conception).

Vector & Enumeration

Le **Vector** était la seule séquence auto-redimensionnable dans Java 1.0 / 1.1, ce qui favorisa son utilisation. Ses défauts sont trop nombreux pour être décrits ici (se référer à la première version de ce livre, disponible sur le CD ROM de ce livre et téléchargeable sur *www.BruceEckel.com*). Fondamentalement, il s'agit d'une **ArrayList** avec des noms de méthodes longs et maladroits. Dans la bibliothèque de conteneurs Java 2, la classe **Vector** a été adaptée afin de pouvoir s'intégrer comme une **Collection** et une **List**, et donc dans l'exemple suivant la méthode **Collections2.fill()** est utilisée avec succès. Ceci peut mener à des effets pervers, car on pourrait croire que la classe **Vector** a été améliorée alors qu'elle n'a été incluse que pour supporter du code pré-Java2.

La version Java 1.0 / 1.1 de l'itérateur a choisi d'inventer un nouveau nom, « énumération », au lieu d'utiliser un terme déjà familier pour tout le monde. L'interface **Enumeration** est plus petite qu'**Iterator**, ne possédant que deux méthodes, et utilise des noms de méthodes plus longs : **boolean hasMoreElements()** renvoie **true** si l'énumération contient encore des éléments, et **Object nextElement()** renvoie l'élément suivant de cette énumération si il y en a encore (autrement elle génére une exception).

Enumeration n'est qu'une interface, et non une implémentation, et même les nouvelles bibliothèques utilisent encore quelquefois l'ancienne **Enumeration** - ce qui est malheureux mais généralement sans conséquence. Bien qu'il faille utiliser un **Iterator** quand on le peut, on peut encore rencontrer des bibliothèques qui renvoient des **Enumeration**s.

Toute **Collection** peut produire une **Enumeration** via la méthode **Collections.enumerations()**, comme le montre cet exemple :

```
//: c09:Enumerations.java
// Vectors et Enumerations de Java 1.0 / 1.1.
import java.util.*;
import com.bruceeckel.util.*;

class Enumerations {
  public static void main(String[] args) {
    Vector v = new Vector();
    Collections2.fill(
      v, Collections2.countries, 100);
    Enumeration e = v.elements();
    while(e.hasMoreElements())
      System.out.println(e.nextElement());
    // Produit une Enumeration à partir d'une Collection:
    e = Collections.enumeration(new ArrayList());
  }
} ///:~
```

La classe **Vector** de Java 1.0 / 1.1 ne dispose que d'une méthode **addElement()**, mais **fill()** utilise la méthode **add()** qui a été copiée quand **Vector** a été transformé en **List**. Pour produire une **Enumeration**, il faut appeler **elements()**, on peut alors l'utiliser pour réaliser une itération en avant.

La dernière ligne crée une **ArrayList** et utilise **enumeration()** pour adapter une **Enumeration** à partir de l'**Iterator** de l'**ArrayList**. Ainsi, si on a du vieux code qui requiert une **Enumeration**, on peut tout de même utiliser les nouveaux conteneurs.

Hashtable

Comme on a pu le voir dans les tests de performances de ce chapitre, la **Hashtable** de base est très similaire au **HashMap**, et ce même jusqu'aux noms des méthodes. Il n'y a aucune raison d'utiliser **Hashtable** au lieu de **HashMap** dans du nouveau code.

Stack

Le concept de la pile a déjà été introduit plus tôt avec les **LinkedList**s. Ce qui est relativement étrange à propos de la classe **Stack** Java 1.0 / 1.1 est qu'elle est *dérivée* de **Vector** au lieu d'utiliser un **Vector** comme composant de base. Elle possède donc toutes les caractéristiques et comportements d'un **Vector** en plus des comportements additionnels d'une **Stack**. Il est difficile de savoir si les concepteurs ont choisi délibérément cette approche en la jugeant particulièrement pratique ou si c'était juste une conception naïve.

Voici une simple illustration de **Stack** qui « pousse » chaque ligne d'un tableau **String** :

```
//: c09:Stacks.java
// Illustration de la classe Stack.
import java.util.*;

public class Stacks {
  static String[] months = {
```

```
    "January", "February", "March", "April",
    "May", "June", "July", "August", "September",
    "October", "November", "December" };
  public static void main(String[] args) {
    Stack stk = new Stack();
    for(int i = 0; i < months.length; i++)
      stk.push(months[i] + " ");
    System.out.println("stk = " + stk);
    // Traiter une pile comme un Vector :
    stk.addElement("The last line");
    System.out.println(
      "element 5 = " + stk.elementAt(5));
    System.out.println("popping elements:");
    while(!stk.empty())
      System.out.println(stk.pop());
  }
} ///:~
```

Chaque ligne du tableau **months** est insérée dans la **Stack** avec **push()**, et récupérée par la suite au sommet de la pile avec un **pop()**. On peut aussi réaliser des opérations de **Vector** sur l'objet **Stack**. Ceci est possible car, de par les propriétés de l'héritage, une **Stack** *est* un **Vector**. Donc toutes les opérations qui peuvent être effectuées sur un **Vector** peuvent aussi être réalisées sur une **Stack**, comme **elementAt()**.

Ainsi que mentionné précédemment, il vaut mieux utiliser une **LinkedList** si on souhaite un comportement de pile.

BitSet

Un **BitSet** est utile si on veut stocker efficacement un grand nombre d'informations on-off. Cette classe n'est efficace toutefois que sur le plan de la taille ; si le but recherché est un accès performant, mieux vaut se tourner vers un tableau de quelque type natif.

De plus, la taille minimum d'un **BitSet** est celle d'un **long**, soit 64 bits. Ce qui implique que si on stocke n'importe quelle quantité plus petite, telle que 8 bits, un **BitSet** introduira du gaspillage ; il vaudrait donc mieux créer sa propre classe, ou utiliser un tableau, pour stocker les flags si la taille est un problème.

Un conteneur normal se redimensionne si on ajoute de nouveaux éléments, et un **BitSet** n'échappe pas à la règle. L'exemple suivant montre comme le **BitSet** fonctionne :

```
//: c09:Bits.java
// Illustration des BitSets.
import java.util.*;

public class Bits {
  static void printBitSet(BitSet b) {
    System.out.println("bits: " + b);
    String bbits = new String();
    for(int j = 0; j < b.size() ; j++)
```

```
      bbits += (b.get(j) ? "1" : "0");
    System.out.println("bit pattern: " + bbits);
  }
  public static void main(String[] args) {
    Random rand = new Random();
    // Récupère les LSB (Less Significant Bits - NdT: bit de poids fort) de nextInt() :
    byte bt = (byte)rand.nextInt();
    BitSet bb = new BitSet();
    for(int i = 7; i >=0; i--)
      if(((1 << i) &  bt) != 0)
        bb.set(i);
      else
        bb.clear(i);
    System.out.println("byte value: " + bt);
    printBitSet(bb);

    short st = (short)rand.nextInt();
    BitSet bs = new BitSet();
    for(int i = 15; i >=0; i--)
      if(((1 << i) &  st) != 0)
        bs.set(i);
      else
        bs.clear(i);
    System.out.println("short value: " + st);
    printBitSet(bs);

    int it = rand.nextInt();
    BitSet bi = new BitSet();
    for(int i = 31; i >=0; i--)
      if(((1 << i) &  it) != 0)
        bi.set(i);
      else
        bi.clear(i);
    System.out.println("int value: " + it);
    printBitSet(bi);

    // Teste les bitsets >= 64 bits:
    BitSet b127 = new BitSet();
    b127.set(127);
    System.out.println("set bit 127: " + b127);
    BitSet b255 = new BitSet(65);
    b255.set(255);
    System.out.println("set bit 255: " + b255);
    BitSet b1023 = new BitSet(512);
    b1023.set(1023);
    b1023.set(1024);
    System.out.println("set bit 1023: " + b1023);
```

```
    }
} ///:~
```

Le générateur de nombres aléatoires est utilisé pour générer un **byte**, un **short** et un **int** au hasard, et chacun est transformé en motif de bits dans un **BitSet**. Ceci fonctionne bien puisque la taille d'un **BitSet** est de 64 bits, et donc aucune de ces opérations ne nécessite un changement de taille. Un **BitSet** de 512 bits est alors créé. Le constructeur alloue de la place pour deux fois ce nombre de bits. Cependant, on peut tout de même positionner le bit 1024 ou même au delà.

Résumé

Pour résumer les conteneurs fournis dans la bibliothèque standard Java :

1. Un tableau associe des indices numériques à des objets. Il stocke des objets d'un type connu afin de ne pas avoir à transtyper le résultat quand on récupère un objet. Il peut être multidimensions, et peut stocker des scalaires. Cependant, sa taille ne peut être modifiée une fois créé.

2. Une **Collection** stocke des éléments, alors qu'une **Map** stocke des paires d'éléments associés.

3. Comme un tableau, une **List** associe aussi des indices numériques à des objets - on peut penser aux tableaux et aux **List**s comme à des conteneurs ordonnés. Une **List** se redimensionne automatiquement si on y ajoute des éléments. Mais une **List** ne peut stocker que des références sur des **Object**s, elle ne peut donc stocker des scalaires et il faut toujours transtyper le résultat quand on récupère une référence sur un **Object** du conteneur.

4. Utiliser une **ArrayList** si on doit effectuer un grand nombre d'accès aléatoires, et une **LinkedList** si on doit réaliser un grand nombre d'insertions et de suppressions au sein de la liste.

5. Le comportement de files, files doubles et piles est fourni via les **LinkedList**s.

6. Une **Map** est une façon d'associer non pas des nombres, mais des *objets* à d'autres objets. La conception d'un **HashMap** est focalisée sur les temps d'accès, tandis qu'un **TreeMap** garde ses clefs ordonnées, et n'est donc pas aussi rapide qu'un **HashMap**.

7. Un **Set** n'accepte qu'une instance de valeur de chaque objet. Les **HashSet**s fournissent des temps d'accès optimaux, alors que les **TreeSet**s gardent leurs éléments ordonnés.

8. Il n'y a aucune raison d'utiliser les anciennes classes **Vector**, **Hashtable** et **Stack** dans du nouveau code.

Les conteneurs sont des outils qu'on peut utiliser jour après jour pour rendre les programmes plus simples, plus puissants et plus efficaces.

Exercices

1. Créez un tableau de **double**s et remplissez-le avec **fill()** en utilisant **RandDoubleGenerator**. Affichez les résultats.

2. Créez une nouvelle classe **Gerbil** possédant un **int gerbilNumber** initialisé dans le constructeur (similaire à l'exemple **Mouse** de ce chapitre). Donnez-lui une méthode **hop()**

qui affiche son numéro de gerboise et un message indiquant qu'elle est en train de sauter. Créez une **ArrayList** et ajoutez-y un ensemble d'objets **Gerbil**. Maintenant utilisez la méthode **get()** pour parcourir la **List** et appelez **hop()** pour chaque **Gerbil**.

3. Modifiez l'exercice 2 afin d'utiliser un **Iterator** lors du parcours de la **List** pour appeler **hop()**.

4. Prenez la classe **Gerbil** de l'exercice 2 et placez-la dans une **Map** à la place, en associant le nom (une **String**) de la **Gerbil** à chaque **Gerbil** (la valeur) stockée dans la table. Récupérer un **Iterator** sur l'ensemble des clefs (obtenu via **keySet()**) et utilisez-le pour parcourir la **Map**, récupérez la **Gerbil** pour chaque clef, imprimez son nom et dites-lui de sauter avec la méthode **hop()**.

5. Créez une **List** (essayez avec une **ArrayList** et une **LinkedList**) et remplissez-la en utilisant **Collections2.countries**. Triez cette liste et affichez-la, puis appliquez-lui **Collections.shuffle()** plusieurs fois, en l'imprimant à chaque fois pour voir les effets de cette méthode.

6. Montrez que vous ne pouvez ajouter que des objets **Mouse** dans une **MouseList**.

7. Modifiez **MouseList.java** de manière à ce qu'elle hérite de **ArrayList** au lieu d'utiliser la composition. Illustrez le problème de cette approche.

8. Réparez **CatsAndDogs.java** en créant un conteneur **Cats** (en utilisant **ArrayList**) qui n'accepte que des objets **Cats**.

9. Créez un conteneur qui encapsule un tableau de **String**s, qui ne stocke et ne renvoie que des **String**s, afin de ne pas avoir de transtypage à faire lorsqu'on l'utilise. Si le tableau interne n'est pas assez grand pour l'insertion suivante, le conteneur devra se redimensionner automatiquement. Dans **main()**, comparez les performances de votre conteneur avec une **ArrayList** contenant des **String**s.

10. Répétez l'exercice 9 pour un conteneur d'**int**s, et comparez les performances avec une **ArrayList** contenant des objets **Integer**. Dans le test de performances, incluez en plus le fait d'incrémenter chaque objet du conteneur.

11. Créez un tableau pour chaque type primitif et un tableau de **String**s, remplissez chaque tableau en utilisant un générateur fourni parmi les utilitaires de **com.bruceeckel.util**, et afficher chaque tableau en utilisant la méthode **print()** appropriée.

12. Créez un générateur qui produise des noms de personnages de vos films préférés (que pensez-vous de *Snow White* ou *Star Wars*), et revienne au début quand il n'a plus de noms à proposer. Utilisez les utilitaires de **com.bruceeckel.util** pour remplir un tableau, une **ArrayList**, une **LinkedList** et les deux types de **Set** disponibles, puis imprimez chaque conteneur.

13. Créez une classe contenant deux objets **String**, et rendez-la **Comparable** afin que la comparaison ne porte que sur la première **String**. Remplissez un tableau et une **ArrayList** avec des objets de cette classe, en utilisant le générateur **geography**. Montrez que le tri fonctionne correctement. Créez maintenant un **Comparator** qui porte sur la deuxième **String**, et montrez que le tri fonctionne aussi ; effectuez une recherche en utilisant votre **Comparator**.

14. Modifiez l'exercice 13 afin d'utiliser un tri alphabétique.

15. Utilisez **Arrays2.RandStringGenerator** pour remplir un **TreeSet** utilisant un tri alphabétique. Imprimez le **TreeSet** pour vérifier l'ordre.

16. Créez une **ArrayList** et une **LinkedList**, et remplissez-les en utilisant le générateur **Collections2.capitals**. Imprimez chaque liste en utilisant un **Iterator** ordinaire, puis insérer une liste dans l'autre en utilisant un **ListIterator**, en insérant un élément de la deuxième liste entre chaque élément de la première liste. Réalisez maintenant cette insertion en partant de la fin de la première liste et en la parcourant à l'envers.

17. Ecrivez une méthode qui utilise un **Iterator** pour parcourir une **Collection** et imprime le code de hachage de chaque objet du conteneur. Remplissez tous les types de **Collection**s existants avec des objets et utilisez votre méthode avec chaque conteneur.

18. Réparez le problème d'**InfiniteRecursion.java**.

19. Créez une classe, puis créez un tableau d'objets de votre classe. Remplissez une **List** à partir du tableau. Créez un sous-ensemble de la **List** en utilisant **subList()**, puis supprimez cet ensemble de la **List** avec **removeAll()**.

20. Modifiez l'exercice 6 du Chapitre 7 afin de stocker les **Rodent**s dans une **ArrayList** et utilisez un **Iterator** pour parcourir la séquence des **Rodent**s. Rappelez-vous qu'une **ArrayList** ne stocke que des **Object**s et qu'on doit donc les transtyper pour retrouver le comportement d'un **Rodent**.

21. En vous basant sur **Queue.java**, créez une classe **DoubleFile** et testez-la.

22. Utilisez un **TreeMap** dans **Statistics.java**. Ajoutez maintenant du code afin de tester les différences de performances entre **HashMap** et **TreeMap** dans ce programme.

23. Créez une **Map** et un **Set** contenant tous les pays dont le nom commence par un 'A'.

24. Remplissez un **Set** à l'aide de **Collections2.countries**, utilisez plusieurs fois les mêmes données et vérifiez que le **Set** ne stocke qu'une seule instance de chaque donnée. Testez ceci avec les deux types de **Set**.

25. A partir de **Statistics.java**, créez un programme qui lance le test plusieurs fois et vérifie si un nombre a tendance à apparaître plus souvent que les autres dans les résultats.

26. Réécrivez **Statistics.java** en utilisant un **HashSet** d'objets **Counter** (vous devrez modifiez la classe **Counter** afin qu'elle fonctionne avec un **HashSet**). Quelle approche semble la meilleure ?

27. Modifiez la classe de l'exercice 13 afin qu'elle puisse être stockée dans un **HashSet** et comme clef dans un **HashMap**.

28. Créez un **SlowSet** en vous inspirant de **SlowMap.java**.

29. Appliquez les tests de **Map1.java** à **SlowMap** pour vérifier que cette classe fonctionne. Corrigez dans **SlowMap** tout ce qui ne marche pas correctement.

30. Implémentez le reste de l'interface **Map** pour **SlowMap**.

31. Modifiez **MapPerformance.java** afin d'inclure les tests pour **SlowMap**.

32. Modifiez **SlowMap** afin qu'elle utilise une seule **ArrayList** d'objets **MPair** au lieu de deux **ArrayList**s. Vérifiez que la version modifiée fonctionne correctement. Utilisez **MapPerformance.java** pour tester cette nouvelle **Map**. Modifiez maintenant la méthode **put()** afin qu'elle effectue un **sort()** après chaque insertion, et modifiez **get()** afin d'utiliser **Collections.binarySearch()** pour récupérer la clef. Comparez les performances de la nouvelle version avec les anciennes.

33. Ajoutez un champ **char** à **CountedString** qui soit aussi initialisé dans le constructeur, et modifiez les méthodes **hashCode()** et **equals()** afin d'inclure la valeur de ce **char**.

34. Modifiez **SimpleHashMap** afin de signaler les collisions, et testez ce comportement en ajoutant les mêmes données plusieurs fois afin de voir les collisions.

35. Modifiez **SimpleHashMap** afin de signaler le nombre d'objets testés lorsqu'une collision survient. Autrement dit, combien d'appels à **next()** doivent être effectués sur l'**Iterator** parcourant la **LinkedList** pour trouver l'occurence recherchée ?

36. Implémentez les méthodes **clear()** et **remove()** pour **SimpleHashMap**.

37. Implémentez le reste de l'interface **Map** pour **SimpleHashMap**.

38. Ajoutez une méthode **private rehash()** à **SimpleHashMap** qui soit invoquée lorsque le facteur de charge dépasse 0,75. Au cours du rehachage, doublez le nombre de seaux et recherchez le premier nombre premier qui soit supérieur à cette valeur pour déterminer le nouveau nombre de seaux.

39. Créez et testez un **SimpleHashSet** en vous inspirant de **SimpleHashMap.java**.

40. Modifiez **SimpleHashMap** afin d'utiliser des **ArrayList**s au lieu de **LinkedList**s. Modifiez **MapPerformance.java** afin de comparer les performances des deux implémentations.

41. Consultez la documentation HTML du JDK (téléchargeable sur *java.sun.com*) de la classe **HashMap**. Créez un **HashMap**, remplissez-le avec des éléments et déterminez son facteur de charge. Testez la vitesse d'accès aux éléments de ce dictionnaire ; essayez d'augmenter cette vitesse en créant un nouveau **HashMap** d'une capacité initiale supérieure et en copiant l'ancien dictionnaire dans le nouveau.

42. Dans le Chapitre 8, localisez l'exemple **GreenHouse.java** comportant 3 fichiers. Dans **Controller.java**, la classe **EventSet** n'est qu'un conteneur. Changez le code afin d'utiliser une **LinkedList** au lieu d'un **EventSet**. Remplacer **EventSet** par **LinkedList** ne suffira pas, il vous faudra aussi utiliser un **Iterator** pour parcourir l'ensemble d'événements.

43. (Difficile). Créez votre propre classe de dictionnaire haché, particularisée pour un type particulier de clefs : **String** par exemple. Ne la faites pas dériver de **Map**. A la place, dupliquez les méthodes afin que les méthodes **put()** et **get()** n'acceptent que des objets **String** comme clef, et non des **Object**s. Tout ce qui touche les clefs ne doit pas impliquer de types génériques, mais uniquement des **String**s afin d'éviter les surcoûts liés au transtypage. Votre but est de réaliser l'implémentation la plus rapide possible. Modifiez **MapPerformance.java** afin de tester votre implémentation contre un **HashMap**.

44. (Difficile). Trouvez le code source de **List** dans la bibliothèque de code source Java fournie dans toutes les distributions Java. Copiez ce code et créez une version spéciale appelée **intList** qui ne stocke que des **int**s. Réfléchissez à ce qu'il faudrait pour créer une version spéciale de **List** pour chacun des types scalaires. Réfléchissez maintenant à ce qu'il faudrait si on veut créer une classe de liste chaînée qui fonctionne avec tous les types scalaires. Si les types paramétrés sont implémentés un jour dans Java, ils fourniront un moyen de réaliser ce travail automatiquement (entre autres).

[44]Il est toutefois possible de s'enquérir de la taille du **vector**, et la méthode **at()** effectue, *elle*, un contrôle sur les indices.

[45]C'est l'un des points où le C++ est indiscutablement supérieur à Java, puisque le C++

supporte la notion de *types paramétrés* grâce au mot-clef **template**.

[46]Le programmeur C++ remarquera comme le code pourrait être réduit en utilisant des arguments par defaut et des templates. Le programmeur Python, lui, notera que cette bibliothèque est complètement inutile dans ce langage.

[47]Par Joshua Bloch de chez Sun.

[48]Ces données ont été trouvées sur Internet, et parsées ensuite par un programme Python (cf. *www.Python.org*).

[49]Voici un endroit où la surcharge d'opérateur serait appréciée.

[50]Si ces accélérations ne suffisent pas pour répondre aux besoins de performance du programme, il est toujours possible d'accélérer les accès en implémentant une nouvelle **Map** et en la particularisant pour le type spécifique destiné à être stocké pour éviter les délais dûs au transtypage en **Object** et inversement. Pour atteindre des niveaux encore plus élevés de performance, les fanas de vitesse pourront se référer à *The Art of Computer Programming, Volume 3: Sorting and Searching, Second Edition* de Donald Knuth pour remplacer les listes par des tableaux qui possèdent deux avantages supplémentaires : leurs caractéristiques peuvent être optimisées pour le stockage sur disque et ils peuvent économiser la plus grande partie du temps passée à créer et détruire les enregistrements individuels.

Chapitre 10 - Gestion des erreurs avec les exceptions

La philosophie de base de Java est que « un code mal formé ne sera pas exécuté ».

L'instant idéal pour détecter une erreur est la compilation, bien avant que vous essayiez d'exécuter le programme. Malheureusement, toutes les erreurs ne peuvent être détectées à cette étape. Les problèmes restants doivent être gérés à l'exécution avec un certain formalisme qui permet à l'initiateur de l'erreur de passer une information suffisante à un récepteur qui va savoir comment traiter proprement cette anomalie.

En C et dans d'autres langages plus anciens, il pouvait y avoir plusieurs formalismes, et ils étaient généralement établis comme des conventions et pas comme partie intégrante du langage de programmation. la gestion des erreurs se faisant typiquement en retournant par exemple une valeur spécifique ou en positionnant un drapeau [flag], le récepteur avait la charge de tester ces données et de déterminer qu'il y avait eu une anomalie. Malgré cela au fil des ans on se rendit compte que les programmeurs qui utilisent des librairies avaient tendance à se croire infaillible « oui des erreurs peuvent se produire chez les autres mais pas dans *mon* code ! » Alors, de façon quasi naturelle, ils ne souhaitaient pas tester les conditions d'erreur (elles étaient parfois trop grossières pour être testées [51]. Si vous *étiez* assez méticuleux pour tester toutes les conditions d'erreur à chaque appel de fonction votre code pouvait devenir d'une illisibilité cauchemardesque. Parce que les programmeurs avaient toujours la possibilité de coupler un système avec ces langages ils étaient réticents à admettre la vérité : cette approche de la *gestion des erreurs* était un obstacle à la création de grands programmes, robustes et faciles à maintenir.

La solution est prendre la nature primitive de la gestion des erreurs et de renforcer le formalisme. Cette solution à une longue histoire, puisque l'implémentation de la *gestion des erreurs* remonte au systèmes d'exploitation des années soixante et même au « **ON ERROR GOTO** » du BASIC. Mais la gestion des erreurs du C++ était basée sur ADA, et Java est basé sur le C++ (bien qu'il ressemble plus a du pascal objet).

Le « Mot » exception est pris dans le sens « je fais exception de ce cas ». A l'instant ou le problème se produit vous pouvez ne pas savoir quoi en faire mais vous savez que vous ne pouvez pas continuer sans vous en soucier ; vous devez vous arrêter et quelqu'un quelque part doit savoir que faire. Donc vous transmettez le problème dans un contexte plus général ou quelqu'un est qualifié pour prendre la décision appropriée (comme dans une chaîne de commandement).

L'autre apport significatif de la gestion des exceptions est qu'elle simplifie le code de la gestion des erreurs. Au lieu de tester une condition d'erreur et la traiter à différents endroits de votre programme, vous n'avez plus besoin de placer le code de gestion des erreurs à l'appel de la méthode (puisque l'exception garantira que quelqu'un la lèvera) et vous avez juste besoin de traiter le problème à un seul endroit, appelé *Exception Handler.* Cette gestion vous permet d'économiser du code, et de séparer le code que vous voulez exécuter du code qui doit être exécuté quand des erreurs se produisent. Généralement, la lecture l'écriture et le déboguage du code est plus claire avec les exceptions qu'avec l'ancienne méthode.

La gestion des exceptions étant supportée par le Compilateur JAVA, il y a de nombreux exemples qui peuvent être écrits dans ce livre sans apprendre le mécanisme de gestion des exceptions. Ce chapitre vous instruit sur le code dont vous avez besoin pour gérer de façon adéquate les

exceptions, et la façon de générer vos propres exceptions si une de vos méthodes venait à ne pas fonctionner.

Les exceptions de base

Une *condition exceptionnelle* est un problème qui interdit de continuer la méthode ou la partie de code que vous êtes en train d'exécuter. il est important de distinguer une condition exceptionnelle d'un problème normal. Avec une condition exceptionnelle vous ne pouvez pas continuer l'exécution car vous n'avez pas suffisamment d'informations pour la traiter dans le *contexte courant*. Tout ce que vous pouvez faire est de changer de contexte et d'essayer de régler ce problème dans un contexte de plus haut niveau. C'est ce qui se passe quand vous générez une exception.

Un exemple simple est la division. Si vous vous apprêtez à diviser par Zéro, il est intéressant de vérifier que le dénominateur est non nul avant d'effectuer la division. Mais qu'est ce que cela signifie que le dénominateur soit égal à zéro ? Peut être savez vous, dans le contexte, de cette méthode particulière comment agir avec un dénominateur égal à Zéro. Mais si c'est une valeur inattendue, vous ne pouvez pas la traiter et vous devez donc générer une exception plutôt que de continuer l'exécution.

Quand vous générez une exception, plusieurs actions sont déclenchées. Premièrement, l'objet Exception est instancié de la même façon que tout objet java est crée : à la volée avec l'instruction **new**. Ensuite le chemin courant d'exécution (celui que vous ne pouvez continuer) est stoppé et la référence à l'objet exception est sortie du contexte courant. A cet instant le mécanisme des gestion des exceptions prend la main et commence à chercher un endroit approprié pour continuer à exécuter le programme. Cet endroit est le *Gestionnaire d'exception* (Exception Handler), dont la tâche est de résoudre le problème de façon à ce que le programme puisse essayer de lancer ou une autre tâche ou juste continuer en séquence.

Pour prendre un exemple simple, considérons une instance d'objet appelée **t**. Il est possible qu'il vous soit passé une référence qui n'ai pas été initialisée, donc vous pouvez vouloir vérifier son initialisation avant d'appeler une méthode sur cette instance. Vous pouvez envoyer l'information sur l'erreur dans un contexte plus général en créant un objet représentant votre information et en la lançant depuis votre contexte. Cette action s'appelle *générer une exception*. Cette action ressemble à ceci :

```
if(t == null)
  throw new NullPointerException();
```

Cette chaîne de caractères peut être extraite ultérieurement de différentes façons, nous verrons cela brièvement.

Le mot-clé **throw** déclenche une série d'événements relativement magiques. Typiquement, vous allez d'abord utilisez **new** pour instancier un objet qui représente la condition d'erreur. Vous transmettez la référence résultante au **throw**. L'objet est en effet retourné par la méthode, même si ce n'est pas ce type d'objet que la méthode doit renvoyer. Une vision simpliste de la gestion des exceptions est de la considérer comme un mécanisme de retour mais vous allez au devant de problèmes si vous poursuivez cette analogie trop loin. Vous pouvez aussi sortir de l'exécution normale en lançant une exception. Mais une valeur est retournée et la méthode ou l'environnement

se termine.

Toute similitude avec un retour ordinaire d'une méthode s'arrête ici parce que l'endroit *où* vous arrivez est complètement différent de celui d'un retour normal d'une méthode. (Vous atterrissez chez le gestionnaire d'exception approprié qui peut être très éloigné, plusieurs niveau plus bas sur la pile d'appels, de l'endroit ou l'exception a été générée.)

De plus, vous pouvez lancer n'importe quel type d'objet **Throwable**. Ainsi vous générerez une classe d'exception pour chaque type d'erreur. l'information à propos de l'erreur est contenue à la fois dans l'objet exception et implicitement dans le type de l'objet exception choisi, afin que quelqu'un dans le contexte supérieur sache quoi faire de votre exception (Souvent, la seule information est le type de l'objet exception rien d'autre de significatif est stocké.)

Attraper une exception

Si une méthode génère une exception elle doit supposer qu'elle sera intercepté et levée. Un des avantages de la gestion des exceptions dans Java est qu'elle vous permet de vous concentrer sur le problème que vous essayez de résoudre à un endroit, et enfin de placer le code concernant les erreurs à un autre endroit.

Pour comprendre comment une exception est levée vous devez comprendre le concept de *région surveillée* qui est une région de code qui peut générer des exceptions et qui est suivie par le code qui traite ces exceptions.

Le bloc try

Si vous êtes à l'intérieur d'une méthode et que vous générez une exception (ou une méthode à l'intérieur de celle ci génère une exception), cette méthode va sortir dans le processus de génération de l'exception. Si vous ne voulez pas qu'un **throw** provoque la sortie de la méthode vous pouvez spécifier un bloc spécial à l'intérieur de celle-ci qui va intercepter l'exception. C'est le Bloc **try** appelé ainsi car vous essayez vos différents appels de méthode ici . Le bloc **try** est une section ordinaire précédé du mot clé **try**.

```
try {
  // Code that might generate exceptions
}
```

Si vous vouliez tester les erreurs attentivement dans un langage de programmation qui ne supporte pas la gestion des exceptions vous devriez entourez l'appel de chaque méthode avec le code de vérification de l'initialisation et de vérification des erreurs, même si vous appeliez la même méthode plusieurs fois. Avec la Gestion des exceptions vous mettez tout dans le bloc try et capturer toute les exceptions en un seul endroit. Cela signifie que votre code est beaucoup plus simple à lire et à écrire car le bon code est séparé de celui de la gestion des erreurs.

Les gestionnaires d'exceptions

Bien sûr, l'exception générée doit être traitée quelque part. Cet endroit est le *gestionnaire d'excep-*

tions, et il y en a un pour chaque type d'exception que vous voulez intercepter. Les gestionnaires d'exceptions sont placés juste derrière le bloc try. Et reconnaissables par le mot clé **catch**.

```
try {
  // Code that might generate exceptions
} catch(Type1 id1) {
  // Handle exceptions of Type1
} catch(Type2 id2) {
  // Handle exceptions of Type2
} catch(Type3 id3) {
  // Handle exceptions of Type3
}
// etc...
```

Chaque clause du catch (gestionnaire d'exception) est comme une méthode qui ne prend qu'un argument d'un type particulier. Les identifiants (**id1**,**id2**, et ainsi de suite) peuvent êtres utilisés dans les gestionnaire d'exception, comme des paramètres de méthodes. Quelque fois vous n'utilisez pas l'identifiant car le type de l'exception vous fournit assez d'informations pour la traiter mais il doit toujours être présent.

Le gestionnaire doit être placé juste derrière le bloc try. Si une exception est générée, le mécanisme de gestion des exceptions va à la recherche du premier gestionnaire d .exception dont le type correspond à celui de l'exception, cette recherche se termine quand une des clauses Catch est exécutée. Seulement une clause catch sera exécutée ; ce n'est pas comme un **switch** ou vous devez positionner un break après chaque **case** pour empêcher les autres conditions de s'exécuter.

Prenez note qu'avec le bloc try, différentes méthodes peuvent générer la même exception mais vous n'aurez besoin que d'un seul gestionnaire d'exception.

Terminaison contre Restauration

Ce sont les deux modèles de base dans la théorie de la gestion des exceptions. Dans la *terminaison* (supportée par Java et C++) on suppose que l'erreur est si critique qu'il n'est pas possible de recommencer l'exécution à partir de l'endroit ou c'est produit l'exception. Celui qui génère l'exception décide qu'il n'y a pas de solution pour restaurer la situation et ne *veut* pas revenir en arrière.

L'autre solution est appelée *restauration*. Cela signifie que le gestionnaire d'exception est censé agir pour corriger la situation, et ainsi la méthode incriminée est de nouveau exécutée avec un succès présumé. Si vous voulez utilisez la restauration, cela signifie que vous voulez que l'exécution continue après le traitement de l'exception. Dans cette optique votre exception est résolue par un appel de méthode qui est la façon dont vous devriez résoudre vos problèmes en Java si vous voulez avoir un comportement de Type restauration dans la gestion de vos exceptions. autrement placez votre bloc try dans un bloc while qui continuera a exécuter le bloc **try** tant que le résultat ne sera pas satisfaisant.

Historiquement, les programmeur utilisant des systèmes d'exploitation qui supportaient la gestion des exceptions restauratrice finissaient par utiliser un code qui ressemblait à celui de la terminaison laissant de côté la restauration. Bien que la restauration ait l'air attractive au départ elle n'est pas aisée à mettre en oeuvre. La raison dominante est le *couplage* que cela générait : votre gestionnaire d'exception doit être conscient de l'endroit ou est générée l'exception et contenir du code générique

indépendant de la localisation de sa génération. Cela rend le code difficile à écrire et à maintenir, surtout dans le cadre de grands systèmes où les exceptions peuvent surgir de nulle part.

Créez vos propres Exceptions

Vous n'êtes pas obligés de vous cantonner aux exceptions Java. Ceci est important car vous aurez souvent besoin de créez vos exceptions pour distinguer une erreur que votre librairie est capable de générer, mais qui n'a pas été prévue lorsque la hiérarchie des exceptions Java a été pensée.
Pour créer votre propre classe d'exception, vous devez hériter d'un type d'exception déjà existant, de préférence une qui est proche de votre nouvelle exception (parfois ce n'est pas possible). La façon la plus simple de créer une exception est de laisser le compilateur créer le constructeur par défaut, ainsi vous n'avez pas besoin d'écrire de code.

```
//: c10:SimpleExceptionDemo.java
// Inheriting your own exceptions.
class SimpleException  extends Exception {}
public  class SimpleExceptionDemo {
  public  void f()  throws SimpleException {
    System.out.println(
      " throwsing SimpleException from f()");
     throws new SimpleException ();
  }
  public  static  void main(String[] args) {
    SimpleExceptionDemo sed =
      new SimpleExceptionDemo();
    try {
      sed.f();
    } catch(SimpleException e) {
      System.err.println("Caught it!");
    }
  }
} ///:~
```

Quand le compilateur crée le constructeur par défaut, celui ci automatiquement appelle le constructeur de base par défaut de la classe. Bien sûr vous n'obtenez pas un constructeur **SimpleException(String)** par défaut mais ce constructeur n'est guère utilisé. Comme vous le constaterez la chose la plus importante à propos d'une exception est le nom de sa classe, donc la plupart du temps une exception comme celle décrite au dessus est satisfaisante.
Ici le résultat est envoyé vers le flux d'*erreur standard* de la console en écrivant dans **System.err**. C'est habituellement un meilleur endroit pour diriger les informations sur les erreurs que **System.Out**, qui peut être re-dirigé. Si vous envoyez le flux de sortie vers **System.err** il ne sera pas re-dirigé de la même façon que **System.out** donc l'utilisateur pourra le remarquer plus aisément.
Créer une classe d'exception dont le constructeur a aussi un constructeur qui prend un argument de type **string** est assez simple :

```
//: c10:FullConstructors.java
// Inheriting your own exceptions.
```

```
class MyException  extends Exception {
  public MyException() {}
  public MyException(String msg) {
    super(msg);
  }
}
public  class FullConstructors {
  public  static void f()  throws MyException {
    System.out.println(
      "throwing MyException from f()");
    throw  new MyException();
  }
  public  static void g()  throws MyException {
    System.out.println(
      "throwing MyException from g()");
    throw  new MyException( "Originated in g()");
  }
  public  static  void main(String[] args) {
    try {
      f();
    } catch(MyException e) {
      e.printStackTrace(System.err);
    }
    try {
      g();
    } catch(MyException e) {
      e.printStackTrace(System.err);
    }
  }
} ///:~
```

Le code ajouté est minime, deux constructeurs qui définissent la façon dont **MyException** est crée. Dans le second constructeur, le constructeur de base avec un argument de type **String** est appelé de façon explicité avec le mot clé **super**.

L'information de la pile des appels est redirigée vers System.err ainsi on notera plus simplement dans l'événement que system.out a été redirigé.

La sortie du programme est :

```
throwing MyException from f()
MyException
        at FullConstructors.f(FullConstructors.java:16)
        at FullConstructors.main(FullConstructors.java:24)
throwing MyException from g()
MyException: Originated in g()
        at FullConstructors.g(FullConstructors.java:20)
        at FullConstructors.main(FullConstructors.java:29)
```

Vous Pouvez noter l'absence de message de détail dans le **MyException** généré depuis **f()**.

La création d'exception personnalisées peut être plus poussée. Vous pouvez ajouter des constructeurs et des membres supplémentaires.

```
//: c10:ExtraFeatures.java
// Further embellishment of exception classes.
class MyException2  extends Exception {
  public MyException2() {}
  public MyException2(String msg) {
    super(msg);
  }
  public MyException2(String msg,  int x) {
    super(msg);
    i = x;
  }
  public  int val() { return i; }
  private int i;
}
public  class ExtraFeatures {
  public  static void f() throws MyException2 {
    System.out.println(
      "throwing MyException2 from f()");
    throw  new MyException2();
  }
  public  static void g() throws MyException2 {
    System.out.println(
      "throwing MyException2 from g()");
    throw  new MyException2( "Originated in g()");
  }
  public  static void h()  throws MyException2 {
    System.out.println(
      "throwing MyException2 from h()");
    throw  new MyException2(
      "Originated in h()", 47);
  }
  public  static  void main(String[] args) {
    try {
      f();
    } catch(MyException2 e) {
      e.printStackTrace(System.err);
    }
    try {
      g();
    } catch(MyException2 e) {
      e.printStackTrace(System.err);
    }
    try {
      h();
    } catch(MyException2 e) {
```

```
        e.printStackTrace(System.err);
        System.err.println( "e.val() = " + e.val());
      }
    }
} ///:~
```

Un membre *i* a été ajouté, avec une méthode qui lit cette donnée et un constructeur supplémentaire qui l'initialise. Voici la sortie :

```
Throwing MyException2 from f()
MyException2
        at ExtraFeatures.f(ExtraFeatures.java:22)
        at ExtraFeatures.main(ExtraFeatures.java:34)
Throwing MyException2 from g()
MyException2: Originated in g()
        at ExtraFeatures.g(ExtraFeatures.java:26)
        at ExtraFeatures.main(ExtraFeatures.java:39)
Throwing MyException2 from h()
MyException2: Originated in h()
        at ExtraFeatures.h(ExtraFeatures.java:30)
        at ExtraFeatures.main(ExtraFeatures.java:44)
e.val() = 47
```

Puisqu'une exception est juste une sorte d'objet vous pouvez continuer à enrichir vos classes d'exceptions mais gardez à l'esprit que toutes ces améliorations peuvent êtres ignorées par les clients programmeurs utilisant vos paquettages puisqu'ils vont probablement regarder quelle exception pourra être générée et rien de plus. (C'est de cette façon que sont utilisées le plus souvent les librairies d'exceptions Java).

Spécifier des Exceptions

En java vous êtes obligé de fournir au programmeur qui appelle vos méthodes la liste des exceptions pouvant être générées. C'est correct, en effet le programmeur peut ainsi exactement savoir quel code écrire pour attraper toute exception potentielle. Bien sur si le code source est disponible le programmeur pourra y jeter un Sil et rechercher le mot clé **throw**, mais souvent une librairie est livrée sans les sources. Pour éviter que Cela soit un problème java fournit une syntaxe (et vous *oblige* à l'utiliser) qui vous permet d'informer formellement le programmeur client quelle exceptions est générée par cette méthode, pour que le programmeur puisse les gérer. c'est la *spécification des exceptions* et cela fait partie intégrante de la déclaration de la méthode, elle apparaît après la liste des arguments.

La spécification des exceptions utilise un nouveau mot clé **throws** suivi par la liste des type d'exceptions possibles. Vos définitions de méthodes pourront ressembler à ceci :

```
void f()  throws TooBig, TooSmall, DivZero {  //...
```

Si vous écrivez

```
void f() {  // ...
```

Cela signifie qu'aucune exception en pourra être générée par la méthode. (*Excepté* les exceptions de type **RuntimeException**, qui peut être générée n'importe où nous verrons cela plus tard.)
Vous ne pouvez pas mentir à propos de la spécification des exceptions, si votre méthode génère des exceptions sans les gérer le compilateur le détectera et vous dira que vous devez soit gérer ces exceptions ou indiquer en spécifiant que cette exception peut être générée dans votre méthode. En Forçant la spécification des exceptions de haut en bas ,Java garantit que la correction des exceptions est assurée au *moment de la compilation*.
Il y a un endroit ou vous pouvez tricher : vous déclarez générer une exception que vous ne n'allez pas générer. Le compilateur vous prend au pied de la lettre, et force l'utilisateur de la méthode à agir comme si elle pouvait générer l'exception. Ceci à l'effet positif d'être un réceptacle pour cette exception, ainsi vous pourrez générer l'exception sans devoir modifier le code existant. C'. est aussi important pour créer des classes **abstraites** et des **interfaces** dont les classes dérivées ou les implémentations puissent générer des exceptions.

Attraper n'importe quelle exception

Il est possible de créer un gestionnaire qui intercepte n'importe quel type d'exception. Ceci est réalisé en interceptant le classe de base d'exception **Exception** (Il y a d'autre types d'exception de base mais **Exception** est celle qui est pertinent pour tout les types d'activités de programmation).

```
catch(Exception e) {
  System.err.println("Caught an exception");
}
```

Ce code interceptera toute les exceptions donc si vous l'utilisez vous voudrez le placer à la *fin* de votre liste de gestionnaires pour éviter la préemption sur certains gestionnaires qui pourraient dans l'autre cas le suivre.
Comme la classe **Exception** est la base de toute les classes d'exception qui sont utiles au programmeur, vous n'avez pas beaucoup d'informations spécifiques sur l'exception, mais vous pouvez utiliser les méthode venant de *son* type de base **Throwable** :
String getMessage() String getLocalizedMessage() Donne le message détaillé ou un message adapté à cette localisation particulière.
String toString() Retourne une courte description de l'objet Throwable ,incluant le message détaillé si il y en a un.
void printStackTrace() void printStackTrace(PrintStream) void printStackTrace(PrintWriter) Imprime l'objet throwable ainsi que la trace de la pile des appels de l'objet Throwable. La pile d'appels vous montre la séquence d'appels de méthodes qui vous ont conduit à l'endroit où l'exception a été levée. La première version imprime les informations vers la sortie standard la seconde et la troisième vers un flux de votre choix (Dans le chapitre 11, vous comprendrez pourquoi il y a deux types de flux).
Throwable fillInStackTrace() Enregistre des informations liées à cet objet **Throwable** concernant l'état de la « stack frame ». Utile quand une application relance une exception (nous approfondirons cela plus loin).
En plus, vous pouvez utiliser des méthodes provenant de la classe mère de **Throwable** qui est objet

(ancêtre de tout type de base). Celle qui pourrait être utile pour les exceptions est **getClass()** qui retourne un objet représentant la classe de cet objet. Vous pouvez interroger cet objet de **Class** avec les méthodes **getName()** or **toString()**. Vous pouvez aussi faire des choses plus sophistiquées avec l'objet **Class** qui ne sont nécessaires dans la gestion des exceptions. Les objets **Class** seront étudiés plus loin dans ce livre.

Voici un exemple qui montre un emploi simple de la méthode d'**Exception** de base :

```
//: c10:ExceptionMethods.java
// Demonstrating the Exception Methods.
public class ExceptionMethods {
  public static void main(String[] args) {
    try {
      throw new Exception( "Here's my Exception");
    } catch(Exception e) {
      System.err.println("Caught Exception");
      System.err.println(
        "e.getMessage(): " + e.getMessage());
      System.err.println(
        "e.getLocalizedMessage(): " +
         e.getLocalizedMessage());
      System.err.println("e.toString(): " + e);
      System.err.println( "e.printStackTrace():");
      e.printStackTrace(System.err);
    }
  }
} ///:~
```

La sortie du programme est :

```
Caught Exception
e.getMessage(): Here's my Exception
e.getLocalizedMessage(): Here's my Exception
e.toString(): java.lang.Exception:
   Here's my Exception
e.printStackTrace():
java.lang.Exception: Here's my Exception
at ExceptionMethods.main(ExceptionMethods.java:7)
java.lang.Exception:
   Here's my Exception
at ExceptionMethods.main(ExceptionMethods.java:7)
```

Vous pouvez observer que les méthodes donnent successivement plus d'informations, chacune est effectivement une évolution de la précédente.

Relancer une exception

Quelques fois vous voudrez relancer une exception que vous venez d'intercepter, particulièrement

quand vous utilisez la classe **Exception** pour attraper n'importe quelle exception. Comme vous avez déjà la référence de l'exception courante vous pouvez tout simplement la re-lancer.

```
catch(Exception e) {
  System.err.println( "An exception was thrown");
  throw e;
}
```

La redirection de l'exception envoie l'exception au gestionnaire d'exception de contexte supérieur le plus proche. Tout autre clause **catch** placée après dans le même bloc **try** est ignorée. De plus, toute information concernant l'objet exception est préservée, donc le gestionnaire d'exception du niveau supérieur qui peut attraper ce type d'exception spécifique peut extraire toute les informations de cet objet.

Si vous redirigez simplement l'exception courante, l'information que vous restituerez fera référence aux origines de l'exception et non de l'endroit ou vous la redirigez. Si vous voulez placez de nouvelles informations dans la pile des appels, vous pouvez le faire en appelant la méthode **fillInStackTrace()**, qui retourne un objet exception qu'elle a crée en ajoutant les informations actuelles de la pile dans l'ancien objet exception. Voyons à quoi cela ressemble.

```
//: c10:Rethrowing.java
// Demonstrating fillInStackTrace()
public  class Rethrowing {
  public  static void f()  throws Exception {
    System.out.println(
      "originating the exception in f()");
    throw  new Exception( "thrown from f()");
  }
  public  static void g()  throws Throwable {
    try {
      f();
    } catch(Exception e) {
      System.err.println(
        "Inside g(), e.printStackTrace()");
      e.printStackTrace(System.err);
      throw e;  // 17
      // throw e.fillInStackTrace(); // 18
    }
  }
  public  static void
  main(String[] args) throws Throwable {
    try {
      g();
    } catch(Exception e) {
      System.err.println(
        "Caught in main, e.printStackTrace()");
      e.printStackTrace(System.err);
    }
  }
} ///:~
```

Les numéros des lignes importantes sont marquées dans les commentaires, avec la ligne 17 non mise en commentaire (cf. le code ci-dessus) l'exécution produit comme sortie :

```
originating the exception in f()
Inside g(), e.printStackTrace()
java.lang.Exception: thrown from f()
        at Rethrowing.f(Rethrowing.java:8)
        at Rethrowing.g(Rethrowing.java:12)
        at Rethrowing.main(Rethrowing.java:24)
Caught in main, e.printStackTrace()
java.lang.Exception: thrown from f()
        at Rethrowing.f(Rethrowing.java:8)
        at Rethrowing.g(Rethrowing.java:12)
        at Rethrowing.main(Rethrowing.java:24)
```

Ainsi la trace de la pile d'exception se souvient toujours de sont vrai point d'origine, sans tenir compte du nombre de fois qu'il sera relancé.
Avec la ligne 17 mise en commentaire et la ligne 18 décommentée **fillInStackTrace()** est utilisée ce qui donne le résultat suivant :

```
originating the exception in f()
Inside g(), e.printStackTrace()
java.lang.Exception: thrown from f()
        at Rethrowing.f(Rethrowing.java:8)
        at Rethrowing.g(Rethrowing.java:12)
        at Rethrowing.main(Rethrowing.java:24)
Caught in main, e.printStackTrace()
java.lang.Exception: thrown from f()
        at Rethrowing.g(Rethrowing.java:18)
        at Rethrowing.main(Rethrowing.java:24)
```

Grâce à **fillInStackTrace()** la ligne 18 devient l'origine de l'exception.
La classe **Throwable** doit apparaître dans la spécification d'exception pour **g()** et **main()** parce que **fillInStackTrace()** produit une référence à un objet **Throwable**. Comme **Throwable** est la classe de base de toute **Exception** il est possible d'avoir un objet de type **Throwable** qui ne soit *pas* une **Exception** et donc que le gestionnaire d'exception du **main()** ne le traite pas. Pour être sur que tout soit correct le compilateur force une spécification d'exception pour **Throwable**. Par exemple l'exception dans le programme suivant n'est *pas* interceptée dans **main()** :

```
//: c10:ThrowOut.java
public  class ThrowOut {
  public  static void
  main(String[] args) throws Throwable {
    try {
      throw  new Throwable();
    } catch(Exception e) {
      System.err.println("Caught in main()");
    }
  }
```

```
} ///:~
```

Il est aussi possible de rediriger une exception différente de celle que vous avez intercepté. En procédant ainsi vous obtenez une effet similaire a l'utilisation de **fillInStackTrace()** . l'information sur l'emplacement originel de l'exception est perdu, et l'information qu'il vous reste est celle relative au nouveau **throw()** :

```
//: c10:RethrowNew.java
// Rethrow a different object
// from the one that was caught.
class OneException  extends Exception {
  public OneException(String s) {  super(s); }
}
class TwoException  extends Exception {
  public TwoException(String s) {  super(s); }
}
public  class RethrowNew {
  public  static void f()  throws OneException {
    System.out.println(
      "originating the exception in f()");
    throw  new OneException( "thrown from f()");
  }
  public  static  void main(String[] args)
  throws TwoException {
    try {
      f();
    } catch(OneException e) {
      System.err.println(
        "Caught in main, e.printStackTrace()");
      e.printStackTrace(System.err);
      throw  new TwoException( "from main()");
    }
  }
} ///:~
```

Le résultat de l'exécution est :

```
originating the exception in f()
Caught in main, e.printStackTrace()
OneException: thrown from f()
        at RethrowNew.f(RethrowNew.java:17)
        at RethrowNew.main(RethrowNew.java:22)
Exception in thread  "main" TwoException: from main()
        at RethrowNew.main(RethrowNew.java:27)
```

L'exception finale sait seulement qu'elle vient de **main()** et non de **f()**.
Vous n'avez pas à vous soucier de la destruction des exceptions précédentes. Ce sont tout des objets crées avec l'instruction **new()**, le garbage collector les détruits donc automatiquement.

Les exceptions Java standard

La classe Java **Throwable** décrit tout ce qui peut être généré comme exception. Il y a deux sortes d'objets **Throwable** (« Sortes de » = « Héritées de »). Error représente les erreurs systèmes et de compilation dont vous n'avez pas à vous soucier (excepté quelques cas spécifiques). **Exception** est le type de base qui peut être généré à partir de n'importe quelle méthode de classe de librairie Java standard. Les **Exceptions** sont donc le type de base qui intéresse le programmeur Java.

La meilleure façon d'avoir un aperçu des exceptions est de lire la documentation HTML de Java qui est disponible à *java.sun.com*. Il est utile de le faire juste pour s'apercevoir de la variété des exception, mais vous verrez rapidement que rien à part le nom ne distingue une exception d'une autre. Comme le nombre d'exceptions Java continue de s'accroître il est inutile d'en imprimer la liste. Chaque nouvelle librairie que vous achèterez à un éditeur aura probablement ses propres exceptions. Le point important à comprendre est le concept et ce que vous devez faire avec les exceptions.

Le principe de base est que le nom de l'exception représente le problème qui est apparu, et le nom de l'exception est censé être relativement explicite. Toutes les exceptions ne sont pas toutes définies dans **java.lang** ; certaines ont été crées pour être utilisées dans d'autres libraires telles que **util**, **net** et **io** ce que l'on peut voir à partir du nom complet des classes dont elles ont héritées. Ainsi toutes les exception I/O (Entrée/Sortie) héritent de **java.io.IOException**.

Le cas particulier RuntimeException

Le premier exemple du chapitre était :

```
if(t == null)
  throw new NullPointerException();
```

Cela peut paraître effrayant de penser que l'on devra vérifier le **null** pour chaque référence passée à une méthode(puisque on ne sait pas si l'appelant à passé une référence valide). Heureusement vous n'avez pas à le faire. C'est une des taches standard de run-time checking que Java exécute pour vous, et si un appel est fait à une référence contenant la valeur **null**, Java va automatiquement générer une **NullPointerException**. Donc le code ci-dessus est superflu.

Il y a tout un groupe d'exceptions qui sont dans cette catégorie. Elles sont générées automatiquement par Java et vous n'avez pas besoin de les inclure dans vos spécifications d'exception. Elles sont regroupées dans une seule classe de base nommée RuntimeExceptions, qui est un parfait exemple de l'héritage : cela établit une famille de types qui ont des caractéristiques et des comportements communs. Aussi vous n'avez jamais besoin de spécifier qu'une méthode peut générer une RuntimeException puisque c'est géré. Puisque cela indique la présence de bogues vous n'interceptez jamais de **RuntimeException** c'est géré automatiquement. Si vous étiez obligés de tester les RuntimeException votre code pourrait devenir illisible. Bien que vous n'interceptiez pas de RuntimeException dans vos propre paquettages vous pourriez décider de générer des RuntimeException.

Que se passe t'il quand vous n'interceptez pas de telles exceptions ? Comme le compilateur ne demande pas de spécifications d'exceptions pour celles ci il est probable qu'une **RuntimeException** puisse remonter jusqu'à votre méthode **main()** sans être interceptée. Pour voir ce qui se passe dans ce cas là essayez l'exemple suivant :

```
//: c10:NeverCaught.java
```

```
// Ignoring RuntimeExceptions.
public  class NeverCaught {
  static  void f() {
    throw  new RuntimeException( "From f()");
  }
  static  void g() {
    f();
  }
  public  static  void main(String[] args) {
    g();
  }
} ///:~
```

vous pouvez déjà voir qu'une **RuntimeException** (ou tout ce qui en hérite) est un cas particulier, puisque le compilateur ne demande pas de spécifications pour ce type.
La trace est :

```
Exception in thread "main"
java.lang.RuntimeException: From f()
        at NeverCaught.f(NeverCaught.java:9)
        at NeverCaught.g(NeverCaught.java:12)
        at NeverCaught.main(NeverCaught.java:15)
```

La réponse est donc : Si une **RuntimeException** arrive jusqu'à la méthode **main()** sans être interceptée, **PrintStackTrace()** est appelée pour cette exception à la sortie du programme.
Gardez à l'esprit que vous pouvez ignorer seulement les **RuntimeException** dans vos développements puisque la gestion de toutes les autres exceptions est renforcée par le compilateur. Le raisonnement est qu'une **RuntimeRxception** représente une erreur de programmation.

1. Une erreur que vous ne pouvez pas intercepter(recevoir une référence transmise par un programme client avec la valeur null par exemple).
2. Une erreur que vous auriez du prévoir dans votre code (comme **ArrayIndexOutOfBoundsException** ou vous auriez du faire attention à la taille du tableau).

Vous pouvez voir quel bénéfice apportent les exceptions dans ce cas puisque cela aide le processus de déboguage.
Il est intéressant de noter que vous ne pouvez pas ranger le gestion des exceptions Java dans un outil à utilisation unique. Oui c'est fait pour gérer c'est affreuses run-time error qui vont apparaître à causes de forces hors de contrôle de votre code mais c'est aussi essentiel pour certains types de bogues de développement que le compilateur ne peut détecter.

Faire le ménage avec finally

Il y a souvent un morceau de code que vous voulez exécuter qu'une exception ou pas soit générée au coeur d'un bloc **try**. C'est généralement destiné à des opérations autre que le rafraîchissement de la mémoire (puisque le garbage collector s'en occupe). Pour faire ceci il faut utiliser **finally** à la fin de chaque gestionnaire d'exceptions. L'image complète d'un paragraphe gestion d'exception est :

```
try {
  // The guarded region: Dangerous activities
  // that might throw A, B, or C
} catch(A a1) {
  // Handler for situation A
} catch(B b1) {
  // Handler for situation B
} catch(C c1) {
  // Handler for situation C
} finally {
  // Activities that happen every time
}
```

Pour démontrer que la clause **finally** est toujours exécutée lancer ce programme :

```
//: c10:FinallyWorks.java
// The finally clause is always executed.
class ThreeException  extends Exception {}
public  class FinallyWorks {
  static  int count = 0;
  public  static  void main(String[] args) {
    while(true) {
      try {
        // Post-increment is zero first time:
        if(count++ == 0)
          throw  new ThreeException();
        System.out.println("No exception");
      } catch(ThreeException e) {
        System.err.println("ThreeException");
      } finally {
        System.err.println( "In finally clause");
        if(count == 2)  break; // out of "while"
      }
    }
  }
} ///:~
```

Ce programme vous donne aussi une aide pour comprendre comment gérer le fait que Java(comme les exceptions en C++) ne vous permet de reprendre l'exécution de l'endroit où l'exception a été générée, comme vu plus tôt. Si vous placez votre bloc **try** dans une boucle, vous pouvez établir un condition qui doit être validée avant de continuer le programme. Vous pouvez aussi ajouter un compteur de type **static** ou d'autres périphériques afin de permettre à la boucle différente approches avant d'abandonner. De cette façon vous pouvez obtenir un plus grand niveau de robustesse dans vos programmes.

La sortie est :

```
ThreeException
In finally clause
```

```
No exception
In finally clause
```

Qu'une exception soit générée ou pas la clause **finally** est exécutée.

À Quoi sert le finally ?

Dans un langage sans ramasse-miettes [Garbage Collector] *et* sans appel automatique aux destructeurs, **finally** est important parce qu'il permet au programmeur de garantir la libération de la mémoire indépendamment de ce qui se passe dans le bloc **try**. Mais Java a un ramasse-miettes. Donc il n'a aucun destructeurs à appeler. Donc quand avez vous besoin de la clause **finally** en Java ?

Finally est nécessaire quand vous avez besoin de remettre à l'état original *autre* chose que le mémoire. C'est une sorte de nettoyage comme un fichier ouvert une connexion réseau quelque chose affiché à l'écran ou un switch dans le monde extérieur comme modélisé dans l'exemple suivant :

```
//: c10:OnOffSwitch.java
// Why use finally?
class Switch {
  boolean state =  false;
  boolean read() {  return state; }
  void on() { state =  true; }
  void off() { state =  false; }
}
class OnOffException1  extends Exception {}
class OnOffException2  extends Exception {}
public  class OnOffSwitch {
  static Switch sw =  new Switch();
  static  void f() throws
    OnOffException1, OnOffException2 {}
  public  static  void main(String[] args) {
    try {
      sw.on();
      // Code that can throw exceptions...
      f();
      sw.off();
    } catch(OnOffException1 e) {
      System.err.println("OnOffException1");
      sw.off();
    } catch(OnOffException2 e) {
      System.err.println("OnOffException2");
      sw.off();
    }
  }
} ///:~
```

L'objectif ici est d'être sûr que le switch est Off quand la méthode **main()** se termine donc

sw.off() est placée à la fin de **main()** et à la fin de chaque gestionnaire d'exception mais il se peut qu'une exception soit générée et qu'elle ne soit pas interceptée ici. Vous pouvez placer le code de nettoyage dans un seul endroit le bloc **finally** :

```
//: c10:WithFinally.java
// Finally Guarantees cleanup.
public class WithFinally {
  static Switch sw = new Switch();
  public static void main(String[] args) {
    try {
      sw.on();
      // Code that can throw exceptions...
      OnOffSwitch.f();
    } catch(OnOffException1 e) {
      System.err.println("OnOffException1");
    } catch(OnOffException2 e) {
      System.err.println("OnOffException2");
    } finally {
      sw.off();
    }
  }
} ///:~
```

Ici le **sw.off()** à été placée à un endroit unique ou on est sûr qu'il sera exécutée quoi qu'il arrive.

Même dans le cas où l'exception n'est pas interceptée dans l'ensemble des clauses **catch**, **finally** sera exécuté avant que le mécanisme de gestion d'exception recherche un gestionnaire de plus haut niveau.

```
//: c10:AlwaysFinally.java
// Finally is always executed.
class FourException extends Exception {}
public class AlwaysFinally {
  public static void main(String[] args) {
    System.out.println(
      "Entering first try block");
    try {
      System.out.println(
        "Entering second try block");
      try {
        throw new FourException();
      } finally {
        System.out.println(
          "finally in 2nd try block");
      }
    } catch(FourException e) {
      System.err.println(
        "Caught FourException in 1st try block");
    } finally {
```

```
        System.err.println(
          "finally in 1st try block");
      }
    }
  } ///:~
```

La sortie de ce programme vous montre ce qui arrive :

```
Entering first try block
Entering second try block
finally in 2nd  try block
Caught FourException in 1st try block
finally in 1st  try block
```

Le mot clé **finally** sera aussi exécuté dans des situations ou les mots clés **break** et **continue** sont impliqués. Notez qu'avec break continue et finally il n'y a pas besoin de **goto** en Java.

Le défaut : l'exception perdue

En général l'implémentation de Java est de grande qualité mais malheureusement il y a un petit défaut. Bien qu'une exception soit une indication de crise dans votre programme et ne doit jamais être ignorée il est possible qu'une exception soit perdue. Cela est possible dans une certaine configuration de la clause **finally** :

```
//: c10:LostMessage.java
// How an exception can be lost.
class VeryImportantException  extends Exception {
  public String toString() {
    return  "A very important exception!";
  }
}
class HoHumException  extends Exception {
  public String toString() {
    return  "A trivial exception";
  }
}
public  class LostMessage {
  void f()  throws VeryImportantException {
    throw  new VeryImportantException();
  }
  void dispose()  throws HoHumException {
    throw  new HoHumException();
  }
  public  static  void main(String[] args)
      throws Exception {
    LostMessage lm = new LostMessage();
```

```
      try {
        lm.f();
      } finally {
        lm.dispose();
      }
    }
} ///:~
```

Le résultat est :

```
Exception in thread  "main" A trivial exception
    at LostMessage.dispose(LostMessage.java:21)
    at LostMessage.main(LostMessage.java:29)
```

Vous pouvez voir qu'il ne reste aucune trace de **VeryImportantException** qui est tout simplement remplacée par **HoHumException** dans la clause **finally**. Ceci est un faille sérieuse puisqu'il signifie qu'une exception peut être complètement perdue et bien plus difficile et subtile à détecter que les autres. C++ lui au contraire traite la situation en considérant qu'une seconde exception générée avant la première est une erreur de programmation. Peut être qu' une future version de Java corrigera ce défaut (autrement vous pouvez exclure du bloc **try-catch** toute méthode susceptible de générer une exception telle que **dispose()**).

Restriction d'Exceptions

Quand vous surchargez une méthode vous ne pouvez générer que les exceptions qui ont été spécifiées dans la classe de base de la version de la méthode. c'est une restriction très utile puisqu'elle garantit que le code qui fonctionne avec la version de classe de base de la méthode fonctionnera avec tout les objets dérivant de cette classe (un concept fondamental de la POO),incluant les exceptions.
Cet exemple démontre les restrictions imposée (à la compilation) pour les exceptions :

```
//: c10:StormyInning.java
// Overridden methods may throw only the
// exceptions specified in their base-class
// versions, or exceptions derived from the
// base-class exceptions.
class BaseballException  extends Exception {}
class Foul  extends BaseballException {}
class Strike  extends BaseballException {}
abstract  class Inning {
  Inning() throws BaseballException {}
  void event ()  throws BaseballException {
   // Doesn't actually have to throw anything
  }
  abstract  void atBat()  throws Strike, Foul;
  void walk() {} // Throws nothing
}
```

```
class StormException  extends Exception {}
class RainedOut  extends StormException {}
class PopFoul  extends Foul {}
interface Storm {
  void event()  throws RainedOut;
  void rainHard()  throws RainedOut;
}
public  class StormyInning  extends Inning
    implements Storm {
  // OK to add new exceptions for
  // constructors, but you must deal
  // with the base constructor exceptions:
  StormyInning() throws RainedOut,
    BaseballException {}
  StormyInning(String s) throws Foul,
    BaseballException {}
 // Regular methods must conform to base class:
 //! void walk() throws PopFoul {} //Compile error
 // Interface CANNOT add exceptions to existing
  // methods from the base class:
//! public void event() throws RainedOut {}
  // If the method doesn't already exist in the
  // base class, the exception is OK:
  public  void rainHard()  throws RainedOut {}
 // You can choose to not throw any exceptions,
  // even if base version does:
  public  void event() {}
  // Overridden methods can throw
  // inherited exceptions:
  void atBat()  throws PopFoul {}
  public  static  void main(String[] args) {
    try {
      StormyInning si = new StormyInning();
      si.atBat();
    } catch(PopFoul e) {
      System.err.println("Pop foul");
    } catch(RainedOut e) {
      System.err.println("Rained out");
    } catch(BaseballException e) {
      System.err.println("Generic error");
    }
    // Strike not thrown in derived version.
    try {
      // What happens if you upcast?
      Inning i = new StormyInning();
      i.atBat();
      // You must catch the exceptions from the
```

```
        // base-class version of the method:
      } catch(Strike e) {
        System.err.println("Strike");
      } catch(Foul e) {
        System.err.println("Foul");
      } catch(RainedOut e) {
        System.err.println("Rained out");
      } catch(BaseballException e) {
        System.err.println(
          "Generic baseball exception");
      }
    }
  } ///:~
```

Dans **Inning()** vous pouvez voir qu'à la fois le constructeur et la méthode **event()** déclarent qu'elles peuvent lancer une exception alors qu'elles ne le feront jamais. C'est tout à fait légitime puisqu'elles vous laissent la possibilité de forcer l'utilisateur à attraper des exceptions dans des versions surchargées de **event()**. Le même principe s'applique pour les méthodes **abstraites** [abstract].

L'**interface Storm** est intéressante parce qu'elle contient une méthode qui est définie dans **Inning** et une autre qui ne l'est pas. Toutes génèrent un nouveau type d'exception, **RainedOut**. Quand vous regardez **StormyInning extends Inning implements Storm**. Vous voyez que la méthode **event()** de **Storm** *ne peut pas* changer l'interface de la méthode **event()** de **Inning**. Une fois encore cela est sensé car lorsque que vous travaillez avec la classe de base vous ne sauriez pas si vous attrapez la bonne chose. Bien sûr si un méthode décrite dans l'interface n'est pas dans la classe de base telle que **Rainhard()**, ainsi il n'y a aucun problème lorsqu'elle génèrent des exceptions.

La restriction sur les exceptions ne s'appliquent pas aux constructeurs. Vous pouvez voir dans **Stormyinning** un constructeur qui peut générer toutes les exceptions qu'il veut sans être contraint par le constructeur de base. Malgré tout puis un constructeur de classe de base doit bien être appelé à un moment ou à un autre (ici le constructeur par défaut est appelé automatiquement) le constructeur de la classe dérivée doit déclarer toute les exceptions du constructeur de base dans ses spécifications d'exception. un constructeur de classe dérivée ne peut pas intercepter les exceptions de sa classe de base.

La raison pour laquelle **StormyInning.Walk()** ne compilera pas est qu'elle génère une exception alors que Inning.walk() n'en génère pas. Si cela était permis vous pourriez écrire le code qui appelle **Inning.walk()** et qui n'aurait pas à gérer une seule exception mais lorsque vous substitueriez à **Inning** un objet d'un de ses classes dérivées des exceptions seraient levées et votre code serait interrompu. En forçant les méthodes de classes dérivées à se conformer aux spécifications d'exceptions des classes de base la substituabilité des objets est maintenue.

La méthode surchargée **event()** montre qu'un méthode héritée d'une classe peut choisir de ne pas générer d'exceptions même si la méthode de la classe de base le fait. Ceci est de nouveau bon puisque cela ne rend caduque aucun code qui a été écrit présumant que la classe de base génère des exceptions. La même logique s'applique à **atBat()** qui génère une exception **PopFoul**, une exception qui est dérivée de l'exception **Foul** par la version de base de **atBat()**. De cette façon si quelqu'un écrit du code qui fonctionne avec **Inning()** et qui appelle **atBat()** doit intercepter les exception **Foul**. Puisque **PopFoul** hérite de **Foul** le gestionnaire d'exception interceptera **PopFoul**.

Le dernier point intéressant est **main()** Ici vous pouvez voir que si vous avez à faire à un objet StormyInning le compilateur vous oblige à intercepter que les exceptions qui sont spécifiques à la

classe mais si vous caster l'objet le compilateur vous oblige (avec raison) à gérer toutes les exceptions du type de base. Toute ces contraintes produisent du code de gestion des exceptions beaucoup plus robustes.

Il est utile de réaliser que bien la spécifications des exceptions soit renforcée au moment de l'héritage par le compilateur, la spécification des exceptions n'est pas une partie d'un type d'une méthode, qui est composée seulement d'un nom de méthode et d'un type d'arguments. vous ne pouvez pas surcharger une méthode basée sur une spécification d'exception. De plus il n'est pas obligatoire qu'une exception spécifiée dans méthode d'une classe de base soit présente dans la méthode d'une classe en héritant. C'est assez différent des règles de l'héritage, où un méthode qui existe dans une classe de base doit aussi exister dans sa fille. Autrement dit l'interface de spécification des exceptions d'une classe ne peut que se restreindre alors que c'est le contraire pour l'interface de la classe durant l'héritage.

Les constructeurs

Quand on écrit du codes avec des spécifications d'exceptions il y a une question particulière que l'on doit se poser « si une exception se produit est ce que la situation sera rétablie à l'état initial ? » dans la plupart des cas vous êtes sauvés mais dans celui des constructeurs il y a un problème. Le constructeur place l'objet dans un état de démarrage sain mais peut accomplir certaines opérations, telles que ouvrir un fichier- qui ne seront pas terminées tant que l'utilisateur n'aura pas terminé de travailler avec l'objet et appelé une certaine méthode de restauration. Si vous générez une exception dans le constructeur ces restaurations peuvent ne pas s'effectuer de façon appropriée. Cela signifie que vous devez être extrêmement prudent quand vous écrivez votre propre constructeur.

Comme vous avez venez d'apprendre le mot clé **finally** vous pouvez penser que c'est la solution mais ce n'est pas si simple, parce que Finally fait le nettoyage du code à chaque exécution même dans les cas ou vous ne le souhaiteriez pas. Aussi si vous faites la restauration dans le **finally** vous devez placer un indicateur qui vous indiquera de ne rien faire dans le finally si le constructeur c'est bien déroulé mais cela n'est pas particulièrement élégant (vous liez votre code d'un endroit à un autre) c'est mieux de ne pas faire ce genre d'opérations dans le **finally** à moins d'y être forcé.

Dans l'exemple suivant une classe appelée **InputFile** est crée elle vous permet d'ouvrir un fichier et d'en lire une ligne (convertie en **String**) à la fois . Elle utilise les classes **FileReader** et **Input-Buffer** de la librairie I/O standard de Java dont nous discuterons au chapitre 11 mais elles sont assez simple donc vous n'aurez probablement pas de problème à comprendre leur comportement de base :

```
//: c10:Cleanup.java
// Paying attention to exceptions
// in constructors.
import java.io.*;
class InputFile {
  private BufferedReader in;
  InputFile(String fname) throws Exception {
    try {
      in =
        new BufferedReader(
          new FileReader(fname));
```

```
        // Other code that might throw exceptions
      } catch(FileNotFoundException e) {
        System.err.println(
          "Could not open " + fname);
        // Wasn't open, so don't close it
        throw e;
      } catch(Exception e) {
        // All other exceptions must close it
        try {
          in.close();
        } catch(IOException e2) {
          System.err.println(
            "in.close() unsuccessful");
        }
        throw e; // Rethrow
      } finally {
        // Don't close it here!!!
      }
    }
    String getLine() {
      String s;
      try {
        s = in.readLine();
      } catch(IOException e) {
        System.err.println(
          "readLine() unsuccessful");
        s = "failed";
      }
      return s;
    }
    void cleanup() {
      try {
        in.close();
      } catch(IOException e2) {
        System.err.println(
          "in.close() unsuccessful");
      }
    }
  }
  public class Cleanup {
    public static void main(String[] args) {
      try {
        InputFile in =
          new InputFile( "Cleanup.java");
        String s;
        int i = 1;
        while((s = in.getLine()) != null)
```

```
        System.out.println(""+ i++ + color= ": " + s);
       in.cleanup();
      } catch(Exception e) {
       System.err.println(
        "Caught in main, e.printStackTrace()");
       e.printStackTrace(System.err);
      }
     }
    } ///:~
```

Le constructeur d'**InputFile** prend une **String** comme paramètre, qui contient le nom du fichier que vous voulez ouvrir, à l'intérieur du bloc **try** il crée un **FileReader**. Un **FileReader** n'est pas particulièrement utile à moins que vous ne le détourniez et l'utilisiez pour créer un **BuffereReader** avec lequel vous puissiez communiquer, un des bénéfices de l'**InputFile** est qu'il combine ses deux actions.

Si le constructeur de **FileReader** échoue il génère une **FileNotFoundException** qui doit être interceptée séparément puisque dans ce cas la vous ne voulez pas fermer le fichier qui n'a pas été ouvert. Toute *autre* clause catch doit fermer le fichier puisqu'il *a été* ouvert (bien sûr c'est plus délicat plus d'une méthode peut générer une **FileNotfoundException** dans ce cas vous pouvez vouloir séparer cela dans plusieurs blocs **try**). La méthode **close()** doit lancer une exception ce qui est donc essayé et capté bien que cela soit dans le bloc d'une autre clause **catch** - c'est juste une autre paire d'accolades pour le compilateur Java. Après l'exécution des opérations locales, l'exception est relancée, ce qui est adéquat puisque le constructeur a échoué, et vous ne voudriez pas que la méthode appelé ai a assumer que l'objet ai été correctement crée et soit valide.

Dans cet exemple, qui n'utilise pas la technique du drapeau mentionnée précédemment, la cause **finally** n'est définitivement pas la place pour **close()** (ndt. fermer) le fichier, puisque cela fermerait le fichier à chaque finalisation du constructeur. Comme nous désirons que le fichier soit ouvert tout le temps de la durée de vie de l'objet **InputFile** cela ne serait pas approprié.

La méthode **getLine()** renvoie un String contenant la prochaine ligne du fichier. Elle appelle **readLine()**, qui peut lancer une exception, mais cette exception est provoqué donc **getLine()** ne lance pas d'exceptions. Une des finalités de la conception des exceptions est soit de porter complètement une exception à ce niveau, de la porter partiellement et de passer la même exception (ou une différente), ou soit de la passer tout simplement. La passer, quand c'est approprié, peut certainement simplifier le codage. La méthode **getLine()** devient :

```
String getLine()  throws IOException {
 return in.readLine();
}
```

Mais bien sûr, l'appelant a maintenant la responsabilité de porter toute **IOException** qui puisse apparaître.

La méthode **cleanUp()** doit être appelée par l'utilisateur lorsqu'il a fini d'utiliser l'objet **InputFile()**. Ceci relâchera les ressources système (comme les poignées de file) qui étaient utilisées par les objets **BufferedReader** ou/et **FileReader**. Vous ne voudrez pas faire cela tant que vous n'aurez pas fini avec l'objet InputFile, au point de le laisser partir. Vous devrez penser a mettre une fonctionnalité de ce type dans une méthode **finaliste()**, mais comme il est mentionné dans le Chapitre 4 vous ne pouvez pas toujours être sûr que finaliste() sera appelé (même si vous *pouvez* être sûr de son appel, vous ne savez pas *quand*). C'est un des à-cotés de Java : tout nettoyage - autre que le net-

toyage de la mémoire - ne se lance pas automatiquement, donc vous devez informer le programmeur client qu'il est responsable, et garantir au mieux que possible que le nettoyage s'exécute par l'usage de **finalise()**.

Dans **Cleanup.java** un **InputFile** est crée pour ouvrir le même fichier source qui crée le programme, le fichier est lu une ligne à la fois, et les numéros de ligne sont ajoutés. Toutes les exceptions sont causées génériquement dans **main()**, si bien que vous pouvez choisir une plus grande finesse.

Un des bénéfices de cet exemple est de vous montrer pourquoi les exceptions sont abordés à ce point du livre - vous ne pouvez pas faire de l'E/S [I/O] sans utiliser les exceptions. Les exceptions sont tellement intégrés à la programmation en Java, spécialement parce que le compilateur les demande, que vous pouvez accomplir tellement peu de choses sans savoir comment travailler avec elles.

Indication d'Exception

Quand une exception est lancée, le système d'indication des exceptions regarde dans les « plus proches » identifiants dans l'ordre de leur écriture. Quand il trouve une correspondance, l'exception est considérée comme identifiée, et aucune autre recherche n'est lancée.

Identifier une exception ne requiert pas de correspondance exacte entre l'exception et sont identifiant. Une objet de classe dérivée pourra correspondre à un identifiant de la classe de base, comme montré dans cet exemple :

```
//: c10:Human.java
// Catching exception hierarchies.
class Annoyance  extends Exception {}
class Sneeze  extends Annoyance {}
public  class Human {
  public  static  void main(String[] args) {
    try {
      throw  new Sneeze();
    } catch(Sneeze s) {
      System.err.println("Caught Sneeze");
    } catch(Annoyance a) {
      System.err.println("Caught Annoyance");
    }
  }
} ///:~
```

L'exception **Sneeze** sera piégée par le premier **catch** (ndt. cause) qui correspondra - qui sera le premier, bien sûr. Cependant, si vous retirez la première cause du piège, laissant seulement :

```
try {
      throw  new Sneeze();
    } catch(Annoyance a) {
      System.err.println("Caught Annoyance");
    }
```

Le code sera toujours fonctionnel puisqu'il piège la classe de base de **Sneeze**. Placé d'une

autre manière, **catch(Annoyancee)** identifiera une **Annoyance** *ou tout autre classe qui en est dérivée*. C'est utile puisque si vous décidez d'ajouter plus d'exceptions dérivées à une méthode, alors le code du programmeur client n'aura pas besoin d'être modifié tant que le client piégera les exceptions de la classe de base.

Si vous essayez de « masquer » les exceptions de la classe dérivée en plaçant le piège de classe-de-base en premier, comme ceci :

```
try {
    throw  new Sneeze();
  } catch(Annoyance a) {
    System.err.println("Caught Annoyance");
  } catch(Sneeze s) {
    System.err.println("Caught Sneeze");
  }
```

le compilateur vous donnera un message d'erreur, en voyant que la cause d'identification de **Sneeze** ne sera jamais atteinte.

Recommandations pour les exceptions

Utilisez les exceptions pour :

1. Fixer le problème et appeler la méthode causant cette exception une nouvelle fois.
2. Corriger les choses et continuez sans ré-essayer la méthode.
3. Calculer quelques résultats alternatif à la place de ce que devrait produire la méthode.
4. Faire ce que vous désirez dans le contexte courant et relancer la *même* exception dans un contexte plus haut.
5. Faire ce que vous désirez dans le contexte courant et lancer une exception *différente* dans un contexte plus haut.
6. Terminer le programme.
7. Simplifier. (Si votre schéma d'exceptions rend les choses plus compliqués, alors il est douloureux et ennuyer à utiliser.)
8. Rendre votre librairie et programme plus sûr. (C'est un investissement à court-terme pour le déboguage, et un investissement à long-terme pour la robustesse de l'application.)

Résumé

La recherche amélioré d'erreur est l'une des puissantes voies qui peut augmenter la robustesse de votre code. La recherche d'erreur est une préoccupation fondamentale pour chaque programme que vous écrivez, mais c'est surtout important en Java, où l'un des buts primaire est de crée des composants de programme pour d'autre utilisations. **Pour créer un système robuste, chaque composant doit être robuste.**
Les buts de l'identification d'exception en Java sont de simplifier la création de programmes larges, reliables en utilisant le moins de code possible, et avec plus d'assurance que votre application n'ai

pas d'erreur non identifiée.

Les exceptions ne sont pas terriblement difficile à apprendre, et sont l'une des chose qui procurent des bénéfices immédiat à votre projet. Heureusement, Java impose tous les aspects des exceptions donc cela garantie qu'elle seront employés uniformément par à la fois les concepteurs de librairies et les programmeurs clients.

Exercices

Les solutions aux exercices sélectionnées peuvent être trouvés dans le document électronique *The Thinking in Java Annotated Solution Guide*,, disponible pour une modique somme à *www.BruceEckel.com*.

1. Créez une classe avec un **main()** qui lance un objet de la classe **Exception** dans un bloc **try**. Donnez au constructeur d'**Exception** un argument de type **String**. Piégez l'exception dans une cause **catch** et affichez l'argument du **String**. Ajoutez une cause **finally** et affichez un message pour prouver que vous vous situez là.

2. Créez votre propre classe d'exception utilisant le mot-clé **extends**. Écrivez un constructeur pour cette classe qui prendra un argument de type **String** et stockez le dedans l'objet par une référence au **String**. Écrivez une méthode qui imprime en sortie le **String** stocké. Créez une cause **try-catch** pour exercer votre nouvelle exception.

3. Écrivez une classe avec une méthode qui lance une exception du type de celle crée dans l'exercice 2. Éssayez de le compiler sans spécification d'exception afin de voir ce que dit le compilateur. Ajoutez la spécification d'exception adéquate. Essayez en sortie votre classe et son exception dans une cause **try-catch**.

4. Définissez une référence objet et initialisez la à **null**. Essayez d'appeler cette méthode grâce à cette référence. Maintenant enveloppez le code dans une clause **try-catch** afin de capter l'exception.

5. Créez une classe ayant deux méthodes, **f()** et **g( )**. Dans **g()**, lancez une exception d'un nouveau type que vous définirez. Dans **f()**, captez son exception et, dans la clause **catch**, lancez une exception différente (d'un second type que vous définirez). Testez votre code dans **main( )**.

6. Créez trois nouveaux types d'exceptions. Écrivez une classe avec une méthode qui lance les trois. Dans **main()**, appelez la méthode mais en utilisant seulement une seule clause **catch** qui piégera les trois types d'exceptions.

7. Écrivez un code pour générer et capter une **ArrayIndexOutOfBoundsException**.

8. Créez votre propre comportement de type-résurrection en utilisant une boucle **while** qui se répète tant qu'une exception n'est pas lancée.

9. Créez une hiérarchie à trois niveaux d'exceptions. Maintenant créez un classe-de-base **A** avec une méthode qui lance une exception à la base de votre hiérarchie. Héritez **B** depuis **A** et outrepassez la méthode afin qu'elle lance une exception au second niveau de votre hiérarchie. Répétez en faisant hériter la classe **C** de la classe **B**. Dans **main()**, créez un **C** et surtypez le en **A**, puis appelez la méthode.

10. Démontrez qu'un constructeur d'une classe-dérivé ne peut pas capter les exceptions

lancées par son constructeur de sa classe-de-base.

11. Montrez que **OnOffSwitch.java** peut échouer en lançant une **RuntimeException** dans le bloc **try**.

12. Montrez que **WithFinally.java** n'échoue pas en lançant une **RuntimeException** dans le bloc try.

13. Modifiez l'Exercice 6 en ajoutant une clause **finally**. Vérifiez que votre clause **finally** est exécutée, même si une **NullPointerException** est lancée.

14. Créez un exemple où vous utiliserez un drapeau pour contrôler si le nettoyage du code est appelé, comme décrit dans le second paragraphe après l'en-tête « Constructors ».

15. Modifiez **StormyInning.java** en ajoutant une exception de type **UmpireArgument**, et de les méthodes qui lancent cette exception. Testez la hiérarchie modifiée.

16. Enlevez la première cause d'identification (catch) dans **Human.java** et vérifiez que le code compile et s'exécute toujours.

17. Ajoutez un second niveau de perte d'exception à **LostMessage.java** afin que **HoHumException** soit lui-même remplacé par une troisième exception.

18. Dans le Chapitre 5, trouvez deux programmes appelés **Assert.java** et modifiez les pour lancer leur propres types d'exception au lieu d'imprimer vers **System.err**. Cette exception doit être une classe interne qui étends **RuntimeException**.

19. Ajoute un jeu d'exceptions appropriés à **c08:GreenhouseControls.java**.

[51]Le programmeur C peut regarder la valeur renvoyée par **printf()** en exemple à ceci.
[52] "2">C'est une amélioration significative sur la gestion d'exception de C++, qui ne piège pas les violations des spécifications d'exceptions avant l'exécution, quand ce n'est pas très utile.
[53]La gestion d'exception de C++ n'a pas de cause **finally** parcqu'elle est reliée au destructeur pour accomplir ce type de nettoyage.
[54]Un destructeur est une fonction qui est toujours appelé quand un objet devient inutilisé. Vous savez toujours exactement où et quand le constructeur est appelé. C++ possède des appels automatique au destructeur, mais le Pascal Objet de Delphi dans ces versions 1 et 2 n'en a pas (ce qui change la pensée et l'usage du concept d'un destructeur pour ce langage).<
[55]ISO C++ a ajouté des contraintes similaires qui nécessitent que les exceptions dérivées-de-méthode soient les même que, ou dérivées des, exceptions lancées par la méthode de la classe-de-base. C'est un cas dans lequel C++ est actuellement capable de rechercher des spécifications d'exception au moment de la compilation.
[56]En C++, un *destructeur* pourrait gérer cela pour vous.

Chapitre 11 - Le système d'E/S de Java

La création d'un bon système d'entrée/sortie, pour le designer du langage, est l'une des tâches les plus difficiles.

Cette difficulté est mise en évidence par le nombre d'approches différentes. Le défi semblant être dans la couverture de toutes les éventualités. Non seulement il y a de nombreuses sources et de réceptacles d'E/S avec lesquelles vous voudrez communiquer (fichiers, la console, connections réseau), mais vous voudrez converser avec elles de manières très différentes (séquentielle, accès-aléatoire, mise en mémoire tampon, binaire, caractère, par lignes, par mots, etc.).

Les designers de bibliothèque Java ont attaqué ce problème en créant de nombreuses classes. En fait, il y a tellement de classes pour le système d'E/S de Java que cela peut être intimidant au premier abord (ironiquement, le design d'E/S de Java prévient maintenant d'une explosion de classes). Il y a eu aussi un changement significatif dans la bibliothèque d'E/S après Java 1.0, quand la bibliothèque orientée-**byte** d'origine a été complétée par des classes d'E/S de base Unicode orientées-**char**. La conséquence étant qu'il vous faudra assimiler un bon nombre de classes avant de comprendre suffisamment la représentation de l'E/S Java afin de l'employer correctement. De plus, il est plutôt important de comprendre l'évolution historique de la bibliothèque E/S, même si votre première réaction est « me prenez pas la tête avec l'historique, montrez moi seulement comment l'utiliser ! » Le problème est que sans un point de vue historique vous serez rapidement perdu avec certaines des classes et lorsque vous devrez les utiliser vous ne pourrez pas et ne les utiliserez pas.

Ce chapitre vous fournira une introduction aux diverses classes d'E/S que comprend la bibliothèque standard de Java et la manière de les employer.

La classe File

Avant d'aborder les classes qui effectivement lisent et écrivent des données depuis des streams (flux), nous allons observer un utilitaire fournit avec la bibliothèque afin de vous assister lors des traitements de répertoire de fichiers.

La classe **File** possède un nom décevant — vous pouvez penser qu'elle se réfère a un fichier, mais pas du tout. Elle peut représenter soit le *nom* d'un fichier particulier ou bien les *noms* d'un jeu de fichiers dans un dossier. Si il s'agit d'un jeu de fichiers, vous pouvez faire appel a ce jeu avec la méthode **list()**, et celle-ci renverra un tableau de **String**. Il est de bon sens de renvoyer un tableau plutôt qu'une classe containeur plus flexible parce que le nombre d'éléments est fixé, et si vous désirez le listing d'un répertoire différent vous créez simplement un autre objet **File**. En fait, « CheminDeFichier ou FilePath » aurait été un meilleur nom pour cette classe. Cette partie montre un exemple d'utilisation de cette classe, incluant **l'interface** associée **FilenameFilter**.

Lister un répertoire

Supposons que vous désirez voir le listing d'un répertoire. L'objet **File** peut être listé de deux manières. Si vous appelez **list()** sans arguments, vous obtiendrez la liste complète du contenu de l'objet **File** . Pourtant, si vous désirez une liste restreinte — par exemple, cependant si vous voulez tous les fichiers avec une extension **.java** — à ce moment là vous utiliserez un « filtre de répertoire », qui est une classe montrant de quelle manière sélectionner les objets **File** pour la visualisation.

Voici le code de l'exemple. Notez que le résultat a été trié sans effort (par ordre alphabétique)

en utilisant la méthode **java.utils.Array.sort()** et l'**AlphabeticComparator** défini au Chapitre 9 :

```
//: c11:DirList.java
// Affiche le listing d'un répertoire.
import java.io.*;
import java.util.*;
import com.bruceeckel.util.*;

public class DirList {
public static void main(String[] args) {
File path = new File(".");
String[] list;
if(args.length == 0)
  list = path.list();
else
  list = path.list(new DirFilter(args[0]));
Arrays.sort(list,
  new AlphabeticComparator());
for(int i = 0; i < list.length; i++)
  System.out.println(list[i]);
}
}

class DirFilter implements FilenameFilter {
String afn;
DirFilter(String afn) { this.afn = afn; }
public boolean accept(File dir, String name) {
// Information du chemin de répertoire :
String f = new File(name).getName();
return f.indexOf(afn) != -1;
}
} ///:~
```

La classe **DirFilter** « implémente » l'**interface FilenameFilter**. Il est utile de voir combien est simple l'**interface FilenameFilter** :

```
public interface FilenameFilter {
boolean accept(File dir, String name);
}
```

Cela veut dire que ce type d'objet ne s'occupe que de fournir une méthode appelée **accept()**. La finalité derrière la création de cette classe est de fournir la méthode **accept()** à la méthode **list()** de telle manière que **list()** puisse « rappeler » **accept()** pour déterminer quelle noms de fichiers doivent êtres inclus dans la liste. Ainsi, cette technique fait souvent référence à *un rappel automatique* ou parfois à un [functor] (c'est à dire, **DirFilter** est un functor parce que sa seule fonction est de maintenir une méthode) ou la *Command Pattern* (une entité, ensemble de caractéristiques, de commandes). Parce que **list()** prend un objet **FilenameFilter** comme argument, cela veut dire que l'on peut passer un objet de n'importe quelle classe implémentant **FilenameFilter** afin de choisir (même lors de l'exécution) comment la méthode **list()** devra se comporter. L'objectif d'un rappel est

de fournir une flexibilité dans le comportement du code.

DirFilter montre que comme une **interface** ne peut contenir qu'un jeu de méthodes, vous n'êtes pas réduit a l'écriture seule de ces méthodes. (Vous devez au moins fournir les définitions pour toutes les méthodes dans une interface, de toutes les manières.) Dans ce cas, le constructeur de **DirFilter** est aussi créé.

La méthode **accept()** doit accepter un objet **File** représentant le répertoire où un fichier en particulier se trouve, et un **String** contenant le nom de ce fichier. Vous pouvez choisir d'utiliser ou ignorer l'un ou l'autre de ces arguments, mais vous utiliserez probablement au moins le nom du fichier. Rappelez vous que la méthode **list()** fait appel à **accept()** pour chacun des noms de fichier de l'objet répertoire pour voir lequel doit être inclus — ceci est indique par le résultat **booléen** renvoyé par **accept()**.

Pour être sûr que l'élément avec lequel vous êtes en train de travailler est seulement le nom du fichier et qu'il ne contient pas d'information de chemin, tout ce que vous avez a faire est de prendre l'objet **String** et de créer un objet **File** en dehors de celui-ci, puis d'appeler **getName()**, qui éloigne toutes les informations de chemin (dans l'optique d'une indépendance vis-à-vis de la plate-forme). Puis **accept()** utilise la méthode **indexOf()** de la classe **String** pour voir si la chaîne de caractères recherchée **afn** apparaît n'importe où dans le nom du fichier. Si **afn** est trouvé à l'intérieur de la chaîne de caractères, la valeur retournée sera l'indice de départ d'**afn**, mais si il n'est pas trouvé la valeur retourné sera - 1. Gardez en tête que ce n'est qu'une simple recherche de chaîne de caractères et qui ne possède pas d'expression « globale » de comparaison d'assortiment — comme « fo?.b?r* » — qui est beaucoup plus difficile a réaliser.

La méthode **list()** renvoie un tableau. Vous pouvez interroger ce tableau sur sa longueur et puis vous déplacer d'un bout a l'autre de celui-ci en sélectionnant des éléments du tableau. Cette aptitude de passer facilement un tableau dedans et hors d'une méthode est une amélioration immense supérieure au comportement de C et C++.

Les classes internes anonymes

Cet exemple est idéal pour une réécriture utilisant une classe interne anonyme (décrite au Chapitre 8). Tout d'abord, une méthode **filter()** est créé retournant une référence à un **FilenameFilter** :

```
//: c11:DirList2.java
// Utilisation de classes internes anonymes.
import java.io.*;
import java.util.*;
import com.bruceeckel.util.*;

public class DirList2 {
public static FilenameFilter
filter(final String afn) {
// Creation de la classe anonyme interne :
return new FilenameFilter() {
  String fn = afn;
  public boolean accept(File dir, String n) {
    // Strip path information:
```

```
    String f = new File(n).getName();
    return f.indexOf(fn) != -1;
  }
}; // Fin de la classe anonyme interne.
}
public static void main(String[] args) {
File path = new File(".");
String[] list;
if(args.length == 0)
  list = path.list();
else
  list = path.list(filter(args[0]));
Arrays.sort(list,
  new AlphabeticComparator());
for(int i = 0; i < list.length; i++)
  System.out.println(list[i]);
}
} ///:~
```

Notez que l'argument de **filter()** doit être **final**. Ceci est requis par la classe interne anonyme pour qu'elle puisse utiliser un objet hors de sa portée.

Cette conception est une amélioration puisque la classe **FilenameFilter** est maintenant fortement liée à **DirList2**. Cependant, vous pouvez reprendre cette approche et aller plus loin en définissant la classe anonyme interne comme un argument de **list()**, auquel cas c'est encore plus léger :

```
//: c11:DirList3.java
// Construction de la classe anonyme interne «sur-place ».
import java.io.*;
import java.util.*;
import com.bruceeckel.util.*;

public class DirList3 {
public static void main(final String[] args) {
File path = new File(".");
String[] list;
if(args.length == 0)
  list = path.list();
else
  list = path.list(new FilenameFilter() {
    public boolean
    accept(File dir, String n) {
      String f = new File(n).getName();
      return f.indexOf(args[0]) != -1;
    }
  });
Arrays.sort(list,
  new AlphabeticComparator());
```

```
    for(int i = 0; i < list.length; i++)
      System.out.println(list[i]);
    }
    } ///:~
```

L'argument de **main()** est maintenant **final**, puisque la classe anonyme interne utilise directement **args[0]**.

Ceci vous montre comment les classes anonymes internes permettent la création de classes rapides-et-propres pour résoudre des problèmes. Étant donné que tout en Java tourne autour des classes, cela peut être une technique de code utile. Un avantage étant que cela garde le code permettant de résoudre un problème particulier isolé dans un même lieu. D'un autre côté, cela n'est pas toujours facile à lire, donc vous devrez l'utiliser judicieusement.

Vérification et création de répertoires

La classe **File** est bien plus qu'une représentation d'un fichier ou d'un répertoire existant. Vous pouvez aussi utiliser un objet **File** pour créer un nouveau répertoire ou un chemin complet de répertoire si ils n'existent pas. Vous pouvez également regarder les caractéristiques des fichiers (taille, dernière modification, date, lecture/écriture), voir si un objet **File** représente un fichier ou un répertoire, et supprimer un fichier. Ce programme montre quelques unes des méthodes disponibles avec la classe **File** (voir la documentation HTML à *java.sun.com* pour le jeu complet) :

```
//: c11:MakeDirectories.java
// Démonstration de l'usage de la classe File pour
// creer des répertoire et manipuler des fichiers.
import java.io.*;

public class MakeDirectories {
private final static String usage = "Usage:MakeDirectories path1 ...\n" +
"Creates each path\n" +
"Usage:MakeDirectories -d path1 ...\n" +
"Deletes each path\n" +
"Usage:MakeDirectories -r path1 path2\n" +
"Renames from path1 to path2\n";
private static void usage() {
System.err.println(usage);
System.exit(1);
}
private static void fileData(File f) {
System.out.println(
  "Absolute path: " + f.getAbsolutePath() +
  "\n Can read: " + f.canRead() +
  "\n Can write: " + f.canWrite() +
  "\n getName: " + f.getName() +
  "\n getParent: " + f.getParent() +
  "\n getPath: " + f.getPath() +
  "\n length: " + f.length() +
```

```
    "\n lastModified: " + f.lastModified());
  if(f.isFile())
    System.out.println("it's a file");
  else if(f.isDirectory())
    System.out.println("it's a directory");
  }
  public static void main(String[] args) {
  if(args.length < 1) usage();
  if(args[0].equals("-r")) {
    if(args.length != 3) usage();
    File
      old = new File(args[1]),
      rname = new File(args[2]);
    old.renameTo(rname);
    fileData(old);
    fileData(rname);
    return; // Sortie de main
  }
  int count = 0;
  boolean del = false;
  if(args[0].equals("-d")) {
    count++;
    del = true;
  }
  for( ; count < args.length; count++) {
    File f = new File(args[count]);
    if(f.exists()) {
      System.out.println(f + " exists");
      if(del) {
        System.out.println("deleting..." + f);
        f.delete();
      }
    }
    else { // N'existe pas
      if(!del) {
        f.mkdirs();
        System.out.println("created " + f);
      }
    }
    fileData(f);
  }
  }
  } ///:~
```

Dans **fileData()** vous pourrez voir diverses méthodes d'investigation de fichier employées pour afficher les informations sur le fichier ou sur le chemin du répertoire.

La première méthode pratiquée par **main()** est **renameTo()**, laquelle vous permet de re-

nommer (ou déplacer) un fichier vers un nouveau chemin de répertoire signalé par l'argument, qui est un autre objet **File**. Ceci fonctionne également avec des répertoire de n'importe quelle longueur.

Si vous expérimentez le programme ci-dessus, vous découvrirez que vous pouvez créer un chemin de répertoire de n'importe quelle complexité puisque **mkdirs()** s'occupera de tout.

Entrée et sortie

Les bibliothèques d'E/S utilisent souvent l'abstraction d'un flux [stream], qui représente n'importe quelle source ou réceptacle de données comme un objet capable de produire et de recevoir des parties de données. Le flux cache les détails de ce qui arrive aux données dans le véritable dispositif d'E/S.

Les classes de la bibliothèque d'E/S Java sont divisées par entrée et sortie, comme vous pouvez le voir en regardant en ligne la hiérarchie des classes Java avec votre navigateur Web. Par héritage, toute dérivée des classes **InputStream** ou **Reader** possède des méthodes de base nommées **read()** pour lire un simple byte ou un tableau de bytes. De la même manière, toutes les dérivés des classes **OutputStream** ou **Writer** ont des méthodes basiques appelées **write()** pour écrire un seul byte ou un tableau de bytes. Cependant, de manière générale vous n'utiliserez pas ces méthodes ; elles existent afin que les autres classes puissent les utiliser — ces autres classes ayant des interfaces plus utiles. Ainsi, vous créerez rarement votre objet flux [stream] par l'emploi d'une seule classe, mais au lieu de cela en plaçant les objets ensemble sur plusieurs couches pour arriver à la fonctionnalité désirée. Le fait de créer plus d'un objet pour aboutir à un seul flux est la raison primaire qui rend la bibliothèque de flux Java confuse.

Il est utile de ranger les classes suivant leurs fonctionnalités. Pour Java 1.0, les auteurs de la bibliothèque commencèrent par décider que toutes les classes n'ayant rien à voir avec l'entrée hériteraient de l'**InputStream** et toutes les classes qui seraient associées avec la sortie seraient héritées depuis **OutputStream**.

Les types d'InputStream

Le boulot d'InputStream est de représenter les classes qui produisent l'entrée depuis différentes sources. Ces sources peuvent êtres :

Tableau 11-1. Les types d'InputStream

Classe	Fonction	Arguments du Constructeur
		Mode d'emploi
ByteArray-InputStream	Autorise un tampon en mémoire pour être utilisé comme **InputStream**	Le tampon depuis lequel extraire les bytes.
		Comme une source de données. Connectez le a un objet **FilterInputStream** pour fournir une interface pratique.
StringBuffer-InputStream	Convertit un **String** en un **InputStream**	Un **String**. L'implémentation fondamentale utilise actuellement un **StringBuffer**.
		Comme une source de données. Connectez le a un objet **FilterInputStream** pour fournir une interface pratique.
File-InputStream	Pour lire les information depuis un fichier.	Un **String** représentant le nom du fichier, ou un objet **File** ou **FileDescriptor**.
		Comme une source de données. Connectez le a un objet **FilterInputStream** pour fournir une interface pratique.
Piped-InputStream	Produit la donnée qui sera écrite vers le **PipedOutput-Stream** associé. Applique le concept de « tuyauterie ».	**PipedOutputStream**
		Comme une source de données. Connectez le a un objet **FilterInputStream** pour fournir une interface pratique.
Sequence-InputStream	Convertit deux ou plusieurs objets **InputStream** dans seul **InputStream**.	Deux objets **InputStream** ou une **Énumeration** pour un récipient d'objets **InputStream**.
		Comme une source de données. Connectez le a un objet **FilterInputStream** pour fournir une interface pratique.
Filter-InputStream	Classe abstraite qui est une interface pour les décorateurs lesquels fournissent une fonctionnalité profitable aux autres classe **InputStream**. Voir Tableau 11-3.	Voir Tableau 11-3.
		Voir Tableau 11-3.

Les types d'OutputStream

Cette catégorie contient les classes qui décident de l'endroit où iront vos données de sorties : un tableau de bytes (pas de **String**, cependant ; vraisemblablement vous pouvez en créer un en utilisant le tableau de bytes), un fichier, ou un « tuyau.»

En complément, le **FilterOutputStream** fournit une classe de base pour les classes de « décoration » qui attachent des attributs ou des interfaces utiles aux flux de sortie.

Tableau 11-2. Les types d'OutputStream

Classe	Fonction	Arguments du constructeur
		Mode d'emploi
ByteArray-OutputStream	Crée un tampon en mémoire. Toutes les données que vous envoyez vers le flux sont placées dans ce tampon.	En option la taille initiale du tampon.
		Pour désigner la destination de vos données. Connectez le à un objet **FilterOutpuStream** pour fournir une interface pratique.
File-OutputStream	Pour envoyer les informations a un fichier.	Un **String** représentant le nom d'un fichier, ou un objet **File** ou **FileDescriptor**.
		Pour désigner la destination de vos données. Connectez le à un objet **FilterOutpuStream** pour fournir une interface pratique.
Piped-OutputStream	N'importe quelle information que vous écrivez vers celui-ci se termine automatiquement comme une entrée du **PipedInput-Stream** associé. Applique le concept de « tuyauterie. »	**PipedInputStream**
		Pour indiquer la destination de vos données pour une exécution multiple [multithreading]. Connectez le à un objet **FilterOutpuStream** pour fournir une interface pratique.
Filter-OutputStream	Classe abstraite qui est une interface pour les décorateurs qui fournissent des fonctionnalités pratiques aux autres classes d'**OutputStream**. Voir Tableau 11-4.	Voir Tableau 11-4.
		Voir Tableau 11-4.

Ajouter des attributs et des interfaces utiles

L'emploi d'objets en couches pour ajouter dynamiquement et de manière claire des responsabilités aux objets individuels est mentionné comme un Pattern de *Décoration*. (Les Patterns[57] sont le sujet de *Thinking in Patterns with Java*, téléchargeable à <*www.BruceEckel.com*>.) Le Pat-

tern de décoration précise que tous les objets qui entourent votre objet initial possèdent la même interface. Ceci rend l'usage basique des décorateurs claire — vous envoyez le même message à un objet qu'il soit décoré ou non. C'est la raison de l'existence des classes « filter » dans la bibliothèque E/S de Java : la classe abstraite « filter » est la classe de base pour tous les décorateurs. (Un décorateur doit avoir la même interface que l'objet qu'il décore, mais le décorateur peut aussi étendre l'interface, ce qui se produit dans un certain nombre de classes « filter »).

Les décorateurs sont souvent employés quand un simple sous-classement touche un grand nombre de sous-classes pour satisfaire toutes les combinaisons possibles nécessaires — avec tellement de sous-classes que cela devient peu pratique. La bibliothèque d'E/S Java demande différentes combinaisons de caractéristiques, c'est pourquoi le Pattern de décoration est employé. Il y a un désavantage au Pattern de décoration, néanmoins les décorateurs vous donnent une plus grande flexibilité pendant l'écriture d'un programme (puisque vous pouvez facilement mélanger et assembler des attributs [attributes]), mais ils ajoutent de la complexité à votre code. La raison pour laquelle la bibliothèque d'E/S de Java n'est pas pratique d'emploi est que vous devez créer beaucoup de classes — le type « noyau » d'E/S plus tous les décorateurs — afin d'obtenir le simple objet E/S désiré.

Les classes qui procurent l'interface de décoration pour contrôler un **InputStream** ou **OutputStream** particulier sont **FilterInputStream** et **FilterOutputStream** — lesquelles n'ont pas des noms très intuitifs. **FilterInputStream** et **FilterOutputStream** sont des classes abstraites qui sont dérivées depuis les classes de base de la bibliothèque d'E/S, **InputStream** et **OutputStream**, ceci étant l'exigence clef du décorateur (afin qu'il procure une interface commune à tous les objets qui seront décorés).

Lire depuis un InputStream avec FilterInputStream

La classe **FilterInputStream** accomplit deux choses significatives différentes. **DataInputStream** vous permet de lire différents types de données primitives tout aussi bien que des objets **String**. (Toutes les méthodes commencent avec « read, » comme **readByte()**, **readFloat()**, etc.) Ceci, accompagné par **DataOutputStream**, vous permet de déplacer des données primitives d'une place à une autre en passant par un flux. Ces « places » sont déterminées par les classes du Tableau 11-1.

Les classes restantes modifient le comportement interne d'un **InputStream** : s'il est mis en tampon ou pas, si il garde trace des lignes qu'il lit (vous permettant de demander des numéros de ligne ou de régler le numéro de ligne), et si vous pouvez pousser en arrière un caractère seul. Les deux dernières classes ressemblent beaucoup à une ressource pour construire un compilateur (c'est à dire, elles ont étés ajoutées en support pour la construction du compilateur Java), donc vous ne l'utiliserez probablement pas en programmation habituelle.

Vous devrez probablement presque tout le temps mettre en tampon votre entrée, sans prendre en compte l'élément d'E/S auquel vous vous connectez, ainsi il aurait été plus censé pour la bibliothèque d'E/S de faire un cas spécial (ou un simple appel de méthode)pour l'entrée non mise en tampon plutôt que pour l'entrée mise en tampon.

classe	Fonction	Arguments du constructeur
		Mode d'emploi
Data-InputStream	Employé de concert avec **DataOutputStream**, afin de lire des primitives (**int**, **char**, **long**, etc.) depuis un flux de manière portable.	**InputStream**
		Contient une interface complète vous permettant de lire les types de primitives.
Buffered-InputStream	Utilisez ceci pour empêcher une lecture physique chaque fois que vous désirez plus de données. Cela dit « Utiliser un tampon. »	**InputStream**, avec en option la taille du tampon.
		Ceci ne fournit pas une interface en soi, mais une condition permettant d'employer le tampon.
LineNumber-InputStream	Garde trace des numéros de ligne dans le flux d'entrée; vous pouvez appeler **getLineNumber()** et **setLineNumber(int)**.	**InputStream**
		Cela n'ajoute que la numérotation des lignes, de cette façon on attachera certainement un objet interface.
Pushback-InputStream	Possède un tampon retour-chariot d'un byte permettant de pousser le dernier caractère lu en arrière.	**InputStream**
		Généralement employé dans le scanner pour un compilateur et probablement inclus parce qu'il était nécessaire au compilateur Java. Vous ne l'utiliserez probablement pas.

Écrire vers un OutputStream avec FilterOutputStream

Le complément à **DataInputStream** est **DataOutputStream**, lequel formate chacun des types de primitive et objets **String** vers un flux de telle sorte que n'importe quel **DataInputStream**, sur n'importe quelle machine, puisse le lire. Toutes les méthodes commencent par « write », comme **writeByte()**, **writeFloat()**, etc.

À l'origine, l'objectif de **PrintStream** est d'imprimer tous les types de données primitive et objets **String** dans un format perceptible. Ce qui est différent de **DataOutputStream**, dont le but est de placer les éléments de données dans un flux de manière que **DataInputStream** puisse de façon portable les reconstruire.

Les deux méthodes importantes dans un **PrintStream** sont **print()** et **println()**, qui sont surchargées [overloaded] pour imprimer tous les types différents. La différence entre **print()** et **println()** est que le dernier ajoute une nouvelle ligne une fois exécuté.

PrintStream peut être problématique car il piège toutes les **IOException**s (vous devrez tester explicitement le statut de l'erreur avec **checkError()**, lequel retourne **true** si une erreur c'est produite. Aussi, **PrintStream** n'effectue pas l'internationalisation proprement et ne traite pas les sauts de ligne de manière indépendante de la plate-forme (ces problèmes sont résolus avec **Print-**

Writer).

BufferedOutputStream est un modificateur, il dit au flux d'employer le tampon afin de ne pas avoir une écriture physique chaque fois que l'on écrit vers le flux. On employera probablement souvent ceci avec les fichiers, et peut être la console E/S.

Classe	Fonction	Arguments du Constructeur
		Mode d'emploi
Data-OutputStream	Utilisé en concert avec **DataInputStream** afin d'écrire des primitives (int, char, long, etc.) vers un flux de manière portable.	**OutputStream**
		Contient une interface complète vous permettant d'écrire les types de primitives.
PrintStream	Pour produire une sortie formatée. Pendant que **DataOutputStream** manie le *stockage* de données, le **PrintStream** manie l'*affichage*.	**OutputStream**, avec une option **boolean** indiquant que le tampon est vidé avec chaque nouvelle ligne.
		Doit être l'emballage « final » pour votre objet **OutputStream**. Vous l'utiliserez probablement beaucoup.
Buffered-OutputStream	Utilisez ceci en prévention d'une écriture physique à chaque fois que vous envoyez un morceau de donnée. En disant « Utilise un tampon. » Vous pouvez appeler **flush()** pour vider le tampon.	**OutputStream**, avec en option la taille du tampon.
		Ceci ne fournit pas une interface *en soi*, juste la nécessité de l'emploi du tampon soit utilisé. Attache un objet interface.

Lecteurs & écrivains [*Loaders & Writers*]

Java 1.1 apporte quelques modifications significatives à la bibliothèque fondamentale de flux d'E/S (Java 2, cependant, n'apporte pas de modifications fondamentales). Quand vous voyez les classes **Reader** et **Writer** votre première pensée (comme la mienne) doit être que celles-ci ont pour intention de remplacer les classes **InputStream** et **OutputStream**. Mais ce n'est pas le cas. Quoique certains aspects de la bibliothèque originale de flux sont dépréciés (si vous les employez vous recevrez un avertissement de la part du compilateur), les classes **InputStream** et **OutputStream** fournissent pourtant de précieuses fonctions dans le sens d'un E/S orienté **byte**, tandis que les classes **Reader** et **Writer** fournissent une E/S à base de caractères se pliant à l'Unicode. En plus :

1. Java 1.1 a ajouté de nouvelles classes dans la hiérarchie d'**InputStream** et **OutputStream**, donc il est évident qu'elles ne sont pas remplacées.
2. Il y a des fois ou vous devrez employer les classes de la hiérarchie « byte » *en combi-*

naison avec les classes de la hiérarchie « caractère ». Pour cela il y à des classes « passerelles » : **InputStreamReader** convertit un **InputStream** en un **Reader** et **OutputStreamWriter** convertit un **OutputStream** en un **Writer**.

La raison la plus importante des hiérarchies de **Reader** et **Writer** est l'internationalisation. L'ancienne hiérarchie de flux d'E/S ne supporte que des flux de bytes de 8-bit et ne traite pas bien les caractères Unicode de 16-bit. Depuis qu'Unicode est employé pour l'internationalisation (et les **char** natifs de Java sont en Unicode 16-bit), les hiérarchies de **Reader** et **Writer** ont été ajoutées pour supporter l'Unicode dans toutes les opérations d'E/S. En plus, les nouvelles bibliothèques sont conçues pour des opérations plus rapides que l'ancienne.

Comme il est de coutume dans ce livre, j'aurais aimé fournir une synthèse des classes, mais j'ai supposé que vous utiliserez la documentation en ligne pour éclaircir les détails, comme pour la liste exhaustive des méthodes.

Les sources et les réceptacles de données

Presque toutes les classes originales de flux d'E/S Java possèdent des classes **Reader** et **Writer** correspondantes afin de fournir une manipulation native en Unicode. Cependant, il y a certains endroits où les **InputStream**s et les **OutputStream**s orientés-**byte** sont la solution adoptée ; en particulier, les bibliothèques **java.util.zip** sont orientées-**byte** plutôt qu'orientée-**char.** Donc l'approche la plus sage est d'essayer d'utiliser les classes **Reader** et **Writer** chaque fois que c'est possible, et vous découvrirez des situations où il vous faudra employer les bibliothèques orientées-**byte** parce que votre code ne se compilera pas.

Sources & Récipients: Classe Java 1.0	**Classe Java 1.1 correspondante**
InputStream	**Reader** convertisseur : **InputStreamReader**
OutputStream	**Writer** convertisseur : **OutputStreamWriter**
FileInputStream	**FileReader**
FileOutputStream	**FileWriter**
StringBufferInputStream	**StringReader**
(pas de classe correspondante)	**StringWriter**
ByteArrayInputStream	**CharArrayReader**
ByteArrayOutputStream	**CharArrayWriter**
PipedInputStream	**PipedReader**
PipedOutputStream	**PipedWriter**

En général, vous constaterez que les interfaces pour les deux différentes hiérarchies sont semblables sinon identiques.

Modifier le comportement du flux

Pour les **InputStream**s et **OutputStream**s, les flux sont adaptés à des usages particuliers en utilisant des sous-classes « décoratives » de **FilterInputStream** et **FilterOutputStream**. La hié-

rarchie de classe **Reader** et **Writer** poursuit l'usage de ce concept — mais pas exactement.

Dans le tableau suivant, la correspondance est une approximation plus grossière que dans la table précédente. La différence est engendrée par l'organisation de la classe : Quand **BufferedOutputStream** est une sous-classe de **FilterOutputStream, BufferedWriter** n'est *pas* une sous-classe de **FilterWriter** (laquelle, bien qu'elle soit **abstract**, n'a pas de sous-classe et donc semble avoir été mise dedans de manière à réserver la place ou simplement de manière à ce que vous ne sachiez pas où elle se trouve). Cependant, les interfaces pour les classes est plutôt un combat terminé.

Filtres : classe Java 1.0	Classe correspondante en Java 1.1
FilterInputStream	**FilterReader**
FilterOutputStream	**FilterWriter** (classe abstract sans sous-classe)
BufferedInputStream	**BufferedReader** (a aussi **readLine()**)
BufferedOutputStream	**BufferedWriter**
DataInputStream	Utilise **DataInputStream** (sauf quand vous voulez utiliser **readLine()**, alors vous devez utiliser un **BufferedReader**)
PrintStream	**PrintWriter**
LineNumberInputStream	**LineNumberReader**
StreamTokenizer	**StreamTokenizer** (utilise un constructeur qui prend un **Reader** à la place)
PushBackInputStream	**PushBackReader**

Il y a un sens qui est tout à fait clair : Chaque fois que vous voulez utiliser **readLine()**, vous ne devrez plus le faire avec un **DataInputStream** (ceci recevant un message de dépréciation au moment de la compilation), mais utiliser à la place un **BufferedReader**. À part cela, **DataInputStream** est pourtant l'élément « préféré » de la bibliothèque d'E/S.

Pour faire la transition vers l'emploi facile d'un **PrintWriter**, il possède des constructeurs qui prennent n'importe quel objet **OutputStream**, aussi bien que des objets **Writer**. Cependant, **PrintWriter** n'a pas plus de support pour formater comme le faisait **PrintStream** ; les interfaces sont de fait les mêmes.

Le constructeur de **PrintWriter** possède également une option pour effectuer le vidage automatique de la mémoire [automatic flushing], lequel se produit après chaque **println()** si le drapeau du constructeur est levé dans ce sens.

Les classes inchangées

Certaines classes ont été laissées inchangées entre Java 1.0 et Java 1.1 :

Les classes de Java 1.0 qui n'ont pas de classes correspondantes en Java 1.1
DataOutputStream
File

RandomAccessFile
SequenceInputStream

DataOutputStream, en particulier, est utilisé sans modification, donc pour stocker et retrouver des données dans un format transportable vous utiliserez les hiérarchies **InputStream** et **OutputStream**.

Et bien sûr : L'accès aléatoire aux fichiers (RandomAccessFile)

RandomAccessFile est employé pour les fichiers dont la taille de l'enregistrement est connue, de sorte que vous pouvez bouger d'un enregistrement à un autre en utilisant **seek()**, puis lire ou changer les enregistrements. Les enregistrements n'ont pas forcément la même taille ; vous devez seulement être capable de déterminer de quelle grandeur ils sont et où ils sont placés dans le fichier.

D'abord il est un peu difficile de croire que **RandomAccessFile** ne fait pas partie de la hiérarchie d'**InputStream** ou d'**OutputStream**. Cependant, il n'y a pas d'association avec ces hiérarchies autre que quand il arrive de mettre en œuvre les interfaces **DataInput** et **DataOutput** (qui sont également mises en œuvre par **DataInputStream** et **DataOutputStream**). Elle n'utilise même pas la fonctionnalité des classes existantes **InputStream** et **OutpuStream** — il s'agit d'une classe complétement différente, écrite en partant de zéro, avec toutes ses propres méthodes (pour la plupart native). Une raison à cela pouvant être que **RandomAccessFile** a des comportements essentiellement différents des autres types d'E/S, dès qu'il est possible de se déplacer en avant et en arrière dans un fichier. De toute façon, elle reste seule, comme un descendant direct d'**Object**.

Essentiellement, un **RandomAccessFile** fonctionne comme un **DataInputStream** collé ensemble avec un **DataOutputStream**, avec les méthodes **getFilePointer()** pour trouver où on se trouve dans le fichier, **seek()** pour se déplacer vers un nouvel emplacement dans le fichier, et **length()** pour déterminer la taille maximum du fichier. En complément, les constructeurs requièrent un deuxième argument (identique à **fopen()** en C) indiquant si vous effectuez de manière aléatoire une lecture (« **r** ») ou une lecture et écriture (« **rw** »). Il n'y a pas de ressource pour les fichiers en lecture seule, ce qui pourrait suggérer que **RandomAccessFile** aurait mieux fonctionné s'il se trouvait hérité de **DataInputStream**.

Les méthodes de recherche sont valables seulement dans **RandomAccessFile**, qui fonctionne seulement avec des fichiers. Le **BufferedInputStream** permet de marquer « **mark()** » une position (dont la valeur est tenue dans une seule variable interne) et d'annuler cette position « **reset()** », mais c'est limité et pas très pratique.

L'usage typique des flux d'E/S

Bien que pouvez combiner les classes de flux d'E/S de différentes manières, vous utiliserez probablement quelques combinaisons. L'exemple suivant pourra être employé comme une référence de base ; il montre la création et l'utilisation de configurations d'E/S typiques. Notez que chaque configuration commence par un commentaire avec numéro et titre qui correspondent aux titres des paragraphes suivant et fournissant l'explication approprié.

```
//: c11:IOStreamDemo.java
// Configurations typiques de flux d'E/S.
import java.io.*;
```

```
public class IOStreamDemo {
  // Lance les exeptions vers la console :
  public static void main(String[] args)
  throws IOException {
    // 1. Lecture d'entrée par lignes :
    BufferedReader in =      new BufferedReader(
       new FileReader("IOStreamDemo.java"));
    String s, s2 = new String();
    while((s = in.readLine())!= null)
      s2 += s + "\n";
    in.close();

    // 1b. Lecture d'entrée standard :
    BufferedReader stdin =      new BufferedReader(
       new InputStreamReader(System.in));
    System.out.print("Enter a line:");
    System.out.println(stdin.readLine());

    // 2. Entrée depuis la mémoire
    StringReader in2 = new StringReader(s2);
    int c;
    while((c = in2.read()) != -1)
      System.out.print((char)c);

    // 3. Entrée de mémoire formattée
    try {
      DataInputStream in3 =       new DataInputStream(
         new ByteArrayInputStream(s2.getBytes()));
      while(true)
        System.out.print((char)in3.readByte());
    } catch(EOFException e) {
      System.err.println("End of stream");
    }

    // 4. Sortie de fichier
    try {
      BufferedReader in4 =       new BufferedReader(
         new StringReader(s2));
      PrintWriter out1 =       new PrintWriter(
         new BufferedWriter(
           new FileWriter("IODemo.out")));
      int lineCount = 1;
      while((s = in4.readLine()) != null )
        out1.println(lineCount++ + ": " + s);
      out1.close();
    } catch(EOFException e) {
```

```
      System.err.println("End of stream");
    }

    // 5. Stockage et recupération de donnée
    try {
      DataOutputStream out2 =       new DataOutputStream(
        new BufferedOutputStream(
          new FileOutputStream("Data.txt")));
      out2.writeDouble(3.14159);
      out2.writeChars("That was pi\n");
      out2.writeBytes("That was pi\n");
      out2.close();
      DataInputStream in5 =       new DataInputStream(
        new BufferedInputStream(
          new FileInputStream("Data.txt")));
      BufferedReader in5br =       new BufferedReader(
        new InputStreamReader(in5));
      // Doit utiliser DataInputStream pour des données :
      System.out.println(in5.readDouble());
      // Peut maintenant employer le readLine():
      System.out.println(in5br.readLine());
      // Mais la ligne ressort bizzarement.
      // Celle crée avec writeBytes est OK:
      System.out.println(in5br.readLine());
    } catch(EOFException e) {
      System.err.println("End of stream");
    }

    // 6. Lecture/écriture par accès aléatoire aux fichiers [Reading/writing random access
files]
    RandomAccessFile rf =      new RandomAccessFile("rtest.dat", "rw");
    for(int i = 0; i < 10; i++)
      rf.writeDouble(i*1.414);
    rf.close();

    rf =      new RandomAccessFile("rtest.dat", "rw");
    rf.seek(5*8);
    rf.writeDouble(47.0001);
    rf.close();

    rf =      new RandomAccessFile("rtest.dat", "r");
    for(int i = 0; i < 10; i++)
      System.out.println(
        "Value " + i + ": " +
        rf.readDouble());
    rf.close();
  }
```

```
} ///:~
```

Voici les descriptions pour les sections numérotées du programme :

Flux d'Entrée

La partie 1 à 4 démontre la création et l'utilisation des flux d'entrée. La partie 4 montre aussi l'emploi simple d'un flux de sortie.

1. Entrée en tampon du fichier [Buffered input file]

Afin d'ouvrir un fichier pour l'entrée de caractères, on utilise un **FileInputReader** avec un objet **String** ou **File** comme nom de fichier. Pour la vitesse, on désirera que le fichier soit mis en mémoire tampon alors on passera la référence résultante au constructeur à un **BufferedReader**. Puisque **BufferedReader** fournit aussi la méthode **readLine( )**, qui est notre objet final et l'interface depuis laquelle on lit. Quand on cherche la fin du fichier, **readLine( )** renverra **null** qui sera utilisé pour sortir de la boucle **while**.

Le **String s2** est utilisé pour accumuler le contenu entier du fichier (incluant les nouvelles lignes qui doivent être ajoutées puisque **readLine( )** les enlève). **s2** est ensuite employé dans la dernière partie de se programme. Enfin, **close( )** est appelé pour fermer le fichier. Techniquement, **close( )** sera appelé au lancement de **finalize()**, et ceci est supposé se produire (que le garbage collector se mette en route ou pas) lors de la fermeture du programme. Cependant, ceci a été inconséquemment implémenté, c'est pourquoi la seule approche sûre est d'appeler explicitement **close( )** pour les fichiers.

La section 1b montre comment envelopper **System.in** afin de lire l'entrée sur la console. **System.in** est un **DataInputStream** et **BufferedReader** nécessite un argument **Reader**, voilà pourquoi **InputStreamReader** est introduit pour effectuer la traduction.

2. Entrée depuis la mémoire

Cette partie prend le **String s2** qui contient maintenant le contenu entier du fichier et l'utilise pour créer un **StringReader**. Puis **read( )** est utilisé pour lire chaque caractère un par un et les envoie vers la console. Notez que **read( )** renvoie le byte suivant sous la forme d'un **int** et pour cette raison il doit être convertit en **char** afin de s'afficher correctement.

3. Entrée de mémoire formatée

Pour lire une donnée « formatée », vous utiliserez un **DataInputStream**, qui est une classe d'E/S orientée-**byte** (plutôt qu'orientée-**char**). Ainsi vous devrez utiliser toutes les classes **InputStream** plutôt que les classes **Reader**. Bien sur, vous pouvez lire n'importe quoi (du genre d'un fichier) comme des bytes en utilisant les classes **InputStream**, mais ici c'est un **String** qui est utilisé. Pour convertir le **String** en un tableau de bytes, ce qui est approprié pour un **ByteArrayInputStream**, **String** possède une méthode **getBytes()**pour faire le travail. A ce stade, vous avez un **InputStream** adéquat pour porter un **DataInputStream**.

Si on lit les caractères depuis un DataInputStream un byte à chaque fois en utilisant readByte( ), n'importe quelle valeur de byte donne un résultat juste donc la valeur de retour ne peut pas être employée pour détecter la fin de l'entrée. À la place, on peut employer la méthode avalaible( )

pour découvrir combien de caractères sont encore disponibles. Voici un exemple qui montre comment lire un fichier byte par byte :

```
//: c11:TestEOF.java
// Test de fin de fichier
// En lisant un byte a la fois.
import java.io.*;

public class TestEOF {
  // Lance les exeptions vers la console :
  public static void main(String[] args)
  throws IOException {
    DataInputStream in =
      new DataInputStream(
       new BufferedInputStream(
        new FileInputStream("TestEof.java")));
    while(in.available() != 0)
      System.out.print((char)in.readByte());
  }
} ///:~
```

Notons qu'**available()** fonctionne différemment en fonction du type de ressource depuis laquelle on lit; c'est littéralement « le nombre de bytes qui peuvent être lus *sans blocage*. » Avec un fichier cela signifie le fichier entier, mais avec une autre sorte d'entrée cela ne pourra pas être possible, alors employez le judicieusement.

On peut aussi détecter la fin de l'entrée dans des cas comme cela en attrapant une exception. Cependant, l'emploi des exeptions pour le contrôle du flux est considéré comme un mauvais emploi de cette caractéristique.

Cet exemple aussi montre comment écrire des données vers un fichier. Premièrement, un **FileWriter** est crée pour se connecter au fichier. Vous voudrez toujours mettre en tampon la sortie en l'emballant[wrapping it] dans un **BufferedWriter** (essayez de retirer cet emballage pour voir l'impact sur les performances — le tampon tend à accroître dramatiquement les performance des opérations d'E/O). Puis le formatage est changé en un PrintWriter. Le fichier de données ainsi crée est lisible comme un fichier texte normal.

Comme les lignes sont écrites vers le fichier, les numéros de lignes sont ajoutés. Notez que **LineNumberInputStream** n'est *pas* utilisé, parce que c'est une classe idiote et que vous n'en avez pas besoin. Comme il est montré ici, il est superficiel de garder trace de vos propres numéros de lignes.

Quand le flux d'entrée épuisé, **readLine( )** renvoie null. Vous verrez **close( )** explicite pour **out1**, car si vous ne faites pas appel à **close( )** pour tous vos fichiers de sortie, vous pourrez découvrir que les tampons ne seront pas libérés sans qu'ils seront incomplets.

Flux de sortie

Les deux types de flux de sortie sont séparés par la manière dont ils écrivent les données : un les écrit pour une consommation humaine, l'autre les écrit pour une reacquisition par un **DataInputStream**. Le **RandomAccessFile** se tient seul, bien que son format de données soit compatible

avec **DataInputStream** et **DataOutputStream**.

5. Stocker et récupérer des données

Un **PrintWriter** formate les données afin qu'elles soient lisibles par un humain. Cependant, pour sortir des données qui puissent être récupérées par un autre flux, on utilise un **DataOutputStream** pour écrire les données et un **DataInputStream** pour récupère les données. Bien sûr, ces flux pourraient être n'importe quoi, mais ici c'est un fichier qui est employé, mis en mémoire tampon pour à la fois lire et écrire. **DataOutputStream** et **DataInputStream** sont orientés-byte et nécessitent ainsi des **InputStream**s and **OutputStream**s.

Si vous employez un **DataOutputStream** pour écrire les données, alors Java se porte garant de l'exacte récupération des données en employant un **DataInputStream** — sans se soucier du type de plate-forme qui écrit et lit les données. Ce qui est incroyablement valable, comme chacun sait ayant passé du temps a s'inquiéter de la distribution de donnée à des plates-formes spécifiques. Ce problème disparaît si l'on a Java sur les deux plates-formes [58].

Notez que les caractères de la chaîne de caractère sont écrit en utilisant à la fois **writeChars()** et **writeBytes()**. Quand vous exécuterez le programme, vous découvrirez que **writeChars()** donne en sortie des caractères Unicode 16-bit. Lorsque l'on lit la ligne avec **readLine()**, vous remarquerez qu'il y a un espace entre chaque caractère, à cause du byte (ndt : octet ?) supplémentaire inséré par Unicode. Comme il n'y a pas de méthode complémentaire « readChars » dans **DataInputStream**, vous êtes coincés à retirer ces caractères un par un avec **readChar()**. Ainsi pour l'ASCII, il est plus facile d'écrire les caractères sous la forme de bytes suivit par un saut de ligne; employez alors **readLine()** pour relire les bytes comme des lignes régulières ASCII.

Le **writeDouble()** stocke les nombres **double** pour le flux et le **readDouble()** complémentaire les récupère (il y a des méthodes similaires pour lire et écrire les autres types). Mais pour que n'importe quelle méthode de lecture fonctionne correctement, vous devrez connaître l'emplacement exact des éléments de donnée dans le flux, puisqu'il serait possible de lire les **double** stockés comme de simple séquences de bytes, ou comme des **chars**, etc. Donc vous devrez soit avoir un format fixé pour les données dans le fichier ou des informations supplémentaires devront êtres stockés dans le fichier et que vous analyserez pour déterminer l'endroit où les données sont stockées.

6. Accés aléatoire en lecture et écriture aux fichiers

Comme il a été noté précédemment, le **RandomAccessFile** est presque totalement isolé du reste de la hiérarchie d'E/S, protègé par le fait qu'il implémente les interfaces **DataInput** et **DataOutput**. Donc vous ne pouvez l'associer avec un des point des sous-classes **InputStream** et **OutputStream.** Quoiqu'il pourrait sembler raisonnable de traiter un **ByteArrayInputStream** comme un élément d'accès aléatoire, vous pouvez employer un **RandomAccessFile** pour ouvrir simplement un fichier. Vous devez supposer qu'un **RandomAccessFile** est correctement mis en mémoire tampon puisque vous ne pouvez pas ajouter cela.

La seule option disponible est dans le second argument du constructeur : vous pouvez ouvrir un **RandomAccessFile** pour lire (« **r** ») ou lire et écrire (« **rw** »).

Utiliser un **RandomAccessFile** est comme utiliser une combinaison de **DataInputStream** et **DataOutputStream** (parce que cela implémente les interfaces équivalentes). En plus, vous pouvez remarquer que **seek()** est utilisé pour errer dans le fichier et changer une des valeurs.

Un bogue ?

Si vous regardez la partie 5, vous verrez que les données sont écrites *avant* le texte. C'est à cause d'un problème qui a été introduit dans Java 1.1 (et persiste dans Java 2) qui apparaît vraiment comme un bogue pour moi, mais j'en ai rendu compte et les debogueurs de JavaSoft ont dit que c'était la manière dont il était supposé fonctionner (pourtant, le problème n'apparaissait pas dans Java 1.0, ce qui me rend suspicieux). Le problème est montré dans le code suivant :

```
//: c11:IOProblem.java
// Problème dans Java 1.1 et supérieur.
import java.io.*;

public class IOProblem {
  // Lance les exeptions vers la console :
  public static void main(String[] args)
  throws IOException {
    DataOutputStream out =      new DataOutputStream(
       new BufferedOutputStream(
         new FileOutputStream("Data.txt")));
    out.writeDouble(3.14159);
    out.writeBytes("C'était la valeur de pi\n");
    out.writeBytes("C'est pi/2:\n");
    out.writeDouble(3.14159/2);
    out.close();

    DataInputStream in =      new DataInputStream(
       new BufferedInputStream(
         new FileInputStream("Data.txt")));
    BufferedReader inbr =      new BufferedReader(
       new InputStreamReader(in));
    // Les doubles écrit AVANT la ligne de texte
    // sont renvoyés correctement :
    System.out.println(in.readDouble());
    // Lit le lignes du texte :
    System.out.println(inbr.readLine());
    System.out.println(inbr.readLine());
    // Tenter de lire les doubles après la ligne
    // produit une exeption de fin de ligne :
    System.out.println(in.readDouble());
  }
} ///:~
```

Il apparaît que tout ce que vous écrivez après un appel à **writeBytes()** n'est pas récupérable. La réponse est apparament la même que la réponse à la vieille blague de vaudeville : « Docteur, cela fait mal quand je fais cela ! » « Ne fait pas cela ! ».

Flux Piped

Les **PipedInputStream**, **PipedOutputStream**, **PipedReader** et **PipedWriter** sont mention-

nés de manière brève dans ce chapitre. Ce qui n'insinue pas qu'il ne sont pas utiles, mais leur importance n'est pas évidente jusqu'à ce que vous ayez commencé a comprendre le multithreading, étant donné quel les flux piped sont employés pour communiquer entre les threads. Ceci est abordé avec un exemple au chapitre 14.

Standard E/S

Le terme d'*E/S standard* se réfere au concept d'Unix (qui est reproduit sous une certaine forme dans Windows et bien d'autres systèmes d'exploitations) d'un simple flux d'information qui est utilisé par un programme. Toutes les entrées du programme peuvent provenir d'une *entrée standard*, toutes ses sorties peuvent aller vers une *sortie standard*, et tous les messages d'erreur peuvent être envoyés à une *erreur standard*. L'importance de l'E/S standard est que le programme peut être facilement mis en chaîne simultanément et la sortie standard d'un programme peut devenir l'entrée standard pour un autre programme. C'est un outil puissant.

Lire depuis une entrée standard

Suivant le modèle d'E/S standard, Java possède **System.in**, **System.out,** et **System.err**. Tout au long de ce livre vous avez vu comment écrire vers une sortie standard en utilisant **System.out**, qui est déjà pré-envellopé comme un objet **PrintStream. System.err** est semblable a **PrintStream**, mais **System.in** est un **InputStream** brut, sans emballage. Ceci signifie que bien que vous pouvez utiliser **System.out** et **System.err** immédiatement, **System.in** doit être envellopé avant de pouvoir lire depuis celui-ci.

Typiquement, vous désirerez lire l'entrée une ligne à la fois en utilisant **readLine()**, donc vous devrez envelloper **System.in** dans un **BufferedReader.** Pour cela, vous devrez convertir **System.in** en un **Reader** par l'usage d'**InputStreamReader**. Voici un exemple qui fait simplement écho de chaque ligne que vous tapez :

```
//: c11:Echo.java
// Comment lire depuis l'entrée standard.
import java.io.*;

public class Echo {
  public static void main(String[] args)
  throws IOException {
    BufferedReader in =       new BufferedReader(
         new InputStreamReader(System.in));
    String s;
    while((s = in.readLine()).length() != 0)
      System.out.println(s);
    // Une ligne vide met fin au programme.
  }
} ///:~
```

Le sens de l'instruction d'exception est que **readLine()** peut lancer une **IOException**. Notez que pourra généralement être mit en tampon, comme avec la majorité des fluxs

Modifier System.out en un PrintWriter

System.out est un **PrintStream**, qui est un **OutputStream**. **PrintWriter** a un constructeur qui prend un **OutputStream** comme argument. Ainsi, si vous le désirez vous pouvez convertir **System.out** en un **PrintWriter** en utilisant ce constructeur :

Il est important d'utiliser la version à deux arguments du constructeur **PrintWriter** et de fixer le deuxième argument a **true** afin de permettre un vidage automatique, sinon vous ne verriez pas la sortie.

Réorienter l'E/S standard

La classe Java **System** vous permet de rediriger l'entrée, la sortie, et l'erreur standard des flux d'E/S en employant un simple appel aux méthodes statiques :

setIn(InputStream) setOut(PrintStream) setErr(PrintStream)

Réorienter la sortie est particulierement utile si vous commencez soudainement a créer une grande quantité de sortie sur l'écran et qu'il défile jusqu'à la fin plus vite que vous ne pouvez le lire. [59] Réorienter l'entrée est précieux pour un programme en ligne de commande dans lequel vous désirez tester un ordre d' entrée-utilisateur particulièr à plusieurs reprises. Voici un exemple simple qui montre l'utilisation de ces méthodes :

```
//: c11:Redirecting.java
// Demonstration de reorientation d'E/S standard.
import java.io.*;

class Redirecting {
  // Lance les exeptions vers la console :
  public static void main(String[] args)
  throws IOException {
    BufferedInputStream in =
      new BufferedInputStream(
        new FileInputStream(
          "Redirecting.java"));
    PrintStream out =      new PrintStream(
        new BufferedOutputStream(
          new FileOutputStream("test.out")));
    System.setIn(in);
    System.setOut(out);
    System.setErr(out);

    BufferedReader br =
      new BufferedReader(
        new InputStreamReader(System.in));
    String s;
    while((s = br.readLine()) != null)
      System.out.println(s);
    out.close(); // Rappelez-vous de ça !
  }
```

```
} ///:~
```

Ce programme attache la sortie standard à un ficher, et redirige la sortie standard et l'erreur standard vers un autre fichier.

La redirection d'E/S manipule les fluxs de bytes, mais pas les fluxs de caractères, ainsi **InputStream**s et **OutputStream**s sont plus utilisés que les **Reader**s et **Writer**s.

Compression

La librairie (ndt : ou bibliothèque) d'E/S Java contient des classes pour supporter la lecture et l'écriture de flux dans des formats compressés. Ceux-ci sont envellopés autour des classes existantes d'E/S pour fournir des fonctionnalités de compression.

Ces classes ne sont pas dérivée des classes **Reader** et **Writer**, mais à la place font partie des hiérarchies d'**InputStream** et **OutputStream**. Ceci parcque la libraire de compression fonctionne avec des bytes, pas des caractères. Cependant, vous serez parfois forcés de mixer les deux types de fluxs. (Rappellez-vous que vous pouvez utiliser **InputStreamReader** et **OutputStreamWriter** pour fournirune conversion facile entre un type et un autre.)

Classe de compression	Fonction
CheckedInputStream	**GetCheckSum()** fait une checksum (vérification du nombre de bits transmis afin de deceler des erreurs de transmition) pour n'importe quel **InputStream** (non pas une simple décompression).
CheckedInputStream	**GetCheckSum()** fait une checksum pour n'importe quel **OutputStream** (non pas une simple compression).
DeflaterOutputStream	Classe de base pour les classes de compression.
ZipOutputStream	Un **DeflaterOutputStream** qui compresse les données au format Zip.
GZIPOutputStream	Un **DeflaterOutputStream** qui compresse les données au format GZIP.
InflaterInputStream	Classe de base pour les classes de décompression.
ZipInputStream	Un **InflaterInputStream** qui décompresse les données qui sont stockées au format Zip.
GZIPInputStream	Un **InflaterInputStream** qui décompresse les données qui sont stockées au format GZIP.

Bien qu'il y ait de nombreux algorithmes de compression, Zip et GZIP sont peut-être ceux employés le plus courament. Ainsi vous pouvez facilement manipuler vos données compressées avec les nobreux outils disponibles pour écrire et lire ces formats.

Compression simple avec GZIP

L'interface GZIP est simple et est ainsi la plus appropriée quand vous avez un simple flux de

donnée que vous désirez compresser (plutôt qu'un récipient (container) de pièces différentes de données. Voici un exemple qui compresse un simple fichier :

```
//: c11:GZIPcompress.java
// Utilise la compression GZIP pour compresser un fichier
// dont le nom est passé en ligne de commande.
import java.io.*;
import java.util.zip.*;

public class GZIPcompress {
  // Lance les exceptions vers la console :
  public static void main(String[] args)
  throws IOException {
    BufferedReader in =      new BufferedReader(
       new FileReader(args[0]));
    BufferedOutputStream out =      new BufferedOutputStream(
       new GZIPOutputStream(
         new FileOutputStream("test.gz")));
    System.out.println("Writing file");
    int c;
    while((c = in.read()) != -1)
      out.write(c);
    in.close();
    out.close();
    System.out.println("Reading file");
    BufferedReader in2 =      new BufferedReader(
       new InputStreamReader(
         new GZIPInputStream(
           new FileInputStream("test.gz"))));
    String s;
    while((s = in2.readLine()) != null)
      System.out.println(s);
  }
} ///:~
```

L'emploi des classes de compression est simple — vous envellopez simplement votre flux de sortie dans un **GZIPOutputStream** ou un **ZipOutputStream** et votre flux d'entrée dans un **GZIPInputStream** ou un **ZipInputStream**. Tout le reste étant de l'écriture normale d'entrée et de sortie. C'est un exemple de mélange de fluxs orientés-**char** avec des fluxs orientés-**byte** : **in** utilise la classe **Reader**, vu que le constructeur de **GZIPOutputStream** peut seulement accepter un objet **OutputStream**, et non pas un objet **Writer**. Quand le fichier est ouvert, le **GZIPInputStream** est convertit en un **Reader**.

La librairie qui supporte le format Zip est bien plus vaste. Avec elle vous pouvez facilement stocker des fichiers multiples, et il y a même une classe séparée pour amener le procédé de lecture d'un fichier Zip simple. La librairie utilise le format Zip standard de manière à ce qu'il fonctionne avec tous les outils couramment téléchargeables sur l'Internet. L'exemple suivant prend la même forme que l'exemple précédent, mais il manipule autant d'arguments de ligne de commande que vous le désirez. En plus, il met en valeur l'emploi de la classe **Checksum** pour calculer et vérifier la

somme de contrôle [checksum] pour le fichier. Il y a deux sortes de **Checksum** : **Adler32** (qui est rapide) et **CRC32** (qui est plus lent mais légèrement plus précis).

```
//: c11:ZipCompress.java
// Emploi de la compression Zip pour compresser n'importe quel
// nombre de fichiers passés en ligne de commande.
import java.io.*;
import java.util.*;
import java.util.zip.*;

public class ZipCompress {
  // Lance les exeptions vers la console :
  public static void main(String[] args)
  throws IOException {
    FileOutputStream f =      new FileOutputStream("test.zip");
    CheckedOutputStream csum =      new CheckedOutputStream(
       f, new Adler32());
    ZipOutputStream out =      new ZipOutputStream(
       new BufferedOutputStream(csum));
    out.setComment("A test of Java Zipping");
    // Pas de getComment() correspondant, bien que.
    for(int i = 0; i < args.length; i++) {
      System.out.println(
        "Writing file " + args[i]);
      BufferedReader in =       new BufferedReader(
         new FileReader(args[i]));
      out.putNextEntry(new ZipEntry(args[i]));
      int c;
      while((c = in.read()) != -1)
        out.write(c);
      in.close();
    }
    out.close();
    // Validation de Checksum seulement après que
    // le fichier est été fermé !
    System.out.println("Checksum: " +
      csum.getChecksum().getValue());
    // Maintenant extrait les fichiers :
    System.out.println("Reading file");
    FileInputStream fi =       new FileInputStream("test.zip");
    CheckedInputStream csumi =      new CheckedInputStream(
       fi, new Adler32());
    ZipInputStream in2 =      new ZipInputStream(
       new BufferedInputStream(csumi));
    ZipEntry ze;
    while((ze = in2.getNextEntry()) != null) {
      System.out.println("Reading file " + ze);
      int x;
```

```
        while((x = in2.read()) != -1)
          System.out.write(x);
      }
      System.out.println("Checksum: " +
        csumi.getChecksum().getValue());
      in2.close();
      // Méthode alternative pour ouvrir et lire
      // les fichiers zip :
      ZipFile zf = new ZipFile("test.zip");
      Enumeration e = zf.entries();
      while(e.hasMoreElements()) {
        ZipEntry ze2 = (ZipEntry)e.nextElement();
        System.out.println("File: " + ze2);
        // ... et extrait les données comme précédemment.
      }
    }
} ///:~
```

Pour chaque fichier à ajouter à l'archive, vous devez appeler **putNextEntry()** et lui passer un objet **ZipEntry**. L'objet **ZipEntry** contient une interface extensible qui vous permet d'obtenir et de positionner toutes les données disponibles sur cette entrée précise dans votre fichier Zip : nom, tailles compressé et non-compressé, date, somme de contrôle CRC, données supplémentaires, commentaire, méthode de compression, et si il s'agit d'une entrée de répertoire. Toutefois, même si le format Zip possède une méthode pour établir un mot de passe, il n'est pas supporté dans la librairie Zip de Java. Et bien que **CheckedInputStream** et **CheckedOutputStream** supportent les deux contrôles de somme **Adler32** et **CRC32**, la classe **ZipEntry** supporte seulement une interface pour la CRC (Contrôle de Redondance Cyclique). C'est une restriction sous-jacente du format Zip, mais elle pourrait vous limiter d'utiliser l'**Adler32** plus rapide.

Pour extraire les fichiers, **ZipInputStream** a une méthode **getNextEntry()** qui renvoie la **ZipEntry** suivante si il y en a une. Comme alternative plus succincte, vous pouvez lire le fichier en utilisant un objet **ZipFile**, lequel possède une méthode **entries()** pour renvoyer une **Enumeration** au**ZipEntries**.

Afin de lire la somme de contrôle vous devrez d'une manière ou d'une autre avoir accès à l'objet **Checksum** associé. Ici, une référence vers les objets **CheckedOutputStream** et **CheckedInputStream** est retenue, mais vous pourriez aussi juste vous en tenir à une référence à l'objet **Checksum.**

Une méthode déconcertante dans les flux de Zip est **setComment()**. Comme montré plus haut, vous pouvez établir un commentaire lorsque vous écrivez un fichier, mais il n'y a pas de manière pour récupérer le commentaire dans le **ZipInputStream**. Les commentaires sont apparemment complètement supportés sur une base d'entrée-par-entrée par l'intermédiaire de ZipEntry.

Bien sûr, vous n'êtes pas limité aux fichiers lorsque vous utilisez les librairies **GZIP** et **Zip** — vous pouvez compresser n'importe quoi, y compris les données a envoyer en passant par une connexion réseau.

ARchives Java (JARs)

Le format Zip est aussi employé dans le format de fichier JAR (Java ARchive), qui est une manière de rassembler un groupe de fichiers dans un seul fichier compressé, tout à fait comme Zip. Cependant, comme tout le reste en Java, les fichiers JAR sont multi-plate-forme donc vous n'avez pas a vous soucier des distributions de plate-forme. Vous pouvez aussi inclure des fichiers audio et image en plus des fichiers class.

Les fichiers JAR sont particulièrement utile quand on a affaire à l'Internet. Avant les fichiers JAR, votre navigateur Web devait faire des requêtes répétées sur un serveur Web afin de télécharger tous les fichiers qui composaient une applet. En plus, chacun de ces fichiers n'était pas compressé. En combinant tous les fichiers d'une applet précise dans un seul fichier JAR, une seule requête au serveur est nécessaire et le transfert est plus rapide en raison de la compression. Et chaque entrée dans un fichier JAR peut être signée digitalement pour la sécurité (se référer à la documentation de Java pour les détails).

Un JAR consiste en un seul fichier contenant une collection de fichiers zippés ensemble avec un « manifeste » qui en fait la description. (Vous pouvez créer votre propre fichier manifeste; sinon le programme **jar** le fera pour vous.) Vous pouvez trouver plus de précisions sur les manifestes dans la documentation HTML du JDK.

L'utilitaire **jar** qui est fourni avec le JDK de Sun compresse automatiquement les fichiers de votre choix. Vous lui faites appel en ligne de commande :

```
jar [options] destination [manifest] inputfile(s)
```

Les options sont simplement une collection de lettres (aucun trait d'union ou autre indicateur n'est nécessaire). Les utilisateurs noterons la similitude avec les options **tar**. Celles-ci sont :

c	Crée une archive nouvelle ou vide.
t	Établit la table des matières.
x	Extrait tous les fichiers.
x file	Extrait le fichier nommé.
f	Dit : « Je vais vous donner le nom du fichier. » Si vous n'utilisez pas ceci, jar considère que sont entrée viendra de l'entrée standard, ou , si il crée un fichier, sa sortie ira vers la sortie standard.
m	Dit que le premier argument sera le nom du fichier manifeste crée par l'utilisateur.
v	Génère une sortie « verbose » décrivant ce que jar effectue.
0	Stocke seulement les fichiers; ne compresse pas les fichiers (utilisé pour créer un fichier JAR que l'on peut mettre dans le classpath).
M	Ne crée pas automatiquement un fichier manifeste.

Si un sous-répertoire est inclus dans les fichiers devant être placés dans le fichier JAR, ce sous-répertoire est ajouté automatiquement, incluant tous ces sous-répertoire, etc. Les informations de chemin sont ainsi préservées.

Voici quelques façon typiques d'invoquer **jar** :

```
jar cf myJarFile.jar *.class
```

Ceci crée un fichier JAR appelé **myJarFile.jar** qui contient tous les fichiers class du répertoire courant, avec la génération automatique d'un fichier manifeste.

```
jar cmf myJarFile.jar myManifestFile.mf *.class
```

Comme l'exemple précèdent, mais ajoute un fichier manifeste crée par l'utilisateur nommé **myManifestFile.mf**.

```
jar tf myJarFile.jar
```

Produit une table des matières des fichiers dans **myJarFile.jar**.

```
jar tvf myJarFile.jar
```

Ajoute le drapeau « verbose » pour donner des informations plus détaillées sur les fichiers dans **myJarFile.jar**.

```
jar cvf myApp.jar audio classes image
```

Supposant que **audio**, **classes,** et **image** sont des sous-répertoires, ceci combine tous les sous-répertoires dans le fichier **myApp.jar**. Le drapeau « verbose » est aussi inclus pour donner contrôle d'information supplémentaire pendant que le programme **jar** travaille.

Si vous créez un fichier JAR en utilisant l'option **o**, ce fichier pourra être placé dans votre CLASSPATH :

```
CLASSPATH="lib1.jar;lib2.jar;"
```

Ainsi Java pourra chercher dans **lib1.jar** et **lib2**pour trouver des fichiers class.

L'outil **jar** n'est pas aussi utile que l'utilitaire **zip**. Par exemple, vous ne pouvez ajouter ou mettre à jour un fichier JAR existant; vous pouvez créer des fichiers JAR seulement à partir de zéro. Aussi, vous ne pouvez déplacer les fichiers dans un fichier JAR, les effaçant dès qu'ils sont déplacés. Cependant un fichier JAR crée sur une plate-forme sera lisible de manière transparente par l'outil **jar** sur n'importe quelle autre plate-forme (un problème qui apparaît parfois avec l'utilitaire **zip**.

Comme vous le verrez dans le chapitre 13, les fichiers JAR sont aussi utilisés pour emballer les JavaBeans.

La sérialisation objet

La *sérialisation objet* en Java vous permet de prendre n'importe quel objet qui implémente l'interface **Serializable** et le dévie en une séquence de bytes qui pourront ensuite être complètement restaurés pour régénérer l'objet original. C'est même vrai à travers un réseau, ce qui signifie que le mécanisme de sérialisation compense automatiquement des différences dans les systèmes d'exploitation. C'est à dire, vous pouvez créer un objet sur un machine Windows, le sérialiser, et l'envoyer à travers le réseau sur une machine Unix où il sera correctement reconstruit. Vous n'avez pas à vous soucier de la représentation des données sur les différentes machines, l'ordonnancement des bytes, ou tout autres détails.

Par elle-même, la sérialisation objet est intéressante parce qu'elle vous permet de mettre en

application la *persistance légère* [lightweight persistence]. Rappelez-vous que la persistance signifie que la durée de vie de l'objet n'est pas déterminée tant qu'un programme s'exécute — l'objet vit *dans l'intervalle* des invocations du programme. En prenant un objet sérialisable et en l'écrivant sur le disque, puis en ressortant cet objet lors de la remise en route du programme, vous êtes alors capable de produire l'effet de persistance. La raison pour laquelle elle est appelée « légère » est que vous pouvez simplement définir un objet en employant un certain type de mot-clé pour la « persistance » et de laisser le système prendre soin des détails (bien que cela peut bien arriver dans le futur). À la place de cela, vous devrez sérialiser et désérialiser explicitement les objets dans votre programme.

La sérialisation objet a été ajoutée au langage pour soutenir deux caractéristiques majeures. La *remote method invocation* (RMI) de Java permet aux objets qui vivent sur d'autres machines de se comporter comme si ils vivaient sur votre machine. Lors de l'envoi de messages aux objets éloignés, la sérialisation d'objet est nécessaire pour transporter les arguments et les valeurs retournées. RMI est abordé au Chapitre 15.

La sérialisation des objets est aussi nécessaire pour les JavaBeans, décrit au Chapitre 13. Quand un Bean est utilisé, son information d'état est généralement configuré au moment de la conception. Cette information d'état doit être stockée et récupérée ultérieurement quand le programme est démarré; la sérialisation objet accomplit cette tâche.

Sérialiser un objet est assez simple, aussi longtemps que l'objet implémente l'interface **Serializable** (cette interface est juste un drapeau et n'a pas de méthode). Quand la sérialisation est ajoutée au langage, de nombreuses classes sont changés pour les rendre sérialisables, y compris tous les enveloppeurs [wrappers] pour les types de primitives, toutes les classes de récipients [container], et bien d'autres. Même les objets **Class** peuvent être sérialisés. (Voir le Chapitre 12 pour ce que cela implique.)

Pour sérialiser un objet, vous créez une sorte d'objet **OutputStream** et l'enveloppez ensuite dans un objet **ObjectOutputStream**. À ce point vous avez seulement besoin d'appeler **writeObject()** et votre objet est sérialisé et envoyé à l'**OutputStream**. Pour inverser le processus, vous enveloppez un **InputStream** dans un **ObjectInputStream** et appelez **readObject()**. Ce qui renvoie, comme d'habitude, une référence à un **Objet** dont on a sur-forcé le type [upcast], ainsi vous devrez sous-forcer pour préparer les objets directement.

Un aspect particulièrement astucieux de la sérialisation objet est qu'elle ne sauve pas uniquement une image de votre objet, mais cela s'occupe aussi de toutes les références contenues dans votre objet et sauve *ces* objets, et poursuit dans toutes les références de ces objets, etc. Ceci est parfois rapporté comme le « Web des objets » auquel un simple objet peut être connecté, et il comprend des tableaux de références aux objets aussi bien que d'objets membres. Si vous devez entretenir votre propre schéma de sérialisation, entretenir le code pour suivre tous ces liens serait un peu un casse-tête. Pourtant, la sérialisation d'objet Java semble s'en sortir sans faute, sans aucun doute en utilisant un algorithme optimalisé qui traverse le Web des objets. L'exemple suivant teste le mécanisme de sérialisation en créant un « vers » d'objets liés, chacun d'entre eux ayant un lien jusqu'au prochain segment dans le vers en plus d'un tableau de références aux objets d'une classe différent, **Data** :

```
//: c11:Worm.java
// Demontre la sérialisation des objets.
import java.io.*;
```

```
class Data implements Serializable {
  private int i;
  Data(int x) { i = x; }
  public String toString() {
    return Integer.toString(i);
  }
}

public class Worm implements Serializable {
  // Génère une valeur d'int aléatoire :
  private static int r() {
    return (int)(Math.random() * 10);
  }
  private Data[] d = {
    new Data(r()), new Data(r()), new Data(r())
  };
  private Worm next;
  private char c;
  // Value de i == nombre de segments
  Worm(int i, char x) {
    System.out.println(" Worm constructor: " + i);
    c = x;
    if(--i > 0)
      next = new Worm(i, (char)(x + 1));
  }
  Worm() {
    System.out.println("Default constructor");
  }
  public String toString() {
    String s = ":" + c + "(";
    for(int i = 0; i < d.length; i++)
      s += d[i].toString();
    s += ")";
    if(next != null)
      s += next.toString();
    return s;
  }
  // Lance les exeptions vers la console :
  public static void main(String[] args)
  throws ClassNotFoundException, IOException {
    Worm w = new Worm(6, 'a');
    System.out.println("w = " + w);
    ObjectOutputStream out =      new ObjectOutputStream(
        new FileOutputStream("worm.out"));
    out.writeObject("Worm storage");
    out.writeObject(w);
    out.close(); // Vide aussi la sortie
```

```
    ObjectInputStream in =      new ObjectInputStream(
      new FileInputStream("worm.out"));
    String s = (String)in.readObject();
    Worm w2 = (Worm)in.readObject();
    System.out.println(s + ", w2 = " + w2);
    ByteArrayOutputStream bout =      new ByteArrayOutputStream();
    ObjectOutputStream out2 =      new ObjectOutputStream(bout);
    out2.writeObject("Worm storage");
    out2.writeObject(w);
    out2.flush();
    ObjectInputStream in2 =      new ObjectInputStream(
      new ByteArrayInputStream(
        bout.toByteArray()));
    s = (String)in2.readObject();
    Worm w3 = (Worm)in2.readObject();
    System.out.println(s + ", w3 = " + w3);
  }
} ///:~
```

L'essentiel de tout cela est de rendre quelque chose raisonnablement complexe qui ne puisse pas facilement être sérialisé. L'action de sérialiser, cependant, est plutôt simple. Une fois que l'**ObjectOutputStream** est crée depuis un autre flux, **writeObject()** sérialise l'objet. Notez aussi l'appel de **writeObject()** pour un **String**. Vous pouvez aussi écrire tous les types de données primitives utilisant les même méthodes qu'**DataOutputStream** (ils partagent la même interface).

Il y a deux portions de code séparées qui ont une apparence similaire. La première écrit et lit et la seconde, pour varier, écrit et lit un **ByteArray**. Vous pouvez lire et écrire un objet en utilisant la sérialisation vers n'importe quel **DataInputStream** ou **DataOutputStream** incluant, comme vous le verrez dans le Chapitre 15, un réseau. La sortie d'une exécution donne :

```
Worm constructor: 6
Worm constructor: 5
Worm constructor: 4
Worm constructor: 3
Worm constructor: 2
Worm constructor: 1
w = :a(262):b(100):c(396):d(480):e(316):f(398)
Worm storage, w2 = :a(262):b(100):c(396):d(480):e(316):f(398)
Worm storage, w3 = :a(262):b(100):c(396):d(480):e(316):f(398)
```

Vous pouvez voir que l'objet déserialisé contient vraiment tous les liens qui étaient dans l'objet original.

Notons qu'aucun constructeur, même pas le constructeur par défaut, n'est appelé dans le processus de désérialisation d'un objet **Serializable**. L'objet entier est restauré par récupération des données depuis l'**InputStream**.

La sérialisation objet est orientée-**byte**, et ainsi emploie les hiérarchies d'**InputStream** et d'**OutputStream**.

Trouver la classe

Vous devez vous demander ce qui est nécessaire pour qu'un objet soit récupéré depuis son état sérialisé. Par exemple, supposons que vous sérialisez un objet et que vous l'envoyez comme un fichier à travers un réseau vers une autre machine. Un programme sur l'autre machine pourra-t-il reconstruire l'objet en utilisant seulement le contenu du fichier ?

La meilleure manière de répondre a cette question est (comme d'habitude) en accomplissant une expérience. Le fichier suivant file dans le sous-répertoire pour ce chapitre :

```
//: c11:Alien.java
// Une classe sérializable.
import java.io.*;

public class Alien implements Serializable {
} ///:~
```

Le fichier qui crée et sérialise un objet **Alien** va dans le même répertoire :

```
//: c11:FreezeAlien.java
// Crée un fichier de sortie sérialisé.
import java.io.*;

public class FreezeAlien {
  // Lance les exeptions vers la console:
  public static void main(String[] args)
  throws IOException {
    ObjectOutput out =
      new ObjectOutputStream(
        new FileOutputStream("X.file"));
    Alien zorcon = new Alien();
    out.writeObject(zorcon);
  }
} ///:~
```

Plutôt que de saisir et de traiter les exeptions, ce programme prend une approche rapide et sale qui passe les exeptions en dehors de **main()**, ainsi elle seront reportés en ligne de commande.

Une fois que le programme est compilé et exécuté, copiez le **X.file** résultant dans un sous répertoire appelé**xfiles**, où va le code suivant :

```
//: c11:xfiles:ThawAlien.java
// Essaye de récupérer un fichier sérialisé sans
// la classe de l'objet qui est stocké dans ce fichier.
import java.io.*;

public class ThawAlien {
  public static void main(String[] args)
  throws IOException, ClassNotFoundException {
    ObjectInputStream in =      new ObjectInputStream(
       new FileInputStream("X.file"));
```

```
    Object mystery = in.readObject();
    System.out.println(mystery.getClass());
  }
} ///:~
```

Ce programme ouvre le fichier et lit dans l'objet **mystery** avec succès. Pourtant, dès que vous essayez de trouver quelque chose à propos de l'objet — qui nécessite l'objet **Class** pour **Alien** — la Machine Virtuelle Java (JVM) ne peut pas trouver **Alien.class** (à moins qu'il arrive qu'il soit dans le Classpath, ce qui n'est pas le cas dans cet exemple). Vous obtiendrez un **ClassNotFoundException**. (Une fois encore, toute les preuves de l'existence de vie alien disparaît avant que la preuve de son existence soit vérifiée !)

Si vous espérez en faire plus après avoir récupéré un objet qui a été sérialisé, vous devrez vous assurer que la JVM puisse trouver les fichiers **.class** soit dans le chemin de class local ou quelque part sur l'Internet.

Contrôler la sérialisation

Comme vous pouvez le voir, le mécanisme de sérialisation par défaut est d'un usage trivial. Mais que faire si vous avez des besoins spéciaux ? Peut-être que vous avez des problèmes de sécurité spéciaux et que vous ne voulez pas sérialiser des parties de votre objet, ou peut-être que cela n'a pas de sens pour un sous-objet d'être sérialisé si cette partie doit être de nouveau crée quand l'objet est récupéré.

Vous pouvez contrôler le processus de sérialisation en implémentant l'interface **Externalizable** à la place de l'interface **Serializable**. L'interface **Externalizable** étend l'interface **Serializable** et ajoute deux méthodes, **writeExternal()** et **readExternal(),** qui sont automatiquement appelées pour votre objet pendant la sérialisation et la désérialisation afin que vous puissiez exécuter vos opérations spéciales.

L'exemple suivant montre des implémentations simple des méthodes de l'interface **Externalizable**. Notez que **Blip1** et **Blip2** sont presque identiques à l'exception d'une subtile différence (voyez si vous pouvez la découvrir en regardant le code) :

```
//: c11:Blips.java
// Emploi simple d'Externalizable & un piège.
import java.io.*;
import java.util.*;

class Blip1 implements Externalizable {
  public Blip1() {
    System.out.println("Blip1 Constructor");
  }
  public void writeExternal(ObjectOutput out)
      throws IOException {
    System.out.println("Blip1.writeExternal");
  }
  public void readExternal(ObjectInput in)
    throws IOException, ClassNotFoundException {
    System.out.println("Blip1.readExternal");
```

```
  }
}

class Blip2 implements Externalizable {
  Blip2() {
    System.out.println("Blip2 Constructor");
  }
  public void writeExternal(ObjectOutput out)
      throws IOException {
    System.out.println("Blip2.writeExternal");
  }
  public void readExternal(ObjectInput in)
     throws IOException, ClassNotFoundException {
    System.out.println("Blip2.readExternal");
  }
}

public class Blips {
  // Lance les exeptions vers la console :
  public static void main(String[] args)
  throws IOException, ClassNotFoundException {
    System.out.println("Constructing objects:");
    Blip1 b1 = new Blip1();
    Blip2 b2 = new Blip2();
    ObjectOutputStream o =      new ObjectOutputStream(
       new FileOutputStream("Blips.out"));
    System.out.println("Saving objects:");
    o.writeObject(b1);
    o.writeObject(b2);
    o.close();
    //Maintenant faites les revenir :
    ObjectInputStream in =      new ObjectInputStream(
       new FileInputStream("Blips.out"));
    System.out.println("Recovering b1:");
    b1 = (Blip1)in.readObject();
    // OOPS! Lance une exeption :
//! System.out.println("Recovering b2:");
//! b2 = (Blip2)in.readObject();
  }
} ///:~
```

La sortie pour ce programme est :

```
Constructing objects:
Blip1 Constructor
Blip2 Constructor
Saving objects:
Blip1.writeExternal
```

```
Blip2.writeExternal
Recovering b1:
Blip1 Constructor
Blip1.readExternal
```

La raison pour laquelle l'objet **Blip2** n'est pas récupéré est que le fait d'essayer provoque une exeption. Vous pouvez voir la différence entre**Blip1** et **Blip2** ? Le constructeur de **Blip1** est **public**, tandis que le constructeur de **Blip2** ne l'est pas, et cela lance une exeption au recouvrement. Essayez de rendre le constructor de **Blip2 public** et retirez les commentaires **//!** pour voir les résultats correct.

Quand **b1** est récupéré, le constructeur par défaut **Blip1** est appelé. Ceci est différent de récupérer un objet **Serializable**, dans lequel l'objet est construit entièrement de ses bits enregistrés, sans appel au constructeur. Avec un objet **Externalizable**, tous les comportements de construction par défaut se produisent (incluant les initialisations à ce point du champ de définition), et *alors* **readExternal()** est appelé. Vous devez prendre en compte ceci en particulier, le fait que toute la construction par défaut a toujours lieu pour produire le comportement correct dans vos objets **Externalizable**.

Voici un exemple qui montre ce que vous devez faire pour complétement stocker et retrouver un objet **Externalizable** :

```
//: c11:Blip3.java
// Reconstruction d'un objet externalizable.
import java.io.*;
import java.util.*;

class Blip3 implements Externalizable {
  int i;
  String s; // Aucune initialisation
  public Blip3() {
    System.out.println("Blip3 Constructor");
    // s, i n'est pas initialisé
  }
  public Blip3(String x, int a) {
    System.out.println("Blip3(String x, int a)");
    s = x;
    i = a;
    // s & i initialisé seulement dans le non
    // constructeur par défaut.
  }
  public String toString() { return s + i; }
  public void writeExternal(ObjectOutput out)
  throws IOException {
    System.out.println("Blip3.writeExternal");
    // Vous devez faire ceci :
    out.writeObject(s);
    out.writeInt(i);
  }
```

```
  public void readExternal(ObjectInput in)
  throws IOException, ClassNotFoundException {
    System.out.println("Blip3.readExternal");
    // Vous devez faire ceci :
    s = (String)in.readObject();
    i =in.readInt();
  }
  public static void main(String[] args)
  throws IOException, ClassNotFoundException {
    System.out.println("Constructing objects:");
    Blip3 b3 = new Blip3("A String ", 47);
    System.out.println(b3);
    ObjectOutputStream o =      new ObjectOutputStream(
       new FileOutputStream("Blip3.out"));
    System.out.println("Saving object:");
    o.writeObject(b3);
    o.close();
    // Maintenant faites le revenir :
    ObjectInputStream in =      new ObjectInputStream(
       new FileInputStream("Blip3.out"));
    System.out.println("Recovering b3:");
    b3 = (Blip3)in.readObject();
    System.out.println(b3);
  }
} ///:~
```

Les champs **s** et **i** sont initialisés seulement dans le second constructeur, mais pas dans le constructeur par défaut. Ceci signifie que si vous n'initialisez pas **s** et **i** dans **readExternal()**, il sera alors **null** (parce que le stockage pour l'objet arrive nettoyé à zéro dans la première étape de la création de l'objet). Si vous enlevez les commentaires sur les deux lignes suivant les phrases « Vous devez faire ceci » et lancez le programme, vous verrez que lorsque l'objet est récupéré, **s** est **null** et **i** est zéro.

Si vous l'héritage se fait depuis un objet **Externalizable**, vous appellerez typiquement les versions classe-de-base de **writeExternal()** et **readExternal()** pour fournir un stockage et une récupération propre des composants de classe-de-base.

Ainsi pour faire fonctionner correctement les choses vous ne devrez pas seulement écrire les données importantes depuis l'objet pendant la méthode **writeExternal()** (il n'y a pas de comportement par défaut qui écrit n'importe quels objets pour un objet **Externalizable** object), mais vous devrez aussi récupérer ces données dans la méthode **readExternal()**. Ceci peut être un petit peu confus au premier abord parce que le constructeur par défaut du comportement pour un objet **Externalizable** peut le faire ressembler à une sorte de stockage et de récupération ayant lieu automatiquement. Ce n'est pas le cas.

Le mot-clé « transient »

Quand vous contrôlez la sérialisation, il peut y avoir un sous-objet précis pour qui vous ne voulez pas que le mécanisme de sérialisation java sauve et restaure automatiquement. C'est commu-

nément le cas si le sous-objet représente des informations sensibles que vous ne désirez pas sérialiser, comme un mot de passe. Même si cette information est **private** dans l'objet, une fois qu'elle est sérialisée il est possible pour quelqu'un d'y accéder en lisant un fichier ou en interceptant une transmission réseau.

Une manière de prévenir les parties sensibles de votre objet d'être sérialisé est d'implémenter votre classe comme **Externalizable**, comme montré précédemment. Ainsi rien n'est sérialisé automatiquement et vous pouvez sérialiser explicitement seulement les parties nécessaires dans **writeExternal()**.

Si vous travaillez avec un objet **Serializable**, néanmoins, toutes les sérialisation arrivent de façon automatique. Pour contrôler ceci, vous pouvez fermer la sérialisation sur une base de champ-par-champ en utilisant le mot **transient**, lequel dit « Ne t'embarrasse pas a sauver ou restaurer ceci — Je me charge de ça. »

Par exemple, considérons un objet **Login** qui conserve les informations à propos d'un login de session particulier. Supposez que, dès que vous vérifiez le login, vous désirez stocker les données, mais sans le mot de passe. La manière la plus simple pour réaliser ceci est en d'implémentant **Serializable** et en marquant le champ **password** comme **transient**. Voici ce à quoi cela ressemble :

```
//: c11:Logon.java
// Explique le mot « transient. »
import java.io.*;
import java.util.*;

class Logon implements Serializable {
  private Date date = new Date();
  private String username;
  private transient String password;
  Logon(String name, String pwd) {
    username = name;
    password = pwd;
  }
  public String toString() {
    String pwd =      (password == null) ? "(n/a)" : password;
    return "logon info: \n   " +
      "username: " + username +
      "\n   date: " + date +
      "\n   password: " + pwd;
  }
  public static void main(String[] args)
  throws IOException, ClassNotFoundException {
    Logon a = new Logon("Hulk", "myLittlePony");
    System.out.println( "logon a = " + a);
      ObjectOutputStream o =       new ObjectOutputStream(
        new FileOutputStream("Logon.out"));
    o.writeObject(a);
    o.close();
    // Délai :
    int seconds = 5;
```

```
    long t = System.currentTimeMillis()
        + seconds * 1000;
    while(System.currentTimeMillis() < t)
      ;
    // Maintenant faites les revenir :
    ObjectInputStream in =      new ObjectInputStream(
       new FileInputStream("Logon.out"));
    System.out.println(
      "Recovering object at " + new Date());
    a = (Logon)in.readObject();
    System.out.println( "logon a = " + a);
  }
} ///:~
```

Vous pouvez voir que les champs **date** et **username** sont normaux (pas **transient**), et ils sont ainsi sérialisés automatiquement. Pourtant, **password** est **transient**, et donc n'est pas stocké sur le disque; aussi le mécanisme de sérialisation ne fait aucune tentative pour le récupérer. La sortie donne :

```
logon a = logon info:
  username: Hulk
  date: Sun Mar 23 18:25:53 PST 1997
  password: myLittlePony
Recovering object at Sun Mar 23 18:25:59 PST 1997
logon a = logon info:
  username: Hulk
  date: Sun Mar 23 18:25:53 PST 1997
  password: (n/a)
```

Lorsque l'objet est récupéré, le champ de **password** est **null**. Notez que **toString()** est obligé de contrôler la valeur **null** de **password** parcequ'il essaye d'assembler un objet String utilisant l'opérateur surchargé ' + ', et que cet opérateur est confronté à une référence de type **null**, on aurait donc un **NullPointerException.** (Les nouvelles versions de Java contiendront peut être du code pour résoudre ce problème.)

Vous pouvez aussi voir que le champ date est stocké sur et récupéré depuis le disque et n'en génère pas une nouvelle.

Comme les objets **Externalizable** ne stockent pas tous leurs champs par défaut, le mot-clé transient est a employer avec les objets **Serializable** seulement.

Une alternative à Externalizable

Si vous n'êtes pas enthousiasmé par l'implémentation de l'interface **Externalizable**, il y a une autre approche. Vous pouvez implémenter l'interface **Serializable** et *ajouter* (notez que je dit « ajouter » et non pas « imposer » ou « implémenter ») des méthodes appelées **writeObject()** et **readObject()** qui seront automatiquement appelées quand l'objet est sérialisé et désérialisé, respectivement. C'est à dire, si vous fournissez ces deux méthodes elles seront employées à la place de la sérialisation par défaut.

Ces méthodes devront avoir ces signatures exactes :

```
private void
  writeObject(ObjectOutputStream stream)
    throws IOException;

private void
  readObject(ObjectInputStream stream)
    throws IOException, ClassNotFoundException
```

De toute façon, tout ce qui est défini dans une **interface** est automatiquement **public** donc **writeObject()** et **readObject()** doivent être **private**, à ce moment là ils feront partie d'une **interface**. Puisque vous devez suivre exactement les signatures, l'effet est le même que si vous implémentiez une **interface.**

Il apparaîtra que lorsque vous appelez **ObjectOutputStream.writeObject()**, l'objet **Serializable** que vous lui transmettez est interrogé (utilisant la réflexion, pas de doute) pourvoir si il implémente son propre **writeObject()**. Si c'est le cas, le processus normale de sérialisation est omis et est le **writeObject()**appelé. Le même type de situation existe pour **readObject()**.

Il y a une autre entorse. À l'intérieur de votre **writeObject( )**, vous pouvez choisir d'exécuter l'action **writeObject()** par défaut en appelant **defaultWriteObject()**. Également, dans **readObject()**vous pouvez appeler **defaultReadObject()**. Voici un exemple simple qui démontre comment vous pouvez contrôler le stockage et la récupération d'un objet **Serializable :**

```
//: c11:SerialCtl.java
// Contrôler la sérialisation en ajoutant vos propres
// méthodes writeObject() et readObject().
import java.io.*;

public class SerialCtl implements Serializable {
  String a;
  transient String b;
  public SerialCtl(String aa, String bb) {
    a = "Not Transient: " + aa;
    b = "Transient: " + bb;
  }
  public String toString() {
    return a + "\n" + b;
  }
  private void
    writeObject(ObjectOutputStream stream)
      throws IOException {
    stream.defaultWriteObject();
    stream.writeObject(b);
  }
  private void
    readObject(ObjectInputStream stream)
      throws IOException, ClassNotFoundException {
    stream.defaultReadObject();
```

```
    b = (String)stream.readObject();
  }
  public static void main(String[] args)
  throws IOException, ClassNotFoundException {
    SerialCtl sc =
      new SerialCtl("Test1", "Test2");
    System.out.println("Before:\n" + sc);
    ByteArrayOutputStream buf =
      new ByteArrayOutputStream();
    ObjectOutputStream o =      new ObjectOutputStream(buf);
    o.writeObject(sc);
    // Maintenant faites les revenir :
    ObjectInputStream in =      new ObjectInputStream(
       new ByteArrayInputStream(
         buf.toByteArray()));
    SerialCtl sc2 = (SerialCtl)in.readObject();
    System.out.println("After:\n" + sc2);
  }
} ///:~
```

Dans cet exemple, un champ **String** est normal et le second est **transient**, pour prouver que le champ non-**transient** est sauvé par la méthode **defaultWriteObject()** et le que le champ **transient** est sauvé et récupéré explicitement. Les champs sont initialisés dans le constructeur plutôt qu'au point de définition pour prouver qu'ils n'ont pas été initialisés par un certain mécanisme automatique durant la sérialisation.

Si vous employez le mécanisme par défaut pour écrire les parties non-**transient** de votre objet, vous devrez appeler **defaultWriteObject()** comme la première action dans **writeObject()** et **defaultReadObject()** comme la première action dans **readObject()**. Ce sont d'étranges appels à des méthodes. Il apparaît, par exemple, que vous appelez **defaultWriteObject()** pour un **ObjectOutputStream** et ne lui passez aucun argument, mais il tourne d'une manière ou d'une autre autour et connaît la référence à votre objet et comment écrire toutes les parties non-**transient**. Étrange.

Le stockage et la récupération des objets **transient** utilisent un code plus familier. Et cependant, pensez à ce qu'il se passe ici. Dans **main()**, un objet **SerialCtl** est créé, puis est sérialisé en un **ObjectOutputStream.** (Notez dans ce cas qu'un tampon est utilisé à la place d'un fichier — c'est exactement pareil pour tout l' **ObjectOutputStream**.) La sérialisation survient à la ligne :

```
o.writeObject(sc);
```

La méthode **writeObject()** doit examiner **sc** pour voir si il possède sa propre méthode **writeObject()**. (Non pas en contrôlant l'interface — il n'y en a pas — ou le type de classe, mais en recherchant en fait la méthode en utilisant la réflexion.) Si c'est le cas, elle l'utilise. Une approche similaire garde true pour **readObject()**. Peut-être que c'est la seule réelle manière dont ils peuvent résoudre le problème, mais c'est assurément étrange.

Versioning

Il est possible que vous désiriez changer la version d'une classe sérialisable (les objets de la classe original peuvent être stockés dans un base de donnée, par exemple). Ceci est supporté mais

vous devrez probablement le faire seulement dans les cas spéciaux, et cela requiert un profondeur supplémentaire de compréhension qui ne sera pas tenté d'atteindre ici. Les documents HTML du JDK téléchargeables depuis *java.sun.com* couvrent ce sujet de manière très approfondie.

Vous pourrez aussi noter dans la documentation HTML du JDK que de nombreux commentaires commencent par :

Attention : *Les objets sérialisés de cette classe ne seront pas compatibles avec les futures versions de Swing. Le support actuel de la sérialisation est approprié pour le stockage à court terme ou le RMI entre les applications. ...*

Ceci parce que le mécanisme de versionning est trop simple pour fonctionner de manière fiable dans toutes les situations, surtout avec les JavaBeans. Ils travaillent sur un correction de la conception, et c'est le propos de l'avertissement.

Utiliser la persistence

Il est plutôt attrayant de faire appel à la technologie de la sérialisation pour stocker certains états de votre programme afin que vous puissiez facilement récupérer le programme dans l'état actuel plus tard. Mais avant de de pouvoir faire cela, il faut répondre à certaines questions. Qu'arrive-t-il si vous sérialisez deux objets qui ont tous les deux une référence à un troisième objet ? Quand vous récupérez ces deux objets depuis leur état sérialisé, aurez vous une seule occurrence du troisième objet ? Que ce passe-t-il si vous sérialisez vos deux objets pour séparer les fichiers et les désérialisez dans différentes parties de votre code ?

Voici un exemple qui montre le problème :

```
//: c11:MyWorld.java
import java.io.*;
import java.util.*;

class House implements Serializable {}

class Animal implements Serializable {
  String name;
  House preferredHouse;
  Animal(String nm, House h) {
    name = nm;
    preferredHouse = h;
  }
  public String toString() {
    return name + "[" + super.toString() +
      "], " + preferredHouse + "\n";
  }
}

public class MyWorld {
  public static void main(String[] args)
  throws IOException, ClassNotFoundException {
    House house = new House();
```

```
    ArrayList  animals = new ArrayList();
    animals.add(
      new Animal("Bosco the dog", house));
    animals.add(
      new Animal("Ralph the hamster", house));
    animals.add(
      new Animal("Fronk the cat", house));
    System.out.println("animals: " + animals);

    ByteArrayOutputStream buf1 =
      new ByteArrayOutputStream();
    ObjectOutputStream o1 =      new ObjectOutputStream(buf1);
    o1.writeObject(animals);
    o1.writeObject(animals); // Écrit un 2éme jeu
    // Écrit vers un flux différent :
    ByteArrayOutputStream buf2 =
      new ByteArrayOutputStream();
    ObjectOutputStream o2 =      new ObjectOutputStream(buf2);
    o2.writeObject(animals);
    // Now get them back:
    ObjectInputStream in1 =      new ObjectInputStream(
       new ByteArrayInputStream(
         buf1.toByteArray()));
    ObjectInputStream in2 =      new ObjectInputStream(
       new ByteArrayInputStream(
         buf2.toByteArray()));
    ArrayList animals1 =
      (ArrayList)in1.readObject();
    ArrayList animals2 =
      (ArrayList)in1.readObject();
    ArrayList animals3 =
      (ArrayList)in2.readObject();
    System.out.println("animals1: " + animals1);
    System.out.println("animals2: " + animals2);
    System.out.println("animals3: " + animals3);
  }
} ///:~
```

Une chose intéressante ici est qu'il est possible d'utiliser la sérialisation d'objet depuis et vers un tableau de bytes comme une manière de faire une « copie en profondeur » de n'importe quel objet qui est**Serializable.** (une copie en profondeur veux dire que l'on copie la structure complète des objets, plutôt que seeulement l'objet de base et ses références.) La copie est abordée en profondeur dans l'Annexe A.

Les objets **Animal** contiennent des champs de type **House**. Dans **main()**, une **ArrayList** de ces **Animal**s est crée et est sérialisée deux fois vers un flux et ensuite vers un flux distinct. Quand ceci est désérialisé et affiché, on obtient les résultats suivant pour une exécution (les objets seront dans des emplacements mémoire différents à chaque exécution) :

```
animals: [Bosco the dog[Animal@1cc76c], House@1cc769
, Ralph the hamster[Animal@1cc76d], House@1cc769
, Fronk the cat[Animal@1cc76e], House@1cc769
]
animals1: [Bosco the dog[Animal@1cca0c], House@1cca16
, Ralph the hamster[Animal@1cca17], House@1cca16
, Fronk the cat[Animal@1cca1b], House@1cca16
]
animals2: [Bosco the dog[Animal@1cca0c], House@1cca16
, Ralph the hamster[Animal@1cca17], House@1cca16
, Fronk the cat[Animal@1cca1b], House@1cca16
]
animals3: [Bosco the dog[Animal@1cca52], House@1cca5c
, Ralph the hamster[Animal@1cca5d], House@1cca5c
, Fronk the cat[Animal@1cca61], House@1cca5c
]
```

Bien sur vous vous attendez à ce que les objets déserialisés aient des adresses différentes des originaux. Mais notez que dans **animals1** et **animals2** les mêmes adresses apparaissent, incluant les références à l'objet **House** que tous les deux partagent. D'un autre coté, quand **animals3** est récupéré le système n'a pas de moyen de savoir que les objets de l'autre flux sont des alias des objets du premier flux, donc il crée un réseau d'objets complétement différent.

Aussi longtemps que vos sérialisez tout dans un flux unique, vous pourrez récupérer le même réseau d'objets que vous avez écrits, sans aucune duplication accidentelle d'objets. Bien sûr, vous pouvez modifier l'état de vos objets entre la période d'écriture du premier et du dernier, mais c'est de votre responsabilité — les objets seront écrit dans l'état où ils sont quel qu'il soit (et avec les connexions quelles qu'elles soient qu'ils ont avec les autres objets) au moment ou vous les sérialiserez.

La chose la plus sûre à faire si vous désirez sauver l'état d'un système est de sérialiser comme une opération « atomique. » Si vous sérialisez quelque chose, faites une autre action, et en sérialisez une autre en plus, etc., alors vous ne stockerez pas le système sûrement. Au lieu de cela, mettez tous les objets qui comprennent l'état de votre système dans un simple conteneur et écrivez simplement ce conteneur à l'extérieur en une seule opération. Ensuite vous pourrez aussi bien le récupérer avec un simple appel à une méthode.

L'exemple suivant est un système imaginaire de conception assistée par ordinateur (CAD) qui démontre cette approche. En plus, il projette dans le sujet des champs **static** — si vous regardez la documentation vous verrez que **Class** est **Serializable**, donc il sera facile de stocker les champs **static** en sérialisant simplement l'objet **Class**. Cela semble comme une approche sensible, en tous cas.

```
//: c11:CADState.java
// Sauve et récupère l'état de la
// simulation d'un système de CAD.
import java.io.*;
import java.util.*;

abstract class Shape implements Serializable {
```

```
  public static final int
    RED = 1, BLUE = 2, GREEN = 3;
  private int xPos, yPos, dimension;
  private static Random r = new Random();
  private static int counter = 0;
  abstract public void setColor(int newColor);
  abstract public int getColor();
  public Shape(int xVal, int yVal, int dim) {
    xPos = xVal;
    yPos = yVal;
    dimension = dim;
  }
  public String toString() {
    return getClass() +
      " color[" + getColor() +
      "] xPos[" + xPos +
      "] yPos[" + yPos +
      "] dim[" + dimension + "]\n";
  }
  public static Shape randomFactory() {
    int xVal = r.nextInt() % 100;
    int yVal = r.nextInt() % 100;
    int dim = r.nextInt() % 100;
    switch(counter++ % 3) {
      default:
      case 0: return new Circle(xVal, yVal, dim);
      case 1: return new Square(xVal, yVal, dim);
      case 2: return new Line(xVal, yVal, dim);
    }
  }
}

class Circle extends Shape {
  private static int color = RED;
  public Circle(int xVal, int yVal, int dim) {
    super(xVal, yVal, dim);
  }
  public void setColor(int newColor) {
    color = newColor;
  }
  public int getColor() {
    return color;
  }
}

class Square extends Shape {
  private static int color;
```

```
  public Square(int xVal, int yVal, int dim) {
    super(xVal, yVal, dim);
    color = RED;
  }
  public void setColor(int newColor) {
    color = newColor;
  }
  public int getColor() {
    return color;
  }
}

class Line extends Shape {
  private static int color = RED;
  public static void
  serializeStaticState(ObjectOutputStream os)
      throws IOException {
    os.writeInt(color);
  }
  public static void
  deserializeStaticState(ObjectInputStream os)
      throws IOException {
    color = os.readInt();
  }
  public Line(int xVal, int yVal, int dim) {
    super(xVal, yVal, dim);
  }
  public void setColor(int newColor) {
    color = newColor;
  }
  public int getColor() {
    return color;
  }
}

public class CADState {
  public static void main(String[] args)
  throws Exception {
    ArrayList shapeTypes, shapes;
    if(args.length == 0) {
      shapeTypes = new ArrayList();
      shapes = new ArrayList();
      // Ajoute des références aux objets de class :
      shapeTypes.add(Circle.class);
      shapeTypes.add(Square.class);
      shapeTypes.add(Line.class);
      // Fait quelques formes :
```

```
      for(int i = 0; i < 10; i++)
        shapes.add(Shape.randomFactory());
      // Établit toutes les couleurs statiques en GREEN:
      for(int i = 0; i < 10; i++)
        ((Shape)shapes.get(i))
          .setColor(Shape.GREEN);
      // Sauve le vecteur d'état :
      ObjectOutputStream out =       new ObjectOutputStream(
          new FileOutputStream("CADState.out"));
      out.writeObject(shapeTypes);
      Line.serializeStaticState(out);
      out.writeObject(shapes);
    } else { // C'est un argument de ligne de commande
      ObjectInputStream in =       new ObjectInputStream(
          new FileInputStream(args[0]));
      // Read in the same order they were written:
      shapeTypes = (ArrayList)in.readObject();
      Line.deserializeStaticState(in);
      shapes = (ArrayList)in.readObject();
    }
    // Affiche les formes :
    System.out.println(shapes);
  }
} ///:~
```

La classe **Shape implemente Serializable**, donc tout ce qui est hérité de **Shape** est aussi automatiquement **Serializable**. Chaque Shape contient des données, et chaque classe **Shape** dérivée contient un champ **static** qui détermine la couleur de tous ces types de **Shape**s. (Placer un champ static dans la classe de base ne donnera qu'un seul champ, puisque les champs static ne sont pas reproduit dans les classes dérivés.) Les méthodes dans les classes de base peuvent être surpassées [*overridden*] pour établir les couleurs des types variables (les méthodes **static** ne sont pas dynamiquement délimitées, donc ce sont des méthodes normales). La méthode **randomFactory()** crée un **Shape** différent chaque fois que vous y faites appel, utilisant des valeurs aléatoires pour les données du **Shape**.

Dans **main()**, une **ArrayList** est employée pour tenir les objets de **Class** et les autres pour tenir les formes. Si vous ne fournissez pas un argument en ligne de commande la **shapeTypes ArrayList** est créé et les objets Class sont ajoutés, et ensuite l'**ArrayList** de **shapes** est créé et les objets **Shape** sont ajoutés. Puis, toutes les valeurs **static** de couleur (**color**) sont établies à **GREEN**, et tout est sérialisé vers le fichier **CADState.out**.

Si l'on fournit un argument en ligne de commande (vraisemblablement **CADState.out**), ce fichier est ouvert et utilisé pour restituer l'état du programme. Dans les deux situations, l'**ArrayList** de **Shape**s est affichée. Le résultat d'une exécution donne :

```
>java CADState
[class Circle color[3] xPos[-51] yPos[-99] dim[38]
, class Square color[3] xPos[2] yPos[61] dim[-46]
, class Line color[3] xPos[51] yPos[73] dim[64]
```

```
, class Circle color[3] xPos[-70] yPos[1] dim[16]
, class Square color[3] xPos[3] yPos[94] dim[-36]
, class Line color[3] xPos[-84] yPos[-21] dim[-35]
, class Circle color[3] xPos[-75] yPos[-43] dim[22]
, class Square color[3] xPos[81] yPos[30] dim[-45]
, class Line color[3] xPos[-29] yPos[92] dim[17]
, class Circle color[3] xPos[17] yPos[90] dim[-76]
]

>java CADState CADState.out
[class Circle color[1] xPos[-51] yPos[-99] dim[38]
, class Square color[0] xPos[2] yPos[61] dim[-46]
, class Line color[3] xPos[51] yPos[73] dim[64]
, class Circle color[1] xPos[-70] yPos[1] dim[16]
, class Square color[0] xPos[3] yPos[94] dim[-36]
, class Line color[3] xPos[-84] yPos[-21] dim[-35]
, class Circle color[1] xPos[-75] yPos[-43] dim[22]
, class Square color[0] xPos[81] yPos[30] dim[-45]
, class Line color[3] xPos[-29] yPos[92] dim[17]
, class Circle color[1] xPos[17] yPos[90] dim[-76]
]
```

Vous pouvez voir que les valeurs d'**xPos**, **yPos**, et **dim** sont toutes stockées et récupérées avec succès, mais qu'il y a quelque chose d'anormal dans la récupération d'informations **static**. Les « 3 » rentrent bien, mais ne sortent pas de la même manière. Les **Circle**s ont une valeur de 1 (**RED**, ce qui est la définition), et les **Square**s ont une valeur de 0 (rappelez-vous, ils sont initialisés dans le constructeur). C'est comme si les **static**s n'étaient pas sérialisées du tout ! Ce qui est correct — malgré que la class **Class** est **Serializable**, cela n'agit pas comme on pouvait l'espérer. Donc si vous désirez sérialiser des **static**s, vous devrez le faire vous-même.

C'est à quoi les méthodes **static serializeStaticState()** et **deserializeStaticState()** dans **Line** servent. Vous pouvez voir qu'elles sont appelés explicitement comme une partie du processus de stockage et de récupération. (Notez que l'ordre d'écriture vers le fichier sérialisé et de sa relecture doit être conservé.) Ainsi pour que **CADState.java** s'exécute correctement vous devez :

1. Ajouter un **serializeStaticState()** et un **deserializeStaticState()** aux figures [shapes].
2. Enlever **ArrayList shapeTypes** et tout le code s'y rattachant.
3. Ajouter des appels aux nouvelles méthodes statiques de sérialisation et de sérialisation dans les figures [shapes].

Un autre sujet auquel vous devez penser est la sécurité, vu que la sérialisation sauve aussi les données **private**. Si vous vous avez orientation de sécurité, ces champs doivent être marqués comme **transient**. Mais alors vous devez concevoir une manière sûre pour stocker cette information de sorte que quand vous faites une restauration vous pouvez remettre à l'état initial [reset] ces variables **privates**.

Tokenizer l'entrée

Tokenizing est le processus de casser une séquence de caractères en un séquence de « to-

kens, » qui sont des morceaux de texte délimités par quoi que vous vous choisissez. Par exemple, vos jetons [tokens] peuvent être des mots, et ensuite ils pourront être délimités par un blanc et de la ponctuation. Il y a deux classes fournies dans la librairie standard de Java qui peuvent être employées pour la tokenisation : **StreamTokenizer** and **StringTokenizer**.

StreamTokenizer

Bien que **StreamTokenizer** ne soit pas dérivé d'**InputStream** ou **OutputStream**, il ne fonctionne qu'avec les objets InputStreams, donc il appartient à juste titre à la partie d'E/S de la librairie.

Considérons un programme qui compte les occurrences de mots dans un fichier texte :

```
//: c11:WordCount.java
// Compte les mots dans un fichier, produit
// les résultats dans un formulaire classé.
import java.io.*;
import java.util.*;

class Counter {
  private int i = 1;
  int read() { return i; }
  void increment() { i++; }
}

public class WordCount {
  private FileReader file;
  private StreamTokenizer st;
  // Un TreeMap conserve les clés dans un ordre classé :
  private TreeMap counts = new TreeMap();
  WordCount(String filename)
    throws FileNotFoundException {
    try {
      file = new FileReader(filename);
      st = new StreamTokenizer(
        new BufferedReader(file));
      st.ordinaryChar('.');
      st.ordinaryChar('-');
    } catch(FileNotFoundException e) {
      System.err.println(
        "Could not open " + filename);
      throw e;
    }
  }
  void cleanup() {
    try {
      file.close();
    } catch(IOException e) {
      System.err.println(
        "file.close() unsuccessful");
```

```
    }
  }
  void countWords() {
    try {
      while(st.nextToken() !=       StreamTokenizer.TT_EOF) {
        String s;
        switch(st.ttype) {
          case StreamTokenizer.TT_EOL:
            s = new String("EOL");
            break;
          case StreamTokenizer.TT_NUMBER:
            s = Double.toString(st.nval);
            break;
          case StreamTokenizer.TT_WORD:
            s = st.sval; // Déjà un String
            break;
          default: // un seul caractère dans type
            s = String.valueOf((char)st.ttype);
        }
        if(counts.containsKey(s))
          ((Counter)counts.get(s)).increment();
        else
          counts.put(s, new Counter());
      }
    } catch(IOException e) {
      System.err.println(
        "st.nextToken() unsuccessful");
    }
  }
  Collection values() {
    return counts.values();
  }
  Set keySet() { return counts.keySet(); }
  Counter getCounter(String s) {
    return (Counter)counts.get(s);
  }
  public static void main(String[] args)
  throws FileNotFoundException {
    WordCount wc =      new WordCount(args[0]);
    wc.countWords();
    Iterator keys = wc.keySet().iterator();
    while(keys.hasNext()) {
      String key = (String)keys.next();
      System.out.println(key + ": "
              + wc.getCounter(key).read());
    }
    wc.cleanup();
```

```
  }
} ///:~
```

Présenter les mots sous une forme classée est facile à faire en stockant les données dans un TreeMap, qui organise automatiquement ces clés dans un ordre classé (voir le Chapitre 9). Quand vous avez un jeu de clés en utilisant **keySet()** , elles seront automatiquement rangées dans l'ordre.

Pour ouvrir le fichier, un **FileReader** est utilisé, et pour changer le fichier en mots un **StreamTokenizer** est créé depuis le **FileReader** enveloppé dans un **BufferedReader**. Dans **StreamTokenizer**, il y a une liste par défaut de séparateurs, et vous pouvez en ajouter d'autres avec un jeu de méthodes. Ici, **ordinaryChar()** est utilisé pour dire « Ce caractère n'a aucune signification pour que je m'y intéresse, » donc l'analyseur ne les incluras pas comme des parties de tous les mots qu'il va créer. Par exemple, dire **st.ordinaryChar('.')** veut dire que les points ne seront pas inclus comme des parties des mots qui seront analysés. Vous pourrez trouver plus d'information dans la documentation HTML du JDK à *<java.sun.com>*.

Dans **countWords()**, les tokens sont tirés un à un depuis le stream, et l'information **ttype** est utilisée pour déterminer ce qu'il faut faire avec chaque token, vu qu'un token peut être une fin de ligne, un nombre, une chaîne de caractère [string], ou un simple caractère.

Une fois qu'un token est trouvé, le **counts** de **TreeMap** est appelé pour voir si il contient déjà le token sous la forme d'une clé. Si c'est le cas, l'objet **Counter** correspondant est incrémenté pour indiquer qu'une autre instance de ce mots a été trouvée. Si ce n'est pas le cas, un nouveau **Counter** est créé — vu que le constructeur de **Counter** initialise sa valeur à un, ceci agit aussi pour compter le mot.

WordCount n'est pas un type de **TreeMap**, donc il n'est pas hérité. Il accompli un type de fonctionnalité spécifique, ainsi bien que les méthodes **keys()** et **values()** doivent être re-exposées, cela ne veut pas forcément dire que l'héritage doit être utilisé vu qu'un bon nombre de méthodes de **TreeMap** sont ici inappropriés. En plus, les autres méthodes comme **getCounter()**, qui prend le **Counter** pour un **String** particulier, et **sortedKeys()**, qui produit un **Iterator**, terminant le changement dans la forme d'interface de **WordCount**..

Dans **main()** vous pouvez voir l'emploi d'un WordCount pour ouvrir et compter les mots dans un fichier — cela ne prend que deux lignes de code. Puis un **Iterator** est extrait vers une liste triée de clés (mots), et est employé pour retirer chaque clé et **Count** associés. L'appel à **cleanup()** est nécessaire pour s'assurer que le fichier est fermé.

StringTokenizer

Bien qu'il ne fasse pas partie de la bibliothèque d'E/S, le **StringTokenizer** possède des fonctions assez similaires a **StreamTokenizer** comme il sera décrit ici.

Le **StringTokenizer** renvoie les tokens dans une chaîne de caractère un par un. Ces tokens sont des caractères consécutifs délimités par des tabulations, des espaces, et des nouvelles lignes. Ainsi, les tokens de la chaîne de caractère « Où est mon chat ? » sont « Où, » « est, » « mon, » « chat, » « ?. » Comme le **StreamTokenizer**, vous pouvez appeler le **StringTokenizer** pour casser l'entrée de n'importe quelle manière, mais avec **StringTokenizer** vous effectuez cela en passant un second argument au constructeur, qui est un **String** des délimiteurs que vous utiliserez. En général, si vous désirez plus de sophistication, employez un **StreamTokenizer**.

Vous appelez un objet **StringTokenizer** pour le prochain token dans la chaîne de caractère en

utilisant la méthode **nextToken()**, qui renvoie soit un token ou une chaîne de caractère vide pour indiquer qu'il ne reste pas de tokens.

Comme exemple, le programme suivant exécute une analyse limitée d'une phrase, cherchant des phrases clés pour indiquer si la joie ou la tristesse sont sous-entendues.

```
//: c11:AnalyzeSentence.java
// Cherche des séries particulières dans les phrases.
import java.util.*;

public class AnalyzeSentence {
  public static void main(String[] args) {
    analyze("I am happy about this");
    analyze("I am not happy about this");
    analyze("I am not! I am happy");
    analyze("I am sad about this");
    analyze("I am not sad about this");
    analyze("I am not! I am sad");
    analyze("Are you happy about this?");
    analyze("Are you sad about this?");
    analyze("It's you! I am happy");
    analyze("It's you! I am sad");
  }
  static StringTokenizer st;
  static void analyze(String s) {
    prt("\nnew sentence >> " + s);
    boolean sad = false;
    st = new StringTokenizer(s);
    while (st.hasMoreTokens()) {
      String token = next();
      // Cherche jusqu'à ce l'on trouve un des
      // deux tokens de départ :
      if(!token.equals("I") &&
         !token.equals("Are"))
        continue; // Haut de la boucle while
      if(token.equals("I")) {
        String tk2 = next();
        if(!tk2.equals("am")) // Doit être après I
          break; // Sortie de la boucle while
        else {
          String tk3 = next();
          if(tk3.equals("sad")) {
            sad = true;
            break; // Sortie de la boucle while
          }
          if (tk3.equals("not")) {
            String tk4 = next();
            if(tk4.equals("sad"))
              break; // Laisse sad faux
```

```
            if(tk4.equals("happy")) {
              sad = true;
              break;
            }
          }
        }
      }
      if(token.equals("Are")) {
        String tk2 = next();
        if(!tk2.equals("you"))
          break; // Doit être après Are
        String tk3 = next();
        if(tk3.equals("sad"))
          sad = true;
        break; // Sortie de la boucle while
      }
    }
    if(sad) prt("Sad detected");
  }
  static String next() {
    if(st.hasMoreTokens()) {
      String s = st.nextToken();
      prt(s);
      return s;
    }
    else
      return "";
  }
  static void prt(String s) {
    System.out.println(s);
  }
} ///:~
```

Pour que chaque chaîne de caractère soit analysée, une boucle **while** est entrée et les tokens sont poussés hors de la chaîne de caractères. Notez la première déclaration de **if**, qui dit de **continuer** (retourne au début de la boucle et recommence encore) si le token est ni un « I » ou un « Are. » Ceci signifie qu'il attrapera les tokens que si un « I » ou un « Are » est trouvé. Vous pourriez penser utiliser le == à la place de la méthode **equals()**, mais cela ne fonctionne pas correctement, comme == compare les références de valeur tandis qu'**equals()** compare les contenus.

La logique du reste de la méthode **analyze()** est que le pattern (motif) dont on recherche la présence est « I am sad, » « I am not happy, » or « Are you sad? » Sans la déclaration **break**, le code réalisant ça serait bien plus confus qu'il ne l'est. Vous devez être conscient qu'un parseur typique (ceci en est un exemple primitif) possède normalement une table de ces tokens et un morceau de code qui bouge de l'un à l'autre des statuts dans la table quand les nouveaux tokens sont lus.

Vous pourrez voir le **StringTokenizer** seulement comme un raccourci pour un type de **StreamTokenizer** simple et spécifique. Cependant, si vous avez un **String** que vous désirez tokenizer et que **StringTokenizer** est trop limité, tout ce que vous avez à faire est de le retourner dans un

flux avec **StringBufferInputStream** puis de l'utiliser pour créer un **StreamTokenizer** beaucoup plus puissant.

Vérifier le style de capitalization

Dans cette partie on observera un exemple un peut plus complet d'utilisation du Java E/S. Ce projet est directement utile puisqu'il accomplit une vérification de style pour être sur que la capitalisation se conforme au style Java comme décrit sur*java.sun.com/docs/codeconv/index.html*. Il ouvre chaque fichier **.java** dans le répertoire courant et extrait tous les noms de classe et d'identifiants, puis nous indique si affiche l'un d'entre eux ne correspond pas au style Java.

Afin que le programme s'exécute correctement, on devra d'abord construire un dépôt de noms de classes pour conserver tous les noms de classes dans la bibliothèque standard de Java. On réalise ceci en se déplaçant dans tous les sous-répertoires du code source de la bibliothèque standard Java et en exécutant **ClassScanner** dans chaque sous répertoire. Donnant comme arguments le nom du fichier dépositaire (utilisant le même chemin et nom chaque fois) et l'option -lign de commande **-a** pour indiquer que les noms de classes devront être ajoutés au dépôt.

Afin d'utiliser le programme pour vérifier votre code, indiquez lui le chemin et le nom du dépôt à employer. Il vérifiera toutes les classes et les identifiants du répertoire courant et vous vous dira lesquels ne suivent pas le style de capitalisation Java.

Vous devrez être conscient que ce programme n'est pas parfait ; il y a quelques fois ou il signalera ce qui lui semble être un problème mais en regardant le code vous verrez qu'il n'y a rien à changer. C'est quelque peut agaçant, mais c'est tellement plus simple que d'essayer de trouver tous ces cas en vérifiant de vos yeux votre code.

La classe **MultiStringMap** est un outil qui nous permet d'organiser un groupe de chaînes de caractères sur chaque entrée de clé. Il utilise un **HashMap** (cette fois avec héritage) avec la clé comme simple chaîne de caractère qui est organisée sur la value de l'**ArrayList**. La méthode **add()** vérifiesimplement si il y a déjà une clé dans le **HashMap**, si ce n'est pas le cas elle en ajoute une à cet endroit. La méthode **getArrayList()** produit une **ArrayList** pour une clé particulière, et **printValues()**, qui est essentiellement utile pour le deboguage, affiche toutes les valeurs **ArrayList** par **ArrayList** .

Pour avoir rester simple, les noms de classe des bibliothèques standard de Java sont toutes placées dans un objet **Properties** (de la bibliothèque standard de Java). Rappelons qu'un objet **Properties** est un **HashMap** qui ne garde que des objets**String** pour à la fois la clé et les valeurs d'entrée. Cependant, il peut être sauvé vers le disque et restauré depuis le disque par un appel de méthode, donc c'est l'idéal pour le dépôt des noms. Actuellement, on ne désire qu'une liste de noms, et un **HashMap** ne peut pas accepter une **null** valeur à la fois pour ces clés ou ces valeurs d'entrée.

Pour les classes et identifiants qui sont découvert pour les fichiers d'un certain répertoire, deux MultiStringMaps sont utilisés : **classMap** et **identMap**. Aussi, quand le programme démarre il charge le dépôt de nom de classe standard dans l'objet **Properties** appelé **classes**, et quand un nouveau nom de classe est découvert dans le répertoire local il est aussi ajouté à **classes** en plus de **classMap**. De cette manière, **classMap** peut être employé pour se déplacer à travers toutes les classes du répertoire local, et **classes** peut être utilisé pour voir si le token courant est un nom de classe (qui indique la définition d'un objet ou le début d'une méthode, donc saisit le token suivant — jusqu'au point virgule — et le place dans **identMap**).

Le constructeur par défaut pour **ClassScanner** crée une liste de noms de fichier, en utilisant

JavaFilter implémentation du **FilenameFilter**, montré à la fin du fichier. Ensuite il appelle **scanListing()** pour chaque nom de fichier.

Dans**scanListing()** le fichier de code source est ouvert et transformé en un **StreamTokenizer**. Dans la documentation, passer **true** de **slashStarComments()** à **slashSlashComments()** est supposé enlever ces commentaires, mais cela semble être quelque peu défectueux, car cela semble ne pas fonctionner. Au lieu de cela, ces lignes sont marquées en tant que commentaires et les commentaires sont extrait par une autre méthode. Pour cela, le « / » doit être capturé comme un caractère ordinaire plutôt que de laisser le **StreamTokenizer** l'absorber comme une partie de commentaire, et la méthode **ordinaryChar()** dit au **StreamTokenizer** de faire cela. C'est aussi vrai pour les points (« . »), vu que nous voulons que l'appel de la méthode soit séparé en deux en identifiants individuels. Pourtant, l'underscore (soulignement), qui est ordinairement traité par **StreamTokenizer** comme un caractère individuel, doit être laissé comme une partie des identifiants puisqu'il apparaît dans des valeurs **static final** comme **TT_EOF**, etc., employé dans tant de nombreux programmes. La méthode **wordChars()** prend une rangée de caractères que l'on désire ajouter à ceux qui sont laissé dans un token qui à) été analysé comme un mot. Finalement, lorsqu'on fait l'analyse pour une ligne de commentaire ou que l'on met de coté une ligne on veux savoir quand arrive la fin de ligne, c'est pourquoi en appelant **eolIsSignificant(true)** l'EOL (*End Of Line* : fin de ligne) apparaîtra plutôt que sera absorbé par le **StreamTokenizer**.

Le reste du **scanListing()** lit et réagit aux tokens jusqu'à la fin du fichier, annonçant quand **nextToken()** renvoie la valeur **final static StreamTokenizer.TT_EOF**.

Si le token est un « / » il est potentiellement un commentaire, donc **eatComments()** est appelé pour voir avec lui. La seule autre situation qui nous intéresse est quand il s'agit d'un mot, sur lequel il y a des cas spéciaux.

Si le mot est **class** ou **interface** alors le prochain token représente une classe ou une interface, et il est placé dans **classes** et **classMap**. Si le mot est **import** ou **package**, alors on n'a plus besoin du reste de la ligne. Tout le reste doit être un identifiant (auquel nous sommes intéressé) ou un mot-clé (qui ne nous intéresse pas, mais ils sont tous en minuscule de toutes manières donc cela n'abîmera rien de les placer dedans). Ceux-ci sont ajoutés à **identMap**.

La méthode **discardLine()** est un outil simple qui cherche la fin de ligne. Notons que caque fois que l'on trouve un nouveau token, l'on doit effectuer le contrôle de la fin de ligne.

La méthode **eatComments()** est appelée toutes les fois qu'un slash avant est rencontré dans la boucle d"analyse principale. Cependant, cela ne veut pas forcément dire qu'un commentaire ai été trouvé, donc le prochain token doit être extrait pour voir si c'est un autre slash avant (dans ce cas la ligne est rejetée) ou un astérisque. Mais si c'est n'est pas l'un d'entre eux, cela signifie que le token qui vient d'être sorti est nécessaire dans la boucle d'analyse principale ! Heureusement, la **pushBack()** méthode nous permet de « pousser en arrière » le token courant dans le flux d'entrée pour que quand la boucle d'analyse principale appelle **nextToken()** elle prendra celui que l'on viens tout juste de pousser en arrière.

Par commodité, la méthode produit un tableau de tous les noms dans le récipient **classes**. Cette méthode n'est pas utilisée dans le programme mais est utiles pour le déboguage.

Les deux prochaines méthodes sont celles dans lesquelles prend place la réelle vérification. Dans **checkClassNames()**, les noms des classe sont extraits depuis le **classMap** (qui, rappelons le, contient seulement les noms dans ce répertoire, organisés par nom de fichier de manière à ce que le nom de fichier puisse être imprimé avec le nom de classe errant). Ceci est réalisé en poussant chaque **ArrayList** associé et en allant plus loin que ça, en regardant pour voir si le premier caractère

est en minuscule. Si c'est le cas, le message approprié est imprimé.

Dans **checkIdentNames()**, une approche similaire est prise : chaque nom d'identifiant est extrait depuis **identMap**. Si le nom n'est pas dans la liste **classes**, il est supposé être un identifiant ou un mot-clé. Un cas spécial est vérifié : si la longueur de l'identifiant est trois et tout les caractères sont en majuscule, cette identifiant est ignoré puisqu'il est probablement une valeur **static final** comme **TT_EOF**. Bien sur, ce n'est pas un algorithme parfait, mais il assume que l'on avertira par la suite de tous les identifiants complétement-majuscules qui sont hors sujet.

Au lieu de reporter tout les identifiants qui commencent par un caractère majuscule, cette méthode garde trace de celles qui ont déjà été rapportées dans **ArrayList** appelées **reportSet()**. Ceci traite l'**ArrayList** comme un « jeu » qui vous signale lorsqu'un élément se trouve déjà dans le jeu. L'élément est produit en ajoutant au nom de fichier l'identifiant. Si l'élément n'est pas dans le jeu, il est ajouté puis un le rapport est effectué.

Le reste du listing est composé de **main()**, qui s'en occupe lui même en manipulant les arguments de ligne de commande et prenant en compte de l'endroit où l'on a construit le dépôt de noms de classe de la bibliothèque standard Java ou en vérifiant la validité du code que l'on a écrit. Dans les deux cas il fait un objet **ClassScanner.**

Si l'on construit ou utilisons un dépôt, on doit essayer d'ouvrir les dépôts existants. En créant un objet **File** et en testant son existence, on peut décider que l'on ouvre le fichier et **load()** la liste de **classes Properties** dans **ClassScanner**. (Les classes du dépôt s'ajoutent, ou plutôt recouvrent, les classes trouvées par le constructeur **ClassScanner**.) Si l'on fournit seulement un seul argument en ligne de commande cela signifie que l'on désire accomplir une vérification des noms de classes et d'identifiants, mais si l'on fournit deux arguments (le second étant un **« -a »**) on construit un dépôt de noms de classes. Dans ce cas, un fichier de sortie est ouvert et la méthode **Properties.save()** est utilisée pour écrire la liste dans un fichier, avec une chaîne de caractère fournissant l'information d'en tête du fichier.

Résumé

La bibliothèque Java de flux d'E/S satisfait les exigence de base : on peut faire la lecture et l'écriture avec la console, un fichier, un bloc de mémoire, ou même à travers l'Internet (comme on pourra le voir au Chapitre 15). Avec l'héritage, on peut créer de nouveaux types d'objets d'entrée et de sortie. Et l'on peut même ajouter une simple extension vers les types d'objets qu'un flux pourrait accepter en redéfinissant la méthode **toString()** qui est automatiquement appelée lorsque l'on passe un objet à une méthode qui attendait un **String** (« conversion automatique de type » limitée à Java).

Il y a des questions laissées sans réponses par la documentation et la conception de la bibliothèque de flux. Par exemple, il aurait été agréable de pouvoir dire que l'on désire lancer une exception si l'on essaye d'écrire par dessus un fichier en l'ouvrant pour la sortie – certains systèmes de programmation permettant de spécifier que l'on désire ouvrir un fichier de sortie, mais seulement si il n'existe pas déjà. En Java, il apparaît que l'on est supposé utiliser un objet **File** pour déterminer qu'un fichier existe, puisque si on l'ouvre comme un **FileOutputStream** ou un **FileWriter** il sera toujours écrasé.

La bibliothèque de flux d'E/S amène des sentiments mélangés ; elle fait plus de travail et est portable. Mais si vous ne comprenez pas encore le decorator pattern, la conception n'est pas intuitive, il y a donc des frais supplémentaires pour l'apprendre et l'enseigner. Elle est aussi incomplète : il n'y a pas de support pour le genre de format de sortie que presque chaque paquetage d'E/S d'autres langages supportent.

Cependant, dès que vous *pourrez* comprendre le decorator pattern et commencerez à utiliser la bibliothèque pour des situations demandant sa flexibilité, vous pourrez commencer a bénéficier de cette conception, à tel point que ce que ça vous coûtera en lignes de codes supplémentaires ne vous ennuiera pas beaucoup.

Si vous n'avez pas trouvé ce que vous cherchiez dans ce chapitre (qui a seulement été une introduction, et n'est pas censée être complète), vous pourrez trouver une couverture détaillé dans *Java E/S,* de Elliotte Rusty Harold (O'Reilly, 1999).

Exercices

Les solutions aux exercices choisis peuvent être trouvées dans le document électronique *The Thinking in Java Annotated Solution Guide*, disponible pour un faible coût sur *www.BruceEckel.com*.

1. Ouvrez un fichier de manière à pouvoir lire le fichier une ligne à la fois. Lisez chaque ligne comme un **String** et placez cet objet **String** dans un . Affichez toutes les lignes dans le **LinkedList** en ordre inverse.
2. Modifiez l'exercice 1 pour que le nom du fichier à lire soit entré comme argument de ligne de commande.
3. Modifiez l'exercice 2 pour ouvrir aussi un fichier texte dans lequel vous pourrez écrire. Écrivez les lignes dans l'**ArrayList**, avec les numéros de ligne (ne pas essayer d'utiliser la classe « LineNumber »), vers le fichier.
4. Modifiez l'exercice 2 pour imposer les majuscules à toutes les lignes de l'**ArrayList** et envoyer les résultats à **System.out**.
5. Modifiez l'exercice 2 pour qu'il prenne un argument de ligne de commande supplémentaire des mots à trouver dans le fichier. Afficher toutes les lignes dans lesquelles on trouve le mot.
6. Modifier **DirList.java** pour que le **FilenameFilter** ouvre en fait chaque fichier et reçoive le fichier sur la base de n'importe quel argument tiré de la ligne de commande existant dans ce fichier.
7. Créez une classe nommée **SortedDirList** avec un constructeur qui prend l'information de chemin de fichier et construit une liste triée du répertoire à partir des fichiers du répertoire. Créez deux méthodes **list()** surchargées avec un constructeur qui présenterons l'une ou l'autre la liste complète ou un sous-ensemble de la liste basée sur un argument. Ajoutez une méthode **size()** qui prendra un nom de fichier et présentera la taille de ce fichier.
8. Modifiez **WordCount.java** pour qu'il produise à la place un classement alphabétique, utilisant l'outil du Chapitre 9.
9. Modifiez **WordCount.java** pour qu'il utilise une classe contenant un **String** et une valeur de numérotation pour stocker chaque différent mot, et un **Set** de ces objets pour maintenir la liste de ces mots.
10. Modifiez **IOStreamDemo.java** afin qu'il utilise **LineNumberInputStream** pour garder trace du compte de ligne. Notez qu'il est plus facile de garder seulement les traces de manière programmatique.
11. En commençant avec la partie 4 de **IOStreamDemo.java**, écrivez un programme qui

compare les performances à l'écriture d'un fichier en utilisant une E/S mise en mémoire tampon et l'autre non.

12. Modifiez la partie 5 d'**IOStreamDemo.java** pour éliminer les espaces dans la ligne produite par le premier appel à **in5br.readLine()**. Faites cela en employant une boucle **while** et **readChar()**.

13. Réparez le programme **CADState.java** comme décrit dans le texte.

14. Dans **Blips.java**, copiez le fichier et renommez le en **BlipCheck.java** et renommez la classe **Blip2** en **BlipCheck** (rendez la **public** et enlevez la portée publique de la classe **Blips** dans le processus). Enlevez les marques //**!** dans le fichier et lancez le programme incluant les mauvaises lignes. Ensuite, enlevez les commentaires du constructeur par défaut de **BlipCheck**. Lancez le et expliquez pourquoi il fonctionne. Notez qu'après la compilation, vous devrez exécuter le programme avec **« java Blips »** parce que la méthode main() est encore dans **Blips**.

15. Dans **Blip3.java**, enlevez les commentaires des deux lignes après les phrases « Vous devez faire ceci : » et lancez le programme. Expliquez le résultat et pourquoi il diffère de quand les deux lignes sont dans le programme.

16. (Intermédiaire) Dans le chapitre 8, repérez l'exemple **GreenhouseControls.java**, qui se compose de trois fichiers. Dans **GreenhouseControls.java**, la classe interne **Restart()** contient un jeu d'événements codé durement. Changez le programme pour qu'il lise les événements et leurs heures relatives depuis un fichier texte. (Défi : Utilisez une *méthode de fabrique* d'un design pattern pour construire les événements — voir *Thinking in Patterns with Java*, télechargeable à *www.BruceEckel.com.*)

[57] *Design Patterns*, Erich Gamma *et al.*, Addison-Wesley 1995.

[58] XML est un autre moyen de résoudre le problème de déplacer les donnée entre différentes plates-formes informatiques, et ne dépend pas du fait d'avoir Java sur toutes les plate-formes. Cependant, des outils Java existent qui supportent XML.

[59] Le chapitre 13 montre une solution même plus commode pour cela : un programme GUI avec une zone de texte avec ascenseur.

Chapitre 12 - Identification dynamique de type

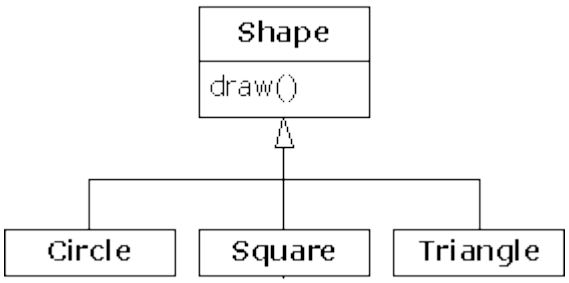
Le principe de l'identification dynamique de type (*Run-Time Type Identification, RTTI*) semble très simple à première vue : connaître le type exact d'un objet à partir d'une simple référence sur un type de base.

Néanmoins, le besoin de RTTI dévoile une pléthore de problèmes intéressants (et souvent complexes) en conception orientée objet, et renforce la question fondamentale de comment structurer ses programmes.

Ce chapitre indique de quelle manière Java permet de découvrir dynamiquement des informations sur les objets et les classes. On le retrouve sous deux formes : le RTTI « classique », qui suppose que tous les types sont disponibles à la compilation et à l'exécution, et le mécanisme de « réflexion », qui permet de découvrir des informations sur les classes uniquement à l'exécution. Le RTTI « classique » sera traité en premier, suivi par une discussion sur la réflexion.

Le besoin de RTTI

Revenons à notre exemple d'une hiérarchie de classes utilisant le polymorphisme. Le type générique est la classe de base **Forme**, et les types spécifiques dérivés sont **Cercle**, **Carre** et **Triangle** :



C'est un diagramme de classe hiérarchique classique, avec la classe de base en haut et les classes dérivées qui en découlent. Le but usuel de la programmation orientée objet est de manipuler dans la majorité du code des références sur le type de base (**Forme**, ici), tel que si vous décidez de créer une nouvelle classe (**Rhomboïde**, dérivée de **Forme**, par exemple), ce code restera inchangé. Dans notre exemple, la méthode liée dynamiquement dans l'interface **Forme** est **draw()**, ceci dans le but que le programmeur appelle **draw()** à partir d'une référence sur un objet de type **Forme**. **draw()** est redéfinie dans toutes les classes dérivées, et parce que cette méthode est liée dynamiquement, le comportement attendu arrivera même si l'appel se fait à partir d'une référence générique sur **Forme**. C'est ce que l'on appelle le polymorphisme.

Ainsi, on peut créer un objet spécifique (**Cercle**, **Carre** ou **Triangle**), le transtyper à **Forme** (oubliant le type spécifique de l'objet), et utiliser une référence anonyme à **Forme** dans le reste du programme.

Pour avoir un bref aperçu du polymorphisme et du transtypage ascendant (*upcast*), vous pouvez coder l'exemple ci-dessous :

```
//: c12:Formes.java
import java.util.*;

class Forme {
  void draw() {
    System.out.println(this + ".draw()");
  }
}

class Cercle extends Forme {
  public String toString() { return "Cercle"; }
}

class Carre extends Forme {
  public String toString() { return "Carre"; }
}

class Triangle extends Forme {
  public String toString() { return "Triangle"; }
}

public class Formes {
  public static void main(String[] args) {
    ArrayList s = new ArrayList();
    s.add(new Cercle());
    s.add(new Carre());
    s.add(new Triangle());
    Iterator e = s.iterator();
    while(e.hasNext())
      ((Shape)e.next()).draw();
  }
} ///:~
```

La classe de base contient une méthode **draw()** qui utilise indirectement **toString()** pour afficher un identifiant de la classe en utilisant **this** en paramètre de **System.out.println()**. Si cette fonction rencontre un objet, elle appelle automatiquement la méthode **toString()** de cet objet pour en avoir une représentation sous forme de chaîne de caractères.

Chacune des classes dérivées redéfinit la méthode **toString()** (de la classe **Object**) pour que **draw()** affiche quelque chose de différent dans chaque cas. Dans **main()**, des types spécifiques de **Forme** sont créés et ajoutés dans un **ArrayList**. C'est à ce niveau que le transtypage ascendant intervient car un **ArrayList** contient uniquement des **Object**s. Comme tout en Java (à l'exception des types primitifs) est **Object**, un **ArrayList** peut aussi contenir des **Forme**s. Mais lors du transtypage en **Object**, il perd toutes les informations spécifiques des objets, par exemple que ce sont des **Forme**s. Pour le **ArrayList**, ce sont juste des **Object**s.

Lorsqu'on récupère ensuite un élément de l'**ArrayList** avec la méthode **next()**, les choses se corsent un peu. Comme un **ArrayList** contient uniquement des **Object**s, **next()** va naturellement renvoyer une référence sur un **Object**. Mais nous savons que c'est en réalité une référence sur une

Forme, et nous désirons envoyer des messages de **Forme** à cet objet. Donc un transtypage en **Forme** est nécessaire en utilisant le transtypage habituel « **(Forme)** ». C'est la forme la plus simple de RTTI, puisqu'en Java l'exactitude de tous les typages est vérifiée à l'exécution. C'est exactement ce que signifie RTTI : à l'exécution, le type de tout objet est connu.

Dans notre cas, le RTTI est seulement partiel : l'**Object** est transtypé en **Forme**, mais pas en **Cercle**, **Carre** ou **Triangle**. Ceci parce que la seule chose que nous *savons* à ce moment là est que l'**ArrayList** est rempli de **Forme**s. A la compilation, ceci est garanti uniquement par vos propres choix (ndt : le compilateur vous fait confiance), tandis qu'à l'exécution le transtypage est effectivement vérifié.

Maintenant le polymorphisme s'applique et la méthode exacte qui a été appelée pour une **Forme** est déterminée selon que la référence est de type **Cercle**, **Carre** ou **Triangle**. Et en général, c'est comme cela qu'il faut faire ; on veut que la plus grosse partie du code ignore autant que possible le type spécifique des objets, et manipule une représentation générale de cette famille d'objets (dans notre cas, **Forme**). Il en résulte un code plus facile à écrire, lire et maintenir, et vos conceptions seront plus faciles à implémenter, comprendre et modifier. Le polymorphisme est donc un but général en programmation orientée objet.

Mais que faire si vous avez un problème de programmation qui peut se résoudre facilement si vous connaissez le type exact de la référence générique que vous manipulez ? Par exemple, supposons que vous désiriez permettre à vos utilisateurs de colorier toutes les formes d'un type particulier en violet. De cette manière, ils peuvent retrouver tous les triangles à l'écran en les coloriant. C'est ce que fait le RTTI : on peut demander à une référence sur une **Forme** le type exact de l'objet référencé.

L'objet Class

Pour comprendre comment marche le RTTI en Java, il faut d'abord savoir comment est représentée l'information sur le type durant l'exécution. C'est le rôle d'un objet spécifique appelé *l'objet Class*, qui contient toutes les informations relative à la classe (on l'appelle parfois *meta-class*). En fait, l'objet **Class** est utilisé pour créer tous les objets « habituels » d'une classe.

Il y a un objet **Class** pour chacune des classes d'un programme. Ainsi, à chaque fois qu'une classe est écrite et compilée, un unique objet de type **Class** est aussi créé (et rangé, le plus souvent, dans un fichier **.class** du même nom). Durant l'exécution, lorsqu'un nouvel objet de cette classe doit être créé, la MachineVirtuelle Java (*Java Virtual Machine, JVM*) qui exécute le programme vérifie d'abord si l'objet **Class** associé est déjà chargé. Si non, la JVM le charge en cherchant un fichier **.class** du même nom. Ainsi un programme Java n'est pas totalement chargé en mémoire lorsqu'il démarre, contrairement à beaucoup de langages classiques.

Une fois que l'objet **Class** est en mémoire, il est utilisé pour créer tous les objets de ce type.

Si cela ne vous semble pas clair ou si vous ne le croyez pas, voici un programme pour le prouver :

```
//: c12:Confiseur.java
// Étude du fonctionnement du chargeur de classes.

class Bonbon {
  static {
    System.out.println("Charge Bonbon");
```

```
  }
}

class Gomme {
  static {
    System.out.println("Charge Gomme");
  }
}

class Biscuit {
  static {
    System.out.println("Charge Biscuit");
  }
}

public class Confiseur {
  public static void main(String[] args) {
    System.out.println("Début méthode main");
    new Bonbon();
    System.out.println("Après création Gomme");
    try {
      Class.forName("Gomme");
    } catch(ClassNotFoundException e) {
      e.printStackTrace(System.err);
    }
    System.out.println(
      "Après Class.forName(\"Gomme\")");
    new Biscuit();
    System.out.println("Après création Biscuit");
  }
} ///:~
```

Chacune des classes **Bonbon**, **Gomme** et **Biscuit** a une clause **static** qui est exécutée lorsque la classe est chargée la première fois. L'information qui est affichée vous permet de savoir quand cette classe est chargée. Dans la méthode **main()**, la création des objets est dispersée entre des opérations d'affichages pour faciliter la détection du moment du chargement.

Une ligne particulièrement intéressante est :

```
Class.forName("Gomme");
```

Cette méthode est une méthode **static** de **Class** (qui appartient à tous les objets **Class**). Un objet **Class** est comme tous les autres objets, il est donc possible d'obtenir sa référence et de la manipuler (c'est ce que fait le chargeur de classes). Un des moyens d'obtenir une référence sur un objet **Class** est la méthode **forName()**, qui prend en paramètre une chaîne de caractères contenant le nom (attention à l'orthographe et aux majuscules !) de la classe dont vous voulez la référence. Elle retourne une référence sur un objet **Class**.

Le résultat de ce programme pour une JVM est :

```
Début méthode main
Charge Bonbon
Après création Bonbon
Charge Gomme
Après Class.forName("Gomme")
Charge Biscuit
Après création Biscuit
```

On peut noter que chaque objet **Class** est chargé uniquement lorsque c'est nécessaire, et que l'initialisation **static** est effectuée au chargement de la classe.

Les littéraux Class

Java fournit une deuxième manière d'obtenir une référence sur un objet de type **Class**, en utilisant *le littéral class*. Dans le programme précédent, on aurait par exemple :

```
Gomme.class;
```

ce qui n'est pas seulement plus simple, mais aussi plus sûr puisque vérifié à la compilation. Comme elle ne nécessite pas d'appel à une méthode, elle est aussi plus efficace.

Les littéraux Class sont utilisables sur les classes habituelles ainsi que sur les interfaces, les tableaux et les types primitifs. De plus, il y a un attribut standard appelé **TYPE** qui existe pour chacune des classes englobant des types primitifs. L'attribut **TYPE** produit une référence à l'objet **Class** associé au type primitif, tel que :

... est équivalent à ...	
boolean.class	**Boolean.TYPE**
char.class	**Character.TYPE**
byte.class	**Byte.TYPE**
short.class	**Short.TYPE**
int.class	**Integer.TYPE**
long.class	**Long.TYPE**
float.class	**Float.TYPE**
double.class	**Double.TYPE**
void.class	**Void.TYPE**

Ma préférence va à l'utilisation des « **.class** » si possible, car cela est plus consistant avec les classes habituelles.

Vérifier avant de transtyper

Jusqu'à présent, nous avons vu différentes utilisations de RTTI dont :

1. Le transtypage classique ; i.e. « **(Forme)** », qui utilise RTTI pour être sûr que le transtypage est correct et lancer une **ClassCastException** si un mauvais transtypage est ef-

fectué.

2. L'objet **Class** qui représente le type d'un objet. L'objet **Class** peut être interrogé afin d'obtenir des informations utiles durant l'exécution.

En C++, le transtypage classique « **(Forme)** » *n'effectue pas* de RTTI. Il indique seulement au compilateur de traiter l'objet avec le nouveau type. En Java, qui effectue cette vérification de type, ce transtypage est souvent appelé “transtypage descendant sain”. La raison du terme « descendant » est liée à l'historique de la représentation des diagrammes de hiérarchie de classes. Si transtyper un **Cercle** en une **Forme** est un transtypage ascendant, alors transtyper une **Forme** en un **Cercle** est un transtypage descendant. Néanmoins, on sait que tout **Cercle** est aussi une **Forme**, et le compilateur nous laisse donc librement effectuer un transtypage descendant ; par contre toute **Forme** *n'est pas nécessairement* un **Cercle**, le compilateur ne permet donc pas de faire un transtypage descendant sans utiliser un transtypage explicite.

Il existe une troisième forme de RTTI en Java. C'est le mot clef **instanceof** qui vous indique si un objet est d'un type particulier. Il retourne un **boolean** afin d'être utilisé sous la forme d'une question, telle que :

```
if(x instanceof Chien)
  ((Chien)x).aboyer();
```

L'expression ci-dessus vérifie si un objet **x** appartient à la classe **Chien** *avant* de transtyper **x** en **Chien**. Il est important d'utiliser **instanceof** avant un transtypage descendant lorsque vous n'avez pas d'autres informations vous indiquant le type de l'objet, sinon vous risquez d'obtenir une **ClassCastException**.

Le plus souvent, vous rechercherez un type d'objets (les triangles à peindre en violet par exemple), mais vous pouvez aisément identifier tous les objets en utilisant **instanceof**. Supposons que vous ayez une famille de classes d'animaux de compagnie (**Pet**) :

```
//: c12:Pets.java
class Pet {}
class Chien extends Pet {}
class Carlin extends Chien {}
class Chat extends Pet {}
class Rongeur extends Pet {}
class Gerbil extends Rongeur {}
class Hamster extends Rongeur {}

class Counter { int i; } ///:~
```

La classe **Counter** est utilisée pour compter le nombre d'animaux de compagnie de chaque type. On peut le voir comme un objet **Integer** que l'on peut modifier.

En utilisant **instanceof**, tous les animaux peuvent être comptés :

```
//: c12:PetCount.java
// Utiliser instanceof.
import java.util.*;

public class PetCount {
```

```
  static String[] typenames = {
    "Pet", "Chien", "Carlin", "Chat",
    "Rongeur", "Gerbil", "Hamster",
  };
  // Les exceptions remontent jusqu'à la console :
  public static void main(String[] args)
  throws Exception {
    ArrayList pets = new ArrayList();
    try {
      Class[] petTypes = {
        Class.forName("Chien"),
        Class.forName("Carlin"),
        Class.forName("Chat"),
        Class.forName("Rongeur"),
        Class.forName("Gerbil"),
        Class.forName("Hamster"),
      };
      for(int i = 0; i < 15; i++)
        pets.add(
          petTypes[
            (int)(Math.random()*petTypes.length)]
            .newInstance());
    } catch(InstantiationException e) {
      System.err.println("Instantiation impossible");
      throw e;
    } catch(IllegalAccessException e) {
      System.err.println("Accès impossible");
      throw e;
    } catch(ClassNotFoundException e) {
      System.err.println("Classe non trouvée");
      throw e;
    }
    HashMap h = new HashMap();
    for(int i = 0; i < typenames.length; i++)
      h.put(typenames[i], new Counter());
    for(int i = 0; i < pets.size(); i++) {
      Object o = pets.get(i);
      if(o instanceof Pet)
        ((Counter)h.get("Pet")).i++;
      if(o instanceof Chien)
        ((Counter)h.get("Chien")).i++;
      if(o instanceof Carlin)
        ((Counter)h.get("Carlin")).i++;
      if(o instanceof Chat)
        ((Counter)h.get("Chat")).i++;
      if(o instanceof Rongeur)
        ((Counter)h.get("Rongeur")).i++;
```

```
      if(o instanceof Gerbil)
        ((Counter)h.get("Gerbil")).i++;
      if(o instanceof Hamster)
        ((Counter)h.get("Hamster")).i++;
    }
    for(int i = 0; i < pets.size(); i++)
      System.out.println(pets.get(i).getClass());
    for(int i = 0; i < typenames.length; i++)
      System.out.println(
        typenames[i] + " quantité : " +
        ((Counter)h.get(typenames[i])).i);
  }
} ///:~
```

Bien sûr, cet exemple est imaginaire - vous utiliseriez probablement un attribut de classe (**static**) pour chaque type que vous incrémenteriez dans le constructeur pour mettre à jour les compteurs. Vous feriez cela *si* vous avez accès au code source de ces classes et pouvez le modifier. Comme ce n'est pas toujours le cas, le RTTI est bien pratique.

Utiliser les littéraux de classe

Il est intéressant de voir comment l'exemple précédent **PetCount.java** peut être réécrit en utilisant les littéraux de classe. Le résultat est plus satisfaisant sur bien des points :

```
//: c12:PetCount2.java
// Utiliser les littéraux de classe.
import java.util.*;

public class PetCount2 {
  public static void main(String[] args)
  throws Exception {
    ArrayList pets = new ArrayList();
    Class[] petTypes = {
      // Littéraux de classe:
      Pet.class,
      Chien.class,
      Carlin.class,
      Chat.class,
      Rongeur.class,
      Gerbil.class,
      Hamster.class,
    };
    try {
      for(int i = 0; i < 15; i++) {
        // on ajoute 1 pour éliminer Pet.class:
        int rnd = 1 + (int)(
          Math.random() * (petTypes.length - 1));
        pets.add(
```

```
            petTypes[rnd].newInstance());
          }
        } catch(InstantiationException e) {
          System.err.println("Instantiation impossible");
          throw e;
        } catch(IllegalAccessException e) {
          System.err.println("Accès impossible");
          throw e;
        }
        HashMap h = new HashMap();
        for(int i = 0; i < petTypes.length; i++)
          h.put(petTypes[i].toString(),
            new Counter());
        for(int i = 0; i < pets.size(); i++) {
          Object o = pets.get(i);
          if(o instanceof Pet)
            ((Counter)h.get("class Pet")).i++;
          if(o instanceof Chien)
            ((Counter)h.get("class Chien")).i++;
          if(o instanceof Carlin)
            ((Counter)h.get("class Carlin")).i++;
          if(o instanceof Chat)
            ((Counter)h.get("class Chat")).i++;
          if(o instanceof Rongeur)
            ((Counter)h.get("class Rongeur")).i++;
          if(o instanceof Gerbil)
            ((Counter)h.get("class Gerbil")).i++;
          if(o instanceof Hamster)
            ((Counter)h.get("class Hamster")).i++;
        }
        for(int i = 0; i < pets.size(); i++)
          System.out.println(pets.get(i).getClass());
        Iterator keys = h.keySet().iterator();
        while(keys.hasNext()) {
          String nm = (String)keys.next();
          Counter cnt = (Counter)h.get(nm);
          System.out.println(
            nm.substring(nm.lastIndexOf('.') + 1) +
            " quantité: " + cnt.i);
        }
      }
    } ///:~
```

Ici, le tableau **typenames** a été enlevé, on préfère obtenir de l'objet **Class** les chaînes identifiant les types. Notons que ce système permet au besoin de différencier classes et interfaces.

On peut aussi remarquer que la création de **petTypes** ne nécessite pas l'utilisation d'un block **try** puisqu'il est évalué à la compilation et ne lancera donc aucune exception, contrairement à

Class.forName().

Quand les objets **Pet** sont créés dynamiquement, vous pouvez voir que le nombre aléatoire généré est compris entre un (ndt inclus) et **petTypes.length** (ndt exclus), donc ne peut pas prendre la valeur zéro. C'est parce que zéro réfère à **Pet.class**, et que nous supposons que créer un objet générique **Pet** n'est pas intéressant. Cependant, comme **Pet.class** fait partie de **petTypes**, le nombre total d'animaux familiers est compté.

Un instanceof dynamique

La méthode **isInstance** de **Class** fournit un moyen d'appeler dynamiquement l'opérateur **instanceof**. Ainsi, toutes ces ennuyeuses expressions **instanceof** peuvent être supprimées de l'exemple **PetCount** :

```
//: c12:PetCount3.java
// Utiliser isInstance().
import java.util.*;

public class PetCount3 {
  public static void main(String[] args)
  throws Exception {
    ArrayList pets = new ArrayList();
    Class[] petTypes = {
      Pet.class,
      Chien.class,
      Carlin.class,
      Chat.class,
      Rongeur.class,
      Gerbil.class,
      Hamster.class,
    };
    try {
      for(int i = 0; i < 15; i++) {
        // Ajoute 1 pour éliminer Pet.class:
        int rnd = 1 + (int)(
          Math.random() * (petTypes.length - 1));
        pets.add(
          petTypes[rnd].newInstance());
      }
    } catch(InstantiationException e) {
      System.err.println("Instantiation impossible");
      throw e;
    } catch(IllegalAccessException e) {
      System.err.println("Accès impossible");
      throw e;
    }
    HashMap h = new HashMap();
    for(int i = 0; i < petTypes.length; i++)
      h.put(petTypes[i].toString(),
```

```
        new Counter());
    for(int i = 0; i < pets.size(); i++) {
      Object o = pets.get(i);
      // Utiliser isInstance pour automatiser
      // l'utilisation des instanceof :
      // Ndt: Pourquoi ce ++j ????
      for (int j = 0; j < petTypes.length; ++j)
        if (petTypes[j].isInstance(o)) {
          String key = petTypes[j].toString();
          ((Counter)h.get(key)).i++;
        }
    }
    for(int i = 0; i < pets.size(); i++)
      System.out.println(pets.get(i).getClass());
    Iterator keys = h.keySet().iterator();
    while(keys.hasNext()) {
      String nm = (String)keys.next();
      Counter cnt = (Counter)h.get(nm);
      System.out.println(
        nm.substring(nm.lastIndexOf('.') + 1) +
        " quantity: " + cnt.i);
    }
  }
} ///:~
```

On peut noter que l'utilisation de la méthode **isInstance()** a permis d'éliminer les expressions instanceof. De plus, cela signifie que de nouveaux types d'animaux familiers peuvent être ajoutés simplement en modifiant le tableau **petTypes** ; le reste du programme reste inchangé (ce qui n'est pas le cas lorsqu'on utilise des **instanceof**).

instanceof vs. équivalence de classe

Lorsque vous demandez une information de type, il y a une différence importante entre l'utilisation d'une forme de **instanceof** (**instanceof** ou **isInstance()**, qui produisent des résultats équivalents) et la comparaison directe des objets **Class**. Voici un exemple qui illustre cette différence :

```
//: c12:FamilyVsExactType.java
// La différence entre instanceof et class

class Base {}
class Derived extends Base {}

public class FamilyVsExactType {
  static void test(Object x) {
    System.out.println("Teste x de type " +
      x.getClass());
    System.out.println("x instanceof Base " +
```

```
      (x instanceof Base));
    System.out.println("x instanceof Derived " +
      (x instanceof Derived));
    System.out.println("Base.isInstance(x) " +
      Base.class.isInstance(x));
    System.out.println("Derived.isInstance(x) " +
      Derived.class.isInstance(x));
    System.out.println(
      "x.getClass() == Base.class " +
      (x.getClass() == Base.class));
    System.out.println(
      "x.getClass() == Derived.class " +
      (x.getClass() == Derived.class));
    System.out.println(
      "x.getClass().equals(Base.class)) " +
      (x.getClass().equals(Base.class)));
    System.out.println(
      "x.getClass().equals(Derived.class)) " +
      (x.getClass().equals(Derived.class)));
  }
  public static void main(String[] args) {
    test(new Base());
    test(new Derived());
  }
} ///:~
```

La méthode **test()** effectue une vérification du type de son argument en utilisant les deux formes de **instanceof**. Elle récupère ensuite la référence sur l'objet **Class** et utilise == et **equals()** pour tester l'égalité entre les objets **Class**. Le résultat est le suivant :

```
Teste x de type class Base
x instanceof Base true
x instanceof Derived false
Base.isInstance(x) true
Derived.isInstance(x) false
x.getClass() == Base.class true
x.getClass() == Derived.class false
x.getClass().equals(Base.class)) true
x.getClass().equals(Derived.class)) false
Teste x de type class Derived
x instanceof Base true
x instanceof Derived true
Base.isInstance(x) true
Derived.isInstance(x) true
x.getClass() == Base.class false
x.getClass() == Derived.class true
x.getClass().equals(Base.class)) false
x.getClass().equals(Derived.class)) true
```

Il est rassurant de constater que **instanceof** et **isIntance()** produisent des résultats identiques, de même que **equals()** et ==. Mais les tests eux-mêmes aboutissent à des conclusions différentes. **instanceof** teste le concept de type et signifie « es-tu de cette classe, ou d'une classe dérivée ? ». Autrement, si on compare les objets **Class** en utilisant ==, il n'est plus question d'héritage - l'objet est de ce type ou non.

La syntaxe du RTTI

Java effectue son identification dynamique de type (RTTI) à l'aide de l'objet **Class**, même lors d'un transtypage. La classe **Class** dispose aussi de nombreuses autres manières d'être utilisée pour le RTTI.

Premièrement, il faut obtenir une référence sur l'objet **Class** approprié. Une manière de le faire, comme nous l'avons vu dans l'exemple précédent, est d'utiliser une chaîne de caractères et la méthode **Class.forName()**. C'est très pratique car il n'est pas nécessaire d'avoir un objet de ce type pour obtenir la référence sur l'objet **Class**. Néanmoins, si vous avez déjà un objet de ce type, vous pouvez retrouver la référence à l'objet **Class** en appelant une méthode qui appartient à la classe racine **Object** : **getClass()**. Elle retourne une référence sur l'objet **Class** représentant le type actuel de l'objet. **Class** a de nombreuses méthodes intéressantes, comme le montre l'exemple suivant :

```
//: c12:ToyTest.java
// Teste la classe Class.

interface HasBatteries {}
interface Waterproof {}
interface ShootsThings {}
class Toy {
  // Commenter le constructeur par
  // défault suivant pour obtenir
  // NoSuchMethodError depuis (*1*)
  Toy() {}
  Toy(int i) {}
}

class FancyToy extends Toy
    implements HasBatteries,
      Waterproof, ShootsThings {
  FancyToy() { super(1); }
}

public class ToyTest {
  public static void main(String[] args)
  throws Exception {
    Class c = null;
    try {
      c = Class.forName("FancyToy");
    } catch(ClassNotFoundException e) {
      System.err.println("Ne trouve pas FancyToy");
      throw e;
```

```
    }
    printInfo(c);
    Class[] faces = c.getInterfaces();
    for(int i = 0; i < faces.length; i++)
      printInfo(faces[i]);
    Class cy = c.getSuperclass();
    Object o = null;
    try {
      // Nécessite un constructeur par défaut :
      o = cy.newInstance(); // (*1*)
    } catch(InstantiationException e) {
      System.err.println("Instanciation impossible");
      throw e;
    } catch(IllegalAccessException e) {
      System.err.println("Accès impossible");
      throw e;
    }
    printInfo(o.getClass());
  }
  static void printInfo(Class cc) {
    System.out.println(
      "Class nom: " + cc.getName() +
      " est une interface ? [" +
      cc.isInterface() + "]");
  }
} ///:~
```

On peut voir que la classe **FancyToy** est assez compliquée, puisqu'elle hérite de **Toy** et implémente les interfaces **HasBatteries**, **Waterproof** et **ShootThings**. Dans **main()**, une référence de **Class** est créée et initialisée pour la classe **FancyToy** en utilisant **forName()** à l'intérieur du block **try** approprié.

La méthode **Class.getInterfaces()** retourne un tableau d'objets **Class** représentant les interfaces qui sont contenues dans l'objet en question.

Si vous avez un objet **Class**, vous pouvez aussi lui demander la classe dont il hérite directement en utilisant la méthode **getSuperclass()**. Celle-ci retourne, bien sûr, une référence de **Class** que vous pouvez interroger plus en détail. Cela signifie qu'à l'éxécution, vous pouvez découvrir la hiérarchie de classe complète d'un objet.

La méthode **newInstance()** de **Class** peut, au premier abord, ressembler à un autre moyen de **cloner()** un objet. Néanmoins, vous pouvez créer un nouvel objet avec **newInstance()** sans un objet existant, comme nous le voyons ici, car il n'y a pas d'objets **Toy** - seulement **cy** qui est une référence sur l'objet **Class** de **y**. C'est un moyen de construire un « constructeur virtuel », qui vous permet d'exprimer « je ne sais pas exactement de quel type vous êtes, mais créez-vous proprement ». Dans l'exemple ci-dessus, **cy** est seulement une référence sur **Class** sans aucune autre information à la compilation. Et lorsque vous créez une nouvelle instance, vous obtenez une référence sur un **Object**. Mais cette référence pointe sur un objet **Toy**. Bien entendu, avant de pouvoir envoyer d'autres messages que ceux acceptés par **Object**, vous devez l'examiner un peu plus et effectuer quelques transtypages. De plus, la classe de l'objet créé par **newInstance()** doit avoir un constructeur par dé-

faut. Dans la prochaine section, nous verrons comment créer dynamiquement des objets de classes utilisant n'importe quel constructeur, avec l'API de réflexion Java.

La dernière méthode dans le listing est **printInfo()**, qui prend en paramètre une référence sur **Class**, récupère son nom avec **getName()**, et détermine si c'est une interface avec **isInterface()**.

Le résultat de ce programme est :

```
Class nom: FancyToy est une interface ? [false]
Class nom: HasBatteries est une interface ? [true]
Class nom: Waterproof est une interface ? [true]
Class nom: ShootsThings est une interface ? [true]
Class nom: Toy est une interface ? [false]
```

Ainsi, avec l'objet **Class**, vous pouvez découvrir vraiment tout ce que vous voulez savoir sur un objet.

Réflexion : information de classe dynamique

Si vous ne connaissez pas le type précis d'un objet, le RTTI vous le dira. Néanmoins, il y a une limitation : le type doit être connu à la compilation afin que vous puissiez le détecter en utilisant le RTTI et faire quelque chose d'intéressant avec cette information. Autrement dit, le compilateur doit connaître toutes les classes que vous utilisez pour le RTTI.

Ceci peut ne pas paraître une grande limitation à première vue, mais supposons que l'on vous donne une référence sur un objet qui n'est pas dans l'espace de votre programme. En fait, la classe de l'objet n'est même pas disponible lors de la compilation. Par exemple, supposons que vous récupériez un paquet d'octets à partir d'un fichier sur disque ou via une connexion réseau et que l'on vous dise que ces octets représentent une classe. Puisque le compilateur ne peut pas connaître la classe lorsqu'il compile le code, comment pouvez vous utilisez cette classe ?

Dans un environnement de travail traditionnel cela peut sembler un scénario improbable. Mais dès que l'on se déplace dans un monde de la programmation plus vaste, il y a des cas importants dans lesquels cela arrive. Le premier est la programmation par composants, dans lequel vous construisez vos projets en utilisant le *Rapid Application Development* (RAD) dans un constructeur d'application. C'est une approche visuelle pour créer un programme (que vous voyez à l'écran comme un « formulaire » (*form*)) en déplaçant des icônes qui représentent des composants dans le formulaire. Ces composants sont alors configurés en fixant certaines de leurs valeurs. Cette configuration durant la conception nécessite que chacun des composants soit instanciable, qu'il dévoile une partie de lui-même et qu'il permette que ses valeurs soient lues et fixées. De plus, les composants qui gèrent des événements dans une GUI doivent dévoiler des informations à propos des méthodes appropriées pour que l'environnement RAD puisse aider le programmeur à redéfinir ces méthodes de gestion d'événements. La réflexion fournit le mécanisme pour détecter les méthodes disponibles et produire leurs noms. Java fournit une structure de programmation par composants au travers de JavaBeans (décrit dans le chapitre 13).

La classe **Class** (décrite précédemment dans ce chapitre) supporte le concept de *réflexion*, et une bibliothèque additionnelle, **java.lang.reflect**, contenant les classes **Field**, **Method**, et **Constructor** (chacune implémentant l'interface **Member**). Les objets de ce type sont créés dynamiquement par la JVM pour représenter les membres correspondants d'une classe inconnue. On peut alors utiliser les constructeurs pour créer de nouveaux objets, les méthodes **get()** et **set()** pour

lire et modifier les champs associés à des objets **Field**, et la méthode **invoke()** pour appeler une méthode associée à un objet **Method**. De plus, on peut utiliser les méthodes très pratiques **getFields()**, **getMethods()**, **getConstructors()**, etc. retournant un tableau représentant respectivement des champs, méthodes et constructeurs (pour en savoir plus, jetez un oeil à la documentation en ligne de la classe **Class**). Ainsi, l'information sur la classe d'objets inconnus peut être totalement déterminée dynamiquement, sans rien en savoir à la compilation.

Il est important de noter qu'il n'y a rien de magique dans la réflexion. Quand vous utilisez la réflexion pour interagir avec des objets de type inconnu, la JVM va simplement regarder l'objet et voir qu'il appartient à une classe particulière (comme une RTTI ordinaire) mais, avant toute autre chose, l'objet **Class** doit être chargé. Le fichier **.class** pour ce type particulier doit donc être disponible pour la JVM, soit localement sur la machine ou via le réseau. La vraie différence entre le RTTI et la réflexion est donc qu'avec le RTTI, le compilateur ouvre et examine le fichier .class à la compilation. Dit autrement, vous pouvez appeler toutes les méthodes d'un objet “normalement”. Avec la réflexion, le fichier **.class** n'est pas disponible à la compilation ; il est ouvert et examiné à l'exécution.

Un extracteur de méthodes de classe

Vous aurez rarement besoin d'utiliser directement les outils de réflexion ; ils sont utilisés pour supporter d'autres caractéristiques de Java, telles que la sérialisation (Chapitre 11), JavaBeans (Chapitre 13) et RMI (Chapitre 15). Néanmoins, il est quelquefois utile d'extraire dynamiquement des informations sur une classe. Un outil très utile est un extracteur de méthode de classe. Comme mentionné précédemment, chercher le code définissant une classe ou sa documentation en ligne montre uniquement les méthodes définies ou redéfinies dans cette définition de classe. Mais il peut y en avoir des douzaines d'autre qui proviennent des classes de base. Les retrouver est fastidieux et long [60]. Heureusement, la réflexion fournit un moyen d'écrire un outil simple qui va automatiquement montrer l'interface entière. Voici comment il fonctionne :

```
//: c12:ShowMethods.java
// Utiliser la réflexion pour montrer toutes les méthodes
// d'une classe, même si celles ci sont définies dans la
// classe de base.
import java.lang.reflect.*;

public class ShowMethods {
  static final String usage =    "usage: \n" +
    "ShowMethods qualified.class.name\n" +
    "Pour montrer toutes les méthodes or: \n" +
    "ShowMethods qualified.class.name word\n" +
    "Pour rechercher les méthodes contenant 'word'";
  public static void main(String[] args) {
    if(args.length < 1) {
      System.out.println(usage);
      System.exit(0);
    }
    try {
      Class c = Class.forName(args[0]);
      Method[] m = c.getMethods();
```

```
        Constructor[] ctor = c.getConstructors();
        if(args.length == 1) {
          for (int i = 0; i < m.length; i++)
            System.out.println(m[i]);
          for (int i = 0; i < ctor.length; i++)
            System.out.println(ctor[i]);
        } else {
          for (int i = 0; i < m.length; i++)
            if(m[i].toString()
               .indexOf(args[1])!= -1)
              System.out.println(m[i]);
          for (int i = 0; i < ctor.length; i++)
            if(ctor[i].toString()
               .indexOf(args[1])!= -1)
            System.out.println(ctor[i]);
        }
      } catch(ClassNotFoundException e) {
        System.err.println("Classe non trouvée : " + e);
      }
    }
  } ///:~
```

Les méthodes de **Class getMethods()** et **getConstructors()** retournent respectivement un tableau de **Method** et **Constructor**. Chacune de ces classes a de plus des méthodes pour obtenir les noms, arguments et valeur retournée des méthodes qu'elles représentent. Mais vous pouvez aussi utiliser simplement **toString()**, comme ici, pour produire une chaîne de caractères avec la signature complète de la méthode. Le reste du code sert juste pour l'extraction des informations de la ligne de commande, déterminer si une signature particulière correspond à votre chaîne cible (en utilisant **indexOf()**), et afficher le résultat.

Ceci montre la réflexion en action, puisque le résultat de **Class.forName()** ne peut pas être connu à la compilation, donc toutes les informations sur la signature des méthodes est extraite à l'exécution. Si vous étudiez la documentation en ligne sur la réflexion, vous verrez qu'il est possible de créér et d'appeler une méthode d'un objet qui était totalement inconnu lors de la compilation (nous verrons des exemples plus loin dans ce livre). Encore une fois, c'est quelque chose dont vous n'aurez peut être jamais besoin de faire vous même- le support est là pour le RMI et la programmation par JavaBeans- mais il est intéressant.

Une expérience intéressante est de lancer :

```
java ShowMethods ShowMethods
```

Ceci produit une liste qui inclut un constructeur par défaut **public**, bien que vous puissiez voir à partir du code source qu'aucun constructeur n'ait été défini. Le constructeur que vous voyez est celui qui est automatiquement généré par le compilateur. Si vous définissez maintenant **ShowMethods** comme une classe non **public** (par exemple, amie), le constructeur par défaut n'apparaît plus dans la liste. Le constructeur pas défaut généré a automatiquement le même accès que la classe.

L'affichage de **ShowMethods** est toujours un peu ennuyeuse. Par exemple, voici une portion de l'affichage produit en invoquant **java ShowMethods java.lang.String** :

```
public boolean
  java.lang.String.startsWith(java.lang.String,int)
public boolean
  java.lang.String.startsWith(java.lang.String)
public boolean
  java.lang.String.endsWith(java.lang.String)
```

Il serait préférable que les préfixes comme **java.lang** puissent être éliminés. La classe **StreamTokenizer** introduite dans le chapitre précédent peut nous aider à créer un outil résolvant ce problème :

```
//: com:bruceeckel:util:StripQualifiers.java
package com.bruceeckel.util;
import java.io.*;

public class StripQualifiers {
  private StreamTokenizer st;
  public StripQualifiers(String qualified) {
    st = new StreamTokenizer(
      new StringReader(qualified));
    st.ordinaryChar(' '); // garde les espaces
  }
  public String getNext() {
    String s = null;
    try {
      int token = st.nextToken();
      if(token != StreamTokenizer.TT_EOF) {
        switch(st.ttype) {
          case StreamTokenizer.TT_EOL:
            s = null;
            break;
          case StreamTokenizer.TT_NUMBER:
            s = Double.toString(st.nval);
            break;
          case StreamTokenizer.TT_WORD:
            s = new String(st.sval);
            break;
          default: //il y a un seul caractère dans ttype
            s = String.valueOf((char)st.ttype);
        }
      }
    } catch(IOException e) {
      System.err.println("Erreur recherche token");
    }
    return s;
  }
  public static String strip(String qualified) {
    StripQualifiers sq =
```

```
      new StripQualifiers(qualified);
    String s = "", si;
    while((si = sq.getNext()) != null) {
      int lastDot = si.lastIndexOf('.');
      if(lastDot != -1)
        si = si.substring(lastDot + 1);
      s += si;
    }
    return s;
  }
} ///:~
```

Pour faciliter sa réutilisation, cette classe est placée dans **com.bruceeckel.util**. Comme vous pouvez le voir, elle utilise la classe **StreamTokenizer** et la manipulation des **String** pour effectuer son travail.

La nouvelle version du programme utilise la classe ci-dessus pour clarifier le résultat :

```
//: c12:ShowMethodsClean.java
// ShowMethods avec élimination des préfixes
// pour faciliter la lecture du résultat.
import java.lang.reflect.*;
import com.bruceeckel.util.*;

public class ShowMethodsClean {
  static final String usage =    "usage: \n" +
    "ShowMethodsClean qualified.class.name\n" +
    "Pour montrer toutes les méthodes or: \n" +
    "ShowMethodsClean qualified.class.name word\n" +
    "Pour rechercher les méthodes contenant 'word'";

  public static void main(String[] args) {
    if(args.length < 1) {
      System.out.println(usage);
      System.exit(0);
    }
    try {
      Class c = Class.forName(args[0]);
      Method[] m = c.getMethods();
      Constructor[] ctor = c.getConstructors();
      // Conversion en un tableau de chaînes simplifiées :
      String[] n =
        new String[m.length + ctor.length];
      for(int i = 0; i < m.length; i++) {
        String s = m[i].toString();
        n[i] = StripQualifiers.strip(s);
      }
      for(int i = 0; i < ctor.length; i++) {
```

```
            String s = ctor[i].toString();
            n[i + m.length] =
              StripQualifiers.strip(s);
          }
          if(args.length == 1)
            for (int i = 0; i < n.length; i++)
              System.out.println(n[i]);
          else
                for (int i = 0; i < n.length; i++)
              if(n[i].indexOf(args[1])!= -1)
                System.out.println(n[i]);
        } catch(ClassNotFoundException e) {
          System.err.println("Classe non trouvée : " + e);
        }
      }
    } ///:~
```

La classe **ShowMethodsClean** est semblable à la classe **ShowMethods**, excepté qu'elle transforme les tableaux de **Method** et **Constructor** en un seul tableau de **String**. Chaque **String** est ensuite appliquée à **StripQualifiers.strip()** pour enlever les préfixes des méthodes.

Cet outil peut réellement vous faire gagner du temps lorsque vous programmez, quand vous ne vous souvenez pas si une classe a une méthode particulière et que vous ne voulez pas explorer toute sa hiérarchie dans la documentation en ligne, ou si vous ne savez pas si cette classe peut faire quelque chose avec, par exemple, un objet **Color**.

Le chapitre 13 contient une version graphique de ce programme (adapté pour extraire des informations sur les composants Swing) que vous pouvez laisser tourner pendant que vous écrivez votre code, pour des recherches rapides.

Résumé

L'identification dynamique de type (RTTI) permet de découvrir des informations de type à partir d'une référence sur une classe de base inconnue. [je n'arrive pas à traduire cette phrase: *Thus, it's ripe for misuse by the novice since it might make sense before polymorphic method calls do*. Prop1:Ainsi, il mûrit pour sa mauvaise utilisation par le novice puisque il pourrait être utilise de le faire avant un appel de méthode polymorphique. Prop2 (JQ): Malheureusement ces informations peuvent conduire le novice à négliger les concepts du polymorphisme, puisqu'elles sont plus faciles à appréhender.] Pour beaucoup de gens habitués à la programmation procédurale, il est difficile de ne pas organiser leurs programmes en ensembles d'expressions **switch**. Ils pourraient faire la même chose avec le RTTI et perdraient ainsi l'importante valeur du polymorphisme dans le développement et la maintenance du code. L'intention de Java est de vous faire utiliser des appels de méthodes polymorphiques dans votre code, et de vous faire utiliser le RTTI uniquement lorsque c'est nécessaire.

Néanmoins, utiliser des appels de méthodes polymorphiques nécessite que vous ayez le contrôle de la définition des classes de base car il est possible que lors du développement de votre programme vous découvriez que la classe de base ne contient pas une méthode dans vous avez besoin. Si la classe de base provient d'une bibliothèque ou si elle est contrôlée par quelqu'un d'autre, une so-

lution à ce problème est le RTTI : vous pouvez créer un nouveau type héritant de cette classe auquel vous ajoutez la méthode manquante. Ailleurs dans le code, vous détectez ce type particulier et appelez cette méthode spécifique. Ceci ne détruit ni le polymorphisme ni l'extensibilité du programme car ajouter un nouveau type ne vous oblige pas à chasser les expressions switch dans votre programme. Cependant, lorsque vous ajoutez du code qui requiert cette nouvelle fonctionnalité dans votre programme principal, vous devez utiliser le RTTI pour détecter ce type particulier.

Mettre la nouvelle caractéristique dans la classe de base peut signifier que, pour le bénéfice d'une classe particulière, toutes les autres classes dérivées de cette classe de base devront contenir des bouts de code inutiles de la méthode. Cela rend l'interface moins claire et ennuie celui qui doit redéfinir des méthodes abstraites dérivant de la classe de base. Supposez que vous désiriez nettoyer les becs? [*spit valves*] de tous les instruments à vent de votre orchestre. Une solution est de mettre une méthode **nettoyerBec()** dans la classe de base **Instrument**, mais c'est ennuyeux car cela implique que les instruments **Electronique**s et à **Percussion** ont aussi un bec. Le RTTI fournit une solution plus élégante dans ce cas car vous pouvez placer la méthode dans une classe spécifique (**Vent** dans notre cas), où elle est appropriée. Néanmoins, une solution encore meilleure est de mettre une méthode **prepareInstrument()** dans la classe de base, mais il se peut que vous ne la trouviez pas la première fois que vous ayez à résoudre le problème, et croyiez à tort que l'utilisation du RTTI est nécessaire.

Enfin, le RTTI permettra parfois de résoudre des problèmes d'efficacité. Si votre code utilise le polymorphisme, mais qu'il s'avère que l'un de vos objets réagisse à ce code très général d'une manière particulièrement inefficace, vous pouvez reconnaître ce type en utilisant le RTTI et écrire un morceau de code spécifique pour améliorer son efficacité. Attention toutefois à programmer pour l'efficacité trop tôt. C'est un piège séduisant. Il est préférable d'avoir un programme qui marche d'abord, et décider ensuite s'il est assez rapide, et seulement à ce moment là vous attaquer aux problèmes de performances — avec un *profiler*.

Exercices

Les solutions de certains exercices se trouvent dans le document électronique *Guide des solutions annoté de Penser en Java*, disponible à prix réduit depuis *www.BruceEckel.com*.

1. Ajouter **Rhomboïde àFormes.java**. Créer un **Rhomboïde**, le transtyper en une **Forme**, ensuite le re-transtyper en un **Rhomboïde**. Essayer de le transtyper en un **Cercle** et voir ce qui se passe.

2. Modifier l'exercice 1 afin d'utiliser **instanceof** pour vérifier le type avant d'effectuer le transtypage descendant.

3. Modifier **Formes.java** afin que toutes les formes d'un type particulier puissent être mises en surbrillance (utiliser un drapeau). La méthode **toString()** pour chaque classe dérivée de **Forme** devra indiquer si cette **Forme** est mise en surbrillance ou pas.

4. Modifier **Confiseur.java** afin que la création de chaque type d'objet soit contrôlée par la ligne de commande. Par exemple, si la ligne de commande est "**java Confiseur Bonbon**", seul l'objet **Bonbon** sera créé. Noter comment vous pouvez contrôler quels objets **Class** sont chargés via la ligne de commande.

5. Ajouter un nouveau type de **Pet** à **PetCount3.java**. Vérifier qu'il est correctement créé et compté dans la méthode **main()**.

6. Écrire une méthode qui prend un objet en paramètre et affiche récursivement tous les

classes de sa hiérarchie.

7. Modifier l'exercice 6 afin d'utiliser **Class.getDeclaredFields()** pour afficher aussi les informations sur les champs de chaque classe.

8. Dans **ToyTest.java**, commenter le constructeur par défaut de **Toy** et expliquer ce qui arrive.

9. Ajouter une nouvelle **interface** dans **ToyTest.java** et vérifier qu'elle est correctement détectée et affichée.

10. Créer un nouveau type de conteneur qui utilise une **private ArrayList** pour stocker les objets. Déterminer le type du premier objet déposé, ne permettre ensuite à l'utilisateur que d'insérer des objets de ce type.

11. Écrire un programme qui détermine si un tableau de **char** est un type primitif ou réellement un objet.

12. Implanter **nettoyerBec( )** comme décrit dans le résumé.

13. Modifier l'exercice 6 afin d'utiliser la réflexion à la place du RTTI.

14. Modifier l'exercice 7 afin d'utiliser la réflexion à la place du RTTI.

15. Dans **ToyTest.java**, utiliser la réflexion pour créer un objet **Toy** en n'utilisant pas le constructeur par défaut.

16. Étudier l'interface **java.lang.Class** dans la documentation HTML de Java à *java.sun.com*. Écrire un programme qui prend en paramètre le nom d'une classe via la ligne de commande, et utilise les méthodes de **Class** pour extraire toutes les informations disponibles pour cette classe. Tester le programme sur une classe de la bibliothèque standard et sur une des vôtres.

[60] Spécialement dans le passé. Néanmoins, Sun a grandement amélioré la documentation HTML de Java et il est maintenant plus aisé de voir les méthodes des classes de base.

Chapitre 13 - Création de fenêtres & d'Applets

Une directive de conception fondamentale est « rendre les choses simples faciles, et les choses difficiles possibles ».

L'objectif initial de la conception de la bibliothèque d'interface utilisateur graphique *[graphical user interface (GUI)]* en Java 1.0 était de permettre au programmeur de construire une GUI qui a un aspect agréable sur toutes les plateformes. Ce but n'a pas été atteint. L'*Abstract Window Toolkit* (AWT) de Java 1.0 produit au contraire une GUI d'aspect aussi mé diocre sur tous les systèmes. De plus, elle est restrictive : on ne peut utiliser que quatre fontes, et on n'a accès à aucun des éléments de GUI sophistiqués disponibles dans son système d'exploitation.Le modèle de programmation de l'AWT Java 1.0 est aussi maladroit et non orienté objet. Un étudiant d'un de mes séminaires (qui était chez Sun lors de la création de Java) m'a expliqué pourquoi : l'AWT original avait été imaginé, conçu, et implémenté en un mois. Certainement une merveille de productivité, mais aussi un exemple du fait que la conception est importante.

La situation s'est améliorée avec le modèle d'événements de l'AWT de Java 1.1, qui a une approche beaucoup plus claire et orientée objet, avec également l'ajout des JavaBeans, un modèle de programmation par composants qui est tourné vers la création facile d'environnements de programmation visuels. Java 2 termine la transformation depuis l'ancien AWT de Java 1.0 en remplaçant à peu près tout par les *Java Foundation Classes* (JFC), dont la partie GUI est appelée « Swing ». Il s'agit d'un ensemble riche de JavaBeans faciles à utiliser et faciles à comprendre, qui peuvent être glissés et déposés (ou programmés manuellement) pour créer une GUI qui vous donnera (enfin) satisfaction. La règle de la « version 3 » de l'industrie du logiciel (un produit n'est bon qu'à partir de la version 3) semble également se vérifier pour les langages de programmation.

Ce chapitre ne couvre que la bibliothèque moderne Swing de Java 2, et fait l'hypothèse raisonnable que Swing est la bibliothèque finale pour les GUI Java. Si pour une quelconque raison vous devez utiliser le « vieux » AWT d'origine (parce que vous maintenez du code ancien ou que vous avez des limitations dans votre browser), vous pouvez trouver cette présentation dans la première édition de ce livre, téléchargeable à *www.BruceEckel.com* (également incluse sur le CD ROM fourni avec ce livre).

Dès le début de ce chapitre, vous verrez combien les choses sont différentes selon que vous voulez créer une applet ou une application normale utilisant Swing, et comment créer des programmes qui sont à la fois des applets et des applications, de sorte qu'ils puissent être exécutés soit dans un browser soit depuis la ligne de commande. Presque tous les exemples de GUI seront exécutables aussi bien comme des applets que comme des applications.

Soyez conscients que ceci n'est pas un glossaire complet de tous les composants Swing, ou de toutes les méthodes pour les classes décrites. Ce qui se trouve ici a pour but d'être simple. La bibliothèque Swing est vaste, et le but de ce chapitre est uniquement de vous permettre de démarrer avec les notions essentielles et d'être à l'aise avec les concepts. Si vous avez besoin de plus, alors Swing peut probablement vous donner ce que vous voulez si vous avez la volonté de faire des recherches.

Je suppose ici que vous avez téléchargé et installé les documents (gratuits) de la bibliothèque Java au format HTML depuis *java.sun.com* et que vous parcourrez les classes **javax.swing** dans cette documentation pour voir tous les détails et toutes les méthodes de la bibliothèque Swing.

Grâce à la simplicité de la conception de Swing, ceci sera souvent suffisant pour régler votre problème. Il y a de nombreux livres (assez épais) dédiés uniquement à Swing et vous vous y réfèrerez si vous avez besoin d'approfondir, ou si vous voulez modifier le comportement par défaut de Swing.

En apprenant Swing vous découvrirez que :

1. Swing est un bien meilleur modèle de programmation que ce que vous avez probablement vu dans d'autres langages et environnements de développement. Les JavaBeans (qui seront présentées vers la fin de ce chapitre) constituent l'ossature de cette bibliothèque.

2. Les « GUI builders »(environnements de programmation visuels) sont un *must-have* d'un environnement de développement Java complet. Les JavaBeans et Swing permettent au « builder » d'écrire le code pour vous lorsque vous placez les composants sur des formulaires à l'aide d'outils graphiques. Ceci permet non seulement d'accélérer le développement lors de la création de GUI, mais aussi d'expérimenter plus et de ce fait d'essayer plus de modèles et probablement d'en obtenir un meilleur.

3. La simplicité et la bonne conception de Swing signifie que même si vous utilisez un GUI builder plutôt que du codage manuel, le code résultant sera encore compréhensible ; ceci résout un gros problème qu'avaient les GUI builders, qui généraient souvent du code illisible.

Swing contient tous les composants attendus dans une interface utilisateur moderne, depuis des boutons contenant des images jusqu'à des arborescences et des tables. C'est une grosse bibliothèque mais elle est conçue pour avoir la complexité appropriée pour la tâche à effectuer ; si quelque chose est simple, vous n'avez pas besoin d'écrire beaucoup de code, mais si vous essayez de fairedes choses plus compliquées, votre code devient proportionnellemnt plus complexe. Ceci signifie qu'il y a un point d'entrée facile, mais que vous avez la puissance à votre disposition si vous en avez besoin.

Ce que vous aimerez dans Swing est ce qu'on pourrait appeler l' « orthogonalité d'utilisation ». C'est à dire, une fois que vous avez compris les idées générales de la bibliothèque, vous pouvez les appliquer partout. Principalement grâce aux conventions de nommage standards, en écrivant ces exemples je pouvais la plupart du temps deviner le nom des méthodes et trouver juste du premier coup, sans rechercher quoi que ce soit. C'est certainement la marque d'une bonne conception de la bibliothèque. De plus, on peut en général connecter des composants entre eux et les choses fonctionnent correctement.

Pour des question de rapidité, tous les composants sont « légers » , et Swing est écrit entièrement en Java pour la portabilité.

La navigation au clavier est automatique ; vous pouvez exécuter une application Swing sans utiliser la souris, et ceci ne réclame aucune programmation supplémentaire. Le scrolling se fait sans effort, vous emballez simplement votre composant dans un **JScrollPane** lorque vous l'ajoutez à votre formulaire. Des fonctionnalités telles que les « infobulles » [tool tips] ne demandent qu'une seule ligne de code pour les utiliser.

Swing possède également une fonctionnalité assez avancée, appelée le « pluggable look and feel », qui signifie que l'apparence de l'interface utilisateur peut être modifiée dynamiquement pour s'adapter aux habitudes des utilisateurs travaillant sur des plateformes et systèmes d'exploitation différents. Il est même possible (quoique difficile) d'inventer son propre « look and feel ».

L'applet de base

Un des buts de la conception Java est de créer des *applets*, qui sont des petits programmes s'exécutant à l'intérieur d'un browser Web. Parce qu'ils doivent être sûrs, les applets sont limitées dans ce qu'ils peuvent accomplir. Toutefois, les applets sont un outil puissant de programmation côté client, un problème majeur pour le Web.

Les restrictions des applets

Programmer dans un applet est tellement restrictif qu'on en parle souvent comme étant « dans le bac à sable », car il y a toujours quelqu'un (le système de sécurité Java lors de l'exécution *[Java run-time security system]*) qui vous surveille.

Cependant, on peut aussi sortir du bac à sable et écrire des applications normales plutôt que des applets ; dans ce cas on accède aux autres fonctionnalités de son OS. Nous avons écrit des applications tout au long de ce livre, mais il s'agissait d'*applications console*, sans aucun composant graphique. Swing peut aussi être utilisé pour construire des interfaces utilisateur graphiques pour des applications normales.

On peut en général répondre à la question de savoir ce qu'une applet peut faire en considérant ce qu'elle est *supposée* faire : étendre les fonctionnalités d'une page Web dans un browser. Comme en tant que surfeur sur le Net on ne sait jamais si une page Web vient d'un site amical ou pas, on veut que tout le code qu'il exécute soit sûr. De ce fait, les restrictions les plus importantes sont probablement :

1. *Une applet ne peut pas toucher au disque local.* Ceci signifie écrire *ou* lire, puisqu'on ne voudrait pas qu'une applet lise et transmette des informations privées sur Internet sans notre permission. Ecrire est interdit, bien sûr car ce serait une invitation ouverte aux virus. Java permet une *signature digitale* des applets. Beaucoup de restrictions des applets sont levées si on autorise des *applets de confiance [trusted applets]* (celles qui sont signées par une source dans laquelle on a confiance) à accéder à sa machine.

2. Les applets peuvent mettre du temps à s'afficher, car il faut les télécharger complètement à chaque fois, nécesitant un accès au serveur pour chacune des classes. Le browser peut mettre l'applet dans son cache, mais ce n'est pas garanti. Pour cette raison, on devrait toujours empaqueter ses applets dans un fichier JAR (Java ARchive) qui rassemble tous les composants de l'applet (y compris d'autres fichiers **.class** ainsi que les images et les sons) en un seul fichier compressé qui peut être téléchargé en une seule transaction avec le serveur. La « signature digitale » existe pour chacun des fichiers contenus dans le fichier JAR.

Les avantages d'une applet

Si on admet ces restrictions, les applets ont des avantages, en particulier pour la création d'applications client/serveur ou réseaux :

1. *Il n'y a pas de problème d'installation*. Une applet est réellement indépendante de la plateforme (y compris pour jouer des fichiers audio, etc.) et donc il n'est pas nécessaire de modifier le code en fonction des plateformes ni d'effectuer des « réglages » à l'installation. En fait, l'installation est automatique chaque fois qu'un utilisateur charge la page Web qui contient les applets, de sorte que les mises à jour se passent en silence et automatiquement. Dans les systèmes client/serveur traditionnels, créer et installer une nouvelle version du logi-

ciel client est souvent un cauchemar.

2. *On ne doit pas craindre un code défectueux affectant le système de l'utilisateur,* grâce au système de sécurité inclus au coeur du langage Java et de la structure de l'applet. Ceci, ainsi que le point précédent, rend Java populaire pour les applications *intranet* client/serveur qui n'existent qu'à l'intérieur d'une société ou dans une zone d'opérations réduite, où l'environnement utilisateur (le navigateur Web et ses extensions) peut être spécifié et/ou contrôlé.

Comme les applets s'intègrent automatiquement dans le code HTML, on dispose d'un système de documentation intégré et indépendant de la plateforme pour le support de l'applet. C'est une astuce intéressante, car on a plutôt l'habitude d'avoir la partie documentation du programme plutôt que l'inverse.

Les squelettes d'applications

Les bibliothèques sont souvent groupées selon leurs fonctionnalités. Certaines bibliothèques, par exemple, sont utilisées telles quelles. Les classes **String** et **ArrayList** de la bibliothèque standard Java en sont des exemples. D'autres bibliothèques sont conçues comme des briques de construction pour créer d'autres classes. Une certaine catégorie de bibliothèques est le *squelette d'applications [application framework],* dont le but est d'aider à la construction d'applications en fournissant une classe ou un ensemble de classes reproduisant le comportement de base dont vous avez besoin dans chaque application d'un type donné. Ensuite, pour particulariser le comportement pour ses besoins propres, on hérite de la classe et on redéfinit les méthodes qui nous intéressent. Un squelette d'application est un bon exemple de la règle « séparer ce qui change de ce qui ne change pas », car on essaie de localiser les parties spécifiques d'un programme dans les méthodes redéfinies [62].

les applets sont construites à l'aide d'un squelette d'application. On hérite de la classe **JApplet** et on redéfinit les méthodes adéquates. Quelques méthodes contrôlent la création et l'exécution d'une applet dans une page Web :

Method	Operation
init()	Appelée automatiquement pour effectuer la première initialisation de l'applet, y compris la disposition des composants. Il faut toujours redéfinir cette méthode.
start()	Appelée chaque fois que l'applet est rendue visible du navigateur Web, pour permettre à l'applet de démarrer ses opérations normales (en particuliers celles qui sont arrêtées par **stop()**). Appelée également après **init()**.
stop()	Appelée chaque fois que l'applet redevient invisible du navigateur Web, pour permettre à l'applet d'arrêter les opérations coûteuses. Appelée également juste avant **destroy()**.
destroy()	Appelée lorsque l'applet est déchargée de la page pour effectuer la libération finale des ressources lorsque l'applet n'est plus utilisée.

Avec ces informations nous sommes prêts à créer une applet simple :

```
//: c13:Applet1.java
// Très simple applet.
import javax.swing.*;
import java.awt.*;

public class Applet1 extends JApplet {
  public void init() {
  getContentPane().add(new JLabel("Applet!"));
}
} ///:~
```

Remarquons qu'il n'est pas obligatoire qu'une applet ait une méthode **main()**. C'est complètement câblé dans le squelette d'application ; on met tout le code de démarrage dans **init()**.

Dans ce programme, la seule activité consiste à mettre un label de texte sur l'applet, à l'aide de la classe **JLabel** (l'ancien AWT s'était approprié le nom **Label** ainsi que quelques autres noms de composants, de sorte qu'on verra souvent un « **J** » au début des composants Swing). Le constructeur de cette classe reçoit une **String** et l'utilise pour créer le label. Dans le programme ci-dessus ce label est placé dans le formulaire.

La méthode **init()** est responsable du placement de tous les composants du formulaire en utilisant la méthode **add()**. On pourrait penser qu'il devrait être suffisant d'appeler simplement add(), et c'est ainsi que cela se passait dans l'ancien AWT. Swing exige quant à lui qu'on ajoute tous les composants sur la « vitre de contenu » *[content pane]* d'un formulaire, et il faut donc appeler **getContentPane()** au cours de l'opération **add()**.

Exécuter des applets dans un navigateur Web

Pour exécuter ce programme, il faut le placer dans une page Web et visualiser cette page à l'aide d'un navigateur Web configuré pour exécuter des applets Java. Pour placer une applet dans une page Web, on place un tag spécial dans le source HTML de cette page [63] pour dire à la page comment charger et exécuter cette applet.

Ce système était très simple lorsque Java lui-même était simple et que tout le monde était dans le même bain et incorporait le même support de Java dans les navigateurs Web. Il suffisait alors d'un petit bout de code HTML dans une page Web, tel que :

```
<applet code=Applet1 width=100 height=50>
</applet>
```

Ensuite vint la guerre des navigateurs et des langages, et nous (programmeurs aussi bien qu'utilisateurs finaux) y avons perdu. Au bout d'un moment, JavaSoft s'est rendu compte qu'on ne pouvait plus supposer que les navigateurs supportaient la version correcte de Java, et que la seule solution était de fournir une sorte d'extension qui se conformerait au mécanisme d'extension des navigateurs. En utilisant ce mécanisme d'extensions (qu'un fournisseur de navigateur ne peut pas supprimer -dans l'espoir d'y gagner un avantage sur la concurrence- sans casser aussi toutes les autres extensions), JavaSoft garantit que Java ne pourra pas être supprimé d'un navigateur Web par un fournisseur qui y serait hostile.

Avec Internet Explorer, le mécanisme d'extensions est le contrôle ActiveX, et avec Netscape

c'est le plug-in. Dans le code HTML, il faut fournir les tags qui supportent les deux. Voici à quoi ressemble la page HTML la plus simple pour **Applet1** : [64]

```
//:! c13:Applet1.html
<html><head><title>Applet1</title></head><hr>
<OBJECT
classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
width="100" height="50"
align="baseline"  codebase="http://java.sun.com/products/plugin/1.2.2/jinstall-1_2_2-
win.cab#Version=1,2,2,0">
<PARAM NAME="code" VALUE="Applet1.class">
<PARAM NAME="codebase" VALUE=".">
<PARAM NAME="type" VALUE="application/x-java-applet;version=1.2.2">
<COMMENT>
<EMBED type=  "application/x-java-applet;version=1.2.2"
  width="200" height="200" align="baseline"
  code="Applet1.class" codebase="."
pluginspage="http://java.sun.com/products/plugin/1.2/plugin-install.html">
<NOEMBED>
</COMMENT>
No Java 2 support for APPLET!!
</NOEMBED>
</EMBED>
</OBJECT>
<hr></body></html>
///:~
```

Certaines de ces lignes étaient trop longues et ont été coupées pour être insérées dans cette page. Le code inclus dans les sources de ce livre (sur le CD ROM de ce livre, et téléchargeable sur *www.BruceEckel.com*) fonctionne sans qu'on doive s'occuper de corriger les sauts de lignes.

Le contenu de code fournit le nom du fichier **.class** dans lequel se trouve l'applet. Les paramètres **width** et **height** spécifient la taille initiale de l'applet (en pixels, comme avant). Il y a d'autres éléments qu'on peut placer dans le tag applet : un endroit où on peut trouver d'autres fichiers .class sur Internet (**codebase**), des informations d'alignement (**align**), un identificateur spécial permettant à des applets de communiquer entre eux (**name**), et des paramètres des applets pour fournir des informations que l'applet peut récupérer. Les paramètres sont de la forme :

```
<param name="identifier" value = "information">
```

et il peut y en avoir autant qu'on veut.

Le package de codes source de ce livre fournit une page HTML pour chacune des applets de ce livre, et de ce fait de nombreux exemples du tag applet. Vous pouvez trouver une description complète et à jour des détails concernant le placement d'applets dans des pages web à l'adresse *java.sun.com*.

Utilisation de Appletviewer

Le JDK de Sun (en téléchargement libre depuis *java.sun.com*) contient un outil appelé l'*Ap-*

pletviewer qui extrait le tag du fichier HTML et exécute les applets sans afficher le texte HTML autour. Comme l'Appletviewer ignore tout ce qui n'est pas tags APPLET, on peut mettre ces tags dans le source Java en commentaires :

```
// <applet code=MyApplet width=200 height=100>
// </applet>
```

De cette manière, on peut lancer « **appletviewer MyApplet.java** » et il n'est pas nécessaire de créer de petits fichiers HTML pour lancer des tests. Par exemple, on peut ajouter ces tags HTML en commentaires dans **Applet1.java**:

```
//: c13:Applet1b.java
// Embedding the applet tag for Appletviewer.
// <applet code=Applet1b width=100 height=50>
// </applet>
import javax.swing.*;
import java.awt.*;

public class Applet1b extends JApplet {>
  public void >init() {
  getContentPane().add(new JLabel("Applet!"));
}
} ///:~
```

Maintenant on peut invoquer l'applet avec la commande

```
appletviewer Applet1b.java
```

Tester les applets

On peut exécuter un test simple sans aucune connexion réseau en lançant son navigateur Web et en ouvrant le fichier HTML contenant le tag applet. Au chargement du fichier HTML, le navigateur découvre le tag applet et part à la recherche du fichier **.class** spécifié par le contenu de **code**. Bien sûr, il utilise le CLASSPATH pour savoir où chercher, et si le fichier **.class** n'est pas dans le CLASSPATH, il émettra un message d'erreur dans sa ligne de status pour signaler qu'il n'a pas pu trouver le fichier **.class.**

Quand on veut essayer ceci sur son site Web les choses sont un peu plus compliquées. Tout d'abord il faut *avoir* un site Web, ce qui pour la plupart des gens signifie avoir un Fournisseur d'Accès à Internet (FAI) *[Internet Service Provider (ISP)]*. Comme l'applet est simplement un fichier ou un ensemble de fichiers, le FAI n'a pas besoin de fournir un support particulier pour Java. Il faut disposer d'un moyen pour copier les fichiers HTML et les fichiers **.class** depuis chez vous vers le bon répertoire sur la machine du FAI. Ceci est normalement fait avec un programme de File Transfer Protocol (FTP), dont il existe beaucoup d'exemples disponibles gratuitement ou comme sharewares. Il semblerait donc que tout ce qu'il y a à faire est d'envoyer les fichiers sur la machine du FAI à l'aide de FTP, et ensuite de se connecter au site et au fichier HTML en utilisant son navigateur ; si l'applet se charge et fonctionne, alors tout va bien, n'est-ce pas ?

C'est là qu'on peut se faire avoir. Si le navigateur de la machine client ne peut pas localiser le fichier **.class** sur le serveur, il va le rechercher à l'aide du CLASSPATH sur la machine *locale*. De ce

fait l'applet pourrait bien ne pas se charger correctement depuis le serveur, mais tout paraît correct lors du test parce que le navigateur le trouve sur la machine locale. Cependant, lorsque quelqu'un d'autre se connecte, son navigateur ne la trouvera pas. Donc lorsque vous testez, assurez vous d'effacer les fichiers **.class** (ou **.jar**) de votre machine locale pour vérifier qu'ils existent au bon endroit sur le serveur.

Un des cas les plus insidieux qui me soit arrivé s'est produit lorsque j'ai innocemment placé une applet dans un package. Après avoir téléchargé sur le serveur le fichier HTML et l'applet, le serveur fut trompé sur le chemin d'accès à l'applet à cause du nom du package. Cependant, mon navigateur l'avait trouvé dans le CLASSPATH local. J'étais donc le seul à pouvoir charger correctement l'applet. J'ai mis un certain temps à découvrir que l'instruction **package** était la coupable. En général il vaut mieux ne pas utiliser l'instruction **package** dans une applet.

Exécuter des applets depuis la ligne de commande

Parfois on voudrait qu'un programme fenêtré fasse autre chose que se trouver dans une page Web. Peut-être voudrait-on aussi faire certaines des choses qu'une application « normale » peut faire, mais en gardant la glorieuse portabilité instantanée fournie par Java. Dans les chapitres précédents de ce livre, nous avons fait des applications de ligne de commande, mais dans certains environnements (le Macintosh par exemple) il n'y a pas de ligne de commande. Voilà un certain nombre de raisons pour vouloir construire un programme fenêtré n'étant pas une applet. C'est certainement un désir légitime.

La bibliothèque Swing nous permet de construire une application qui conserve le « look and feel » du système d'exploitation sous-jacent. Si vous voulez faire des applications fenêtrées, cela n'a de sens [65] que si vous pouvez utiliser la dernière version de Java et ses outils associés, afin de pouvoir fournir des applications qui ne perturberont pas vos utilisateurs. Si pour une raison ou une autre vous devez utiliser une ancienne version de Java, réfléchissez-y bien avant de vous lancer dans la construction d'une application fenêtrée importante.

On a souvent besoin de créer une classe qui peut être appelée aussi bien comme une fenêtre que comme une applet. C'est particulièrement utile lorsqu'on teste des applets, car il est souvent plus facile et plus simple de lancer l'applet-application depuis la ligne de commande que de lancer un navigateur Web ou l'Appletviewer.

Pour créer une applet qui peut être exécutée depuis la ligne de commande, il suffit d'ajouter un **main()** à l'applet, dans lequel on construit une instance de l'applet dans un **JFrame** [66]. En tant qu'exemple simple, observons **Applet1b.java** modifié pour fonctionner aussi bien en tant qu'application qu'en tant qu'applet :

```
//: c13:Applet1c.java
// Une application et une applet.
// <applet code=Applet1c width=100 height=50>
// </applet>
import javax.swing.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class Applet1c extends JApplet {
  public void init() {
```

```
  getContentPane().add(new JLabel("Applet!"));
 }
  // Un main() pour l'application :
  public static void main(String[] args) {
  JApplet applet = new Applet1c();
  JFrame frame = new JFrame("Applet1c");
   // Pour fermer l'application :
  Console.setupClosing(frame);
  frame.getContentPane().add(applet);
  frame.setSize(100,50);
  applet.init();
  applet.start();
  frame.setVisible(true);
 }
} ///:~
```

main() est le seul élément ajouté à l'applet, et le reste de l'applet n'est pas modifié. L'applet est créée et ajoutée à un JFrame pour pouvoir être affichée.

La ligne :

```
Console.setupClosing(frame);
```

permet la fermeture propre de la fenêtre. **Console** vient de **com.bruceeckel.swing** et sera expliqué un peu plus tard.

On peut voir que dans **main()**, l'applet est explicitement initialisée et démarrée, car dans ce cas le navigateur n'est pas là pour le faire. Bien sûr, ceci ne fournit pas le comportement complet du navigateur, qui appelle aussi **stop()** et **destroy()**, mais dans la plupart des cas c'est acceptable. Si cela pose un problème, on peut forcer les appels soi-même href="#fn67">[67].

Notez la dernière ligne :

```
frame.setVisible(true);
```

Sans elle, on ne verrait rien à l'écran.

Un squelette d'affichage

Bien que le code qui transforme des programmes en applets et applications produise des résultats corrects, il devient perturbant et gaspille du papier s'il est utilisé partout. Au lieu de cela, le squelette d'affichage ci-après sera utilisé pour les exemples Swing du reste de ce livre :

```
//: com:bruceeckel:swing:Console.java
// Outil pour exécuter des démos Swing depuis
// la console, aussi bien applets que JFrames.
package com.bruceeckel.swing;
import javax.swing.*;
import java.awt.event.*;

public class Console {
```

```
  // Crée une chaîne de titre à partir du nom de la classe :
  public static String title(Object o) {
  String t = o.getClass().toString();
   // Enlever le mot "class":
   if(t.indexOf("class") != -1)
    t = t.substring(6);
   return t;
 }
  public static void setupClosing(JFrame frame) {
   // La solution JDK 1.2 Solution avec une
   // classe interne anonyme :
  frame.addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e) {
      System.exit(0);
    }
  });
   // La solution amelioree en JDK 1.3 :
   // frame.setDefaultCloseOperation(
   //     EXIT_ON_CLOSE);
 }
  public static void
 run(JFrame frame, int width, int height) {
  setupClosing(frame);
  frame.setSize(width, height);
  frame.setVisible(true);
 }
  public static void
 run(JApplet applet, int width, int height) {
  JFrame frame = new JFrame(title(applet));
  setupClosing(frame);
  frame.getContentPane().add(applet);
  frame.setSize(width, height);
  applet.init();
  applet.start();
  frame.setVisible(true);
 }
  public static void
 run(JPanel panel, int width, int height) {
  JFrame frame = new >JFrame(title(panel));
  setupClosing(frame);
  frame.getContentPane().add(panel);
  frame.setSize(width, height);
  frame.setVisible(true);
 }
 } ///:~
```

Comme c'est un outil que vous pouvez utiliser vous-mêmes, il est placé dans la bibliothèque **com.bruceeckel.swing**. La classe **Console** contient uniquement des méthodes **static**. La première

est utilisée pour extraire le nom de la classe (en utilisant RTTI) depuis n'importe quel objet, et pour enlever le mot « class », qui est ajouté normalement au début du nom par **getClass()**. On utilise les méthodes de **String** : **indexOf()** pour déterminer si le mot « class » est présent, et **substring()** pour générer la nouvelle chaîne sans « class » ou le blanc de fin. Ce nom est utilisé pour étiqueter la fenêtre qui est affichée par les méthodes **run()**.

setupClosing() est utilisé pour cacher le code qui provoque la sortie du programme lorsque la JFrame est fermée. Le comportement par défaut est de ne rien faire, donc si on n'appelle pas**setupClosing()** ou un code équivalent pour le **JFrame**, l'application ne se ferme pas. Une des raisons pour laquelle ce code est caché plutôt que d'être placé directement dans les méthodes run() est que cela nous permet d'utiliser la méthode en elle-même lorsque ce qu'on veut faire est plus complexe que ce que fournit **run()** . Mais il isole aussi un facteur de changement : Java 2 possède deux manières de provoquer la fermeture de certains types de fenêtres. En JDK 1.2, la solution est de créer une nouvelle classe **WindowAdapter**et d'implémenter **windowClosing()**, comme vu plus haut (la signification de ceci sera expliquée en détails plus tard dans ce chapitre). Cependant lors de la création du JDK 1.3, les concepteurs de la librairie ont observé qu'on a normalement besoin de fermer des fenêtres chaque fois qu'on crée un programme qui n'est pas une applet, et ils ont ajouté**setDefaultCloseOperation()**à **JFrame** et **JDialog**. Du point de vue de l'écriture du code, la nouvelle méthode est plus agréable à utiliser, mais ce livre a été écrit alors qu'il n'y avait pas de JDK 1.3 implémenté sur Linux et d'autres plateformes, et donc dans l'intérêt de la portabilité toutes versions, la modification a été isolée dans **setupClosing()**.

La méthode **run()** est surchargée pour fonctionner avec les **JApplet**s, les **JPanel**s, et les **JFrame**s. Remarquez que **init()** et **start()** ne sont appelées que s'il s'agit d'une **JApplet**.

Maintenant toute applet peut être lancée de la console en créant un **main()** contenant une ligne comme celle-ci :

```
Console.run(new MyClass(), 500, 300);
```

dans laquelle les deux derniers arguments sont la largeur et la hauteur de l'affichage. Voici **Applet1c.java** modifié pour utiliser **Console** :

```
//: c13:Applet1d.java
// Console exécute des applets depuis la ligne de commande.
// <applet code=Applet1d width=100 height=50>
// </applet>
import javax.swing.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class Applet1d extends JApplet {
  public void init() {
  getContentPane().add(new JLabel("Applet!"));
}
  public static void main(String[] args) {
  Console.run(new Applet1d(), 100, 50);
}
} ///:~
```

Ceci permet l'élimination de code répétitif tout en fournissant la plus grande flexibilité pour

lancer les exemples.

Utilisation de l'Explorateur Windows

Si vous utilisez Windows, vous pouvez simplifier le lancement d'un programme Java en ligne de commande en configurant l'explorateur Windows (le navigateur de fichiers de Windows, *pas* Internet Explorer) de façon à pouvoir double-cliquer sur un fichier **.class** pour l'exécuter. Il y a plusieurs étapes à effectuer.

D'abord, téléchargez et installez le langage de programmation Perl depuis *www.Perl.org*. Vous trouverez sur ce site les instructions et la documentation sur ce langage.

Ensuite, créez le script suivant sans la première et la dernière lignes (ce script fait partie du package de sources de ce livre) :

```
//:! c13:RunJava.bat
@rem = '--*-Perl-*--
@echo off
perl -x -S "%0" %1 %2 %3 %4 %5 %6 %7 %8 %9
goto endofperl
@rem ';
#!perl
$file = $ARGV[0];
$file =~ s/(.*)\..*/\1/;
$file =~ s/(.*\\)*(.*)/$+/;
´java $file´;
__END__
:endofperl
///:~
```

Maintenant, ouvrez l'explorateur Windows, sélectionnez Affichage, Options des dossiers, et cliquez sur l'onglet "Types de fichiers". Cliquez sur le bouton "Nouveau type...". Comme "Description du type", entrez "fichier classe Java". Comme "Extension associée", entrez class. Sous "Actions", cliquez sur le bouton "Nouveau...". Comme "Action" entrez "open", et pour "Application utilisée pour effectuer l'action" entrez une ligne telle que celle-ci :

```
"c:\aaa\Perl\RunJava.bat" "%L"
```

en personnalisant le chemin devant RunJava.bat en fonction de l'endroit où vous avez placé le fichier batch.

Une fois cette installation effectuée, vous pouvez exécuter tout programme Java simplement en double-cliquant sur le fichier **.class** qui contient un **main()**.

Création d'un bouton

La création d'un bouton est assez simple: il suffit d'appeler le constructeur **JButton** avec le label désiré sur le bouton. On verra plus tard qu'on peut faire des choses encore plus jolies, comme par exemple y mettre des images graphiques.

En général on créera une variable pour le bouton dans la classe courante, afin de pouvoir s'y

référer plus tard.

Le **JButton** est un composant possédant sa propre petite fenêtre qui sera automatiquement repeinte lors d'une mise à jour. Ceci signifie qu'on ne peint pas explicitement un bouton ni d'ailleurs les autres types de contrôles; on les place simplement sur le formulaire et on les laisse se repeindre automatiquement. Le placement d'un bouton sur un formulaire se fait dans **init() :**

```
//: c13:Button1.java
// Placement de boutons sur une applet.
// <applet code=Button1 width=200 height=50>
// </applet>
import javax.swing.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class Button1 extends JApplet {
JButton
b1 = new JButton("Button 1"),
b2 = new JButton("Button 2");
  public void init() {
Container cp = getContentPane();
cp.setLayout(new FlowLayout());
cp.add(b1);
cp.add(b2);
}
  public static void main(String[] args) {
Console.run(new Button1(), 200, 50);
}
} ///:~
```

On a ajouté ici quelque chose de nouveau : avant d'ajouter un quelconque élément sur la "surface de contenu"*[content pane]*, on lui attribue un nouveau gestionnaire de disposition *[layout manager]*, de type **FlowLayout**. Le *layout manager* définit la façon dont la surface décide implicitement de l'emplacement du contrôle dans le formulaire. Le comportement d'une applet est d'utiliser le **BorderLayout**, mais cela ne marchera pas ici car (comme on l'apprendra plus tard dans ce chapitre lorsqu'on verra avec plus de détails le contrôle de l'organisation d'un formulaire) son comportement par défaut est de couvrir entièrement chaque contrôle par tout nouveau contrôle ajouté. Cependant, **FlowLayout** provoque l'alignement des contrôles uniformément dans le formulaire, de gauche à droite et de haut en bas.

Capture d'un événement

Vous remarquerez que si vous compilez et exécutez l'applet ci-dessus, rien ne se passe lorsqu'on appuie sur le bouton. C'est à vous de jouer et d'écrire le code qui définira ce qui va se passer. La base de la programmation par événements, qui est très importante dans les interfaces utilisateurs graphiques, est de lier les événements au code qui répond à ces événements.

Ceci est effectué dans Swing par une séparation claire de l'interface (les composants graphiques) et l'implémentation (le code que vous voulez exécuter quand un événement arrive sur un composant). Chaque composant Swing peut répercuter tous les événements qui peuvent lui arriver,

et il peut répercuter chaque type d'événement individuellement. Donc si par exemple on n'est pas intéressé par le fait que la souris est déplacée par-dessus le bouton, on n'enregistre pas son intérêt pour cet événement. C'est une façon très directe et élégante de gérer la programmation par événements, et une fois qu'on a compris les concepts de base on peut facilement utiliser les composants Swing qu'on n'a jamais vus auparavant. En fait, ce modèle s'étend à tout ce qui peut être classé comme un JavaBean (que nous verrons plus tard dans ce chapitre).

Au début, on s'intéressera uniquement à l'événement le plus important pour les composants utilisés. Dans le cas d'un **JButton**, l'événement intéressant est le fait qu'on appuie sur le bouton. Pour enregistrer son intérêt à l'appui sur un bouton, on appelle la méthode **addActionListener()** de **JButton**. Cette méthode attend un argument qui est un objet qui implémente l'interface **ActionListener**, qui contient une seule méthode appelée **actionPerformed()**. Donc tout ce qu'il faut faire pour attacher du code à un **JButton** est d'implémenter l'interface **ActionListener** dans une classe et d'enregistrer un objet de cette classe avec le **JButton** à l'aide de **addActionListener()**. La méthode sera appelée lorsque le bouton sera enfoncé (ceci est en général appelé un *callback*).

Mais que doit être le résultat de l'appui sur ce bouton ? On aimerait voir quelque chose changer à l'écran; pour cela on va introduire un nouveau composant Swing : le **JTextField**. C'est un endroit où du texte peut être tapé, ou dans notre cas modifié par le programme. Bien qu'il y ait plusieurs façons de façons de créer un **JTextField**, la plus simple est d'indiquer au constructeur uniquement quelle largeur on désire pour ce champ. Une fois le **JTextField** placé sur le formulaire, on peut modifier son contenu en utilisant la méthode **setText()** (il y a beaucoup d'autres méthodes dans **JTextField**, que vous pouvez découvrir dans la documentation HTML pour le JDK depuis *java.sun.com*). Voilà à quoi ça ressemble :

```
//: c13:Button2.java
// Réponse aux appuis sur un bouton.
// <applet code=Button2 width=200 height=75>
// </applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class Button2 extends JApplet {
JButton
  b1 = new JButton("Button 1"),
  b2 = new JButton("Button 2");
JTextField txt = new JTextField(10);
  class BL implements ActionListener {
   public void actionPerformed(ActionEvent e){
    String name =
      ((JButton)e.getSource()).getText();
    txt.setText(name);
   }
}
BL al = new BL();
  public void init() {
  b1.addActionListener(al);
```

```
    b2.addActionListener(al);
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(b1);
    cp.add(b2);
    cp.add(txt);
  }
    public static void main(String[] args) {
    Console.run(new Button2(), 200, 75);
  }
  } ///:~
```

Dans **init()**, **addActionListener()** est utilisée pour enregistrer l'objet **BL** pour chacun des boutons.

Il est souvent plus pratique de coder l'**ActionListener** comme une classe anonyme interne *[anonymous inner class]*, particulièrement lorsqu'on a tendance à n'utiliser qu'une seule instance de chaque classe listener. **Button2.java** peut être modifié de la façon suivante pour utiliser une classe interne anonyme :

```
//: c13:Button2b.java
// Utilisation de classes anonymes internes.
// <applet code=Button2b width=200 height=75>
// </applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class Button2b extends JApplet {
JButton
  b1 = new JButton("Button 1"),
  b2 = new JButton("Button 2");
JTextField txt = new JTextField(10);
ActionListener al = new ActionListener() {
   public void actionPerformed(ActionEvent e){
    String name =
      ((JButton)e.getSource()).getText();
    txt.setText(name);
  }
};
  public void init() {
  b1.addActionListener(al);
  b2.addActionListener(al);
  Container cp = getContentPane();
  cp.setLayout(new FlowLayout());
  cp.add(b1);
  cp.add(b2);
```

```
    cp.add(txt);
  }
    public static void main(String[] args) {
    Console.run(new Button2b(), 200, 75);
  }
  } ///:~
```

L'utilisation d'une classe anonyme interne sera préférée (si possible) pour les exemples de ce livre.

Zones de texte

Un **JTextArea** est comme un **JTextField**, sauf qu'il peut avoir plusieurs lignes et possède plus de fonctionnalités. Une méthode particulièrement utile est **append()**; avec cette méthode on peut facilement transférer une sortie dans un **JTextArea**, faisant de ce fait d'un programme Swing une amélioration (du fait qu'on peut scroller en arrière) par rapport à ce qui a été fait jusqu'ici en utilisant des programmes de ligne de commande qui impriment sur la sortie standard. Comme exemple, le programme suivant remplit un **JTextArea** avec la sortie du générateur **geography** du chapitre 9 :

```
//: c13:TextArea.java
// Utilisation du contrôle JTextArea.
// <applet code=TextArea width=475 height=425>
// </applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import java.util.*;
import com.bruceeckel.swing.*;
import com.bruceeckel.util.*;

public class TextArea extends JApplet {
JButton
  b = new JButton("Add Data"),
  c = new JButton("Clear Data");
JTextArea t = new JTextArea(20, 40);
Map m = new HashMap();
  public void init() {
   // Utilisation de toutes les données :
  Collections2.fill(m,
    Collections2.geography,
    CountryCapitals.pairs.length);
  b.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e){
      for(Iterator it= m.entrySet().iterator();
          it.hasNext();){
        Map.Entry me = (Map.Entry)(it.next());
        t.append(me.getKey() + ": "
```

```
            + me.getValue() + "\n");
        }
      }
    });
    c.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e){
        t.setText("");
      }
    });
    Container cp = getContentPane();
    cp.setLayonew FlowLayout());
    cp.add(new JScrollPane(t));
    cp.add(b);
    cp.add(c);
  }
    public static void main(String[] args) {
    Console.run(new TextArea(), 475, 425);
  }
  } ///:~
```

Dans init(), le Map est rempli avec tous les pays et leurs capitales. Remarquons que pour chaque bouton l'**ActionListener** est créé et ajouté sans définir de variable intermédiaire, puisqu'on n'aura plus jamais besoin de s'y référer dans la suite du programme. Le bouton "Add Data" formate et ajoute à la fin toutes les données, tandis que le bouton "Clear Data" utilise **setText()** pour supprimer tout le texte du **JTextArea**.

Lors de l'ajout du JTextArea à l'applet, il est enveloppé dans un **JScrollPane**, pour contrôler le scrolling quand trop de texte est placé à l'écran. C'est tout ce qu'il y a à faire pour fournir des fonctionnalités de scrolling complètes. Ayant essayé d'imaginer comment faire l'équivalent dans d'autres environnements de programmation de GUI, je suis très impressionné par la simplicité et la bonne conception de composants tels que le **JScrollPane**.

Contrôle de la disposition

La façon dont on place les composants sur un formulaire en Java est probablement différente de tout système de GUI que vous avez utilisé. Premièrement, tout est dans le code ; il n'y a pas de ressources qui contrôlent le placement des composants. Deuxièmement, la façon dont les composants sont placés dans un formulaire est contrôlée non pas par un positionnement absolu mais par un *layout manager* qui décide comment les composants sont placés, selon l'ordre dans lequel on les ajoute (**add()**). La taille, la forme et le placement des composants seront notablement différents d'un *layout manager* à l'autre. De plus, les gestionnaires de disposition s'adaptent aux dimensions de l'applet ou de la fenêtre de l'application, de sorte que si la dimension de la fenêtre est changée, la taille, la forme et le placement des composants sont modifiés en conséquence.

JApplet, **JFrame, JWindow** et **JDialog** peuvent chacun fournir un **Container** avec **getContentPane()** qui peut contenir et afficher des **Component**s. Dans **Container,** il y a une méthode appelée **setLayout()** qui permet de choisir le *layout manager*. D'autres classes telles que **JPanel** contiennent et affichent des composants directement, et donc il faut leur imposer directement le *layout manager*, sans utiliser le *content pane*.

Dans cette section nous allons explorer les divers gestionnaires de disposition en créant des boutons (puisque c'est ce qu'il y a de plus simple). Il n'y aura aucune capture d'événements de boutons puisque ces exemples ont pour seul but de montrer comment les boutons sont disposés.

BorderLayout

L'applet utilise un *layout manager* par défaut : le **BorderLayout** (certains des exemples précédents ont modifié le *layout manager* par défaut pour **FlowLayout**). Sans autre information, il prend tout ce qu'on lui ajoute (**add()**) et le place au centre, en étirant l'objet dans toutes les directions jusqu'aux bords.

Cependant le **BorderLayout** ne se résume pas qu'à cela. Ce *layout manager* possède le concept d'une zone centrale et de quatre régions périphériques. Quand on ajoute quelque chose à un *panel* qui utilise un **BorderLayout**, on peut utiliser la méthode **add()** surchargée qui prend une valeur constante comme premier argument. Cette valeur peut être une des suivantes :

BorderLayout.NORTH (en haut) **BorderLayout.SOUTH** (en bas) **BorderLayout.EAST** (à droite) **BorderLayout.WEST** (à gauche) **BorderLayout.CENTER** (remplir le milieu, jusqu'aux autres composants ou jusqu'aux bords)

Si aucune région n'est spécifiée pour placer l'objet, le défaut est **CENTER**.

Voici un exemple simple. Le *layout* par défaut est utilisé, puisque **JApplet** a **BorderLayout** par défaut :

```
//: c13:BorderLayout1.java
// Démonstration de BorderLayout.
// <applet code=BorderLayout1
// width=300 height=250> </applet>
import javax.swing.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class BorderLayout1 extends JApplet {
  public void init() {
  Container cp = getContentPane();
  cp.add(BorderLayout.NORTH,
    new JButton("North"));
  cp.add(BorderLayout.SOUTH,
    new JButton("South"));
  cp.add(BorderLayout.EAST,
    new JButton("East"));
  cp.add(BorderLayout.WEST,
    new JButton("West"));
cp.add(BorderLayout.CENTER,
    new JButton("Center"));
}
  public static void main(String[] args) {
  Console.run(new BorderLayout1(), 300, 250);
}
```

```
} ///:~
```

Pour chaque placement autre que **CENTER**, l'élément qu'on ajoute est comprimé pour tenir dans le plus petit espace le long d'une dimension et étiré au maximum le long de l'autre dimension. **CENTER**, par contre, s'étend dans chaque dimension pour occuper le milieu.

FlowLayout

Celui-ci aligne simplement les composants sur le formulaire, de gauche à droite jusqu'à ce que l'espace du haut soit rempli, puis descend d'une rangée et continue l'alignement.

Voici un exemple qui positionne le *layout manager* en **FlowLayout** et place ensuite des boutons sur le formulaire. On remarquera qu'avec **FlowLayout** les composants prennent leur taille naturelle. Un **JButton**, par exemple, aura la taille de sa chaîne.

```
//: c13:FlowLayout1.java
// Démonstration de FlowLayout.
// <applet code=FlowLayout1
// width=300 height=250> </applet>
import javax.swing.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class FlowLayout1 extends JApplet {
  public void init() {
  Container cp = getContentPane();
  cp.setLayout(new FlowLayout());
   for(int i = 0; i     cp.add(new JButton("Button " + i));
}
  public static void main(String[] args) {
  Console.run(new FlowLayout1(), 300, 250);
}
} ///:~
```

Tous les composants sont compactés à leur taille minimum dans un **FlowLayout**, ce qui fait qu'on peut parfois obtenir un comportement surprenant. Par exemple, vu qu'un **JLabel** prend la taille de sa chaîne, une tentative de justifier à droite son texte ne donne pas de modification de l'affichage dans un **FlowLayout**.

GridLayout

Un **GridLayout** permet de construire un tableau de composants, et lorsqu'on les ajoute ils sont placés de gauche à droite et de haut en bas dans la grille. Dans le constructeur on spécifie le nombre de rangées et de colonnes nécessaires, et celles-ci sont disposées en proportions identiques.

```
//: c13:GridLayout1.java
// Démonstration de GridLayout.
// <applet code=GridLayout1
// width=300 height=250> </applet>
```

```
import javax.swing.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class GridLayout1 extends JApplet {
  public void init() {
  Container cp = getContentPane();
  cp.setLayout(new GridLayout(7,3));
   for(int i = 0; i     cp.add(new JButton("Button " + i));
}
  public static void main(String[] args) {
  Console.run(new GridLayout1(), 300, 250);
}
} ///:~
```

Dans ce cas il y a 21 cases mais seulement 20 boutons. La dernière case est laissée vide car il n'y a pas d'équilibrage dans un **GridLayout**.

GridBagLayout

Le **GridBagLayout**nous donne un contrôle fin pour décider exactement comment les régions d'une fenêtre vont se positionner et se replacer lorsque la fenêtre est redimensionnée. Cependant, c'est aussi le plus compliqué des *layout managers*, et il est assez difficile à comprendre. Il est destiné principalement à la génération de code automatique par un constructeur d'interfaces utilisateurs graphiques *[GUI builder]* (les bons *GUI builders* utilisent **GridBagLayout** plutôt que le placement absolu). Si votre modèle est compliqué au point que vous sentiez le besoin d'utiliser le **GridBagLayout**, vous devrez dans ce cas utiliser un outil *GUI builder* pour générer ce modèle. Si vous pensez devoir en connaître les détails internes, je vous renvoie à *Core Java 2* par Horstmann & Cornell (Prentice-Hall, 1999), ou un livre dédié à Swing, comme point de départ.

Positionnement absolu

Il est également possible de forcer la position absolue des composants graphiques de la façon suivante :

1. Positionner un *layout manager* **null** pour le **Container** : **setLayout(null)**.
2. Appeler **setBounds()** ou **reshape()** (selon la version du langage) pour chaque composant, en passant un rectangle de limites avec ses coordonnées en pixels. Ceci peut se faire dans le constructeur, ou dans **paint()**, selon le but désiré.

Certains *GUI builders* utilisent cette approche de manière extensive, mais ce n'est en général pas la meilleure manière de générer du code. Les *GUI builders* les plus efficaces utilisent plutôt **GridBagLayout**.

BoxLayout

Les gens ayant tellement de problèmes pour comprendre et utiliser **GridBagLayout**, Swing contient également le **BoxLayout**, qui offre la plupart des avantages du **GridBagLayout** sans en avoir la complexité, de sorte qu'on peut souvent l'utiliser lorsqu'on doit coder à la main des *layouts*

(encore une fois, si votre modèle devient trop compliqué, utilisez un *GUI builder* qui générera les **GridBagLayout**s à votre place). BoxLayout permet le contrôle du placement des composants soit verticalement soit horizontalement, et le contrôle de l'espace entre les composants en utilisant des choses appelées *struts* (entretoises) et *glue* (colle). D'abord, voyons comment utiliser **BoxLayout** directement, en faisant le même genre de démonstration que pour les autres layout managers :

```
//: c13:BoxLayout1.java
// BoxLayouts vertical et horizontal.
// <applet code=BoxLayout1
// width=450 height=200> </applet>
import javax.swing.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class BoxLayout1 extends JApplet {
  public void init() {
  JPanel jpv = new JPanel();
  jpv.setLayout(
    new BoxLayout(jpv, BoxLayout.Y_AXIS));
   for(int i = 0; i     jpv.add(new JButton("" + i));
  JPanel jph = new JPanel();
  jph.setLayout(
    new BoxLayout(jph, BoxLayout.X_AXIS));
   for(int i = 0; i     jph.add(new JButton("" + i));
  Container cp = getContentPane();
  cp.add(BorderLayout.EAST, jpv);
  cp.add(BorderLayout.SOUTH, jph);
}
  public static void main(String[] args) {
  Console.run(new BoxLayout1(), 450, 200);
}
} ///:~
```

Le constructeur du **BoxLayout** est un peu différent des autres *layout managers* : on fournit le **Container** que le **BoxLayout** doit contrôler comme premier argument, et la direction du *layout* comme deuxième argument.

Pour simplifier les choses, il y a un *container* spécial appelé **Box** qui utilise **BoxLayout** comme *manager* d'origine. L'exemple suivant place les composants horizontalement et verticalement en utilisant **Box**, qui possède deux méthodes **static** pour créer des *boxes* avec des alignements verticaux et horizontaux :

```
//: c13:Box1.java
// BoxLayouts vertical et horizontal.
// <applet code=Box1
// width=450 height=200> </applet>
import javax.swing.*;
import java.awt.*;
import com.bruceeckel.swing.*;
```

```
public class Box1 extends JApplet {
  public void init() {
  Box bv = Box.createVerticalBox();
   for(int i = 0; i     bv.add(new JButton("" + i));
  Box bh = Box.createHorizontalBox();
   for(int i = 0; i     bh.add(new JButton("" + i));
  Container cp = getContentPane();
  cp.add(BorderLayout.EAST, bv);
  cp.add(BorderLayout.SOUTH, bh);
}
  public static void main(String[] args) {
  Console.run(new Box1(), 450, 200);
}
} ///:~
```

Une fois qu'on a obtenu un **Box**, on le passe en second argument quand on ajoute des composants au *content pane*.

Les *struts* ajoutent de l'espace entre les composants, mesuré en pixels. Pour utiliser un *strut*, on l'ajoute simplement entre les ajouts de composants que l'on veut séparer :

```
//: c13:Box2.java
// Ajout de struts.
// <applet code=Box2
// width=450 height=300> </applet>
import javax.swing.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class Box2 extends JApplet {
  public void init() {
  Box bv = Box.createVerticalBox();
   for(int i = 0; i     bv.add(new JButton("" + i));
    bv.add(Box.createVerticalStrut(i*10));
   }
  Box bh = Box.createHorizontalBox();
   for(int i = 0; i     bh.add(new JButton("" + i));
    bh.add(Box.createHorizontalStrut(i*10));
   }
  Container cp = getContentPane();
  cp.add(BorderLayout.EAST, bv);
  cp.add(BorderLayout.SOUTH, bh);
}
  public static void main(String[] args) {
  Console.run(new Box2(), 450, 300);
}
} ///:~
```

```
//: c13:Box3.java
// Utilisation de Glue.
// <applet code=Box3
// width=450 height=300> </applet>
import javax.swing.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class Box3 extends JApplet {
  public void init() {
  Box bv = Box.createVerticalBox();
  bv.add(new JLabel("Hello"));
  bv.add(Box.createVerticalGlue());
  bv.add(new JLabel("Applet"));
  bv.add(Box.createVerticalGlue());
  bv.add(new JLabel("World"));
  Box bh = Box.createHorizontalBox();
  bh.add(new JLabel("Hello"));
  bh.add(Box.createHorizontalGlue());
  bh.add(new JLabel("Applet"));
  bh.add(Box.createHorizontalGlue());
  bh.add(new JLabel("World"));
  bv.add(Box.createVerticalGlue());
  bv.add(bh);
  bv.add(Box.createVerticalGlue());
  getContentPane().add(bv);
}
  public static void main(String[] args) {
  Console.run(new Box3(), 450, 300);
}
} ///:~
```

Un *strut* fonctionne dans une direction, mais une *rigid area* (surface rigide) fixe l'espace entre les composants dans chaque direction :

```
//: c13:Box4.java
// Des Rigid Areas sont comme des paires de struts.
// <applet code=Box4
// width=450 height=300> </applet>
import javax.swing.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class Box4 extends JApplet {
  public void init() {
  Box bv = Box.createVerticalBox();
  bv.add(new JButton("Top"));
```

```
    bv.add(Box.createRigidArea(
      new Dimension(120, 90)));
    bv.add(new JButton("Bottom"));
    Box bh = Box.createHorizontalBox();
    bh.add(new JButton("Left"));
    bh.add(Box.createRigidArea(
      new Dimension(160, 80)));
    bh.add(new JButton("Right"));
    bv.add(bh);
    getContentPane().add(bv);
  }
    public static void main(String[] args) {
    Console.run(new Box4(), 450, 300);
  }
  } ///:~
```

Il faut savoir que les *rigid areas* sont un peu controversées. Comme elles utilisent des valeurs absolues, certaines personnes pensent qu'elles causent plus de problèmes qu'elles n'en résolvent.

La meilleure approche ?

Swing est puissant; il peut faire beaucoup de choses en quelques lignes de code. Les exemples de ce livre sont raisonnablement simples, et dans un but d'apprentissage il est normal de les écrire à la main. On peut en fait réaliser pas mal de choses en combinant des *layouts* simples. A un moment donné, il devient cependant déraisonnable de coder à la main des formulaires de GUI. Cela devient trop compliqué et ce n'est pas une bonne manière d'utiliser son temps de programmation. Les concepteur de Java et de Swing ont orienté le langage et ses bibliothèques de manière à soutenir des outils de construction de GUI, qui ont été créés dans le but de rendre la tâche de programmation plus facile. A partir du moment où on comprend ce qui se passe dans les layouts et comment traiter les événements (décrits plus loin), il n'est pas particulièrement important de connaître effectivement tous les détails sur la façon de positionner les composants à la main. Laissons les outils appropriés le faire pour nous (Java est après tout conçu pour augmenter la productivité du programmeur).

Le modèle d'événements de Swing

Dans le modèle d'événements de Swing, un composant peut initier (envoyer *[fire]*) un événement. Chaque type d'événement est représenté par une classe distincte. Lorsqu'un événement est envoyé, il est reçu par un ou plusieurs écouteurs *[listeners]*, qui réagissent à cet événement. De ce fait, la source d'un événement et l'endroit où cet événement est traité peuvent être séparés. Puisqu'on utilise en général les composants Swing tels quels, mais qu'il faut écrire du code appelé lorsque les composants reçoivent un événement, ceci est un excellent exemple de la séparation de l'interface et de l'implémentation.

Chaque écouteur d'événements *[event listener]* est un objet d'une classe qui implémente une **interface** particulière de type *listener*. En tant que programmeur, il faut créer un objet *listener* et l'enregistrer avec le composant qui envoie l'événement. Cet enregistrement se fait par appel à une méthode **addXXXListener()** du composant envoyant l'événement, dans lequel **XXX** représente le

type d'événement qu'on écoute.On peut facilement savoir quels types d'événements peuvent être gérés en notant les noms des méthodes addListener, et si on essaie d'écouter des événements erronés, l'erreur sera signalée à la compilation. On verra plus loin dans ce chapitre que les JavaBeans utilisent aussi les noms des méthodes addListener pour déterminer quels événements un Bean peut gérer.

Toute notre logique des événements va se trouver dans une classe *listener*. Lorsqu'on crée une classe *listener*, la seule restriction est qu'elle doit implémenter l'interface appropriée. On peut créer une classe *listener* globale, mais on est ici dans un cas où les classes internes sont assez utiles, non seulement parce qu'elles fournissent un groupement logique de nos classes *listener* à l'intérieur des classes d'interface utilisateur ou de logique métier qu'elles desservent, mais aussi (comme on le verra plus tard) parce que le fait qu'un objet d'une classe interne garde une référence à son objet parent fournit une façon élégante d'appeler à travers les frontières des classes et des sous-systèmes.

Jusqu'ici, tous les exemples de ce chapitre ont utilisé le modèle d'événements Swing, mais le reste de cette section va préciser les détails de ce modèle.

Evénements et types de *listeners*

Chaque composant Swing contient des méthodes **addXXXListener()** et **removeXXXListener()** de manière à ce que les types de *listeners* adéquats puissent être ajoutés et enlevés de chaque composant. On remarquera que le **XXX** dans chaque cas représente également l'argument de cette méthode, par exemple : **addMyListener(MyListener m)**. Le tableau suivant contient les événements, listeners et méthodes de base associées aux composants de base qui supportent ces événements particuliers en fournissant les méthodes **addXXXListener()** et **removeXXXListener()**. Il faut garder en tête que le modèle d'événements est destiné à être extensible, et donc on pourra rencontrer d'autres types d'événements et de *listeners* non couverts par ce tableau.

Evénement, interface *listener* et méthodes add et remove	**Composants supportant cet événement**
ActionEvent ActionListener addActionListener() removeActionListener()	**JButton, JList, JTextField, JMenuItem** et ses dérivés, comprenant **JCheckBoxMenuItem, JMenu,** et **JpopupMenu.**
AdjustmentEvent AdjustmentListener addAdjustmentListener() removeAdjustmentListener()	**JScrollbar** et tout ce qu'on crée qui implémente l'**interface Adjustable.**
ComponentEvent ComponentListener addComponentListener() removeComponentListener()	***Component** et ses dérivés, comprenant **JButton, JCanvas, JCheckBox, JComboBox, Container, JPanel, JApplet, JScrollPane, Window, JDialog, JFileDialog, JFrame, JLabel, JList, JScrollbar, JTextArea,** et **JTextField.**
ContainerEvent ContainerListener addContainerListener() removeContainerListener()	**Container** et ses dérivés, comprenant **JPanel, JApplet, JScrollPane, Window, JDialog, JFileDialog,** et **JFrame.**
FocusEvent FocusListener addFocusListener() removeFocusListener()	**Component** et **dérivés*.**

KeyEvent KeyListener addKeyListener() removeKeyListener()	**Component** et **dérivés*.**
MouseEvent (à la fois pour les clics et pour le déplacement) **MouseListener addMouseListener() removeMouseListener()**	**Component** et **dérivés*.**
MouseEvent [68](à la fois pour les clics et pour le déplacement) **MouseMotionListener addMouseMotionListener() removeMouseMotionListener()**	**Component** et **dérivés*.**
WindowEvent WindowListener addWindowListener() removeWindowListener()	**Window** et ses dérivés, comprenant **JDialog, JFileDialog, and JFrame.**
ItemEvent ItemListener addItemListener() removeItemListener()	**JCheckBox, JCheckBoxMenuItem, JComboBox, JList,** et tout ce qui implémente l'interface **ItemSelectable.**
TextEvent TextListener addTextListener() removeTextListener()	Tout ce qui est dérivé de **JTextComponent,** comprenant **JTextArea** et **JTextField.**

On voit que chaque type de composant ne supporte que certains types d'événements. Il semble assez difficile de rechercher tous les événements supportés par chaque composant. Une approche plus simple consiste à modifier le programme **ShowMethodsClean.java** du chapitre 12 de manière à ce qu'il affiche tous les *event listeners* supportés par tout composant Swing entré.

Le chapitre 12 a introduit la *réflexion* et a utilisé cette fonctionnalité pour rechercher les méthodes d'une classe donnée, soit une liste complète des méthodes, soit un sous-ensemble des méthodes dont le nom contient un mot-clé donné. La magie dans ceci est qu'il peut automatiquement nous montrer *toutes* les méthodes d'une classe sans qu'on soit obligé de parcourir la hiérarchie des héritages en examinant les classes de base à chaque niveau. De ce fait, il fournit un outil précieux permettant de gagner du temps pour la programmation : comme les noms de la plupart des méthodes Java sont parlants et descriptifs, on peut rechercher les noms de méthodes contenant un mot particulier. Lorsqu'on pense avoir trouvé ce qu'on cherchait, il faut alors vérifier la documentation en ligne.

Comme dans le chapitre 12 on n'avait pas encore vu Swing, l'outil de ce chapitre était une application de ligne de commande. En voici une version plus pratique avec interface graphique, spécialisée dans la recherche des méthodes addListener dans les composants Swing :

```
//: c13:ShowAddListeners.java
// Affiche les methodes "addXXXListener"
// d'une classe Swing donnee.
// <applet code = ShowAddListeners
// width=500 height=400></applet>
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
```

```
import java.lang.reflect.*;
import java.io.*;
import com.bruceeckel.swing.*;
import com.bruceeckel.util.*;

public class ShowAddListeners extends JApplet {
  Class cl;
  Method[] m;
  Constructor[] ctor;
  String[] n = new String[0];
  JTextField name = new JTextField(25);
  JTextArea results = new JTextArea(40, 65);
  class NameL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      String nm = name.getText().trim();
      if(nm.length() == 0) {
        results.setText("No match");
        n = new String[0];
        return;
      }
      try {
        cl = Class.forName("javax.swing." + nm);
      } catch(ClassNotFoundException ex) {
        results.setText("No match");
        return;
      }
      m = cl.getMethods();
      // Conversion en un tableau de Strings :
      n = new String[m.length];
      for(int i = 0; i < m.length; i++)
        n[i] = m[i].toString();
      reDisplay();
    }
  }
  void reDisplay() {
    // Creation de l'ensemble des resultats :
    String[] rs = new String[n.length];
    int j = 0;
    for (int i = 0; i < n.length; i++)
      if(n[i].indexOf("add") != -1 && n[i].indexOf("Listener") != -1)
        rs[j++] =  n[i].substring(n[i].indexOf("add"));
    results.setText("");
    for (int i = 0; i < j; i++)
      results.append( StripQualifiers.strip(rs[i]) + "\n");
  }
  public void init() {
```

```
    name.addActionListener(new NameL());
    JPanel top = new JPanel();
    top.add(new JLabel(
      "Swing class name (press ENTER):"));
    top.add(name);
    Container cp = getContentPane();
    cp.add(BorderLayout.NORTH, top);
    cp.add(new JScrollPane(results));
  }
  public static void main(String[] args) {
    Console.run(new ShowAddListeners(), 500,400);
  }
} ///:~
```

La classe **StripQualifiers** définie au chapitre 12 est réutilisée ici en important la bibliothèque **com.bruceeckel.util**.

L'interface utilisateur graphique contient un **JTextField name** dans lequel on saisit le nom de la classe Swing à rechercher. Les résultats sont affichés dans une **JTextArea**.

On remarquera qu'il n'y a pas de boutons ou autres composants pour indiquer qu'on désire lancer la recherche. C'est parce que le **JTextField** est surveillé par un **ActionListener**. Lorsqu'on y fait un changement suivi de ENTER, la liste est immédiatement mise à jour. Si le texte n'est pas vide, il est utilisé dans **Class.forName()** pour rechercher la classe. Si le nom est incorrect, **Class.forName()** va échouer, c'est à dire qu'il va émettre une exception. Celle-ci est interceptée et le JTextArea est positionné à "No match". Mais si on tape un nom correct (les majuscules/minuscules comptent), **Class.forName()** réussit et **getMethods()** retourne un tableau d'objets **Method**. Chacun des objets du tableau est transformé en **String** à l'aide de **toString()** (cette méthode fournit la signature complète de la méthode) et ajoutée à **n**, un tableau de **String**s. Le tableau **n** est un membre de la **classe ShowAddListeners** et est utilisé pour mettre à jour l'affichage chaque fois que **reDisplay()** est appelé.

reDisplay() crée un tableau de **String**s appelé **rs** (pour "result set" : ensemble de résultats). L'ensemble des résultats est conditionnellement copié depuis les **String**s de **n** qui contiennent add et Listener. **indexOf()** et **substring()** sont ensuite utilisés pour enlever les qualificatifs tels que public, static, etc. Enfin, **StripQualifiers.strip()** enlève les qualificatifs de noms.

Ce programme est une façon pratique de rechercher les capacités d'un composant Swing. Une fois connus les événements supportés par un composant donné, il n'y a pas besoin de rechercher autre chose pour réagir à cet événement. Il suffit de :

1. Prendre le nom de la classe événement et retirer le mot **Event**. Ajouter le mot **Listener** à ce qui reste. Ceci donne le nom de l'interface *listener* qu'on doit implémenter dans une classe interne.
2. Implémenter l'interface ci-dessus et écrire les méthodes pour les événements qu'on veut intercepter. Par exemple, on peut rechercher les événements de déplacement de la souris, et on écrit donc le code pour la méthode mouseMoved() de l'interface **MouseMotionListener** (il faut également implémenter les autres méthodes, bien sûr, mais il y a souvent un raccourci que nous verrons bientôt).
3. Créer un objet de la classe listener de l'étape 2. L'enregistrer avec le composant avec

la méthode dont le nom est fourni en ajoutant **add** au début du nom du *listener*. Par exemple, **addMouseMotionListener()**.

Voici quelques-unes des interfaces *listeners* :

Interface Listener et Adapter	Methodes de l'interfaces
ActionListener	**actionPerformed(ActionEvent)**
AdjustmentListener	**AdjustmentValueChanged(Adjustmentevent)**
ComponentListener **ComponentAdapter**	**componentHidden(ComponentEvent)** **componentShown(ComponentEvent)** **componentmoved(ComponentEvent)** **componentResized(ComponentEvent)**
ContainerListener **ContainerAdapter**	**componentAdded(ContainerEvent)** **componentRemoved(ContainerEvent)**
FocusListener **FocusAdapter**	**focusGained(FocusEvent)** **focusLost(FocusEvent)**
KeyListener **KeyAdapter**	**keyPressed(KeyEvent)** **keyReleased(KeyEvent)** **keyTyped(KeyEvent)**
MouseListener **MouseAdapter**	**mouseClicked(MouseEvent)** **mouseEntered(MouseEvent)** **mouseExited(MouseEvent)** **mousePressed(MouseEvent)** **mouseReleased(MouseEvent)**
MouseMotionListener **MouseMotionAdapter**	**mouseDragged(MouseEvent)** **mouseMoved(MouseEvent)**
WindowListener **WindowAdapter**	**windowOpened(WindowEvent)** **windowClosing(WindowEvent)** **windowClosed(WindowEvent)** **windowActivated(WindowEvent)** **windowDeactived(WindowEvent)** **windowIconified(WindowEvent)** **windowDeiconified(windowEvent)**
ItemListener	ItemStateChanged(ItemEvent)

Ce n'est pas une liste exhaustive, en partie du fait que le modèle d'événements nous permet de créer nos propres types d'événements et *listeners* associés. De ce fait, on rencontrera souvent des bibliothèques qui ont inventé leurs propres événements, et la connaissance acquise dans ce chapitre nous permettra de comprendre l'utilisation de ces événements.

Utilisation de *listener adapters* pour simplifier

Dans le tableau ci-dessus, on peut voir que certaines interfaces *listener* ne possèdent qu'une seule méthode. Celles-ci sont triviales à implémenter puisqu'on ne les implémentera que lorsqu'on désire écrire cette méthode particulière. Par contre, les interfaces *listener* qui ont plusieurs méthodes

peuvent être moins agréables à utiliser. par exemple, quelque chose qu'il faut toujours faire en créant une application est de fournir un **WindowListener** au **JFrame** de manière à pouvoir appeler **System.exit()** pour sortir de l'application lorsqu'on reçoit l'événement **windowClosing()**. Mais comme **WindowListener** est une **interface**, il faut implémenter chacune de ses méthodes même si elles ne font rien. Cela peut être ennuyeux.

Pour résoudre le problème, certaines (mais pas toutes) des interfaces *listener* qui ont plus d'une méthode possèdent des adaptateurs *[adapters]*, dont vous pouvez voir les noms dans le tableau ci-dessus. Chaque adaptateur fournit des méthodes vides par défaut pour chacune des méthodes de l'interface. Ensuite il suffit d'hériter de cet adaptateur et de redéfinir uniquement les méthodes qu'on doit modifier. Par exemple, le **WindowListener** qu'on utilisera normalement ressemble à ceci (souvenez-vous qu'il a été encapsulé dans la classe **Console** de **com.bruceeckel.swing**) :

```
class MyWindowListener extends WindowAdapter {
  public void windowClosing(WindowEvent e) {
System.exit(0);
}
}
```

Le seul but des adaptateurs est de faciliter la création des classes *listener*.

Il y a cependant un désavantage lié aux adaptateurs, sous la forme d'un piège. Supposons qu'on écrive un **WindowAdapter** comme celui ci-dessus :

```
class MyWindowListener extends WindowAdapter {
  public void WindowClosing(WindowEvent e) {
System.exit(0);
}
}
```

Ceci ne marche pas, mais il nous rendra fous à comprendre pourquoi, car tout va compiler et s'exécuter correctement, sauf que la fermeture de la fenêtre ne fera pas sortir du programme. Voyez-vous le problème ? Il est situé dans le nom de la méthode : **WindowClosing()** au lieu de **windowClosing()**. Une simple erreur de majuscule se traduit par l'ajout d'une méthode nouvelle. Ce n'est cependant pas cette méthode qui est appelée lorsque la fenêtre est fermée, de sorte qu'on n'obtient pas le résultat attendu. En dépit de cet inconvénient, une interface garantit que les méthodes sont correctement implémentées.

Surveiller plusieurs événements

Pour nous prouver que ces événements sont bien déclenchés, et en tant qu'expérience intéressante, créons une applet qui surveille les autres comportement d'un **JButton**, autres que le simple fait qu'il soit appuyé ou pas. Cet exemple montre également comment hériter de notre propre objet bouton, car c'est ce qui est utilisé comme cible de tous les événements intéressants. Pour cela, il suffit d'hériter de **JButton** name="fnB69" href="#fn69">[69].

La classe **MyButton** est une classe interne de TrackEvent, de sorte que **MyButton** peut aller dans la fenêtre parent et manipuler ses champs textes, ce qu'il faut pour pouvoir écrire une information d'état dans les champs du parent. Bien sûr ceci est une solution limitée, puisque **MyButton** peut être utilisée uniquement avec **TrackEvent**. Ce genre de code est parfois appelé "fortement

couplé" :

```
//: c13:TrackEvent.java
// Montre les evenements lorsqu'ils arrivent.
// <applet code=TrackEvent
//  width=700 height=500></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import com.bruceeckel.swing.*;

public class TrackEvent extends JApplet {
HashMap h = new HashMap();
String[] event = {
  "focusGained", "focusLost", "keyPressed",
  "keyReleased", "keyTyped", "mouseClicked",
  "mouseEntered", "mouseExited","mousePressed",
  "mouseReleased", "mouseDragged", "mouseMoved"
};
MyButton
b1 = new MyButton(Color.blue, "test1"),
b2 = new MyButton(Color.red, "test2");
 class MyButton extends JButton {
  void report(String field, String msg) {
((JTextField)h.get(field)).setText(msg);
}
FocusListener fl = new FocusListener() {
   public void focusGained(FocusEvent e) {
  report("focusGained", e.paramString());
}
   public void focusLost(FocusEvent e) {
  report("focusLost", e.paramString());
}
};
KeyListener kl = new KeyListener() {
   public void keyPressed(KeyEvent e) {
  report("keyPressed", e.paramString());
}
   public void keyReleased(KeyEvent e) {
  report("keyReleased", e.paramString());
}
   public void keyTyped(KeyEvent e) {
  report("keyTyped", e.paramString());
}
};
MouseListener ml = new MouseListener() {
   public void mouseClicked(MouseEvent e) {
```

```
    report("mouseClicked", e.paramString());
  }
    public void mouseEntered(MouseEvent e) {
    report("mouseEntered", e.paramString());
  }
    public void mouseExited(MouseEvent e) {
    report("mouseExited", e.paramString());
  }
    public void mousePressed(MouseEvent e) {
    report("mousePressed", e.paramString());
  }
    public void mouseReleased(MouseEvent e) {
    report("mouseReleased", e.paramString());
  }
  };
  MouseMotionListener mml =
    new MouseMotionListener() {
    public void mouseDragged(MouseEvent e) {
    report("mouseDragged", e.paramString());
  }
    public void mouseMoved(MouseEvent e) {
    report("mouseMoved", e.paramString());
  }
  };
    public MyButton(Color color, String label) {
     super(label);
  setBackground(color);
  addFocusListener(fl);
  addKeyListener(kl);
  addMouseListener(ml);
  addMouseMotionListener(mml);
  }
  }
   public void init() {
  Container c = getContentPane();
  c.setLayout(new GridLayout(event.length+1,2));
    for(int i = 0; i JTextField t = new JTextField();
  t.setEditable(false);
  c.add(new JLabel(event[i], JLabel.RIGHT));
  c.add(t);
  h.put(event[i], t);
  }
  c.add(b1);
  c.add(b2);
  }
   public static void main(String[] args) {
  Console.run(new TrackEvent(), 700, 500);
```

```
} 
} ///:~
```

Dans le constructeur de **MyButton**, la couleur des boutons est positionnée par un appel à **SetBackground()**. Les *listeners* sont tous installés par de simples appels de méthodes.

La classe TrackEvent contient une **HashMap** pour contenir les chaînes représentant le type d'événement et les **JTextField**s dans lesquels l'information sur cet événement est conservée. Bien sûr, ceux-ci auraient pu être créés en statique plutôt qu'en les mettant dans une **HashMap**, mais je pense que vous serez d'accord que c'est beaucoup plus facile à utiliser et modifier. En particulier, si on a besoin d'ajouter ou supprimer un nouveau type d'événement dans **TrackEvent**, il suffit d'ajouter ou supprimer une chaîne dans le tableau **event**, et tout le reste est automatique.

Lorsque **report()** est appelé on lui donne le nom de l'événement et la chaîne des paramètres de cet événement. Il utilise le **HashMap h** de la classe externe pour rechercher le **JTextField** associé à l'événement portant ce nom, et place alors la chaîne des paramètres dans ce champ.

Cet exemple est amusant à utiliser car on peut réellement voir ce qui se passe avec les événements dans son programme.

Un catalogue de composants Swing

Maintenant que nous connaissons les *layout managers* et le modèle d'événements, nous sommes prêts pour voir comment utiliser les composants Swing. Cette section est une visite non exhaustive des composants Swing et des fonctionnalités que vous utiliserez probablement la plupart du temps. Chaque exemple est conçu pour être de taille raisonnable de manière à pouvoir facilement récupérer le code dans d'autres programmes.

Vos pouvez facilement voir à quoi ressemble chacun de ces exemples en fonctionnement, en visualisant les pages HTML dans le code source téléchargeable de ce chapitre.

Gardez en tête :

1. La documentation HTML de *java.sun.com* comprend toutes les classes et méthodes de Swing (seules quelques-unes sont montrées ici).
2. Grâce aux conventions de nommage utilisées pour les événements Swing, il est facile de deviner comment écrire et installer un gestionnaire d'un événement de type donné. Utilisez le programme de recherche **ShowAddListeners.java** introduit plus avant dans ce chapitre pour faciliter votre investigation d'un composant particulier.
3. Lorsque les choses deviendront compliquées, passez à un *GUI builder*.

Boutons

Swing comprend un certain nombre de boutons de différents types. Tous les boutons, boîtes à cocher *[check boxes]*, boutons radio *[radio buttons]*, et même les éléments de menus *[menu items]* héritent de **AbstractButton**(qui, vu qu'ils comprennent les éléments de menus, auraient probablement été mieux nommés AbstractChooser ou quelque chose du genre). Nous verrons l'utilisation des éléments de menus bientôt, mais l'exemple suivant montre les différents types de boutons existants :

```
//: c13:Buttons.java
// Divers boutons Swing.
```

```
// <applet code=Buttons
//  width=350 height=100></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.plaf.basic.*;
import javax.swing.border.*;
import com.bruceeckel.swing.*;

public class Buttons extends JApplet {
JButton jb = new JButton("JButton");
BasicArrowButton
up = new BasicArrowButton(
BasicArrowButton.NORTH),
down = new BasicArrowButton(
BasicArrowButton.SOUTH),
right = new BasicArrowButton(
BasicArrowButton.EAST),
left = new BasicArrowButton(
BasicArrowButton.WEST);
  public void init() {
Container cp = getContentPane();
cp.setLayout(new FlowLayout());
cp.add(jb);
cp.add(new JToggleButton("JToggleButton"));
cp.add(new JCheckBox("JCheckBox"));
cp.add(new JRadioButton("JRadioButton"));
JPanel jp = new JPanel();
jp.setBorder(new TitledBorder("Directions"));
jp.add(up);
jp.add(down);
jp.add(left);
jp.add(right);
cp.add(jp);
}
  public static void main(String[] args) {
Console.run(new Buttons(), 350, 100);
}
} ///:~
```

On commence par le **BasicArrowButton**de **javax.swing.plaf.basic**, puis on continue avec les divers types de boutons. Si vous exécutez cet exemple, vous verrez que le *toggle button* (bouton inverseur) garde sa dernière position, enfoncé ou relâché. Mais les boîtes à cocher et les boutons radio se comportent de manière identique, on les clique pour les (dé)sélectionner (ils sont hérités de **JToggleButton**).

Groupes de boutons

Si on désire des boutons radio qui se comportent selon un "ou exclusif", il faut les ajouter à un groupe de boutons. Mais, comme l'exemple ci-dessous le montre, tout **AbstractButton** peut être ajouté à un **ButtonGroup**.

Pour éviter de répéter beaucoup de code, cet exemple utilise la réflexion pour générer les groupes de différents types de boutons. Ceci peut se voir dans **makeBPanel()**, qui crée un groupe de boutons et un **JPanel**. Le second argument de **makeBPanel()** est un tableau de **String**. Pour chaque **String**, un bouton de la classe désignée par le premier argument est ajouté au **JPanel** :

```
//: c13:ButtonGroups.java
// Utilise la reflexion pour creer des groupes
// de differents types de AbstractButton.
// <applet code=ButtonGroups
//  width=500 height=300></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.border.*;
import java.lang.reflect.*;
import com.bruceeckel.swing.*;

public class ButtonGroups extends JApplet {
  static String[] ids = {
   "June", "Ward", "Beaver",
   "Wally", "Eddie", "Lumpy",
};
  static JPanel
makeBPanel(Class bClass, String[] ids) {
ButtonGroup bg = new ButtonGroup();
JPanel jp = new JPanel();
String title = bClass.getName();
title = title.substring(
title.lastIndexOf('.') + 1);
jp.setBorder(new TitledBorder(title));
   for(int i = 0; i AbstractButton ab = new JButton("failed");
    try {
     // Obtient la méthode de construction dynamique
     // qui demande un argument String :
   Constructor ctor = bClass.getConstructor(
      new Class[] { String.class });
     // Creation d'un nouve objet :
   ab = (AbstractButton)ctor.newInstance(
      new Object[]{ids[i]});
} catch(Exception ex) {
   System.err.println("can't create " +
     bClass);
}
```

```
bg.add(ab);
jp.add(ab);
}
  return jp;
}
 public void init() {
Container cp = getContentPane();
cp.setLayout(new FlowLayout());
cp.add(makeBPanel(JButton.class, ids));
cp.add(makeBPanel(JToggleButton.class, ids));
cp.add(makeBPanel(JCheckBox.class, ids));
cp.add(makeBPanel(JRadioButton.class, ids));
}
 public static void main(String[] args) {
Console.run(new ButtonGroups(), 500, 300);
}
} ///:~
```

Ceci ajoute un peu de complexité à ce qui est un processus simple. Pour obtenir un comportement de "OU exclusif" avec des boutons, on crée un groupe de boutons et on ajoute au groupe chaque bouton pour lequel on désire ce comportement. Lorsqu'on exécute le programme, on voit que tous les boutons, à l'exception de **JButton,** montrent ce comportement de "OU exclusif".

Icones

On peut utiliser un **Icon** dans un **JLabel** ou tout ce qui hérite de **AbstractButton** (y compris **JButton**, **JCheckBox**, **JRadioButton,** et les différents types de **JMenuItem**). L'utilisation d'**Icon**s avec des **JLabel**s est assez directe (on verra un exemple plus tard). L'exemple suivant explore toutes les façons d'utiliser des **Icon**s avec des boutons et leurs descendants.

Vous pouvez utiliser les fichiers gif que vous voulez, mais ceux utilisés dans cet exemple font partie de la livraison du code de ce livre, disponible à *www.BruceEckel.com*. Pour ouvrir un fichier et utiliser l'image, il suffit de créer un **ImageIcon** et de lui fournir le nom du fichier. A partir de là on peut utiliser l'**Icon** obtenu dans le programme.

Remarquez que l'information de chemin est codée en dur dans cet exemple; vous devrez changer ce chemin pour qu'il corresponde à l'emplacement des fichiers des images.

```
//: c13:Faces.java
// Comportement des Icones dans des  Jbuttons.
// <applet code=Faces
//  width=250 height=100></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

public class Faces extends JApplet {
  // L'information de chemin suivante est nécessaire
```

```
  // pour l'exécution via une applet directement depuis le disque :
  static String path =
   "C:/aaa-TIJ2-distribution/code/c13/";
  static Icon[] faces = {
   new ImageIcon(path + "face0.gif"),
   new ImageIcon(path + "face1.gif"),
   new ImageIcon(path + "face2.gif"),
   new ImageIcon(path + "face3.gif"),
   new ImageIcon(path + "face4.gif"),
};
JButton
jb = new JButton("JButton", faces[3]),
jb2 = new JButton("Disable");
  boolean mad = false;
  public void init() {
Container cp = getContentPane();
cp.setLayout(new FlowLayout());
jb.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e){
     if(mad) {
     jb.setIcon(faces[3]);
     mad = false;
    } else {
     jb.setIcon(faces[0]);
     mad = true;
    }
   jb.setVerticalAlignment(JButton.TOP);
   jb.setHorizontalAlignment(JButton.LEFT);
}
});
jb.setRolloverEnabled(true);
jb.setRolloverIcon(faces[1]);
jb.setPressedIcon(faces[2]);
jb.setDisabledIcon(faces[4]);
jb.setToolTipText("Yow!");
cp.add(jb);
jb2.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e){
     if(jb.isEnabled()) {
     jb.setEnabled(false);
     jb2.setText("Enable");
    } else {
     jb.setEnabled(true);
     jb2.setText("Disable");
    }
}
});
```

```
cp.add(jb2);
}
  public static void main(String[] args) {
Console.run(new Faces(), 400, 200);
}
} ///:~
```

Un **Icon** peut être utilisé dans de nombreux constructeurs, mais on peut aussi utiliser **setIcon()** pour ajouter ou changer un **Icon**. Cet exemple montre également comment un **JButton** (ou un quelconque **AbstractButton**) peut positionner les différentes sortes d'icones qui apparaissent lorsque des choses se passent sur ce bouton : lorsqu'il est enfoncé, invalidé, ou lorsqu'on roule par dessus *[rolled over]* (la souris passe au-dessus sans cliquer). On verra que ceci donne au bouton une sensation d'animation agréable.

Infobulles [Tooltips]

L'exemple précédent ajoutait un *tool tip* au bouton. La plupart des classes qu'on utilisera pour créer une interface utilisateur sont dérivées de **JComponent**, qui contient une méthode appelée **setToolTipText(String)**. Donc pratiquement pour tout ce qu'on place sur un formulaire, il suffit de dire (pour un objet **jc** de toute classe dérivée de **JComponent**) :

```
jc.setToolTipText("My tip");
```

et lorsque la souris reste au-dessus de ce JComponent pour un temps prédéterminé, une petite boîte contenant le texte va apparaître à côté de la souris.

Champs de texte [Text Fields]

Cet exemple montre le comportement supplémentaire dont sont capables les **JTextField**s :

```
//: c13:TextFields.java
// Champs de texte et événements Java.
// <applet code=TextFields width=375
// height=125></applet>
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.text.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

public class TextFields extends JApplet {
JButton
b1 = new JButton("Get Text"),
b2 = new JButton("Set Text");
JTextField
t1 = new JTextField(30),
t2 = new JTextField(30),
```

```
t3 = new JTextField(30);
String s = new String();
UpperCaseDocument
ucd = new UpperCaseDocument();
  public void init() {
t1.setDocument(ucd);
ucd.addDocumentListener(new T1());
b1.addActionListener(new B1());
b2.addActionListener(new B2());
DocumentListener dl = new T1();
t1.addActionListener(new T1A());
Container cp = getContentPane();
cp.setLayout(new FlowLayout());
cp.add(b1);
cp.add(b2);
cp.add(t1);
cp.add(t2);
cp.add(t3);
}
  class T1 implements DocumentListener {
   public void changedUpdate(DocumentEvent e){}
   public void insertUpdate(DocumentEvent e){
t2.setText(t1.getText());
t3.setText("Text: "+ t1.getText());
}
   public void removeUpdate(DocumentEvent e){
t2.setText(t1.getText());
}
}
  class T1A implements ActionListener {
   private int count = 0;
   public void actionPerformed(ActionEvent e) {
t3.setText("t1 Action Event " + count++);
}
}
  class B1 implements ActionListener {
   public void actionPerformed(ActionEvent e) {
    if(t1.getSelectedText() == null)
   s = t1.getText();
    else
   s = t1.getSelectedText();
t1.setEditable(true);
}
}
  class B2 implements ActionListener {
   public void actionPerformed(ActionEvent e) {
ucd.setUpperCase(false);
```

```
t1.setText("Inserted by Button 2: " + s);
ucd.setUpperCase(true);
t1.setEditable(false);
}
}
  public static void main(String[] args) {
Console.run(new TextFields(), 375, 125);
}
}

class UpperCaseDocument extends PlainDocument {
  boolean upperCase = true;
  public void setUpperCase(boolean flag) {
upperCase = flag;
}
  public void insertString(int offset,
String string, AttributeSet attributeSet)
    throws BadLocationException {
     if(upperCase)
    string = string.toUpperCase();
     super.insertString(offset,
    string, attributeSet);
}
} ///:~
```

Le **JTextField t3** est inclus pour servir d'emplacement pour signaler lorsque l'*action listener* du **JTextField t1** est lancé.On verra que l'action listener d'un **JTextField** n'est lancé que lorsqu'on appuie sur la touche enter.

Le **JTextField t1** a plusieurs *listener*s attachés. le *listener* T1 est un **Document Listener** qui répond à tout changement dans le document (le contenu du **JTextField**, dans ce cas). Il copie automatiquement tout le texte de **t1** dans **t2**. De plus, le document **t1** est positionné à une classe dérivée de **PlainDocument**, appelée **UpperCaseDocument**, qui force tous les caractères en majuscules. Il détecte automatiquement les retours en arrière *[backspaces]*et effectue l'effacement, tout en ajustant le curseur et gérant tout de la manière attendue.

Bordures

JComponent possède une méthode appelée **setBorder()**, qui permet de placer différentes bordures intéressantes sur tout composant visible. L'exemple suivant montre certaines des bordures existantes, en utilisant la méthode **showBorder()** qui crée un **JPanel** et lui attache une bordure à chaque fois. Il utilise aussi RTTI pour trouver le nom de la bordure qu'on utilise (en enlevant l'-information du chemin), et met ensuite le nom dans un **JLabel** au milieu du panneau :

```
//: c13:Borders.java
// Diverses bordures Swing.
// <applet code=Borders
//  width=500 height=300></applet>
```

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.border.*;
import com.bruceeckel.swing.*;

public class Borders extends JApplet {
  static JPanel showBorder(Border b) {
    JPanel jp = new JPanel();
    jp.setLayout(new BorderLayout());
    String nm = b.getClass().toString();
    nm = nm.substring(nm.lastIndexOf('.') + 1);
    jp.add(new JLabel(nm, JLabel.CENTER),
      BorderLayout.CENTER);
    jp.setBorder(b);
    return jp;
  }
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.setLayout(new GridLayout(2,4));
    cp.add(showBorder(new TitledBorder("Title")));
    cp.add(showBorder(new EtchedBorder()));
    cp.add(showBorder(new LineBorder(Color.blue)));
    cp.add(showBorder(
      new MatteBorder(5,5,30,30,Color.green)));
    cp.add(showBorder(
      new BevelBorder(BevelBorder.RAISED)));
    cp.add(showBorder(
      new SoftBevelBorder(BevelBorder.LOWERED)));
    cp.add(showBorder(new CompoundBorder(
      new EtchedBorder(),
      new LineBorder(Color.red))));
  }
  public static void main(String[] args) {
    Console.run(new Borders(), 500, 300);
  }
} ///:~
```

On peut également créer ses propres bordures et les placer dans des boutons, labels, et cetera, tout ce qui est dérivé de JComponent.

JScrollPanes

La plupart du temps on laissera le **JScrollPane** tel quel, mais on peut aussi contrôler quelles barres de défilement sont autorisées, verticales, horizontales, les deux ou ni l'une ni l'autre :

```
//: c13:JScrollPanes.java
```

```
// Contrôle des scrollbars d'un JScrollPane.
// <applet code=JScrollPanes width=300 height=725>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.border.*;
import com.bruceeckel.swing.*;

public class JScrollPanes extends JApplet {
  JButton
    b1 = new JButton("Text Area 1"),
    b2 = new JButton("Text Area 2"),
    b3 = new JButton("Replace Text"),
    b4 = new JButton("Insert Text");
  JTextArea
    t1 = new JTextArea("t1", 1, 20),
    t2 = new JTextArea("t2", 4, 20),
    t3 = new JTextArea("t3", 1, 20),
    t4 = new JTextArea("t4", 10, 10),
    t5 = new JTextArea("t5", 4, 20),
    t6 = new JTextArea("t6", 10, 10);
  JScrollPane
    sp3 = new JScrollPane(t3,
      JScrollPane.VERTICAL_SCROLLBAR_NEVER,
      JScrollPane.HORIZONTAL_SCROLLBAR_NEVER),
    sp4 = new JScrollPane(t4,
      JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
      JScrollPane.HORIZONTAL_SCROLLBAR_NEVER),
    sp5 = new JScrollPane(t5,
      JScrollPane.VERTICAL_SCROLLBAR_NEVER,
      JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS),
    sp6 = new JScrollPane(t6,
      JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
      JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS);
  class B1L implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      t5.append(t1.getText() + "\n");
    }
  }
  class B2L implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      t2.setText("Inserted by Button 2");
      t2.append(": " + t1.getText());
      t5.append(t2.getText() + "\n");
    }
  }
```

```
  class B3L implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      String s = " Replacement ";
      t2.replaceRange(s, 3, 3 + s.length());
    }
  }
  class B4L implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      t2.insert(" Inserted ", 10);
    }
  }
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    // Création de Borders pour les composants:
    Border brd = BorderFactory.createMatteBorder(
      1, 1, 1, 1, Color.black);
    t1.setBorder(brd);
    t2.setBorder(brd);
    sp3.setBorder(brd);
    sp4.setBorder(brd);
    sp5.setBorder(brd);
    sp6.setBorder(brd);
    // Initialisation des listeners et ajout des composants:
    b1.addActionListener(new B1L());
    cp.add(b1);
    cp.add(t1);
    b2.addActionListener(new B2L());
    cp.add(b2);
    cp.add(t2);
    b3.addActionListener(new B3L());
    cp.add(b3);
    b4.addActionListener(new B4L());
    cp.add(b4);
    cp.add(sp3);
    cp.add(sp4);
    cp.add(sp5);
    cp.add(sp6);
  }
  public static void main(String[] args) {
    Console.run(new JScrollPanes(), 300, 725);
  }
} ///:~
```

L'utilisation des différents arguments du constructeur de **JScrollPane** contrôle la présence des scrollbars. Cet exemple est également un peu "habillé" à l'aide de bordures.

Un mini-éditeur

Le contrôle **JTextPane** fournit un support important pour l'édition de texte, sans grand effort. L'exemple suivant en fait une utilisation très simple, en ignorant le plus gros des fonctionnalités de la classe :

```
//: c13:TextPane.java
// Le contrôle JTextPane est un petit éditeur de texte.
// <applet code=TextPane width=475 height=425>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;
import com.bruceeckel.util.*;

public class TextPane extends JApplet {
  JButton b = new JButton("Add Text");
  JTextPane tp = new JTextPane();
  static Generator sg =
    new Arrays2.RandStringGenerator(7);
  public void init() {
    b.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e){
        for(int i = 1; i < 10; i++)
          tp.setText(tp.getText() + sg.next() + "\n");
      }
    });
    Container cp = getContentPane();
    cp.add(new JScrollPane(tp));
    cp.add(BorderLayout.SOUTH, b);
  }
  public static void main(String[] args) {
    Console.run(new TextPane(), 475, 425);
  }
} ///:~
```

Le bouton ajoute simplement au hasard du texte généré. Le but du **JTextPane** est de permettre la modification de texte sur place, de sorte qu'on ne trouvera pas de méthode **append()**. Dans ce cas-ci (il est admis qu'il s'agit d'un piètre usage des capacités de **JTextPane**), le texte doit être saisi, modifié, et replacé dans le panneau en utilisant **setText()**.

Remarquez les fonctionnalités intrinsèques du **JTextPane**, telles que le retour à la ligne automatique. Il y a de nombreuses autres fonctionnalités à découvrir dans la documentation du JDK.

Boîtes à cocher *[Check boxes]*

Une boîte à cocher *[check box]* permet d'effectuer un choix simple vrai/faux ; il consiste en une petite boîte et un label. La boîte contient d'habitude un petit x (ou tout autre moyen d'indiquer

qu'elle est cochée) ou est vide, selon qu'elle a été sélectionnée ou non.

On crée normalement une **JCheckBox** en utilisant un constructeur qui prend le label comme argument. On peut obtenir ou forcer l'état, et également obtenir ou forcer le label si on veut le lire ou le modifier après la création de la **JCheckBox**.

Chaque fois qu'une **JCheckBox** est remplie ou vidée, un événement est généré, qu'on peut capturer de la même façon que pour un bouton, en utilisant un **ActionListener**. L'exemple suivant utilise un **JTextArea** pour lister toutes les boîtes à cocher qui ont été cochées :

```
//: c13:CheckBoxes.java
// Utilisation des JCheckBoxes.
// <applet code=CheckBoxes width=200 height=200>
// </applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class CheckBoxes extends JApplet {
  JTextArea t = new JTextArea(6, 15);
  JCheckBox
    cb1 = new JCheckBox("Check Box 1"),
    cb2 = new JCheckBox("Check Box 2"),
    cb3 = new JCheckBox("Check Box 3");
  public void init() {
    cb1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e){
        trace("1", cb1);
      }
    });
    cb2.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e){
        trace("2", cb2);
      }
    });
    cb3.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e){
        trace("3", cb3);
      }
    });
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(new JScrollPane(t));
    cp.add(cb1);
    cp.add(cb2);
    cp.add(cb3);
  }
  void trace(String b, JCheckBox cb) {
```

```
      if(cb.isSelected())
        t.append("Box " + b + " Set\n");
      else
        t.append("Box " + b + " Cleared\n");
    }
    public static void main(String[] args) {
      Console.run(new CheckBoxes(), 200, 200);
    }
} ///:~
```

La méthode **trace()** envoie le nom et l'état de la **JCheckBox** sélectionnée au **JTextArea** en utilisant **append()**, de telle sorte qu'on voit une liste cumulative des boîtes à cocher qui ont été sélectionnées, et quel est leur état.

Boutons radio

Le concept d'un bouton radio en programmation de GUI provient des autoradios d'avant l'ère électronique, avec des boutons mécaniques : quand on appuie sur l'un d'eux, tout autre bouton enfoncé est relâché. Ceci permet de forcer un choix unique parmi plusieurs.

Pour installer un groupe de **JRadioButton**s, il suffit de les ajouter à un **ButtonGroup** (il peut y avoir un nombre quelconque de **ButtonGroup**s dans un formulaire). En utilisant le second argument du constructeur, on peut optionnellement forcer à **true** l'état d'un des boutons. Si on essaie de forcer à **true** plus d'un bouton radio, seul le dernier forcé sera à **true**.

Voici un exemple simple d'utilisation de boutons radio. On remarquera que les événement des boutons radio s'interceptent comme tous les autres :

```
//: c13:RadioButtons.java
// Utilisation des JRadioButtons.
// <applet code=RadioButtons
// width=200 height=100> </applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class RadioButtons extends JApplet {
  JTextField t = new JTextField(15);
  ButtonGroup g = new ButtonGroup();
  JRadioButton
    rb1 = new JRadioButton("one", false),
    rb2 = new JRadioButton("two", false),
    rb3 = new JRadioButton("three", false);
  ActionListener al = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      t.setText("Radio button " +
        ((JRadioButton)e.getSource()).getText());
    }
```

```
  };
  public void init() {
    rb1.addActionListener(al);
    rb2.addActionListener(al);
    rb3.addActionListener(al);
    g.add(rb1); g.add(rb2); g.add(rb3);
    t.setEditable(false);
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(t);
    cp.add(rb1);
    cp.add(rb2);
    cp.add(rb3);
  }
  public static void main(String[] args) {
    Console.run(new RadioButtons(), 200, 100);
  }
} ///:~
```

Pour afficher l'état un champ texte est utilisé. Ce champ est déclaré non modifiable car il est utilisé uniquement pour afficher des données et pas pour en recevoir. C'est une alternative à l'utilisation d'un **JLabel**.

Boîtes combo (listes à ouverture vers le bas) *[combo boxes (drop-down lists)]*

Comme les groupes de boutons radio, une *drop-down list* est une façon de forcer l'utilisateur à choisir un seul élément parmi un groupe de possibilités. C'est cependant un moyen plus compact, et il est plus facile de modifier les éléments de la liste sans surprendre l'utilisateur (on peut modifier dynamiquement les boutons radio, mais ça peut devenir visuellement perturbant).

La **JComboBox** java n'est pas comme la combo box de Windows qui permet de sélectionner dans une liste *ou* de taper soi-même une sélection. Avec une **JComboBox** on choisit un et un seul élément de la liste. Dans l'exemple suivant, la **JComboBox** démarre avec un certain nombre d'éléments et ensuite des éléments sont ajoutés lorsqu'on appui sur un bouton.

```
//: c13:ComboBoxes.java
// Utilisation des drop-down lists.
// <applet code=ComboBoxes
// width=200 height=100> </applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class ComboBoxes extends JApplet {
  String[] description = { "Ebullient", "Obtuse",
    "Recalcitrant", "Brilliant", "Somnescent",
    "Timorous", "Florid", "Putrescent" };
```

```
  JTextField t = new JTextField(15);
  JComboBox c = new JComboBox();
  JButton b = new JButton("Add items");
  int count = 0;
  public void init() {
    for(int i = 0; i < 4; i++)
      c.addItem(description[count++]);
    t.setEditable(false);
    b.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e){
        if(count < description.length)
          c.addItem(description[count++]);
      }
    });
    c.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e){
        t.setText("index: "+ c.getSelectedIndex()
          + "   " + ((JComboBox)e.getSource())
          .getSelectedItem());
      }
    });
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(t);
    cp.add(c);
    cp.add(b);
  }
  public static void main(String[] args) {
    Console.run(new ComboBoxes(), 200, 100);
  }
} ///:~
```

Le **JTextField** affiche l'index sélectionné, qui est le numéro séquentiel de l'élément sélectionné, ainsi que le label du bouton radio.

Listes *[List boxes]*

Les listes sont différentes des **JComboBox,** et pas seulement en apparence. Alors qu'une **JComboBox** s'affiche vers le bas lorsqu'on l'active, une **JList** occupe un nombre fixe de lignes sur l'écran tout le temps et ne se modifie pas. Si on veut voir les éléments de la liste, il suffit d'appeler **getSelectedValues(),** qui retourne un tableau de **String** des éléments sélectionnés.

Une **JList** permet la sélection multiple : si on "control-clique" sur plus d'un élément (en enfonçant la touche control tout en effectuant des clics souris) l'élément initial reste surligné et on peut en sélectionner autant qu'on veut. Si on sélectionne un élément puis qu'on en "shift-clique" un autre, tous les éléments entre ces deux-là seront aussi sélectionnés. Pour supprimer un élément d'un groupe on peut le "control-cliquer".

```
//: c13:List.java
// <applet code=List width=250
// height=375> </applet>
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.border.*;
import com.bruceeckel.swing.*;

public class List extends JApplet {
  String[] flavors = { "Chocolate", "Strawberry",
    "Vanilla Fudge Swirl", "Mint Chip",
    "Mocha Almond Fudge", "Rum Raisin",
    "Praline Cream", "Mud Pie" };
  DefaultListModel lItems=new DefaultListModel();
  JList lst = new JList(lItems);
  JTextArea t = new JTextArea(flavors.length,20);
  JButton b = new JButton("Add Item");
  ActionListener bl = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      if(count  < flavors.length) {
        lItems.add(0, flavors[count++]);
      } else {
        // Invalider, puisqu'il n'y a plus
        // de parfums a ajouter a la liste
        b.setEnabled(false);
      }
    }
  };
  ListSelectionListener ll =    new ListSelectionListener() {
      public void valueChanged(
        ListSelectionEvent e) {
          t.setText("");
          Object[] items=lst.getSelectedValues();
          for(int i = 0; i < t.append(items[i] + "\n");
        }
    };
  int count = 0;
  public void init() {
    Container cp = getContentPane();
    t.setEditable(false);
    cp.setLayout(new FlowLayout());
    // Création de Bords pour les composants:
    Border brd = BorderFactory.createMatteBorder(
      1, 1, 2, 2, Color.black);
    lst.setBorder(brd);
```

```
    t.setBorder(brd);
    // Ajout des quatre premiers élément à la liste
    for(int i = 0; i < 4; i++)
      lItems.addElement(flavors[count++]);
    // Ajout des éléments au Content Pane pour affichage
    cp.add(t);
    cp.add(lst);
    cp.add(b);
    // Enregistrement des listeners d'événements
    lst.addListSelectionListener(ll);
    b.addActionListener(bl);
  }
  public static void main(String[] args) {
    Console.run(new List(), 250, 375);
  }
} ///:~
```

Quand on appuie sur le bouton il ajoute des élément au *début* de la liste (car le second argument de **addItem()** est 0).

On peut également voir qu'une bordure a été ajoutée aux listes.

Si on veut simplement mettre un tableau de **String**s dans une **JList**, il y a une solution beaucoup plus simple : passer le tableau au constructeur de la **JList**, et il construit la liste automatiquement. La seule raison d'utiliser le modèle de liste de l'exemple ci-dessus est que la liste peut être manipulée lors de l'exécution du programme.

Les **JList**s ne fournissent pas de support direct pour le scrolling. Bien évidemment, il suffit d'encapsuler la **JList** dans une **JScrollPane** et tous les détails sont automatiquement gérés.

Panneaux à tabulations *[Tabbed panes]*

Les **JTabbedPane** permettent de créer un dialogue tabulé, avec sur un côté des tabulations semblables à celles de classeurs de fiches, permettant d'amener au premier plan un autre dialogue en cliquant sur l'une d'entre elles.

```
//: c13:TabbedPane1.java
// Démonstration de Tabbed Pane.
// <applet code=TabbedPane1
// width=350 height=200> </applet>
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class TabbedPane1 extends JApplet {
  String[] flavors = { "Chocolate", "Strawberry",
    "Vanilla Fudge Swirl", "Mint Chip",
    "Mocha Almond Fudge", "Rum Raisin",
```

```
    "Praline Cream", "Mud Pie" };
  JTabbedPane tabs = new JTabbedPane();
  JTextField txt = new JTextField(20);
  public void init() {
    for(int i = 0; i  < flavors.length; i++)
      tabs.addTab(flavors[i],
        new JButton("Tabbed pane " + i));
    tabs.addChangeListener(new ChangeListener(){
      public void stateChanged(ChangeEvent e) {
        txt.setText("Tab selected: " +
          tabs.getSelectedIndex());
      }
    });
    Container cp = getContentPane();
    cp.add(BorderLayout.SOUTH, txt);
    cp.add(tabs);
  }
  public static void main(String[] args) {
    Console.run(new TabbedPane1(), 350, 200);
  }
} ///:~
```

En Java, l'utilisation d'un mécanisme de panneaux à tabulations est important car, pour la programmation d'applets, l'utilisation de dialogues *pop-ups* est découragé par l'apparition automatique d'un petit avertissement à chaque dialogue qui surgit d'une applet.

Lors de l'exécution de ce programme on remarquera que le **JTabbedPane** empile automatiquement les tabulations s'il y en a trop pour une rangée. On peut s'en apercevoir en redimensionnant la fenêtre lorsque le programme est lancé depuis la ligne de commande.

Boîtes de messages

Les environnements de fenêtrage comportent classiquement un ensemble standard de boîtes de messages qui permettent d'afficher rapidement une information à l'utilisateur, ou lui demander une information. Dans Swing, ces boîtes de messages sont contenues dans les **JOptionPane**s. Il y a de nombreuses possibilités (certaines assez sophistiquées), mais celles qu'on utilise le plus couramment sont les messages et les confirmations, appelées en utilisant **static JOptionPane.showMessageDialog()** et **JOptionPane.showConfirmDialog()**. L'exemple suivant montre un sous-ensemble des boîtes de messages disponibles avec **JOptionPane** :

```
//: c13:MessageBoxes.java
// Démonstration de JoptionPane.
// <applet code=MessageBoxes
// width=200 height=150> </applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;
```

```
public class MessageBoxes extends JApplet {
  JButton[] b = { new JButton("Alert"),
    new JButton("Yes/No"), new JButton("Color"),
    new JButton("Input"), new JButton("3 Vals")
  };
  JTextField txt = new JTextField(15);
  ActionListener al = new ActionListener() {
    public void actionPerformed(ActionEvent e){
      String id =
        ((JButton)e.getSource()).getText();
      if(id.equals("Alert"))
        JOptionPane.showMessageDialog(null,
          "There's a bug on you!", "Hey!",
          JOptionPane.ERROR_MESSAGE);
      else if(id.equals("Yes/No"))
        JOptionPane.showConfirmDialog(null,
          "or no", "choose yes",
          JOptionPane.YES_NO_OPTION);
      else if(id.equals("Color")) {
        Object[] options = { "Red", "Green" };
        int sel = JOptionPane.showOptionDialog(
          null, "Choose a Color!", "Warning",
          JOptionPane.DEFAULT_OPTION,
          JOptionPane.WARNING_MESSAGE, null,
          options, options[0]);
          if(sel != JOptionPane.CLOSED_OPTION)
            txt.setText(
              "Color Selected: " + options[sel]);
      } else if(id.equals("Input")) {
        String val = JOptionPane.showInputDialog(
            "How many fingers do you see?");
        txt.setText(val);
      } else if(id.equals("3 Vals")) {
        Object[] selections = {
          "First", "Second", "Third" };
        Object val = JOptionPane.showInputDialog(
          null, "Choose one", "Input",
          JOptionPane.INFORMATION_MESSAGE,
          null, selections, selections[0]);
        if(val != null)
          txt.setText(
            val.toString());
      }
    }
  };
  public void init() {
```

```
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    for(int i = 0; i  < b.length; i++) {
     b[i].addActionListener(al);
      cp.add(b[i]);
    }
    cp.add(txt);
  }
  public static void main(String[] args) {
    Console.run(new MessageBoxes(), 200, 200);
  }
} ///:~
```

On remarquera que **showOptionDialog()** et **showInputDialog()** retournent des objets contenant la valeur entrée par l'utilisateur.

Menus

Chaque composant capable de contenir un menu, y compris **JApplet**, **JFrame**, **JDialog** et leurs descendants, possède une méthode **setJMenuBar()** qui prend comme paramètre un **JMenuBar** (il ne peut y avoir qu'un seul **JMenuBar** sur un composant donné). On ajoute les **JMenu**s au **JMenuBar**, et les **JMenuItem**s aux **JMenu**s. On peut attacher un **ActionListener** à chaque **JMenuItem**, qui sera lancé quand l'élément de menu est sélectionné.

Contrairement à un système qui utilise des ressources, en Java et Swing il faut assembler à la main tous les menus dans le code source. Voici un exemple très simple de menu :

```
//: c13:SimpleMenus.java
// <applet code=SimpleMenus
// width=200 height=75> </applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class SimpleMenus extends JApplet {
  JTextField t = new JTextField(15);
  ActionListener al = new ActionListener() {
    public void actionPerformed(ActionEvent e){
      t.setText(
        ((JMenuItem)e.getSource()).getText());
    }
  };
  JMenu[] menus = { new JMenu("Winken"),
    new JMenu("Blinken"), new JMenu("Nod") };
  JMenuItem[] items = {
    new JMenuItem("Fee"), new JMenuItem("Fi"),
    new JMenuItem("Fo"),  new JMenuItem("Zip"),
```

```
    new JMenuItem("Zap"), new JMenuItem("Zot"),
    new JMenuItem("Olly"), new JMenuItem("Oxen"),
    new JMenuItem("Free") };
  public void init() {
    for(int i = 0; i < items.length; i++) {
      items[i].addActionListener(al);
      menus[i%3].add(items[i]);
    }
    JMenuBar mb = new JMenuBar();
    for(int i = 0; i < menus.length; i++)
      mb.add(menus[i]);
    setJMenuBar(mb);
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(t);
  }
  public static void main(String[] args) {
    Console.run(new SimpleMenus(), 200, 75);
  }
} ///:~
```

L'utilisation de l'opérateur modulo *[modulus]* dans **i%3** distribue les éléments de menus parmi les trois **JMenu**s. Chaque **JMenuItem** doit avoir un ActionListener attaché ; ici le même **ActionListener** est utilisé partout mais on en aura besoin normalement d'un pour chaque **JMenuItem**.

JMenuItem hérite d'**AbstractButton**, et il a donc certains comportements des boutons. En lui-même, il fournit un élément qui peut être placé dans un menu déroulant. Il y a aussi trois types qui héritent de **JMenuItem** : **JMenu** pour contenir d'autres **JMenuItem**s (pour réaliser des menus en cascade), **JCheckBoxMenuItem**, qui fournit un marquage pour indiquer si l'élément de menu est sélectionné ou pas, et **JRadioButtonMenuItem**, qui contient un bouton radio.

En tant qu'exemple plus sophistiqué, voici à nouveau les parfums de crèmes glacées, utilisés pour créer des menus. Cet exemple montre également des menus en cascade, des mnémoniques clavier, des **JCheckBoxMenuItem**s, et la façon de changer ces menus dynamiquement :

```
//: c13:Menus.java
// Sous-menus, éléments de menu avec boîtes à cocher, permutations de  menus,
// mnémoniques (raccourcis) et commandes d'actions.
// <applet code=Menus width=300
// height=100> </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

public class Menus extends JApplet {
  String[] flavors = { "Chocolate", "Strawberry",
    "Vanilla Fudge Swirl", "Mint Chip",
```

```
    "Mocha Almond Fudge", "Rum Raisin",
    "Praline Cream", "Mud Pie" };
  JTextField t = new JTextField("No flavor", 30);
  JMenuBar mb1 = new JMenuBar();
  JMenu
    f = new JMenu("File"),
    m = new JMenu("Flavors"),
    s = new JMenu("Safety");
  // Approche alternative :
  JCheckBoxMenuItem[] safety = {
    new JCheckBoxMenuItem("Guard"),
    new JCheckBoxMenuItem("Hide")
  };
  JMenuItem[] file = {
    new JMenuItem("Open"),
  };
  // Une seconde barre de menu pour échanger :
  JMenuBar mb2 = new JMenuBar();
  JMenu fooBar = new JMenu("fooBar");
  JMenuItem[] other = {
    // Ajouter un raccourci de menu (mnémonique) est très
    // simple, mais seuls les JMenuItems peuvent les avoir
    // dans leurs constructeurs:
    new JMenuItem("Foo", KeyEvent.VK_F),
    new JMenuItem("Bar", KeyEvent.VK_A),
    // Pas de raccourci :
    new JMenuItem("Baz"),
  };
  JButton b = new JButton("Swap Menus");
  class BL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      JMenuBar m = getJMenuBar();
      setJMenuBar(m == mb1 ? mb2 : mb1);
      validate(); // Rafraîchissement de la fenêtre
    }
  }
  class ML implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      JMenuItem target = (JMenuItem)e.getSource();
      String actionCommand =
        target.getActionCommand();
      if(actionCommand.equals("Open")) {
        String s = t.getText();
        boolean chosen = false;
        for(int i = 0; i < flavors.length; i++)
          if(s.equals(flavors[i])) chosen = true;
        if(!chosen)
```

```
            t.setText("Choose a flavor first!");
          else
            t.setText("Opening "+ s +". Mmm, mm!");
        }
      }
    }
    class FL implements ActionListener {
      public void actionPerformed(ActionEvent e) {
        JMenuItem target = (JMenuItem)e.getSource();
        t.setText(target.getText());
      }
    }
    // Alternativement, on peut créer une classe
  // différente pour chaque JMenuItem. Ensuite
  // il n'est plus nécessaire de rechercher de laquelle il s'agit :
    class FooL implements ActionListener {
      public void actionPerformed(ActionEvent e) {
        t.setText("Foo selected");
      }
    }
    class BarL implements ActionListener {
      public void actionPerformed(ActionEvent e) {
        t.setText("Bar selected");
      }
    }
    class BazL implements ActionListener {
      public void actionPerformed(ActionEvent e) {
        t.setText("Baz selected");
      }
    }
    class CMIL implements ItemListener {
      public void itemStateChanged(ItemEvent e) {
        JCheckBoxMenuItem target =
          (JCheckBoxMenuItem)e.getSource();
        String actionCommand =
          target.getActionCommand();
        if(actionCommand.equals("Guard"))
          t.setText("Guard the Ice Cream! " +
            "Guarding is " + target.getState());
        else if(actionCommand.equals("Hide"))
          t.setText("Hide the Ice Cream! " +
            "Is it cold? " + target.getState());
      }
    }
    public void init() {
      ML ml = new ML();
      CMIL cmil = new CMIL();
```

```
    safety[0].setActionCommand("Guard");
    safety[0].setMnemonic(KeyEvent.VK_G);
    safety[0].addItemListener(cmil);
    safety[1].setActionCommand("Hide");
    safety[0].setMnemonic(KeyEvent.VK_H);
    safety[1].addItemListener(cmil);
    other[0].addActionListener(new FooL());
    other[1].addActionListener(new BarL());
    other[2].addActionListener(new BazL());
    FL fl = new FL();
    for(int i = 0; i < flavors.length; i++) {
      JMenuItem mi = new JMenuItem(flavors[i]);
      mi.addActionListener(fl);
      m.add(mi);
      // Ajout de séparateurs par  intervalles:
      if((i+1) % 3 == 0)
        m.addSeparator();
    }
    for(int i = 0; i  < safety.length; i++)
      s.add(safety[i]);
    s.setMnemonic(KeyEvent.VK_A);
    f.add(s);
    f.setMnemonic(KeyEvent.VK_F);
    for(int i = 0; i < file.length; i++) {
      file[i].addActionListener(fl);
      f.add(file[i]);
    }
    mb1.add(f);
    mb1.add(m);
    setJMenuBar(mb1);
    t.setEditable(false);
    Container cp = getContentPane();
    cp.add(t, BorderLayout.CENTER);
    // Installation du système d'échange de menus:
    b.addActionListener(new BL());
    b.setMnemonic(KeyEvent.VK_S);
    cp.add(b, BorderLayout.NORTH);
    for(int i = 0; i < other.length; i++)
      fooBar.add(other[i]);
    fooBar.setMnemonic(KeyEvent.VK_B);
    mb2.add(fooBar);
  }
  public static void main(String[] args) {
    Console.run(new Menus(), 300, 100);
  }
} ///:~
```

Dans ce programme j'ai placé les éléments de menus dans des tableaux et ensuite j'ai parcouru chaque tableau en appelant **add()** pour chaque **JMenuItem**. Ceci rend l'ajout ou la suppression d'un élément de menu un peu moins fastidieux.

Ce programme crée deux **JMenuBar**s pour démontrer que les barres de menu peuvent être échangées dynamiquement à l'exécution du programme. On peut voir qu'un **JMenuBar** est composé de **JMenu**s, et que chaque **JMenu** est composé de **JMenuItem**s, **JCheckBoxMenuItem**s, ou même d'autres **JMenu**s (qui produisent des sous-menus). Une fois construit un **JMenuBar**, il peut être installé dans le programme courant avec la méthode **setJMenuBar()**. Remarquons que lorsque le bouton est cliqué, il regarde quel menu est installé en appelant **getJMenuBar()**, et à ce moment il le remplace par l'autre barre de menu.

Lorsqu'on teste "Open", il faut remarquer que l'orthographe et les majuscules/minuscules sont cruciaux, et que Java ne signale pas d'erreur s'il n'y a pas correspondance avec "Open". Ce genre de comparaison de chaînes est une source d'erreurs de programmation.

Le cochage et le décochage des éléments de menus est pris en compte automatiquement. Le code gérant les **JCheckBoxMenuItem**s montre deux façons différentes de déterminer ce qui a été coché : la correspondance des chaînes (qui, comme mentionné ci-dessus, n'est pas une approche très sûre bien qu'on la rencontre) et la correspondance de l'objet cible de l'événement. Il montre aussi que la méthode **getState()** peut être utilisée pour connaître son état. On peut également changer l'état d'un **JCheckBoxMenuItem** à l'aide de **setState()**.

Les événements des menus sont un peu inconsistants et peuvent prêter à confusion : les **JMenuItem**s utilisent des **ActionListener**s, mais les **JCheckboxMenuItem**s utilisent des **ItemListener**s. Les objets **JMenu** peuvent aussi supporter des **ActionListener**s, mais ce n'est généralement pas très utile. En général, on attache des *listeners* à chaque **JMenuItem**, **JCheckBoxMenuItem**, ou **JRadioButtonMenuItem**, mais l'exemple montre des **ItemListener**s et des **ActionListener**s attachés aux divers composants de menus.

Swing supporte les mnémoniques, ou raccourcis clavier, de sorte qu'on peut sélectionner tout ce qui est dérivé de **AbstractButton** (bouton, menu, élément, et cetera) en utilisant le clavier à la place de la souris. C'est assez simple : pour le **JMenuItem** on peut utiliser le constructeur surchargé qui prend en deuxième argument l'identificateur de la touche. Cependant, la plupart des **AbstractButton**s n'ont pas ce constructeur; une manière plus générale de résoudre ce problème est d'utiliser la méthode **setMnemonic()**. L'exemple ci-dessus ajoute une mnémonique au bouton et à certains des éléments de menus : les indicateurs de raccourcis apparaissent automatiquement sur les composants.

On peut aussi voir l'utilisation de **setActionCommand()**. Ceci paraît un peu bizarre car dans chaque cas la commande d'action *[action command]* est exactement la même que le label sur le composant du menu. Pourquoi ne pas utiliser simplement le label plutôt que cette chaîne de remplacement ? Le problème est l'internationalisation. Si on réoriente ce programme vers une autre langue, on désire changer uniquement le label du menu, et pas le code (ce qui introduirait à coup sûr d'autres erreurs). Donc pour faciliter ceci pour les codes qui testent la chaîne associée à un composant de menu, la commande d'action peut être invariante tandis que le label du menu peut changer. Tout le code fonctionne avec la commande d'action, de sorte qu'il n'est pas touché par les modifications des labels des menus. Remarquons que dans ce programme on ne recherche pas des commandes d'actions pour tous les composants de menus, de sorte que ceux qui ne sont pas examinés n'ont pas de commande d'action positionnée.

La plus grosse partie du travail se trouve dans les *listeners*. **BL** effectue l'échange des **JMe-**

nuBars. Dans **ML**, l'approche du "qui a sonné ?" est utilisée en utilisant la source de l'**ActionEvent** et en l'émettant vers un **JMenuItem**, en faisant passer la chaîne de la commande d'action à travers une instruction **if** en cascade.

Le *listener* **FL** est simple, bien qu'il gère les différents parfums du menu parfums. Cette approche est utile si la logique est simple, mais en général, on utilisera l'approche de **FooL**, **BarL** et **BazL**, dans lesquels ils sont chacun attaché à un seul composant de menu de sorte qu'il n'est pas nécessaire d'avoir une logique de détection supplémentaire, et on sait exactement qui a appelé le *listener*. Même avec la profusion de classes générées de cette façon, le code interne tend à être plus petit et le traitement est plus fiable.

On peut voir que le code d'un menu devient rapidement long et désordonné. C'est un autre cas où l'utilisation d'un *GUI builder* est la solution appropriée. Un bon outil gérera également la maintenance des menus.

Menus pop-up

La façon la plus directe d'implémenter un **JPopupMenu** est de créer une classe interne qui étend **MouseAdapter**, puis d'ajouter un objet de cette classe interne à chaque composant pour lequel on veut créer le pop-up :

```
//: c13:Popup.java
// Création de menus popup avec Swing.
// <applet code=Popup
//  width=300 height=200></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

public class Popup extends JApplet {
  JPopupMenu popup = new JPopupMenu();
  JTextField t = new JTextField(10);
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(t);
    ActionListener al = new ActionListener() {
      public void actionPerformed(ActionEvent e){
        t.setText(
          ((JMenuItem)e.getSource()).getText());
      }
    };
    JMenuItem m = new JMenuItem("Hither");
    m.addActionListener(al);
    popup.add(m);
    m = new JMenuItem("Yon");
    m.addActionListener(al);
    popup.add(m);
```

```
    m = new JMenuItem("Afar");
    m.addActionListener(al);
    popup.add(m);
    popup.addSeparator();
    m = new JMenuItem("Stay Here");
    m.addActionListener(al);
    popup.add(m);
    PopupListener pl = new PopupListener();
    addMouseListener(pl);
    t.addMouseListener(pl);
  }
  class PopupListener extends MouseAdapter {
    public void mousePressed(MouseEvent e) {
      maybeShowPopup(e);
    }
    public void mouseReleased(MouseEvent e) {
      maybeShowPopup(e);
    }
    private void maybeShowPopup(MouseEvent e) {
      if(e.isPopupTrigger()) {
        popup.show(
          e.getComponent(), e.getX(), e.getY());
      }
    }
  }
  public static void main(String[] args) {
    Console.run(new Popup(), 300, 200);
  }
} ///:~
```

Le même **ActionListener** est ajouté à chaque **JMenuItem** de façon à prendre le texte du label du menu et l'insérer dans le **JTextField**.

Dessiner

Dans un bon outil de GUI, dessiner devrait être assez facile, et ça l'est dans la bibliothèque Swing. Le problème de tout exemple de dessin est que les calculs qui déterminent où vont les éléments sont souvent beaucoup plus compliqués que les appels aux sous-programmes de dessin, et que ces calculs sont souvent mélangés aux appels de dessin, de sorte que l'interface semble plus compliquée qu'elle ne l'est.

Pour simplifier, considérons le problème de la représentation de données sur l'écran. Ici, les données sont fournies par la méthode intrinsèque **Math.sin()** qui est la fonction mathématique sinus. Pour rendre les choses un peu plus intéressantes, et pour mieux montrer combien il est facile d'utiliser les composants Swing, un curseur sera placé en bas du formulaire pour contrôler dynamiquement le nombre de cycles du sinus affiché. De plus, si on redimensionne la fenêtre, on verra le sinus s'adapter de lui-même à la nouvelle taille de la fenêtre.

Dans l'exemple suivant, toute l'intelligence concernant le dessin est contenue dans la classe

SineDraw; la classe **SineWave** configure simplement le programme et le curseur. Dans **SineDraw**, la méthode **setCycles()** fournit un moyen pour permettre à un autre objet (le curseur dans ce cas) de contrôler le nombre de cycles.

```
//: c13:SineWave.java
// Dessin avec Swing, en utilisant un JSlider.
// <applet code=SineWave
//  width=700 height=400></applet>
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;

class SineDraw extends JPanel {
  static final int SCALEFACTOR = 200;
  int cycles;
  int points;
  double[] sines;
  int[] pts;
  SineDraw() { setCycles(5); }
  public void setCycles(int newCycles) {
    cycles = newCycles;
    points = SCALEFACTOR * cycles * 2;
    sines = new double[points];
    pts = new int[points];
    for(int i = 0; i < points; i++) {
      double radians = (Math.PI/SCALEFACTOR) * i;
      sines[i] = Math.sin(radians);
    }
    repaint();
  }
  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    int maxWidth = getWidth();
    double hstep = (double)maxWidth/(double)points;
    int maxHeight = getHeight();
    for(int i = 0; i < points; i++) {
      pts[i] = (int)(sines[i] * maxHeight/2 * .95
                + maxHeight/2);
    g.setColor(Color.red);
    for(int i = 1; i < points; i++) {
      int x1 = (int)((i - 1) * hstep);
      int x2 = (int)(i * hstep);
      int y1 = pts[i-1];
      int y2 = pts[i];
      g.drawLine(x1, y1, x2, y2);
    }
```

```
    }
  }

  public class SineWave extends JApplet {
    SineDraw sines = new SineDraw();
    JSlider cycles = new JSlider(1, 30, 5);
    public void init() {
      Container cp = getContentPane();
      cp.add(sines);
      cycles.addChangeListener(new ChangeListener(){
        public void stateChanged(ChangeEvent e) {
          sines.setCycles(
            ((JSlider)e.getSource()).getValue());
        }
      });
      cp.add(BorderLayout.SOUTH, cycles);
    }
    public static void main(String[] args) {
      Console.run(new SineWave(), 700, 400);
    }
  } ///:~
```

Tous les membres de données et tableaux sont utilisés dans le calcul des points du sinus : **cycles** indique le nombre de périodes complètes de sinus désiré, **points** contient le nombre total de points qui sera tracé, **sines** contient les valeurs de la fonction sinus, et **pts** contient les coordonnées y des points qui seront tracés sur le **JPanel**. La méthode **setCycles()** construit le tableau selon le nombre de points nécessaires et remplit le tableau **sines** de valeurs. En appelant **repaint()**, **setCycles** force l'appel de **paintComponent()** , afin que le reste des calculs et le dessin aient lieu.

La première chose à faire lorsqu'on redéfinit **paintComponent()** est d'appeler la version de base de la méthode. Ensuite on peut faire ce que l'on veut; normalement cela signifie utiliser les méthodes de **Graphics** qu'on peut trouver dans la documentation de **java.awt.Graphics** (dans la documentation HTML de *java.sun.com*) pour dessiner et peindre des pixels sur le **JPanel**. On peut voir ici que la plupart du code concerne l'exécution des calculs, les deux seules méthodes qui manipulent effectivement l'écran sont **setColor()** et **drawLine()**. Vous aurez probablement la même sensation lorsque vous créerez votre propre programme d'affichage de données graphiques : vous passerez la plus grande partie du temps à déterminer ce qu'il faut dessiner, mais le dessin en lui-même sera assez simple.

Lorsque j'ai créé ce programme, j'ai passé le plus gros de mon temps à obtenir la courbe du sinus à afficher. Ceci fait, j'ai pensé que ce serait bien de pouvoir modifier dynamiquement le nombre de cycles. Mes expériences de programmation de ce genre de choses dans d'autres langages me rendaient un peu réticent, mais cette partie se révéla la partie la plus facile du projet. J'ai créé un **JSlider** (les arguments sont respectivement la valeur de gauche du **JSlider**, la valeur de droite, et la valeur initiale, mais il existe d'autres constructeurs) et je l'ai déposé dans le **JApplet**. Ensuite j'ai regardé dans la documentation HTML et j'ai vu que le seul *listener* était le **addChangeListener**, qui était déclenché chaque fois que le curseur était déplacé suffisamment pour produire une nouvelle valeur. La seule méthode pour cela était évidemment appelée **stateChanged()**, qui fournit un objet **ChangeEvent**, de manière à pouvoir rechercher la source de la modification et obtenir la nouvelle

valeur.En appelant **setCycles()** des objets **sines**, la nouvelle valeur est prise en compte et le **JPanel** est redessiné.

En général, on verra que la plupart des problèmes Swing peuvent être résolus en suivant un processus semblable, et on verra qu'il est en général assez simple, même si on n'a pas utilisé auparavant un composant donné.

Si le problème est plus compliqué, il y a d'autres solutions plus sophistiquées, par exemple les composants JavaBeans de fournisseurs tiers, et l'API Java 2D. Ces solutions sortent du cadre de ce livre, mais vous devriez les prendre en considération si votre code de dessin devient trop coûteux.

Boîtes de dialogue

Une boîte de dialogue est une fenêtre qui est issue d'une autre fenêtre. Son but est de traiter un problème spécifique sans encombrer la fenêtre d'origine avec ces détails. Les boîtes de dialogue sont fortement utilisées dans les environnements de programmation à fenêtres, mais moins fréquemment utilisées dans les applets.

Pour créer une boîte de dialogue, il faut hériter de **JDialog**, qui est simplement une sorte de **Window**, comme les **JFrame**s. Un **JDialog** possède un *layout manager* (qui est par défaut le **BorderLayout**) auquel on ajoute des *listeners* d'événements pour traiter ceux-ci. Il y a une différence importante : on ne veut pas fermer l'application lors de l'appel de **windowClosing()**. Au lieu de cela, on libère les ressources utilisées par la fenêtre de dialogue en appelant **dispose()**. Voici un exemple très simple :

```
//: c13:Dialogs.java
// Création et utilisation de boîtes de dialogue.
// <applet code=Dialogs width=125 height=75>
// </applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;

class MyDialog extends JDialog {
  public MyDialog(JFrame parent) {
    super(parent, "My dialog", true);
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(new JLabel("Here is my dialog"));
    JButton ok = new JButton("OK");
    ok.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e){
        dispose(); // Ferme le dialogue
      }
    });
    cp.add(ok);
    setSize(150,125);
  }
```

```
  }

  public class Dialogs extends JApplet {
    JButton b1 = new JButton("Dialog Box");
    MyDialog dlg = new MyDialog(null);
    public void init() {
      b1.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e){
          dlg.show();
        }
      });
      getContentPane().add(b1);
    }
    public static void main(String[] args) {
      Console.run(new Dialogs(), 125, 75);
    }
  } ///:~
```

Une fois le JDialog créé, la méthode **show()** doit être appelée pour l'afficher et l'activer. Pour que le dialogue se ferme, il faut appeler **dispose()**.

On remarquera que tout ce qui sort d'une applet, y compris les boîtes de dialogue, n'est pas digne de confiance. C'est à dire qu'on obtient un avertissement dans la fenêtre qui apparaît. Ceci est dû au fait qu'en théorie il serait possible de tromper l'utilisateur et lui faire croire qu'il a à faire avec une de ses applications normales et de le faire taper son numéro de carte de crédit, qui partirait alors sur le Web. Une applet est toujours attachée à une page Web et visible dans un navigateur, tandis qu'une boîte de dialogue est détachée, et tout ceci est donc possible en théorie. Le résultat est qu'il n'est pas fréquent de voir une applet qui utilise une boîte de dialogue.

L'exemple suivant est plus complexe; la boîte de dialogue est composée d'une grille (en utilisant **GridLayout**) d'un type de bouton particulier qui est défini ici comme la classe **ToeButton**. Ce bouton dessine un cadre autour de lui et, selon son état, un blanc, un x ou un o au milieu. Il démarre en blanc, et ensuite, selon à qui c'est le tour, se modifie en x ou en o. Cependant, il transformera le x en o et vice versa lorsqu'on clique sur le bouton (ceci rend le principe du tic-tac-toe seulement un peu plus ennuyeux qu'il ne l'est déjà). De plus, la boîte de dialogue peut être définie avec un nombre quelconque de rangées et de colonnes dans la fenêtre principale de l'application.

```
  //: c13:TicTacToe.java
  // Démonstration de boîtes de dialogue
  // et création de vos propres composants.
  // <applet code=TicTacToe
  //  width=200 height=100></applet>
  import javax.swing.*;
  import java.awt.*;
  import java.awt.event.*;
  import com.bruceeckel.swing.*;

  public class TicTacToe extends JApplet {
    JTextField
```

```
    rows = new JTextField("3"),
    cols = new JTextField("3");
  static final int BLANK = 0, XX = 1, OO = 2;
  class ToeDialog extends JDialog {
    int turn = XX; // Démarre avec x a jouer
    // w = nombre de cellules en largeur
    // h = nombre de cellules en hauteur
    public ToeDialog(int w, int h) {
      setTitle("The game itself");
      Container cp = getContentPane();
      cp.setLayout(new GridLayout(w, h));
      for(int i = 0; i < w * h; i++)
        cp.add(new ToeButton());
      setSize(w * 50, h * 50);
      // fermeture du dialogue en JDK 1.3 :
      //#setDefaultCloseOperation(
      //#  DISPOSE_ON_CLOSE);
      // fermeture du dialogue en JDK 1.2 :
      addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent e){
          dispose();
        }
      });
    }
    class ToeButton extends JPanel {
      int state = BLANK;
      public ToeButton() {
        addMouseListener(new ML());
      }
      public void paintComponent(Graphics g) {
        super.paintComponent(g);
        int x1 = 0;
        int y1 = 0;
        int x2 = getSize().width - 1;
        int y2 = getSize().height - 1;
        g.drawRect(x1, y1, x2, y2);
        x1 = x2/4;
        y1 = y2/4;
        int wide = x2/2;
        int high = y2/2;
        if(state == XX) {
          g.drawLine(x1, y1,
            x1 + wide, y1 + high);
          g.drawLine(x1, y1 + high,
            x1 + wide, y1);
        }
        if(state == OO) {
```

```
            g.drawOval(x1, y1,
              x1 + wide/2, y1 + high/2);
          }
        }
        class ML extends MouseAdapter {
          public void mousePressed(MouseEvent e) {
            if(state == BLANK) {
              state = turn;
              turn = (turn == XX ? OO : XX);
            }
            else
              state = (state == XX ? OO : XX);
            repaint();
          }
        }
      }
    }
    class BL implements ActionListener {
      public void actionPerformed(ActionEvent e) {
        JDialog d = new ToeDialog(
          Integer.parseInt(rows.getText()),
          Integer.parseInt(cols.getText()));
        d.setVisible(true);
      }
    }
    public void init() {
      JPanel p = new JPanel();
      p.setLayout(new GridLayout(2,2));
      p.add(new JLabel("Rows", JLabel.CENTER));
      p.add(rows);
      p.add(new JLabel("Columns", JLabel.CENTER));
      p.add(cols);
      Container cp = getContentPane();
      cp.add(p, BorderLayout.NORTH);
      JButton b = new JButton("go");
      b.addActionListener(new BL());
      cp.add(b, BorderLayout.SOUTH);
    }
    public static void main(String[] args) {
      Console.run(new TicTacToe(), 200, 100);
    }
  } ///:~
```

Comme les **static**s peuvent être uniquement au niveau le plus extérieur de la classe, les classes internes ne peuvent pas avoir de données static ni de classes internes **static**.

La méthode **paintComponent()** dessine le carré autour du panneau, et le x ou le o. C'est rempli de calculs fastidieux, mais c'est direct.

Les clics de souris sont capturés par le **MouseListener**, qui vérifie d'abord si le panneau a déjà quelque chose d'écrit sur lui. Si ce n'est pas le cas, on recherche la fenêtre parente pour déterminer à qui est le tour, et on positionne l'état du **ToeButton** en conséquence. Le **ToeButton** retrouve le parent par le mécanisme de la classe interne, et passe au tour suivant. Si le bouton affiche déjà un x ou un o, son affichage est inversé. On peut voir dans ces calculs l'usage pratique du if-else ternaire décrit au Chapitre 3. On repeint le **ToeButton** chaque fois qu'il change d'état.

Le constructeur de **ToeDialog** est assez simple : il ajoute à un **GridLayout** autant de boutons que demandé, puis redimensionne chaque bouton à 50 pixels.

TicTacToe installe l'ensemble de l'application par la création de **JTextField**s (pour entrer le nombre de rangées et colonnes de la grille de boutons) et le bouton «go» avec son **ActionListener**. Lorsqu'on appuie sur le bouton, les données dans les **JTextField**s doivent être récupérées et, puisqu'elles sont au format **String**, transformées en **int**s en utilisant la méthode statique **Integer.parseInt()**.

Dialogues pour les fichiers *[File dialogs]*

Certains systèmes d'exploitation ont des boîtes de dialogue standard pour gérer certaines choses telles que les fontes, les couleurs, les imprimantes, et cetera. En tout cas, pratiquement tous les systèmes d'exploitation graphiques fournissent les moyens d'ouvrir et de sauver les fichiers, et le **JFileChooser** de Java les encapsule pour une utilisation facile.

L'application suivante utilise deux sortes de dialogues **JFileChooser**, un pour l'ouverture et un pour la sauvegarde. La plupart du code devrait maintenant être familière, et toute la partie intéressante se trouve dans les *action listeners* pour les différents clics de boutons :

```
//: c13:FileChooserTest.java
// Démonstration de boîtes de dialogues de fichiers.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

public class FileChooserTest extends JFrame {
  JTextField
    filename = new JTextField(),
    dir = new JTextField();
  JButton
    open = new JButton("Open"),
    save = new JButton("Save");
  public FileChooserTest() {
    JPanel p = new JPanel();
    open.addActionListener(new OpenL());
    p.add(open);
    save.addActionListener(new SaveL());
    p.add(save);
    Container cp = getContentPane();
    cp.add(p, BorderLayout.SOUTH);
    dir.setEditable(false);
```

```
    filename.setEditable(false);
    p = new JPanel();
    p.setLayout(new GridLayout(2,1));
    p.add(filename);
    p.add(dir);
    cp.add(p, BorderLayout.NORTH);
  }
  class OpenL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      JFileChooser c = new JFileChooser();
      // Démontre le dialogue "Open" :
      int rVal =
        c.showOpenDialog(FileChooserTest.this);
      if(rVal == JFileChooser.APPROVE_OPTION) {
        filename.setText(
          c.getSelectedFile().getName());
          dir.setText(
            c.getCurrentDirectory().toString());
      }
      if(rVal == JFileChooser.CANCEL_OPTION) {
        filename.setText("You pressed cancel");
        dir.setText("");
      }
    }
  }
  class SaveL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      JFileChooser c = new JFileChooser();
      // Démontre le dialogue "Save" :
      int rVal =
        c.showSaveDialog(FileChooserTest.this);
      if(rVal == JFileChooser.APPROVE_OPTION) {
        filename.setText(
          c.getSelectedFile().getName());
          dir.setText(
            c.getCurrentDirectory().toString());
      }
      if(rVal == JFileChooser.CANCEL_OPTION) {
        filename.setText("You pressed cancel");
        dir.setText("");
      }
    }
  }
  public static void main(String[] args) {
    Console.run(new FileChooserTest(), 250, 110);
  }
} ///:~
```

Pour un dialogue d'ouverture de fichier, on appelle **showOpenDialog()**, et pour un dialogue de sauvegarde de fichier, on appelle **showSaveDialog()**. Ces commandes ne reviennent que lorsque le dialogue est fermé. L'objet **JFileChooser** existe encore, de sorte qu'on peut en lire les données. Les méthodes **getSelectedFile()** et **getCurrentDirectory()** sont deux façons d'obtenir les résultats de l'opération. Si celles-ci renvoient **null**, cela signifie que l'utilisateur a abandonné le dialogue.

HTML sur des composants Swing

Tout composant acceptant du texte peut également accepter du texte HTML, qu'il reformatera selon les règles HTML. Ceci signifie qu'on peut très facilement ajouter du texte de fantaisie à un composant Swing. Par exemple :

```
//: c13:HTMLButton.java
// Mettre du texte HTML sur des composants Swing.
// <applet code=HTMLButton width=200 height=500>
// </applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class HTMLButton extends JApplet {
JButton b = new JButton("<html><b><font size=+2>"+
  "<center>Hello!<br><i>Press me now!");
public void init() {
  b.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e){
      getContentPane().add(new JLabel("<html>"+
        "<i><font size=+4>Kapow!"));
      // Force un réalignement pour
      // inclure le nouveau label:
      validate();
    }
  });
  Container cp = getContentPane();
  cp.setLayout(newFlowLayout());
  cp.add(b);
}
public static void main(String[] args) {
  Console.run(newHTMLButton(), 200, 500);
}
} ///:~
```

Le texte doit commencer avec , et ensuite on peut utiliser les *tags* HTML normaux. Remarquons que les *tags* de fermeture ne sont pas obligatoires.

L'**ActionListener** ajoute au formulaire un nouveau **JLabel** contenant du texte HTML. Comme ce label n'est pas ajouté dans la méthode **init()**, on doit appeler la méthode **validate()** du conteneur de façon à forcer une redisposition des composants (et de ce fait un affichage du nouveau

label).

On peut également ajouter du texte HTML à un **JTabbedPane**, **JMenuItem**, **JToolTip**, **JRadioButton** ou un **JCheckBox**.

Curseurs *[sliders]* et barres de progression *[progress bars]*

Un *slider*(qu'on a déjà utilisé dans l'exemple du sinus) permet à l'utilisateur de rentrer une donnée en déplaçant un point en avant ou en arrière, ce qui est intuitif dans certains cas (un contrôle de volume, par exemple). Un *progress bar* représente une donnée en remplissant proportionnellement un espace vide pour que l'utilisateur ait une idée de la valeur. Mon exemple favori pour ces composants consiste à simplement lier le curseur à la barre de progression, de sorte que lorsqu'on déplace le curseur la barre de progression change en conséquence :

```
//: c13:Progress.java
// Utilisation de progress bars et de sliders.
// <applet code=Progress
//  width=300 height=200></applet>
importjavax.swing.*;
importjava.awt.*;
importjava.awt.event.*;
importjavax.swing.event.*;
importjavax.swing.border.*;
importcom.bruceeckel.swing.*;

public class Progress extends JApplet {
JProgressBar pb = new JProgressBar();
JSlider sb =
  new JSlider(JSlider.HORIZONTAL, 0, 100, 60);
public void init() {
  Container cp = getContentPane();
  cp.setLayout(new GridLayout(2,1));
  cp.add(pb);
  sb.setValue(0);
  sb.setPaintTicks(true);
  sb.setMajorTickSpacing(20);
  sb.setMinorTickSpacing(5);
  sb.setBorder(new TitledBorder("Slide Me"));
  pb.setModel(sb.getModel()); // Partage du modèle
  cp.add(sb);
}
public static void main(String[] args) {
  Console.run(newProgress(), 300, 200);
}
} ///:~
```

La clé du lien entre les deux composant se trouve dans le partage de leur modèle, dans la ligne :

```
pb.setModel(sb.getModel());
```

Naturellement, on pourrait aussi contrôler les deux composants en utilisant un listener, mais ceci est plus direct pour les cas simples.

Le **JProgressBar** est assez simple, mais le **JSlider**a un grand nombre d'options, telles que l'orientation et les graduations mineures et majeures. Remarquons la simplicité de l'ajout d'une bordure avec titre.

Arbres *[Trees]*

L'utilisation d'un **JTree**peut être aussi simple que ceci :

```
add(newJTree(
newObject[] {"this", "that", "other"}));
```

Ceci affiche un arbre rudimentaire. L'API pour les arbres est vaste, probablement une des plus importantes de Swing. On peut faire à peu près tout ce qu'on veut avec des arbres, mais les tâches plus complexes demandent davantage de recherches et d'expérimentations.

Heureusement, il y a un niveau intermédiaire fourni dans la bibliothèque : les composants arbres par défaut, qui font en général ce dont on a besoin, de sorte que la plupart du temps on peut se contenter de ces composants, et ce n'est que dans des cas particuliers qu'on aura besoin d'approfondir et de comprendre plus en détail les arbres.

L'exemple suivant utilise les composants arbres par défaut pour afficher un arbre dans une applet. Lorsqu'on appuie sur le bouton, un nouveau sous-arbre est ajouté sous le noeud sélectionné (si aucun noeud n'est sélectionné, le noeud racine est utilisé) :

```
//: c13:Trees.java
// Exemple d'arbre Swing simple. Les arbres peuvent
// être beaucoup plus complexes que celui-ci.
// <applet code=Trees
//  width=250 height=250></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.tree.*;
import com.bruceeckel.swing.*;

// Prend un tableau de Strings et crée un noeud à partir
// du premier élément, et des feuilles avec les autres :
class Branch {
DefaultMutableTreeNode r;
public Branch(String[] data) {
  r =  new DefaultMutableTreeNode(data[0]);
  for(int i = 1; i  < data.length; i++)
r.add(new DefaultMutableTreeNode(data[i]));
}
public DefaultMutableTreeNode node() {
```

```
    return r ;
  }
  }

  public class Trees extends JApplet {
  String[][] data = {
    { "Colors", "Red", "Blue", "Green"},
    { "Flavors", "Tart", "Sweet", "Bland"},
    { "Length", "Short", "Medium", "Long"},
    { "Volume", "High", "Medium", "Low"},
    { "Temperature", "High", "Medium", "Low"},
    { "Intensity", "High", "Medium", "Low"},
  };
  static int i = 0;
  DefaultMutableTreeNode root, child, chosen;
  JTree tree;
  DefaultTreeModel model;
  public void init() {
    Container cp = getContentPane();
    root = new DefaultMutableTreeNode("root");
    tree = new JTree(root);
    // On l'ajoute et on le rend scrollable :
    cp.add(new JScrollPane(tree),
      BorderLayout.CENTER);
    // Obtention du modèle de l'arbre :
    model =(Default TreeModel)tree.getModel();
    JButton test = new JButton("Press me");
    test.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e){
        if(i < data.length) {
          child = new Branch(data[i++]).node();
          // Quel est le dernier élément cliqué ?
          chosen = (DefaultMutableTreeNode)
            tree.getLastSelectedPathComponent();
          if(chosen ==null) chosen = root;
          // Le modèle créera l'événement approprié.
          // En réponse, l'arbre se remettra à jour :
          model.insertNodeInto(child, chosen, 0);
          // Ceci place le nouveau noeud
          // sur le noeud actuellement sélectionné.
        }
      }
    });
    // Change les couleurs des boutons :
    test.setBackground(Color.blue);
    test.setForeground(Color.white);
    JPanel p = new JPanel();
```

```
    p.add(test);
    cp.add(p, BorderLayout.SOUTH);
  }
  public static void main(String[] args) {
    Console.run(new Trees(), 250, 250);
  }
} ///:~
```

La première classe, Branch, est un outil qui prend un tableau et construit un **DefaultMutableTreeNode**avec la première **String** comme racine, et le reste des **String**s du tableau pour les feuilles. Ensuite **node()** peut être appelé pour créer la racine de cette branche.

La classe **Trees** contient un tableau de **String**s à deux dimensions à partir duquel des **Branch**es peuvent être créées, ainsi qu'un **static int i** pour servir d'index à travers ce tableau. L'objet **DefaultMutableTreeNode** contient les noeuds, mais la représentation physique à l'écran est contrôlée par le **JTree** et son modèle associé, le **DefaultTreeModel**. Notons que lorsque le **JTree** est ajouté à l'applet, il est encapsulé dans un **JScrollPane** : c'est suffisant pour permettre un scrolling automatique.

Le **JTree** est contrôlé par son modèle. Lorsqu'on modifie les données du modèle, celui-ci génère un événement qui force le **JTree** à effectuer les mises à jour nécessaires de la partie visible de la représentation de l'arbre. Dans **init()**, le modèle est obtenu par appel à **getModel()**. Lorsqu'on appuie sur le bouton, une nouvelle branche est créée. Ensuite le composant actuellement sélectionné est recherché (on utilise la racine si rien n'est sélectionné) et la méthode **insertNodeInto()** du modèle effectue la modification de l'arbre et provoque sa mise à jour.

Un exemple comme ci-dessus peut vous donner ce dont vous avez besoin pour utiliser un arbre. Cependant les arbres ont la possibilité de faire à peu près tout ce qui est imaginable ; chaque fois que le mot *default* apparaît dans l'exemple ci-dessus, on peut y substituer sa propre classe pour obtenir un comportement différent. Mais attention : la plupart de ces classes a une interface importante, de sorte qu'on risque de passer du temps à comprendre la complexité des arbres. Cependant, on a affaire ici à une bonne conception, et les autres solutions sont en général bien moins bonnes.

Tables

Comme les arbres, les tables en Swing sont vastes et puissantes. Elles sont destinées principalement à être la populaire grille d'interface avec les bases de données via la Connectivité Bases de Données Java : *Java DataBase Connectivity* (JDBC, présenté dans le Chapitre 15) et pour cela elles ont une flexibilité énorme, que l'on paie en complexité. Il y a ici suffisamment pour servir de base à un tableur complet et pourrait probablement être le sujet d'un livre complet. Cependant, il est également possible de créer une name="Index1768">**JTable**relativement simple si on en comprend les bases.

La **JTable** contrôle la façon dont les données sont affichées, tandis que le **TableModel** contrôle les données elles-mêmes. Donc pour créer une **JTable** on créera d'abord un **TableModel**. On peut implémenter complètement l'interface **TableModel**, mais il est souvent plus simple d'hériter de la classe utilitaire **AbstractTableModel** :

```
//: c13:Table.java
// Démonstration simple d'une JTable.
```

```
// <applet code=Table
//  width=350 height=200></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.table.*;
import javax.swing.event.*;
import com.bruceeckel.swing.*;

public class Table extends JApplet {
JTextArea  txt = new JTextArea(4, 20);
// Le TableModel contrôle toutes les données :
classDataModel extends AbstractTableModel {
  Object[][] data = {
    {"one", "two", "three", "four"},
    {"five", "six", "seven", "eight"},
    {"nine", "ten", "eleven", "twelve"},
  };
  // Imprime les données lorsque la table change :
  class TML implements TableModelListener {
    public void tableChanged(TableModelEvent e){
      txt.setText(""); // Vider le texte
      for(inti = 0;  i < data.length; i++){
      for(intj = 0; j < data[0].length: j++)
        txt.append(data[i][j] + " ");
        txt.append("\n");
      }
    }
  }
  public DataModel() {
    addTableModelListener(new TML());
  }
  public int getColumnCount() {
    return data[0].length;
  }
  public int getRowCount() {
    return data.length;
  }
  public Object getValueAt(int row, intc ol) {
    return data[row][col];
  }
  public void setValueAt(Object val, int row, int col) {
    data[row][col] = val;
    // Indique que le changement a eu lieu :
    fireTableDataChanged();
  }
```

```
    public boolean isCellEditable(int row, int col) {
      return true;
    }
  }
  public void init() {
    Container cp = getContentPane();
    JTable table = new Jtable(new DataModel());
    cp.add(new JScrollPane(table));
    cp.add(BorderLayout.SOUTH, txt);
  }
  public static void main(String[] args) {
    Console.run(new Table(), 350, 200);
  }
  } ///:~
```

DataModelcontient un tableau de données, mais on pourrait aussi obtenir les données depuis une autre source telle qu'une base de données. Le constructeur ajoute un **TableModelListener** qui imprime le tableau chaque fois que la table est modifiée. Les autres méthodes suivent les conventions de nommage des Beans ; elles sont utilisées par la **JTable** lorsqu'elle veut présenter les informations contenues dans **DataModel**. **AbstractTableModel** fournit des méthodes par défaut pour **setValueAt()** et **isCellEditable()** qui interdisent les modifications de données, de sorte que ces méthodes devront être redéfinies si on veut pouvoir modifier les données.

Une fois obtenu un **TableModel**, il suffit de le passer au constructeur de la **JTable**. Tous les détails concernant l'affichage, les modifications et la mise à jour seront automatiquement gérés. Cet exemple place également la **JTable** dans un **JScrollPane**.

Sélection de l'aspect de l'interface *[Look & Feel]*

Un des aspects très intéressants de Swing est le name="Index1770">*Pluggable Look & Feel*. Il permet à un programme d'émuler le *look and feel* de divers environnements d'exploitation. On peut même faire toutes sortes de choses comme par exemple changer le *look and feel* pendant l'exécution du programme. Toutefois, en général on désire soit sélectionner le *look and feel* toutes plateformes (qui est le Metal de Swing), soit sélectionner le *look and feel* du système courant, de sorte à donner l'impression que le programme Java a été créé spécifiquement pour ce système. Le code permettant de sélectionner chacun de ces comportements est assez simple, mais il faut l'exécuter *avant* de créer les composants visuels, car ceux-ci seront créés selon le *look and feel* courant et ne seront pas changés par le simple changement de *look and feel* au milieu du programme (ce processus est compliqué et peu banal, et nous en laisserons le développement à des livres spécifiques sur Swing).

En fait, si on veut utiliser le *look and feel* toutes plateformes (metal) qui est la caractéristique des programmes Swing, il n'y a rien de particulier à faire, c'est la valeur par défaut. Si au contraire on veut utiliser le *look and feel* de l'environnement d'exploitation courant, il suffit d'insérer le code suivant, normalement au début du **main()** mais de toutes façons avant d'ajouter des composants :

```
try{
UIManager.setLookAndFeel(UIManager.
  getSystemLookAndFeelClassName());
```

```
} catch(Exception e) {}
```

Il n'y a pas besoin de faire quoi que ce soit dans la clause **catch** car le **UIManager** se positionnera par défaut au *look and feel* toutes plateformes si votre tentative d'installation d'un des autres échoue. Toutefois, pour le *debug*, l'exception peut être utile, de sorte qu'on peut au minimum placer une instruction d'impression dans la clause **catch**.

Voici un programme qui utilise un argument de ligne de commande pour sélectionner un *look and feel*, et montre à quoi ressemblent différents composants dans le *look and feel* choisi :

```
//: c13:LookAndFeel.java
// Sélection de divers looks & feels.
Import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import com.bruceeckel.swing.*;

public class LookAndFeel extends JFrame {
String[] choices = {
  "eeny", "meeny", "minie", "moe", "toe", "you"
};
Component[] samples = {
  new JButton("JButton"),
  new JTextField("JTextField"),
  new JLabel("JLabel"),
  new JCheckBox("JCheckBox"),
  new JRadioButton("Radio"),
  new JComboBox(choices),
  new JList(choices),
};
public LookAndFeel() {
  super("Look And Feel");
  Container cp = getContentPane();
  cp.setLayout(new FlowLayout());
  for(int i = 0; i <samples.length; i++)
    cp.add(samples[i]);
}
private static void usageError() {
  System.out.println(
    "Usage:LookAndFeel [cross|system|motif]");
  System.exit(1);
}
public static void main(String[] args) {
  if(args.length == 0) usageError();
  if(args[0].equals("cross")) {
    try{
      UIManager.setLookAndFeel(UIManager.
```

```
            getCrossPlatformLookAndFeelClassName());
          } catch(Exception e) {
            e.printStackTrace(System.err);
          }
        } elseif(args[0].equals("system")) {
          try{
            UIManager.setLookAndFeel(UIManager.
              getSystemLookAndFeelClassName());
          } catch(Exception e) {
            e.printStackTrace(System.err);
          }
        } elseif(args[0].equals("motif")) {
          try{
            UIManager.setLookAndFeel("com.sun.java."+
              "swing.plaf.motif.MotifLookAndFeel");
          } catch(Exception e) {
            e.printStackTrace(System.err);
          }
        } else usageError();
        // Remarquons que le look & feel doit être positionné
        // avant la création des composants.
        Console.run(new LookAndFeel(), 300, 200);
      }
      } ///:~
```

Il est également possible de créer un package de *look and feel* sur mesure, par exemple si on crée un environnement de travail pour une société qui désire une apparence spéciale. C'est un gros travail qui est bien au-delà de la portée de ce livre (en fait vous découvrirez qu'il est au-delà de la portée de beaucoup de livres dédiés à Swing).name="_Toc481064830">

Le presse-papier *[clipboard]*

JFC permet des opérations limitées avec le presse-papier système (dans le package **java.awt.datatransfer**). On peut copier des objets **String** dans le presse-papier en tant que texte, et on peut coller du texte depuis le presse-papier dans des objets **String**. Bien sûr, le presse-papier est prévu pour contenir n'importe quel type de données, mais la représentation de ces données dans le presse-papier est du ressort du programme effectuant les opérations de couper et coller. L'API *Java clipboard* permet ces extensions à l'aide du concept de «parfum» *[flavor]*. Les données en provenance du presse-papier sont associées à un ensemble de *flavors*dans lesquels on peut les convertir (par exemple, un graphe peut être représenté par une chaîne de nombres ou par une image) et on peut vérifier si les données contenues dans le presse-papier acceptent le *flavor* qui nous intéresse.

Le programme suivant est une démonstration simple de couper, copier et coller des données **String** dans une **face="Georgia">JTextArea**. On remarquera que les séquences clavier utilisées normalement pour couper, copier et coller fonctionnent également. Mais si on observe un **JTextField** ou un **JTextArea** dans tout autre programme, on verra qu'ils acceptent aussi automatiquement les séquences clavier du presse-papier. Cet exemple ajoute simplement un contrôle du presse-papier par le programme, et on peut utiliser ces techniques pour capturer du texte du presse-papier depuis

autre chose qu'un **JTextComponent**.

```
//: c13:CutAndPaste.java
// Utilisation du presse-papier.
Import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.awt.datatransfer.*;
import com.bruceeckel.swing.*;

public class CutAndPaste extends JFrame  {
JMenuBar mb = new JMenuBar();
JMenu edit = new JMenu("Edit");
JMenuItem
  cut = new JMenuItem("Cut"),
  copy = new JMenuItem("Copy"),
  paste = new JMenuItem("Paste");
JTextArea text = new JTextArea(20, 20);
Clipboard clipbd =
  getToolkit().getSystemClipboard();
public CutAndPaste()  {
  cut.addActionListener(new CutL());
  copy.addActionListener(new CopyL());
  paste.addActionListener(new PasteL());
  edit.add(cut);
  edit.add(copy);
  edit.add(paste);
  mb.add(edit);
  setJMenuBar(mb);
  getContentPane().add(text);
}
classCopyL implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    String selection = text.getSelectedText();
    if(selection == null)
      return;
    StringSelection clipString =      new StringSelection(selection);
    clipbd.setContents(clipString,clipString);
  }
}
classCutL implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    String selection = text.getSelectedText();
    if(selection == null)
      return;
    StringSelection clipString =      new StringSelection(selection);
    clipbd.setContents(clipString, clipString);
    text.replaceRange("",
```

```
        text.getSelectionStart(),
        text.getSelectionEnd());
    }
  }
  class PasteL  implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      Transferable clipData =       clipbd.getContents(CutAndPaste.this);
      try{
        String clipString =        (String)clipData.
           getTransferData(
             DataFlavor.stringFlavor);
        text.replaceRange(clipString,
          text.getSelectionStart(),
          text.getSelectionEnd());
      } catch(Exception ex) {
        System.err.println("Not String flavor");
      }
    }
  }
  public static void main(String[] args) {
    Console.run(newCutAndPaste(), 300, 200);
  }
  } ///:~
```

La création et l'ajout du menu et du **JTextArea** devraient être maintenant une activité naturelle. Ce qui est différent est la création du champ **Clipboard clipbd**, qui est faite à l'aide du **Toolkit**.

Toutes les actions sont effectuées dans les *listeners*. Les listeners **CopyL** et **CutL** sont identiques à l'exception de la dernière ligne de **CutL**, qui efface la ligne qui a été copiée. Les deux lignes particulières sont la création d'un objet **StringSelection** à partir du String, et l'appel à **setContents()** avec ce **StringSelection**. C'est tout ce qu'il y a à faire pour mettre une **String** dans le presse-papier.

Dans PasteL, les données sont extraites du presse-papier à l'aide de name="Index1777">**getContents()**. Ce qu'il en sort est un objet **Transferable**assez anonyme, et on ne sait pas exactement ce qu'il contient. Un moyen de le savoir est d'appeler **getTransferDataFlavors()**, qui renvoie un tableau d'objets name="Index1780">**DataFlavor**indiquant quels *flavors* sont acceptés par cet objet. On peut aussi le demander directement à l'aide de **isDataFlavorSupported()**, en passant en paramètre le *flavor* qui nous intéresse. Dans ce programme, toutefois, on utilise une approche téméraire : on appelle **getTransferData()**en supposant que le contenu accepte le *flavor* **String**, et si ce n'est pas le cas le problème est pris en charge par le traitement d'exception.

Dans le futur, on peut s'attendre à ce qu'il y ait plus de *flavors* acceptés.

Empaquetage d'une applet dans un fichier JAR

Une utilisation importante de l'utilitaire JAR est l'optimisation du chargement d'une applet. En Java 1.0, les gens avaient tendance à entasser tout leur code dans une seule classe, de sorte que

le client ne devait faire qu'une seule requête au serveur pour télécharger le code de l'applet. Ceci avait pour résultat des programmes désordonnés, difficiles à lire (et à maintenir), et d'autre part le fichier **.class** n'était pas compressé, de sorte que le téléchargement n'était pas aussi rapide que possible.

Les fichiers JAR résolvent le problème en compressant tous les fichiers **.class** en un seul fichier qui est téléchargé par le navigateur. On peut maintenant avoir une conception correcte sans se préoccuper du nombre de fichiers **.class** qui seront nécessaires, et l'utilisateur aura un temps de téléchargement beaucoup plus court.

Prenons par exemple **TicTacToe.java**. Il apparaît comme une seule classe, mais en fait il contient cinq classes internes, ce qui fait six au total. Une fois le programme compilé on l'emballe dans un fichier JAR avec l'instruction :

```
jar cf TicTacToe.jar *.class
```

Ceci suppose que dans le répertoire courant il n'y a que les fichiers **.class** issus de **TicTacToe.java** (sinon on emporte du bagage supplémentaire).

On peut maintenant créer une page HTML avec le nouveau *tag* name=**archive** pour indiquer le nom du fichier JAR. Voici pour exemple le *tag* utilisant l'ancienne forme du *tag* HTML :

```
<head><title>TicTacToe Example Applet
</title></head>
<body>
<applet code=TicTacToe.class
    archive=TicTacToe.jar
    width=200 height=100>
</applet>
</body>
```

Techniques de programmation

La programmation de *GUI* en Java étant une technologie évolutive, avec des modifications très importantes entre Java 1.0/1.1 et la bibliothèque Swing, certains styles de programmation anciens ont pu s'insinuer dans des exemples qu'on peut trouver pour Swing. D'autre part, Swing permet une meilleure programmation que ce que permettaient les anciens modèles. Dans cette partie, certains de ces problèmes vont être montrés en présentant et en examinant certains styles de programmation.

Lier des événements dynamiquement

Un des avantages du modèle d'événements Swing est sa flexibilité. On peut ajouter ou retirer un comportement sur événement à l'aide d'un simple appel de méthode. L'exemple suivant le montre :

```
//: c13:DynamicEvents.java
// On peut modifier dynamiquement le comportement sur événement.
// Montre également plusieurs actions pour un événement.
// <applet code=DynamicEvents
```

```
//  width=250 height=400></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import com.bruceeckel.swing.*;

public class DynamicEvents extends JApplet {
ArrayList v = new ArrayList();
int i = 0;
JButton
  b1 = new JButton("Button1",
  b2 = new JButton("Button2");
JTextArea txt = new JTextArea();
class B implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    txt.append("A button was pressed\n");
  }
}
class CountListener implements ActionListener {
  int index;
  public CountListener(int i) { index = i; }
  public void actionPerformed(ActionEvent e) {
    txt.append("Counted Listener "+index+"\n");
  }
}
class B1 implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    txt.append("Button 1 pressed\n");
    ActionListener a = new CountListener(i++);
    v.add(a);
    b2.addActionListener(a);
  }
}
class B2 implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    txt.append("Button2 pressed\n");
    int end = v.size() - 1;
    if(end >= 0) {
      b2.removeActionListener(
        (ActionListener)v.get(end));
      v.remove(end);
    }
  }
}
public void init() {
  Container cp = getContentPane();
```

```
    b1.addActionListener(new B());
    b1.addActionListener(new B1());
    b2.addActionListener(new B());
    b2.addActionListener(new B2());
    JPanel p = new JPanel();
    p.add(b1);
    p.add(b2);
    cp.add(BorderLayout.NORTH, p);
    cp.add(new JScrollPane(txt));
  }
  public static void main(String[] args) {
    Console.run(newDynamicEvents(), 250, 400);
  }
  } ///:~
```

Les nouvelles astuces dans cet exemple sont :

1. Il y a plus d'un *listener* attaché à chaque **Button**. En règle générale, les composants gèrent les événements en tant que *multicast*, ce qui signifie qu'on peut enregistrer plusieurs *listeners* pour un seul événement. Pour les composants spéciaux dans lesquels un événement est géré en tant que *unicast*, on obtiendra une exception **TooManyListenersException**.

2. Lors de l'exécution du programme, les *listeners* sont ajoutés et enlevés du **Button b2** dynamiquement. L'ajout est réalisé de la façon vue précédemment, mais chaque composant a aussi une méthode **removeXXXListener()** pour enlever chaque type de *listener*.

Ce genre de flexibilité permet une grande puissance de programmation.

Il faut remarquer qu'il n'est pas garanti que les listeners d'événements soient appelés dans l'ordre dans lequel ils sont ajoutés (bien que la plupart des implémentations le fasse de cette façon).

Séparation entre la logique applicative *[business logic]* et la logique de l'interface utilisateur *[UI logic]*

En général on conçoit les classes de manière à ce que chacune fasse une seule chose. Ceci est particulièrement important pour le code d'une interface utilisateur, car il arrive souvent qu'on lie ce qu'on fait à la manière dont on l'affiche. Ce genre de couplage empêche la réutilisation du code. Il est de loin préférable de séparer la logique applicative de la partie *GUI*. De cette manière, non seulement la logique applicative est plus facile à réutiliser, mais il est également plus facile de récupérer la *GUI*.

Un autre problème concerne les systèmes répartis *[multitiered systems]*, dans lesquels les objets applicatifs se trouvent sur une machine séparée. Cette centralisation des règles applicatives permet des modifications ayant un effet immédiat pour toutes les nouvelles transactions, ce qui est une façon intéressante d'installer un système. Cependant, ces objets applicatifs peuvent être utilisés dans de nombreuses applications, et de ce fait ils ne devraient pas être liés à un mode d'affichage particulier. Ils devraient se contenter d'effectuer les opérations applicatives, et rien de plus.

L'exemple suivant montre comme il est facile de séparer la logique applicative du code *GUI* :

```
//: c13:Separation.java
```

```
// Séparation entre la logique GUI et les objets applicatifs.
// <applet code=Separation
// width=250 height=150> </applet>
import javax.swing.*;
import java.awt.*;
import javax.swing.event.*;
import java.awt.event.*;
import java.applet.*;
import com.bruceeckel.swing.*;

class BusinessLogic {
private int modifier;
public BusinessLogic(int mod) {
  modifier = mod;
}
public void setModifier(int mod) {
  modifier = mod;
}
public int getModifier() {
  return modifier;
}
// Quelques opérations applicatives :
public int calculation1(int arg) {
  return arg * modifier;
}
public int calculation2(int arg) {
  return arg + modifier;
}
}

public class Separation extends JApplet {
JTextField
  t = new JTextField(15),
  mod = new JTextField(15);
BusinessLogic bl = new BusinessLogic(2);
JButton
  calc1 = new JButton("Calculation 1",
  calc2 = new JButton("Calculation 2");
static int getValue(JTextField tf) {
  try{
    return Integer.parseInt(tf.getText());
  } catch(NumberFormatException e) {
    return0;
  }
}
class Calc1L implements ActionListener {
  public void actionPerformed(ActionEvent e) {
```

```
        t.setText(Integer.toString(
          bl.calculation1(getValue(t))));
      }
    }
    class Calc2L implements ActionListener {
      public void actionPerformed(ActionEvent e) {
        t.setText(Integer.toString(
          bl.calculation2(getValue(t))));
      }
    }
    // Si vous voulez que quelque chose se passe chaque fois
    // qu'un JTextField est modifié, ajoutez ce listener :
    class ModL implements DocumentListener {
      public void changedUpdate(DocumentEvent e) {}
      public void insertUpdate(DocumentEvent e) {
        bl.setModifier(getValue(mod));
      }
      public void removeUpdate(DocumentEvent e) {
        bl.setModifier(getValue(mod));
      }
    }
    public void init() {
      Container cp = getContentPane();
      cp.setLayout(new FlowLayout());
      cp.add(t);
      calc1.addActionListener(new Calc1L());
      calc2.addActionListener(new Calc2L());
      JPanel p1 = new JPanel();
      p1.add(calc1);
      p1.add(calc2);
      cp.add(p1);
      mod.getDocument(). addDocumentListener(new ModL());
      JPanel p2 = new JPanel();
      p2.add(new JLabel("Modifier:"));
      p2.add(mod);
      cp.add(p2);
    }
    public static void main(String[] args) {
      Console.run(newSeparation(), 250, 100);
    }
    } ///:~
```

On peut voir que **BusinessLogic** est une classe toute simple, qui effectue ses opérations sans même avoir idée qu'elle puisse être utilisée dans un environnement GUI. Elle se contente d'effectuer son travail.

Separationgarde la trace des détails de l'interface utilisateur, et elle communique avec **BusinessLogic** uniquement à travers son interface **public**. Toutes les opérations sont concentrées sur l'é-

change d'informations bidirectionnel entre l'interface utilisateur et l'objet **BusinessLogic**. De même, **Separation** fait uniquement son travail. Comme **Separation** sait uniquement qu'il parle à un objet **BusinessLogic** (c'est à dire qu'il n'est pas fortement couplé), il pourrait facilement être transformé pour parler à d'autres types d'objets.

Penser à séparer l'interface utilisateur de la logique applicative facilite également l'adaptation de code existant pour fonctionner avec Java.

Une forme canonique

Les classes internes, le modèle d'événements de Swing, et le fait que l'ancien modèle d'événements soit toujours disponible avec de nouvelles fonctionnalités des bibliothèques qui reposent sur l'ancien style de programmation, ont ajouté un nouvel élément de confusion dans le processus de conception du code. Il y a maintenant encore plus de façons d'écrire du mauvais code.

A l'exception de circonstances qui tendent à disparaître, on peut toujours utiliser l'approche la plus simple et la plus claire : les classes *listener* (normalement des classes internes) pour tous les besoins de traitement d'événements. C'est la forme utilisée dans la plupart des exemples de ce chapitre.

En suivant ce modèle on devrait pouvoir réduire les lignes de programmes qui disent : «Je me demande ce qui a provoqué cet événement». Chaque morceau de code doit se concentrer sur une action, et non pas sur des vérifications de types. C'est la meilleure manière d'écrire le code; c'est non seulement plus facile à conceptualiser, mais également beaucoup plus facile à lire et à maintenir.

Jusqu'ici, dans ce livre, nous avons vu que Java permet de créer des morceaux de code réutilisables. L'unité de code la plus réutilisable est la classe, car elle contient un ensemble cohérent de caractéristiques (champs) et comportements (méthodes) qui peuvent être réutilisées soit directement par combinaison, soit par héritage.

L'héritage et le polymorphisme sont des éléments essentiels de la programmation orientée objet, mais dans la majorité des cas, lorsqu'on bâtit une application, en fait on désire disposer de composants qui font exactement ce qu'on veut. On aimerait placer ces éléments dans notre conception comme l'ingénieur électronicien qui assemble des puces sur un circuit. On sent bien qu'il devrait y avoir une façon d'accélérer ce style de programmation modulaire.

La programmation visuelle est devenue très populaire d'abord avec le Visual Basic (VB) de Microsoft, ensuite avec une seconde génération d'outils, avec Delphi de Borland (l'inspiration principale de la conception des JavaBeans). Avec ces outils de programmation, les composants sont représentés visuellement, ce qui est logique car ils affichent d'habitude un composant visuel tel qu'un bouton ou un champ de texte. La représentation visuelle est en fait souvent exactement l'aspect du composant lorsque le programme tournera. Une partie du processus de programmation visuelle consiste à faire glisser un composant d'une palette pour le déposer dans un formulaire. Pendant qu'on fait cette opération, l'outil de construction d'applications génère du code, et ce code entraînera la création du composant lors de l'exécution du programme.

Le simple fait de déposer des composants dans un formulaire ne suffit généralement pas à compléter le programme. Il faut souvent modifier les caractéristiques d'un composant, telles que sa couleur, son texte, à quelle base de données il est connecté, et cetera. Des caractéristiques pouvant être modifiées au moment de la conception s'appellent des *propriétés* [properties]. On peut manipuler les propriétés du composant dans l'outil de construction d'applications, et ces données de configuration sont sauvegardées lors de la construction du programme, de sorte qu'elles puissent être ré-

générées lors de son exécution.

Vous êtes probablement maintenant habitués à l'idée qu'un objet est plus que des caractéristiques ; c'est aussi un ensemble de comportements. A la conception, les comportements d'un composant visuel sont partiellement représentés par des *événements* [events], signifiant : «ceci peut arriver à ce composant». En général on décide de ce qui se passera lorsqu'un événement apparaît en liant du code à cet événement.

C'est ici que se trouve le point critique du sujet : l'outil de construction d'applications utilise la réflexion pour interroger dynamiquement le composant et découvrir quelles propriétés et événements le composant accepte. Une fois connues, il peut afficher ces propriétés et en permettre la modification (tout en sauvegardant l'état lors de la construction du programme), et afficher également les événements. En général, on double-clique sur un événement et l'outil crée la structure du code relié à cet événement. Tout ce qu'il reste à faire est d'écrire le code qui s'exécute lorsque cet événement arrive.

Tout ceci fait qu'une bonne partie du travail est faite par l'outil de construction d'applications. On peut alors se concentrer sur l'aspect du programme et ce qu'il est supposé faire, et s'appuyer sur l'outil pour s'occuper du détail des connexions. La raison pour laquelle les outils de programmation visuels on autant de succès est qu'ils accélèrent fortement le processus de construction d'une application, l'interface utilisateur à coup sûr, mais également d'autres parties de l'application.

Qu'est-ce qu'un Bean ?

Une fois la poussière retombée, un composant est uniquement un bloc de code, normalement intégré dans une classe. La clé du système est la capacité du constructeur d'applications de découvrir les propriétés et événements de ce composant. Pour créer un composant VB, le programmeur devait écrire un bout de code assez compliqué, en suivant certaines conventions pour exposer les propriétés et événements. Delphi est un outil de programmation visuelle de seconde génération, pour lequel il est beaucoup plus facile de créer un composant visuel. Java de son côté a porté la création de composants visuels à son état le plus avancé, avec les JavaBeans, car un Bean est tout simplement une classe. Il n'y a pas besoin d'écrire de code supplémentaire ou d'utiliser des extensions particulières du langage pour transformer quelque chose en Bean. La seule chose à faire, en fait, est de modifier légèrement la façon de nommer les méthodes. C'est le nom de la méthode qui dit au constructeur d'applications s'il s'agit d'une propriété, d'un événement, ou simplement une méthode ordinaire.

Dans la documentation Java, cette convention de nommage est par erreur désignée comme un modèle de conception *[design pattern]*. Ceci est maladroit, car les modèles de conception (voir *Thinking in Patterns with Java*, téléchargeable à *www.BruceEckel.com*) sont suffisamment difficiles à comprendre sans ajouter ce genre de confusions. Ce n'est pas un modèle de conception, c'est uniquement une convention de nommage assez simple :

1. Pour une propriété nommée **xxx**, on crée deux méthodes : **getXxx()** et **setXxx()**. Remarquons que la première lettre après get ou set est transformée automatiquement en minuscule pour obtenir le nom de la propriété. Le type fourni par la méthode get est le même que le type de l'argument de la méthode set. Le nom de la propriété et le type pour les méthodes get et set ne sont pas liés.
2. Pour une propriété de type boolean, on peut utiliser les méthodes get et set comme ci-dessus, ou utiliser is au lieu de get.
3. Les méthodes ordinaires du Bean ne suivent pas la convention de nommage ci-dessus,

mais elles sont **public**.

4. Pour les événements, on utilise la technique Swing du *listener*. C'est exactement la même chose que ce qu'on a déjà vu : **addFooBarListener(FooBarListener)** et **removeFooBarListener(FooBarListener)** pour gérer un **FooBarEvent**. La plupart du temps, les événements intégrés satisfont les besoins, mais on peut créer ses propres événements et interfaces *listeners*.

Le point 1 ci-dessus répond à la question que vous vous êtes peut-être posée en comparant un ancien et un nouveau code : un certain nombre de méthodes ont subi de petits changements de noms, apparemment sans raison. On voit maintenant que la plupart de ces changements avaient pour but de s'adapter aux conventions de nommage get et set de manière à transformer les composants en Beans.

On peut utiliser ces règles pour créer un Bean simple :

```
//: frogbean:Frog.java
// Un JavaBean trivial.
package frogbean;
import java.awt.*;
import java.awt.event.*;

class Spots {}

public class Frog {
  private int jumps;
  private Color color;
  private Spots spots;
  private boolean jmpr;
  public int getJumps() { return jumps; }
  public void setJumps(int newJumps) {
    jumps = newJumps;
  }
  public Color getColor() { return color; }
  public void setColor(Color newColor) {
    color = newColor;
  }
  public Spots getSpots() { return spots; }
  public void setSpots(Spots newSpots) {
    spots = newSpots;
  }
  public boolean isJumper() { return jmpr; }
  public void setJumper(boolean j) { jmpr = j; }
  public void addActionListener(
      ActionListener l) {
    //...
  }
  public void removeActionListener(
      ActionListener l) {
    // ...
```

```
  }
  public void addKeyListener(KeyListener l) {
    // ...
  }
  public void removeKeyListener(KeyListener l) {
    // ...
  }
  // Une méthode public "ordinaire" :
  public void croak() {
    System.out.println("Ribbet!");
  }
} ///:~
```

Tout d'abord, on voit qu'il s'agit d'une simple classe. En général, tous les champs seront **private**, et accessibles uniquement à l'aide des méthodes. En suivant la convention de nommage, les propriétés sont **jumps**, **color**, **spots** et **jumper** (remarquons le passage à la minuscule pour la première lettre du nom de la propriété). Bien que le nom de l'identificateur interne soit le même que le nom de la propriété dans les trois premiers cas, dans **jumper** on peut voir que le nom de la propriété n'oblige pas à utiliser un identificateur particulier pour les variables internes (ou même, en fait, d'*avoir* des variable internes pour cette propriété).

Les événements gérés par ce Bean sont **ActionEvent** et **KeyEvent**, basés sur le nom des méthodes add et remove pour le *listener* associé. Enfin on remarquera que la méthode ordinaire **croak()** fait toujours partie du Bean simplement parce qu'il s'agit d'une méthode **public**, et non parce qu'elle se conforme à une quelconque convention de nommage.

Extraction des informations sur les Beans *[BeanInfo]* à l'aide de l'introspecteur *[Introspector]*

L'un des points critiques du système des Beans est le moment où on fait glisser un bean d'une palette pour le déposer dans un formulaire. L'outil de construction d'applications doit être capable de créer le Bean (il y arrive s'il existe un constructeur par défaut) et, sans accéder au code source du Bean, extraire toutes les informations nécessaires à la création de la feuille de propriétés et de traitement d'événements.

Une partie de la solution est déjà évidente depuis la fin du Chapitre 12 : la *réflexion* Java permet de découvrir toutes les méthodes d'une classe anonyme. Ceci est parfait pour résoudre le problème des Beans, sans avoir accès à des mots clés du langage spéciaux, comme ceux utilisés dans d'autres langages de programmation visuelle. En fait, une des raisons principales d'inclure la réflexion dans Java était de permettre les Beans (bien que la réflexion serve aussi à la sérialisation des objets et à l'invocation de méthodes à distance *[RMI : remote method invocation]*). On pourrait donc s'attendre à ce qu'un outil de construction d'applications doive appliquer la réflexion à chaque Bean et à fureter dans ses méthodes pour trouver les propriétés et événements de ce Bean.

Ceci serait certainement possible, mais les concepteurs du langage Java voulaient fournir un outil standard, non seulement pour rendre les Beans plus faciles à utiliser, mais aussi pour fournir une plate-forme standard pour la création de Beans plus complexes. Cet outil est la classe **Introspector**, la méthode la plus importante de cette classe est le **static getBeanInfo()**. On passe la référence d'une **Class** à cette méthode , elle l'interroge complètement et retourne un objet **BeanInfo**

qu'on peut disséquer pour trouver les propriétés, méthodes et événements.

Vous n'aurez probablement pas à vous préoccuper de tout ceci, vous utiliserez probablement la plupart du temps des beans prêts à l'emploi, et vous n'aurez pas besoin de connaître toute la magie qui se cache là-dessous. Vous ferez simplement glisser vos Beans dans des formulaires, vous en configurerez les propriétés et vous écrirez des traitements pour les événements qui vous intéressent. Toutefois, c'est un exercice intéressant et pédagogique d'utiliser l'**Introspector** pour afficher les informations sur un Bean, et voici donc un outil qui le fait :

```
//: c13:BeanDumper.java
// Introspection d'un Bean.
// <applet code=BeanDumper width=600 height=500>
// </applet>
import java.beans.*;
import java.lang.reflect.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

public class BeanDumper extends JApplet {
  JTextField query =
    new JTextField(20);
  JTextArea results = new JTextArea();
  public void prt(String s) {
    results.append(s + "\n");
  }
  public void dump(Class bean){
    results.setText("");
    BeanInfo bi = null;
    try {
      bi = Introspector.getBeanInfo(
        bean, java.lang.Object.class);
    } catch(IntrospectionException e) {
      prt("Couldn't introspect " +
        bean.getName());
      return;
    }
    PropertyDescriptor[] properties =  bi.getPropertyDescriptors();
    for(int i = 0; i < properties.length; i++) {
      Class p = properties[i].getPropertyType();
      prt("Property type:\n  " + p.getName() + "Property name:\n  " + properties[i].get-
Name());
      Method readMethod =  properties[i].getReadMethod();
      if(readMethod != null)
        prt("Read method:\n  " + readMethod);
      Method writeMethod = properties[i].getWriteMethod();
      if(writeMethod != null)
```

```
        prt("Write method:\n  " + writeMethod);
        prt("====================");
      }
      prt("Public methods:");
      MethodDescriptor[] methods =      bi.getMethodDescriptors();
      for(int i = 0; i < methods.length; i++)
        prt(methods[i].getMethod().toString());
      prt("======================");
      prt("Event support:");
      EventSetDescriptor[] events =
        bi.getEventSetDescriptors();
      for(int i = 0; i < events.length; i++) {
        prt("Listener type:\n  " + events[i].getListenerType().getName());
        Method[] lm =  events[i].getListenerMethods();
        for(int j = 0; j < lm.length; j++)
          prt("Listener method:\n  " + lm[j].getName());
        MethodDescriptor[] lmd =  events[i].getListenerMethodDescriptors();
        for(int j = 0; j < lm.length; j++)
          prt("Method descriptor:\n  " + lmd[j].getMethod());
        Method addListener =  events[i].getAddListenerMethod();
        prt("Add Listener Method:\n  " + addListener);
        Method removeListener = events[i].getRemoveListenerMethod();
        prt("Remove Listener Method:\n  " +  removeListener);
        prt("====================");
      }
    }
    class Dumper implements ActionListener {
      public void actionPerformed(ActionEvent e) {
        String name = query.getText();
        Class c = null;
        try {
          c = Class.forName(name);
        } catch(ClassNotFoundException ex) {
          results.setText("Couldn't find " + name);
          return;
        }
        dump(c);
      }
    }
    public void init() {
      Container cp = getContentPane();
      JPanel p = new JPanel();
      p.setLayout(new FlowLayout());
      p.add(new JLabel("Qualified bean name:"));
      p.add(query);
      cp.add(BorderLayout.NORTH, p);
```

```
    cp.add(new JScrollPane(results));
    Dumper dmpr = new Dumper();
    query.addActionListener(dmpr);
    query.setText("frogbean.Frog");
    // Force evaluation
    dmpr.actionPerformed(
      new ActionEvent(dmpr, 0, ""));
  }
  public static void main(String[] args) {
    Console.run(new BeanDumper(), 600, 500);
  }
} ///:~
```

BeanDumper.dump() est la méthode qui fait tout le travail. Il essaie d'abord de créer un objet **BeanInfo**, et en cas de succès il appelle les méthodes de BeanInfo qui fournissent les informations sur les propriétés, méthodes et événements. Dans **Introspector.getBeanInfo()**, on voit qu'il y a un second argument. Celui-ci dit à l'**Introspector** où s'arrêter dans la hiérarchie d'héritage. Ici, il s'arrête avant d'analyser toutes les méthodes d'**Object**, parce qu'elles ne nous intéressent pas.

Pour les propriétés, **getPropertyDescriptors()** renvoie un tableau de **PropertyDescriptor**s. Pour chaque **PropertyDescriptor**, on peut appeler **getPropertyType()** pour connaître la classe d'un objet passé par les méthodes de propriétés. Ensuite, on peut obtenir le nom de chaque propriété (issu du nom des méthodes) à l'aide de **getName()**, la méthode pour la lire à l'aide de **getReadMethod()**, et la méthode pour la modifier à l'aide de **getWriteMethod()**. Ces deux dernières méthodes retournent un objet **Method** qui peut être utilisé pour appeler la méthode correspondante de l'objet (ceci fait partie de la réflexion).

Pour les méthodes **public** (y compris les méthodes des propriétés), **getMethodDescriptors()** renvoie un tableau de **MethodDescriptor**s. Pour chacun de ces descripteurs, on peut obtenir l'objet **Method** associé, et imprimer son nom.

Pour les événements, **getEventSetDescriptors()** renvoie un tableau de (que pourrait-il renvoyer d'autre ?) **EventSetDescriptor**s. Chacun de ces descripteurs peut être utilisé pour obtenir la classe du *listener*, les méthodes de cette classe *listener*, et les méthodes pour ajouter et enlever ce *listener*. Le programme **BeanDumper** imprime toutes ces informations.

Au démarrage, le programme force l'évaluation de **frogbean.Frog**. La sortie, après suppression de détails inutiles ici, est :

```
class name: Frog
Property type:
  Color
Property name:
  color
Read method:
  public Color getColor()
Write method:
  public void setColor(Color)
====================Property type:
  Spots
```

```
Property name:
  spots
Read method:
  public Spots getSpots()
Write method:
  public void setSpots(Spots)
====================Property type:
  boolean
Property name:
  jumper
Read method:
  public boolean isJumper()
Write method:
  public void setJumper(boolean)
====================Property type:
  int
Property name:
  jumps
Read method:
  public int getJumps()
Write method:
  public void setJumps(int)
====================Public methods:
public void setJumps(int)
public void croak()
public void removeActionListener(ActionListener)
public void addActionListener(ActionListener)
public int getJumps()
public void setColor(Color)
public void setSpots(Spots)
public void setJumper(boolean)
public boolean isJumper()
public void addKeyListener(KeyListener)
public Color getColor()
public void removeKeyListener(KeyListener)
public Spots getSpots()
======================Event support:
Listener type:
  KeyListener
Listener method:
  keyTyped
Listener method:
  keyPressed
Listener method:
  keyReleased
Method descriptor:
  public void keyTyped(KeyEvent)
```

```
Method descriptor:
  public void keyPressed(KeyEvent)
Method descriptor:
  public void keyReleased(KeyEvent)
Add Listener Method:
  public void addKeyListener(KeyListener)
Remove Listener Method:
  public void removeKeyListener(KeyListener)
====================Listener type:
  ActionListener
Listener method:
  actionPerformed
Method descriptor:
  public void actionPerformed(ActionEvent)
Add Listener Method:
  public void addActionListener(ActionListener)
Remove Listener Method:
  public void removeActionListener(ActionListener)
====================
```

La liste des méthodes **public** contient les méthodes qui ne sont pas associées à une propriété ou un événement, telles que **croak()**, ainsi que celles qui le sont. Ce sont toutes les méthodes pouvant être appelées par programme pour un Bean, et l'outil de construction d'applications peut décider de les lister toutes lors de la création des appels de méthodes, pour nous faciliter la tâche.

Enfin, on peut voir que les événements sont tous triés entre le *listener*, ses méthodes et les méthodes pour ajouter et supprimer les *listeners*. Fondamentalement, une fois obtenu le **BeanInfo**, on peut trouver tout ce qui est important pour un Bean. On peut également appeler les méthodes de ce Bean, bien qu'on n'ait aucune autre information à l'exception de l'objet (ceci est également une caractéristique de la réflexion).

Un Bean plus complexe

L'exemple suivant est un peu plus compliqué, bien que futile. Il s'agit d'un **JPanel** qui dessine un petit cercle autour de la souris chaque fois qu'elle se déplace. Lorsqu'on clique, le mot Bang! apparaît au milieu de l'écran, et un *action listener* est appelé.

Les propriétés modifiables sont la taille du cercle, ainsi que la couleur, la taille et le texte du mot affiché lors du clic. Un BangBean a également ses propres **addActionListener()** et **removeActionListener()**, de sorte qu'on peut y attacher son propre listener qui sera appelé lorsque l'utilisateur clique sur le **BangBean**. Vous devriez être capables de reconnaître le support des propriétés et événements :

```
//: bangbean:BangBean.java
// Un Bean graphique.
package bangbean;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
```

```
import java.io.*;
import java.util.*;
import com.bruceeckel.swing.*;

public class BangBean extends JPanel
    implements Serializable {
  protected  int xm, ym;
  protected  int cSize = 20; // Taille du cercle
  protected  String text = "Bang!";
  protected  int fontSize = 48;
  protected  Color tColor = Color.red;
  protected  ActionListener actionListener;
  public BangBean() {
    addMouseListener(new ML());
    addMouseMotionListener(new MML());
  }
  public int getCircleSize() { return cSize; }
  public void setCircleSize(int newSize) {
    cSize = newSize;
  }
  public String getBangText() { return text; }
  public void setBangText(String newText) {
    text = newText;
  }
  public int getFontSize() { return fontSize; }
  public void setFontSize(int newSize) {
    fontSize = newSize;
  }
  public Color getTextColor() { return tColor; }
  public void setTextColor(Color newColor) {
    tColor = newColor;
  }
  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.setColor(Color.black);
    g.drawOval(xm - cSize/2, ym - cSize/2,
      cSize, cSize);
  }
  // Ceci est un "unicast listener", qui est
  // la forme la plus simple de gestion des listeners :
  public void addActionListener (
      ActionListener l)
        throws TooManyListenersException {
    if(actionListener != null)
      throw new TooManyListenersException();
    actionListener = l;
  }
```

```
  public void removeActionListener(
      ActionListener l) {
    actionListener = null;
  }
  class ML extends MouseAdapter {
    public void mousePressed(MouseEvent e) {
      Graphics g = getGraphics();
      g.setColor(tColor);
      g.setFont(
        new Font(
          "TimesRoman", Font.BOLD, fontSize));
      int width =
        g.getFontMetrics().stringWidth(text);
      g.drawString(text,
        (getSize().width - width) /2,
        getSize().height/2);
      g.dispose();
      // Appel de la méthode du listener :
      if(actionListener != null)
        actionListener.actionPerformed(
          new ActionEvent(BangBean.this,
            ActionEvent.ACTION_PERFORMED, null));
    }
  }
  class MML extends MouseMotionAdapter {
    public void mouseMoved(MouseEvent e) {
      xm = e.getX();
      ym = e.getY();
      repaint();
    }
  }
  public Dimension getPreferredSize() {
    return new Dimension(200, 200);
  }
} ///:~
```

La première chose qu'on remarquera est que **BangBean** implémente l'interface **Serializable**. Ceci signifie que l'outil de construction d'applications peut conserver toutes les informations sur le **BangBean** en utilisant la sérialisation, lorsque les valeurs des propriétés ont été ajustées par l'utilisateur. Lors de la création du Bean au moment de l'exécution du programme, ces propriétés sauvegardées sont restaurées de manière à obtenir exactement ce qu'elles valaient lors de la conception.

On peut voir que tous les champs sont **private**, ce qu'on fait en général avec un Bean : autoriser l'accès uniquement à travers des méthodes, normalement en utilisant le système de propriétés.

En observant la signature de **addActionListener()**, on voit qu'il peut émettre une **TooManyListenersException**. Ceci indique qu'il est *unicast*, ce qui signifie qu'il signale à un seul *listener* l'arrivée d'un événement. En général on utilise des événements *multicast*, de sorte que de nombreux *listeners* puissent être notifiés de l'arrivée d'un événement. Cependant on entre ici dans des prob-

lèmes que vous ne pouvez pas comprendre avant le chapitre suivant; on en reparlera donc (sous le titre «*JavaBeans revisited*»). Un événement *unicast* contourne le problème.

Lorsqu'on clique avec la souris, le texte est placé au milieu du **BangBean**, et si le champ de l'**actionListener** n'est pas nul, on appelle son **actionPerformed()**, ce qui crée un nouvel objet **ActionEvent**. Lorsque la souris est déplacée, ses nouvelles coordonnées sont lues et le panneau est redessiné (ce qui efface tout texte sur ce panneau, comme on le remarquera).

Voici la classe **BangBeanTest** permettant de tester le *bean* en tant qu'applet ou en tant qu'application :

```
//: c13:BangBeanTest.java
// <applet code=BangBeanTest
// width=400 height=500></applet>
import bangbean.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import com.bruceeckel.swing.*;

public class BangBeanTest extends JApplet {
  JTextField txt = new JTextField(20);
  // Affiche les actions lors des tests :
  class BBL implements ActionListener {
    int count = 0;
    public void actionPerformed(ActionEvent e){
      txt.setText("BangBean action "+ count++);
    }
  }
  public void init() {
    BangBean bb = new BangBean();
    try {
      bb.addActionListener(new BBL());
    } catch(TooManyListenersException e) {
      txt.setText("Too many listeners");
    }
    Container cp = getContentPane();
    cp.add(bb);
    cp.add(BorderLayout.SOUTH, txt);
  }
  public static void main(String[] args) {
    Console.run(new BangBeanTest(), 400, 500);
  }
} ///:~
```

Lorsqu'un *Bean* est dans un environnement de développement, cette classe n'est pas utilisée, mais elle est utile pour fournir une méthode de test rapide pour chacun de nos *Beans*. **BangBeanTest** place un **BangBean** dans une applet, attache un simple **ActionListener** au **BangBean** pour afficher un compteur d'événements dans le JTextField chaque fois qu'un **ActionEvent** arrive. En

temps normal, bien sûr, l'outil de construction d'applications créerait la plupart du code d'utilisation du Bean.

Lorsqu'on utilise le **BangBean** avec le **BeanDumper**, ou si on place le **BangBean** dans un environnement de développement acceptant les *Beans*, on remarquera qu'il y a beaucoup plus de propriétés et d'actions que ce qui n'apparaît dans le code ci-dessus. Ceci est dû au fait que **BangBean** hérite de **JPanel**, et comme **JPanel** est également un *Bean*, on en voit également ses propriétés et événements.

Empaquetage d'un Bean

Pour installer un *Bean* dans un outil de développement visuel acceptant les *Beans*, le *Bean* doit être mis dans le conteneur standard des *Beans*, qui est un fichier JAR contenant toutes les classes, ainsi qu'un fichier *manifest* qui dit «ceci est un *Bean*». Un fichier manifest est un simple fichier texte avec un format particulier. Pour le **BangBean**, le fichier manifest ressemble à ceci (sans la première et la dernière ligne) :

```
//:! :BangBean.mf
Manifest-Version: 1.0

Name: bangbean/BangBean.class
Java-Bean: True
///:~
```

La première ligne indique la version de la structure du fichier manifest, qui est 1.0 jusqu'à nouvel ordre de chez Sun. la deuxième ligne (les lignes vides étant ignorées) nomme le fichier **BangBean.class**, et la troisième précise que c'est un Bean. Sans la troisième ligne, l'outil de construction de programmes ne reconnaîtra pas la classe en tant que Bean.

Le seul point délicat est de s'assurer d'avoir le bon chemin dans le champ «Name:». Si on retourne à **BangBean.java**, on voit qu'il est dans le **package bangbean** (et donc dans un sous-répertoire appelé bangbean qui est en dehors du classpath), et le nom dans le fichier manifest doit contenir cette information de package. De plus, il faut placer le fichier manifest dans le répertoire *au-dessus* de la racine du package, ce qui dans notre cas veut dire le placer dans le répertoire au-dessus du répertoire bangbean. Ensuite il faut lancer **jar** depuis le répertoire où se trouve le fichier manifest, de la façon suivante :

```
jar cfm BangBean.jar BangBean.mf bangbean
```

Ceci suppose qu'on veut que le fichier JAR résultant s'appelle **BangBean.jar**, et qu'on a placé le fichier manifest dans un fichier appelé **BangBean.mf**.

On peut se demander ce qui se passe pour les autres classes générées lorsqu'on a compilé **BangBean.java**. Eh bien, elles ont toutes abouti dans le sous-répertoire **bangbean**. Lorsqu'on donne à la commande **jar** le nom d'un sous-répertoire, il embarque tout le contenu de ce sous-répertoire dans le fichier jar (y compris, dans notre cas, le code source original **BangBean.java** qu'on peut ne pas vouloir inclure dans les Beans). Si on regarde à l'intérieur du fichier JAR, on découvre que le fichier manifest n'y est pas, mais que **jar** a créée son propre fichier manifest (partiellement sur la base du nôtre) appelé **MANIFEST.MF** et l'a placé dans le sous-répertoire **META-INF** (pour meta-information). Si on ouvre ce fichier manifest on remarquera également qu'une information de signature numérique a été ajoutée par **jar** pour chaque fichier, de la forme :

```
Digest-Algorithms: SHA MD5
SHA-Digest: pDpEAG9NaeCx8aFtqPI4udSX/O0=MD5-Digest: O4NcS1-
hE3Smnzlp2hj6qeg==
```

En général, on ne se préoccupe pas de tout ceci, et lors de modifications il suffit de modifier le fichier manifest d'origine et rappeler **jar** pour créer un nouveau fichier JAR pour le *Bean*. On peut aussi ajouter d'autres *Beans* dans le fichier JAR en ajoutant simplement leurs informations dans le manifest.

Une chose à remarquer est qu'on mettra probablement chaque Bean dans son propre sous-répertoire, puisque lorsqu'on crée un fichier JAR on passe à la commande **jar** le nom d'un sous-répertoire et qu'il place tout ce qui est dans ce répertoire dans le fichier JAR. On peut voir que **Frog** et **BangBean** sont chacun dans leur propre sous-répertoire.

Une fois le Bean convenablement inséré dans un fichier JAR, on peut l'installer dans un environnement de développement de programmes acceptant les Beans. La façon de le faire varie d'un outil à l'autre, mais Sun fournit gratuitement un banc de test pour les JavaBeans dans leur *Beans Development Kit* (BDK) appelé la *beanbox* (le BDK se télécharge à partir de *java.sun.com/beans*). Pour placer un Bean dans la *beanbox*, il suffit de copier le fichier JAR dans le sous-répertoire jars du BDK avant de lancer la *beanbox*.

Un support des Beans plus sophistiqué

On a vu comme il était simple de fabriquer un Bean. Mais on n'est pas limité à ce qu'on a vu ici. L'architecture des JavaBeans fournit un point d'entrée simple, mais peut aussi s'adapter à des cas plus complexes. Ceux-ci ne sont pas du ressort de ce livre, mais on va les introduire rapidement ici. On trouvera plus de détails à *java.sun.com/beans*.

Un endroit où l'on peut apporter des perfectionnements est le traitement des propriétés. Les exemples ci-dessus n'ont montré que des propriétés uniques, mais il est également possible de représenter plusieurs propriétés dans un tableau. C'est ce qu'on appelle une *propriété indexée [indexed property]*. Il suffit de fournir les méthodes appropriées (également en suivant une convention de nommage pour les noms de méthodes) et l'**Introspector** reconnaît une propriété indexée, de sorte qu'un outil de construction d'applications puisse y répondre correctement.

Les propriétés peuvent être *liées* [bound], ce qui signifie qu'elles avertiront les autres objets à l'aide d'un **PropertyChangeEvent**. Les autres objets peuvent alors décider de se modifier eux-mêmes suite à la modification de ce Bean.

Les propriétés peuvent être *contraintes* [constrained], ce qui signifie que les autres objets peuvent mettre leur veto sur la modification de cette propriété si c'est inacceptable. Les autres objets sont avertis à l'aide d'un **PropertyChangeEvent**, et ils peuvent émettre un **PropertyVetoException** pour empêcher la modification et pour rétablir les anciennes valeurs.

On peut également modifier la façon de représenter le Bean lors de la conception :

1. On peut fournir une feuille de propriétés spécifique pour un Bean particulier. La feuille de propriétés normale sera utilisée pour tous les autres Beans, mais la feuille spéciale sera automatiquement appelée lorsque ce Bean sera sélectionné.

2. On peut créer un éditeur spécifique pour une propriété particulière, de sorte que la feuille de propriétés normale est utilisée, mais si on veut éditer cette propriété, c'est cet édi-

teur qui est automatiquement appelé.

3. On peut fournir une classe **BeanInfo** spécifique pour un Bean donné, pour fournir des informations différentes de celles créées par défaut par l'**Introspector**.

4. Il est également possible de valider ou dévalider le mode expert dans tous les **FeatureDescriptor**s pour séparer les caractéristiques de base de celles plus compliquées.

Davantage sur les Beans

Il y a un autre problème qui n'a pas été traité ici. Chaque fois qu'on crée un Bean, on doit s'attendre à ce qu'il puisse être exécuté dans un environnement *multithread*. Ceci signifie qu'il faut comprendre les problèmes du *threading,* qui sera présenté au Chapitre 14. On y trouvera un paragraphe appelé «*JavaBeans revisited*» qui parlera de ce problème et de sa solution.

Il y a plusieurs livres sur les JavaBeans, par exemple *JavaBeans* par Elliotte Rusty Harold (IDG, 1998).

Résumé

De toutes les librairies Java, c'est la librairie de GUI qui a subi les changements les plus importants de Java 1.0 à Java 2. L'AWT de Java 1.0 était nettement critiqué comme étant une des moins bonnes conceptions jamais vues, et bien qu'il permette de créer des programmes portables, la GUI résultante était aussi médiocre sur toutes les plateformes. Il était également limité, malaisé et peu agréable à utiliser en comparaison des outils de développement natifs disponibles sur une plateforme donnée.

Lorsque Java 1.1 introduisit le nouveau modèle d'événements et les JavaBeans, la scène était installée. Il était désormais possible de créer des composants de GUI pouvant être facilement glissés et déposés à l'intérieur d'outils de développement visuels. De plus, la conception du modèle d'événements et des Beans montre l'accent mis sur la facilité de programmation et la maintenabilité du code (ce qui n'était pas évident avec l'AWT du 1.0). Mais le travail ne s'est terminé qu'avec l'apparition des classes JFC/Swing. Avec les composants Swing, la programmation de GUI toutes plateformes devient une expérience civilisée.

En fait, la seule chose qui manque est l'outil de développement, et c'est là que se trouve la révolution. Visual Basic et Visual C++ de Microsoft nécessitent des outils de développement de Microsoft, et il en est de même pour Delphi et C++ Builder de Borland. Si on désire une amélioration de l'outil, on n'a plus qu'à croiser les doigts et espérer que le fournisseur le fera. Mais java est un environnement ouvert, et de ce fait, non seulement il permet la compétition des outils, mais il l'encourage. Et pour que ces outils soient pris au sérieux, ils doivent permettre l'utilisation des JavaBeans. Ceci signifie un terrain de jeu nivelé : si un meilleur outil de développement apparaît, on n'est pas lié à celui qu'on utilisait jusqu'alors, on peut migrer vers le nouvel outil et augmenter sa productivité. Cet environnement compétitif pour les outils de développement ne s'était jamais vu auparavant, et le marché résultant ne peut que générer des résultats positifs pour la productivité du programmeur.

Ce chapitre était uniquement destiné à vous fournir une introduction à la puissance de Swing et vous faire démarrer, et vous avez pu voir comme il était simple de trouver son chemin à travers les librairies. Ce qu'on a vu jusqu'ici suffira probablement pour une bonne part à vos besoins en développement d'interfaces utilisateurs. Cependant, Swing ne se limite pas qu'à cela. Il est destiné à être une boîte à outils de conception d'interfaces utilisateurs très puissante. Il y a probablement une

façon de faire à peu près tout ce qu'on peut imaginer.

Si vous ne trouvez pas ici ce dont vous avez besoin, fouillez dans la documentation en ligne de Sun, et recherchez sur le Web, et si cela ne suffit pas, cherchez un livre consacré à Swing. Un bon endroit pour démarrer est *The JFC Swing Tutorial*, par Walrath & Campione (Addison Wesley, 1999).

exercices

Les solutions des exercices sélectionnés se trouvent dans le document électronique *The Thinking in Java Annotated Solution Guide*, disponible pour une faible somme depuis *www.BruceEckel.com*.

1. Créer une applet/application utilisant la classe **Console** comme montré dans ce chapitre. Inclure un champ texte et trois boutons. Lorsqu'on appuie sur chaque bouton, faire afficher un texte différent dans le champ texte.

2. Ajouter une boîte à cocher à l'applet de l'exercice 1, capturer l'événement, et insérer un texte différent dans le champ texte.

3. Créer une applet/application utilisant **Console**. Dans la documentation HTML de *java.sun.com*, trouver le **JPasswordField** et l'ajouter à ce programme. Si l'utilisateur tape le mot de passe correct, utiliser **JOptionPane** pour fournir un message de succès à l'utilisateur.

4. Créer une applet/application utilisant **Console**, et ajouter tous les composants qui ont une méthode **addActionListener()** (rechercher celles-ci dans la documentation HTML de *java.sun.com*; conseil : utiliser l'index). Capturer ces événements et afficher un message approprié pour chacun dans un champ texte.

5. Créer une applet/application utilisant **Console**, avec un **JButton** et un **JTextField**. Ecrire et attacher le listener approprié de sorte que si le bouton a le focus, les caractères tapés dessus apparaissent dans le **JTextField**.

6. Créer une applet/application utilisant **Console**. Ajouter à la fenêtre principale tous les composants décrits dans ce chapitre, y compris les menus et une boîte de dialogue.

7. Modifier **TextFields.java** de sorte que les caractères de **t2** gardent la casse qu'ils avaient lorsqu'ils ont été tapés, plutôt que les forcer automatiquement en majuscules.

8. Rechercher et télécharger un ou plusieurs des environnements de développement de GUI disponibles sur Internet, ou acheter un produit du commerce. Découvrir ce qu'il faut faire pour ajouter **Bangbean** à cet environnement et pour l'utiliser.

9. Ajouter **Frog.class** au fichier manifest comme montré dans ce chapitre, et lancer **jar** pour créer un fichier JAR contenant à la fois **Frog** et **BangBean**. Ensuite, télécharger et installer le BDK de Sun ou utiliser un outil de développement admettant les Beans et ajouter le fichier JAR à votre environnement de manière à pouvoir tester les deux Beans.

10. Créer un JavaBean appelé **Valve** qui contient deux propriétés : un **boolean** appelé on et un **int** appelé level. Créer un fichier manifest, utiliser **jar** pour empaqueter le Bean, puis le charger dans la *beanbox* ou dans un outil de développement acceptant les Beans, de manière à pouvoir le tester.

11. Modifier **MessageBoxes.java** de manière à ce qu'il ait un **ActionListener** individuel pour chaque bouton (au lieu d'un correspondant au texte du bouton).

12. Surveiller un nouveau type d'événement dans **TrackEvent.java** en ajoutant le nouveau code de traitement de l'événement. Il faudra découvrir vous-même le type d'événement que vous voulez surveiller.

13. Créer un nouveau type de bouton hérité de **JButton**. Chaque fois qu'on appuie sur le bouton, celui-ci doit modifier sa couleur selon une couleur choisie de façon aléatoire. Voir **ColorBoxes.java** au Chapitre 14 pour un exemple de la manière de générer aléatoirement une valeur de couleur.

14. Modifier TextPane.java de manière à utiliser un **JTextArea** à la place du **JTextPane**.

15. Modifier **Menus.java** pour utiliser des boutons radio au lieu de boîtes à cocher dans les menus.

16. Simplifier **List.java** en passant le tableau au constructeur et en éliminant l'ajout dynamique d'éléments à la liste.

17. Modifier **SineWave.java** pour transformer **SineDraw** en JavaBean en lui ajoutant des méthodes get et set.

18. Vous vous souvenez du jouet permettant de faire des dessins avec deux boutons, un qui contrôle le mouvement vertical, et un qui contrôle le mouvement horizontal ? En créer un, en utilisant **SineWave.java** comme point de départ. A la place des boutons, utiliser des curseurs. Ajouter un bouton d'effacement de l'ensemble du dessin.

19. Créer un indicateur de progression asymptotique qui ralentit au fur et à mesure qu'il s'approche de la fin. Ajouter un comportement aléatoire de manière à donner l'impression qu'il se remet à accélérer.

20. Modifier **Progress.java** de manière à utiliser un *listener* plutôt que le partage du modèle pour connecter le curseur et la barre de progression.

21. Suivre les instructions du paragraphe «Empaquetage d'une applet dans un fichier JAR» pour placer TicTacToe.java dans un fichier JAR. Créer une page HTML avec la version brouillonne et compliquée du tag applet, et la modifier pour utiliser le tag archive de manière à utiliser le fichier JAR (conseil : commencer par utiliser la page HTML pour **TicTacToe.java** qui est fournie avec le code source pour ce livre).

22. Créer une applet/application utilisant **Console**. Celle-ci doit avoir trois curseurs, un pour le rouge, un pour le vert et un pour le bleu de **java.awt.Color**. Le reste du formulaire sera un **JPanel** qui affiche la couleur fixée par les trois curseurs. Ajouter également des champs textes non modifiables qui indiquent les valeurs courantes des valeurs RGB.

23. Dans la documentation HTML de **javax.swing**, rechercher le **JColorChooser**. Ecrire un programme avec un bouton qui ouvre le sélectionneur de couleur comme un dialogue.

24. Presque tous les composants Swing sont dérivés de **Component**, qui a une méthode **setCursor()**. Rechercher ceci dans la documentation HTML Java . Créer une applet et modifier le curseur selon un des curseurs disponibles dans la classe **Cursor**.

25. En prenant comme base **ShowAddListeners.java**, créer un programme avec toutes les fonctionnalités de **ShowMethodsClean.java** du Chapitre 12.

[61] Une variante est appelée le principe de l'étonnement minimum, qui dit essentiellement : «ne pas surprendre l'utilisateur».

[62] Ceci est un exemple du modèle de conception *[design pattern]* appelé la *méthode du*

modèle [template method].

[63] On suppose que le lecteur connaît les bases du HTML. Il n'est pas très difficile à comprendre, et il y a beaucoup de livres et de ressources disponibles.

[64] Cette page (en particulier la partie clsid) semblait bien fonctionner avec le JDK1.2.2 et le JDK1.3rc-1. Cependant il se peut que vous ayez parfois besoin de changer le tag dans le futur. Des détails peuvent être trouvés à *java.sun.com*.

[65] Selon moi. Et après avoir étudié Swing, vous n'aurez plus envie de perdre votre temps sur les parties plus anciennes.

[66] Comme décrit plus avant, Frame était déjà utilisé par AWT, de sorte que Swing utilise JFrame.

[67] Ceci s'éclaircira après avoir avancé dans ce chapitre. D'abord faire de la référence **JApplet** un membre static de la classe (au lieu d'une variable locale de **main()**), et ensuite appeler **applet.stop()** et **applet.destroy()** dans **WindowAdapter.windowClosing()** avant d'appeler **System.exit()**.

[68] Il n'y a pas de **MouseMotionEvent** bien qu'il semble qu'il devrait y en avoir un. Le cliquage et le déplacement sont combinés dans **MouseEvent**, de sorte que cette deuxième apparition de **MouseEvent** dans le tableau n'est pas une erreur.

[69] En Java 1.0/1.1 on ne pouvait pas hériter de l'objet bouton de façon exploitable. C'était un des nombreux défauts de conception.

Chapitre 14 - Les Threads multiples

Les objets permettent une division d'un programme en sections indépendantes. Souvent, nous avons aussi besoin de découper un programme en unités d'exécution indépendantes.

Chacune de ces unités d'exécution est appelé un *thread*, et vous programmez comme si chaque thread s'exécutait par elle même et avait son propre CPU. Un mécanisme sous-jacent s'occupe de diviser le temp CPU pour vous, mais en général vous n'avez pas besoin d'y penser, c'est ce qui fait de la programmation multi-threads une tâche beaucoup plus facile.

Un *processus* est un programme s'exécutant dans son propre espace d'adressage. Un système d'exploitation multitâche est capable d'exécuter plusieurs processus (programme) en même temps, en faisant comme si chacun d'eux s'exécutait de façon indépendante, en accordant périodiquement des cycles CPU à chaque processus. Un thread est un flot de contrôle séquentiel à l'intérieur d'un processus. Un processus peut contenir plusieurs threads s'exécutant en concurrence.

Il existe plusieurs utilisations possibles du multithreading, mais en général, vous aurez une partie de votre programme attachée à un événement ou une ressource particulière, et vous ne voulez pas que le reste de votre programme dépende de ça. Vous créez donc un thread associé à cet événement ou ressource et laissez celui-ci s'exécuter indépendamment du programme principal. Un bon exemple est celui d'un bouton « quitter » - vous ne voulez pas être obliger de vérifier le bouton quitter dans chaque partie de code que vous écrivez dans votre programme, mais vous voulez que le bouton quitter réponde comme si vous le vérifiez régulièrement. En fait, une des plus immédiatement indiscutables raisons d'être du multithreading est de produire des interfaces utilisateurs dynamiques. *[responsive user interface]*

Interfaces utilisateurs dynamiques *[Responsive user interfaces]*

Comme point de départ, considérons un programme qui contient des opérations nécessitant beaucoup de temps CPU finissant ainsi par ignorer les entrées de l'utilisateur et répondrait mal. Celle-ci, une combinaison applet/application, affichera simplement le résultat d'un compteur actif *[running counter]* :

```
//: c14:Counter1.java
// Une interface utilisateur sans répondant.
// <applet code=Counter1 width=300 height=100>
// </applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class Counter1 extends JApplet {
  private int count = 0;
  private JButton
    start = new JButton("Start"),
    onOff = new JButton("Toggle");
  private JTextField t = new JTextField(10);
```

```
  private boolean runFlag = true;
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(t);
    start.addActionListener(new StartL());
    cp.add(start);
    onOff.addActionListener(new OnOffL());
    cp.add(onOff);
  }
  public void go() {
    while (true) {
      try {
        Thread.sleep(100);
      } catch(InterruptedException e) {
        System.err.println("Interrupted");
      }
      if (runFlag)
        t.setText(Integer.toString(count++));
    }
  }
  class StartL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      go();
    }
  }
  class OnOffL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      runFlag = !runFlag;
    }
  }
  public static void main(String[] args) {
    Console.run(new Counter1(), 300, 100);
  }
} ///:~
```

À ce niveau, le code Swing et de l'applet devrait être raisonnablement familier depuis le chapitre 13. La méthode **go()** est celle dans laquelle le programme reste occupé: il met le valeur courante de **count** dans le **JTextField t**, avant d'incrémenter **count**.

La boucle infinie à l'intérieur de **go()** appelle **sleep()**. **sleep()** doit être associé avec un objet **Thread**, ce qui montre que toute application a un thread qui lui est associé. (En effet, Java est basé sur des threads et il y en a toujours qui tourne avec votre application.) Donc, que vous utilisiez explicitement les threads ou non, vous pouvez accéder au thread courant utilisé par votre programme avec **Thread** et la méthode static **sleep()**

Notez que **sleep()** peut déclencher une **InterruptedException**, alors que le déclenchement d'une telle exception est considéré comme un moyen hostile d'arrêter un thread et devrait être découragé. (Encore une fois les exceptions sont destinées aux conditions exceptionnelles et non pour

le controle du flot normal.) L'interruption d'un thread en sommeil sera incluse dans une future fonctionnalité du langage.

Quand le bouton **start** est pressé, **go()** est appelé. En examinant **go()**, vous pouvez naïvement pensé (comme je le fais) que cela autorisera le multithreading parce que elle se met en sommeil *[it goes to sleep]*. C'est le cas, tant que la méthode n'est pas en sommeil, tout ce passe comme si le CPU pouvait être occupé à gérer les autres boutons. *[=That is, while the method is asleep, it seems like the CPU could be busy monitoring other button presses.]* Mais il s'avère que le vrai problème est que **go()** ne s'achève jamais, puisqu'il s'agit d'une boucle infinie, ce qui signifie que **actionPerformed()** ne s'achève jamais. Comme vous êtes bloqué dans **actionPerformed()** par la première touche appuyée, le programme ne peut pas capturer les autres événements. (Pour sortir vous devez tué le processus; le moyen le plus facile de le faire est de presser les touches Control-C dans la console, si vous avez lancé le programme depuis la console. Si vous l'avez démarrer dans un navigateur, vous devez fermer la fenêtre du navigateur.)

Le problème de base ici est que **go()** doit continuer de réaliser ses opérations, et dans le même temps elle doit retourner à la fonction appelante de façon à ce que **actionPerformed()** puisse se terminer et que l'interface utilisateur puisse continuer à répondre à l'utilisateur. Mais une méthode conventionnelle comme **go()** ne peut pas continuer *et* dans le même temps rendre le contrôle au reste du programme. Cela à l'air d'une chose impossible à accomplir, comme si le CPU devait être à deux endroits à la fois, mais c'est précisément l'illusion que les threads permettent.

Le modèle des threads (et son support de programmation Java) est une commodité de programmation pour simplifier le jonglage entre plusieurs opérations simultanées dans un même programme. Avec les threads, le temp CPU sera éclaté et distribué entre les différentes threads. Chaque thread a l'impression d'avoir constamment le CPU pour elle toute seule, mais le temps CPU est distribué entre les différentes threads. L'exception à cela est si votre programme tourne sur plusieurs CPUs. Mais ce qui est intéressant dans le threading c'est de permettre une abstraction par rapport à cette couche, votre code n'a pas besoin de savoir si il tournera sur un ou plusieurs CPUs. Ainsi, les threads sont un moyen de créer de façon transparente des programmes portables.

Le threading réduit l'efficacité informatique, mais la net amélioration dans la conception des programmes, la gestion de ressources *[resource balancing]* , et le confort d'utilisation est souvent valable. Bien sûr, si vous avez plus d'un CPU, alors le système d'exploitation pourra décider quel CPU attribué à quel jeux de threads ou attribué une seule thread et la totalité du programme pourra s'exécuter beaucoup plus vite. Le multi-tâche et le multithreading tendent à être l'approche la plus raisonnable pour utiliser les systèmes multi-processeurs.

Héritage de Thread

Le moyen le plus simple de créer une thread est d'hériter de la classe **Thread**, qui a toutes les connexions nécessaires pour créer et faire tourner les threads. La plus importante méthode de **Thread** est **run()**, qui doit être redéfinie pour que le thread fasse ce que vous lui demander. Ainsi, **run()**, contient le code qui sera exécuté « simultanément » avec les autres threads du programme.

L'exemple suivant créée un certain nombre de threads dont il garde la trace en attribuant à chaque thread un numéro unique, généré à l'aide d'une variable **static**. La méthode **run()** du **Thread** est redéfinie pour décrémenter le compteur à chaque fois qu'elle passe dans sa boucle et se termine quand le compteur est à zéro (au moment où **run()** retourne, le thread se termine).

```
//: c14:SimpleThread.java
```

```
// Un example très simple de Threading.

public class SimpleThread extends Thread {
  private int countDown = 5;
  private static int threadCount = 0;
  private int threadNumber = ++threadCount;
  public SimpleThread() {
    System.out.println("Making " + threadNumber);
  }
  public void run() {
    while(true) {
      System.out.println("Thread " +
        threadNumber + "(" + countDown + "#004488">")");
      if(--countDown == 0) return;
    }
  }
  public static void main(String[] args) {
    for(int i = 0; i < 5; i++)
      new SimpleThread().start();
    System.out.println("All Threads Started");
  }
} ///:~
```

Une méthode **run()** a toujours virtuellement une sorte de boucle qui continue jusqu'à ce que le thread ne soit plus nécessaire, ainsi vous pouvez établir la condition sur laquelle arrêter cette boucle (ou, comme dans le cas précédent, simplement retourner de **run()**). Souvent, **run()** est transformé en une boucle infinie, ce qui signifie que à moins qu'un facteur extérieur ne cause la terminaison de **run()**, elle continuera pour toujours.

Dans **main()** vous pouvez voir un certain nombre de threads créé et lancée. La méthode **start()** de la classe **Thread** procède à une initialisation spéciale du thread et appelle ensuite **run()**. Les étapes sont donc: le constructeur est appelé pour construire l'objet, puis **start()** configure le thread et appelle **run()**. Si vous n'appelez pas **start()** (ce que vous pouvez faire dans le constructeur, si c'est approprié) le thread ne sera jamais démarré.

La sortie d'une exécution de ce programme (qui peut être différente d'un exécution à l'autre) est:

```
Making 1
Making 2
Making 3
Making 4
Making 5
Thread 1(5)
Thread 1(4)
Thread 1(3)
Thread 1(2)
Thread 2(5)
Thread 2(4)
```

```
Thread 2(3)
Thread 2(2)
Thread 2(1)
Thread 1(1)
All Threads Started
Thread 3(5)
Thread 4(5)
Thread 4(4)
Thread 4(3)
Thread 4(2)
Thread 4(1)
Thread 5(5)
Thread 5(4)
Thread 5(3)
Thread 5(2)
Thread 5(1)
Thread 3(4)
Thread 3(3)
Thread 3(2)
Thread 3(1)
```

Vous aurez remarqué que nulle part dans cet exemple **sleep()** n'est appelé, mais malgré cela la sortie indique que chaque thread a eu une portion de temps CPU dans lequel s'exécuter. Cela montre que **sleep()**, bien que relié à l'existence d'un thread pour pouvoir s'exécuter, n'est pas impliqué dans l'activation et la désactivation du threading. C'est simplement une autre méthode [NDT: de la classe Thread].

Vous pouvez aussi voir que les threads ne sont pas exécutés dans l'ordre où ils sont créés. En fait, l'ordre dans lequel le CPU s'occupe d'un ensemble de threads donné est indéterminé, à moins que vous ne rentriez dans l'ajustement des priorités en utilisant la méthode **setPriority()** de **Thread**.

Quand **main()** créée les objets **Thread** elle le capture aucune des références sur ces objets. Un objet ordinaire serait la proie rêvée pour le garbage collector, mais pas un **Thread**. Chaque **Thread** « s'enregistre » lui-même, il y a alors quelque part une référence sur celui-ci donc le garbage collector ne peut pas le détruire.

Threading pour une une interface réactive

Il est maintenant possible de résoudre le problème de **Counter1.java** avec un thread. Le truc est de placer une sous-tâche — est la boucle de la méthode **go()** — dans la méthode **run()** du thread. Quand l'utilisateur presse le bouton **start**, le thread est démarré, mais quand la *creation* du thread est terminé, donc que le thread tourne, le principal travail du programme (chercher et répondre aux événements de l'interface utilisateur) peut continuer. Voici la solution:

```
//: c14:Counter2.java
// Une interface utilisateur réactive grace aux threads.
// <applet code=Counter2 width=300 height=100>
// </applet>
import javax.swing.*;
```

```
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

public class Counter2 extends JApplet {
  private class SeparateSubTask color="#0000ff">extends Thread {
    private int count = 0;
    private boolean runFlag = color="#0000ff">true;
    SeparateSubTask() { start(); }
    void invertFlag() { runFlag = !runFlag; }
    public void run() {
      while (true) {
       try {
        sleep(100);
       } catch(InterruptedException e) {
        System.err.println("Interrupted");
       }
       if(runFlag)
        t.setText(Integer.toString(count++));
      }
    }
  }
  private SeparateSubTask sp = null;
  private JTextField t = new JTextField(10);
  private JButton
    start = new JButton("Start"),
    onOff = new JButton("Toggle");
  class StartL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      if(sp == null)
        sp = new SeparateSubTask();
    }
  }
  class OnOffL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      if(sp != null)
        sp.invertFlag();
    }
  }
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(t);
    start.addActionListener(new StartL());
    cp.add(start);
    onOff.addActionListener(new OnOffL());
    cp.add(onOff);
```

```
  }
  public static void main(String[] args) {
    Console.run(new Counter2 (), 300, 100);
  }
} ///:~
```

Counter2 est un programme simple, son seul travail est de préparer et maintenir l'interface utilisateur. Mais maintenant, quand l'utilisateur presse le bouton **start**, le code gérant l'événement n'appelle plus de méthode. Au lieu de ça un thread de la classe **SeparateSubTask** est créer et la boucle d'événements de **Counter2** continue.

La classe **SeparateSubTask** est une simple extension de **Thread** avec un constructeur qui lance le thread en appelant **start()**, et un **run()** qui contient essentiellement le code de « **go()** » de **Counter1.java**.

La classe **SeparateSubTask** étant une classe interne, elle peut accéder directement au **JTextField t** dans **Counter2**; comme vous pouvez le voir dans **run()**. **SeparateSubTask** peut accéder au champ **t** sans permission spéciale malgré que ce champ soit déclaré **private** dans la classe externe — il est toujours bon de rendre les champs « aussi privé que possible » de façon à ce qu'ils ne soient pas accidentellement changés à l'extérieur de votre classe.

Quand vous pressez le bouton **onOff**, **runFlag** est inversé dans l'objet **SeparateSubTask**. Ce thread (quand il regarde le flag) peut se démarrer ou s'arrêter tout seul. Presser le bouton **onOff** produit un temps de réponse apparent. Bien sûr, la réponse n'est pas vraiment instantanée contrairement à ce qui se passe sur un système fonctionnant par interruptions. Le compteur ne s'arrête que lorsque le thread a le CPU et s'aperçoit que le flag a changé.

Vous pouvez voir que la classe interne **SeparateSubTask**est **private**, ce qui signifie que l'on peut donner à ces champs et méthodes l'accès par défaut (excepté pour **run()** qui doit être **public** puisque elle est **public** dans la classe de base). La classe interne **private** ne peut être accéder par personne sauf **Counter2**, les deux classes sont ainsi fortement couplés. Chaque fois que vous constatez que deux classes apparaissent comme fortement couplées, vous remarquerez le gain en codage et en maintenance que les classes internes peuvent apportées.

Dans l'exemple ci-dessus, vous pouvez voir que la classe thread est séparée de la classe principale du programme. C'est la solution la plus sensé et c'est relativement facile à comprendre. Il existe toutefois une forme alternative qui vous verrez souvent utiliser, qui n'est pas aussi claire mais qui est souvent plus concise (ce qui augmente probablement sa popularité). Cette forme combine la classe du programme principal avec la classe thread en faisant de la classe du programme principal un thread. Comme pour un programme GUI la classe du programme principal doit hérité soit de **Frame** soit d'**Applet**, une interface doit être utilisée pour coller à la nouvelle fonctionnalité. Cette interface est appelée **Runnable**, elle contient la même méthode de base que **Thread**. En fait, **Thread** implémente aussi **Runnable**, qui spécifie seulement qu'il y a une méthode **run()**

L'utilisation de la combinaison programme/thread n'est pas vraiment évidente. Quand vous démarrez le programme, vous créez un objet **Runnable**, mais vous ne démarrez pas le thread. Ceci doit être fait explicitement. C'est ce que vous pouvez voir dans le programme qui suit, qui reproduit la fonctionnalité de **Counter2**:

```
//: c14:Counter3.java
// Utilisation de l'interface Runnable pour
```

```
// transformer la classe principale en thread.
// <applet code=Counter3 width=300 height=100>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

public class Counter3
    extends JApplet implements Runnable {
  private int count = 0;
  private boolean runFlag = true;
  private Thread selfThread = null;
  private JButton
    start = new JButton("Start"),
    onOff = new JButton("Toggle");
  private JTextField t = new JTextField(10);
  public void run() {
    while (true) {
      try {
        selfThread.sleep(100);
      } catch(InterruptedException e) {
        System.err.println("Interrupted");
      }
      if(runFlag)
        t.setText(Integer.toString(count++));
    }
  }
  class StartL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      if(selfThread == null) {
        selfThread = new Thread(Counter3.this);
        selfThread.start();
      }
    }
  }
  class OnOffL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      runFlag = !runFlag;
    }
  }
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(t);
    start.addActionListener(new StartL());
    cp.add(start);
```

```
    onOff.addActionListener(new OnOffL());
    cp.add(onOff);
  }
  public static void main(String[] args) {
    Console.run(new Counter3(), 300, 100);
  }
} ///:~
```

Maintenant **run()** est définie dans la classe, mais reste en sommeil après la fin de **init()**. Quand vous pressez le bouton **start**, le thread est créé (si il n'existe pas déjà) dans cette expression obscure:

```
new Thread(Counter3.this);
```

Quand une classe a une interface **Runnable**, cela signifie simplement qu'elle définit une méthode **run()**, mais n'entraîne rien d'autre de spécial — contrairement à une classe héritée de thread elle n'a pas de capacité de threading naturelle. Donc pour produire un thread à partir d'un objet **Runnable** vous devez créer un objet **Thread** séparé comme ci-dessus, passer l'objet **Runnable** au constructeur spécial de **Thread**. Vous pouvez alors appeler **start()** sur ce thread:

```
selfThread.start();
```

L'initialisation usuelle est alors effectuée puis la méthode **run()** est appelé.

L'aspect pratique de l'interface **Runnable** est que tout appartient à la même classe. Si vous avez besoin d'accéder à quelque chose, vous le faites simplement sans passer par un objet séparé. Cependant, comme vous avez pu le voir dans le précédent exemple, cet accès est aussi facile en utilisant des classes internes. .

Créer plusieurs threads

Considérons la création de plusieurs threads différents. Vous ne pouvez pas le faire avec l'exemple précédent, vous devez donc revenir à plusieurs classes séparées, héritant de **Thread** pour encapsuler **run()**. Il s'agit d'une solution plus générale et facile à comprendre. Bien que l'exemple précédent montre un style de codage que vous rencontrerez souvent, je ne vous le recommande pas pour la plupart des cas car ce style est juste un peu plus confus et moins flexible.

L'exemple suivant reprend la forme des exemples précédents avec des compteurs et des boutons bascules. Mais maintenant toute l'information pour un compteur particulier, le bouton et le champ texte inclus, est dans son propre objet qui hérite de **Thread**. Tous les champs de **Ticker** sont **private**, ce qui signifie que l'implémentation de **Ticker** peut changer à volonté, ceci inclus la quantité et le type des composants pour acquérir et afficher l'information. Quand un objet **Ticker** est créé, le constructeur ajoute ces composants au content pane de l'objet externe:

```
//: c14:Counter4.java
// En conservant votre thread comme une classe distincte
// vous pouvez avoir autant de threads que vous voulez
// <applet code=Counter4 width=200 height=600>
// <param name=size value="12"></applet>
import javax.swing.*;
```

```
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

public class Counter4 extends JApplet {
  private JButton start = new JButton("Start");
  private boolean started = false;
  private Ticker[] s;
  private boolean isApplet = true;
  private int size = 12;
  class Ticker extends Thread {
    private JButton b = new JButton(color="#004488">"Toggle");
    private JTextField t = new JTextField(10);
    private int count = 0;
    private boolean runFlag = color="#0000ff">true;
    public Ticker() {
      b.addActionListener(new ToggleL());
      JPanel p = new JPanel();
      p.add(t);
      p.add(b);
      // Appelle JApplet.getContentPane().add():
      getContentPane().add(p);
    }
    class ToggleL implements ActionListener {
      public void actionPerformed(ActionEvent e) {
        runFlag = !runFlag;
      }
    }
    public void run() {
      while (true) {
        if (runFlag)
          t.setText(Integer.toString(count++));
        try {
          sleep(100);
        } catch(InterruptedException e) {
          System.err.println("Interrupted");
        }
      }
    }
  }
  class StartL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      if(!started) {
        started = true;
        for (int i = 0; i < s.length; i++)
          s[i].start();
      }
```

```
    }
  }
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    // Obtient le paramètre "size" depuis la page Web
    if (isApplet) {
      String sz = getParameter("size");
      if(sz != null)
        size = Integer.parseInt(sz);
    }
    s = new Ticker[size];
    for (int i = 0; i < s.length; i++)
      s[i] = new Ticker();
    start.addActionListener(new StartL());
    cp.add(start);
  }
  public static void main(String[] args) {
    Counter4 applet = new Counter4();
    // Ce n'est pas une applet, donc désactive le flag et
    // produit la valeur du paramètre depuis les arguments:
    applet.isApplet = false;
    if(args.length != 0)
      applet.size = Integer.parseInt(args[0]);
    Console.run(applet, 200, applet.size * 50);
  }
} ///:~
```

Ticker contient non seulement l'équipement pour le threading, mais aussi le moyen de contrôler et d'afficher le thread. Vous pouvez créer autant de threads que vous voulez sans créer explicitement les composants graphiques.

On utilise dans **Counter4** un tableau d'objets **Ticker** appelé **s**. Pour un maximum de flexibilité, la taille du tableau est initialisée en utilisant les paramètres de l'applet donnés dans la page Web. Voici ce à quoi ressemble le paramètre de taille sur la page, inclus dans le tag d'applet:

```
<param name=size value="20">
```

Les mots **param**, **name**, et **value** sont tous des mots clés HTML. **name** est ce à quoi vous ferez référence dans votre programme, et **value** peut être n'importe quelle chaîne, pas seulement quelque chose qui s'analyse comme un nombre.

Vous remarquerez que la détermination de la taille du tableau **s** est faite dans la méthode **init()**, et non comme une définition en ligne de **s**. En fait, vous *ne pouvez pas* écrire dans la définition de la classe (en dehors de toutes méthodes):

```
int size = Integer.parseInt(getParameter("#004488">"size"));
Ticker[] s = new Ticker[size];
```

Vous pouvez compiler ce code, mais vous obtiendrez une étrange « null-pointer exception » à

l'exécution. Cela fonctionne correctement si vous déplacez l'initialisation avec **getParameter()** dans **init()**. Le framework de l'applet réalise les opérations nécessaires à l'initialisation pour récupérer les paramètres avant d'entrer dans **init()**.

En plus, ce code est prévu pour être soit une applet, soit une application. Si c'est une application l'argument **size** est extrait de la ligne de commande (ou une valeur par défaut est utilisée).

Une fois la taille du tableau établie, les nouveaux objets **Ticker** sont créés; en temps que partie du constructeur, les boutons et champs texte sont ajoutés à l'applet pour chaque **Ticker**.

Presser le bouton **start** signifie effectuer une boucle sur la totalité du tableau de **Ticker**s et appeler **start()** pour chacun d'eux. N'oubliez pas que **start()** réalise l'initialisation du thread nécessaire et appelle ensuite **run()** pour chaque thread.

Le listener **ToggleL** inverse simplement le flag dans **Ticker** et quand le thread associé en prend note il peut réagir en conséquence.

Un des intérêt de cet exemple est qu'il vous permet de créer facilement un grand jeux de sous-tâches indépendantes et de contrôler leur comportement. Avec ce programme, vous pourrez voir que si on augmente le nombre de sous-tâches, votre machine montrera probablement plus de divergence dans les nombres affichés à cause de la façon dont les threads sont servis.

Vous pouvez également expérimenter pour découvrir combien la méthode **sleep(100)** est importante dans **Ticker.run()**. Si vous supprimer **sleep()**, les choses vont bien fonctionner jusqu'à ce que vous pressiez un bouton. Un thread particulier à un **runFlag** faux et **run()** est bloquer dans une courte boucle infinie, qu'il parait difficile d'arrêter au cours du multithreading, la dynamique et la vitesse du programme le fait vraiment boguer. *[so the responsiveness and speed of the program really bogs down.]*

Threads démons

Un thread démon est un thread qui est supposé proposer un service général en tache de fond aussi longtemps que le programme tourne, mais il n'est pas l'essence du programme. Ainsi, quand tous les threads non démons s'achève, le programme se termine. A l'inverse, si des threads non démons tournent encore le programme ne se termine pas. (Il y a, par exemple, un thread qui tourne **main()**.)

Vous pouvez savoir si un thread est un démon en appelant **isDaemon()**, et vous pouvez activer ou désactiver la « démonité » *[« daemonhood »]* d'un thread avec **setDaemon()**. Si un thread est un démon, alors toutes les threads qu'il créera seront automatiquement des démons.

L'exemple suivant montre des threads démons:

```
//: c14:Daemons.java
// Un comportement démoniaque

import java.io.*;

class Daemon extends Thread {
  private static final int SIZE = 10;
  private Thread[] t = new Thread[SIZE];
  public Daemon() {
    setDaemon(true);
```

```
    start();
  }
  public void run() {
    for(int i = 0; i < SIZE; i++)
      t[i] = new DaemonSpawn(i);
    for(int i = 0; i < SIZE; i++)
      System.out.println(
        "t[" + i + "].isDaemon() = "
        + t[i].isDaemon());
    while(true)
      yield();
  }
}

class DaemonSpawn extends Thread {
  public DaemonSpawn(int i) {
    System.out.println(
      "DaemonSpawn " + i + " started");
    start();
  }
  public void run() {
    while(true)
      yield();
  }
}

public class Daemons {
  public static void main(String[] args)
  throws IOException {
    Thread d = new Daemon();
    System.out.println(
      "d.isDaemon() = " + d.isDaemon());
    // Autorise le thread démon à finir
    // son processus de démarrage
    System.out.println("Press any key");
    System.in.read();
  }
} ///:~
```

Le thread **Daemon** positionne son flag démon à « vrai », il engendre alors un groupe d'autre threads pour montrer qu'ils sont aussi des démons. Il tombe alors dans une boucle infinie qui appelle **yield()** pour rendre le contrôle aux autres processus. Dans une première version de ce programme, la boucle infinie incrémentait un compteur **int**, mais cela semble entraîner un arrêt du programme. Utiliser **yield()** rend le programme plutôt nerveux.

Il n'y a rien pour empêcher le programme de se terminer une fois que **main()** a fini sont travail, puisqu'il n'y a plus que des threads démons qui tournent. Vous pouvez donc voir le résultat du démarrage de tous les threads démons, on appelle read sur **System.in** pour que le programme at-

tende qu'une touche soit pressée avant de s'arrêter. Sans cela vous ne voyez qu'une partie des résultats de la création des threads démons. (Essayez de remplacer l'appel à **read()** par des appels à **sleep()** de différents tailles pour voir ce qui se passe.)

Partager des ressources limitées

Vous pouvez penser un programme à une seule thread comme une seule entité se déplaçant dans votre espace problème et n'effectuant qu'une seule chose à la fois. Puisqu'il y a seulement une entité, vous n'avez jamais penser au problème de deux entité essayant d'utiliser la même ressource en même temps, comme deux personnes essayant de se garer sur la même place, passer la même porte en même temps, ou encore parler en même temps.

Avec le multithreading, les chose ne sont plus seules, mais vous avez maintenant la possibilité d'avoir deux ou trois threads essayant d'utiliser la même ressource limitée à la fois. Les collisions sur une ressource doivent être évitées ou alors vous aurez deux threads essayant d'accéder au même compte bancaire en même temps, imprimer sur la même imprimante, ou ajuster la même soupape, etc...

Considérons une variation sur les compteurs déjà utilisées plus haut dans ce chapitre. Dans l'exemple suivant, chaque thread contient deux compteurs incrémentés et affichés dans **run()**. En plus, un autre thread de la classe **Watcher** regarde les compteurs pour voir si ils restent toujours les même. Ceci apparaît comme une activité inutile puisqu'en regardant le code il apparaît évident que que les compteurs resteront toujours les même. Mais c'est là que la surprise arrive. Voici la première version du programme:

```
//: c14:Sharing1.java
// Les problèmes avec le partage
// de ressource et les threads
// <applet code=Sharing1 width=350 height=500>
// <param name=size value="12">
// <param name=watchers value="15">
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

public class Sharing1 extends JApplet {
  private static int accessCount = 0;
  private static JTextField aCount =
    new JTextField("0", 7);
  public static void incrementAccess() {
    accessCount++;
    aCount.setText(Integer.toString(accessCount));
  }
  private JButton
    start = new JButton("Start"),
    watcher = new JButton("Watch");
  private boolean isApplet = true;
```

```
  private int numCounters = 12;
  private int numWatchers = 15;
  private TwoCounter[] s;
  class TwoCounter extends Thread {
    private boolean started = color="#0000ff">false;
    private JTextField
      t1 = new JTextField(5),
      t2 = new JTextField(5);
    private JLabel l =
      new JLabel("count1 == count2");
    private int count1 = 0, count2 = 0;
    // Ajoute les composants visuels comme un panel
    public TwoCounter() {
      JPanel p = new JPanel();
      p.add(t1);
      p.add(t2);
      p.add(l);
      getContentPane().add(p);
    }
    public void start() {
      if(!started) {
        started = true;
        super.start();
      }
    }
    public void run() {
      while (true) {
        t1.setText(Integer.toString(count1++));
        t2.setText(Integer.toString(count2++));
        try {
          sleep(500);
        } catch(InterruptedException e) {
          System.err.println("Interrupted");
        }
      }
    }
    public void synchTest() {
      Sharing1.incrementAccess();
      if(count1 != count2)
        l.setText("Unsynched");
    }
  }
  class Watcher extends Thread {
    public Watcher() { start(); }
    public void run() {
      while(true) {
        for(int i = 0; i < s.length; i++)
```

```
          s[i].synchTest();
        try {
          sleep(500);
        } catch(InterruptedException e) {
          System.err.println("Interrupted");
        }
      }
    }
  }
  class StartL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      for(int i = 0; i < s.length; i++)
        s[i].start();
    }
  }
  class WatcherL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      for(int i = 0; i < numWatchers; i++)
        new Watcher();
    }
  }
  public void init() {
    if(isApplet) {
      String counters = getParameter("size");
      if(counters != null)
        numCounters = Integer.parseInt(counters);
      String watchers = getParameter("watchers");
      if(watchers != null)
        numWatchers = Integer.parseInt(watchers);
    }
    s = new TwoCounter[numCounters];
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    for(int i = 0; i < s.length; i++)
      s[i] = new TwoCounter();
    JPanel p = new JPanel();
    start.addActionListener(new StartL());
    p.add(start);
    watcher.addActionListener(new WatcherL());
    p.add(watcher);
    p.add(new JLabel("Access Count"));
    p.add(aCount);
    cp.add(p);
  }
  public static void main(String[] args) {
    Sharing1 applet = new Sharing1();
    // Ce n'est pas une applet, donc désactive le flag et
```

```
    // produit la valeur du paramètre depuis les arguments:
    applet.isApplet = false;
    applet.numCounters =
      (args.length == 0 ? 12 :
        Integer.parseInt(args[0]));
    applet.numWatchers =
      (args.length < 2 ? 15 :
        Integer.parseInt(args[1]));
    Console.run(applet, 350,
      applet.numCounters * 50);
  }
} ///:~
```

Comme avant, chaque compteur contient ses propres composants graphiques: deux champs textes et un label qui au départ indique que les comptes sont equivalents. Ces composants sont ajoutés au content pane de l'objet de la classe externe dans le constructeur de **TwoCounter**.

Comme un thread **TwoCounter** est démarré par l'utilisateur en pressant un bouton, il est possible que **start()** puisse être appelé plusieurs fois. Il est illégal d'appeler **Thread.start()** plusieurs fois pour une même thread (une exception est lancée). Vous pouvez voir la machinerie éviter ceci avec le flag **started** et la méthode redéfinie **start()**.

Dans **run()**, **count1** and **count2** sont incrémentés et affichés de manière à ce qu'ils devraient rester identiques. Ensuite **sleep()** est appelé; sans cet appel le programme feinte *[balks]* parce qu'il devient difficile pour le CPU de passer d'une tache à l'autre.

La méthode **synchTest()** effectue l'activité apparemment inutile de vérifier si **count1** est équivalent à **count2**; si ils ne sont pas équivalent elle positionne l'étiquette à « Unsynched » pour indiquer ceci. Mais d'abord, elle appelle une méthode statique de la classe **Sharing1** qui incrémente et affiche un compteur d'accès pour montrer combien de fois cette vérification c'est déroulée avec succès. (La raison de ceci apparaîtra dans les prochaines variations de cet exemple.)

La classe **Watcher** est un thread dont le travail est d'appeler **synchTest()** pour tous les objets **TwoCounter** actifs. Elle le fait en parcourant le tableau maintenu dans l'objet **Sharing1**. Vous pouvez voir le **Watcher** comme regardant constamment par dessus l'épaule des objets **TwoCounter**.

Sharing1 contient un tableau d'objets **TwoCounter** qu'il initialise dans **init()** et démarre comme thread quand vous pressez le bouton « start ». Plus tard, quand vous pressez le bouton « Watch », un ou plusieurs watchers sont créés et *freed upon the unsuspecting **TwoCounter** threads. [NDT: mon dico est gros mais j'y comprend rien...]*

Notez que pour faire fonctionner ceci en tant qu'applet dans un browser, votre tag d'applet devra contenir ces lignes:

```
<param name=size value="20">
<param name=watchers value="1">
```

Vous pouvez expérimenter le changement de la largeur (*width*), la hauteur (*height*), et des paramètres pour l'adapter à vos goûts. En changeant **size** et **watchers** vous changerez le comportement du programme. Ce programme est prévu pour fonctionner comme une application autonome en passant les arguments sur la ligne de commande (ou en utilisant les valeurs par défaut).

La surprise arrive ici. Dans **TwoCounter.run()**, la boucle infinie se répète en exécutant seulement les deux lignes suivantes:

```
t1.setText(Integer.toString(count1++));
t2.setText(Integer.toString(count2++));
```

(elle dort aussi mais ce n'est pas important ici). En exécutant le programme, vous découvrirez que **count1** et **count2** seront observés (par les **Watcher**s) comme étant inégaux par moments. A ce moment, la suspension a eu lieu *entre* l'exécution des deux lignes précédentes, et le thread **Watcher** s'est exécuté et a effectuée la comparaison juste à ce moment, trouvant ainsi les deux compteurs différents.

Cet exemple montre un problème fondamental de l'utilisation des threads. Vous ne savez jamais quand un threads sera exécuté. Imaginez-vous assis à une table avec une fourchette, prêt à piquer la dernière bouchée de nourriture dans votre assiette et comme votre fourchette est prête à la prendre, la bouchée disparaît soudainement (parce que votre thread a été suspendu et qu'un autre thread est venu voler votre nourriture). C'est ce problème qu'il faut traiter.

Quelques fois vous ne faites pas attention si une ressource est accéder en même temps que vous essayez de l'utiliser (la bouchée est sur une autre assiette). Mais pour que le multithreading fonctionne, vous avez besoin d'un moyen d'empêcher que deux threads accèdent à la même ressource, particulièrement durant les périodes critiques.

Prévoir ce type de collisions est simplement un problème de placer un verrou sur une ressource quand un thread y accède. Le premier thread qui accède à la ressource la verrouille, ensuite les autres threads ne peuvent plus accéder à cette ressource jusqu'à ce qu'elle soit déverrouillée, à chaque fois un autre thread la verrouille et l'utilise, etc. Si le siège avant d'une voiture est la ressource limitée, les enfants qui crient « à moi! » revendiquent le verrou.

Comment Java partage les ressources

Java possède un support intégré pour prévoir les collisions sur un type de ressources: la memoire est un objet. Alors que vous rendez typiquement les éléments de données d'une classe **private** et accéder à cette mémoire seulement à travers des méthodes, vous pouvez prévoir les collisions en rendant une méthode particulière **synchronized**. Un seul thread à la fois peut appeler une méthode **synchronized** pour un objet particulier (bien que ce thread puisse appeler plus d'une des méthodes synchronized sur cet objet). Voici des méthodes **synchronized** simples:

```
synchronized void f() { /* ... */ }
synchronized void g(){ /* ... */ }
```

Chaque objet contient un seul lock (également appelé *monitor*) qui fait automatiquement partie de l'objet (vous n'avez pas à écrire de code spécial). Quand vous appeler une méthode **synchronized**, cet objet est verrouillé et aucune autre méthode **synchronized** de cet objet ne peut être appelé jusqu'à ce que la première se termine et libère le verrou. Dans l'exemple précédent, si **f()** est appelé pour un objet, **g()** ne peut pas être appelé pour le même objet jusqu'à ce que **f()** soit terminé et libère le verrou. Ainsi, il y a un seul verrou partagé par toutes les méthodes **synchronized** d'un objet particulier et ce verrou protège la mémoire commune de l'écriture par plus d'une méthode à un instant donné (i.e., plus d'un thread à la fois).

Il y a aussi un seul verrou par classe (appartenant à l'objet **Class** pour la classe), ainsi les méthodes **synchronized static** peuvent verrouiller les autres empêchant un accès simultané aux don-

nées **static** sur la base de la classe.

Remarquez que si vous voulez protéger d'autres ressources contre un accès simultanée par des threads multiples, vous pouvez le faire en forçant l'accès à d'autre ressource en passant par des méthodes synchronized.

Muni de ce nouveau mot clé la solution est entre nos mains: nous utiliserons simplement le mot clé **synchronized** pour les méthodes de **TwoCounter**. L'exemple suivant est le même que le précédent, avec en plus le nouveau mot clé:

```
//: c14:Sharing2.java
// Utilisant le mot clé synchronized pour éviter
// les accès multiples à une ressource particulière.
// <applet code=Sharing2 width=350 height=500>
// <param name=size value="12">
// <param name=watchers value="15">
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

public class Sharing2 extends JApplet {
  TwoCounter[] s;
  private static int accessCount = 0;
  private static JTextField aCount =
    new JTextField("0", 7);
  public static void incrementAccess() {
    accessCount++;
    aCount.setText(Integer.toString(accessCount));
  }
  private JButton
    start = new JButton("Start"),
    watcher = new JButton("Watch");
  private boolean isApplet = true;
  private int numCounters = 12;
  private int numWatchers = 15;

  class TwoCounter extends Thread {
    private boolean started = color="#0000ff">false;
    private JTextField
      t1 = new JTextField(5),
      t2 = new JTextField(5);
    private JLabel l =
      new JLabel("count1 == count2");
    private int count1 = 0, count2 = 0;
    public TwoCounter() {
      JPanel p = new JPanel();
```

```
    p.add(t1);
    p.add(t2);
    p.add(l);
    getContentPane().add(p);
  }
  public void start() {
    if(!started) {
      started = true;
      super.start();
    }
  }
  public synchronized void run() {
    while (true) {
      t1.setText(Integer.toString(count1++));
      t2.setText(Integer.toString(count2++));
      try {
        sleep(500);
      } catch(InterruptedException e) {
        System.err.println("Interrupted");
      }
    }
  }
  public synchronized void synchTest() {
    Sharing2.incrementAccess();
    if(count1 != count2)
      l.setText("Unsynched");
  }
}

class Watcher extends Thread {
  public Watcher() { start(); }
  public void run() {
    while(true) {
      for(int i = 0; i < s.length; i++)
        s[i].synchTest();
      try {
        sleep(500);
      } catch(InterruptedException e) {
        System.err.println("Interrupted");
      }
    }
  }
}
class StartL implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    for(int i = 0; i < s.length; i++)
      s[i].start();
```

```
    }
  }
  class WatcherL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      for(int i = 0; i < numWatchers; i++)
        new Watcher();
    }
  }
  public void init() {
    if(isApplet) {
      String counters = getParameter("size");
      if(counters != null)
        numCounters = Integer.parseInt(counters);
      String watchers = getParameter("watchers");
      if(watchers != null)
        numWatchers = Integer.parseInt(watchers);
    }
    s = new TwoCounter[numCounters];
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    for(int i = 0; i < s.length; i++)
      s[i] = new TwoCounter();
    JPanel p = new JPanel();
    start.addActionListener(new StartL());
    p.add(start);
    watcher.addActionListener(new WatcherL());
    p.add(watcher);
    p.add(new Label("Access Count"));
    p.add(aCount);
    cp.add(p);
  }
  public static void main(String[] args) {
    Sharing2 applet = new Sharing2();
    // Ce n'est pas une applet, donc place le flag et
    // récupère les valeurs de paramètres depuis args:
    applet.isApplet = false;
    applet.numCounters =
      (args.length == 0 ? 12 :
        Integer.parseInt(args[0]));
    applet.numWatchers =
      (args.length < 2 ? 15 :
        Integer.parseInt(args[1]));
    Console.run(applet, 350,
      applet.numCounters * 50);
  }
} ///:~
```

Vous noterez que *les deux méthodes* **run()** et **synchTest()** sont **synchronized**. Si vous syn-

chronizez seulement une des méthodes, alors l'autre est libre d'ignorer l'objet verrouillé et peut être appelé en toute impunité. C'est un point important: chaque méthode qui accède à une ressource partagée critique doit être **synchronized** ou ça ne fonctionnera pas correctement.

Maintenant un nouveau problème apparaît. Le **Watcher** ne peut jamais voir *[get a peek]* ce qui ce passe parce que la méthode **run()** est entièrement **synchronized**, et comme **run()** tourne toujours pour chaque objet le verrou est toujours fermé, **synchTest()** ne peut jamais être appelé. Vous pouvez le voir parce que **accessCount** ne change jamais.

Ce que nous aurions voulu pour cet exemple est un moyen d'isoler seulement *une partie* du code de **run()**. La section de code que vous voulez isoler de cette manière est appelé une *section critique* et vous utilisez le mot clé **synchronized** d'une manière différente pour créer une section critique. Java supporte les sections critiques à l'aide d'un *synchronized block;* cette fois **synchronized** est utilisé pour spécifier l'objet sur lequel le verrou est utilisé pour synchronizer le code encapsulé:

```
synchronized(syncObject) {
  // Ce code ne peut être accéder
  // que par un thread à la fois
}
```

Avant l'entrée dans le bloc synchronisé, le verrou doit être acquis sur **syncObject**. Si d'autres threads possède déjà le verrou, l'entrée dans le bloc est impossible jusqu'à ce que le verrou soit libéré.

L'exemple **Sharing2** peut être modifié en supprimant le mot clé **synchronized** de la méthode **run()** et de mettre à la place un bloc **synchronized** autour des deux lignes critiques. Mais quel objet devrait être utilisé comme verrou? Celui qui est déjà respecté par **synchTest()**, qui est l'objet courant (**this**)! Ainsi la méthode **run()** modifié ressemble à:

```
public void run() {
   while (true) {
     synchronized(this) {
       t1.setText(Integer.toString(count1++));
       t2.setText(Integer.toString(count2++));
     }
     try {
       sleep(500);
     } catch(InterruptedException e) {
       System.err.println("Interrupted");
     }
   }
 }
```

C'est le seul changement qui doit être fait à **Sharing2.java**, et vous verrez que bien que les compteurs ne soit jamais désynchronisés (d'après ce que **Watcher** est autorisé à voir d'eux), il y a toujours un accès adéquat fourni au **Watcher** pendant l'exécution de **run()**.

Bien sûr, toutes les synchronisations dépendent de la diligence du programmeur: chaque morceau de code qui peut accéder à une ressource partagée doit être emballé dans un bloc synchronisé approprié.

Efficacité de la synchronisation

Comme avoir deux méthodes écrivant dans le même morceau de données n'apparaît *jamais* comme une bonne idée, il semblerait avoir du sens que toutes les méthodes soit automatiquement **synchronized** et d'éliminer le mot clé **synchronized** ailleurs. (Bien sûr, l'exemple avec **synchronized run()** montre que ça ne fonctionnerait pas.) Mais il faut savoir qu'acquérir un verrou n'est pas une opération légère; cela multiplie le coût de l'appel de méthode (c'est à dire entrer et sortir de la méthode, pas exécuter le corps de cette méthode) par au moins quatre fois, et peut être très différent suivant votre implémentation. Donc si vous savez qu'une méthode particulière ne posera pas de problèmes particuliers il est opportun de ne pas utiliser le mot clé **synchronized**. D'un autre coté, supprimer le mot clé **synchronized** parce que vous pensez que c'est un goulot d'étranglement pour les performances en espérant qu'il n'y aura pas de collisions est une invitation au désastre.

JavaBeans revisités

Maintenant que vous comprenez la synchronisation, vous pouvez avoir un autre regard sur les Javabeans. Quand vous créez un Bean, vous devez assumez qu'il sera exécuté dans un environnement multithread. Ce qui signifie que:

1. Autant que possible, toutes les méthodes **public** d'un Bean devront être **synchronized**. Bien sûr, cela implique un désagrément lié à l'augmentation de temps d'exécution du à **synchronized**. Si c'est un problème, les méthodes qui ne poseront pas de problème de sections critiques peuvent être laissées non-**synchronized**, mais gardez en mémoire que ce n'est pas toujours aussi évident. Les méthodes qui donne l'accès aux attributs ont tendance à être petite (comme **getCircleSize()** dans l'exemple suivant) et/ou « atomique » en fait, l'appel de méthode exécute un si petit code que l'objet ne peut pas être changé durant l'exécution. Rendre ce type de méthodes non-**synchronized** ne devrait pas avoir d'effet important sur la vitesse d'exécution de votre programme. Vous devriez de même rendre toutes les méthodes **public** d'un Bean **synchronized** et supprimer le mot clé **synchronized** seulement quand vous savez avec certitude que c'est nécessaire et que ça fera une différence.

2. Quand vous déclenchez un *multicast* event à un banc de listeners intéressé par cet événement, vous devez vous assurer que tous les listeners seront ajoutés et supprimés durant le déplacement dans la liste.

Le premier point est assez facile à comprendre, mais le second point exige un petit effort. Considérez l'exemple **BangBean.java** presenté dans le précédent chapitre. Il évitait la question du multithreading en ignorant le mot clé **synchronized** (qui n'avait pas été encore introduit) et en rendant l'événement unicast. Voici cet exemple modifié pour fonctionner dans un environnement multitâche et utilisant le multicasting pour les événements:

```
//: c14:BangBean2.java
// Vous devriez écrire vos Beans de cette façon pour qu'ils
// puissent tournés dans un environement multithread.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.io.*;
import com.bruceeckel.swing.*;
```

```
public class BangBean2 extends JPanel
    implements Serializable {
  private int xm, ym;
  private int cSize = 20; // Taille du cercle
  private String text = "Bang!";
  private int fontSize = 48;
  private Color tColor = Color.red;
  private ArrayList actionListeners =
    new ArrayList();
  public BangBean2() {
    addMouseListener(new ML());
    addMouseMotionListener(new MM());
  }
  public synchronized int getCircleSize() {
    return cSize;
  }
  public synchronized void
  setCircleSize(int newSize) {
    cSize = newSize;
  }
  public synchronized String getBangText() {
    return text;
  }
  public synchronized void
  setBangText(String newText) {
    text = newText;
  }
  public synchronized int getFontSize() {
    return fontSize;
  }
  public synchronized void
  setFontSize(int newSize) {
    fontSize = newSize;
  }
  public synchronized Color getTextColor() {
    return tColor;
  }
  public synchronized void
  setTextColor(Color newColor) {
    tColor = newColor;
  }
  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.setColor(Color.black);
    g.drawOval(xm - cSize/2, ym - cSize/2,
      cSize, cSize);
```

```
  }
  // C'est un listener multicast, qui est
  // plus typiquement utilisé que l'approche
  // unicast utilisée dans BangBean.java:
  public synchronized void
    addActionListener(ActionListener l) {
    actionListeners.add(l);
  }
  public synchronized void
    removeActionListener(ActionListener l) {
    actionListeners.remove(l);
  }
  // Remarquez qu'elle n'est pas synchronized:
  public void notifyListeners() {
    ActionEvent a =
      new ActionEvent(BangBean2.this,
        ActionEvent.ACTION_PERFORMED, null);
    ArrayList lv = null;
    // Effectue une copie profonde de la liste au cas où
    // quelqu'un ajouterait un listener pendant que nous
    // appelons les listeners:
    synchronized(this) {
      lv = (ArrayList)actionListeners.clone();
    }
    // Apelle toutes les méthodes listeners:
    for(int i = 0; i < lv.size(); i++)
      ((ActionListener)lv.get(i))
        .actionPerformed(a);
  }
  class ML extends MouseAdapter {
    public void mousePressed(MouseEvent e) {
      Graphics g = getGraphics();
      g.setColor(tColor);
      g.setFont(
        new Font(
          "TimesRoman", Font.BOLD, fontSize));
      int width =
        g.getFontMetrics().stringWidth(text);
      g.drawString(text,
        (getSize().width - width) /2,
        getSize().height/2);
      g.dispose();
      notifyListeners();
    }
  }
  class MM extends MouseMotionAdapter {
    public void mouseMoved(MouseEvent e) {
```

```
        xm = e.getX();
        ym = e.getY();
        repaint();
      }
    }
    public static void main(String[] args) {
      BangBean2 bb = new BangBean2();
      bb.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e){
          System.out.println("ActionEvent" + e);
        }
      });
      bb.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e){
          System.out.println("BangBean2 action");
        }
      });
      bb.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e){
          System.out.println("More action");
        }
      });
      Console.run(bb, 300, 300);
    }
  } ///:~
```

Ajouter **synchronized** aux méthodes est un changement facile. Toutefois, motez que dans **addActionListener()** et **removeActionListener()** que les **ActionListener**s sont maintenant ajoutés et supprimés d'une **ArrayList**, ainsi vous pouvez en avoir autant que vous voulez.

La méthode **paintComponent()** n'est pas non plus **synchronized**. Décider de synchroniser les méthodes surchargées n'est pas aussi clair que quand vous ajoutez vos propres méthodes. Dans cet exemple, il semblerait que **paint()** fonctionne correctement qu'il soit **synchronized** ou non. Mais les points que vous devez considérés sont:

1. Cette méthode modifie-t-elle l'état des variables « critiques » de l'objet. Pour découvrir si les variables sont « critiques » vous devez déterminer si elles seront lues ou modifiées par d'autres threads dans le programme. (Dans ce cas, la lecture ou la modification sont pratiquement toujours réalisées par des méthodes **synchronized**, ainsi vous pouvez examiner juste celle-là.) Dans le cas de **paint()**, aucune modification n'apparaît.

2. Cette méthode dépend-elle de l'état de ces variables « critiques »? Si une méthode **synchronized** modifie une variable que votre méthode utilise, alors vous devriez vouloir mettre aussi votre méthode **synchronized**. En vous basant la dessus, vous devriez observer que **cSize** est changé par des méthodes **synchronized** et en conséquence **paint()** devrait être **synchronized**. Ici, pourtant, vous pouvez vous demander « Quelle serait la pire chose qui arriverai si **cSize** changeait durant **paint()**? » Quand vous voyez que ce n'est rien de trop grave, et un effet transitoire en plus, vous pouvez décider de laisser **paint()** un-**synchronized** pour éviter le temps supplémentaire de l'appel de méthode **synchronized**.

3. Un troisième indice est de noter si la version de la classe de base de **paint()** est **synchronized**, ce qui n'est pas le cas. Ce n'est pas un argument hermétique, juste un indice. Dans ce cas, par exemple, un champ qui *est* changé via des méthodes **synchronizes** (il s'agit de **cSize**) a été mélangé dans la formule de **paint()** et devrait avoir changé la situation. Notez, cependant, que **synchronized** n'est pas hérité — c'est à dire que si une méthode est **synchronized** dans une classe de base, alors il *n'est pas* automatiquement **synchronized** dans la version redéfinie de la classe dérivée .

Le code de test dans **TestBangBean2** a été modifié depuis celui du chapitre précédent pour démontrer la possibilité de multicast de **BangBean2** en ajoutant des listeners supplémentaires.

Blocage *[Blocking]*

Un thread peut être dans un des quatre états suivants:

1. *Nouveau (New)*: L'objet thread a été créé mais il n'a pas encore été démarré donc il ne peut pas tourné.

2. *Runnable*: Cela signifie qu'un thread *peut* tourner quand le mécanisme de découpage du temps a des cycles CPU disponibles pour le thread. Ainsi, le thread pourra ou non tourner, mais il n'y a rien pour l'empêcher de tourner si le scheduler peut l'organiser; il n'est ni mort ni bloqué.

3. *Mort (Dead)*: La façon normale pour un thread de mourir est de terminer sa méthode **run()**. Vous pouvez également appelé **stop()**, mais cela déclenche une exception qui est une sous-classe de **Error** (ce qui signifie que vous n'êtes pas obligé de placer l'appel dans un bloc **try**). Souvenez vous que déclencher une exception devrait être un événement spécial et non une partie de l'exécution normale du programme; ainsi l'utilisation de **stop()** est déprécier dans Java 2. Il y a aussi une méthode **destroy()** (qui n'a jamais été implémentée) qui vous ne devriez jamais utiliser si vous pouvez l'éviter puisqu'elle est radicale et ne libère pas les verrous de l'objet.

4. *Bloqué (Blocked)*: Le thread pourrait tourner mais quelque chose l'en empêche. Tant qu'un thread est dans un état bloqué le scheduler le sautera et ne lui donnera pas de temps CPU. Jusqu'à ce que le thread repasse dans un état runnable il ne réalisera aucune opérations.

Passer à l'état bloqué

L'état bloqué est le plus intéressant, et nécessite un examen. Un thread peut devenir bloqué pour cinq raisons:

1. Vous avez mis le thread à dormir en appelant **sleep(milliseconds),** auquel cas il ne tournera pas pendant le temps spécifé.

2. Vous avez suspendu l'exécution du thread avec **suspend()**. Il ne redeviendra pas runnable avant que le thread ne reçoive le message **resume()** (ces possibilités sont dépréciés dans Java2, et seront examinés plus tard).

3. Vous avez suspendu l'exécution du thread avec **wait()**. Il ne redeviendra pas runnable tant qu'il n'aura pas reçu l'un des messages **notify()** ou **notifyAll()**. (Oui, c'est comme le point 2, mais il y a une distinction qui sera vu plus tard.)

4. Le thread attend la fin d'une I/O.

5. Le thread essaye d'appelé une méthode **synchronized** sur un autre objet, et le verrou de cet objet n'est pas libre.

Vous pouvez aussi appelé **yield()** (une méthode de la classe **Thread**) pour volontairement donner le CPU afin que d'autres threads puisse tournées. Toutefois, il se passe la même chose si le scheduler décide que votre thread a eu assez de temps et saute à un autre thread. En fait, rien n'empêche le scheduler de partir de votre thread et de donner du temps à un autre thread. Quand un thread est bloqué, c'est qu'il y a une raison pour qu'il ne puisse continuer à tourner.

L'exemple suivant montre les cinq façons de devenir bloqué. Il existe en intégralité dans un seul fichier appelé **Blocking.java,** mais il sera examiné ici par morceau. (Vous remarquerez les tags « Continued » et « Continuing » qui permette à l'outil d'extraction de recoller les morceaux.)

Étant donné que cet exemple montre des méthodes dépréciées, vous obtiendrez des messages de depreciation lors de la compilation.

D'abord le programme de base:

```
//: c14:Blocking.java
// Démontre les différentes façon de
// bloqué un thread.
// <applet code=Blocking width=350 height=550>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import com.bruceeckel.swing.*;

////////////// Le framework de base ////////////
class Blockable extends Thread {
  private Peeker peeker;
  protected JTextField state = new JTextField(30);
  protected int i;
  public Blockable(Container c) {
    c.add(state);
    peeker = new Peeker(this, c);
  }
  public synchronized int read() { return i; }
  protected synchronized void update() {
    state.setText(getClass().getName()
      + " state: i = " + i);
  }
  public void stopPeeker() {
    // peeker.stop(); Déprécié en Java 1.2
    peeker.terminate(); // L'approche préférée
  }
}
```

```
class Peeker extends Thread {
  private Blockable b;
  private int session;
  private JTextField status = new JTextField(30);
  private boolean stop = false;
  public Peeker(Blockable b, Container c) {
    c.add(status);
    this.b = b;
    start();
  }
  public void terminate() { stop = color="#0000ff">true; }
  public void run() {
    while (!stop) {
      status.setText(b.getClass().getName()
        + " Peeker " + (++session)
        + "; value = " + b.read());
       try {
        sleep(100);
      } catch(InterruptedException e) {
        System.err.println("Interrupted");
      }
    }
  }
} ///:Continued
```

La classe **Blockable** est destinée à être la classe de base pour toutes les classes de cet exemple qui démontre le blocking. Un objet **Blockable** contient un **JTextField** appelé **state** qui est utilisé pour afficher l'information sur l'objet. La méthode qui affiche cette information est **update()**. Vous pouvez voir qu'elle utilise **getClass().getName()** pour produire le nom de la classe plutôt que de l'afficher directement; c'est parce que **update()** ne peut pas connaître le nom exact de la classe pour laquelle elle est appelée, puisque ce sera une classe dérivée de **Blockable**.

L'indicateur de changement dans **Blockable** est un **int i,** qui sera incrémenter par la méthode **run()** de la classe dérivé.

Il y a un thread de la classe **Peeker** qui est démarré pour chaque objet **Blockable**, et le travail de **Peeker** est de regarder son objet **Blockable** associé pour voir les changements de **i** en appelant **read()** et en les reportant dans son **status JTextField**. C'est important: Notez que **read()** et **update()** sont toutes les deux **synchronized**, ce qui signifie qu'elles nécessite que le verrou de l'objet soit libre.

Dormant (Sleeping)

Le premier test de ce programme est avec **sleep():**

```
///:Continuing
////////////// Bloqué via sleep() ///////////
class Sleeper1 extends Blockable {
  public Sleeper1(Container c) { super(c); }
```

```
  public synchronized void run() {
    while(true) {
      i++;
      update();
       try {
        sleep(1000);
      } catch(InterruptedException e) {
        System.err.println("Interrupted");
      }
    }
  }
}

class Sleeper2 extends Blockable {
  public Sleeper2(Container c) { super(c); }
  public void run() {
    while(true) {
      change();
       try {
        sleep(1000);
      } catch(InterruptedException e) {
        System.err.println("Interrupted");
      }
    }
  }
  public synchronized void change() {
      i++;
      update();
  }
} ///:Continued
```

Dans **Sleeper1** la méthode **run()** est entièrement **synchronized**. Vous verrez que le **Peeker** associé avec cet objet tournera tranquillement *jusqu'à* vous démarriez le thread, ensuite le **Peeker** sera gelé. C'est une forme de blocage: puisque **Sleeper1.run()** est **synchronized**, et une fois que le thread démarre dans **run()**, la méthode ne redonne jamais le verrou de l'objet et le **Peeker** est bloqué.

Sleeper2 fournit une solution en rendant **run()** un-**synchronized**. Seule la méthode **change()** est **synchronized**, ce qui signifie que tant que **run()** est dans **sleep()**, le **Peeker** peut accéder à la méthode **synchronized** dont il a besoin, nommée **read()**. Ici vous verrez que **Peeker** continue à tourner quand vous démarrez le thread **Sleeper2**.

Suspension et reprise

La partie suivante de l'exemple introduit le concept de suspension. La classe **Thread** a une méthode **suspend()** pour arrêter temporairement le thread et **resume()** qui le redémarre au point ou il était arrêté. **resume()** doit être appelé par un thread extérieur à celui suspendu, et dans ce cas il y a une classe séparée appelé **Resumer** qui fait juste ça. Chacune des classes démontrant suspend/re-

sume a un resumer associé:

```
///:Continuing
////////// Bloqué via suspend() //////////
class SuspendResume extends Blockable {
  public SuspendResume(Container c) {
    super(c);
    new Resumer(this);
  }
}

class SuspendResume1 extends SuspendResume {
  public SuspendResume1(Container c) { super(c);}
  public synchronized void run() {
    while(true) {
      i++;
      update();
      suspend(); // Déprecié en Java 1.2
    }
  }
}

class SuspendResume2 extends SuspendResume {
  public SuspendResume2(Container c) { super(c);}
  public void run() {
    while(true) {
      change();
      suspend(); // Déprecié en Java 1.2
    }
  }
  public synchronized void change() {
      i++;
      update();
  }
}

class Resumer extends Thread {
  private SuspendResume sr;
  public Resumer(SuspendResume sr) {
    this.sr = sr;
    start();
  }
  public void run() {
    while(true) {
      try {
        sleep(1000);
      } catch(InterruptedException e) {
        System.err.println("Interrupted");
```

```
        }
        sr.resume(); // Déprecié en Java 1.2
      }
    }
  } ///:Continued
```

SuspendResume1 a aussi une méthode **synchronized run()**. Une nouvelle fois, lorsque vous démarrerez ce thread vous verrez que son **Peeker** associé se bloque en attendant que le verrou devienne disponible, ce qui n'arrive jamais. Ce problème est corrigé comme précédemment dans **SuspendResume2**, qui ne place pas la méthode **run()** entièrement **synchronize** mais utilise une méthode **synchronized change()** séparée.

Vous devez être au courant du fait que Java 2 déprécie l'utilisation de **suspend()** et **resume()**, parce que **suspend()** prend le verrou et est sujet aux deadlock. En fait, vous pouvez facilement avoir un certain nombre d'objets verrouillés s'attendant les uns les autres, et cela entraînera le gel du programme. Alors que vous pouvez voir leurs utilisations dans des programmes plus vieux, vous ne devriez pas utiliser **suspend()** et **resume()**. La solution adéquate est décrite plus tard dans ce chapitre.

Attendre et notifier

Dans les deux premiers exemples, il est important de comprendre que ni **sleep()** ni **suspend()** ne libère le verrou lorsqu'ils sont appelés. Vous devez faire attention à cela en travaillant avec les verrous. D'un autre coté, la méthode **wait()** libère le verrou quand elle est appelée, ce qui signifie que les autres méthodes **synchronized** de l'objet thread peuvent être appelée pendant un **wait()**. Dans les deux classes suivantes, vous verrez que la méthode **run()** est totalement **synchronized** dans les deux cas, toutefois, le **Peeker** a encore un accès complet aux méthodes **synchronized** pendant un **wait()**. C'est parce que **wait()** libère le verrou sur l'objet quand il suspend la méthode dans laquelle il a été appelé.

Vous verrez également qu'il y a deux formes de **wait()**. La première prend un argument en milli-secondes qui a la même signification que dans **sleep()**: pause pour cette période de temps. La différence est que dans **wait()**, le verrou de l'objet est libéré *et* vous pouvez sortir du **wait()** à cause d'un **notify()** aussi bien que parce que le temps est écoulé.

La seconde forme ne prend pas d'arguments, et signifie que le **wait()** continuera jusqu'à ce qu'un **notify()** arrive et ne sera pas automatiquement terminé après un temps donné.

Le seul point commun entre **wait()** et **notify()** est que ce sont toutes les deux des méthodes de la classe de base **Object** et non une partie de **Thread** comme le sont **sleep()**, **suspend()**, et **resume()**. Alors que cela parait un peu étrange au début — avoir quelque chose exclusivement pour le threading comme partie de la classe de base universelle — c'est essentiel puisqu'elles manipulent le verrou qui fait aussi partie de chaque objet. Le résultat est que vous pouvez mettre un **wait()** dans n'importe quelle méthode **synchronized**, du fait qu'il y a du threading intervenant dans cette classe particulière. En fait, la *seule* place où vous pouvez appeler **wait()** ou **notify()** est dans une méthode ou un bloc **synchronized**. Si vous appelez **wait()** ou **notify()** dans une méthode qui n'est pas **synchronized**, le programme se compilera, mais quand vous l'exécuterez vous obtiendrez une **IllegalMonitorStateException** avec le message peu intuitif « current thread not owner. » Notez que **sleep()**, **suspend(),** et **resume()** peuvent toutes être appelées dans des méthodes non-**synchronized** puisqu'elle ne manipulent pas le verrou.

Vous pouvez appelez **wait()** ou **notify()** seulement pour votre propre verrou. De nouveau,

vous pouvez compiler un code qui essaye d'utiliser le mauvais verrou, mais il se produira le même message **IllegalMonitorStateException** que précédemment. Vous ne pouvez pas tricher avec le verrou de quelqu'un d'autre, mais vous pouvez demander à un autre objet de réaliser une opération qui manipule son verrou. Une approche possible est de créer une méthode **synchronized** qui appelle **notify()** pour son propre objet. Toutefois, dans **Notifier** vous verrez que **notify()** est appelé dans un bloc **synchronized**:

```
synchronized(wn2) {
  wn2.notify();
}
```

où **wn2** est l'objet de type **WaitNotify2**. Cette méthode, qui ne fait pas partie de **WaitNotify2**, acquiert le verrou sur l'objet **wn2**, à ce point il lui est possible d'appeler **notify()** pour **wn2** et vous n'obtiendrez pas d'**IllegalMonitorStateException**.

wait() est typiquement utilisé quand vous êtes arrivé à un point où vous attendez qu'une autre condition, sous le contrôle de forces extérieures à votre thread, change et que vous ne voulez pas attendre activement à l'intérieur du thread. Donc **wait()** vous autorise à mettre votre thread en sommeil en attendant que le monde change, et c'est seulement quand un **notify()** ou **notifyAll()** arrive que le thread se réveille et controle les changements. Ainsi, on dispose d'un moyen de synchronization entre les threads.

Bloqué sur I/O

Si un flux est en attente de l'activité d'une I/O, il se bloquera automatiquement. Dans la portion suivante de l'exemple, les deux classes travaillent avec les objets génériques **Reader** et **Writer**, mais dans le framework de test un piped stream sera créé afin de permettre au deux threads de se passer des données de façon sûre (ce qui est le but des piped streams).

Le **Sender** place des données dans le **Writer** et s'endort pour un temps tiré au hasard. Cependant **Receiver** n'a pas de **sleep()**, **suspend()**, ou **wait()**. Mais quand il appelle **read()** il se bloque automatiquement quand il n'y a pas d'autre données.

```
///:Continuing
class Sender extends Blockable { color="#009900">// envoie
  private Writer out;
  public Sender(Container c, Writer out) {
    super(c);
    this.out = out;
  }
  public void run() {
    while(true) {
      for(char c = 'A'; c <= 'z'; c++) {
        try {
          i++;
          out.write(c);
          state.setText("Sender sent: "  + (char)c);
          sleep((int)(3000 * Math.random()));
        } catch(InterruptedException e) {
          System.err.println("Interrupted");
```

```
        } catch(IOException e) {
          System.err.println("IO problem");
        }
      }
    }
  }
}

class Receiver extends Blockable {
  private Reader in;
  public Receiver(Container c, Reader in) {
    super(c);
    this.in = in;
  }
  public void run() {
    try {
      while(true) {
        i++; // Montre que peeker est en vie
        // Bloque jusqu'à ce que les caractères soient là:
        state.setText("Receiver read: "
          + (char)in.read());
      }
    } catch(IOException e) {
      System.err.println("IO problem");
    }
  }
} ///:Continued
```

Les deux classes placent également des informations dans leurs champs **state** et change **i** afin que le **Peeker** puisse voir que le thread tourne.

Tester

La classe principale de l'applet est étonnamment simple parce la majorité du travail a été mis dans le framework **Blockable**. En fait, un tableau d'objets **Blockable** est créé, et puisque chacun est un thread, il réalise leur propre activité quand vous pressez le bouton « start ». Il y a aussi un bouton et une clause **actionPerformed()** pour stopper tout les objets **Peeker**, qui donne une démonstration de de l'alternative à la méthode dépréciée **stop()** de **Thread**.

Pour établir la connexion entre les objets **Sender** et **Receiver**, un **PipedWriter** et un **PipedReader** sont créés. Notez que le **PipedReader in** doit être connecté au **PipedWriter out** via un argument du constructeur. Après ça, les données placées dans **out** peuvent être extraites de **in**, comme si elles passaient dans un tube (d'où le nom) (*[NDT: un pipe est un tube en anglais]*). Les objets **in** et **out** sont alors passés respectivement aux constructeurs de **Receiver** et **Sender**, qui les traitent comme des objets **Reader** et **Writer** (ils sont *upcast*).

Le tableau de références **Blockable b** n'est pas initialisé à son point de définition parce que les piped streams ne peuvent pas être établis avant cette définition (l'utilisation du bloc **try** évite cela).

```
///:Continuing
////////// Test de tout //////////
public class Blocking extends JApplet {
  private JButton
    start = new JButton("Start"),
    stopPeekers = new JButton("#004488">"Stop Peekers");
  private boolean started = false;
  private Blockable[] b;
  private PipedWriter out;
  private PipedReader in;
  class StartL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      if(!started) {
        started = true;
        for(int i = 0; i < b.length; i++)
          b[i].start();
      }
    }
  }
  class StopPeekersL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      // Demonstration de la meilleure
      // alternative à Thread.stop():
      for(int i = 0; i < b.length; i++)
        b[i].stopPeeker();
    }
  }
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    out = new PipedWriter();
    try {
      in = new PipedReader(out);
    } catch(IOException e) {
      System.err.println("PipedReader problem");
    }
    b = new Blockable[] {
      new Sleeper1(cp),
      new Sleeper2(cp),
      new SuspendResume1(cp),
      new SuspendResume2(cp),
      new WaitNotify1(cp),
      new WaitNotify2(cp),
      new Sender(cp, out),
      new Receiver(cp, in)
    };
    start.addActionListener(new StartL());
```

```
    cp.add(start);
    stopPeekers.addActionListener(
      new StopPeekersL());
    cp.add(stopPeekers);
  }
  public static void main(String[] args) {
    Console.run(new Blocking(), 350, 550);
  }
} ///:~
```

Dans **init()**, notez la boucle qui parcourt la totalité du tableau et ajoute les champs **state** et **peeker.status** à la page.

Quand les threads **Blockable** sont initialement créés, chacune crée et démarre automatiquement son propre **Peeker**. Donc vous verrez les **Peeker**s tournés avant que les threads **Blockable** ne démarrent. C'est important, puisque certains des **Peeker**s seront bloqués et stoperont quand les threads **Blockable** démarreront, et c'est essentiel de voir et de comprendre cet aspect particulier du blocage.

Interblocage *[Deadlock]*

Puisse que les threads peuvent être bloqués *et* puisse que les objets peuvent avoir des méthodes **synchronized** qui empêchent les threads d'accéder à cet objet jusqu'à ce que le verrou de synchronisation soit libéré, il est possible pour un thread de rester coincé attendant un autre thread, qui à son tour attend un autre thread, etc., jusqu'à ce que la chaîne ramène à un thread en attente sur le premier. Vous obtenez une boucle continue de threads s'attendant les uns les autres et aucun ne peut bouger. C'est ce qu'on appelle un *interblocage (ou deadlock)*. Le pire c'est que cela n'arrive pas souvent, mais quand cela vous arrive c'est frustrant à déboguer.

Il n'y a pas de support du langage pour aider à éviter les interblocages; c'est à vous de les éviter en faisant attention à la conception. Ce ne sont pas des mots pour rassurer la personne qui essaie de déboguer un programme générant des inter-blocages.

La dépréciation de stop(), suspend(), resume(), et destroy() en Java 2

Un changement qui a été fait dans Java 2 pour réduire les possibilités d'inter-blocage est la dépréciation des méthodes de **Thread’ stop()**, **suspend()**, **resume()**, et **destroy()**.

La méthode **stop()** est dépréciée parce qu'elle ne libère pas les verrous que le thread a acquis, et si les objets sont dans un état inconsistent (« damaged ») les autres threads peuvent les voir et les modifiés dans cet état. Le problème résultant peut être subtil et difficile à détecter. Plutôt que d'utiliser **stop()**, vous devriez suivre l'exemple de **Blocking.java** et utiliser un drapeau pour dire au thread quand se terminer en sortant de sa méthode **run()**.

Il existe des situations où un thread se bloque — comme quand il attend une entrée — mais il ne peut pas positionner un drapeau comme il le fait dans **Blocking.java**. Dans ces cas, vous ne devriez pas utiliser **stop()**, mais plutôt la méthode **interrupt()** de **Thread** pour écrire le code bloquant:

```
//: c14:Interrupt.java
// L'approche alternative pour utiliser
```

```
// stop() quand un thread est bloqué.
// <applet code=Interrupt width=200 height=100>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

class Blocked extends Thread {
  public synchronized void run() {
    try {
      wait(); // Bloque
    } catch(InterruptedException e) {
      System.err.println("Interrupted");
    }
    System.out.println("Exiting run()");
  }
}

public class Interrupt extends JApplet {
  private JButton
    interrupt = new JButton("Interrupt");
  private Blocked blocked = new Blocked();
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(interrupt);
    interrupt.addActionListener(
      new ActionListener() {
        public
        void actionPerformed(ActionEvent e) {
          System.out.println("Button pressed");
          if(blocked == null) color="#0000ff">return;
          Thread remove = blocked;
          blocked = null; // pour le libérer
          remove.interrupt();
        }
      });
    blocked.start();
  }
  public static void main(String[] args) {
    Console.run(new Interrupt(), 200, 100);
  }
} ///:~
```

Le **wait()** dans **Blocked.run()** produit le thread bloqué. Quand vous pressez le bouton, la référence **blocked** est placée à **null** donc le garbage collector le nettoiera, la méthode **interrupt()** de l'objet est alors appelée. La première fois que vous pressez le bouton vous verrez le thread sortir,

mais ensuite il n'y a plus de thread à tuer donc vous voyez juste que le bouton a été pressé.

Les méthodes **suspend()** et **resume()** finissent par être sujettes aux inter-blocages. Quand vous appelez **suspend()**, le thread cible s'arrête mais il conserve les verrous qu'il a acquis à ce point. Ainsi aucuns autres threads ne peut accéder aux ressources verrouillées jusqu'à ce que le thread soit redémarré. Un thread qui veut redémarrer le thread cible et aussi essaye d'utiliser une des ressources verrouillées produit un inter-blocage. Vous ne devriez pas utiliser **suspend()** et **resume()**, mais plutôt mettre un drapeau dans votre classe **Thread** pour indiqué si le thread devrait être actif ou suspendu. Si le drapeau indique que le thread est suspendu, le thread rentre dans un **wait()**. Quand le drapeau indique que le thread devrait être redémarré le thread est réactivé avec **notify()**. Un exemple peut être produit en modifiant **Counter2.java**. Alors que l'effet est similaire, vous remarquerez que l'organisation du code est assez différente — des classes internes anonymes sont utilisées pour tous les listeners et le **Thread** est une classe interne, ce qui rend la programmation légèrement plus convenable puisqu'on élimine certaines complexités nécessaires dans **Counter2.java**:

```
//: c14:Suspend.java
// L'approche alternative à l'utilisation de suspend()
// et resume(), qui sont déprecié dans Java 2.
// <applet code=Suspend width=300 height=100>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

public class Suspend extends JApplet {
  private JTextField t = new JTextField(10);
  private JButton
    suspend = new JButton("Suspend"),
    resume = new JButton("Resume");
  private Suspendable ss = new Suspendable();
  class Suspendable extends Thread {
    private int count = 0;
    private boolean suspended = color="#0000ff">false;
    public Suspendable() { start(); }
    public void fauxSuspend() {
      suspended = true;
    }
    public synchronized void fauxResume() {
      suspended = false;
      notify();
    }
    public void run() {
      while (true) {
        try {
          sleep(100);
          synchronized(this) {
            while(suspended)
              wait();
```

```
          }
        } catch(InterruptedException e) {
          System.err.println("Interrupted");
        }
        t.setText(Integer.toString(count++));
      }
    }
  }
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(t);
    suspend.addActionListener(
      new ActionListener() {
        public
        void actionPerformed(ActionEvent e) {
          ss.fauxSuspend();
        }
      });
    cp.add(suspend);
    resume.addActionListener(
      new ActionListener() {
        public
        void actionPerformed(ActionEvent e) {
          ss.fauxResume();
        }
      });
    cp.add(resume);
  }
  public static void main(String[] args) {
    Console.run(new Suspend(), 300, 100);
  }
} ///:~
```

Le drapeau **suspended** dans **Suspendable** est utilisé pour activer ou désactiver la suspension. Pour suspendre, le drapeau est placé à **true** en appelant **fauxSuspend()** et ceci est détecté dans **run()**. Le **wait()**, comme décrit plus haut dans ce chapitre, doit être **synchronized** afin qu'il ait le verrou de l'objet. Dans **fauxResume()**, le drapeau **suspended** est placé à **false** et **notify()** est appelé — puisque ceci reveille **wait()** dans une clause **synchronized** la méthode **fauxResume()** doit aussi être **synchronized** afin qu'elle acquiert le verrou avant d'appeler **notify()** (ainsi le verrou est libre pour que le **wait()** se réveille avec). Si vous suivez le style montré dans ce programme vous pouvez éviter d'utiliser **suspend()** et **resume()**.

La méthode **destroy()** de **Thread** n'a jamais été implémenter; c'est comme un **suspend()** qui ne peut pas être réactivé, donc elle a les mêmes problèmes d'inter-blocage que **suspend()**. Toutefois, ce n'est pas une méthode dépréciée et elle devrait être implémentée dans une future version de Java (après la 2) pour des situations spéciales où le risque d'inter-blocage est acceptable.

Vous devez vous demander pourquoi ces méthodes, maintenant dépréciées, étaient incluses

dans Java dans un premier temps. Il semblerait admissible qu'une erreur significative soit simplement supprimée (et donne encore un autre coup aux arguments pour l'exceptionnel conception et l'infaillibilité claironné par les commerciaux de Sun). La partie réconfortante à propos des changements est que cela indique clairement que ce sont les techniciens et non les commerciaux qui dirige le show — ils découvrent un problème et ils le fixent. Je trouve cela beaucoup plus promettant et encourageant que de laisser le problème parce que « fixer le problème serait admettre une erreur. » Cela signifie que Java continuera à évoluer, même si cela signifie une petite perte de confort pour les programmeurs Java. Je préfère accepter cet inconvénient plutôt que de voir le langage stagné.

La *priorité* d'un thread dit à l'ordonnanceur *[scheduler]* l'importance de ce thread. Si il y a un certain nombre de threads bloqués et en attente d'exécution, l'ordonnanceur exécutera celui avec la plus haute priorité en premier. Cependant, cela ne signifie pas que les threads avec des priorités plus faibles ne tourneront pas (en fait, vous ne pouvez pas avoir d'interblocage à cause des priorités). Les threads de priorités plus faibles ont juste tendance à tourner moins souvent.

Bien qu'il soit intéressant de connaître et de jouer avec les priorités, en pratique vous n'avez pratiquement jamais besoin de gérer les priorités par vous même. Donc soyez libre de passer le reste de cette session si les priorités ne vous intéressent pas.

Lire et changer les priorités

Vous pouvez lire la priorité d'un thread avec **getPriority()** et la changer avec **setPriority()**. La forme des précédents exemples « counter » peut être utiliser pour montrer l'effet des changements de priorités. Dans cette applet vous verrez que les compteurs ralentissent quand les threads associés ont leurs priorités diminuées:

```
//: c14:Counter5.java
// Ajuster les priorités des threads.
// <applet code=Counter5 width=450 height=600>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

class Ticker2 extends Thread {
  private JButton
    b = new JButton("Toggle"),
    incPriority = new JButton("up"),
    decPriority = new JButton("down");
  private JTextField
    t = new JTextField(10),
    pr = new JTextField(3); // Affiche la priorité
  private int count = 0;
  private boolean runFlag = true;
  public Ticker2(Container c) {
    b.addActionListener(new ToggleL());
    incPriority.addActionListener(new UpL());
    decPriority.addActionListener(new DownL());
```

```
    JPanel p = new JPanel();
    p.add(t);
    p.add(pr);
    p.add(b);
    p.add(incPriority);
    p.add(decPriority);
    c.add(p);
  }
  class ToggleL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      runFlag = !runFlag;
    }
  }
  class UpL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      int newPriority = getPriority() + 1;
      if(newPriority > Thread.MAX_PRIORITY)
        newPriority = Thread.MAX_PRIORITY;
      setPriority(newPriority);
    }
  }
  class DownL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      int newPriority = getPriority() - 1;
      if(newPriority < Thread.MIN_PRIORITY)
        newPriority = Thread.MIN_PRIORITY;
      setPriority(newPriority);
    }
  }
  public void run() {
    while (true) {
      if(runFlag) {
        t.setText(Integer.toString(count++));
        pr.setText(
          Integer.toString(getPriority()));
      }
      yield();
    }
  }
}

public class Counter5 extends JApplet {
  private JButton
    start = new JButton("Start"),
    upMax = new JButton("#004488">"Inc Max Priority"),
    downMax = new JButton("#004488">"Dec Max Priority");
  private boolean started = false;
```

```
  private static final int SIZE = 10;
  private Ticker2[] s = new Ticker2[SIZE];
  private JTextField mp = new JTextField(3);
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    for(int i = 0; i < s.length; i++)
      s[i] = new Ticker2(cp);
    cp.add(new JLabel(
      "MAX_PRIORITY = " + Thread.MAX_PRIORITY));
    cp.add(new JLabel("MIN_PRIORITY = "
      + Thread.MIN_PRIORITY));
    cp.add(new JLabel("#004488">"Group Max Priority = "));
    cp.add(mp);
    cp.add(start);
    cp.add(upMax);
    cp.add(downMax);
    start.addActionListener(new StartL());
    upMax.addActionListener(new UpMaxL());
    downMax.addActionListener(new DownMaxL());
    showMaxPriority();
    // Affiche récursivement les groupes de thread parents:
    ThreadGroup parent =
      s[0].getThreadGroup().getParent();
    while(parent != null) {
      cp.add(new Label(
        "Parent threadgroup max priority = "
        + parent.getMaxPriority()));
      parent = parent.getParent();
    }
  }
  public void showMaxPriority() {
    mp.setText(Integer.toString(
      s[0].getThreadGroup().getMaxPriority()));
  }
  class StartL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      if(!started) {
        started = true;
        for(int i = 0; i < s.length; i++)
          s[i].start();
      }
    }
  }
  class UpMaxL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      int maxp =
```

```
          s[0].getThreadGroup().getMaxPriority();
        if(++maxp > Thread.MAX_PRIORITY)
          maxp = Thread.MAX_PRIORITY;
        s[0].getThreadGroup().setMaxPriority(maxp);
        showMaxPriority();
      }
    }
    class DownMaxL implements ActionListener {
      public void actionPerformed(ActionEvent e) {
        int maxp =
          s[0].getThreadGroup().getMaxPriority();
        if(--maxp < Thread.MIN_PRIORITY)
          maxp = Thread.MIN_PRIORITY;
        s[0].getThreadGroup().setMaxPriority(maxp);
        showMaxPriority();
      }
    }
    public static void main(String[] args) {
      Console.run(new Counter5(), 450, 600);
    }
  } ///:~
```

Ticker2 suit la forme établie plus tôt dans ce chapitre, mais il y a un **JTextField** supplémentaire pour afficher la priorité du thread et deux bouttons de plus pour incrémenter et décrémenter la priorité.

Vous pouvez également noter l'utilisation de **yield()**, qui rend volontairement la main à l'ordonnanceur. Sans cela le mécanisme de multithreading fonctionne encore, mais vous remarquerez qu'il tourne plus lentement (essayez de supprimer l'appel à **yield()** pour le voir). Vous pouvez aussi appeler **sleep()**, mais alors la vitesse de comptage sera controller par la durée du **sleep()** au lieu de la priorité.

Le **init()** dans **Counter5** crée un tableau de dix **Ticker2**s; leurs bouttons et champs sont placés sur le formulaire par le constructeur de **Ticker2**. **Counter5** ajoute des bouttons pour tout démarrer aussi bien que pour incrémenter et décrémenter la priorité maximum du groupe de thread. En plus, il y a des labels qui affichent les priorités maximum et minimum possibles pour un thread et un **JTextField** pour montrer la priorité maximum du groupe de thread. (La prochaine section décrira les groupes de threads.) Finalement, les priorités des groupes de threads parents sont aussi afficher comme labels.

Quand vous pressez un bouton « up » ou « down », la priorité du **Tricker2** est rapportée et incrémenter ou décrémenter en conséquence.

Quand vous exécutez ce programme, vous noterez plusieurs choses. Tout d'abord, la priorité du groupe de thread est par défaut cinq. Même si vous décrémentez la priorité maximum en dessous de cinq avant de démarrer les threads (ou avant de les créer, ce qui nécessite un changement du code), chaque thread aura une priorité par défaut de cinq.

Le test simple est de prendre un compteur et de décrémenter sa priorité jusqu'à un, et observez qu'il compte beaucoup plus lentement. Mais maintenant essayez de l'incrémenter de nouveau. Vous pouvez le ramené à la priorité du groupe de thread, mais pas plus haut. Maintenant décrémen-

tez la priorité du groupe de threads. Les priorités des threads restent inchangées, mais si vous essayez de les modifier dans un sens ou dans l'autre vous verrez qu'elles sauteront automatiquement à la priorité du groupe de thread. Les nouvelles threads auront également une priorité par défaut qui peut être plus haute que la priorité du groupe. (Ainsi la priorité du groupe n'est pas un moyen d'empêcher les nouveaux threads d'avoir des priorités plus hautes que celles existantes.)

Finalement, essayez d'incrémenter la priorité maximum du groupe. On ne peut pas le faire. Vous pouvez seulement réduire la priorité maximum d'un groupe de thread, pas l'augmenter.

Les groupes de threads

Tous les threads appartiennent à un groupe de thread. Ce peut être soit le groupe de thread par défaut, soit un groupe de thread que vous spécifiez explicitement quand vous créez le thread. A la création, le thread est attaché à un groupe et ne peut pas en changer. Chaque application a au moins un thread qui appartient au groupe de thread système. Si vous créez plus de threads sans spécifier de groupe, ils appartiendront aussi au groupe de thread système.

Les groupes de threads doivent aussi appartenir à d'autres groupes de threads. Le groupe de thread auquel un nouveau appartient doit être spécifié dans le constructeur. Si vous créez un groupe de thread sans spécifier de groupe auquel le rattacher, il sera placer dans le groupe de thread système. Ainsi, tous les groupes de threads de votre application auront en fin de compte le groupe de thread système comme parent.

La raison de l'existence des groupes de threads est difficile à déterminer à partir de la littérature, qui tend à être confuse sur le sujet. Il est souvent cité des raisons de sécurités. D'après Arnold & Gosling,« Les threads d'un groupe peuvent modifier les autres threads du groupe, y compris ceux situés plus bas dans la hiérarchie. Un thread ne peut pas modifier les threads extérieurs à son propre groupe ou les groupes qu'il contient. » Il est difficile de savoir ce que « modifier » est supposé signifier ici. L'exemple suivant montre un thread dans un sous-groupe « feuille » modifier les priorités de tous les threads de son arbre de groupe de thread aussi bien qu'appeler une méthode pour tous les threads de son arbre.

```
//: c14:TestAccess.java
// Comment les threads peuvent accéder aux
// autres threads dans un groupe de threads parent.

public class TestAccess {
  public static void main(String[] args) {
    ThreadGroup
      x = new ThreadGroup("x"),
      y = new ThreadGroup(x, "y"),
      z = new ThreadGroup(y, "z");
    Thread
      one = new TestThread1(x, "one"),
      two = new TestThread2(z, "two");
  }
}

class TestThread1 extends Thread {
  private int i;
```

```
  TestThread1(ThreadGroup g, String name) {
    super(g, name);
  }
  void f() {
    i++; // modifie ce thread
    System.out.println(getName() + " f()");
  }
}

class TestThread2 extends TestThread1 {
  TestThread2(ThreadGroup g, String name) {
    super(g, name);
    start();
  }
  public void run() {
    ThreadGroup g =
      getThreadGroup().getParent().getParent();
    g.list();
    Thread[] gAll = new Thread[g.activeCount()];
    g.enumerate(gAll);
    for(int i = 0; i < gAll.length; i++) {
      gAll[i].setPriority(Thread.MIN_PRIORITY);
      ((TestThread1)gAll[i]).f();
    }
    g.list();
  }
} ///:~
```

Dans **main()**, plusieurs **ThreadGroup**s sont créés, placés en feuilles des autres: **x** n'a pas d'autres arguments que son nom (une **String**), ainsi il est automatiquement placé dans le groupe de threads « système », tandis que **y** est sous **x** et **z** est sous **y**. Notez que l'initialisation se passe dans l'ordre textuel donc ce code est légal.

Deux threads sont créés et placés dans différents groupes de threads. **TestThread1** n'a pas de méthode **run()** mais a une méthode **f()** qui modifie le thread et écrit quelque chose afin que vous puissiez voir qu'elle a été appelée. **TestThread2** est une sous classe de **TestThread1** et sa méthode **run()** est assez élaboré. Elle récupère d'abord le groupe de thread du thread courant, puis remonte deux niveaux dans l'arbre d'héritage en utilisant **getParent()**. (Ceci est étudié puisque j'ai intentionnellement placer l'objet **TestThread2** deux niveaux plus bas dans la hiérarchie.) A ce point, un tableau de références sur **Thread** est créé en utilisant la méthode **activeCount()** pour demander combien de threads sont dans ce groupe de thread et tous les groupes fils. La méthode **enumerate()** place les références à toutes ces threads dans le tableau **gAll**, ensuite je me parcourt simplement dans la totalité du tableau en appelant la méthode **f()** pour chaque thread, ainsi qu'en modifiant la priorité. Ainsi, un thread dans un groupe de thread « feuille » modifie les threads dans les groupes de threads parents.

La méthode de déboguage **list()** affiche toutes l'information sur un groupe de thread sur la sortie standard ce qui aide beaucoup lorsqu'on examine le comportement d'un groupe de threads. Voici la sortie du programme:

```
java.lang.ThreadGroup[name=x,maxpri=10]
   Thread[one,5,x]
   java.lang.ThreadGroup[name=y,maxpri=10]
      java.lang.ThreadGroup[name=z,maxpri=10]
         Thread[two,5,z]
one f()
two f()
java.lang.ThreadGroup[name=x,maxpri=10]
   Thread[one,1,x]
   java.lang.ThreadGroup[name=y,maxpri=10]
      java.lang.ThreadGroup[name=z,maxpri=10]
         Thread[two,1,z]
```

Non seulement **list()** affiche le nom du **ThreadGroup** ou du **Thread**, mais elle affiche aussi le nom du groupe de thread et sa priorité maximum. Pour les threads, le nom de thread est également affiché, suivi par la priorité du thread et le groupe auquel il appartient. Notez que **list()** indente les threads et groupes de threads pour indiquer qu'ils sont enfant du groupe de threads non indenté.

Vous pouvez voir que **f()** est appelé par la méthode **run()** de **TestThread2**, il est donc évident que tous les threads d'un groupe sont vulnérables. Cependant, vous ne pouvez accéder seulement aux threads embranchés sur votre propre arbre du groupe de thread **système**, et peut-être que c'est ce que signifie « sûr. » Vous ne pouvez pas accéder à l'arbre du groupe de thread système d'un autre.

Controller les groupes de threads

En dehors de l'aspect sécurité, une chose pour lesquelles les groupes de threads semble être utiles est le controle: vous pouvez effectuer certaines opérations sur un groupe de thread entier avec une seule commande. L'exemple suivant le démontre, et les restrictions sur les priorités avec les groupes de threads. Les numéros entre parenthèses donne une référence pour comparer la sortie.

```
//: c14:ThreadGroup1.java
// Comment les groupes de threads contrôlent les priorités
// des threads qui les composent.

public class ThreadGroup1 {
  public static void main(String[] args) {
    // Récupère le thread system & imprime ses Info:
    ThreadGroup sys =  Thread.currentThread().getThreadGroup();
    sys.list(); // (1)
    // Réduit la priorité du groupe de thread système:
    sys.setMaxPriority(Thread.MAX_PRIORITY - 1);
    // Augmente la priorité du thread principal:
    Thread curr = Thread.currentThread();
    curr.setPriority(curr.getPriority() + 1);
    sys.list(); // (2)
    // Essaie de créer un nouveau groupe avec une priorité maximum:
    ThreadGroup g1 = new ThreadGroup("#004488">"g1");
    g1.setMaxPriority(Thread.MAX_PRIORITY);
```

```
// Essaie de créer un nouveau thread avec une priorité maximum:
Thread t = new Thread(g1, "A");
t.setPriority(Thread.MAX_PRIORITY);
g1.list(); // (3)
// Reduit la priorité maximum de g1, puis essaie
// de l'augmenter:
g1.setMaxPriority(Thread.MAX_PRIORITY - 2);
g1.setMaxPriority(Thread.MAX_PRIORITY);
g1.list(); // (4)
// Essaie de créer un nouveau thread avec une priorité maximum:
t = new Thread(g1, "B");
t.setPriority(Thread.MAX_PRIORITY);
g1.list(); // (5)
// Diminue la priorité maximum en dessous
// de la priorité par défaut du thread:
g1.setMaxPriority(Thread.MIN_PRIORITY + 2);
// Regarde la priorité d'un nouveau thread
// avant et après son changement:
t = new Thread(g1, "C");
g1.list(); // (6)
t.setPriority(t.getPriority() -1);
g1.list(); // (7)
// Fait de g2 un groupe de thread fils de g1 et
// essaie d'augmenter sa priorité:
ThreadGroup g2 = new ThreadGroup(g1, "#004488">"g2");
g2.list(); // (8)
g2.setMaxPriority(Thread.MAX_PRIORITY);
g2.list(); // (9)
// Ajoute un banc de nouveaux threads à g2:
for (int i = 0; i < 5; i++)
  new Thread(g2, Integer.toString(i));
// Montre des informations sur tous les groupes de threads
// et threads:
sys.list(); // (10)
System.out.println("Starting all threads:");
Thread[] all = new Thread[sys.activeCount()];
sys.enumerate(all);
for(int i = 0; i < all.length; i++)
  if(!all[i].isAlive())
    all[i].start();
// Suspends & Arrête tous les threads de
// ce groupe et de ses sous-groupes:
System.out.println("All threads started");
sys.suspend(); // Deprecié en Java 2
// On n'arrive jamais ici...
System.out.println("All threads suspended");
sys.stop(); //  Deprecié en Java 2
```

```
    System.out.println("All threads stopped");
  }
} ///:~
```

La sortie qui suit a été édité pour lui permettre de tenir sur la page (le **java.lang** a été supprimé) et pour ajouter des numéros correspondant à ceux des commentaires du listing précédent.

```
(1) ThreadGroup[name=system,maxpri=10]
        Thread[main,5,system]
(2)  ThreadGroup[name=system,maxpri=9]
        Thread[main,6,system]
(3) ThreadGroup[name=g1,maxpri=9]
        Thread[A,9,g1]
(4) ThreadGroup[name=g1,maxpri=8]
        Thread[A,9,g1]
(5) ThreadGroup[name=g1,maxpri=8]
        Thread[A,9,g1]
        Thread[B,8,g1]
(6) ThreadGroup[name=g1,maxpri=3]
        Thread[A,9,g1]
        Thread[B,8,g1]
        Thread[C,6,g1]
(7) ThreadGroup[name=g1,maxpri=3]
        Thread[A,9,g1]
        Thread[B,8,g1]
        Thread[C,3,g1]
(8) ThreadGroup[name=g2,maxpri=3]
(9) ThreadGroup[name=g2,maxpri=3]
(10)ThreadGroup[name=system,maxpri=9]
        Thread[main,6,system]
        ThreadGroup[name=g1,maxpri=3]
          Thread[A,9,g1]
          Thread[B,8,g1]
          Thread[C,3,g1]
          ThreadGroup[name=g2,maxpri=3]
            Thread[0,6,g2]
            Thread[1,6,g2]
            Thread[2,6,g2]
            Thread[3,6,g2]
            Thread[4,6,g2]
    Starting all threads:
    All threads started
```

Tous les programmes ont au moins un thread qui tourne, et la première action de **main()** est

d'appeler la méthode **static** de **Thread** nommée **currentThread()**. Depuis ce thread, le groupe de thread est produit et **list()** est appelé sur le résultat. La sortie est:

```
(1) ThreadGroup[name=system,maxpri=10]
        Thread[main,5,system]
```

Vous pouvez voir que le nom du groupe de thread principal est **system**, et le nom du thread principal est **main**, et il appartient au groupe de thread **system**.

Le second exercice montre que la priorité maximum du groupe **system** peut être réduite et le thread **main** peut avoir sa priorité augmentée.

```
(2) ThreadGroup[name=system,maxpri=9]
        Thread[main,6,system]
```

Le troisième exercice crée un nouveau groupe de thread, **g1**, qui appartient automatiquement au groupe de thread **system** puisqu'il n'est rien spécifié d'autre. Un nouveau thread **A** est placé dans **g1**. Après avoir essayer de positionner la priorité maximum de ce groupe au plus haut niveau possible et la priorité de **A** au niveau le plus élevé, le résultat est:

```
(3) ThreadGroup[name=g1,maxpri=9]
        Thread[A,9,g1]
```

Ainsi, il n'est pas possible de changer la priorité maximum d'un groupe de thread au delà de celle de son groupe de thread parent.

Le quatrième exercice réduit la priorité maximum de **g1** de deux et essaie ensuite de l'augmenter jusqu'à **Thread.MAX_PRIORITY**. Le résultat est:

```
(4) ThreadGroup[name=g1,maxpri=8]
        Thread[A,9,g1]
```

Vous pouvez voir que l'augmentation à la priorité maximum ne fonctionne pas. Vous pouvez seulement diminuer la priorité maximum d'un groupe de thread, pas l'augmenter. Notez également que la priorité du thread **A** n'a pas changé, et est maintenant plus grande que la priorité maximum du groupe de thread. Changer la priorité maximum d'un groupe de thread n'affecte pas les threads existantes.

Le cinquième exercice essaie de créer un nouveau thread avec une priorité au maximum:

```
(5) ThreadGroup[name=g1,maxpri=8]
        Thread[A,9,g1]
        Thread[B,8,g1]
```

Le nouveau thread ne peut pas être changer à une priorité plus haute que la priorité maximum du groupe de thread.

La priorité par défaut du thread pour ce programme est six; c'est la priorité avec laquelle un nouveau thread sera créé et à laquelle il restera si vous ne manipulez pas la priorité. L'exercice 6 diminue la priorité maximum du groupe de thread en dessous de la priorité par défaut pour voir ce qui se passe quand vous créez un nouveau thread dans ces conditions:

```
(6) ThreadGroup[name=g1,maxpri=3]
```

```
        Thread[A,9,g1]
        Thread[B,8,g1]
        Thread[C,6,g1]
```

Étant donné que la priorité maximum du groupe de thread est trois, le nouveau thread est encore créer en utilisant la priorité par défaut de six. Ainsi, la priorité maximum du groupe de thread n'affecte pas la priorité par défaut. (En fait, il ne semble pas y avoir de moyen pour positionner la priorité par défaut des nouveaux threads.)

Après avoir changer la priorité, en essayant de la décrémenter par pas de un, le résultat est:

```
(7) ThreadGroup[name=g1,maxpri=3]
        Thread[A,9,g1]
        Thread[B,8,g1]
        Thread[C,3,g1]
```

La priorité maximum du groupe de thread est forcé seulement lorsque vous essayez de changer la priorité du thread.

Une expérience similaire est effectué en (8) et (9), dans lequel un nouveau groupe de thread **g2** est créé comme un fils de **g1** et sa priorité maximum est changée. Vous pouvez voir qu'il n'est pas impossible pour la priorité maximum de **g2** de devenir plus grande que celle de **g1**:

```
(8) ThreadGroup[name=g2,maxpri=3]
(9)  ThreadGroup[name=g2,maxpri=3]
```

Notez également que **g2** est automatiquement mise à la priorité maximum du groupe de thread **g1** dès la création de **g2**.

Après toutes ces expériences, le système de groupes de threads est entièrement donné:

```
(10)ThreadGroup[name=system,maxpri=9]
        Thread[main,6,system]
        ThreadGroup[name=g1,maxpri=3]
          Thread[A,9,g1]
          Thread[B,8,g1]
          Thread[C,3,g1]
          ThreadGroup[name=g2,maxpri=3]
            Thread[0,6,g2]
            Thread[1,6,g2]
            Thread[2,6,g2]
            Thread[3,6,g2]
            Thread[4,6,g2]
```

Donc à cause des règles sur les groupes de threads, un groupe fils doit toujours avoir une priorité maximum plus petite ou égale à celle de son parent.

La dernière partie de ce programme démontre les méthodes pour un groupe de threads entier. Dans un premier temps le programme parcourt l'intégralité de l'arbre de threads et démarre chaque thread qui ne l'est pas déjà. Par drame, le groupe **system** est alors suspendu et finalement arrêter. (Bien qu'il soit intéressant de voir que **suspend()** et **stop()** fonctionnent sur un groupe de thread en-

tier, vous devez garder en tête que ces méthodes sont dépréciées en Java 2.) Mais quand vous suspendez le groupe **system** vous suspendez aussi le thread **main** et le programme entier s'arrête, donc il ne quittera jamais le point où les threads sont stoppés. Actuellement, si vous stoppez le thread **main** il déclenche un exception **ThreadDeath**, ce n'est donc pas une chose à faire. Puisque **ThreadGroup** hérite de **Object**, qui contient la méthode **wait()**, vous pouvez aussi choisir de suspendre le programme pour un certain nombre de secondes en appelant **wait(secondes * 1000)**. Ce qui doit acquérir le verrou dans un bloc synchronised, bien sûr.

La classe **ThreadGroup** a aussi des méthodes **suspend()** et **resume()** donc vous pouvez arrêter et démarrer un groupe de thread entier et toutes ces threads et sous-groupes avec une seule commande. (Encore un fois, **suspend()** et **resume()** sont déprécié en Java 2.)

Les groupes de threads peuvent paraître un peu mystérieux au premier abord, mais garder en mémoire que vous ne les utiliserez probablement directement très peu souvent.

Runnable revisité

Précédemment dans ce chapitre, j'ai suggéré que vous deviez faire très attention avant de faire d'une applet ou une **Frame** principale une implémentation de **Runnable**. Bien sûr, si vous devez héritez d'une classe *et* que vous voulez ajouter un comportement multitâche à la classe, **Runnable** est la solution correcte. L'exemple final de ce chapitre exploite ceci en créant une classe **Runnable JPanel** qui se peint de différentes couleurs. Cette application est prend des valeurs depuis la ligne de commande pour déterminer la taille de la grille de couleurs et la durée du **sleep()** entre les changements de couleurs. En jouant sur ces valeurs vous découvrirez des possibilités intéressantes et parfois inexplicable des threads:

```
//: c14:ColorBoxes.java
// Utilisation de l'interface Runnable.
// <applet code=ColorBoxes width=500 height=400>
// <param name=grid value="12">
// <param name=pause value="50">
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

class CBox extends JPanel implements Runnable {
  private Thread t;
  private int pause;
  private static final Color[] colors = {
    Color.black, Color.blue, Color.cyan,
    Color.darkGray, Color.gray, Color.green,
    Color.lightGray, Color.magenta,
    Color.orange, Color.pink, Color.red,
    Color.white, Color.yellow
  };
  private Color cColor = newColor();
  private static final Color newColor() {
```

```
    return colors[
      (int)(Math.random() * colors.length)
    ];
  }
  public void paintComponent(Graphics  g) {
    super.paintComponent(g);
    g.setColor(cColor);
    Dimension s = getSize();
    g.fillRect(0, 0, s.width, s.height);
  }
  public CBox(int pause) {
    this.pause = pause;
    t = new Thread(this);
    t.start();
  }
  public void run() {
    while(true) {
      cColor = newColor();
      repaint();
      try {
        t.sleep(pause);
      } catch(InterruptedException e) {
        System.err.println("Interrupted");
      }
    }
  }
}

public class ColorBoxes extends JApplet {
  private boolean isApplet = true;
  private int grid = 12;
  private int pause = 50;
  public void init() {
    // Récupère les paramètres depuis la page web:
    if (isApplet) {
      String gsize = getParameter("grid");
      if(gsize != null)
        grid = Integer.parseInt(gsize);
      String pse = getParameter("pause");
      if(pse != null)
        pause = Integer.parseInt(pse);
    }
    Container cp = getContentPane();
    cp.setLayout(new GridLayout(grid, grid));
    for (int i = 0; i < grid * grid; i++)
      cp.add(new CBox(pause));
  }
```

```
  public static void main(String[] args) {
    ColorBoxes applet = new ColorBoxes();
    applet.isApplet = false;
    if(args.length > 0)
      applet.grid = Integer.parseInt(args[0]);
    if(args.length > 1)
      applet.pause = Integer.parseInt(args[1]);
    Console.run(applet, 500, 400);
  }
} ///:~
```

ColorBoxes est l'applet/application habituelle avec une méthode **init()** qui créé la GUI. Elle positionne la **GridLayout** afin d'avoir une grille de cellules dans chaque dimension. Ensuite elle ajoute le nombre approprié d'objet **CBox** pour remplir la grille, passant la valeur **pause** a chacune. Dans **main()** vous pouvez voir comment **pause** et **grid** ont des valeurs par défaut qui peuvent être changées si vous passez des arguments à la ligne de commande, ou en utilisant des arguments de l'applet.

CBox est là où tout le travail s'effectue. Elle hérite de **JPanel** et implémente l'interface **Runnable** ainsi chaque **JPanel** peut aussi être un **Thread**. Souvenez vous que quand vous implémentez **Runnable**, vous ne faites pas un objet **Thread**, mais juste une classe qui a une méthode **run()**. Ainsi, vous devez créer explicitement un objet **Thread** et passer l'objet **Runnable** au constructeur, puis appelez **start()** (ce qui est fait dans le constructeur). Dans **CBox** ce thread est appelé **t**.

Remarquez le tableau **colors**, qui est une énumération de toutes les couleurs de la classe **Color**. Il est utilisé dans **newColor()** pour produire une couleur sélectionnée au hasard. La couleur de la cellule courante est **cColor**.

paintComponent() est assez simple — elle place juste la couleur à **cColor** et remplit intégralement le **JPanel** avec cette couleur.

Dans **run()**, vous voyez la boucle infinie qui place la **cColor** à une nouvelle couleur prise au hasard et ensuite appelle **repaint()** pour la montrer. Puis le thread passe dans **sleep()** pour le temps spécifié sur la ligne de commande.

C'est précisément parce que le design est flexible et que le threading est attaché à chaque élément **JPanel**, que vous pouvez expérimenter en créant autant de threads que vous voulez. (En réalité, il y a une restriction imposé par le nombre de threads que votre JVM peut confortablement gérés.)

Ce programme fait aussi un benchmark intéressant, puisqu'il peut montrer des différences de performance dramatiques entre deux implémentation du threading dans les JVM.

Trop de threads

A un certain point, vous trouverez que **ColorBoxes** ralentit. Sur ma machine, cela se passe quelque part après une grille 10 x 10. Pourquoi est ce que cela arrive? Vous soupçonner naturellement que Swing ai quelques chose à voir avec ça, donc voici un exemple qui test ce prémisse en faisant moins de threads. Le code suivant est réorganisé afin qu'une **ArrayList implements Runnable** et cette **ArrayList** gère un nombre de blocs de couleurs et en choisit une au hasard pour la mise à jour. Puis un certain nombre de ces objets **ArrayList** sont créés, en fonction d'une approximation de

la dimension de la grille que vous choisissez. Au résultat, vous avez beaucoup moins de threads que de blocs de couleurs, donc si il y a une accélération nous saurons que c'était à cause du trop grand nombre de threads dans l'exemple précédent:

```
//: c14:ColorBoxes2.java
// Compromis dans l'utilisation de thread.
// <applet code=ColorBoxes2 width=600 height=500>
// <param name=grid value="12">
// <param name=pause value="50">
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import com.bruceeckel.swing.*;

class CBox2 extends JPanel {
  private static final Color[] colors = {
    Color.black, Color.blue, Color.cyan,
    Color.darkGray, Color.gray, Color.green,
    Color.lightGray, Color.magenta,
    Color.orange, Color.pink, Color.red,
    Color.white, Color.yellow
  };
  private Color cColor = newColor();
  private static final Color newColor() {
    return colors[
      (int)(Math.random() * colors.length)
    ];
  }
  void nextColor() {
    cColor = newColor();
    repaint();
  }
  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.setColor(cColor);
    Dimension s = getSize();
    g.fillRect(0, 0, s.width, s.height);
  }
}

class CBoxList
  extends ArrayList implements Runnable {
  private Thread t;
  private int pause;
  public CBoxList(int pause) {
    this.pause = pause;
```

```
    t = new Thread(this);
  }
  public void go() { t.start(); }
  public void run() {
    while(true) {
      int i = (int)(Math.random() * size());
      ((CBox2)get(i)).nextColor();
      try {
        t.sleep(pause);
      } catch(InterruptedException e) {
        System.err.println("Interrupted");
      }
    }
  }
  public Object last() { return get(size() - 1);}
}

public class ColorBoxes2 extends JApplet {
  private boolean isApplet = true;
  private int grid = 12;
  // Pause par défaut plus courte que dans ColorBoxes:
  private int pause = 50;
  private CBoxList[] v;
  public void init() {
    // Récupère les paramètres de la page Web:
    if (isApplet) {
      String gsize = getParameter("grid");
      if(gsize != null)
        grid = Integer.parseInt(gsize);
      String pse = getParameter("pause");
      if(pse != null)
        pause = Integer.parseInt(pse);
    }
    Container cp = getContentPane();
    cp.setLayout(new GridLayout(grid, grid));
    v = new CBoxList[grid];
    for(int i = 0; i < grid; i++)
      v[i] = new CBoxList(pause);
    for (int i = 0; i < grid * grid; i++) {
      v[i % grid].add(new CBox2());
      cp.add((CBox2)v[i % grid].last());
    }
    for(int i = 0; i < grid; i++)
      v[i].go();
  }
  public static void main(String[] args) {
    ColorBoxes2 applet = new ColorBoxes2();
```

```
    applet.isApplet = false;
    if(args.length > 0)
      applet.grid = Integer.parseInt(args[0]);
    if(args.length > 1)
      applet.pause = Integer.parseInt(args[1]);
    Console.run(applet, 500, 400);
  }
} ///:~
```

CBox2 est similaire à **CBox**: elle se paint avec une couleur choisie au hasard. Mais c'est tout ce qu'une **CBox2** fait. Tout le threading a été déplacé dans **CBoxList**.

Le **CBoxList** pourrait aussi avoir hérité de **Thread** et avoir un objet membre de type **ArrayList**. Ce design a l'avantage que les méthodes **add()** et **get()** puissent avoir des arguments et un type de valeur de retour spécifiques au lieu de ceux de la classe générique **Object**. (Leurs noms pourraient aussi être changés en quelque chose de plus court.) Cependant, le design utilisé ici semble au premier coup d'œil nécessiter moins de code. En plus, on garde automatiquement tous les autres comportements d'un **ArrayList**. Avec tous les cast et parenthèses nécessaires pour **get()**, ça ne devrait pas être le cas dès que votre code central augmente.

Comme précédemment, quand vous implémentez **Runnable** vous ne disposez pas de tout l'équipement que vous avez avec **Thread**, donc vous devez créer un nouveau **Thread** et passer vous même à son constructeur pour avoir quelque chose à démarrer (par **start()**), comme vous pouvez le voir dans le constructeur de **CBoxList** et dans **go()**. La méthode **run()** choisit simplement un numéro d'élément au hasard dans la liste et appelle **nextColor()** pour cet élément ce qui provoque le sélection au hasard d'une nouvelle couleur.

En exécutant ce programme, vous voyez qu'effectivement il tourne et répond plus vite (par exemple, quand vous l'interrompez, il s'arrête plus rapidement), et il ne semble pas ralentir autant pour de grandes tailles de grilles. Ainsi, un nouveau facteur est ajouter à l'équation du threading: vous devez regarder pour voir si vous n'avez pas « trop de threads » (quelqu'en soit la signification pour votre programme et plate-forme en particulier, le ralentissement dans **ColorBoxes** apparaît comme provenant du fait qu'il n'y a qu'un thread qui répond pour toutes les colorations, et qu'il est ralenti par trop de requêtes). Si vous avez trop de threads, vous devez essayez d'utiliser des techniques comme celle ci-dessus pour « équilibrer » le nombre de threads dans votre programme. Si vous voyez des problèmes de performances dans un programme multithread vous avez maintenant plusieurs solutions à examiner:

1. Avez vous assez d'appels à **sleep()**, **yield(),** et/ou **wait()**?
2. Les appels à **sleep()** sont-ils assez longs?
3. Faites vous tournez trop de threads?
4. Avez vous essayé différentes plates-formes et JVMs?

Des questions comme celles-là sont la raison pour laquelle la programmation multithread est souvent considéré comme un art.

Résumé

Il est vital d'apprendre quand utiliser le multithreading et quand l'éviter. La principale raison

pour l'utiliser est pour gérer un nombre de taches qui mélangées rendront plus efficace l'utilisation de l'ordinateur (y compris la possibilité de distribuer de façon transparente les tâches sur plusieurs CPUs) ou être plus pratique pour l'utilisateur. L'exemple classique de répartition de ressources est l'utilisation du CPU pendant les attentes d'entrées/sorties. L'exemple classique du coté pratique pour l'utilisateur est la surveillance d'un bouton « stop » pendant un long téléchargement.

Les principaux inconvénients du multithreading sont:

1. Ralentissement en attente de ressources partagées
2. Du temp CPU supplémentaire nécessaire pour gérer les threads
3. Complexité infructueuse, comme l'idée folle d'avoir un thread séparé pour mettre à jour chaque élément d'un tableau
4. Pathologies incluant starving [=mourir de faim!?], racing, et inter-blocage [deadlock]

Un avantage supplémentaire des threads est qu'ils substituent des contextes d'exécution « léger » (de l'ordre de 100 instructions) aux contextes d'exécutions lourds des processus (de l'ordre de 1000 instructions). Comme tous les threads d'un processus donné partage le même espace mémoire, un contexte léger ne change que l'exécution du programme et les variables locales. D'un autre coté, un changement de processus — le changement de contexte lourd — doit échanger l'intégralité de l'espace mémoire.

Le threading c'est comme marcher dans un monde entièrement nouveau et apprendre un nouveau langage de programmation entier, ou au moins un nouveau jeu de concepts de langage. Avec l'apparition du support des threads dans beaucoup de systèmes d'exploitation de micro-ordinateurs, des extensions pour les threads sont aussi apparues dans les langages de programmations ou librairies. Dans tous les cas, la programmation de thread (1) semble mystérieuse et requiert un changement dans votre façon de penser la programmation; et (2) apparaît comme similaire au support de thread dans les autres langages, donc quand vous comprenez les threads, vous les comprenez dans une langue commune. Et bien que le support des threads puisse faire apparaître Java comme un langage plus compliqué, n'accuser pas Java. Les threads sont difficiles.

Une des plus grande difficultés des threads se produit lorsque plus d'un thread partagent une ressource — comme la mémoire dans un objet — et que vous devez être sûr que plusieurs threads n'essayent pas de lire et changer cette ressource en même temps. Cela nécessite l'utilisation judicieuse du mot clé **synchronized**, qui est un outil bien utile mais qui doit être bien compris parce qu'il peut introduire silencieusement des situations d'inter-blocage.

En plus, il y a un certain art dans la mise en application des threads. Java est conçu pour résoudre vos problèmes — au moins en théorie. (Créer des millions d'objets pour une analyse d'un ensemble fini d'élément dans l'ingénierie, par exemple, devrait être faisable en Java). Toutefois, il semble qu'il y ai une borne haute au nombre de threads que vous voudrez créer, parce que à un certain point un grand nombre de threads semble devenir lours. Ce point critique n'est pas dans les milliers comme il devrait être avec les objets, mais plutôt à quelques centaines, quelquefois moins que 100. Comme vous créez souvent seulement une poignée de threads pour résoudre un problème, c'est typiquement pas vraiment une limite, bien que dans un design plus général cela devient une contrainte.

Une importante conséquence non-évidente du threading est que, en raison de l'ordonnancement des threads, vous pouvez classiquement rendre vos applications plus rapides en insérant des appels à **sleep()** dans la boucle principale de **run()**. Cela fait définitivement ressembler ça à un art, en particulier quand la longueur du délai semble augmenter les performances. Bien sûr, la raison

pour laquelle cela arrive est que des délais plus court peuvent causer la fin de **sleep()** de l'interruption du scheduler avant que le thread tournant soit prêt à passer au sleep, forçant le scheduler à l'arrêter et le redémarrer plus tard afin qu'il puisse finir ce qu'il était en train de faire puis de passer à sleep. Cela nécessite une réflexion supplémentaire pour réaliser quel désordre regne dans tout ça.

Une chose que vous devez notez comme manquant dans ce chapitre est un exemple d'animation, ce qui est une des chose les plus populaires faite avec des applets. Toutefois, une solution complète (avec du son) à ce problème vient est disponible avec le Java JDK (disponible sur *java.sun.com*) dans la section demo. En plus, nous pouvons nous attendre à un meilleur support d'animation comme partie des futures versions de Java, tandis que des solutions très différentes du Java, non programmables, pour les animations apparaissent sur le Web qui seront probablement supérieures aux approches traditionnelles. Pour des explications à propos du fonctionnement des animations Java, voyez *Core Java 2* par Horstmann & Cornell, Prentice-Hall, 1997. Pour des discussions plus avancées sur le threading, voyez *Concurrent Programming in Java* de Doug Lea, Addison-Wesley, 1997, ou *Java Threads* de Oaks & Wong, O’Reilly, 1997.

Exercices

Les solutions des exercices sélectionnés peuvent être trouvées dan le document électronique *The Thinking in Java Annotated Solution Guide*, disponible pour une petite contribution sur *www.BruceEckel.com*.

1. Faites hériter une classe de **Thread** et redéfinissez la méthode **run()**. Dans **run()**, affichez un message, puis appelez **sleep()**. Répétez cela trois fois, puis sortez de **run()**. Placer un message de démarrage dans le constructeur et redéfinissez **finalize()** pour afficher un message d'arrêt. Faites une classe thread séparée qui appelle **System.gc()** et **System.runFinalization()** dans **run()**, affichant également un message. Faites plusieurs objets threads des deux types et exécutez les pour voir ce qui ce passe.

2. Modifiez **Sharing2.java** pour ajouter un bloc **synchronized** dans la méthode **run()** de **TwoCounter** à la place de la synchronisation sur l'intégralité de la méthode **run()**.

3. Créez deux sous-classes de **Thread** une avec un **run()** qui démarre, capture la référence à un second objet **Thread** et appelle alors **wait()**. Le **run()** de l'autre classe devrait appeler **notifyAll()** pour le premier thread après qu'un certain nombre de secondes se soit écoulé, ainsi le premier thread peut afficher un message.

4. Dans **Ticker2** de **Counter5.java**, supprimez le **yield()** et expliquez les résultats. Remplacez le **yield()** avec un **sleep()** et expliquez les résultats.

5. Dans **ThreadGroup1.java**, remplacez l'appel à **sys.suspend()** par un appel à **wait()** pour le bon groupe de thread, entraînant une attente de deux secondes. Pour que ça marche correctement vous devez acquérir le verrou pour **sys** dans un bloc **synchronized**.

6. Changez **Daemons.java** pour que **main()** ait un **sleep()** à la place d'un **readLine()**. Expérimentez avec différents temps pour le sleep pour voir ce qui se passe.

7. Dans le chapitre 8, localisez l'exemple **GreenhouseControls.java**, qui est constitués de trois fichiers. Dans **Event.java**, la classe **Event** est basée sur l'observation du temps. Changez **Event** pour qu'il soit un **Thread**, et changez le reste du design pour qu'il fonctionne avec le nouvel **Event** basé sur **Thread**.

8. Modifiez l'exercice 7 pour que la classe **java.util.Timer** trouvé dans le JDK 1.3 soit

utilisée pour faire tourner le système.

9. En commençant avec **SineWave.java** du Chapitre 13, créez un programme (une applet/application utilisant la classe **Console**) qui dessine un signal sinus animé qui apparaît en défilant sur la fenêtre comme un oscilloscope, dirigeant l'animation avec un **Thread**. La vitesse de l'animation pourra être controller avec un contrôle **java.swing.JSlider**.

10. Modifiez l'exercice 9 afin que de multiple signaux sinus puissent être contrôlés par des tags HTML ou des paramètres de la ligne de commande.

11. Modifiez l'exercice 9 afin que la classe **java.swing.Timer** soit utilisée pour diriger l'animation. Notez la différence entre celle-ci et **java.util.Timer**.

[70] **Runnable** était dans Java 1.0, tandis que les classes internes n'ont été introduites que dans Java 1.1, qui devait partiellement compter pour l'existence de **Runnable**. De plus, les architectures multithread traditionelles se basent sur une fonction à exécuter plutôt qu'un objet. Ma préférence est toujours d'hériter de **Thread** si possible; cela parait plus clair et plus flexible pour moi.

[71] *The Java Programming Language*, de Ken Arnold et James Gosling, Addison-Wesley 1996 pp 179.

Chapitre 15 - Informatique distribuée

De tous temps, la programmation des machines interconnectées s'est avérée difficile, complexe, et sujette à erreurs.

Le programmeur devait connaître le réseau de façon détaillée, et parfois même son hardware. Il était généralement nécessaire d'avoir une bonne compréhension des différentes couches du protocole réseau, et il existait dans chacune des bibliothèques de réseau un tas de fonctions, bien entendu différentes, concernant la connexion, l'empaquetage et le dépaquetage des blocs d'information, l'émission et la réception de ces blocs, la correction des erreurs et le dialogue. C'était décourageant.

Toutefois, l'idée de base de l'informatique distribuée n'est pas vraiment complexe, et elle est encapsulée de manière très agréable dans les bibliothèques Java. Il faut pouvoir :

- obtenir quelques informations de cette machine-là et les amener sur cette machine-ci, ou vice versa. Ceci est réalisé au moyen de la programmation réseau de base ;
- se connecter à une base de données, qui peut résider quelque part sur le réseau. Pour cela, on utilise la Connectivité Bases de Données Java : *Java DataBase Connectivity* (JDBC), qui est une encapsulation des détails confus, et spécifiques à la plate-forme, du SQL (S*tructured Query Language* - langage structuré d'interrogation de bases de données - utilisé pour de nombreux échanges avec des bases de données) ;
- fournir des services via un serveur Web. C'est le rôle des *servlets* Java et des Pages Serveur Java : *Java Server Pages* (JSP) ;
- exécuter de manière transparente des méthodes appartenant à des objets Java résidant sur des machines distantes, exactement comme si ces objets résidaient sur la machine locale. Pour cela, on utilise l'Invocation de Méthode Distante de Java : *Remote Method Invocation* (RMI) ;
- utiliser du code écrit dans d'autres langages, tournant sous d'autres architectures. C'est l'objet de *Common Object Request Broker Architecture* (CORBA), qui est directement mis en oeuvre par Java ;
- séparer les questions relatives à la connectivité de la logique concernant le résultat cherché, et en particulier les connexions aux bases de données incluant la gestion des transactions et la sécurité. C'est le domaine des *Enterprise JavaBeans* (EJB). Les EJB ne représentent pas réellement une architecture distribuée, mais les applications qui en découlent sont couramment utilisées dans un système client-serveur en réseau ;
- ajouter et enlever, facilement et dynamiquement, des fonctionnalités provenant d'un réseau considéré comme un système local. C'est ce que propose la fonctionnalité Jini de Java.

Ce chapitre a pour but d'introduire succinctement toutes ces fonctionnalités. Il faut noter que chacune représente un vaste sujet, et pourrait faire l'objet d'un livre à elle toute seule, aussi ce chapitre n'a d'autre but que de vous rendre ces concepts familiers, et en aucun cas de faire de vous un expert (toutefois, vous aurez largement de quoi faire avec l'information que vous trouverez ici à propos de la programmation réseau, des servlets et des JSP).

La programmation réseau

Une des grandes forces de Java est de pouvoir travailler en réseau sans douleur. Les concepteurs de la bibliothèque réseau de java en ont fait quelque chose d'équivalent à la lecture et l'écriture de fichiers, avec la différence que les « fichiers » résident sur une machine distante et que c'est elle qui décide de ce qu'elle doit faire au sujet des informations que vous demandez ou de celles que vous envoyez. Autant que possible, les menus détails du travail en réseau ont été cachés et sont pris en charge par la JVM et l'installation locale de Java. Le modèle de programmation utilisé est celui d'un fichier ; en réalité, la connexion réseau (Une » *socket* « - littéralement douille, ou prise, NdT) est encapsulée dans des objets stream, ce qui permet d'utiliser les mêmes appels de méthode que pour les autres objets stream. De plus, les fonctionnalités de multithreading de Java viennent à point lorsqu'on doit traiter plusieurs connexions simultanées.

Cette section introduit le support réseau de Java en utilisant des exemples triviaux.

Identifier une machine

Il semble évident que, pour pouvoir appeler une machine depuis une autre, en ayant la certitude d'être connecté à une machine en particulier, il doit exister quelque chose comme un identifiant unique sur le réseau. Les anciens réseaux se contentaient de fournir des noms de machines uniques sur le réseau local. Mais Java travaille sur l'Internet, et cela nécessite un moyen d'identifier chaque machine de manière unique par rapport à toutes les autres *dans le monde entier.* C'est la raison d'être de l'adresse IP (Internet Protocol - protocole internet, NdT) qui existe sous deux formes :

1. La forme familière la forme DNS (*Domain Name System,* Système de Nommage des Domaines). Mon nom de domaine est **bruceeckel.com**, et si j'avais dans mon domaine un ordinateur nommé **Opus**, son nom de domaine serait **Opus.bruceeckel.com**. C'est exactement le type de nom que vous utilisez lorsque vous envoyez du courrier à quelqu'un, et il est souvent associé à une adresse World Wide Web.

2. Sinon, on peut utiliser la forme du quadruplet pointé, c'est à dire quatre nombre séparés par des points, par exemple **123.255.28.120**.

Dans les deux cas, l'adresse IP est représentée en interne comme un nombre sur 32 bits [72] (et donc chaque nombre du quadruplet ne peut excéder 255), et il existe un objet spécial Java pour représenter ce nombre dans l'une des formes décrites ci-dessus en utilisant la méthode de la bibliothèque **java.net : static InetAddress.getByName( )**. Le résultat est un objet du type **InetAddress** qu'on peut utiliser pour construire une socket, « comme on le verra plus loin.

Pour montrer un exemple simple d'utilisation de **InetAddress.getByName( )**, considérons ce qui se passe lorsque vous êtes en communication avec un Fournisseur d'Accès Internet (FAI) - Internet Service Provider (ISP). Chaque fois que vous vous connectez, il vous assigne une adresse IP temporaire. Tant que vous êtes connecté, votre adresse IP est aussi valide que n'importe quelle autre adresse IP sur Internet. Si quelqu'un se connecte à votre machine au moyen de votre adresse IP, alors il peut se connecter à un serveur Web ou FTP qui tournerait sur votre machine. Bien entendu, il faudrait qu'il connaisse votre adresse IP, mais puisqu'une nouvelle vous est assignée à chaque connexion, comment pourrait-il faire ?

Le programme qui suit utilise **InetAddress.getByName( )** pour récupérer votre adresse IP. Pour qu'il fonctionne, vous devez connaître le nom de votre ordinateur. Sous Windows 95/98, aller à Paramètres, « Panneau de Contrôle, « Réseau, « et sélectionnez l'onglet » Identification. » Le Nom d'ordinateur est le nom à utiliser sur la ligne de commande.

```
//: c15:WhoAmI.java
// Affiche votre adresse de réseau lorsque
// vous êtes connectés à Internet.
import java.net.*;

public class WhoAmI {
  public static void main(String[ « args)
      throws Exception {
    if(args.length != 1) {
      System.err.println(
        "Usage: WhoAmI MachineName");
      System.exit(1);
    }
    InetAddress a =
      InetAddress.getByName(args[0]);
    System.out.println(a);
  }
} ///:~
```

Supposons que ma machine ait pour nom » peppy. « Une fois connecté au FAI, je lance le programme :

```
java WhoAmI peppy
```

En retour, j'obtiens un message tel que celui-ci (bien entendu, l'adresse est différente à chaque connexion) :

```
peppy/199.190.87.75
```

Si je donne cette adresse à un ami et qu'il existe un serveur Web tournant sur mon ordinateur, il peut s'y connecter en allant à l'URL *http://199.190.87.75* (du moins tant que je reste connecté). Ceci est parfois une manière commode de distribuer de l'information à d'autres personnes, ou encore de tester la configuration d'un site avant de l'installer sur un « vrai » serveur.

Serveurs et clients

La finalité d'un réseau est de permettre à deux machines de se connecter et ainsi de se « parler ». Une fois que les deux machines se sont trouvées l'une l'autre, elles peuvent entamer une agréable conversation bi-directionnelle. Mais qu'est-ce que chacune peut donc rechercher chez l'autre ? Et d'abord, comment trouvent-elles l'autre ? Tout se passe à peu près comme lorsqu'on est perdu dans un parc d'attractions : une des machines doit attendre l'appel de l'autre sans bouger : « Hé, où êtes-vous ? »]

La machine qui attend est appelée *serveur*, celle qui cherche *client*. Cette distinction n'est importante que tant que le client cherche à se connecter au serveur. Une fois les machines connectées, la communication se traduit par un processus bi-directionnel et on n'a plus à se préoccuper de savoir qui est le serveur et qui est le client.

Le rôle du serveur est donc d'être en attente d'une demande de connexion, ceci est réalisé par l'objet serveur qu'on crée dans ce but. Le rôle du client est de tenter d'établir une connexion avec un

serveur, ceci est réalisé avec un objet client qu'on crée pour cela. Une fois la connexion établie, aussi bien du côté serveur que du côté client, elle se transforme magiquement en un objet flux d'E/S, et dès lors on peut traiter cette connexion comme une lecture ou une écriture dans un fichier. Ainsi, une fois la connexion établie, il ne reste qu'à utiliser les commandes d'E/S familières vues au Chapitre 11. C'est un des aspects agréables des fonctions de Java en réseau.

Tester les programmes hors réseau

Pour toutes sortes de raisons, il se peut qu'on ne dispose pas de machines client et serveur, pas plus que d'un réseau pour tester nos programmes. Par exemple lorsqu'on réalise des exercices dans une situation d'apprentissage, ou bien qu'on écrive des programmes qui ne sont pas encore suffisamment stables pour être mis sur le réseau. Les créateurs du protocole internet (IP) ont bien appréhendé cette question, et ont créé une adresse spéciale appelée **localhost** qui représente une adresse IP en « boucle locale » permettant d'effectuer des tests en se passant de la présence d'un réseau. Voyez ci-dessous la manière générique de réaliser cette adresse en Java :

```
InetAddress addr = InetAddress.getByName(null);
```

En utilisant **getByName( )** avec un argument **null**, cette fonction utilise par défaut **localhost**. **InetAddress** est utilisée pour désigner une machine particulière, et on doit l'initialiser avant d'aller plus loin. On ne peut pas manipuler le contenu d'une **InetAddress** (mais il est possible de l'imprimer, comme on va le voir dans l'exemple suivant). L'unique manière de créer une **InetAddress** passe par l'une des méthodes membre **static** surchargées de cette classe : **getByName( )** (habituellement utilisée), **getAllByName( )** ou **getLocalHost( )**.

Une autre manière de réaliser l'adresse de la boucle locale est d'utiliser la chaîne **localhost**:

```
InetAddress.getByName("localhost"); ,
```

(en supposant que localhost « est décrit dans la table des hôtes de la machine), ou encore en se servant de la forme « quadruplet pointé » pour désigner l'adresse IP réservée à la boucle locale :

```
InetAddress.getByName("127.0.0.1");
```

Les trois formes aboutissent au même résultat.

Les Ports : un emplacement unique dans la machine

Une adresse IP n'est pas suffisante pour identifier un serveur, en effet plusieurs serveurs peuvent coexister sur une même machine. Chaque machine IP contient aussi des *ports*, et lorsqu'on installe un client ou un serveur il est nécessaire de choisir un port convenant aussi bien à la connexion du client qu'à celle du serveur ; si vous donnez rendez-vous à quelqu'un, l'adresse IP représentera le quartier et le port sera le nom du bar.

Le port n'est pas un emplacement physique dans la machine, mais une abstraction de programmation (surtout pour des raisons de comptabilité). Le programme client sait comment se connecter à la machine via son adresse IP, mais comment se connectera-t-il au service désiré (potentiellement, l'un des nombreux services de cette machine) ? C'est pourquoi les numéros de port représentent un deuxième niveau d'adressage. L'idée sous-jacente est que si on s'adresse à un port particulier, en fait on effectue une demande pour le service associé à ce numéro de port. Un exemple simple de service est l'heure du jour. Typiquement, chaque service est associé à un numéro de port

unique sur une machine serveur donnée. Il est de la responsabilité du client de connaître à l'avance quel est le numéro de port associé à un service donné.

Les services système réservent les numéros de ports 1 à 1024, il ne faut donc pas les utiliser, pas davantage que d'autres numéros de ports dont on saurait qu'ils sont utilisés. Le premier nombre choisi pour les exemples de ce livre est le port 8080 (en souvenir de la vénérable vieille puce 8 bits Intel 8080 de mon premier ordinateur, une machine CP/M).

Les sockets

Une *socket* est une abstraction de programmation représentant les extrémités d'une connexion entre deux machines. Pour chaque connexion donnée, il existe une socket sur chaque machine, on peut imaginer un câble virtuel reliant les deux machines, chaque extrémité enfichée dans une socket. Bien entendu, le hardware sous-jacent ainsi que la manière dont les machines sont connectées ne nous intéressent pas. L'essentiel de l'abstraction est qu'on n'a pas à connaître plus que ce qu'il est nécessaire.

En Java, on crée un objet Socket pour établir une connexion vers une autre machine, puis on crée un **InputStream** et un **OutputStream** (ou, avec les convertisseurs appropriés, un **Reader** et un **Writer**) à partir de ce Socket, afin de traiter la connexion en tant qu'objet flux d'E/S. Il existe deux classes Socket basées sur les flux : **ServerSocket** utilisé par un serveur pour [écouter « les connexions entrantes et **Socket** utilisé par un client afin d'initialiser une connexion. Lorsqu'un client réalise une connexion socket, **ServerSocket** retourne (via la méthode **accept( )**) un **Socket** correspondant permettant les communications du côté serveur. À ce moment-là, on a réellement établi une connexion **Socket** à **Socket** et on peut traiter les deux extrémités de la même manière, car elles *sont* alors identiques. On utilise alors les méthodes **getInputStream( )** et **getOutputStream( )** pour réaliser les objets **InputStream** et **OutputStream** correspondants à partir de chaque **Socket**. À leur tour ils peuvent être encapsulés dans des classes buffers ou de formatage tout comme n'importe quel objet flux décrit au Chapitre 11.

La construction du mot **ServerSocket** est un exemple supplémentaire de la confusion du plan de nommage des bibliothèques Java. **ServerSocket** aurait dû s'appeler « ServerConnector »ou bien n'importe quoi qui n'utilise pas le mot Socket. Il faut aussi penser que **ServerSocket** et **Socket** héritent tous deux de la même classe de base. Naturellement, les deux classes ont plusieurs méthodes communes, mais pas suffisamment pour qu'elles aient une même classe de base. En fait, le rôle de **ServerSocket**est d'attendre qu'une autre machine se connecte, puis à ce moment-là de renvoyer un **Socket** réel. C'est pourquoi **ServerSocket** semble mal-nommé, puisque son rôle n'est pas d'être une socket mais plus exactement de créer un objet **Socket**lorsque quelqu'un se connecte.

Cependant, un **ServerSocket** crée physiquement un » serveur « ou, si l'on préfère, une « prise » à l'écoute sur la machine hôte. Cette « prise » est à l'écoute des connexions entrantes et renvoie une « prise » établie (les points terminaux et distants sont définis) via la méthode **accept( )**. La confusion vient du fait que ces deux « prises » (celle qui écoute et celle qui représente la communication établie) sont associées à la même « prise » serveur. La « prise » qui écoute accepte uniquement les demandes de nouvelles connexions, jamais les paquets de données. Ainsi, même si **ServerSocket** n'a pas beaucoup de sens en programmation, il en a physiquement.]

Lorsqu'on crée un **ServerSocket**, on ne lui assigne qu'un numéro de port. Il n'est pas nécessaire de lui assigner une adresse IP parce qu'il réside toujours sur la machine qu'il représente. En revanche, lorsqu'on crée un **Socket**, il faut lui fournir l'adresse IP et le numéro de port sur lequel on essaie de se connecter (toutefois, le **Socket** résultant de la méthode **ServerSocket.accept( )** contient

toujours cette information).

Un serveur et un client vraiment simples

Cet exemple montre l'utilisation minimale d'un serveur et d'un client utilisant des sockets. Le serveur se contente d'attendre une demande de connexion, puis se sert du **Socket** résultant de cette connexion pour créer un **InputStream** et un **OutputStream**. Ces derniers sont convertis en **Reader** et **Writer**, puis encapsulés dans un **BufferedReader** et un **PrintWriter**. À partir de là , tout ce que lit le **BufferedReader** est renvoyé en écho au **PrintWriter** jusqu'à ce qu'il reconnaisse la ligne » END, « et dans ce cas il clôt la connexion.

Le client établit une connexion avec le serveur, puis crée un **OutputStream**et réalise le même type d'encapsulation que le serveur. Les lignes de texte sont envoyées vers le **PrintWriter**-résultant. Le client crée également un **InputStream** (ici aussi, avec la conversion et l'encapsulation appropriées) afin d'écouter le serveur (c'est à dire, dans ce cas, simplement les mots renvoyés en écho).

Le serveur ainsi que le client utilisent le même numéro de port, et le client se sert de l'adresse de boucle locale pour se connecter au serveur sur la même machine, ce qui évite de tester en grandeur réelle sur un réseau (pour certaines configurations, on peut être amené à se *connecter physiquement* à un réseau afin que le programme fonctionne, même si on ne *communique* pas sur ce réseau.)

Voici le serveur :

```
//: c15:JabberServer.java
// Serveur simplifié dont le rôle se limite à
// renvoyer en écho tout ce que le client envoie.
import java.io.*;
import java.net.*;

public class JabberServer {
  // Choisir un port hors de la plage 1-1024:
  public static final int PORT = 8080;
  public static void main(String[ « args)
      throws IOException {
    ServerSocket s = new ServerSocket(PORT);
    System.out.println("Started: " + s);
    try {
      // Le programme stoppe ici et attend
      // une demande de connexion:
      Socket socket = s.accept();
      try {
        System.out.println(
          "Connection accepted: "+ socket);
        BufferedReader in =
          new BufferedReader(
            new InputStreamReader(
              socket.getInputStream()));
        // Le tampon de sortie est vidé
```

```
        // automatiquement par PrintWriter:
        PrintWriter out =
          new PrintWriter(
            new BufferedWriter(
              new OutputStreamWriter(
                socket.getOutputStream())),true);
        while (true) {
          String str = in.readLine();
          if (str.equals("END")) break;
          System.out.println("Echoing: " + str);
          out.println(str);
        }
       // Toujours fermer les deux sockets...
       } finally {
        System.out.println("closing...");
        socket.close();
       }
      } finally {
       s.close();
      }
    }
} ///:~
```

On remarque que **ServerSocket** ne nécessite qu'un numéro de port, et non une adresse IP (puisqu'il tourne sur *cette* machine !). Lorsqu'on appelle **accept( )** la méthode *bloque* jusqu'à ce qu'un client tente de se connecter. Cela signifie qu'elle est en attente d'une demande de connexion, mais elle ne bloque pas les autres processus (voir le Chapitre 14). Une fois la connexion établie, **accept( )** renvoie un objet **Socket** qui représente cette connexion.

La même logique est utilisée pour le **Socket** renvoyé par **accept( )**. Si **accept( )** échoue, nous devons supposer que le **Socket** n'existe pas et qu'il ne monopolise aucune ressource, et donc qu'il n'est pas besoin de le nettoyer. Au contraire, si **accept( )** se termine avec succès, les instructions qui suivent doivent se trouver dans un bloc **try-finally** pour que **Socket**soit toujours nettoyé si l'une d'entre elles échoue. Il faut porter beaucoup d'attention à cela car les sockets sont des ressources importantes qui ne résident pas en mémoire, et on ne doit pas oublier de les fermer (car il n'existe pas en Java de destructeur qui le ferait pour nous).

Le **ServerSocket** et le **Socket** fournis par **accept( )** sont imprimés sur **System.out**. Leur méthode **toString( )** est donc appelée automatiquement. Voici le résultat :

```
ServerSocket[addr=0.0.0.0,PORT=0,localport=8080]
Socket[addr=127.0.0.1,PORT=1077,localport=8080]
```

En raccourci, on peut voir comment cela « colle » avec ce que fait le client.

Le reste du programme consiste seulement à ouvrir des fichiers pour lire et écrire, sauf que **InputStream** et **OutputStream** sont créés à partir de l'objet **Socket**. Les deux objets **InputStream** et **OutputStream** sont convertis en objets **Reader** et **Writer** au moyen des « classes de conversion **InputStreamReader** et **OutputStreamWriter**, respectivement. On aurait pu travailler directement avec les classes Java 1.0 **InputStream** et **OutputStream**, mais pour la sortie il y a un avantage cer-

tain à utiliser l'approche **Writer**. C'est évident avec **PrintWriter**, qui possède un constructeur surchargé prenant en compte un deuxième argument, un flag **boolean** indiquant que le tampon de sortie doit être automatiquement vidé après chaque **println( )** (mais *non*après les instructions **print( )**). Chaque fois qu'on écrit sur **out**, son buffer doit être vidé afin que l'information parte sur le réseau. Le vidage du tampon est important dans cet exemple particulier car aussi bien le serveur que le client attendent de l'autre une ligne complète avant de la traiter. Si le tampon n'est pas vidé à chaque ligne, l'information ne circule pas sur le réseau tans que le buffer n'est pas plein, ce qui occasionnerait de nombreux problèmes dans cet exemple.

Lorsqu'on écrit des programmes réseau, il faut être très attentif à l'utilisation du vidage automatique de buffer. Chaque fois que l'on vide un buffer, un paquet est créé et envoyé. Dans notre exemple, c'est exactement ce que nous recherchons, puisque si le paquet contenant la ligne n'est pas envoyé, l'échange entre serveur et client sera stoppé. Dit d'une autre manière, la fin de ligne est la fin du message. Mais dans de nombreux cas, les messages ne sont pas découpés en lignes et il sera plus efficace de ne pas utiliser le vidage automatique de buffer et à la place de laisser le gestionnaire du buffer décider du moment pour construire et envoyer un paquet. De cette manière, les paquets seront plus importants et le processus plus rapide.

Remarquons que, comme tous les flux qu'on peut ouvrir, ceux-ci sont tamponnés. À la fin de ce chapitre, vous trouverez un exercice montrant ce qui se passe lorsqu'on ne tamponne pas les flux (tout est ralenti).

La boucle **while** infinie lit les lignes depuis le **BufferedReader in**et écrit sur **System.out** et **PrintWriter out**. Remarquons que **in** et **out** peuvent être n'importe quel flux, ils sont simplement connectés au réseau.

Lorsque le client envoie une ligne contenant juste le mot END, « la boucle est interrompue et le programme ferme le **Socket**.

Et voici le client :

```
//: c15:JabberClient.java
// Client simplifié se contentant d'envoyer
// des lignes au serveur et de lire les lignes
// que le serveur lui envoie.
import java.net.*;
import java.io.*;

public class JabberClient {
  public static void main(String[ « args)
      throws IOException {
// Appeler getByName() avec un argument null revient
    // à utiliser une adresse IP spéciale "Boucle Locale"
// pour faire des tests réseau sur une seule machine.

     InetAddress addr =
      InetAddress.getByName(null);
    // Il est également possible d'utiliser
    // l'adresse ou le nom:
    // InetAddress addr =
    //    InetAddress.getByName("127.0.0.1");
```

```
        // InetAddress addr =
        //   InetAddress.getByName("localhost");
        System.out.println("addr = " + addr);
        Socket socket =
          new Socket(addr, JabberServer.PORT);
        // Le code doit être inclus dans un bloc
        // try-finally afin de garantir
        // que socket sera fermé:
        try {
          System.out.println("socket = " + socket);
          BufferedReader in =       new BufferedReader(
             new InputStreamReader(
               socket.getInputStream()));
          // Le tampon de sortie est automatiquement
          // vidé par PrintWriter:
          PrintWriter out =       new PrintWriter(
             new BufferedWriter(
               new OutputStreamWriter(
                 socket.getOutputStream())),true);
          for(int i = 0; i < 10; i ++) {
            out.println("howdy " + i);
            String str = in.readLine();
            System.out.println(str);
          }
          out.println("END");
        } finally {
          System.out.println("closing...");
          socket.close();
        }
      }
    } ///:~
```

La méthode **main( )** montre qu'il existe trois manières de produire l'**InetAddress** de l'adresse IP de la boucle locale : avec **null**, **localhost**, ou bien l'adresse réservée et explicite **127.0.0.1**. Bien entendu, si l'on désire se connecter à une machine du réseau, il suffit d'y substituer son adresse. Lorsque **InetAddress addr** est imprimée (via l'appel automatique de sa méthode **toString( )**), voici le résultat :

```
    localhost/127.0.0.1
```

Lorsque **getByName( )** a été appelée avec un argument **null**, elle a cherché par défaut **localhost**, ce qui a fourni l'adresse spéciale **127.0.0.1**.

Remarquons que le **Socket** nommé **socket** est créé avec **InetAddress** ainsi que le numéro de port. Pour comprendre ce qui se passe lorsqu'on imprime un de ces objets **Socket**, il faut se souvenir qu'une connexion Internet est déterminée de manière unique à partir de quatre données : **clientHost**, **clientPortNumber**, **serverHost**, et **serverPortNumber**. Lorsque le serveur démarre, il prend en compte le port qui lui est assigné (8080) sur la machine locale (127.0.0.1). Lorsque le client démarre, le premier port suivant disponible sur sa machine lui est alloué, 1077 dans ce cas, qui se

trouve sur la même machine (127.0.0.1) que le serveur. Maintenant, pour que les données circulent entre le client et le serveur, chacun doit savoir où les envoyer. En conséquence, pendant la connexion au serveur connu, le client envoie une adresse de retour afin que le serveur sache où envoyer ses données. Ce qu'on peut voir dans la sortie de l'exemple côté serveur :

```
Socket[addr=127.0.0.1,port=1077,localport=8080]
```

Cela signifie que le serveur vient d'accepter une demande de connexion provenant de 127.0.0.1 sur le port 1077 alors qu'il est à l'écoute sur le port local (8080). Du côté client :

```
Socket[addr=localhost/127.0.0.1,PORT=8080,localport=1077]
```

ce qui signifie que le client vient d'établir une connexion à 127.0.0.1 sur le port 8080 en utilisant le port local 1077.

Il faut remarquer que chaque fois qu'on relance le client, le numéro du port local est incrémenté. Il commence à 1025 (un après le bloc de ports réservé) et ne cesse d'augmenter jusqu'à ce qu'on reboote la machine, auquel cas il recommence à 1025 (sur les machines UNIX, lorsque la limite supérieure de la plage accordée aux sockets est atteinte, on recommence avec la plus petite valeur possible).

L'objet **Socket** créé, le processus consistant à en faire un **BufferedReader** puis un **PrintWriter** est le même que pour le serveur (encore une fois, dans les deux cas on commence par un **Socket**). Ici, le client entame la conversation en envoyant la chaîne « [howdy] » suivie d'un nombre. Remarquons que le buffer doit être vidé à nouveau (ce qui est automatique via le deuxième argument du constructeur de **PrintWriter**). Si le buffer n'est pas vidé, la conversation est complètement suspendue parce que le « howdy » initial ne sera jamais envoyé (le buffer n'est pas assez rempli pour que l'envoi se fasse automatiquement). Chaque ligne renvoyée par le serveur est écrite sur **System.out** pour vérifier que tout fonctionne correctement. Pour arrêter l'échange, le client envoie le mot connu » END. Si le client ne se manifeste plus, le serveur lance une exception.

Remarquons qu'ici aussi le même soin est apporté pour assurer que la ressource réseau que représente le **Socket** est relâchée correctement, au moyen d'un bloc **try-finally**.

Les sockets fournissent une connexion » dédiée « qui persiste jusqu'à ce qu'elle soit explicitement déconnectée (la connexion dédiée peut encore être rompue de manière non explicite si l'un des deux côtés ou un lien intermédiaire de la connexion se plante). La conséquence est que les deux parties sont verrouillées en communication et que la connexion est constamment ouverte. Cela peut paraître une approche logique du travail en réseau, en fait on surcharge le réseau. Plus loin dans ce chapitre on verra une approche différente du travail en réseau, dans laquelle les connexions seront temporaires.

Servir des clients multiples

Le programme **JabberServer** fonctionne, mais ne peut traiter qu'un client à la fois. Dans un serveur typique, on désire traiter plusieurs clients en même temps. La réponse est le multithreading, et dans les langages ne supportant pas cette fonctionnalité cela entraîne toutes sortes de complications. Dans le Chapitre 14 nous avons vu que le multithreading de Java est aussi simple que possible, si l'on considère le multithreading comme un sujet quelque peu complexe. Parce que gérer des threads est relativement simple en Java, écrire un serveur prenant en compte de multiples clients est relativement facile.

L'idée de base est de construire dans le serveur un seul **ServerSocket** et d'appeler **accept( )** pour attendre une nouvelle connexion. Au retour d'**accept( )**, on crée un nouveau thread utilisant le **Socket** résultant, thread dont le travail est de servir ce client particulier. Puis on appelle à nouveau **accept( )** pour attendre un nouveau client.

Dans le code serveur suivant, on remarquera qu'il ressemble à l'exemple **JabberServer.java** sauf que toutes les opérations destinées à servir un client particulier ont été déplacées dans une classe thread séparée :

```
//: c15:MultiJabberServer.java
// Un serveur utilisant le multithreading
// pour traiter un nombre quelconque de clients.
import java.io.*;
import java.net.*;

class ServeOneJabber extends Thread {
  private Socket socket;
  private BufferedReader in;
  private PrintWriter out;
  public ServeOneJabber(Socket s)
      throws IOException {
    socket = s;
    in =
      new BufferedReader(
        new InputStreamReader(
          socket.getInputStream()));
    // Autoriser l'auto-vidage:
    out =
      new PrintWriter(
        new BufferedWriter(
          new OutputStreamWriter(
            socket.getOutputStream())), true);
    // Si l'un des appels ci-dessus résulte en une
    // exception, l'appelant a la responsabilité
    // de fermer socket. Sinon le thread
    // s'en chargera.
    start(); // Appelle run()
  }
  public void run() {
    try {
      while (true) {
        String str = in.readLine();
        if (str.equals("END")) break;
        System.out.println("Echoing: " + str);
        out.println(str);
      }
      System.out.println("closing...");
    } catch(IOException e) {
      System.err.println("IO Exception");
```

```
      } finally {
        try {
          socket.close();
        } catch(IOException e) {
          System.err.println("Socket not closed");
        }
      }
    }
  }

  public class MultiJabberServer {
    static final int PORT = 8080;
    public static void main(String[ « args)
        throws IOException {
      ServerSocket s = new ServerSocket(PORT);
      System.out.println("Server Started");
      try {
        while(true) {
  // On attend ici jusqu'à avoir
  // une demande de connexion:
          Socket socket = s.accept();
          try {
            new ServeOneJabber(socket);
          } catch(IOException e) {
            // En cas d'échec, fermer l'objet socket,
            // sinon le thread le fermera:
            socket.close();
          }
        }
      } finally {
        s.close();
      }
    }
  } ///:~
```

Chaque fois qu'un nouveau client se connecte, le thread **ServeOneJabber** prend l'objet **Socket** produit par **accept( )** dans **main( )**. Puis, comme auparavant, il crée un **BufferedReader** et un objet **PrintWriter**(avec auto-vidage du buffer) à partir du **Socket**. Finalement, il appelle la méthode spéciale **start( )** de la classe **Thread** qui initialise le thread puis appelle **run( )**. On réalise ainsi le même genre de traitement que dans l'exemple précédent : lire quelque chose sur la socket et le renvoyer en écho jusqu'à l'arrivée du signal spécial » END.

À nouveau, il nous faut penser à notre responsabilité en ce qui concerne la ressource socket. Dans ce cas, la socket est créée hors de **ServeOneJabber** et donc la responsabilité doit être partagée. Si le constructeur de **ServeOneJabber** échoue, il suffit qu'il lance une exception vers l'appelant, qui nettoiera alors le thread. Mais dans le cas contraire, l'objet **ServeOneJabber** a la responsabilité de nettoyer le thread, dans sa méthode **run( )**.

Remarquons la simplicité de **MultiJabberServer**. Comme auparavant, on crée un **ServerSo-**

cket, et on appelle **accept( )** pour autoriser une nouvelle connexion. Mais cette fois, la valeur de retour de **accept( )** (un **Socket**) est passée au constructeur de **ServeOneJabber,** qui crée un nouveau thread afin de prendre en compte cette connexion. La connexion terminée, le thread se termine tout simplement.

Si la création du **ServerSocket** échoue, à nouveau une exception est lancée par **main( )**. En cas de succès, le bloc **try-finally** extérieur garantit le nettoyage. Le bloc **try-catch** intérieur protège d'une défaillance du constructeur **ServeOneJabber** ; en cas de succès, le thread **ServeOneJabber** fermera la socket associée.

Afin de tester que le serveur prend réellement en compte plusieurs clients, le programme suivant crée beaucoup de clients (sous la forme de threads) et se connecte au même serveur. Le nombre maximum de threads autorisés est fixé par la constante **final int MAX_THREADS**.

```
//: c15:MultiJabberClient.java
// Client destiné à tester MultiJabberServer
// en lançant des clients multiple.
import java.net.*;
import java.io.*;

class JabberClientThread extends Thread {
  private Socket socket;
  private BufferedReader in;
  private PrintWriter out;
  private static int counter = 0;
  private int id = counter++;
  private static int threadcount = 0;
  public static int threadCount() {
    return threadcount;
  }
  public JabberClientThread(InetAddress addr) {
    System.out.println("Making client " + id);
    threadcount++;
    try {
      socket =
        new Socket(addr, MultiJabberServer.PORT);
    } catch(IOException e) {
      System.err.println("Socket failed");
      // Si la création de socket échoue,
      // il n'y a rien à nettoyer.
    }
    try {
      in =
        new BufferedReader(
          new InputStreamReader(
            socket.getInputStream()));
      // Autoriser l'auto-vidage du tampon:
      out =
        new PrintWriter(
```

```
            new BufferedWriter(
              new OutputStreamWriter(
                socket.getOutputStream())), true);
          start();
        } catch(IOException e) {
          // socket doit être fermé sur n'importe quelle
          // erreur autre que celle de son constructeur:
          try {
            socket.close();
          } catch(IOException e2) {
            System.err.println("Socket not closed");
          }
        }
        // Sinon socket doit être fermé par
        // la méthode run() du thread.
      }
      public void run() {
        try {
          for(int i = 0; i < 25; i++) {
            out.println("Client " + id + ": " + i);
            String str = in.readLine();
            System.out.println(str);
          }
          out.println("END");
        } catch(IOException e) {
          System.err.println("IO Exception");
        } finally {
          // Toujours fermer:
          try {
            socket.close();
          } catch(IOException e) {
            System.err.println("Socket not closed");
          }
          threadcount--; // Fin de ce thread
        }
      }
    }

    public class MultiJabberClient {
      static final int MAX_THREADS = 40;
      public static void main(String[ « args)
          throws IOException, InterruptedException {
        InetAddress addr =
          InetAddress.getByName(null);
        while(true) {
          if(JabberClientThread.threadCount()
            < MAX_THREADS)
```

```
            new JabberClientThread(addr);
          Thread.currentThread().sleep(100);
        }
      }
    } ///:~
```

La variable **threadCount** garde la trace du nombre de **JabberClientThread** existant actuellement. Elle est incrémentée dans le constructeur et décrémentée lorsque **run( )** termine (ce qui signifie que le thread est en train de se terminer). Dans **MultiJabberClient.main( )** le nombre de threads est testé, on arrête d'en créer s'il y en a trop. Dans ce cas la méthode s'endort. De cette manière, certains threads peuvent se terminer et de nouveaux pourront être créés. Vous pouvez faire l'expérience, en changeant la valeur de **MAX_THREADS**, afin de savoir à partir de quel nombre de connexions votre système commence à avoir des problèmes.

Les Datagrammes

Les exemples que nous venons de voir utilisent le Protocole de Contrôle de Transmission *Transmission Control Protocol* (TCP, connu également sous le nom de *stream-based sockets*- sockets basés sur les flux, NdT), qui est conçu pour une sécurité maximale et qui garantit que les données ne seront pas perdues. Il permet la retransmission des données perdues, il fournit plusieurs chemins au travers des différents routeurs au cas où l'un d'eux tombe en panne, enfin les octets sont délivrés dans l'ordre où ils ont été émis. Ces contrôles et cette sécurité ont un prix : TCP a un overhead élevé.

Il existe un deuxième protocole, appelé *User Datagram Protocol* (UDP), qui ne garantit pas que les paquets seront acheminés ni qu'ils arriveront dans l'ordre d'émission. On l'appelle protocole peu sûr « (TCP est un protocole]sûr), ce qui n'est pas très vendeur, mais il peut être très utile car il est beaucoup plus rapide. Il existe des applications, comme la transmission d'un signal audio, pour lesquelles il n'est pas critique de perdre quelques paquets çà et là mais où en revanche la vitesse est vitale. Autre exemple, considérons un serveur de date et heure, pour lequel on n'a pas vraiment à se préoccuper de savoir si un message a été perdu. De plus, certaines applications sont en mesure d'envoyer à un serveur un message UDP et ensuite d'estimer que le message a été perdu si elles n'ont pas de réponse dans un délai raisonnable.

En règle générale, la majorité de la programmation réseau directe est réalisée avec TCP, et seulement occasionnellement avec UDP. Vous trouverez un traitement plus complet sur UDP, avec un exemple, dans la première édition de ce livre (disponible sur le CD ROM fourni avec ce llvre, ou en téléchargement libre depuis *www.BruceEckel.com*).

Utiliser des URLs depuis un applet

Un applet a la possibilité d'afficher n'importe quelle URL au moyen du navigateur sur lequel il tourne. Ceci est réalisé avec la ligne suivante :

```
getAppletContext().showDocument(u);
```

dans laquelle **u** est l'objet **URL**. Voici un exemple simple qui nous redirige vers une autre page Web. Bien que nous soyons seulement redirigés vers une page HTML, nous pourrions l'être de la même manière vers la sortie d'un programme CGI.

```
//: c15:ShowHTML.java
// <applet code=ShowHTML width=100 height=50>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.net.*;
import java.io.*;
import com.bruceeckel.swing.*;

public class ShowHTML extends JApplet {
  JButton send = new JButton("Go");
  JLabel l = new JLabel();
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    send.addActionListener(new Al());
    cp.add(send);
    cp.add(l);
  }
  class Al implements ActionListener {
    public void actionPerformed(ActionEvent ae) {
      try {
        // Ceci pourrait être un programme CGI
        // au lieu d'une page HTML.
        URL u = new URL(getDocumentBase(),
          "FetcherFrame.html");
        // Afficher la sortie de l'URL en utilisant
        // le navigateur Web, comme une page ordinaire:
        getAppletContext().showDocument(u);
      } catch(Exception e) {
        l.setText(e.toString());
      }
    }
  }
  public static void main(String[ « args) {
    Console.run(new ShowHTML(), 100, 50);
  }
} ///:~
```

Il est beau de voir combien la classe **URL** nous évite la complexité. Il est possible de se connecter à un serveur Web sans avoir à connaître ce qui se passe à bas niveau.

Lire un fichier depuis un serveur

Une variante du programme ci-dessus consiste à lire un fichier situé sur le serveur. Dans ce cas, le fichier est décrit par le client :

```
//: c15:Fetcher.java
// <applet code=Fetcher width=500 height=300>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.net.*;
import java.io.*;
import com.bruceeckel.swing.*;

public class Fetcher extends JApplet {
  JButton fetchIt= new JButton("Fetch the Data");
  JTextField f =
    new JTextField("Fetcher.java", 20);
  JTextArea t = new JTextArea(10,40);
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    fetchIt.addActionListener(new FetchL());
    cp.add(new JScrollPane(t));
    cp.add(f); cp.add(fetchIt);
  }
  public class FetchL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      try {
        URL url = new URL(getDocumentBase(),
          f.getText());
        t.setText(url + "\n");
        InputStream is = url.openStream();
        BufferedReader in = new BufferedReader(
          new InputStreamReader(is));
        String line;
        while ((line = in.readLine()) != null)
          t.append(line + "\n");
      } catch(Exception ex) {
        t.append(ex.toString());
      }
    }
  }
  public static void main(String[ « args) {
    Console.run(new Fetcher(), 500, 300);
  }
} ///:~
```

La création de l'objet **URL** est semblable à celle de l'exemple précédent. Comme d'habitude, **getDocumentBase( )** est le point de départ, mais cette fois le nom de fichier est lu depuis le **JTextField**. Un fois l'objet **URL** créé, sa version **String** est affichée dans le **JTextArea** afin que nous puissions la visualiser. Puis un **InputStream** est créé à partir de l'**URL**, qui dans ce cas va simple-

ment créer un flux de caractères vers le fichier. Après avoir été convertie vers un **Reader** et bufferisée, chaque ligne est lue et ajoutée au **JTextArea**. Notons que le **JTextArea** est placé dans un **JScrollPane** afin que le scrolling soit pris en compte automatiquement.

En savoir plus sur le travail en réseau

En réalité, bien d'autres choses à propos du travail en réseau pourraient être traitées dans cette introduction. Le travail en réseau Java fournit également un support correct et étendu pour les URLs, incluant des handlers de protocole pour les différents types de contenu que l'on peut trouver sur un site Internet. Vous découvrirez d'autres fonctionnalités réseau de Java, complètement et minutieusement décrites dans *Java Network Programming* de Elliotte Rusty Harold (O.Reilly, 1997).

Se connecter aux bases de données : Java Database Connectivity (JDBC)

On a pu estimer que la moitié du développement de programmes implique des opérations client/serveur. Une des grandes promesses tenues par Java fut sa capacité à construire des applications de base de données client/serveur indépendantes de la plate-forme. C'est ce qui est réalisé avec Java DataBase Connectivity (JDBC).

Un des problèmes majeurs rencontrés avec les bases de données fut la guerre des fonctionnalités entre les compagnies fournissant ces bases de données. Il existe un langage standard de base de données, Structured Query Language (SQL-92), mais il est généralement préférable de connaître le vendeur de la base de données avec laquelle on travaille, malgré ce standard. JDBC est conçu pour être indépendant de la plate-forme, ainsi lorsqu'on programme on n'a pas à se soucier de la base de données qu'on utilise. Il est toutefois possible d'effectuer à partir de JDBC des appels spécifiques vers une base particulière afin de ne pas être limité dans ce que l'on pourrait faire.

Il existe un endroit pour lequel les programmeurs auraient besoin d'utiliser les noms de type SQL, c'est dans l'instruction SQL TABLE CREATE lorsqu'on crée une nouvelle table de base de données et qu'on définit le type SQL de chaque colonne. Malheureusement il existe des variantes significatives entre les types SQL supportés par différents produits base de données. Des bases de données différentes supportant des types SQL de même sémantique et de même structure peuvent appeler ces types de manières différentes. La plupart des bases de données importantes supportent un type de données SQL pour les grandes valeurs binaires : dans Oracle ce type s'appelle LONG RAW, Sybase le nomme IMAGE, Informix BYTE, et DB2 LONG VARCHAR FOR BIT DATA. Par conséquent, si on a pour but la portabilité des bases de données il faut essayer de n'utiliser que les identifiants de type SQL génériques.

La portabilité est une question d'actualité lorsqu'on écrit pour un livre dont les lecteurs vont tester les exemples avec toute sorte de stockage de données inconnus. J'ai essayé de rendre ces exemples aussi portables qu'il était possible. Remarquez également que le code spécifique à la base de données a été isolé afin de centraliser toutes les modifications que vous seriez obligés d'effectuer pour que ces exemples deviennent opérationnels dans votre environnement.

JDBC, comme bien des APIs Java, est conçu pour être simple. Les appels de méthode que l'on utilise correspondent aux opérations logiques auxquelles on pense pour obtenir des données depuis une base de données, créer une instruction, exécuter la demande, et voir le résultat.

Pour permettre cette indépendance de plate-forme, JDBC fournit un *gestionnaire de driver* qui maintient dynamiquement les objets driver nécessités par les interrogations de la base. Ainsi si

on doit se connecter à trois différentes sortes de base, on a besoin de trois objets driver différents. Les objets driver s'enregistrent eux-même auprès du driver manager lors de leur chargement, et on peut forcer le chargement avac la méthode **Class.forName( )**.

Pour ouvrir une base de données, il faut créer une « URL]de base de données » qui spécifie :

1. Qu'on utilise JDBC avec « jdbc ».
2. Le « sous-protocole » : le nom du driver ou le nom du mécanisme de connectivité à la base de données. Parce que JDBC a été inspiré par ODBC, le premier sous-protocole disponible est la « passerelle » jdbc-odbc », « spécifiée par « odbc »
3. L'identifiant de la base de données. Il varie avec le driver de base de donnée utilisé, mais il existe généralement un nom logique qui est associé par le software d'administration de la base à un répertoire physique où se trouvent les tables de la base de données. Pour qu'un identifiant de base de données ait une signification, il faut l'enregistrer en utilisant le software d'administration de la base (cette opération varie d'une plate-forme à l'autre).

Toute cette information est condensée dans une chaîne de caractères, l'URL de la base de données.]Par exemple, pour se connecter au moyen du protocole ODBC à une base appelée » people, « l'URL de base de données doit être :

```
String dbUrl = "jdbc:odbc:people";
```

Si on se connecte à travers un réseau, l'URL de base de données doit contenir l'information de connexion identifiant la machine distante, et peut alors devenir intimidante. Voici en exemple la base de données CloudScape que l'on appelle depuis un client éloigné en utilisant RMI :

```
jdbc:rmi://192.168.170.27:1099/jdbc:cloudscape:db
```

En réalité cette URL de base de données comporte deux appels jdbc en un. La première partie jdbc:rmi://192.168.170.27:1099/ « utilise RMI pour effectuer une connexion sur le moteur distant de base de données à l'écoute sur le port 1099 de l'adresse IP 192.168.170.27. La deuxième partie de l'URL, [jdbc:cloudscape:db] représente une forme plus connue utilisant le sous-protocole et le nom de la base, mais elle n'entrera en jeu que lorsque la première section aura établi la connexion à la machine distante via RMI.

Lorsqu'on est prêt à se connecter à une base, on appelle la méthode statique **DriverManager.getConnection( )** en lui fournissant l'URL de base de données, le nom d'utilisateur et le mot de passe pour accéder à la base. On reçoit en retour un objet **Connection** que l'on peut ensuite utiliser pour effectuer des demandes et manipuler la base de données.

L'exemple suivant ouvre une base d'« information de contact » et cherche le nom de famille d'une personne, donné sur la ligne de commande. Il sélectionne d'abord les noms des personnes qui ont une adresse email, puis imprime celles qui correspondent au nom donné :

```
//: c15:jdbc:Lookup.java
// Cherche les adresses email dans une
// base de données locale en utilisant JDBC.
import java.sql.*;

public class Lookup {
  public static void main(String[ « args)
```

```
    throws SQLException, ClassNotFoundException {
    String dbUrl = "jdbc:odbc:people";
    String user = "";
    String password = "";
    // Charger le driver (qui s'enregistrera lui-même)
    Class.forName(
      "sun.jdbc.odbc.JdbcOdbcDriver");
    Connection c = DriverManager.getConnection(
      dbUrl, user, password);
    Statement s = c.createStatement();
    // code SQL:
    ResultSet r =
      s.executeQuery(
        "SELECT FIRST, LAST, EMAIL " +
        "FROM people.csv people " +
        "WHERE " +
        "(LAST='" + args[0 « + "') " +
        " AND (EMAIL Is Not Null) " +
        "ORDER BY FIRST");
    while(r.next()) {
      // minuscules et majuscules n'ont
      // aucune importance:
      System.out.println(
        r.getString("Last") + ", "
        + r.getString("fIRST")
        + ": " + r.getString("EMAIL") );
    }
    s.close(); // fermer également ResultSet
  }
} ///:~
```

Une URL de base de données est créée comme précédemment expliqué. Dans cet exemple, il n'y a pas de protection par mot de passe, c'est pourquoi le nom d'utilisateur et le mot de passe sont des chaînes vides.

Une fois la connexion établie avec **DriverManager.getConnection( )** l'objet résultant **Connection** sert à créer un objet **Statement** au moyen de la méthode **createStatement( )**. Avec cet objet **Statement**, on peut appeler **executeQuery( )** en lui passant une chaîne contenant une instruction standard SQL-92 (il n'est pas difficile de voir comment générer cette instruction automatiquement, on n'a donc pas à connaître grand'chose de SQL).

La méthode **executeQuery( )** renvoie un objet **ResultSet**, qui est un itérateur : la méthode **next( )** fait pointer l'objet itérateur sur l'enregistrement suivant dans l'instruction, ou bien renvoie la valeur **false** si la fin de l'ensemble résultat est atteinte. On obtient toujours un objet **ResultSet** en réponse à la méthode **executeQuery( )** même lorsqu'une demande débouche sur un ensemble vide (autrement dit aucune exception n'est générée). Notons qu'il faut appeler la méthode **next( )** au moins une fois avant de tenter de lire un enregistrement de données. Si l'ensemble résultant est vide, ce premier appel de **next( )** renverra la valeur **false**. Pour chaque enregistrement dans l'ensemble résultant, on peut sélectionner les champs en utilisant (entre autres solutions) une chaîne représen-

tant leur nom. Remarquons aussi que la casse du nom de champ est ignorée dans les transactions avec une base de donnée. Le type de données que l'on veut récupérer est déterminé par le nom de la méthode appelée : **getInt( )**, **getString( )**, **getFloat( )**, etc. Á ce moment, on dispose des données de la base en format Java natif et on peut les traiter comme bon nous semble au moyen du code Java habituel.

Faire fonctionner l'exemple

Avec JDBC, il est relativement simple de comprendre le code. Le côté difficile est de faire en sorte qu'il fonctionne sur votre système particulier. La raison de cette difficulté est que vous devez savoir comment charger proprement le driver JDBC, et comment initialiser une base de données en utilisant le software d'administration de la base.

Bien entendu, ce travail peut varier de façon radicale d'une machine à l'autre, mais la manière dont j'ai procédé pour le faire fonctionner sur un système Windows 32 bits vous donnera sans doute des indications qui vous aideront à attaquer votre propre situation.

Étape 1 : Trouver le Driver JDBC

Le programme ci-dessus contient l'instruction :

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

Ceci suggère une structure de répertoire, ce qui est trompeur. Avec cette installation particulière du JDK 1.1, il n'existe pas de fichier nommé **JdbcOdbcDriver.class**, et si vous avez cherché ce fichier après avoir regardé l'exemple vous avez dû être déçus. D'autre exemples publiés utilisent un pseudo nom, comme myDriver.ClassName, « ce qui nous aide encore moins. En fait, l'instruction de chargement du driver jdbc-odbc (le seul existant actuellement dans le JDK) apparaît en peu d'endroits dans la documentation en ligne (en particulier, une page appelée JDBC-ODBC Bridge Driver]). Si l'instruction de chargement ci-dessus ne fonctionne pas, il est possible que le nom ait été changé à l'occasion d'une évolution de version Java, et vous devez repartir en chasse dans la documention.

Étape 2 : Configurer la base de données

Ici aussi, ceci est spécifique à Windows 32-bits ; vous devrez sans doute rechercher quelque peu pour savoir comment faire sur votre propre plate-forme.

Pour commencer, ouvrir le panneau de configuration. Vous devriez trouver deux icônes traitant de ODBC. « Il faut utiliser celle qui parle de ODBC 32-bits, l'autre n'étant là que pour la compatibilité ascendante avec le software ODBC 16-bits, et ne serait d'aucune utilité pour JDBC. En ouvrant l'icône ODBC 32-bits, vous allez voir une boîte de dialogue à onglets, parmi lesquels User DSN, » System DSN,[File DSN, « etc., DSN signifiant » Data Source Name. Il apparaît que pour la passerelle JDBC-ODBC, le seul endroit important pour créer votre base de données est System DSN, mais vous devez également tester votre configuration et créer des demandes, et pour cela vous devez également créer votre base dans File DSN. Ceci permet à l'outil Microsoft Query (qui fait partie de Microsoft Office) de trouver la base. Remarquez que d'autres outils de demande sont disponibles chez d'autres vendeurs.

La base de données la plus intéressante est l'une de celles que vous utilisez déjà. Le standard ODBC supporte différents formats de fichier y compris les vénérables chevaux de labour tels que

DBase. Cependant, il inclut aussi le format « ASCII, champs séparés par des virgules », que n'importe quel outil de données est capable de créer. En ce qui me concerne, j'ai simplement pris ma base de données « people » « que j'ai maintenue depuis des années au moyen de divers outils de gestion d'adresse et que j'ai exportée en tant que fichier « ASCII à champs séparés par des virgules » (ils ont généralement une extension **.csv**). Dans la section » System DSN « j'ai choisi [Add, « puis le driver texte pour mon fichier ASCII csv, puis dé-coché « [use current directory » « pour me permettre de spécifier le répertoire où j'avais exporté mon fichier de données.

Remarquez bien qu'en faisant cela vous ne spécifiez pas réellement un fichier, mais seulement un répertoire. Ceci parce qu'une base de données se trouve souvent sous la forme d'un ensemble de fichiers situés dans un même répertoire (bien qu'elle puisse aussi bien se trouver sous d'autres formes). Chaque fichier contient généralement une seule table, et une instruction SQL peut produire des résultats issus de plusieurs tables de la base (on appelle ceci une *relation*). Une base contenant une seule table (comme ma base « [people ») est généralement appelée *flat-file database*. La plupart des problèmes qui vont au-delà du simple stockage et déstockage de données nécessitent des tables multiples mises en relation par des *relations*afin de fournir les résultats voulus, on les appelle bases de données *relationnelles*.

Étape 3 : Tester la configuration

Pour tester la configuration il faut trouver un moyen de savoir si la base est visible depuis un programme qui l'interrogerait. Bien entendu, on peut tout simplement lancer le programme exemple JDBC ci-dessus, en incluant l'instruction :

```
Connection c = DriverManager.getConnection(
  dbUrl, user, password);
```

Si une exception est lancée, c'est que la configuration était incorrecte.

Cependant, à ce point, il est très utile d'utiliser un outil de génération de requêtes. J'ai utilisé Microsoft Query qui est livré avec Microsoft Office, mais vous pourriez préférer un autre outil. L'outil de requête doit connaître l'emplacement de la base, et Microsoft Query exigeait que j'ouvre l'onglet administrateur ODBC » File DSN « et que j'ajoute une nouvelle entrée, en spécifiant à nouveau le driver texte et le répertoire contenant ma base de données. On peut donner n'importe quel nom à cette entrée, mais il est préférable d'utiliser le nom déjà fourni dans l'onglet » System DSN.]

Ceci fait, vous saurez si votre base est disponible en créant une nouvelle requête au moyen de votre générateur de requêtes.

Étape 4 : Générer votre requête SQL

La requête créée avec Microsoft Query m'a montré que ma base de données était là et prête à fonctionner, mais a aussi généré automatiquement le code SQL dont j'avais besoin pour l'insérer dans mon programme Java. Je voulais une requête qui recherche les enregistrements contenant le même nom que celui qui était fourni sur la ligne de commande d'appel du programme. Ainsi pour commencer, j'ai cherché un nom particulier, [Eckel.]Je voulais également n'afficher que ceux qui avaient une adresse email associée. Voici les étapes que j'ai suivi pour créer cette requête :

1. Ouvrir une nouvelle requête au moyen du Query Wizard. Sélectionner la base « [people » (ceci est équivalent à ouvrir la connexion avec la base au moyen de l'URL de base de données appropriée).

2. Dans la base, sélectionner la table « people ». Dans la table, choisir les colonnes FIRST, LAST, et EMAIL.

3. Sous « Filter Data », choisir LAST et sélectionner « equals » avec comme argument « Eckel ». Cliquer sur le bouton radio « And ».

4. Choisir EMAIL et sélectionner « Is not Null ».

5. Sous « Sort By », « choisir FIRST.

Le résultat de cette requête vous montrera si vous obtenez ce que vous vouliez.

Maintenant cliquez sur le bouton SQL et sans aucun effort de votre part vous verrez apparaître le code SQL correct, prêt pour un copier-coller. Le voici pour cette requête :

```
SELECT people.FIRST, people.LAST, people.EMAIL
FROM people.csv people
WHERE (people.LAST='Eckel') AND
(people.EMAIL Is Not Null)
ORDER BY people.FIRST
```

Il serait très facile de se tromper dans le cas de requêtes plus compliquées, mais en utilisant un générateur de requêtes vous pouvez tester votre requête interactivement et générer automatiquement le code correct. Il serait difficile d'affirmer qu'il est préférable de réaliser cela manuellement.

Étape 5 : Modifier et insérer votre requête

Remarquons que le code ci-dessus ne ressemble pas à celui qui est utilisé dans le programme. Ceci est dû au fait que le générateur de requêtes utilise des noms complètement qualifiés, même lorsqu'une seule table est en jeu (lorsqu'on utilise plus d'une table, la qualification empêche les collisions entre colonnes de même nom appartenant à des tables différentes). Puisque cette requête ne concerne qu'une seule table, on peut - optionnellement - supprimer le qualificateur « people » dans la plupart des noms, de cette manière :

```
SELECT FIRST, LAST, EMAIL
FROM people.csv people
WHERE (LAST='Eckel') AND
(EMAIL Is Not Null)
ORDER BY FIRST
```

En outre, on ne va pas coder en dur le nom à rechercher. Au lieu de cela, il sera créé à partir du nom fourni en argument sur la ligne de commande d'appel du programme. Ces changements effectués l'instruction SQL générée dynamiquement dans un objet **String** devient :

```
"SELECT FIRST, LAST, EMAIL " +
"FROM people.csv people " +
"WHERE " +
"(LAST='" + args[0 « + "') " +
" AND (EMAIL Is Not Null) " +
"ORDER BY FIRST");
```

SQL possède un autre mécanisme d'insertion de noms dans une requête, appelé procédures stockées, *stored procedures*, utilisées pour leur vitesse. Mais pour la plupart de vos expérimenta-

tions sur les bases de données et pour votre apprentissage, il est bon de construire vos chaînes de requêtes en Java.

On peut voir à partir de cet exemple que l'utilisation d'outils généralement disponibles et en particulier le générateur de requêtes simplifie la programmation avec SQL et JDBC..

Une version GUI du programme de recherche

Il serait plus pratique de laisser tourner le programme en permanence et de basculer vers lui pour taper un nom et entamer une recherche lorsqu'on en a besoin. Le programme suivant crée le programme de recherche en tant qu'application/applet, et ajoute également une fonctionnalité de complétion des noms afin qu'on puisse voir les données sans être obligé de taper le nom complet :

```
//: c15:jdbc:VLookup.java
// version GUI de Lookup.java.
// <applet code=VLookup
// width=500 height=200></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.event.*;
import java.sql.*;
import com.bruceeckel.swing.*;

public class VLookup extends JApplet {
  String dbUrl = "jdbc:odbc:people";
  String user = "";
  String password = "";
  Statement s;
  JTextField searchFor = new JTextField(20);
  JLabel completion =
    new JLabel("                    ");
  JTextArea results = new JTextArea(40, 20);
  public void init() {
    searchFor.getDocument().addDocumentListener(
      new SearchL());
    JPanel p = new JPanel();
    p.add(new Label("Last name to search for:"));
    p.add(searchFor);
    p.add(completion);
    Container cp = getContentPane();
    cp.add(p, BorderLayout.NORTH);
    cp.add(results, BorderLayout.CENTER);
    try {
      // Charger le driver (qui s'enregistrera lui-même)
      Class.forName(
        "sun.jdbc.odbc.JdbcOdbcDriver");
      Connection c = DriverManager.getConnection(
        dbUrl, user, password);
```

```
      s = c.createStatement();
    } catch(Exception e) {
      results.setText(e.toString());
    }
  }
  class SearchL implements DocumentListener {
    public void changedUpdate(DocumentEvent e){}
    public void insertUpdate(DocumentEvent e){
      textValueChanged();
    }
    public void removeUpdate(DocumentEvent e){
      textValueChanged();
    }
  }
  public void textValueChanged() {
    ResultSet r;
    if(searchFor.getText().length() == 0) {
      completion.setText("");
      results.setText("");
      return;
    }
    try {
      // compléter automatiquement le nom:
      r = s.executeQuery(
        "SELECT LAST FROM people.csv people " +
        "WHERE (LAST Like '" +
        searchFor.getText()  +
        "%') ORDER BY LAST");
      if(r.next())
        completion.setText(
          r.getString("last"));
      r = s.executeQuery(
        "SELECT FIRST, LAST, EMAIL " +
        "FROM people.csv people " +
        "WHERE (LAST='" +
        completion.getText() +
        "') AND (EMAIL Is Not Null) " +
        "ORDER BY FIRST");
    } catch(Exception e) {
      results.setText(
        searchFor.getText() + "\n");
      results.append(e.toString());
      return;
    }
    results.setText("");
    try {
      while(r.next()) {
```

```
            results.append(
              r.getString("Last") + ", "
              + r.getString("fIRST") +
              ": " + r.getString("EMAIL") + "\n");
            }
          } catch(Exception e) {
            results.setText(e.toString());
          }
        }
        public static void main(String[ « args) {
          Console.run(new VLookup(), 500, 200);
        }
      } ///:~
```

La logique « base de données » est en gros la même, mais on peut remarquer qu'un **DocumentListener** est ajouté pour écouter l'entrée **JTextField** (pour plus de détails, voir l'entrée **javax.swing.JTextField** dans la documentation HTML Java sur *java.sun.com*), afin qu'à chaque nouveau caractère frappé le programme tente de compléter le nom en le cherchant dans la base et en utilisant le premier qu'il trouve (en le plaçant dans le **JLabel completion**, et en l'utilisant comme un texte de recherche). De cette manière, on peut arrêter de frapper des caractères dès qu'on en a tapé suffisamment pour que le programme puisse identifier de manière unique le nom qu'on cherche.

Pourquoi l'API JDBC paraît si complexe

Naviguer dans la documentation en ligne de JDBC semble décourageant. En particulier, dans l'interface **DatabaseMetaData** qui est vraiment énorme, à l'inverse de la plupart des interfaces rencontrées jusque là en Java, on trouve des méthodes telles que **dataDefinitionCausesTransactionCommit( )**, **getMaxColumnNameLength( )**, **getMaxStatementLength( )**, **storesMixedCaseQuotedIdentifiers( )**, **supportsANSI92IntermediateSQL( )**, **supportsLimitedOuterJoins( )**, et ainsi de suite. Qu'en est-il de tout cela ?

Ainsi qu'on l'a dit précédemment, les bases de données semblent être depuis leur création dans un constant état de bouleversement, principalement à cause de la très grande demande en applications de base de données, et donc en outils de base de données. Ce n'est que récemment qu'on a vu une certaine convergence vers le langage commun SQL (et il existe beaucoup d'autres langages d'utilisation courante de base de données). Mais même le « standard » SQL possède tellement de variantes que JDBC doit fournir la grande interface **DatabaseMetaData** afin que le code puisse utiliser les possibilités de la base SQL « standard » particulière avec laquelle on est connecté. Bref, il est possible d'écrire du code SQL simple et portable, mais si on veut optimiser la vitesse le code va se multiplier terriblement au fur et à mesure qu'on découvre les possibilités d'une base de données d'un vendeur particulier.

Bien entendu, ce n'est pas la faute de Java. Ce dernier tente seulement de nous aider à compenser les disparités entre les divers produits de base de données . Mais gardons à l'esprit que la vie est plus facile si l'on peut aussi bien écrire des requêtes génériques sans trop se soucier des performances, ou bien, si l'on veut améliorer les performances, connaître la plate-forme pour laquelle on écrit afin de ne pas avoir à traîner trop de code générique.

Un exemple plus sophistiqué

Un exemple plus intéressant [73] est celui d'une base de données multi-tables résidant sur un serveur. Ici, la base sert de dépôt pour les activités d'une communauté et doit permettre aux gens de s'inscrire pour réaliser ces actions, c'est pourquoi on l'appelle une base de données de communauté d'intérêts, *Community Interests Database* (CID). Cet exemple fournira seulement une vue générale de la base et de son implémentation, il n'est en aucun cas un tutoriel complet à propos du développement des bases de données. De nombreux livres, séminaires, et packages de programmes existent pour vous aider dans la conception et le développement des bases de données.

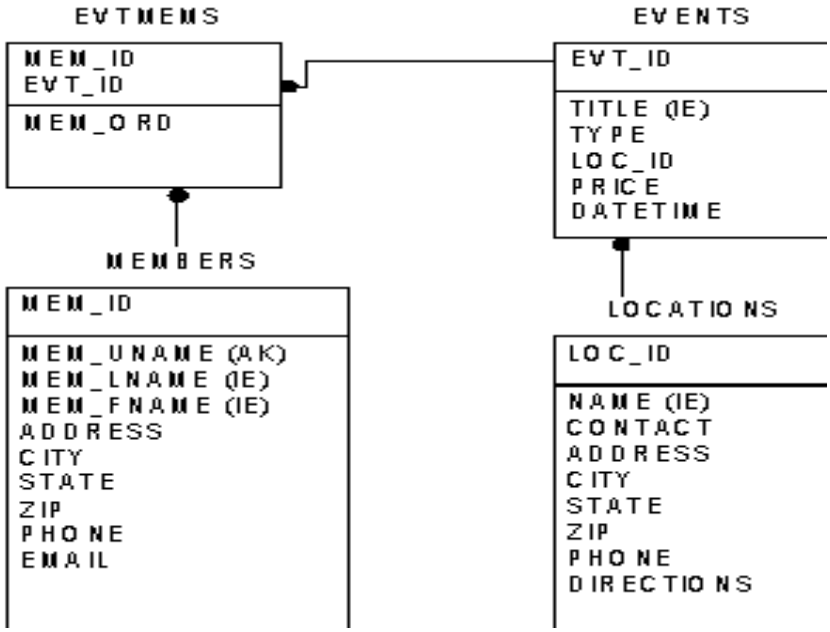
De plus, cet exemple implique l'installation préalable d'une base SQL sur un serveur (bien qu'elle puisse aussi bien tourner sur la machine locale), ainsi que la recherche d'un driver JDBC approprié pour cette base. Il existe plusieurs bases SQL libres, et certaines sont installées automatiquement avec diverses distributions de Linux. Il est de votre responsabilité de faire le choix de la base de données et de localiser son driver JDBC ; cet exemple-ci est basé sur une base SQL nommée « Cloudscape ».

Afin de simplifier les modifications d'information de connexion, le driver de la base, son URL, le nom d'utilisateur et son mot de passe sont placés dans une classe séparée :

```
//: c15:jdbc:CIDConnect.java
// information de connexion à la base de données pour
// la «base de données de communauté d'intérêt» (CID).

public class CIDConnect {
  // Toutes les informations spécifiques à CloudScape:
  public static String dbDriver =
    "COM.cloudscape.core.JDBCDriver";
  public static String dbURL =    "jdbc:cloudscape:d:/docs/_work/JSapienDB";
  public static String user = "";
  public static String password = "";
} ///:~
```

Dans cet exemple, il n'y a pas de protection de la base par mot de passe, le nom d'utilisateur et le mot de passe sont des chaînes vides.

La base de données comprend un ensemble de tables dont voici la structure :



« Members » contient les informations sur les membres de la communauté, « Events » et « Locations » des informations à propos des activités et où on peut les trouver, et « Evtmems » connecte les événements et les membres qui veulent suivre ces événements. On peut constater qu'à une donnée « membre » d'une table correspond une clef dans une autre table.

La classe suivante contient les chaînes SQL qui vont créer les tables de cette base (référez-vous à un guide SQL pour une explication du code SQL) :

```
//: c15:jdbc:CIDSQL.java
// chaînes SQL créant les tables pour la CID.

public class CIDSQL {
  public static String[ « sql = {
    // Créer la table MEMBERS:
    "drop table MEMBERS",
    "create table MEMBERS " +
    "(MEM_ID INTEGER primary key, " +
    "MEM_UNAME VARCHAR(12) not null unique, "+
    "MEM_LNAME VARCHAR(40), " +
    "MEM_FNAME VARCHAR(20), " +
    "ADDRESS VARCHAR(40), " +
    "CITY VARCHAR(20), " +
    "STATE CHAR(4), " +
    "ZIP CHAR(5), " +
    "PHONE CHAR(12), " +
    "EMAIL VARCHAR(30))",
    "create unique index " +
    "LNAME_IDX on MEMBERS(MEM_LNAME)",
    // Créer la table EVENTS:
    "drop table EVENTS",
```

```
    "create table EVENTS " +
    "(EVT_ID INTEGER primary key, " +
    "EVT_TITLE VARCHAR(30) not null, " +
    "EVT_TYPE VARCHAR(20), " +
    "LOC_ID INTEGER, " +
    "PRICE DECIMAL, " +
    "DATETIME TIMESTAMP)",
    "create unique index " +
    "TITLE_IDX on EVENTS(EVT_TITLE)",
    // Créer la table EVTMEMS:
    "drop table EVTMEMS",
    "create table EVTMEMS " +
    "(MEM_ID INTEGER not null, " +
    "EVT_ID INTEGER not null, " +
    "MEM_ORD INTEGER)",
    "create unique index " +
    "EVTMEM_IDX on EVTMEMS(MEM_ID, EVT_ID)",
    // Créer la table LOCATIONS:
    "drop table LOCATIONS",
    "create table LOCATIONS " +
    "(LOC_ID INTEGER primary key, " +
    "LOC_NAME VARCHAR(30) not null, " +
    "CONTACT VARCHAR(50), " +
    "ADDRESS VARCHAR(40), " +
    "CITY VARCHAR(20), " +
    "STATE VARCHAR(4), " +
    "ZIP VARCHAR(5), " +
    "PHONE CHAR(12), " +
    "DIRECTIONS VARCHAR(4096))",
    "create unique index " +
    "NAME_IDX on LOCATIONS(LOC_NAME)",
  };
} ///:~
```

Dans cet exemple, il est intéressant de laisser les exceptions s'afficher sur la console :

```
//: c15:jdbc:CIDCreateTables.java
// Crée les tables d'une base de données pour la
// «community interests database».
import java.sql.*;

public class CIDCreateTables {
  public static void main(String[ « args)
  throws SQLException, ClassNotFoundException,
  IllegalAccessException {
    // Charger le driver (qui s'enregistrera lui-même)
    Class.forName(CIDConnect.dbDriver);
    Connection c = DriverManager.getConnection(
```

```
        CIDConnect.dbURL, CIDConnect.user,
        CIDConnect.password);
      Statement s = c.createStatement();
      for(int i = 0; i < CIDSQL.sql.length; i++) {
        System.out.println(CIDSQL.sql[i]);
        try {
          s.executeUpdate(CIDSQL.sql[i]);
        } catch(SQLException sqlEx) {
          System.err.println(
            "Probably a 'drop table' failed");
        }
      }
      s.close();
      c.close();
    }
  } ///:~
```

Remarquons que les modifications de la base peuvent être contrôlées en changeant **String**s dans la table **CIDSQL**, sans modifier **CIDCreateTables**.

La méthode **executeUpdate( )** renvoie généralement le nombre d'enregistrements affectés par l'instruction SQL. Elle est très souvent utilisée pour exécuter des instructions INSERT, UPDATE, ou DELETEmodifiant une ou plusieurs lignes. Pour les instructions telles que CREATE TABLE, DROP TABLE, et CREATE INDEX, **executeUpdate( )**renvoie toujours zéro.

Pour tester la base, celle-ci est chargée avec quelques données exemples. Ceci est réalisé au moyen d'une série d'INSERTface="Georgia">suivie d'un SELECTafin de produire le jeu de données. Pour effectuer facilement des additions et des modifications aux données de test, ce dernier est construit comme un tableau d'**Object**à deux dimensions, et la méthode **executeInsert( )** peut alors utiliser l'information d'une ligne de la table pour construire la commande SQL appropriée.

```
//: c15:jdbc:LoadDB.java
// Charge et teste la base de données.
import java.sql.*;

class TestSet {
  Object[][ « data = {
    { "MEMBERS", new Integer(1),
      "dbartlett", "Bartlett", "David",
      "123 Mockingbird Lane",
      "Gettysburg", "PA", "19312",
      "123.456.7890",  "bart@you.net" },
    { "MEMBERS", new Integer(2),
      "beckel", "Eckel", "Bruce",
      "123 Over Rainbow Lane",
      "Crested Butte", "CO", "81224",
      "123.456.7890", "beckel@you.net" },
    { "MEMBERS", new Integer(3),
      "rcastaneda", "Castaneda", "Robert",
```

```
    "123 Downunder Lane",
    "Sydney", "NSW", "12345",
    "123.456.7890", "rcastaneda@you.net" },
  { "LOCATIONS", new Integer(1),
    "Center for Arts",
    "Betty Wright", "123 Elk Ave.",
    "Crested Butte", "CO", "81224",
    "123.456.7890",
    "Go this way then that." },
  { "LOCATIONS", new Integer(2),
    "Witts End Conference Center",
    "John Wittig", "123 Music Drive",
    "Zoneville", "PA", "19123",
    "123.456.7890",
    "Go that way then this." },
  { "EVENTS", new Integer(1),
    "Project Management Myths",
    "Software Development",
    new Integer(1), new Float(2.50),
    "2000-07-17 19:30:00" },
  { "EVENTS", new Integer(2),
    "Life of the Crested Dog",
    "Archeology",
    new Integer(2), new Float(0.00),
    "2000-07-19 19:00:00" },
  // Met en relation personnes et événements
  {  "EVTMEMS",
    new Integer(1),  // Dave est mis en relation avec
    new Integer(1),  // l'événement Software.
    new Integer(0) },
  { "EVTMEMS",
    new Integer(2),  // Bruce est mis en relation avec
    new Integer(2),  // l'événement Archeology.
    new Integer(0) },
  { "EVTMEMS",
    new Integer(3),  // Robert est mis en relation avec
    new Integer(1),  // l'événement Software...
    new Integer(1) },
  { "EVTMEMS",
    new Integer(3), // ... et
    new Integer(2), // l'événement Archeology.
    new Integer(1) },
};
// Utiliser les données par défaut:
public TestSet() {}
// Utiliser un autre ensemble de données:
public TestSet(Object[][ « dat) { data = dat; }
```

```
}

public class LoadDB {
  Statement statement;
  Connection connection;
  TestSet tset;
  public LoadDB(TestSet t) throws SQLException {
    tset = t;
    try {
      // Charger le driver (qui s'enregistrera lui-même)
      Class.forName(CIDConnect.dbDriver);
    } catch(java.lang.ClassNotFoundException e) {
      e.printStackTrace(System.err);
    }
    connection = DriverManager.getConnection(
      CIDConnect.dbURL, CIDConnect.user,
      CIDConnect.password);
    statement = connection.createStatement();
  }
  public void cleanup() throws SQLException {
    statement.close();
    connection.close();
  }
  public void executeInsert(Object[ « data) {
    String sql = "insert into " + data[0] « + " values(";
    for(int i = 1; i < data.length; i++) {
      if(data[i « instanceof String)
        sql += "'" + data[i] « + "'";
      else
        sql += data[i];
      if(i < data.length - 1)
        sql += ", ";
    }
    sql += ')';
    System.out.println(sql);
    try {
      statement.executeUpdate(sql);
    } catch(SQLException sqlEx) {
      System.err.println("Insert failed.");
      while (sqlEx != null) {
        System.err.println(sqlEx.toString());
        sqlEx = sqlEx.getNextException();
      }
    }
  }
  public void load() {
    for(int i = 0; i< tset.data.length; i++)
```

```
      executeInsert(tset.data[i]);
    }
    // Lever l'exception en l'envoyant vers la console:
    public static void main(String[ « args)
    throws SQLException {
      LoadDB db = new LoadDB(new TestSet());
      db.load();
      try {
        // Obtenir un ResultSet de la base chargée:
        ResultSet rs = db.statement.executeQuery(
          "select " +
          "e.EVT_TITLE, m.MEM_LNAME, m.MEM_FNAME "+
          "from EVENTS e, MEMBERS m, EVTMEMS em " +
          "where em.EVT_ID = 2 " +
          "and e.EVT_ID = em.EVT_ID " +
          "and m.MEM_ID = em.MEM_ID");
        while (rs.next())
          System.out.println(
            rs.getString(1) + "  " +
            rs.getString(2) + ", " +
            rs.getString(3));
      } finally {
        db.cleanup();
      }
    }
  } ///:~
```

La classe **TestSet** contient un ensemble de données par défaut qui est mis en oeuvre lorsqu'on appelle le constructeur par défaut ; toutefois, il est possible de créer au moyen du deuxième constructeur un objet **TestSet** utilisant un deuxième ensemble de données. L'ensemble de données est contenu dans un tableau à deux dimensions de type **Object** car il peut contenir n'importe quel type, y compris **String** ou des types numériques. La méthode **executeInsert( )** utilise RTTI pour différencier les données **String** (qui doivent être entre guillemets) et les données non-**String** en construisant la commande SQL à partir des données. Après avoir affiché cette commande sur la console, **executeUpdate( )** l'envoie à la base de données.

Le constructeur de **LoadDB** établit la connexion, et **load( )**parcourt les données en appelant **executeInsert( )**pour chaque enregistrement. **Cleanup( )**termine l'instruction et la connexion ; tout ceci est placé dans une clause finally afin d'en garantir l'appel.

Une fois la base chargée, une instruction **executeQuery( )**produit un ensemble résultat. La requête concernant plusieurs tables, nous avons bien un exemple de base de données relationnelle.

On trouvera d'autres informations sur JDBC dans les documents électroniques livrés avec la distribution Java de Sun. Pour en savoir plus, consulter le livre *JDBC Database Access with Java* (Hamilton, Cattel, and Fisher, Addison-Wesley, 1997). D'autres livres à propos de JDBC sortent régulièrement.

Les Servlets

Les accès clients sur l'Internet ou les intranets d'entreprise représentent un moyen sûr de permettre à beaucoup d'utilisateurs d'accéder facilement aux données et ressources [74]. Ce type d'accès est basé sur des clients utilisant les standards du World Wide Web Hypertext Markup Language (HTML) et Hypertext Transfer Protocol (HTTP). L'API Servlet fournit une abstraction pour un ensemble de solutions communes en réponse aux requêtes HTTP.

Traditionnellement, la solution permettant à un client Internet de mettre à jour une base de données est de créer une page HTML contenant des champs texte et un bouton « soumission ». L'utilisateur frappe l'information requise dans les champs texte puis clique sur le bouton « soumission ». Les données sont alors soumises au moyen d'une URL qui indique au serveur ce qu'il doit en faire en lui indiquant l'emplacement d'un programme Common Gateway Interface (CGI) lancé par le serveur, qui prend ces données en argument. Le programme CGI est généralement écrit en Perl, Python, C, C++, ou n'importe quel langage capable de lire sur l'entrée standard et d'écrire sur la sortie standard. Le rôle du serveur Web s'arrête là : le programme CGI est appelé, et des flux standard (ou, optionnellement pour l'entrée, une variable d'environnement) sont utilisés pour l'entrée et la sortie. Le programme CGI est responsable de toute la suite. Il commence par examiner les données et voir si leur format est correct. Si ce n'est pas le cas, le programme CGI doit fournir une page HTML décrivant le problème ; cette page est prise en compte par le serveur Web (via la sortie standard du programme CGI), qui la renvoie à l'utilisateur. Habituellement, l'utilisateur revient à la page précédente et fait une nouvelle tentative. Si les données sont correctes, le programme CGI traite les données de la manière appropriée, par exemple en les ajoutant à une base de données. Il élabore ensuite une page HTML appropriée que le serveur Web enverra à l'utilisateur.

Afin d'avoir une solution basée entièrement sur Java, l'idéal serait d'avoir côté client une applet qui validerait et enverrait les données, et côté serveur une servlet qui les recevrait et les traiterait. Malheureusement, bien que les applets forment une technologie éprouvée et bien supportée, leur utilisation sur le Web s'est révélée problématique car on ne peut être certain de la disponibilité d'une version particulière de Java sur le navigateur Web du client ; en fait, on ne peut même pas être certain que le navigateur Web supporte Java ! Dans un intranet, on peut exiger qu'un support donné soit disponible, ce qui apporte une certaine flexibilité à ce qu'on peut faire, mais sur le Web l'approche la plus sûre est d'effectuer tout le traitement du côté serveur puis de délivrer une page HTML au client. De cette manière, aucun client ne se verra refuser l'utilisation de votre site simplement parce qu'il ne dispose pas dans sa configuration du software approprié.

Parce que les servlets fournissent une excellente solution pour le support de programmation côté serveur, ils représentent l'une des raisons les plus populaires pour passer à Java. Non seulement ils fournissent un cadre pour remplacer la programmation CGI (et éliminer nombre de problèmes CGI épineux), mais tout le code gagne en portabilité inter plate-forme en utilisant Java, et l'on a accès à toutes les API Java (exceptées, bien entendu, celles qui fournissent des GUI, comme Swing).

Le servlet de base

L'architecture de l'API servlet est celle d'un fournisseur de services classique comportant une méthode **service( )** appartenant au software conteneur de la servlet, chargée de recevoir toutes les requêtes client, et les méthodes liées au cycle de vie, **init( )**et **destroy( )**, qui sont appelées seulement lorsque la servlet est chargée ou déchargée (ce qui arrive rarement).

```
public interface Servlet {
```

```
  public void init(ServletConfig config)
    throws ServletException;
  public ServletConfig getServletConfig();
  public void service(ServletRequest req,
    ServletResponse res)
    throws ServletException, IOException;
  public String getServletInfo();
  public void destroy();
}
```

La raison d'être de **getServletConfig( )**est de renvoyer un objet **ServletConfig** contenant l'initialisation et les paramètres de départ de cette servlet. La méthode **getServletInfo( )**renvoie une chaîne contenant des informations à propos de la servlet, telles que le nom de l'auteur, la version, et le copyright.

La classe **GenericServlet** est une implémentation de cette interface et n'est généralement pas utilisée. La classe **HttpServlet** est une extension de **GenericServlet**, elle est explicitement conçue pour traiter le protocole HTTP. C'est cette classe, **HttpServlet**, que vous utiliserez la plupart du temps.

Les attributs les plus commodes de l'API servlet sont les objets auxiliaires fournis par la classe HttpServlet. En regardant la méthode **service( )**de l'interface **Servlet**, on constate qu'elle a deux paramètres : **ServletRequest** et **ServletResponse**. Dans la classe **HttpServlet**, deux objets sont développés pour HTTP : **HttpServletRequest** and **HttpServletResponse**. Voici un exemple simple montrant l'utilisation de **HttpServletResponse** :

```
//: c15:servlets:ServletsRule.java
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class ServletsRule extends HttpServlet {
  int i = 0; // «persistance» de Servlet
  public void service(HttpServletRequest req,
  HttpServletResponse res) throws IOException {
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();
    out.print("<HEAD><TITLE>");
    out.print("A server-side strategy");
    out.print("</TITLE></HEAD><BODY>");
    out.print("<h1>Servlets Rule! " + i++);
    out.print("</h1></BODY>");
    out.close();
  }
} ///:~
```

La classe **ServletsRule** est la chose la plus simple que peut recevoir une servlet. La servlet est initialisée au démarrage en appelant sa méthode **init( )**, en chargeant la servlet après que le conteneur de la servlet soit chargé. Lorsqu'un client envoie une requête à une URL qui semble reliée

à une servlet, le conteneur de servlet intercepte cette demande et effectue un appel de la méthode **service( )**, après avoir créé les objets **HttpServletRequest** et **HttpServletResponse**.

La principale responsabilité de la méthode **service( )**est d'interagir avec la requête HTTP envoyée par le client, et de construire une réponse HTTP basée sur les attributs contenus dans la demande. La méthode **ServletsRule** se contente de manipuler l'objet réponse sans chercher à savoir ce que voulait le client.

Après avoir mis en place le type du contenu de la réponse (ce qui doit toujours être fait avant d'initialiser **Writer** ou **OutputStream**), la méthode **getWriter( )** de l'objet réponse renvoie un objet **PrintWriter**, utilisé pour écrire les données en retour sous forme de caractères (de manière similaire, **getOutputStream( )**fournit un **OutputStream**, utilisé pour les réponses binaires, uniquement dans des solutions plus spécifiques).

Le reste du programme se contente d'envoyer une page HTML au client (on suppose que le lecteur comprend le langage HTML, qui n'est pas décrit ici) sous la forme d'une séquence de **String**s. Toutefois, il faut remarquer l'inclusion du « compteur de passages » représenté par la variable **i**. Il est automatiquement converti en **String** dans l'instruction **print( )**.

En lançant le programme, on peut remarquer que la valeur de **i** ne change pas entre les requêtes vers la servlet. C'est une propriété essentielle des servlets : tant qu'il n'existe qu'une servlet d'une classe particulière chargée dans le conteneur, et jamais déchargée (sauf en cas de fin du conteneur de servlet, ce qui ne se produit normalement que si l'on reboote l'ordinateur serveur), tous les champs de cette classe servlet sont des objets persistants ! Cela signifie que vous pouvez sans effort supplémentaire garder des valeurs entre les requêtes à la servlet, alors qu'avec CGI vous auriez dû écrire ces valeurs sur disque afin de les préserver, ce qui aurait demandé du temps supplémentaire et fini par déboucher sur une solution qui n'aurait pas été inter-plate-forme.

Bien entendu, le serveur Web ainsi que le conteneur de servlet doivent de temps en temps être rebootés pour des raisons de maintenance ou après une coupure de courant. Pour éviter de perdre toute information persistante, les méthodes de servlet **init( )** et **destroy( )** sont appelées automatiquement chaque fois que la servlet est chargée ou déchargée, ce qui nous donne l'opportunité de sauver des données lors d'un arrêt, puis de les restaurer après que la machine ait été rebootée. Le conteneur de la servlet appelle la méthode **destroy( )** lorsqu'il se termine lui-même, et on a donc toujours une opportunité de sauver des données essentielles pour peu que la machine serveur soit intelligemment configurée.

Quand un formulaire est soumis à un servlet, **HttpServletRequest** est préchargée avec les données du formulaire présentées sous la forme de paires clef/valeur. Si on connaît le nom des champs, il suffit d'y accéder directement avec la méthode **getParameter( )** pour connaître leur valeur. Il est également possible d'obtenir un objet **Enumeration** (l'ancienne forme d'un **Iterator**) vers les noms des champs, ainsi que le montre l'exemple qui suit. Cet exemple montre aussi comment un seul servlet peut être utilisé pour produire à la fois la page contenant le formulaire et la réponse à cette page (on verra plus tard une meilleure solution utilisant les JSP). Si **Enumeration** est vide, c'est qu'il n'y a plus de champs ; cela signifie qu'aucun formulaire n'a été soumis. Dans ce cas, le formulaire est élaboré, et le bouton de soumission rappellera la même servlet. Toutefois les champs sont affichés lorsqu'ils existent.

```
//: c15:servlets:EchoForm.java
// Affiche les couples nom-valeur d'un formulaire HTML
import javax.servlet.*;
import javax.servlet.http.*;
```

```
import java.io.*;
import java.util.*;

public class EchoForm extends HttpServlet {
  public void service(HttpServletRequest req,
    HttpServletResponse res) throws IOException {
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();
    Enumeration flds = req.getParameterNames();
    if(!flds.hasMoreElements()) {
      // Pas de formulaire soumis -- on en crée un:
      out.print("<html>");
      out.print("<form method=\"POST\"" +
        " action=\"EchoForm\">");
      for(int i = 0; i < 10; i++)
        out.print("<b>Field" + i + "</b> " +
          "<input type=\"text\""+
          " size=\"20\" name=\"Field" + i +
          "\" value=\"Value" + i + "\"><br>");
      out.print("<INPUT TYPE=submit name=submit"+
      " Value=\"Submit\"></form></html>");
    } else {
      out.print("<h1>Your form contained:</h1>");
      while(flds.hasMoreElements()) {
        String field= (String)flds.nextElement();
        String value= req.getParameter(field);
        out.print(field + " = " + value+ "<br>");
      }
    }
    out.close();
  }
} ///:~
```

On peut penser en lisant cela que Java ne semble pas conçu pour traiter des chaînes de caractères car le formatage des pages à renvoyer est pénible à cause des retours à la ligne, des séquences escape, et du signe + inévitable dans la construction des objets **String**. Il n'est pas raisonnable de coder une page HTML quelque peu conséquente en Java. Une des solutions est de préparer la page en tant que fichier texte séparé, puis de l'ouvrir et de la passer au serveur Web. S'il fallait de plus effectuer des substitutions de chaînes dans le contenu de la page, ce n'est guère mieux car le traitement des chaînes en Java est très pauvre. Si vous rencontrez un de ces cas, il serait préférable d'adopter une solution mieux appropriée (mon choix irait vers Python ; voici une version incluse dans un programme Java appelé JPython) qui génère une page-réponse.

Les Servlets et le multithreading

Le conteneur de servlet dispose d'un ensemble de threads qu'il peut lancer pour traiter les demandes des clients. On peut imaginer cela comme si deux clients arrivant au même moment étaient traités simultanément par la méthode **service( )**. En conséquence la méthode **service( )** doit être

écrite d'une manière sécurisée dans un contexte de thread. Tous les accès aux ressouces communes (fichiers, bases de données) demandent à être protégés par le mot clef **synchronized**.

L'exemple très simple qui suit utilise une clause **synchronized** autour de la méthode **sleep( )**du thread. En conséquence les autres threads seront bloqués jusqu'à ce que le temps imparti (cinq secondes) soit écoulé. Pour tester cela il faut lancer plusieurs instances d'un navigateur puis lancer ce servlet aussi vite que possible ; remarquez alors que chacun d'eux doit attendre avant de voir le jour.

```
//: c15:servlets:ThreadServlet.java
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class ThreadServlet extends HttpServlet {
  int i;
  public void service(HttpServletRequest req,
    HttpServletResponse res) throws IOException {
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();
    synchronized(this) {
      try {
        Thread.currentThread().sleep(5000);
      } catch(InterruptedException e) {
        System.err.println("Interrupted");
      }
    }
    out.print("<h1>Finished " + i++ + "</h1>");
    out.close();
  }
} ///:~
```

On peut aussi synchroniser complètement la servlet en mettant le mot clef **synchronized** juste avant la méthode **service( )**. En réalité, l'unique justification pour utiliser la clause **synchronized** à la place de cela est lorsque la section critique se trouve dans un chemin d'exécution qui ne doit pas être exécuté. Dans un tel cas, il serait préférable d'éviter la contrainte de synchronisation à chaque fois en utilisant une clause **synchronized**. Sinon, chaque thread particulier devrait systématiquement attendre, il vaut donc mieux synchroniser la méthode en entier.

Gérer des sessions avec les servlets

HTTP est un protocole qui ne possède pas la notion de session, on ne peut donc savoir d'un appel serveur à un autre s'il s'agit du même appelant ou s'il s'agit d'une personne complètement différente. Beaucoup d'efforts ont été faits pour créer des mécanismes permettant aux développeurs Web de suivre les sessions. À titre d'exemple, les compagnies ne pourraient pas faire de e-commerce si elles ne gardaient pas la trace d'un client, ainsi que les renseignements qu'il a saisi sur sa liste de courses.

Il existe plusieur méthodes pour suivre une session, mais la plus commune utilise les » cookies persistants, « qui font intégralement partie du standard Internet. Le HTTP Working Group de

l'Internet Engineering Task Force a décrit les cookies du standard officiel dans RFC 2109 (*ds.internic.net/rfc/rfc2109.txt* ou voir *www.cookiecentral.com*).

Un cookie n'est pas autre chose qu'une information de petite taille envoyée par un serveur Web à un navigateur. Le navigateur sauvegarde ce cookie sur le disque local, puis lors de chaque appel à l'URL associée au cookie, ce dernier est envoyé de manière transparente en même temps que l'appel, fournissant ainsi au serveur l'information désirée en retour (en lui fournissant généralement d'une certaine manière votre identité). Toutefois les clients peuvent inhiber la capacité de leur navigateur à accepter les cookies. Si votre site doit suivre un client qui a inhibé cette possibilité, alors une autre méthode de suivi de session doit être intégrée à la main (réécriture d'URL ou champs cachés dans un formulaire), car les fonctionnalités de suivi de session intégrées à l'API servlet sont construites autour des cookies.

La classe Cookie

L'API servlet (à partir de la version 2.0) fournit la classe **Cookie**. Cette classe inclut tous les détails de l'en-tête HTTP et permet de définir différents attributs de cookie. L'utilisation d'un cookie consiste simplement à l'ajouter à l'objet réponse. Le constructeur a deux arguments, le premier est un nom du cookie et le deuxième une valeur. Les cookies sont ajoutés à l'objet réponse avant que l'envoi ne soit effectif.

```
Cookie oreo = new Cookie("TIJava", "2000");
res.addCookie(cookie);
```

Les cookies sont récupérés en appelant la méthode **getCookies( )** de l'objet **HttpServletRequest**, qui renvoie un tableau d'objets Cookie.

```
Cookie[ « cookies = req.getCookies();
```

En appelant **getValue( )**pour chaque cookie, on obtient une **String** initialisée avec le contenu du cookie. Dans l'exemple ci-dessus, **getValue("TIJava")**renverrait une String contenant » 2000.]

La classe Session

Une session consiste en une ou plusieurs requêtes de pages adressées par un client à un site Web durant une période définie. Par exemple, si vous faites vos courses en ligne, la session sera la période démarrant au moment où vous ajoutez un achat dans votre panier jusqu'au moment où vous envoyez effectivement la demande. Chaque achat ajouté au panier déclenchera une nouvelle connexion HTTP, qui n'a aucun rapport ni avec les connexions précédentes ni avec les achats déjà inclus dans votre panier. Pour compenser ce manque d'information, les mécanismes fournis par la spécification des cookies permet au servlet de suivre la session.

Un objet servlet **Session** réside du côté serveur sur le canal de communication ; son rôle est de capturer les données utiles à propos du client pendant qu'il navigue sur votre site Web et qu'il interagit avec lui. Ces données peuvent être pertinentes pour la session actuelle, comme les achats dans le panier, ou bien peuvent être des information d'authentification fournies lors de l'accès du client au site Web, et qu'il n'y a pas lieu de donner à nouveau durant un ensemble particulier de transactions.

La classe **Session** de l'API servlet utilise la classe **Cookie** pour effectuer ce travail. Toutefois, l'objet **Session** n'a besoin que d'une sorte d'identifiant unique stocké chez le client et passé au ser-

veur. Les sites Web peuvent aussi utiliser les autres systèmes de suivi de session mais ces mécanismes sont plus difficiles à mettre en oeuvre car ils n'existent pas dans l'API servlet (ce qui signifie qu'on doit les écrire à la main pour traiter le cas où le client n'accepte pas les cookies).

Voici un exemple implémentant le suivi de session au moyen de l'API servlet :

```
//: c15:servlets:SessionPeek.java
// Utilise la classe HttpSession.
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class SessionPeek extends HttpServlet {
  public void service(HttpServletRequest req,
  HttpServletResponse res)
  throws ServletException, IOException {
    // Obtenir l'Objet Session avant tout
    // envoi vers le client.
    HttpSession session = req.getSession();
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();
    out.println("<HEAD><TITLE> SessionPeek ");
    out.println(" </TITLE></HEAD><BODY>");
    out.println("<h1> SessionPeek </h1>");
    // Un simple compteur pour cette session.
    Integer ival = (Integer)
      session.getAttribute("sesspeek.cntr");
    if(ival==null)
      ival = new Integer(1);
    else
      ival = new Integer(ival.intValue() + 1);
    session.setAttribute("sesspeek.cntr", ival);
    out.println("You have hit this page <b>"
      + ival + "</b> times.<p>");
    out.println("<h2>");
    out.println("Saved Session Data </h2>");
    // Boucler au travers de toutes les données de la session:
    Enumeration sesNames =
      session.getAttributeNames();
    while(sesNames.hasMoreElements()) {
      String name =
        sesNames.nextElement().toString();
      Object value = session.getAttribute(name);
      out.println(name + " = " + value + "<br>");
    }
    out.println("<h3> Session Statistics </h3>");
    out.println("Session ID: "
      + session.getId() + "<br>");
```

```
        out.println("New Session: " + session.isNew()
          + "<br>");
        out.println("Creation Time: "
          + session.getCreationTime());
        out.println("<I>(" +
          new Date(session.getCreationTime())
          + ")</I><br>");
        out.println("Last Accessed Time: " +
          session.getLastAccessedTime());
        out.println("<I>(" +
          new Date(session.getLastAccessedTime())
          + ")</I><br>");
        out.println("Session Inactive Interval: "
          + session.getMaxInactiveInterval());
        out.println("Session ID in Request: "
          + req.getRequestedSessionId() + "<br>");
        out.println("Is session id from Cookie: "
          + req.isRequestedSessionIdFromCookie()
          + "<br>");
        out.println("Is session id from URL: "
          + req.isRequestedSessionIdFromURL()
          + "<br>");
        out.println("Is session id valid: "
          + req.isRequestedSessionIdValid()
          + "<br>");
        out.println("</BODY>");
        out.close();
      }
      public String getServletInfo() {
        return "A session tracking servlet";
      }
    } ///:~
```

À l'intérieur de la méthode **service( )**, la méthode **getSession( )** est appelée pour l'objet requête et renvoie un objet **Session** associé à la requête. L'objet **Session** ne voyage pas sur le réseau, il réside sur le serveur et est associé à un client et à ses requêtes.

La méthode **getSession( )** possède deux versions : sans paramètres, ainsi qu'elle est utilisée ici, et **getSession(boolean)**. L'appel de **getSession(true)** est équivalent à **getSession( ).**Le **boolean** sert à indiquer si on désire créer l'objet session lorsqu'on ne le trouve pas. L'appel le plus probable est **getSession(true)**, d'où la forme **getSession( )**.

L'objet **Session**, s'il n'est pas nouveau, nous donne des informations sur le client à partir de visites antérieures. Si l'objet **Session** est nouveau alors le programme commencera à recueillir des informations à propos des activités du client lors de cette visite. Le recueil de cette information est effectué au moyen des méthodes **setAttribute( )** et **getAttribute( )** de l'objet session.

```
    java.lang.Object getAttribute(java.lang.String)
    void setAttribute(java.lang.String name,
```

```
                java.lang.Object value)
```

L'objet **Session** utilise une simple paire nom/valeur pour garder l'information. Le nom est du type **String**, et la valeur peut être n'importe quel objet dérivé de **java.lang.Object**. **SessionPeek** garde la trace du nombre de fois où le client est revenu pendant cette session, au moyen d'un objet **Integer** nommé **sesspeek.cntr**. Si le nom n'existe pas on crée un **Integer** avec une valeur de un, sinon on crée un **Integer** en incrémentant la valeur du précédent. Le nouvel **Integer** est rangé dans l'objet **Session**. Si on utilise la même clef dans un appel à **setAttribute( )**, le nouvel objet écrase l'ancien. Le compteur incrémenté sert à afficher le nombre de visites du client pendant cette session.

La méthode **getAttributeNames( )** est en relation avec **getAttribute( )** et **setAttribute( )** et renvoie une énumération des noms des objets associés à l'objet **Session**. Une boucle **while** de **SessionPeek** montre cette méthode en action.

Vous vous interrogez sans doute sur la durée de vie d'un objet **Session**. La réponse dépend du conteneur de servlet qu'on utilise ; généralement la durée de vie par défaut est de 30 minutes (1800 secondes), ce que l'on peut voir au travers de l'appel de **getMaxInactiveInterval( )** par **ServletPeek**. Les tests semblent montrer des résultats différents suivant le conteneur de servlet utilisé. De temps en temps l'objet **Session** peut faire le tour du cadran, mais je n'ai jamais rencontré de cas où l'objet **Session** disparaît avant que le temps spécifié par « inactive interval » soit écoulé. On peut tester cela en initialisant « inactive interval » à 5 secondes au moyen de **setMaxInactiveInterval( )**puis voir si l'objet **Session** est toujours là ou au contraire a été détruit à l'heure déterminée. Il se pourrait que vous ayez à étudier cet attribut lorsque vous choisirez un conteneur de servlet.

Faire fonctionner les exemples de servlet

Si vous ne travaillez pas encore sur un serveur d'applications gérant les servlets Sun ainsi que les technologies JSP, il vous faudra télécharger l'implémentation Tomcat des servlets Java et des JSP, qui est une implémentation libre et « open-source » des servlets, et de plus l'implémentation officielle de référence de Sun. Elle se trouve à *jakarta.apache.org*.

Suivez les instructions d'installation de l'implementation Tomcat, puis éditez le fichier **server.xml** pour décrire l'emplacement de votre répertoire qui contiendra vos servlets. Une fois lancé le programme Tomcat vous pouvez tester vos programmes servlet.

Ceci n'était qu'une brève introduction aux servlets ; il existe des livres entiers traitant de ce sujet. Toutefois, cette introduction devrait vous donner suffisamment d'idées pour démarrer. De plus, beaucoup de thèmes développés dans la section suivante ont une compatibilité ascendante avec les servlets.

Les Pages Java Serveur - Java Server Pages

Les Java Server Pages (JSP) sont une extension standard Java définie au-dessus des extensions servlet. Le propos des JSP est de simplifier la création et la gestion des pages Web dynamiques.

L'implémentation de référence Tomcat, déjà mentionnée et disponible librement sur *jakarta.apache.org* "font-style: normal">, supporte automatiquement les JSP.

Les JSP permettent de mélanger le code HTML d'une page Web et du code Java dans le même document. Le code Java est entouré de tags spéciaux qui indiquent au conteneur JSP qu'il

doit utiliser le code pour générer une servlet complètement ou en partie. L'avantage que procurent les JSP est de maintenir un seul document qui est à la fois la page HTML et le code Java qui la gère. Le revers est que celui qui maintient la page JSP doit être autant qualifié en HTML qu'en Java (toutefois, des environnements GUI de construction de JSP devraient apparaître sur le marché).

À partir de là, et tant que le code source JSP de la page n'est pas modifié, tout se passe comme si on avait une page HTML statique associée à des servlets (en réalité, le code HTML est généré par la servlet). Si on modifie le code source de la JSP, il est automatiquement recompilé et rechargé dès que cette page sera redemandée. Bien entendu, à cause de ce dynamisme le temps de réponse sera long lors du premier accès à une JSP. Toutefois, étant donné qu'une JSP est généralement plus utilisée qu'elle n'est modifiée, on ne sera pas généralement affecté par ce délai.

La structure d'une page JSP est à mi-chemin d'une servlet et d'une page HTML. Les tags JSP commencent et finissent comme les tags HTML, sauf qu'ils utilisent également le caractère pourcent (%), ainsi tous les tags JSP ont cette structure :

```
<% ici, le code JSP %>
```

Le premier caractère pourcent doit être suivi d'un autre caractères qui définit précisément le type du code JSP du tag.

Voici un un exemple extrêmement simple de JSP utilisant un appel standard à une bibliothèque Java pour récupérer l'heure courante en millisecondes, et diviser le résultat par 1000 pour produire l'heure en secondes. Une *expression JSP* (**<%=**) est utilisée, puis le résultat du calcul est mis dans une **String** et intégré à la page Web générée :

```
//:! c15:jsp:ShowSeconds.jsp
<html><body>
<H1>The time in seconds is:
<%= System.currentTimeMillis()/1000 %></H1>
</body></html>
///:~
```

Dans les exemples JSP de ce livre, la première et la dernière ligne ne font pas partie du fichier code réel qui est extrait et placé dans l'arborescence du code source de ce livre.

Lorsque le client demande une page JSP, le serveur Web doit avoir été configuré pour relayer la demande vers le conteneur JSP, qui à son tour appelle la page. Comme on l'a déjà dit plus haut, lors du premier appel de la page, les composants spécifiés par la page sont générés et compilés par le conteneur JSP en tant qu'une ou plusieurs servlets. Dans les exemples ci-dessus, la servlet doit contenir le code destiné à configurer l'objet **HttpServletResponse**, produire un objet **PrintWriter** (toujours nommé **out**), et enfin transformer le résultat du calcul en un objet **String** qui sera envoyé vers **out**. Ainsi qu'on peut le voir, tout ceci est réalisé au travers d'une instruction très succincte, mais en moyenne les programmeurs HTML/concepteurs de site Web ne seront pas qualifiés pour écrire un tel code.

Les objets implicites

Les servlets comportent des classes fournissant des utilitaires pratiques, comme **HttpServletRequest**, **HttpServletResponse**, **Session**, etc. Les objets de ces classes font partie de la spécification JSP et sont automatiquement disponibles pour vos JSP sans avoir à écrire une ligne de code supplémentaire. Les objets implicites d'une JSP sont décrits dans le tableau ci-dessous.

Variable implicite	"text-decoration: none">Du Type (javax.servlet)	Description	Visibilité
demande (request)	Sous-type de **HttpServletRequest** dépendant du protocole	La demande qui déclenche l'appel du service.	demande
réponse	Sous-type de **HttpServletResponse** dépendant du protocole	La réponse à la demande.	page
pageContext	**jsp.PageContext**	Le contexte de page encapsule les choses qui dépendent de l'implémentation et fournissent des méthodes commodes ainsi que l'accès à un espace de nommage pour ce JSP.	page
session	Sous-type de **http.HttpSession** dépendant du protocole	L'objet session créé pour le client demandeur. Voir l'objet Session pour les servlets.	session
application	**ServletContext**	Le contexte de servlet obtenu depuis l'objet configuration de servlet (e.g., **getServletConfig()**, **getContext( )**.	app
out	**jsp.JspWriter**	L'objet qui écrit dans le flux sortant.	page
config	**ServletConfig**	Le **ServletConfig** pour ce JSP.	page
page	**java.lang.Object**	L'instance de cette classe d'implémentation de page.s gérant la demande courante.	page

La visibilité de chaque objet est extrêmement variable. Par exemple, l'objet **session** a une visibilité qui dépasse la page, car il englobe plusieurs demandes client et plusieurs pages. L'objet **application** peut fournir des services à un groupe de pages JSP représentant une application Web.

Les directives JSP

Les directives sont des messages adressés au conteneur de JSP et sont reconnaissables au caractère » **@**] :

```
<%@ directive {attr="value"}* %>
```

Les directives n'envoient rien sur le flux **out**, mais sont importantes lorsqu'on définit les attributs et les dépendances de pages avec le conteneur JSP. Par exemple, la ligne :

```
<%@ page language="java" %>
```

exprime le fait que le langage de scripting utilisé dans la page JSP est Java. En fait, la spécification JSP décrit *seulement* "font-style: normal">la sémantique des scripts pour un atttribut de langage égal à [Java. « La raison d'être de cette directive est d'introduire la flexibilité dans la technologie JSP. Dans le futur, si vous aviez à choisir un autre langage, par exemple Python (un bon choix de langage de sripting), alors ce langage devra supporter le Java Run-time Environment en exposant la technologie du modèle objet Java à l'environnement de scripting, en particulier les variables implicites définies plus haut, les propriétés JavaBeans, et les méthodes publiques.

La directive la plus importante est la directive de page. Elle définit un certain nombre d'attributs dépendant de la page et les communique au conteneur JSP. Parmi ces attributs : **language**, **extends**, **import**, **session**, **buffer**, **autoFlush**, **isThreadSafe**, **info** et **errorPage**. Par exemple :

```
<%@ page session=]true « import=]java.util.* « %>
```

Cette ligne indique tout d'abord que la page nécessite une participation à une session HTTP. Puisque nous n'avons pas décrit de directive de langage le conteneur JSP utilisera Java par défaut et la variable de langage de scripting implicite appelée session sera du type **javax.servlet.http.HttpSession**. Si la directive avait été false alors la variable implicite **session** n'aurait pas été disponible. Si la variable **session** n'est pas spécifiée, sa valeur par défaut est » true.]

L'attribut **import** décrit les types disponibles pour l'environnement de scripting. Cet attribut est utilisé tout comme il le serait dans le langage de programmation Java, c'est à dire une liste d'expressions **import** ordinaires séparées par des virgules. Cette liste est importée par l'implémentation de la page JSP traduite et reste disponible pour l'environnement de scripting. À nouveau, ceci est actuellement défini uniquement lorsque la valeur de la directive de langage est » java.]

Les éléments de scripting JSP

Une fois l'environnement de scripting mis en place au moyen des directives on peut utiliser les éléments du langage de scripting. JSP 1.1 possède trois éléments de langage de scripting *declaration*, *scriptlet*, et *expression*. Une déclaration déclare des éléments, un scriptlet est un fragment d'instruction, et une expression est une expression complète du langage. En JSP chaque élément de scripting commence par » **<%**]. Voici la syntaxe de chacun :

```
<%! declaration %>
<%  scriptlet   %>
<%= expression  %>
```

L'espace est facultatif après » <%!], » <%], » <%=], et avant [%>.]

Tous ces tags s'appuient sur la norme XML ; on pourrait dire qu'une page JSP est un document XML. La syntaxe équivalente en XML pour les éléments de scripting ci-dessus serait :

```
<jsp:declaration> declaration </jsp:declaration>
<jsp:scriptlet>   scriptlet   </jsp:scriptlet>
<jsp:expression>  expression  </jsp:expression>
```

De plus, il existe deux types de commentaires :

```
<%-- jsp comment --%>
<!-- html comment -->
```

La première forme crée dans les pages sources JSP des commentaires qui n'apparaîtront jamais dans la page HTML envoyée au client. Naturellement, la deuxième forme n'est pas spécifique à JSP, c'est un commentaire HTML ordinaire. Ceci est intéressant car on peut insérer du code JSP dans un commentaire HTML, le commentaire étant inclus dans la page résultante ainsi que le résultat du code JSP.

Les déclarations servent à déclarer des variables et des méthodes dans les langages de scripting utilisés dans une page JSP (uniquement Java pour le moment). La déclaration doit être une instruction Java complète et ne doit pas écrire dans le flux **out**. Dans l'exemple ci-dessous **Hello.jsp**, les déclarations des variables **loadTime**, **loadDate** et **hitCount** sont toutes des instructions Java complètes qui déclarent et initialisent de nouvelle variables.

```
//:! c15:jsp:Hello.jsp
<%-- This JSP comment will not appear in the
generated html --%>
<%-- This is a JSP directive: --%>
<%@ page import="java.util.*" %>
<%-- These are declarations: --%>
<%!
   long loadTime= System.currentTimeMillis();
   Date loadDate = new Date();
   int hitCount = 0;
%>
<html><body>
<%-- The next several lines are the result of a
JSP expression inserted in the generated html;
the '=' indicates a JSP expression --%>
<H1>This page was loaded at <%= loadDate %> </H1>
<H1>Hello, world! It's <%= new Date() %></H1>
<H2>Here's an object: <%= new Object() %></H2>
<H2>This page has been up
<%= (System.currentTimeMillis()-loadTime)/1000 %>
seconds</H2>
<H3>Page has been accessed <%= ++hitCount %>
times since <%= loadDate %></H3>
<%-- A "scriptlet" that writes to the server
console and to the client page.
Note that the ';' is required: --%>
```

```
<%
  System.out.println("Goodbye");
  out.println("Cheerio");
%>
</body></html>
///:~
```

Lorsque ce programme fonctionne, on constate que les variables **loadTime**, **loadDate** et **hitCount** gardent leurs valeurs entre les appels de la page, il s'agit donc clairement de champs et non de variables locales.

À la fin de l'exemple un scriplet écrit « Goodbye « sur la console du serveur Web et » Cheerio « sur l'objet **JspWriter**implicite **out**. Les scriptlets peuvent contenir tout fragment de code composé d'instructions Java valides. Les scriptlets sont exécutés au moment du traitement de la demande. Lorsque tous les fragments de scriptlet d'une page JSP donnée sont combinés dans l'ordre où ils apparaissent dans la page JSP, ils doivent contenir une instruction valide telle que définie dans le langage de programmation Java. Le fait qu'ils écrivent ou pas dans le flux **out** dépend du code du scriplet. Il faut garder à l'esprit que les scriplets peuvent produire des effets de bord en modifiant les objets se trouvant dans leur champ de visibilité.

Les expressions JSP sont mêlées au code HTML dans la section médiane de **Hello.jsp**. Les expressions doivent être des instructions Java complètes, qui sont évaluées, traduites en **String**, et envoyées à **out**. Si le résultat d'une expression ne peut pas être traduit en **String** alors une exception **ClassCastException** est lancée.

Extraire des champs et des valeurs

L'exemple suivant ressemble à un autre vu précédemment dans la section servlet. La première fois que la page est appelée il détecte s'il n'existe pas de champ et renvoie une page contenant un formulaire, en utilisant le même code que dans l'exemple de la servlet, mais au format JSP. Lorsque le formulaire contenant des champs remplis est envoyé à la même URL JSP, il détecte les champs et les affiche. C'est une technique agréable parce qu'elle permet d'avoir dans un seul fichier, le formulaire destiné au client et le code de réponse pour cette page, ce qui rend les choses plus simples à créer et maintenir.

```
//:! c15:jsp:DisplayFormData.jsp
<%-- Fetching the data from an HTML form. --%>
<%-- This JSP also generates the form. --%>
<%@ page import="java.util.*" %>
<html><body>
<H1>DisplayFormData</H1><H3>
<%
  Enumeration flds = request.getParameterNames();
  if(!flds.hasMoreElements()) { // No fields %>
    <form method="POST"
    action="DisplayFormData.jsp">
<%  for(int i = 0; i < 10; i++) {  %>
      Field<%=i%>: <input type="text" size="20"
      name="Field<%=i%>" value="Value<%=i%>"><br>
```

```
<%  } %>
   <INPUT TYPE=submit name=submit
   value="Submit"></form>
<%} else {
   while(flds.hasMoreElements()) {
     String field = (String)flds.nextElement();
     String value = request.getParameter(field);
%>
     <li><%= field %> = <%= value %></li>
<%  }
  } %>
</H3></body></html>
///:~
```

Ce qui est intéressant dans cet exemple est de montrer comment le code scriptlet et le code HTML peuvent être entremêlés, au point de générer une page HTML à l'intérieur d'une boucle Java **for**. En particulier ceci est très pratique pour construire tout type de formulaire qui sans cela nécessiterait du code HTML répétitif.

Attributs et visibilité d'une page JSP

En cherchant dans la documentation HTML des servlets et des JSP, on trouve des fonctionnalités donnant des informations à propos de la servlet ou de la page JSP actuellement en cours. L'exemple suivant montre quelques-unes de ces données.

```
//:! c15:jsp:PageContext.jsp
<%--Viewing the attributes in the pageContext--%>
<%-- Note that you can include any amount of code
inside the scriptlet tags --%>
<%@ page import="java.util.*" %>
<html><body>
Servlet Name: <%= config.getServletName() %><br>
Servlet container supports servlet version:
<% out.print(application.getMajorVersion() + "."
+ application.getMinorVersion()); %><br>
<%
  session.setAttribute("My dog", "Ralph");
  for(int scope = 1; scope <= 4; scope++) {  %>
    <H3>Scope: <%= scope %> </H3>
<%  Enumeration e =      pageContext.getAttributeNamesInScope(scope);
    while(e.hasMoreElements()) {
      out.println("\t<li>" +
        e.nextElement() + "</li>");
    }
  }
%>
</body></html>
///:~
```

Cet exemple montre également l'utilisation du mélange de code HTML et d'écriture sur **out** pour fabriquer la page HTML résultante.

La première information générée est le nom de la servlet, probablement » JSP « mais cela dépend de votre implémentation. On peut voir également la version courante du conteneur de servlet au moyen de l'objet application. Pour finir, après avoir déclaré les attributs de la session, les noms d'attributs sont affichés avec une certaine visibilité. On n'utilise pas beaucoup les visibilités dans la plupart des programmes JSP ; on les a montré ici simplement pour donner de l'intérêt à l'exemple. Il existe quatre attributs de visibilité, qui sont : la *visibilité de page* (visibilité 1), la *visibilité de demande* (visibilité 2), la *visibilité de session* (visibilité 3 : ici, le seul élément disponible dans la visibilité de session est » My dog, « ajouté juste après la boucle **for**), et la *visibilité d'application* (visibilité 4), basée sur l'objet **ServletContext**. Il existe un **ServletContext** pour chaque application » Web tournant sur une Machine Virtuelle Java (une application » Web est une collection de servlets et de contenus placés dans un sous-ensemble spécifique de l'espace de nommage de l'URL serveur tel que /catalog. Ceci est généralement réalisé au moyen d'un fichier de configuration). Au niveau de visibilité de l'application on peut voir des objets représentant les chemins du répertoire de travail et du répertoire temporaire.

Manipuler les sessions en JSP

Les sessions ont été introduites dans les sections précédentes à propos des servlets, et sont également disponibles dans les JSP. L'exemple suivant utilise l'objet **session** et permet de superviser le temps au bout duquel la session deviendra invalide.

L'objet **session**est fourni par défaut, il est donc disponible sans code supplémentaire. Les appels de **getID( )**, **getCreationTime( )** et **getMaxInactiveInterval( )** servent à afficher des informations sur l'objet session.

Quand on ouvre la session pour la première fois on a, par exemple, **MaxInactiveInterval** égal à 1800 secondes (30 minutes). Ceci dépend de la configuration du conteneur JSP/servlet. **MaxInactiveInterval** est ramené à 5 secondes afin de rendre les choses intéressantes. Si on rafraîchit la page avant la fin de l'intervalle de 5 secondes, alors on voit :

```
Session value for "My dog" = Ralph
```

Mais si on attend un peu plus longtemps, alors » Ralph « devient **null**.

Pour voir comment les informations de sessions sont répercutées sur les autres pages, ainsi que pour comparer le fait d'invalider l'objet session à celui de le laisser se terminer, deux autres JSP sont créées. La première (qu'on atteint avec le bouton » invalidate « de **SessionObject.jsp**) lit l'information de session et invalide explicitement cette session :

```
//:! c15:jsp:SessionObject2.jsp
<%--The session object carries through--%>
<html><body>
<H1>Session id: <%= session.getId() %></H1>
<H1>Session value for "My dog"
<%= session.getValue("My dog") %></H1>
<% session.invalidate(); %>
</body></html>
///:~
```

Pour tester cet exemple, rafraîchir **SessionObject.jsp**, puis cliquer immédiatement sur le bouton invalidate pour activer **SessionObject2.jsp**. À ce moment on voit toujours » Ralph, « immédiatement (avant que l'intervalle de 5 secondes ait expiré). Rafraîchir **SessionObject2.jsp** pour voir que la session a été invalidée manuellement et que Ralph a disparu.

En recommençant avec **SessionObject.jsp**, rafraîchir la page ce qui démarre un nouvel intervalle de 5 secondes, puis cliquer sur le bouton « Keep Around »], ce qui nous amène à la page suivante, **SessionObject3.jsp**, qui N'invalide PAS la session :

```
//:! c15:jsp:SessionObject3.jsp
<%--The session object carries through--%>
<html><body>
<H1>Session id: <%= session.getId() %></H1>
<H1>Session value for "My dog"
<%= session.getValue("My dog") %></H1>
<FORM TYPE=POST ACTION=SessionObject.jsp>
<INPUT TYPE=submit name=submit Value="Return">
</FORM>
</body></html>
///:~
```

Dû au fait que cette page n'invalide pas la session, » Ralph « est toujours là aussi longtemps qu'on rafraîchit la page avant la fin de l'intervalle de 5 secondes. Ceci n'est pas sans ressembler à un » Tomagotchi], et » Ralph « restera là tant que vous jouerez avec lui, sinon il disparaîtra.

Créer et modifier des cookies

Les cookies ont été introduits dans la section précédente concernant les servlets. Ici encore, la concision des JSP rend l'utilisation des cookies plus simple que dans les servlets. L'exemple suivant montre cela en piégeant les cookies liés à une demande en entrée, en lisant et modifiant leur date d'expiration, et en liant un nouveau cookie à la réponse :

```
//:! c15:jsp:Cookies.jsp
<%--This program has different behaviors under
different browsers! --%>
<html><body>
<H1>Session id: <%= session.getId() %></H1>
<%
Cookie[ « cookies = request.getCookies();
for(int i = 0; i < cookies.length; i++) { %>
  Cookie name: <%= cookies[i].getName() %> <br>
  value: <%= cookies[i].getValue() %><br>
  Old max age in seconds:
  <%= cookies[i].getMaxAge() %><br>
  <% cookies[i].setMaxAge(5); %>
  New max age in seconds:
  <%= cookies[i].getMaxAge() %><br>
<% } %>
<%! int count = 0; int dcount = 0; %>
```

```
<% response.addCookie(new Cookie(
   "Bob" + count++, "Dog" + dcount++)); %>
</body></html>
///:~
```

Chaque navigateur ayant sa manière de stocker les cookies, le résultat sera différent suivant le navigateur (ce qui n'est pas rassurant, mais peut-être réparerez-vous un certain nombre de bugs en lisant cela). Par ailleurs, il se peut aussi que l'on ait des résultats différents en arrêtant le navigateur et en le relançant, plutôt que de visiter une autre page puis de revenir à **Cookies.jsp**. Remarquons que l'utilisation des objets session semble plus robuste que l'utilisation directe des cookies.

Après l'affichage de l'identifiant de session, chaque cookie du tableau de cookies arrivant avec l'objet **request** object est affiché, ainsi que sa date d'expiration. La date d'expiration est modifiée et affichée à son tour pour vérifier la nouvelle valeur, puis un nouveau cookie est ajouté à la réponse. Toutefois, il est possible que votre navigateur semble ignorer les dates d'expiration ; il est préférable de jouer avec ce programme en modifiant la date d'expiration pour voir ce qui se passe avec divers navigateurs.

Résumé sur les JSP

Cette section était un bref apperçu des JSP ; cependant avec les sujets abordés ici (ainsi qu'avec le langage Java appris dans le reste du livre, sans oublier votre connaissance personnelle du langage HTML) vous pouvez dès à présent écrire des pages Web sophistiquées via les JSP. La syntaxe JSP n'est pas particulièrement profonde ni compliquée, et si vous avez compris ce qui était présenté dans cette section vous êtes prêts à être productifs en utilisant les JSP. Vous trouverez d'autres informations dans la plupart des livres sur les servlets, ou bien à *java.sun.com*.

La disponibilité des JSP est très agréable, même lorsque votre but est de produire des servlets. Vous découvrirez que si vous vous posez une question à propos du comportement d'une fonctionnalité servlet, il est plus facile et plus rapide d'y répondre en écrivant un programme de test JSP qu'en écrivant une servlet. Ceci est dû en partie au fait qu'on ait moins de code à écrire et qu'on puisse mélanger le code Java et le code HTML, mais l'avantage devient particulièrement évident lorsqu'on voit que le Conteneur JSP se charge de la recompilation et du chargement du JSP à votre place chaque fois que la source est modifiée.

Toutefois, aussi fantastiques que soient les JSP, il vaut mieux garder à l'esprit que la création de pages JSP requiert un plus haut niveau d'habileté que la simple programmation en Java ou la simple création de pages Web. En outre, debugger une page JSP morcelée n'est pas aussi facile que débugger un programme Java, car (pour le moment) les messages d'erreur sont assez obcurs. Cela changera avec l'évolution des systèmes de développement, et peut-être verrons nous d'autres technologies construites au-dessus de Java plus adaptées aux qualités des concepteurs de site web.

RMI (Remote Method Invocation) : Invocation de méthodes distantes

Les approches traditionnelles pour exécuter des instructions à travers un réseau sur d'autres ordinateurs étaient aussi confuses qu'ennuyeuses et sujettes aux erreurs. La meilleure manière d'aborder ce problème est de considérer qu'en fait un objet donné est présent sur une autre machine, on lui envoie un message et l'on obtient le résultat comme si l'objet était instancié sur votre machine locale. Cette simplification est exactement celle que Java 1.1 *Remote Method Invocation* (RMI - Invocation de méthodes distantes) permet de faire. Cette section vous accompagne à travers les étapes

nécessaires pour créer vos propres objets RMI.

Interfaces Remote

RMI utilise beaucoup d'interfaces. Lorsque l'on souhaite créer un objet distant, l'implémentation sous-jacente est masquée en passant par une interface. Ainsi, lorsque qu'un client obtient une référence vers un objet distant, ce qu'il possède réellement est une référence intermédiaire, qui renvoie à un bout de code local capable de communiquer à travers le réseau. Mais nul besoin de se soucier de cela, il suffit de juste d'envoyer des messages par le biais de cette référence intermédiaire.

La création d'une interface distante doit respecter ces directives :

1. L'interface distante doit être **public** (il ne peut y avoir accès package, autrement dit d'accès friendly). Sinon, le client recevra une erreur lorsqu'il tentera d'obtenir un objet distant qui implémente l'interface distante.
2. L'interface distante doit hériter de l'interface **java.rmi.Remote**.
3. Chaque méthode de l'interface distante doit déclarer **java.rmi.RemoteException** dans sa clause **throws** en plus des autres exceptions spécifiques à l'application.
4. Un objet distant passé en argument ou en valeur de retour (soit directement ou inclus dans un objet local), doit être déclaré en tant qu'interface distante et non comme la classe d'implémentation.

Voici une interface distante simple qui représente un service d'heure exacte :

```
//: c15:rmi:PerfectTimeI.java
// L'interface distante PerfectTime.
package c15.rmi;
import java.rmi.*;

interface PerfectTimeI extends Remote {
  long getPerfectTime() throws RemoteException;
} ///:~
```

Cela ressemble à n'importe quelle autre interface mis à part qu'elle hérite de **Remote** et que toutes ses méthodes émettent **RemoteException**. Rappelez vous qu'une **interface** et toutes ses méthodes sont automatiquement **publiques**.

Implémenter l'interface distante

Le serveur doit contenir une classe qui hérite de **UnicastRemoteObject** et qui implémente l'interface distante. Cette classe peut avoir aussi des méthodes supplémentaires, mais bien sûr seules les méthodes appartenant à l'interface distante seront disponibles pour le client puisque celui-ci n'obtiendra qu'une référence vers l'interface, et non vers la classe qui l'implémente.

Vous devez explicitement définir le constructeur de cet objet distant même si vous définissez seulement un constructeur par défaut qui appelle le constructeur de base. Vous devez l'écrire parce qu'il doit émettre **RemoteException**.

Voici l'implémentation de l'interface distante **PerfectTimeI**:

```
//: c15:rmi:PerfectTime.java
// L'implémentation de
// l'objet distant PerfectTime.
package c15.rmi;
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;
import java.net.*;

public class PerfectTime
    extends UnicastRemoteObject
    implements PerfectTimeI {
  // Implémentation de l'interface:
  public long getPerfectTime()
      throws RemoteException {
    return System.currentTimeMillis();
  }
  // Doit implémenter le constructeur
  // pour émettre RemoteException:
  public PerfectTime() throws RemoteException {
    // super(); // Appelé implicitement
  }
  // Inscription auprès du service RMI :
  public static void main(String[ « args) {
    System.setSecurityManager(
      new RMISecurityManager());
    try {
      PerfectTime pt = new PerfectTime();
      Naming.bind(
        "//peppy:2005/PerfectTime", pt);
      System.out.println("Ready to do time");
    } catch(Exception e) {
      e.printStackTrace();
    }
  }
} ///:~
```

Ici, **main( )** se charge de tous les détails de mise en place du serveur. Une application qui met en service des objets RMI doit à un moment :

1. Créer et installer un gestionnaire de sécurité qui supporte RMI. Le seul disponible pour RMI fourni dans la distribution Java est **RMISecurityManager**.
2. Créer une ou plusieurs instances de l'objet distant. Ici, vous pouvez voir la création de l'objet **PerfectTime**.
3. Enregistrer au moins un des objets distants grâce au registre d'objets distants RMI pour des raisons d'amorçage. Un objet distant peut avoir des méthodes qui retournent des références vers les autres objets distants. En le mettant en place, le client ne doit s'adresser au registre qu'une seule fois pour obtenir ce premier objet distant.

Mise en place du registre

Ici, vous pouvez voir un appel à la méthode **statique Naming.bind( )**. Toutefois, cet appel nécessite que le registre fonctionne dans un autre processus de l'ordinateur. Le nom du registre serveur est **rmiregistry**, et sous Windows 32-bit vous utiliserez :

```
start rmiregistry
```

pour démarrer celui-ci en fond. Sous Unix, ce sera :

```
rmiregistry &
```

Comme beaucoup d'applications réseau, le **rmiregistry** est localisé à l'adresse IP de la machine qui l'a démarré, mais il doit aussi écouter un port. Si vous invoquez le **rmiregistry** comme ci-dessus, sans argument, le port écouté par le registre sera par défaut 1099. Si vous souhaitez que ce soit un autre port, vous devez ajouter un argument à la ligne de commande pour le préciser. Dans cet exemple, le port sera 2005, ainsi le **rmiregistry** devra être démarré de cette manière sous Windows 32-bit :

```
start rmiregistry 2005
```

ou pour Unix:

```
rmiregistry 2005 &
```

L'information concernant le port doit aussi être fourni à la commande **bind( )**, ainsi que l'adresse IP de la machine où se trouve le registre. Mais il faut mentionner que cela peut être un problème frustrant en cas de tests de programmes RMI en local (de la même manière que les programmes réseau testés plus loin dans ce chapitre). En effet dans le JDK version 1.1.1, il y a quelques problèmes : [69]

1. **localhost** ne fonctionne pas avec RMI. Aussi, pour faire une expérience avec RMI sur une seule machine, vous devez fournir le nom de la machine. Pour découvrir le nom de votre machine sous Windows 32-bit, allez dans le Panneau de configuration et sélectionnezRéseau. Sélectionnez l'onglet Identification, vous verrez apparaître le nom de l'ordinateur. Dans mon cas, j'ai appelé mon ordinateur Peppy. Il semble que la différence entre majuscules et minuscules soit ignorée.
2. RMI ne fonctionnera pas à moins que votre ordinateur ait une connexion TCP/IP active, même si tous vos composants discutent entre eux sur la machine locale. Cela signifie que vous devez vous connecter à Internet pour essayer de faire fonctionner le programme ou vous obtiendrez quelques messages d'exception obscurs.

En ayant tout cela à l'esprit, la commande **bind( )** devient :

```
Naming.bind("//peppy:2005/PerfectTime", pt);
```

Si vous utilisez le port par défaut 1099, nul besoin de préciser le port, donc vous pouvez dire :

```
Naming.bind("//peppy/PerfectTime", pt);
```

Vous devriez être capable de réaliser un test local en omettant l'adresse IP et en utilisant sim-

plement l'identifiant :

```
Naming.bind("PerfectTime", pt);
```

Le nom pour le service est arbitraire ; il se trouve que c'est PerfectTime ici, comme le nom de la classe, mais un tout autre nom conviendrait. L'important est que ce nom soit unique dans le registre, connu par le client pour qu'il puisse se procurer l'objet distant. Si le nom est déjà utilisé dans le registre, celui renvoie une **AlreadyBoundException**. Pour éviter cela, il faut passer à chaque fois par un appel à **rebind( )** à la place de **bind( )**, en effet **rebind( )** soit ajoute une nouvelle entrée, soit remplace celle existant déjà.

Durant l'exécution de **main( )**, l'objet a été créé et enregistré, il reste ainsi en activité dans le registre, attendant qu'un client arrive et fasse appel à lui. Tant que **rmiregistry** fonctionne et que **Naming.unbind( )** n'est pas appelé avec le nom choisi, l'objet sera présent. C'est pour cela que lors de la conception du code, il faut redémarrer **rmiregistry** après chaque compilation d'une nouvelle version de l'objet distant.

rmiregistry n'est pas forcément démarré en tant que processus externe. S'il est sûr que l'application est la seule qui utilise le registre, celui peut être mis en fonction à l'intérieur même du programme grâce à cette ligne :

```
LocateRegistry.createRegistry(2005);
```

Comme auparavant, nous utilisons le numéro de port 2005 dans cet exemple. C'est équivalent au fait de lancer **rmiregistry 2005** depuis la ligne de commande, mais c'est souvent plus pratique lorsque l'on développez son code RMI puisque cela élimine les étapes supplémentaires qui consistent à arrêter et redémarrer le registre. Une fois cette instruction exécutée, la méthode **bind( )** de la classe **Naming** peut être utilisée comme précédemment.

Si l'on compile et exécute **PerfectTime.java**, cela ne fonctionnera pas même si **rmiregistry** a été correctement mis en route. Cela parce que la machinerie pour RMI n'est pas complète. Il faut d'abord créer les stubs et skeletons qui assurent les opérations de connexions réseaux et permettent de faire comme si l'objet distant était juste un objet local sur de la machine.

Ce qui se passe derrière la scène est complexe. Tous les objets envoyés à un objet distant ou reçus de celui-ci doivent **implémenter Serializable** (pour passer des références distantes plutôt que les objets complets, les arguments peuvent **implémenter Remote**), ainsi on peut considérer que les stubs et les skeletons assurent automatiquement la sérialisation et la déserialisation tout en gérant l'acheminement des arguments et du résultat au travers du réseau. Par chance, vous n'avez rien à connaître de tout cela, mais vous *devez* avoir créé les stubs et les skeletons. C'est une procédure simple : on invoque l'outil **rmic** sur le code compilé, et il crée les fichiers nécessaires. La seule chose obligatoire est donc d'ajouter cette étape à la procédure de compilation.

L'outil **rmic** est particulier en ce qui concerne les packages et les classpaths. **PerfectTime.java** est dans le **package c15.rmi**, et même **rmic** est invoqué dans le même répertoire que celui où se trouve **PerfectTime.class**, **rmic** ne trouvera pas le fichier, puisqu'il se repère grâce au classpath. Vous devez ainsi préciser la localisation du classpath, comme ceci :

```
rmic c15.rmi.PerfectTime
```

La commande ne nécessite pas d'être exécutée à partir du répertoire contenant **PerfectTime.class**, mais les résultats seront placés dans le répertoire courant.

Lorsque **rmic** a été exécuté avec succès, deux nouvelles classes sont obtenues dans le répertoire :

```
PerfectTime_Stub.class
PerfectTime_Skel.class
```

correspondant au stub et au skeleton. Dès lors, vous êtes prêt à faire communiquer le serveur et le client.

Utilisation de l'objet distant

Le but de RMI est de simplifier l'utilisation d'objets distants. La seule chose supplémentaire qui doit être réalisée dans le programme client est de rechercher et de rapatrier depuis le serveur l'interface distante. Après quoi, c'est simplement de la programmation Java classique : envoyer des messages aux objets. Voici le programme qui utilise **PerfectTime**:

```
//: c15:rmi:DisplayPerfectTime.java
// Utilise l'objet distant PerfectTime.
package c15.rmi;
import java.rmi.*;
import java.rmi.registry.*;

public class DisplayPerfectTime {
  public static void main(String[ « args) {
    System.setSecurityManager(
      new RMISecurityManager());
    try {
      PerfectTimeI t =       (PerfectTimeI)Naming.lookup(
          "//peppy:2005/PerfectTime");
      for(int i = 0; i < 10; i++)
        System.out.println("Perfect time = " +
          t.getPerfectTime());
    } catch(Exception e) {
      e.printStackTrace();
    }
  }
} ///:~
```

L'identifiant alpha-numérique est le même que celui utilisé pour enregistrer l'objet avec **Naming**, et la première partie représente l'adresse et le numéro du port. Cette URL permet par exemple de désigner une machine sur Internet.

Ce qui est retourné par **Naming.lookup( )** doit être transtypé vers l'interface distante, *pas* vers la classe. Si l'utilisation de la classe à la place renverrait une exception.

Vous pouvez observer dans l'appel de la méthode

```
t.getPerfectTime( )
```

qu'une fois la référence vers l'objet distant obtenue, la programmation avec celle-ci ou avec un object distant est identique (avec une différence : les méthodes distantes émettent **RemoteEx-**

ception).

Introduction à CORBA

Dans le cadre d'importantes applications distribuées, vos besoins risquent de ne pas être satisfaits par les approches précédentes : comme par exemple, l'intégration de bases existantes, ou l'accès aux services d'un objet serveur sans se soucier de sa localisation physique. Ces situations nécessitent une certaine forme de Remote Procedure Call (RPC), et peut-être une indépendance par rapport au langage. Là, CORBA peut vous aider.

CORBA n'est pas une fonctionnalité du langage ; c'est une technologie d'integration. C'est une spécification que les fabricants peuvent suivre pour implémenter des produits supportant une intégration CORBA. CORBA fait partie du travail réalisé par OMG afin de définir une structure standard pour l'interopérabilité d'objets distribués et indépendants du langage.

CORBA fournit la possibilité de faire des appels à des procédures distantes dans des objets Java et des objets non-Java, et d'interfacer des systèmes existants sans se soucier de leur emplacement. Java ajoute un support réseau et un langage orienté-objet agréable pour construire des applications graphiques ou non. Le modèle objet de l'OMG et celui de Java vont bien ensemble ; par exemple, Java et CORBA mettent tout deux en oeuvre le concept d'interface et le modèle de référence objet.

Principes de base de CORBA

La spécification de l'interopérabilité objet est généralement désignée comme l'Object Manager Architecture (OMA). L'OMA définit deux composants : le Core Object Model et l'OMA Reference Architecture. Le Core Object Model met en place les concepts de base d'objet, d'interface, d'opération, et ainsi de suite (CORBA est un raffinement de Core Object Model). L'OMA Reference Architecture définit l'infrastructure sous-jacente des services et des mécanismes qui permettent aux objets d'inter-opérer. L'OMA Reference Architecture contient l'Object Request Broker (ORB), les Object Services (désignés aussi comme les services CORBA) et les outils communs.

L'ORB est le bus de communication par lequel les objets peuvent réclamer des services auprès des autres objets, sans rien connaître de leur emplacement physique. Cela signifie que ce qui ressemble à un appel de méthode dans le code client est vraiment une opération complexe. D'abord, une connexion avec l'objet servant doit exister et pour créer cette connexion, l'ORB doit savoir où se trouve le code implémentant cette partie serveur. Une fois que la connexion est établie, les arguments de la méthode doivent être arrangés (marshaled), c'est à dire convertis en un flux binaire pour être envoyés à travers le réseau. Les autres informations qui doivent être envoyées sont le nom de la machine serveur, le processus serveur et l'identité de l'objet servant au sein de ce processus. Finalement, l'information est envoyée par l'intermédiaire d'un protocole bas-niveau, elle est décodée du côté du serveur, l'appel est exécuté. L'ORB masque tout de cette complexité au programmeur et rend l'opération presque aussi simple que l'appel d'une méthode d'un objet local.

Cette spécification n'a pas pour but d'expliquer comment le coeur de l'ORB devrait être implémenté, mais elle permet une compatibilité fondamentale entre les différents ORBs des fournisseurs, l'OMG définit un ensemble de services qui sont accessibles par l'intermédiaire d'interfaces standards.

CORBA Interface Definition Language (IDL - Langage de Définition d'Interface)

CORBA a été mis au point pour être transparent vis-à-vis du langage : un objet client peut appeler des méthodes d'un objet serveur d'une classe différente, sans se préoccuper du langage avec lequel elles sont implémentées. Bien sûr, l'objet client doit connaître le nom et les prototypes des méthodes que l'objet servant met à disposition. C'est là que l'IDL intervient. L'IDL de CORBA est un moyen indépendant du langage qui permet de préciser les types de données, les attributs, les opérations, les interfaces, et plus encore. La syntaxe de l'IDL est similaire à celles du C++ ou de Java. La table qui suit montre la correspondance entre quelques uns des concepts communs au trois langages qui peuvent être spécifiés à travers l'IDL de CORBA :

CORBA IDL	**Java**	**C++**
Module	Package	Namespace
Interface	Interface	Pure abstract class
Method	Method	Member function

Le concept d'héritage est également supporté, en utilisant le séparateur deux-points comme en C++. Le programmeur écrit en IDL la description des attributs, des méthodes et des interfaces qui seront implémentés et utilisés par le serveur et les clients. L'IDL est ensuite compilé par un compilateur IDL/Java propre au fournisseur, qui lit le source IDL et génère le code Java.

Le compilateur IDL est un outil extrêmement utile : il ne génère pas juste le source Java équivalent à l'IDL, il génère aussi le code qui sera utilisé pour réunir les arguments des méthodes et pour réaliser les appels distants. Ce code, appelé le code stub et le code skeleton, est organisé en plusieurs fichiers source Java et fait habituellement partie d'un même package Java.

Le service de nommage (naming service)

Le service de nommage est l'un des services fondamentaux de CORBA. L'objet CORBA est accessible par l'intermédiaire d'une référence, une information qui n'a pas de sens pour un lecteur humain. Mais les références peuvent être des chaînes de caractères définies par le programmeur. Cette opération est désignée comme chaînifier la référence, et l'un des composants de l'OMA, le service de nommage, est dévoué à la conversion nom-vers-objet et objet-vers-nom et gère ces correspondances. Puisque le service de nommage joue le rôle d'un annuaire téléphonique que les serveurs et les clients peuvent consulter et manipuler, il fonctionne dans un processus séparé. Créer une correspondance objet-vers-nom est appelé *lier (binding) un objet*, et supprimer cette correspondance est dit *délier (unbinding)*. Obtenir l'objet référence en passant la chaîne de caractères est appelé *résoudre le nom*.

Par exemple, au démarrage, une application serveur peut créer un objet servant, enregistrer l'objet auprès du service de nommage, et attendre que des clients fassent des requêtes. Un client obtient d'abord une référence vers cet objet servant en résolvant le nom et ensuite peut faire des appels auprès du serveur en utilisant cette référence.

A nouveau, la spécification du service de nommage fait partie de CORBA, mais l'application qui l'implémente est mise à disposition par le fournisseur de l'ORB. Le moyen d'accéder à ce service de nommage peut varier d'un fournisseur à l'autre.

Un exemple

Le code montré ici ne sera pas très élaboré car les différents ORBs ont des moyens d'accéder aux services CORBA qui divergent, ainsi les exemples dépendent du fournisseur. L'exemple qui suit utilise JavaIDL, un produit gratuit de Sun, qui fournit un ORB basique, un service de nommage, et un compilateur IDL-vers-Java. De plus, comme Java est encore jeune et en constante évolution, les différents produits CORBA pour Java n'incluent forcément toutes les fonctionnalités de CORBA.

Nous voulons implémenter un serveur, fonctionnant sur une machine donnée, qui soit capable de retourner l'heure exacte. Nous voulons aussi implémenter un client qui demande l'heure exacte. Dans notre cas, nous allons réalisons les deux programmes en Java, mais nous pourrions faire de même avec deux langages différents (ce qui se produit souvent dans la réalité).

Écrire le source IDL

La première étape consiste à écrire une description en IDL des services proposés. Ceci est généralement réalisé par le programmeur du serveur, qui est ensuite libre d'implémenter le serveur dans n'importe quel langage pour lequel un compilateur CORBA IDL existe. Le fichier IDL est communiqué au programme de la partie cliente et devient le pont entre les langages.

L'exemple qui suit montre la description IDL de notre serveur **ExactTime** :

```
//: c15:corba:ExactTime.idl
//# Vous devez installer idltojava.exe de
//# java.sun.com et ajuster le paramétrage pour utiliser
//# votre préprocesseur C local pour compiler
//# ce fichier. Voyez la documentation sur java.sun.com.
module remotetime {
  interface ExactTime {
    string getTime();
  };
}; ///:~
```

C'est donc la déclaration de l'interface **ExactTime** au sein de l'espace de nommage **remotetime**. L'interface est composée d'une seule méthode qui retourne l'heure actuelle dans une **chaîne de caractères**.

Création des stubs et des skeletons

La deuxième étape consiste à compiler l'IDL pour créer le code Java du stub et du skeleton que nous utiliserons pour implémenter le client et le serveur. L'outil fournit par le produit JavaIDL est **idltojava** :

```
idltojava remotetime.idl
```

Cela générera automatiquement à la fois le code pour le stub et celui pour le skeleton. **Idltojava** génère un **package** Java nommé selon le module IDL **remotetime** et les fichiers Java générés sont déposés dans ce sous-répertoire **remotetime**. **_ExactTimeImplBase.java** est le skeleton que nous allons utiliser pour implémenter l'objet servant et **_ExactTimeStub.java** sera utilisé pour le client. Il y a des représentations Java de l'interface IDL dans **ExactTime.java** et toute une série d'autres fichiers de support utilisés, par exemple, pour faciliter l'accès aux fonctions du service de

nommage.

Implémentation du serveur et du client

Ci-dessous, vous pouvez voir le code pour la partie serveur. L'implémentation de l'objet servant est dans la classe **ExactTimeServer**. **RemoteTimeServer** est l'application qui créé l'objet servant, l'enregistre auprès de l'ORB, donne un nom à la référence vers l'objet, et qui ensuite s'assoit tranquillement en attendant les requêtes des clients.

```
//: c15:corba:RemoteTimeServer.java
import remotetime.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
import java.util.*;
import java.text.*;

// Implémentation de l'objet servant
class ExactTimeServer extends _ExactTimeImplBase {
  public String getTime(){
    return DateFormat.
        getTimeInstance(DateFormat.FULL).
          format(new Date(
            System.currentTimeMillis()));
  }
}

// Implémentation de l'application distante
public class RemoteTimeServer {
  public static void main(String[ « args) {
    try {
      // Crée l'ORB et l'initialise:
      ORB orb = ORB.init(args, null);
      // Crée l'objet servant et l'enregistre :
      ExactTimeServer timeServerObjRef =       new ExactTimeServer();
      orb.connect(timeServerObjRef);
      // Obtient la racine du contexte de nommage :
      org.omg.CORBA.Object objRef =       orb.resolve_initial_references(
        "NameService");
      NamingContext ncRef =       NamingContextHelper.narrow(objRef);
      // Associe un nom
      // à la référence de l'objet (binding):
      NameComponent nc =       new NameComponent("ExactTime", "");
      NameComponent[ « path = { nc };
      ncRef.rebind(path, timeServerObjRef);
      // Attend les requêtes des clients :
      java.lang.Object sync =       new java.lang.Object();
      synchronized(sync){
```

```
            sync.wait();
          }
        }
        catch (Exception e) {
          System.out.println(
            "Remote Time server error: " + e);
          e.printStackTrace(System.out);
        }
      }
    } ///:~
```

Comme vous pouvez le constater, implémenter l'objet servant est simple ; c'est une classe Java classique qui hérite du code du skeleton généré par le compilateur IDL. Les choses se complexifient un peu lorsqu'on en vient à interagir avec l'ORB et les autres services CORBA.

Quelques services CORBA

Voici une courte description de ce qui réalise le code relevant de JavaIDL (en évitant la partie de code CORBA qui dépend du fournisseur). La première ligne dans le **main( )** démarre l'ORB parce que bien sûr notre objet servant aura besoin d'interagir avec celui-ci. Juste après l'initialisation de l'ORB, l'objet servant est créé. En réalité, le terme correct serait un *objet servant temporaire (transient)* : un objet qui reçoit les requêtes provenant des clients, et dont la durée de vie est limitée à celle du processus qui l'a créé. Une fois l'objet servant *temporaire* créé, il est enregistré auprès de l'ORB, ce qui signifie alors que l'ORB connaît son existence et peut rediriger les requêtes vers lui.

A partir de là, tout ce dont nous disposons est **timeServerObjRef**, une référence vers un objet qui n'est connu qu'à l'intérieur du processeur serveur actuel. L'étape suivante va consister à associer un nom alphanumérique à cet objet servant ; les clients utiliseront ce nom pour localiser l'objet servant. Cette opération sera réalisée grâce à l'aide du service de nommage. Tout d'abord, nous avons besoin d'une référence vers le service de nommage ; la méthode **resolve_initial_references( )** utilise la référence objet « chainifiée » du service de nommage qui s'appelle NameService dans JavaIDL, et retourne la référence vers l'objet. Celle-ci est transformée en une référence spécifique à **NamingContext** au moyen de la méthode **narrow( )**. Nous pouvons maintenant utiliser les services de nommage.

Pour associer l'objet servant avec une référence objet « chainifiée », nous créons d'abord un objet **NameComponent**, initialisé avec la chaîne de caractères qui sera associée : face="Georgia">« ExactTime ». Ensuite, en utilisant la méthode **rebind( )**, la référence alphanumérique est associée à la référence vers l'objet. Nous utilisons **rebind( )** pour mettre en place une référence, même si celle-ci existe déjà. Un nom est composé dans CORBA d'une séquence de NameComponents (voilà pourquoi nous utilisons un tableau pour associer le nom à la référence).

L'objet est enfin prêt à être utilisé par des clients. A ce moment-là, le serveur entre dans un état d'attente. Encore une fois, ceci est nécessaire puisqu'il s'agit d'un serveur temporaire : sa durée de vie dépend du processus serveur. JavaIDL ne supporte pas actuellement les objets persistants, qui survivent après la fin de l'exécution du processus qui les a créés.

Maintenant que nous avons une idée de ce que fait le code du serveur, regardons le code du client :

```
//: c15:corba:RemoteTimeClient.java
import remotetime.*;
import org.omg.CosNaming.*;
import org.omg.CORBA.*;

public class RemoteTimeClient {
  public static void main(String[ « args) {
    try {
      // Création et initialisation de l'ORB :
      ORB orb = ORB.init(args, null);
      // Obtient la racine du contexte de nommage :
      org.omg.CORBA.Object objRef =       orb.resolve_initial_references(
        "NameService");
      NamingContext ncRef =       NamingContextHelper.narrow(objRef);
      // Résout la référence alphanumérique
      // du serveur d'heure :
      NameComponent nc =       new NameComponent("ExactTime", "");
      NameComponent[ « path = { nc };
      ExactTime timeObjRef =       ExactTimeHelper.narrow(
        ncRef.resolve(path));
      // Effectue une requête auprès de l'objet servant :
      String exactTime = timeObjRef.getTime();
      System.out.println(exactTime);
    } catch (Exception e) {
      System.out.println(
        "Remote Time server error: " + e);
      e.printStackTrace(System.out);
    }
  }
} ///:~
```

Activation du processus du service de nommage

Nous avons enfin un serveur et un client prêts à interopérer. Nous avons pu voir que tous deux ont besoin du service de nommage pour associer et résoudre les références alphanumériques. Le processus du service de nommage doit être démarré avant de faire fonctionner aussi bien le serveur que le client. Dans JavaIDL, le service de nommage est une application Java fournie avec le produit, mais qui peut être différente dans d'autres produits. Le service de nommage de JavaIDL fonctionne dans une JVM et écoute par défaut le port réseau 900.

Activation du serveur et du client

Vos applications serveur et client sont prêtes à démarrer (dans cet ordre, puisque votre serveur est transient (temporaire)). Si tout est bien en place, vous obtiendrez une simple ligne de sortie dans la fenêtre console de votre client, indiquant l'heure courante. Bien sûr, cela ne semble pas très excitant en soit, mais vous devriez prendre en compte une chose : même si ils sont physiquement sur la même machine, les applications client et serveur fonctionnent dans des machines virtuelles

différents et peuvent communiquer à travers une couche de l'intégration sous-jacente, l'ORB et le service de nommage.

Ceci constitue un exemple simple, destiné à fonctionner sans réseau, mais un ORB est généralement configuré pour qu'il rende les localisations transparentes. Lorsque le serveur et le client sont sur des machines différentes, l'ORB peut résoudre des références alphanumériques en utilisant un composant désigné *Implementation Repository*. Bien que le Implementation Repository fasse partie de CORBA, il n'y a quasiment pas de spécification, et donc il est différent d'un fabricant à l'autre.

Ainsi que vous vous en doutez, CORBA représente davantage que ce qui est montré ici, mais ainsi vous aurez compris l'idée de base. Si vous souhaitez plus d'informations à propos de CORBA, le premier endroit à visiter est le site Web de l'OMG : http://www.omg.org. Vous y trouverez la documentation, les white papers et les références vers les autres sources et produits pour CORBA.

Les Applets Java et CORBA

Les applets Java peuvent se comporter comme des clients CORBA. Par ce biais, une applet peut accéder à des informations et des services distants publiés sous la forme d'objets CORBA. Mais une applet ne peut se connecter qu'au serveur depuis lequel elle a été téléchargée, aussi tous les objets CORBA avec lesquels l'applet interagit doivent être situés sur ce serveur. C'est l'opposé de ce que CORBA essaye de faire : vous offrir une totale transparence de localisation.

C'est une question de sécurité réseau. Si vous êtes sur un intranet, la solution est d'éliminer les restrictions de sécurité du navigateur. Ou, de mettre en place une politique de firewall pour la connexion vers des serveurs externes.

Certains produits proposant des ORBs Java offrent des solutions propriétaires à ce problème. Par exemple, certains implémentent ce qui s'appelle un HTTP Tunneling, alors que d'autres ont des fonctionnalités firewall spécifiques.

C'est un point trop complexe pour être exposé dans une annexe, mais c'est quelque chose que vous devriez approfondir.

CORBA face à RMI

Vous avez vu que la principale fonctionnalité de CORBA est le support de RPC, qui permet à vos objets locaux d'appeler des méthodes d'objets distants. Bien sûr, il y a déjà une fonctionnalité native à Java qui réalise exactement la même chose : RMI (voir Chapitre 15). Là où RMI rend possible RPC entre des objets Java, CORBA rend possible RPC entre des objets implémentés dans n'importe quel langage.

Toutefois, RMI peut être utilisé pour appeler des services auprès de code distant non-Java. Tout ce dont vous avez besoin est d'une sorte d'objet Java adaptateur autour du code non-Java sur la partie serveur. L'objet adapteur est connecté à l'extérieur aux objets clients par RMI, et en interne il se connecte au code non-Java en utilisant l'une des techniques vues précédemment, comme JNI ou J/Direct.

Cette approche nécessite que vous écriviez une sorte de couche d'intégration, exactement ce que CORBA fait pour vous, mais vous n'avez pas besoin ici d'un ORB venu de l'extérieur.

Enterprise Java Beans

[70]A partir de maintenant, vous avez été présenté à CORBA et à RMI. Mais pourriez-vous imaginer de tenter le développement d'une application à grande échelle en utilisant CORBA et/ou RMI ? Votre patron vous a demandé de développer une application multi-tiers pour consulter et mettre à jour des enregistrements dans une base de données, le tout à travers une interface Web. Vous vous assoyez et pensez à ce que cela veut vraiment dire. Sûr, vous pouvez écrire une application qui utilise JDBC, une interface Web qui utilise JSP et des Servlets, et un système distribué qui utilise CORBA/RMI. Mais quelles autres considérations devez-vous prendre en compte lorsque vous développez un système basé sur des objets distribués plutôt qu'avec des APIs classiques ? Voici les problèmes :

Performance : Les nouveaux objets distribués que vous avez créé devront être performants puisque ils devront potentiellement répondre à plusieurs clients en même temps. Donc vous allez vouloir mettre en place des techniques d'optimisation comme l'utilisation de cache, la mise en commun des ressources (comme les connexions à des bases de données par JDBC). Vous devrez aussi gérer le cycle de vie de votre objet distribué.

Adaptation à la charge : Les objets distribués doivent aussi s'adapter à l'augmentation de la charge. La scalabilité dans une application distribuée signifie que le nombre d'instances de votre objet distribué peut augmenter et passer sur une autre machine sans la modification du moindre code. Prenez par exemple un système que vous développez en interne comme une petite recherche de clients dans votre organisation depuis une base de données. L'application fonctionne bien quand vous l'utilisez, mais votre patron l'a vue et a dit : "Robert, c'est un excellent système, mettez ça sur notre site Web public maintenant !!!". Est-ce que mon objet distribué est capable de supporter la charge d'une demande potentiellement illimitée.

Sécurité : Mon objet distribué contrôle-t-il l'accès des clients ? Puis-je ajouter de nouveaux utilisateurs et des rôles sans tout recompiler ?

Transactions distribuées : Mon objet distribué peut-il utiliser de manière transparente des transactions distribuées ? Puis-je mettre à jour ma base de données Oracle et Sybase simultanément au sein de la même transaction et les annuler ensemble si un certain critère n'est pas respecté ?

Réutilisation : Ai-je créé mon objet distribué de telle sorte que je puisse le passer d'une application serveur d'un fournisseur à une autre ? Puis-je revendre mon objet distribué (composant) à quelqu'un d'autre ? Puis-je acheter un composant de quelqu'un d'autre et l'utiliser sans avoir à recompiler et à tailler dans son code ?

Disponibilité : Si l'une des machines de mon système tombe en panne, mes clients sont-ils capables de basculer des copies de sauvegarde de mes objets fonctionnant sur d'autres machines ?

Comme vous avez pu le voir ci-dessus, les considérations qu'un développeur doit prendre en compte lorsqu'il développe un système distribué sont complexes, et nous n'avons même pas parlé de la solution du problème qui nous essayons de résoudre à l'origine !

Donc maintenant vous avez une liste de problèmes supplémentaires que vous devez résoudre. Alors comme allez-vous vous y prendre ? Quelqu'un l'a certainement déjà fait ? Ne pourrais-je pas utiliser des modèles de conception bien connus pour m'aider à résoudre ces problèmes ? Soudain, une idée vous vient à l'esprit... « Je pourrais créer un framework qui prend en charge toutes ces contraintes et écrire mes composants en m'appuyant sur ce framework ! »... C'est là que les Entreprise JavaBeans rentrent en jeu.

Sun, avec d'autres fournisseurs d'objets distribués a compris que tôt ou tard toute équipe de

développement serait en train de réinventer la roue. Ils ont donc créé la spécification des Entreprise JavaBeans (EJB). Les EJB sont une spécification d'un modèle de composant côté serveur qui prend en compte toutes les considérations mentionnées plus haut utilisant une approche standard et définie qui permet aux développeurs de créer des composants (qui sont appelés des Entreprise JavaBeans), qui sont isolés du code de la 'machinerie' bas-niveau et focalisés seulement sur la mise en place de la logique métier. Et puisque les EJBs sont définis sur un standard, ils peuvent être utilisés sans être dépendant d'un fournisseur.

JavaBeans contre EJBs

La similitude entre les deux noms amène souvent une confusion entre le modèle de composants JavaBeans et la spécification des Enterprise JavaBeans. Bien que les spécifications des JavaBeans et des Enterprise JavaBeans partagent toutes deux les même objectifs (comme la mise en avant de la réutilisation et la portabilité du code Java entre développements, comme aussi des outils de déploiement qui utilise des modèles de conception standards), les motivations derrière chaque spécification ont pour but de résoudre des problèmes différents.

Les standards définis dans le modèle de composants JavaBeans sont conçus pour créer des composants réutilisables qui sont typiquement utilisés dans des outils de développement IDE et sont communément, mais pas exclusivement, des composants visuels.

La spécification des Enterprise JavaBeans définit un modèle de composants pour développer du code Java côté serveur. Comme les EJBs peuvent potentiellement fonctionner sur différentes plate-formes serveurs (incluant des systèmes qui n'ont pas d'affichage visuel), un EJB ne peut pas utiliser de librairies graphiques tels que AWT ou Swing.

Que définit la spécification des EJBs ?

La spécification des EJBs, actuellement la version 1.1 (public release 2) définit un modèle de composant côté serveur. Elle définit six rôles qui sont utilisés pour réaliser les tâches de développement et de déploiement ainsi que la définition des composants du système.

Les rôles

La spécification des EJBs définit des rôles qui sont utilisés durant le développement, le déploiement et le fonctionnement d'un système distribué. Les fabricants, les administrateurs et les développeurs jouent des rôles variés. Ils permettent la séparation du savoir-faire technique, de celui propre au domaine. Ceci permet au fabricant de proposer un framework technique et aux développeurs de créer des composants spécifiques au domaine comme par exemple un composant de compte bancaire. Un même intervenant peut jouer un ou plusieurs rôles. Les rôles définis dans la spécification des EJBs sont résumés dans la table suivante :

Rôle	**Responsabilité**
Fournisseur d'Enterprise Bean	Le développeur qui est responsable de la création des composants EJB réutilisables. Ces composants sont regroupés dans un fichier jar spécial (fichier ejb-jar).
Assembleur d'Application	Crée et assemble des applications à partir d'une collection de fichiers ejb-jar. Ceci inclut la

	réalisation d'applications mettant en oeuvre la collection d'EJB (comme des Servlets, JSP, Swing, etc.).
Déployeur	Le rôle du déployeur est de prendre la collection de fichiers ejb-jar de l'Assembleur et/ou du Fournisseur d'EJB et de déployer ceux-ci dans un environnement d'exécution (un ou plusieurs Conteneurs d'EJBs).
Fournisseur de Conteneurs/Serveurs d'EJB	Fournit un environnement d'exécution et les outils qui sont utilisés pour déployer, administrer et faire fonctionner les composants EJB.
Administrateur Système	Par dessus tout, le rôle le plus important de l'ensemble du système : mettre en place et faire fonctionner. La gestion d'une application distribuée consiste à ce que les composants et les services soient tous configurés et interagissent ensemble correctement.

Composants EJB

Les composants EJB sont de la logique métier réutilisable. Les composants EJB suivent strictement les standards et les modèles de conception définis dans la spécification des EJBs. Cela permet à ces composants d'être portables et aussi à tous les autres services tels que la sécurité, la mise en cache et les transactions distribuées, d'être mis en oeuvre sur ces composants eux-mêmes. Un fournisseur d'Entreprise Bean est responsable du développement des composants EJB. Les entrailles d'un composant EJB sont traités dans *Qu'est-ce qui compose un composant EJB ?*

Conteneur d'EJB

Le conteneur d'EJB est un environnement d'exécution qui contient et fait fonctionner les composants EJB tout en leur fournissant un ensemble de services. Les responsabilités des conteneurs d'EJB sont définies précisément par la spécification pour permettre une neutralité vis-à-vis du fournisseur. Les conteneurs d'EJB fournissent la machinerie bas-niveau des EJBs, incluant les transactions distribuées, la sécurité, la gestion du cycle de vie des beans, la mise en cache, la gestion de la concurrence et des sessions. Le fournisseur de conteneur d'EJB est responsable de la mise à disposition d'un conteneur d'EJB.

Serveur EJB

Un serveur d'EJB est défini comme un Serveur d'Applications et comporte un ou plusieurs conteneurs d'EJBs. Le fournisseur de serveur EJB est responsable de la mise à disposition d'un serveur EJB. Vous pouvez généralement considérer que le conteneur d'EJB et le serveur EJB sont une seule et même chose.

Java Naming and Directory Interface (JNDI)

Java Naming and Directory Interface (JNDI) est utilisé dans les Entreprise JavaBeans comme

le service de nommage pour les composants EJB sur le réseau et pour les autres services du conteneur comme les transactions. JNDI ressemble fort aux autres standards de nommage et de répertoires tels que CORBA CosNaming et peut être implémenté comme un adaptateur de celui-ci.

Java Transaction API / Java Transaction Service (JTA/JTS)

JTA/JTS est utilisé dans les Enterprise JavaBeans comme API transactionnelle. Un fournisseur d'Entreprise Bean peut utiliser le JTS pour créer un code transactionnel bien que le conteneur d'EJB implémente généralement les transactions dans les EJBs sur les composants EJBs eux-mêmes. Le déployeur peut définir les attributs transactionnels d'un composant EJB au moment du déploiement. Le conteneur d'EJB est responsable de la prise en charge de la transaction qu'elle soit locale ou distribuée. La spécification du JTS est la version Java de CORBA OTS (Object Transaction Service).

CORBA et RMI/IIOP

La spécification des EJBs définit l'interopérabilité avec CORBA. La spécification 1.1 précise que *L'architecture des Entreprise JavaBeans sera compatible avec les protocoles CORBA*. L'interopérabilité avec CORBA passe par l'adaptation des services EJB comme JTS et JNDI aux services CORBA et l'implémentation de RMI à travers le protocole CORBA IIOP.

L'utilisation de CORBA et de RMI/IIOP dans les Entreprise JavaBeans est implémentée dans le conteneur EJB et est sous la responsabilité du fournisseur du conteneur d'EJB. L'utilisation de CORBA et de RMI/IIOP dans le conteneur EJB est invisible pour le composant EJB lui-même. Cela signifie que le fournisseur d'Entreprise Bean peut écrire ses composants EJBs et les déployer dans un conteneur EJB sans s'inquiéter du protocole de communication qui est utilisé.

Qu'est-ce qui compose un composant EJB ?

Un EJB se décompose en un ensemble de pièce, dont le Bean lui-même, l'implémentation d'interfaces et un fichier d'informations. Le tout est rassemblé dans un fichier jar spécial.

Enterprise Bean

L'Enterprise Bean est une classe Java que le fournisseur d'Enterprise Bean développe. Elle implémente une interface EnterpriseBean (pour plus de détails, voir la section qui suit) et fournit l'implémentation des méthodes métier que le composant supporte. La classe n'implémente aucun mécanisme d'autorisation ou d'authentification, de concurrence ou transactionnel.

Interface Home

Chaque Enterprise Bean créé doit être associé à une interface Home. L'interface Home est utilisée comme une Factory de votre EJB. Les clients utilisent l'interface Home pour trouver une instance de votre EJB ou pour créer une nouvelle instance de votre EJB.

Interface Remote

L'interface Remote est l'interface Java qui reflète les méthodes de l'Enterprise Bean que l'on souhaite rendre disponible au monde extérieur. L'interface Remote joue un rôle similaire à celui de l'interface IDL de CORBA.

Descripteur de Déploiement

Le descripteur de Déploiement est un fichier XML qui contient les informations relatives à l'EJB. L'utilisation d'XML permet au Déployeur de facilement changer les attributs propres à l'EJB. Les attributs configurables définis dans le descripteur de déploiement incluent :

- Les noms des interfaces Home et Remote que nécessite l'EJB ;
- Le nom avec lequel sera publiée dans JNDI l'interface Home de l'EJB ;
- Les attributs transactionnels pour chaque méthode de l'EJB ;
- Les listes de contrôle d'accès (Access Control Lists) pour l'authentification.

Fichier EJB-Jar

Le fichier EJB-Jar est un fichier jar Java normal qui contient un EJB, les interface Home et Remote ainsi que le descripteur de déploiement.

Comment travaille un EJB ?

Maintenant que l'on a un fichier EJB-Jar contenant un Bean, les interfaces Home et Remote, et un descripteur de déploiement, voyons un peu comment toutes ces pièces vont ensemble, pourquoi les interfaces Home et Remote sont nécessaires et comment le conteneur d'EJB les utilise.

Le conteneur d'EJB implémente les interfaces Home et Remote qui sont dans le fichier EJB-Jar. Comme mentionné précédemment, l'interface Home met à disposition les méthodes pour créer et trouver votre EJB. Cela signifie que le conteneur d'EJB est responsable de la gestion du cycle de vie de votre EJB. Ce niveau d'abstraction permet aux optimisations d'intervenir. Par exemple, cinq clients peuvent demander simultanément la création d'un EJB à travers l'interface Home, le conteneur d'EJB pourrait n'en créer qu'un seule et partager cet EJB entre les cinq clients. Ceci est réalisé à travers l'interface Remote, qui est aussi implémentée par le conteneur d'EJB. L'objet implémentant Remote joue le rôle d'objet proxy vers l'EJB.

Tous les appels de l'EJB sont redirigés à travers le conteneur d'EJB grâce aux interfaces Home et Remote. Cette abstraction explique aussi pourquoi le conteneur d'EJB peut contrôler la sécurité et le comportement transactionnel.

Types d'EJBs

Il doit y avoir une question dans votre tête depuis le dernier paragraphe : partager le même EJB entre les clients peut certainement augmenter les performances, mais qu'en est-il lorsque je souhaite conserver l'état sur le serveur ?

La spécification des Enterprise JavaBeans définissent différents types d'EJBs qui peuvent avoir différentes caractéristiques et adopter un comportement différent. Deux catégories d'EJBs ont été définies dans cette spécification : les Session Beans et les Entity Beans, et chacune de ces catégories a des variantes.

Session Beans

Les Session Beans sont utilisés pour représenter les cas d'utilisations ou des traitements spécifiques du client. Ils représentent les opérations sur les données persistantes, mais non les don-

nées persistantes elles-mêmes. Il y a deux types de Session Beans : non-persistant (Stateless) and persistant (Stateful). Tous les Session Beans doivent implémenter l'interface javax.ejb.SessionBean. Le conteneur d'EJB contrôle la vie d'un Session Bean.

Les Session Beans non-persistants

Les Session Beans non-persistants sont le type d'EJB le plus simple à implémenter. Ils ne conservent aucun état de leur conversation avec les clients entre les invocations de méthodes donc ils sont facilement réutilisables dans la partie serveur et puisqu'ils peuvent être mis en cache, ils supportent bien les variations de la demande. Lors de l'utilisation de Session Beans non-persistants, tous les états doivent être stockés à l'extérieur de l'EJB.

Les Session Beans persistants

Les Session Beans persistants conservent un état entre l'invocation de leurs méthodes (comme vous l'aviez probablement compris). Ils ont une association 1-1 avec un client et sont capables de conserver leurs états eux-mêmes. Le conteneur d'EJBs a en charge le partage et la mise en cache des Session Beans persistants, ceux-ci passe par la Passivation et l'Activation.

Entity Beans

Les Entity Beans sont des composants qui représentent une donnée persistante et le comportement de cette donnée. Les Entity Beans peuvent être partagés par plusieurs clients, comme une donnée d'une base. Le conteneur d'EJBs a en charge de mettre en cache les Entity Beans et de maintenir leur intégrité. La vie d'un Entity Bean est supérieur à celle du conteneur d'EJBs, donc si un conteneur tombe en panne, l'Entity Bean est censé être encore disponible lors que le conteneur le devient à nouveau.

Il y a deux types d'Entity Beans, ceux dont la persistence est assurée par le Bean lui-même et ceux dont la persistance est assurée par le conteneur.

Un CMP Entity Bean a sa persistence assurée par le conteneur d'EJBs. A travers les attributs spécifiés dans le descripteur de déploiement, le conteneur d'EJBs fera correspondre les attributs de l'Entity Bean avec un stockage persistant (habituellement, mais pas toujours, une base de données). La gestion de la persistence par le conteneur réduit le temps de développement et réduit considérablement le code nécessaire pour l'EJB.

Gestion de la persistence par le Bean (BMP - Bean Managed Persistence)

Un BMP Entity Bean a sa persistence implémentée par le fournisseur de l'Entreprise Bean. Le fournisseur d'Entity Bean a en charge d'implémenter la logique nécessaire pour créer un nouvel EJB, mettre à jour certains attributs des EJBs, supprimer un EJB et trouver un EJB dans le stockage persistence. Cela nécessite habituellement d'écrire du code JDBC pour interagir avec une base de données ou un autre stockage persistant. Avec la gestion de persistence par le Bean (BMP), le développeur a le contrôle total de la manière dont la persistence de l'Entity Bean est réalisée.

Le principe de BMP apporte aussi de la flexibilité là où l'implémentation en CMP n'est pas possible, par exemple si vous souhaitez créer un EJB qui encapsule du code d'un système mainframe existant, vous pouvez écrire votre persistence en utilisant CORBA.

Développer un Enterprise Java Bean

Nous allons maintenant implémenter l'exemple de Perfect Time de la précédente section à propos de RMI sous forme d'un composant Entreprise JavaBean. Cet exemple est un simple Session Bean non-persistant. Les composants Enterprise JavaBean représenteront toujours au moins à une classe et deux interfaces.

La première interface définie est l'interface Remote de notre composant Enterprise JavaBean. Lorsque vous créez votre interface Remote de votre EJB, vous devez suivre ces règles :

1. L'interface Remote doit être **public**.
2. L'interface Remote doit hériter de l'interface **javax.ejb.EJBObject**.
3. Chaque méthode de l'interface Remote doit déclarer **java.rmi.RemoteException** dans sa section **throws** en addition des exceptions spécifiques à l'application.
4. Chaque objet passé en argument ou retourné par valeur (soit directement soit encapsulé dans un objet local) doit être un type de donnée valide pour RMI-IIOP (ce qui inclut les autres objets EJB).

Voici l'interface Remote plutôt simple de notre EJB PerfectTime :

```
//: c15:ejb:PerfectTime.java
//# Vous devez installer le J2EE Java Enterprise
//# Edition de java.sun.com et ajouter j2ee.jar
//# à votre CLASSPATH pour pouvoir compiler
//# ce fichier. Pour plus de détails,
//# reportez vous au site java.sun.com.
//# Interface Remote de PerfectTimeBean
import java.rmi.*;
import javax.ejb.*;

public interface PerfectTime extends EJBObject {
  public long getPerfectTime()
    throws RemoteException;
} ///:~
```

La seconde interface définie est l'interface Home de notre composant Enterprise JavaBeans. L'interface Home est la Factory du composant que vous allez créer. L'interface Home peut définir des méthodes de création ou de recherche. Les méthodes de création créent les instances des EJBs, les méthodes de recherche localisent les EJBs existants et sont utilisés pour les Entity Beans seulement. Lorsque vous créez votre interface Home d'un EJB, vous devez respecter ces quelques règles :

1. L'interface Home doit être **public**.
2. L'interface Home doit hériter de l'interface **javax.ejb.EJBHome**.
3. Chaque méthode de l'interface Home doit déclarer **java.rmi.RemoteException** dans sa section **throws** de même que **javax.ejb.CreateException.**
4. La valeur retournée par une méthode de création doit être une interface Remote.
5. La valeur retournée par une méthode de recherche (pour les Entity Beans uniquement)

doit être une interface Remote, **java.util.Enumeration** ou **java.util.Collection**.

6. Chaque objet passé en argument ou retourné par valeur (sous directement ou encapsulé dans un objet local) doit être un type de donnée valide pour RMI-IIOP (ce qui inclut les autres objets EJB).

La convention standard de nommage des interfaces Home consiste à prendre le nom de l'interface Remote et d'y ajouter à la fin Home. Voici l'interface Home de notre EJB PerfectTime :

```
//: c15:ejb:PerfectTimeHome.java
// Interface Home de PerfectTimeBean.
import java.rmi.*;
import javax.ejb.*;

public interface PerfectTimeHome extends EJBHome {
  public PerfectTime create()
    throws CreateException, RemoteException;
} ///:~
```

Maintenant que nous avons défini les interfaces de notre composant, nous pouvons implémenter la logique métier qu'il y a derrière. Lorsque vous créez la classe d'implémentation de votre EJB, vous devez suivre ces règles (notez que vous pourrez trouver dans la spécification des EJBs la liste complète des règles de développement des Entreprise JavaBeans) :

1. La classe doit être **public**.

2. La classe doit implémenter une interface (soit **javax.ejb.SessionBean**, soit **javax.ejb.EntityBean**).

3. La classe doit définir les méthodes correspondant aux méthodes de l'interface Remote. Notez que la classe n'implémente pas l'interface Remote, c'est le miroir des méthodes de l'interface Remote mais elle n'émet pas **java.rmi.RemoteException**.

4. Définir une ou plusieurs méthodes **ejbCreate( )** qui initialisent votre EJB.

5. La valeur retournée et les arguments de toutes les méthodes doivent être des types de données compatibles avec RMI-IIOP.

```
//: c15:ejb:PerfectTimeBean.java
// Un Session Bean non-persistant
// qui retourne l'heure système courante.
import java.rmi.*;
import javax.ejb.*;

public class PerfectTimeBean
  implements SessionBean {
  private SessionContext sessionContext;
  // retourne l'heure courante
  public long getPerfectTime() {
    return System.currentTimeMillis();
  }
  // méthodes EJB
  public void
```

```
  ejbCreate() throws CreateException {}
  public void ejbRemove() {}
  public void ejbActivate() {}
  public void ejbPassivate() {}
  public void
  setSessionContext(SessionContext ctx) {
    sessionContext = ctx;
  }
}///:~
```

Notez que les méthodes EJB (**ejbCreate( )**, **ejbRemove( )**, **ejbActivate( )**, **ejbPassivate( )**) sont toutes vides. Ces méthodes sont appelées par le conteneur d'EJBs et sont utilisées pour contrôler l'état de votre composant. Comme c'est un exemple simple, nous pouvons les laisser vide. La méthode **setSessionContext( )** transmet un objet javax.ejb.SessionContext qui contient les informations concernant le contexte dans lequel se trouve le composant, telles que la transaction courante et des informations de sécurité.

Après que nous ayons créé notre Enterprise JavaBean, nous avons maintenant besoin de créer le descripteur de déploiement. Dans les EJBs 1.1, le descripteur de déploiement est un fichier XML qui décrit le composant EJB. Le descripteur de déploiement doit être stocké dans un fichier appelé **ejb-jar.xml**.

```
<?xml version="1.0" encoding="Cp1252"?>
<!DOCTYPE ejb-jar PUBLIC '-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans
1.1//EN' 'http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd'>

<ejb-jar>
  <description>Exemple pour le chapitre 15</description>
  <display-name></display-name>
  <small-icon></small-icon>
  <large-icon></large-icon>
  <enterprise-beans>
    <session>
      <ejb-name>PerfectTime</ejb-name>
      <home>PerfectTimeHome</home>
      <remote>PerfectTime</remote>
      <ejb-class>PerfectTimeBean</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
    </session>
  </enterprise-beans>
  <ejb-client-jar></ejb-client-jar>
</ejb-jar>
```

Dans la balise **<session>** de votre descripteur de déploiement, vous pouvez voir que le composant, les interfaces Remote et Home sont définis en partie. Les descripteurs de déploiement peuvent facilement être générés automatiquement grâce à des outils tels que JBuilder.

Par l'intermédiaire de ce descripteur de déploiement standard **ejb-jar.xml**, la spécification des

EJBs 1.1 institue que toutes balises spécifiques aux fournisseurs doivent être stockées dans un fichier séparé. Ceci a pour but d'assurer une compatibilité complète entre les composants et les conteneurs d'EJBs de différents marques.

Maintenant que nous avons créé notre composant et défini sa composition dans le descripteur de déploiement, nous devons alors archiver les fichiers dans un fichier archive Java (JAR). Le descripteur de déploiement doit être placé dans le sous répertoire **META-INF** du fichier Jar.

Une fois que nous avons défini notre composant EJB dans le descripteur de déploiement, le déployeur doit maintenant déployer le composant EJB dans le conteneur d'EJB. A ce moment-là du développement, le processus est plutôt orienté IHM et spécifique à chaque conteneur d'EJBs. Nous avons donc décidé de ne pas documenter entièrement le processus de déploiement dans cette présentation. Chaque conteneur d'EJBs propose cependant à un processus détaillé pour déployer un EJB.

Puisqu'un composant EJB est un objet distribué, le processus de déploiement doit créer certains stubs clients pour appeler le composant EJB. Ces classes seront placées dans le classpath de l'application cliente. Puisque les composants EJB sont implémentés par dessus RMI-IIOP (CORBA) ou RMI-JRMP, les stubs générés peuvent varier entre les conteneurs d'EJBs, néanmoins ce sont des classes générées.

Lorsqu'un programme client désire invoquer un EJB, il doit rechercher le composant EJB dans JNDI et obtenir une référence vers l'interface Home du composant EJB. L'interface HOME peut alors être invoquée pour créer une instance de l'EJB, qui peut à son tour être invoquée.

Dans cet exemple le programme client est un simple programme Java, mais vous devez garder en mémoire qu'il pourrait s'agir aussi bien d'un Servlet, d'un JSP que d'un objet distribué CORBA ou RMI.

Le code de PerfectTimeClient code est le suivant.

```
//: c15:ejb:PerfectTimeClient.java
// Programme Client pour PerfectTimeBean

public class PerfectTimeClient {
public static void
main(String[ « args) throws Exception {
  // Obtient un context JNDI utilisant le
  // service de nommage JNDI :
  javax.naming.Context context =    new javax.naming.InitialContext();
  // Recherche l'interface Home dans le
  // service de nommage JNDI :
  Object ref = context.lookup("perfectTime");
  // Transforme l'objet distant en une interface Home :
  PerfectTimeHome home = (PerfectTimeHome)
    javax.rmi.PortableRemoteObject.narrow(
      ref, PerfectTimeHome.class);
  // Crée un objet distant depuis l'interface Home :
  PerfectTime pt = home.create();
  // Invoque  getPerfectTime()
  System.out.println(
    "Perfect Time EJB invoked, time is: " +
```

```
      pt.getPerfectTime() );
    }
  } ///:~
```

Le déroulement de cet exemple est expliqué dans les commentaires. Notez l'utilisation de la méthode narrow() pour réaliser une sorte de transtypage de l'objet avant qu'un transtypage Java soit fait. Ceci est très similaire à ce qui se passe en CORBA. Notez aussi que l'objet Home devient une Factory pour les objets PerfectTimes.

En résumé

La spécification des Enterprise JavaBeans est un pas important vers la standardisation et la simplification de l'informatique distribuée. C'est une pièce majeure de Java 2, Enterprise Edition Platform et reçoit en plus le concours la communauté travaillant sur les objets distribués. De nombreux outils sont actuellement disponibles ou le seront dans un futur proche pour accélérer le développement de composants EJBs.

Cette présentation avait pour but de donner un bref aperçu de ce que sont les EJBs. Pour plus d'informations à propos de la spécification des Enterprise JavaBeans, vous pouvez vous reporter à la page officielle des Enterprise JavaBeans à l'adresse *http://java.sun.com/products/ejb/*. Vous pourrez y télécharger la dernière spécification ainsi que Java 2, Enterprise Edition Reference Implementation, qui vous permettra de développer et de déployer vos propres composants EJBs.

Jini : services distribués

Cette section [71]donne un aperçu de la technologie Jini de Sun Microsystem. Elle décrit quelques spécificités Jini et montre comment l'architecture Jini aide à augmenter le niveau d'abstraction dans la programmation de systèmes distribués, transformant réellement la programmation réseau en programmation orientée-objets.

Contexte de Jini

Traditionnellement, les systèmes d'exploitation sont conçus dans l'hypothèse qu'un ordinateur aura un processeur, un peu de mémoire et un disque. Lorsque vous démarrez votre ordinateur, la première chose qu'il fait est de chercher un disque. S'il ne le trouve pas, il ne peut assurer sa fonction d'ordinateur. Cependant de plus en plus les ordinateurs apparaissent sous diverses formes : comme des systèmes embarqués qui possèdent un processeur, un peu de mémoire et une connexion réseau mais pas de disque. La première chose que fait, par exemple, un téléphone cellulaire lorsqu'il s'allume est de rechercher le réseau téléphonique. S'il ne le trouve pas, il ne peut assurer sa fonction de téléphone. Cette nouvelle mode dans l'environnement matériel, le passage d'un système centré sur un disque à un système centré sur un réseau, va affecter la manière d'organiser notre logiciel. C'est là qu'intervient Jini.

Jini est une manière de repenser l'architecture de l'ordinateur, étant donné l'importance croissante des réseaux et la prolifération des processeurs dans des systèmes qui n'ont pas de disque dur. Ces systèmes, qui proviennent de nombreux fabricants différents, vont avoir besoin d'interagir à travers le réseau. Le réseau lui-même sera très dynamique : les systèmes et les services seront ajoutés et retirés régulièrement. Jini apporte les mécanismes permettant facilement l'ajout, la suppression et la recherche de systèmes et de services sur le réseau. De plus, Jini propose un modèle de program-

mation qui rend tout cela plus facile pour les programmeurs qui souhaitent voir leurs systèmes discuter entre eux.

S'appuyant sur Java, la sérialisation objet et RMI (qui permet aux objets de bouger à travers le réseau en passant d'une machine virtuelle à une autre), Jini permet d'étendre les bénéfices de la programmation orientée-objet au réseau. Au lieu de nécessiter un accord entre les différents fabricants sur un protocole réseau à travers lequel les systèmes peuvent interagir, Jini permet à ces systèmes de discuter ensemble par l'intermédiaire d'interfaces vers des objets.

Qu'est-ce que Jini ?

Jini est un ensemble d'APIs et de protocoles réseaux qui peuvent vous aider à construire et déployer des systèmes distribués qui sont organisés sous la forme de *fédérations de services* . Un *service* peut être n'importe quoi qui se trouve sur le réseau et qui est prêt à réaliser une fonction utile. Des composants matériels, logiciels, des canaux de communications, les utilisateurs eux-mêmes peuvent être des services. Une imprimante compatible Jini pourra offrir un service d'impression. Une fédération de services est un ensemble de services, actuellement disponibles sur le réseau, que le client (ce qui signifie programme, service ou utilisateur) peut combiner pour s'aider à atteindre à son but.

Pour réaliser une tâche, un client enchaîne les possibilités des services. Par exemple, un programme client peut charger des photographies d'un système de stockage d'image d'un appareil numérique, envoyer les photos vers un service de stockage persistant offert par un disque dur, et transmettre une page contenant les vignettes de chaque image à un service d'impression d'une imprimante couleur. Dans cet exemple, le programme client construit un système distribué constitué de lui-même, le service de stockage d'images, le service de stockage persistant et le service d'impression couleur. Le client et ces services de ce système distribué collaborent pour réaliser une tâche : décharger et stocker les images d'un appareil numérique et imprimer une page de vignettes.

L'idée derrière le mot *fédération* est que la vision Jini d'un réseau n'instaure pas d'autorité de contrôle centrale. Puisque aucun service n'est responsable, l'ensemble de tous les services disponible sur le réseau forme une fédération, un groupe composé de membres égaux. Au lieu d'une autorité centrale, l'infrastructure d'exécution de Jini propose un moyen pour les clients et les services de se trouver mutuellement (à travers un service de recherche, qui stocke la liste des services disponibles à moment donné). Après que les services se sont trouvés, ils sont indépendants. Le client et ces services mis à contribution réalisent leurs tâches indépendamment de l'infrastructure d'exécution de Jini. Si le service de recherche Jini tombe, tous les systèmes distribués mis en place par le service de recherche, avant qu'il ne plante, peuvent continuer les travaux. Jini incorpore même un protocole réseau qui permet aux clients de trouver les services en l'absence d'un service de nommage.

Comment fonctionne Jini

Jini définie une *infrastructure d'exécution* qui réside sur le réseau et met à disposition des mécanismes qui vous permettent d'ajouter, d'enlever, de localiser et d'accéder aux services. L'infrastructure d'exécution se situe à trois endroits : dans les services de recherche qui sont sur le réseau, au niveau des fournisseurs de service (tels que les systèmes supportant Jini), et dans les clients. *Les services de recherche* forment le mécanisme centralisé d'organisation des systèmes basés sur Jini. Lorsque des services deviennent disponibles sur le réseau, ils s'enregistrent eux-même grâce à un service de recherche. Lorsque des clients souhaitent localiser un service pour être assistés dans leur travail, ils consultent le service de recherche.

L'infrastructure d'exécution utilise un protocole au niveau réseau, appelé *discovery (découverte)*, et deux protocoles au niveau objet appelés *join (joindre)* et *lookup (recherche)*. Discovery permet aux clients et aux services de trouver les services de recherche. Join permet au service de s'enregistrer lui-même auprès du service de recherche. Lookup permet à un client de rechercher des services qui peuvent l'aider à atteindre ses objectifs.

Le processus de découverte

Le processus de découverte travaille ainsi : imaginez un disque supportant Jini et offrant un service de stockage persistant. Dès que le disque est connecté au réseau, il diffuse une *annonce de présence* en envoyant un paquet multicast sur un port déterminé. Dans l'annonce de présence, sont inclus une adresse IP et un numéro de port où le disque peut être contacté par le service de recherche.

Les services de recherche scrutent sur le port déterminé les paquets d'annonce de présence. Lorsqu'un service de recherche reçoit une annonce de présence, il l'ouvre et inspecte le paquet. Le paquet contient les informations qui permet au service de recherche de déterminer s'il doit ou non contacter l'expéditeur de ce paquet. Si tel est le cas, il contacte directement l'expéditeur en établissant une connexion TCP à l'adresse IP et sur le numéro de port extraits du paquet. En utilisant RMI, le service de recherche envoie à l'initiateur du paquet un objet appelé un *enregistreur de service (service registrar)*. L'objectif de cet enregistreur de service est de faciliter la communication future avec le service de recherche. Dans le cas d'un disque dur, le service de recherche attablirait une connexion TCP vers le disque dur et lui enverrait un *enregistreur de service*, grâce auquel le disque dur pourra faire enregistrer son service de stockage persistant par le processus de jonction.

Dès lors qu'un fournisseur de service possède un *enregistreur de service*, le produit final du processus de découverte, il est prêt à entreprendre une jonction pour intégrer la fédération des services qui sont enregistrés auprès du service de recherche. Pour réaliser une jonction, le fournisseur de service fait appel à la méthode **register( )**de l' "font-weight: medium">*enregistreur de service*, passant en argument un objet appelé élément de service (service item), un ensemble d'objets qui décrit le service. La méthode **register( )**envoie une copie de cet élément de service au service de recherche, où celui-ci sera stocké. Lorsque ceci est achevé, le fournisseur de service a fini le processus de jonction : son service est maintenant enregistré auprès du service de recherche.

L'élément de service contient plusieurs objets, parmi lesquels un objet appelé un *objet service*, que les clients utilisent pour interagir avec le service. L'élément de service peut aussi inclure une certain nombre d'*attributs*, qui peuvent être n'importe quel objet. Certains de ces attributs sont des icônes, des classes qui fournissent des interfaces graphiques pour le service et des objets apportant plus de détails sur le service.

Les objects service implémentent généralement une ou plusieurs interfaces à travers lesquelles les clients interagissent avec le service. Par exemple, le service de recherche est un service Jini, et son objet service est un service de registre. La méthode **register( )**appelée par les fournisseurs de service durant la jonction est déclarée dans l'interface **ServiceRegistrar**(un membre du package **net.jini.core.lookup**) que tous les services de registre implémentent. Les clients et les fournisseurs de registre discutent avec le service de recherche à travers l'objet de service de registre en invoquant les méthodes déclarées dans l'interface **ServiceRegistrar**. De la même manière, le disque dur fournit un objet service qui implémente l'une des interfaces connues de service de stockage. Les clients peuvent rechercher le disque dur et interagir avec celui-ci par cette interface de service de stockage.

Le processus de recherche

Une fois qu'un service a été enregistré par un service de recherche grâce au processus de jonction, ce service est utilisable par les clients qui le demandent au service de recherche. Pour construire un système distribué de services qui collaborent pour réaliser une tâche, un client doit localiser ses services et s'aider de chacun d'eux. Pour trouver un service, les clients formulent des requêtes auprès des services de recherche par l'intermédiaire d'un processus appelé *recherche*.

Pour réaliser une recherche, un client fait appel à la méthode **lookup**( ) d'un service de registre (comme un fournisseur de service, un client obtient un service de registre grâce au processus de découverte décrit précédemment). Le client passe en argument un modèle de service à **lookup( )**, un objet utilisé comme critère de recherche. Le modèle de service peut inclure une référence à un tableau d'objets **Class**. Ces objets **Class**indiquent au service de recherche le type Java (ou les types)) de l'objet service voulu par le client. Le modèle de service peut aussi inclure un *service ID*, qui identifie de manière unique le service, ainsi que des attributs, qui doivent correspondre exactement aux attributs fournis par le fournisseur de service dans l'élément de service. Le modèle de service peut aussi contenir des critères génériques pour n'importe quel attribut. Par exemple, un critère générique dans le champ service ID correspondra à n'importe quel service ID. La méthode **lookup**( ) envoie le modèle de service au service de recherche, qui exécute la requête et renvoie s'il y en a les objets services correspondants. Le client récupère une référence vers ces objets services comme résultat de la méthode **lookup( )**.

En général, un client recherche un service selon le type Java, souvent une interface. Par exemple, si un client avait besoin d'utiliser une imprimante, il pourrait créer un modèle de service qui comprend un objet **Class** d'une interface connue de services d'impression. Tous les services d'impression implémenteraient cette interface connue. Le service de recherche retournerait un ou plusieurs objets services qui implémentent cette interface. Les attributs peuvent être inclus dans le modèle de service pour réduire le nombre de correspondances de ce genre de recherche par type. Le client pourrait utiliser le service d'impression en invoquant sur l'objet service les méthodes définies dans l'interface.

Séparation de l'interface et de l'implémentation

L'architecture Jini met en place pour le réseau une programmation orientée-objet en permettant aux services du réseau de tirer parti de l'un des fondements des objets : la séparation de l'interface et l'implémentation. Par exemple, un objet service peut permettre aux clients d'accéder au service de différentes manières. L'objet peut réellement représenter le service entier, qui sera donc téléchargé par le client lors de la recherche et exécuté localement ensuite. Autrement, l'objet service peut n'être qu'un proxy vers un serveur distant. Lorsqu'un client invoque des méthodes de l'objet service, il envoie les requêtes au serveur à travers le réseau, qui fait réellement le travail. Une troisième option consiste à partager le travail entre l'objet service local et le serveur distant.

Une conséquence importante de l'architecture Jini est que le protocole réseau utilisé pour communiquer entre l'objet service proxy et le serveur distant n'a pas besoin d'être connu du client. Comme le montre la figure suivante, le protocole réseau est une partie de l'implémentation du service. Ce protocole est une question privée prise en compte par le développeur du service. Le client peut communiquer avec l'implémentation du service à travers un protocole privée car le service injecte un peu de son propre code (l'objet service) au sein de l'espace d'adressage du client. L'objet service ainsi injecté peut communiquer avec le service à travers RMI, CORBA, DCOM, un protocole fait maison construit sur des sockets et des flux ou n'importe quoi d'autre. Le client ne porte

simplement aucune attention quant aux protocoles réseau, puisqu'il ne fait que communiquer avec l'interface publique que le service implémente. L'objet service prend en charge toutes les communications nécessaires sur le réseau.

Le client communique avec le service à travers une interface publique

Différentes implémentations de la même interface d'un service peuvent utiliser des approches et des protocoles réseau totalement différents. Un service peut utiliser un matériel spécialisé pour répondre aux requêtes clientes, ou il peut tout réaliser de manière logicielle. En fait, le choix d'implémentation d'un même service peut évoluer dans le temps. Le client peut être sûr qu'il possède l'objet service qui comprend l'implémentation actuelle de ce service, puisque le client reçoit l'objet service (grâce au service de recherche) du fournisseur du service lui-même. Du point de vue du client, le service ressemble à une interface publique, sans qu'il ait à se soucier de l'implémentation du service.

Abstraction des systèmes distribués

Jini tente d'élever passer le niveau d'abstraction de la programmation de systèmes distribués, passant du niveau du protocole réseau à celui de l'interface objet. Dans l'optique d'une prolifération des systèmes embarqués connectés au réseau, beaucoup de pièces d'un système distribué pourront venir de fournisseurs différents. Jini évite aux fournisseurs de devoir se mettre d'accord sur les protocoles réseau qui permettent à leurs systèmes d'interagir. A la place, les fournisseurs doivent se mettre d'accord sur les interfaces Java à travers lesquelles leurs systèmes peuvent interagir. Les processus de découverte, de jonction et de recherche, fournis par l'infrastructure d'exécution de Jini, permettront aux systèmes de se localiser les uns les autres sur le réseau. Une fois qu'ils se sont localisés, les systèmes sont capables de communiquer entre eux à travers des interfaces Java.

Résumé

Avec Jini pour des réseaux de systèmes locaux, ce chapitre vous a présenté une partie (une partie seulement) des composants que Sun regroupe sous le terme de J2EE : the Java 2 Enterprise Edition. Le but de J2EE est de construire un ensemble d'outils qui permettent au développeur Java de construire des applications serveurs beaucoup plus rapidement et indépendamment de la plate-forme. Construire de telles applications n'est pas seulement difficile et coûteux en temps, mais il est particulièrement dur de les construire en faisant en sorte qu'elles puissent être facilement portées sur une autre plate-formes, et aussi que la logique métier soit isolée des détails relevant de l'implémentation. J2EE met à disposition une structure pour assister la création d'applications serveurs ; ces applications sont très demandées en ce moment, et cette demande semble grandir.

Exercices

On triouvera les solutions des exercices sélectionnés dans le document électronique *The Thinking in Java Annotated Solution Guide*, disponible pour une participation minime à *www.BruceEckel.com*.

1. Compiler et lancer les programmes **JabberServer** et **JabberClient** de ce chapitre. Éditer ensuite les fichiers pour supprimer les « buffering » d'entrée et de sortie, compiler et relancer, observer le résultat.
2. Créer un serveur qui demande un mot de passe avant d'ouvrir un fichier et de l'en-

voyer sur la connexion réseau. Créer un client qui se connecte à ce serveur, donne le mot de passe requis, puis capture et sauve le fichier. Tester la paire de programmes sur votre machine en utilisant **localhost** (l'adresse IP de boucle locale **127.0.0.1** obtenue en appelant **InetAddress.getByName(null)**).

3. Modifier le serveur de l' Exercice 2 afin qu'il utilise le multithreading pour servir plusieurs clients.

4. Modifier **JabberClient.java** afin qu'il n'y ait pas de vidage du tampon de sortie, observer le résultat.

5. Modifier **MultiJabberServer** afin qu'il utilise la technique de « surveillance de thread » « *thread pooling* ». Au lieu que le thread se termine lorsqu'un client se déconnecte, il intègre de lui-même un « pool « de threads disponibles ». Lorsqu'un nouveau client demande à se connecter, le serveur cherche d'abord dans le pool un thread existant capable de traiter la demande, et s'il n'en trouve pas, en crée un. De cette manière le nombre de threads nécessaires va grossir naturellement jusqu'à la quantité maximale nécessaire. L'intérêt du « thread pooling » est d'éviter l'overhead engendré par la création et la destruction d'un nouveau thread pour chaque client.

6. À partir de **ShowHTML.java**, créer une applet qui fournisse un accès protégé par mot de passe à un sous-ensemble particulier de votre site Web.

7. Modifier **CIDCreateTables.java** afin qu'il lise les chaînes SQL depuis un fichier texte plutôt que depuis **CIDSQL**.

8. Configurer votre système afin d'exécuter avec succès **CIDCreateTables.java** et **LoadDB.java**.

9. Modifier **ServletsRule.java** en surchargeant la méthode **destroy( )** afin qu'elle sauvegarde la valeur de **i** dans un fichier, et la méthode **init( )** pour qu'elle restaure cette valeur. Montrer que cela fonctionne en rechargeant le conteneur de servlet. Si vous ne possédez pas de conteneur de servlet, il vous faudra télécharger, installer, et exécuter Tomcat depuis *jakarta.apache.org* afin de travailler avec les servlets.

10. Créer une servlet qui ajoute un cookie à l'objet réponse, lequel sera stocké sur le site client. Ajouter à la servlet le code qui récupère et affiche le cookie. Si vous n'avez pas de conteneur de servlet, il vous faudra télécharger, installer, et exécuter Tomcat depuis *jakarta.apache.org* afin de travailler avec les servlets.

11. Créer une servlet utilisant un objet **Session** stockant l'information de session de votre choix. Dans la même servlet, récupérer et afficher cette information de session. Si vous ne possédez pas de conteneur de servlet, il vous faudra télécharger, installer, et exécuter Tomcat depuis *jakarta.apache.org* afin de travailler avec les servlets.

12. Créer une servlet qui change la valeur de « inactive interval » de l'objet session pour la valeur 5 secondes en appelant **getMaxInactiveInterval( )**. Tester pour voir si la session se termine naturellement après 5 secondes. Si vous n'avez pas de conteneur de servlet, il vous faudra télécharger, installer, et exécuter Tomcat depuis *jakarta.apache.org* afin de travailler avec les servlets.

13. Créer une page JSP qui imprime une ligne de texte en utilisant le tag <H1>. Générer la couleur de ce texte aléatoirement, au moyen du code Java inclus dans la page JSP. Si vous ne possédez pas de conteneur JSP, il vous faudra télécharger, installer, et exécuter Tomcat

depuis *jakarta.apache.org* afin de travailler avec JSP.

14. Modifier la date d'expiration dans **Cookies.jsp** et observer l'effet avec deux navigateurs différents. Constater également les différences entre le fait de visiter à nouveau la même page, et celui de fermer puis réouvrir le navigateur. Si vous ne possédez pas de conteneur JSP, il vous faudra télécharger, installer, et exécuter Tomcat depuis *jakarta.apache.org* afin de travailler avec JSP.

15. Créer une page JSP contenant un champ autorisant l'utilisateur à définir l'heure de fin de session ainsi qu'un second champ contenant les données stockées dans cette session. Le bouton de soumission rafraîchit la page, prend les valeurs courantes de l'heure de fin et les données de la session, et les garde en tant que valeurs par défaut des champs susmentionnés. Si vous ne possédez pas de conteneur JSP, il vous faudra télécharger, installer, et exécuter Tomcat depuis *jakarta.apache.org* afin de travailler avec JSP.

16. (Encore plus difficile). Modifier le programme **VLookup.java** de telle manière qu'un clic sur le nom résultat copie automatiquement ce nom dans les presse-papier (ce qui vous permet de le coller facilement dans votre email). Vous aurez peut-être besoin de revenir en arrière sur le chapitre 13 pour vous remémorer l'utilisation du presse-papier dans les JFC.

[68]Ce qui représente un peu plus de 4 milliards de valeurs, ce qui sera vite épuisé. Le nouveau standard pour les adresses IP utilisera des nombres de 128 octets, qui devraient produire assez d'adresses IP pour le futur proche.

[69]Beaucoup de neurones sont morts après une atroce agonie pour découvrir cette information.

[70]Cette section a été réalisée par Robert Castaneda, avec l'aide de Dave Bartlett.

[71]Cette section a été réalisée par Bill Venners (www.artima.com).

[72]Cela signifie un nombre maximum légèrement supérieur à quatre milliards, ce qui s'avère rapidement insuffisant. Le nouveau standard des adresses IP utilisera un nombre représenté sur 128 bits, ce qui devrait fournir suffisamment d'adresses IP uniques pour le futur prévisible.

[73]Créé par Dave Bartlett.

[74]Dave Bartlett participa activement à ce développement, ainsi qu'à la section JSP.

[75]« A primary tenet of Extreme Programming (XP) ». Voir *www.xprogramming.com*.

Annexe A- Passage et Retour d'Objets

Vous devriez maintenant être conscient que lorsque vous « passez » un objet, vous passez en fait une référence sur cet objet.

Presque tous les langages de programmation possèdent une façon « normale » de passer des objets, et la plupart du temps tout se passe bien. Mais il arrive toujours un moment où on doit faire quelque chose d'un peu hors-norme, et alors les choses se compliquent un peu (voire beaucoup dans le cas du C++). Java ne fait pas exception à la règle, et il est important de comprendre exactement les mécanismes du passage d'arguments et de la manipulation des objets passés. Cette annexe fournit des précisions quant à ces mécanismes.

Ou si vous préférez, si vous provenez d'un langage de programmation qui en disposait, cette annexe répond à la question « Est-ce que Java utilise des pointeurs ? ». Nombreux sont ceux qui ont affirmé que les pointeurs sont difficiles à manipuler et dangereux, donc à proscrire, et qu'en tant que langage propre et pur destiné à alléger le fardeau quotidien de la programmation, Java ne pouvait décemment contenir de telles choses. Cependant, il serait plus exact de dire que Java dispose de pointeurs ; en fait, chaque identifiant d'objet en Java (les scalaires exceptés) est un pointeur, mais leur utilisation est restreinte et surveillée non seulement par le compilateur mais aussi par le système d'exécution. Autrement dit, Java utilise les pointeurs, mais pas les pointeurs arithmétiques. C'est ce que j'ai appelé les « références » ; et vous pouvez y penser comme à des « pointeurs sécurisés », un peu comme des ciseaux de cours élémentaire - ils ne sont pas pointus, on ne peut donc se faire mal avec qu'en le cherchant bien, mais ils peuvent être lents et ennuyeux.

Passage de références

Quand on passe une référence à une méthode, on pointe toujours sur le même objet. Un simple test le démontre :

```
//: appendixa:PassReferences.java
// Le passage de références.

public class PassReferences {
  static void f(PassReferences h) {
    System.out.println("h inside f(): " + h);
  }
  public static void main(String[] args) {
    PassReferences p = new PassReferences();
    System.out.println("p inside main(): " + p);
    f(p);
  }
} ///:~
```

La méthode **toString()** est automatiquement appelée dans l'instruction print, dont **PassReferences** hérite directement de **Object** comme la méthode **toString()** n'est pas redéfinie. La version **toString()** de **Object** est donc utilisée, qui affiche la classe de l'objet suivie de l'adresse mémoire où se trouve l'objet (non pas la référence, mais bien là où est stocké l'objet). La sortie ressemble à ceci :

```
p inside main(): PassReferences@1653748
h inside f(): PassReferences@1653748
```

On peut constater que **p** et **h** référencent bien le même objet. Ceci est bien plus efficace que de créer un nouvel objet **PassReferences** juste pour envoyer un argument à une méthode. Mais ceci amène une importante question.

Aliasing

L'aliasing veut dire que plusieurs références peuvent être attachées au même objet, comme dans l'exemple précédent. Le problème de l'aliasing survient quand quelqu'un *modifie* cet objet. Si les propriétaires des autres références ne s'attendent pas à ce que l'objet change, ils vont avoir des surprises. Ceci peut être mis en évidence avec un simple exemple :

```
//: appendixa:Alias1.java
// Aliasing : deux références sur un même objet.

public class Alias1 {
  int i;
  Alias1(int ii) { i = ii; }
  public static void main(String[] args) {
    Alias1 x = new Alias1(7);
    Alias1 y = x; // Assigne la référence.
    System.out.println("x: " + x.i);
    System.out.println("y: " + y.i);
    System.out.println("Incrementing x");
    x.i++;
    System.out.println("x: " + x.i);
    System.out.println("y: " + y.i);
  }
} ///:~
```

Dans la ligne :

```
Alias1 y = x; // Assigne la référence.
```

une nouvelle référence **Alias1** est créée, mais au lieu de se voir assigner un nouvel objet créé avec **new**, elle reçoit une référence existante. Le contenu de la référence **x**, qui est l'adresse de l'objet sur lequel pointe **x**, est assigné à **y** ; et donc **x** et **y** sont attachés au même objet. Donc quand on incrémente le **i** de **x** dans l'instruction :

```
x.i++
```

le **i** de **y** sera modifié lui aussi. On peut le vérifier dans la sortie :

```
x: 7
y: 7
Incrementing x
x: 8
```

```
y: 8
```

Une bonne solution dans ce cas est tout simplement de ne pas le faire : ne pas aliaser plus d'une référence à un même objet dans la même portée. Le code en sera d'ailleurs plus simple à comprendre et à débugguer. Cependant, quand on passe une référence en argument - de la façon dont Java est supposé le faire - l'aliasing entre automatiquement en jeu, et la référence locale créée peut modifier « l'objet extérieur » (l'objet qui a été créé en dehors de la portée de la méthode). En voici un exemple :

```
//: appendixa:Alias2.java
// Les appels de méthodes aliasent implicitement
// leurs arguments.

public class Alias2 {
  int i;
  Alias2(int ii) { i = ii; }
  static void f(Alias2 reference) {
    reference.i++;
  }
  public static void main(String[] args) {
    Alias2 x = new Alias2(7);
    System.out.println("x: " + x.i);
    System.out.println("Calling f(x)");
    f(x);
    System.out.println("x: " + x.i);
  }
} ///:~
```

Le résultat est :

```
x: 7
Calling f(x)
x: 8
```

La méthode modifie son argument, l'objet extérieur. Dans ce genre de situations, il faut décider si cela a un sens, si l'utilisateur s'y attend, et si cela peut causer des problèmes.

En général, on appelle une méthode afin de produire une valeur de retour et/ou une modification de l'état de l'objet *sur lequel est appelée la méthode* (une méthode consiste à « envoyer un message » à cet objet). Il est bien moins fréquent d'appeler une méthode afin de modifier ses arguments ; on appelle cela « appeler une méthode pour ses *effets de bord* ». Une telle méthode qui modifie ses arguments doit être clairement documentée et prévenir à propos de ses surprises potentielles. A cause de la confusion et des chausses-trappes engendrés, il vaut mieux s'abstenir de modifier les arguments.

S'il y a besoin de modifier un argument durant un appel de méthode sans que cela ne se répercute sur l'objet extérieur, alors il faut protéger cet argument en en créant une copie à l'intérieur de la méthode. Cette annexe traite principalement de ce sujet.

Création de copies locales

En résumé : tous les passages d'arguments en Java se font par référence. C'est à dire que quand on passe « un objet », on ne passe réellement qu'une référence à un objet qui vit en dehors de la méthode ; et si des modifications sont faites sur cette référence, on modifie l'objet extérieur. De plus :

- l'aliasing survient automatiquement durant le passage d'arguments ;
- il n'y a pas d'objets locaux, que des références locales ;
- les références ont une portée, les objets non ;
- la durée de vie d'un objet n'est jamais un problème en Java ;
- le langage ne fournit pas d'aide (tel que « const ») pour éviter qu'un objet ne soit modifié (c'est à dire pour se prémunir contre les effets négatifs de l'aliasing).

Si on ne fait que lire les informations d'un objet et qu'on ne le modifie pas, la forme la plus efficace de passage d'arguments consiste à passer une référence. C'est bien, car la manière de faire par défaut est aussi la plus efficace. Cependant, on peut avoir besoin de traiter l'objet comme s'il était « local » afin que les modifications apportées n'affectent qu'une copie locale et ne modifient pas l'objet extérieur. De nombreux langages proposent de créer automatiquement une copie locale de l'objet extérieur, à l'intérieur de la méthode href="#fn79" name="fnB79">[79]. Java ne dispose pas de cette fonctionnalité, mais il permet tout de même de mettre en oeuvre cet effet.

Passage par valeur

Ceci nous amène à discuter terminologie, ce qui est toujours bon dans un débat. Le sens de l'expression « passage par valeur » dépend de la perception qu'on a du fonctionnement du programme. Le sens général est qu'on récupère une copie locale de ce qu'on passe, mais cela est tempéré par notre façon de penser à propos de ce qu'on passe. Deux camps bien distincts s'affrontent quant au sens de « passage par valeur » :

1. Java passe tout par valeur. Quand on passe un scalaire à une méthode, on obtient une copie distincte de ce scalaire. Quand on passe une référence à une méthode, on obtient une copie de la référence. Ainsi, tous les passages d'arguments se font par valeur. Bien sûr, cela suppose qu'on raisonne en terme de références, mais Java a justement été conçu afin de vous permettre d'ignorer (la plupart du temps) que vous travaillez avec une référence. C'est à dire qu'il permet d'assimiler la référence à « l'objet », car il la déréférence automatiquement lorsqu'on fait un appel à une méthode.

2. Java passe les scalaires par valeur (pas de contestations sur ce point), mais les objets sont passés par référence. La référence est considérée comme un alias sur l'objet ; on ne pense donc *pas* passer une référence, mais on se dit plutôt « je passe l'objet ». Comme on n'obtient pas une copie locale de l'objet quand il est passé à une méthode, il est clair que les objets ne sont pas passés par valeur. Sun semble plutôt soutenir ce point de vue, puisque l'un des mot-clefs « réservés mais non implémentés » est **byvalue** (bien que rien ne précise si ce mot-clef verra le jour).

Après avoir présenté les deux camps et précisé que « cela dépend de la façon dont on considère une référence », je vais tenter de mettre le problème de côté. En fin de compte, ce n'est pas *si* important que cela - ce qui est important, c'est de comprendre que passer une référence permet de modifier l'objet passé en argument.

Clonage d'objets

La raison la plus courante de créer une copie locale d'un objet est qu'on veut modifier cet objet sans impacter l'objet de l'appelant. Si on décide de créer une copie locale, la méthode **clone()** permet de réaliser cette opération. C'est une méthode définie comme **protected** dans la classe de base **Object**, et qu'il faut redéfinir comme **public** dans les classes dérivées qu'on veut cloner. Par exemple, la classe **ArrayList** de la bibliothèque standard redéfinit **clone()**,on peut donc appeler **clone()** sur une **ArrayList** :

```
//: appendixa:Cloning.java
// L'opération clone() ne marche que pour quelques
// composants de la bibliothèque Java standard.
import java.util.*;

class Int {
  private int i;
  public Int(int ii) { i = ii; }
  public void increment() { i++; }
  public String toString() {
    return Integer.toString(i);
  }
}

public class Cloning {
  public static void main(String[] args) {
    ArrayList v = new ArrayList();
    for(int i = 0; i < 10; i++ )
      v.add(new Int(i));
    System.out.println("v: " + v);
    ArrayList v2 = (ArrayList)v.clone();
    // Incrémente tous les éléments de v2 :
    for(Iterator e = v2.iterator();
        e.hasNext(); )
      ((Int)e.next()).increment();
    // Vérifie si les éléments de v ont été modifiés :
    System.out.println("v: " + v);
  }
} ///:~
```

La méthode **clone()** produit un **Object**, qu'il faut alors retranstyper dans le bon type. Cet exemple montre que la méthode **clone()** de **ArrayList** *n'essaie pas* de cloner chacun des objets que l'**ArrayList** contient - l'ancienne **ArrayList** et l'**ArrayList** clonée référencent les mêmes objets. On appelle souvent cela une *copie superficielle*, puisque seule est copiée la « surface » d'un objet. L'objet réel est en réalité constitué de cette « surface », plus les objets sur lesquels les références pointent, plus tous les objets sur lesquels *ces* objets pointent, etc... On s'y réfère souvent en parlant de « réseau d'objets ». On appelle *copie profonde* le fait de copier la totalité de ce fouillis.

On peut voir les effets de la copie superficielle dans la sortie, où les actions réalisées sur **v2** affectent **v** :

```
v: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
v: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Ne pas essayer d'appeler **clone()** sur les objets contenus dans l'**ArrayList** est vraisemblablement une hypothèse raisonnable, car rien ne garantit que ces objets *sont* cloneables [80].

Rendre une classe cloneable

Bien que le méthode clone soit définie dans la classe **Object**, base de toutes les classes, le clonage n'est pas disponible dans toutes les classes [81]. Cela semble contraire à l'idée que les méthodes de la classe de base sont toujours disponibles dans les classes dérivées. Le clonage dans Java va contre cette idée ; si on veut le rendre disponible dans une classe, il faut explicitement ajouter du code pour que le clonage fonctionne.

Utilisation d'une astuce avec protected

Afin d'éviter de rendre chaque classe qu'on crée cloneable par défaut, la méthode **clone()** est **protected** dans la classe de base **Object**. Cela signifie non seulement qu'elle n'est pas disponible par défaut pour le programmeur client qui ne fait qu'utiliser la classe (sans en hériter), mais cela veut aussi dire qu'on ne peut pas appeler **clone()** via une référence à la classe de base (bien que cela puisse être utile dans certaines situations, comme le clonage polymorphique d'un ensemble d'**Object**s). C'est donc une manière de signaler, lors de la compilation, que l'objet n'est pas cloneable - et bizarrement, la plupart des classes de la bibliothèque standard Java ne le sont pas. Donc, si on écrit :

```
Integer x = new Integer(1);
x = x.clone();
```

On aura un message d'erreur lors de la compilation disant que **clone()** n'est pas accessible (puisque **Integer** ne la redéfinit pas et qu'elle se réfère donc à la version **protected**).

Si, par contre, on se trouve dans une classe dérivée d'**Object** (comme le sont toutes les classes), alors on a la permission d'appeler **Object.clone()** car elle est **protected** et qu'on est un héritier. La méthode **clone()** de la classe de base fonctionne - elle duplique effectivement bit à bit l'*objet de la classe dérivée*, réalisant une opération de clonage classique. Cependant, il faut tout de même rendre *sa propre* méthode de clonage **public** pour la rendre accessible. Donc, les deux points capitaux quand on clone sont :

- Toujours appeler **super.clone()**
- Rendre sa méthode clone **public**

On voudra probablement redéfinir **clone()** dans de futures classes dérivées, sans quoi le **clone()** (maintenant **public**) de la classe actuelle sera utilisé, et pourrait ne pas marcher (cependant, puisque **Object.clone()** crée une copie de l'objet, ça pourrait marcher). L'astuce **protected** ne marche qu'une fois - la première fois qu'on crée une classe dont on veut qu'elle soit cloneable héritant d'une classe qui ne l'est pas. Dans chaque classe dérivée de cette classe la méthode **clone()** sera accessible puisqu'il n'est pas possible en Java de réduire l'accès à une méthode durant la dérivation. C'est à dire qu'une fois qu'une classe est cloneable, tout ce qui en est dérivé est cloneable à moins d'utiliser les mécanismes (décrits ci-après) pour « empêcher » le clonage.

Implémenter l'interface Cloneable

Il y a une dernière chose à faire pour rendre un objet cloneable : implémenter l'**interface Clonable**. Cette **interface** est un peu spéciale, car elle est vide !

```
interface Cloneable {}
```

La raison d'implémenter cette **interface** vide n'est évidemment pas parce qu'on va surtyper jusqu'à **Cloneable** et appeler une de ses méthodes. L'utilisation d'**interface** dans ce contexte est considérée par certains comme une « astuce » car on utilise une de ses fonctionnalités dans un but autre que celui auquel on pensait originellement. Implémenter l'**interface Cloneable** agit comme une sorte de flag, codé en dur dans le type de la classe.

L'**interface Cloneable** existe pour deux raisons. Premièrement, on peut avoir une référence transtypée à un type de base et ne pas savoir s'il est possible de cloner cet objet. Dans ce cas, on peut utiliser le mot-clef **instanceof** (décrit au chapitre 12) pour savoir si la référence est connectée à un objet qui peut être cloné :

```
if(myReference instanceof Cloneable) // ...
```

La deuxième raison en est qu'on ne veut pas forcément que tous les types d'objets soient cloneables. Donc **Object.clone()** vérifie qu'une classe implémente l'interface **Cloneable**, et si ce n'est pas le cas, elle génère une exception **CloneNotSupportedException**. Donc en général, on est forcé d'implémenter **Cloneable** comme partie du mécanisme de clonage.

Une fois les détails d'implémentation de **clone()** compris, il est facile de créer des classes facilement duplicables pour produire des copies locales :

```
//: appendixa:LocalCopy.java
// Créer des copies locales avec clone().
import java.util.*;

class MyObject implements Cloneable {
  int i;
  MyObject(int ii) { i = ii; }
  public Object clone() {
    Object o = null;
    try {
      o = super.clone();
    } catch(CloneNotSupportedException e) {
      System.err.println("MyObject can't clone");
    }
    return o;
  }
  public String toString() {
    return Integer.toString(i);
  }
}

public class LocalCopy {
  static MyObject g(MyObject v) {
```

```
    // Passage par référence, modifie l'objet extérieur :
    v.i++;
    return v;
  }
  static MyObject f(MyObject v) {
    v = (MyObject)v.clone(); // Copie locale
    v.i++;
    return v;
  }
  public static void main(String[] args) {
    MyObject a = new MyObject(11);
    MyObject b = g(a);
    // On teste l'équivalence des références,
    // non pas l'équivalence des objets :
    if(a == b)
      System.out.println("a == b");
    else
      System.out.println("a != b");
    System.out.println("a = " + a);
    System.out.println("b = " + b);
    MyObject c = new MyObject(47);
    MyObject d = f(c);
    if(c == d)
      System.out.println("c == d");
    else
      System.out.println("c != d");
    System.out.println("c = " + c);
    System.out.println("d = " + d);
  }
} ///:~
```

Tout d'abord, **clone()** doit être accessible, il faut donc la rendre **public**. Ensuite, il faut que **clone()** commence par appeler la version de **clone()** de la classe de base. La méthode **clone()** appelée ici est celle prédéfinie dans **Object**, et on peut l'appeler car elle est **protected** et donc accessible depuis les classes dérivées.

Object.clone() calcule la taille de l'objet, réserve assez de mémoire pour en créer un nouveau, et copie tous les bits de l'ancien dans le nouveau. On appelle cela une *copie bit à bit*, et c'est typiquement ce qu'on attend d'une méthode **clone()**. Mais avant que **Object.clone()** ne réalise ces opérations, elle vérifie d'abord que la classe est **Cloneable** - c'est à dire, si elle implémente l'interface **Cloneable**. Si ce n'est pas le cas, **Object.clone()** génère une exception **CloneNotSupportedException** pour indiquer qu'on ne peut la cloner. C'est pourquoi il faut entourer l'appel à **super.clone()** dans un bloc try-catch, pour intercepter une exception qui théoriquement ne devrait jamais arriver (parce qu'on a implémenté l'interface **Cloneable**).

Dans **LocalCopy**, les deux méthodes **g()** et **f()** démontrent la différence entre les deux approches concernant le passage d'arguments. **g()** montre le passage par référence en modifiant l'objet extérieur et en retournant une référence à cet objet extérieur, tandis que **f()** clone l'argument, se détachant de lui et laissant l'objet original inchangé. Elle peut alors faire ce qu'elle veut, et même re-

tourner une référence sur ce nouvel objet sans impacter aucunement l'original. À noter l'instruction quelque peu curieuse :

```
v = (MyObject)v.clone();
```

C'est ici que la copie locale est créée. Afin d'éviter la confusion induite par une telle instruction, il faut se rappeler que cet idiome plutôt étrange est tout à fait légal en Java parce que chaque identifiant d'objet est en fait une référence. La référence **v** est donc utilisée pour réaliser une copie de l'objet qu'il référence grâce à **clone(),** qui renvoie une référence au type de base **Object** (car c'est ainsi qu'est définie **Object.clone()**) qui doit ensuite être transtypée dans le bon type.

Dans **main()**, la différence entre les effets des deux approches de passage d'arguments est testée. La sortie est :

```
a == b
a = 12
b = 12
c != d
c = 47
d = 48
```

Il est important de noter que les tests d'équivalence en Java ne regardent pas à l'intérieur des objets comparés pour voir si leurs valeurs sont les mêmes. Les opérateurs == et **!=** comparent simplement les références. Si les adresses à l'intérieur des références sont les mêmes, les références pointent sur le même objet et sont donc « égales ». Les opérateurs testent donc si les références sont aliasées sur le même objet !

Le mécanisme de Object.clone( )

Que se passe-t-il réellement quand **Object.clone()** est appelé, qui rende si essentiel d'appeler **super.clone()** quand on redéfinit **clone()** dans une classe ? La méthode **clone()** dans la classe racine (ie, **Object**) est chargée de la réservation de la mémoire nécessaire au stockage et de la copie bit à bit de l'objet original dans le nouvel espace de stockage. C'est à dire, elle ne crée pas seulement l'emplacement et copie un **Object** - elle calcule précisément la taille de l'objet copié et le duplique. Puisque tout cela se passe dans le code de la méthode **clone()** définie dans la classe de base (qui n'a aucune idée de ce qui est dérivé à partir d'elle), vous pouvez deviner que le processus implique RTTI pour déterminer quel est réellement l'objet cloné. De cette façon, la méthode **clone()** peut réserver la bonne quantité de mémoire et réaliser une copie bit à bit correcte pour ce type.

Quoi qu'on fasse, la première partie du processus de clonage devrait être un appel à **super.clone()**. Ceci pose les fondations de l'opération de clonage en créant une copie parfaite. On peut alors effectuer les autres opérations nécessaires pour terminer le clonage.

Afin de savoir exactement quelles sont ces autres opérations, il faut savoir ce que **Object.clone()** nous fournit. En particulier, clone-t-il automatiquement la destination de toutes les références ? L'exemple suivant teste cela :

```
//: appendixa:Snake.java
// Teste le clonage pour voir si la destination
// des références sont aussi clonées.
```

```
public class Snake implements Cloneable {
  private Snake next;
  private char c;
  // Valeur de i == nombre de segments
  Snake(int i, char x) {
    c = x;
    if(--i > 0)
      next = new Snake(i, (char)(x + 1));
  }
  void increment() {
    c++;
    if(next != null)
      next.increment();
  }
  public String toString() {
    String s = ":" + c;
    if(next != null)
      s += next.toString();
    return s;
  }
  public Object clone() {
    Object o = null;
    try {
      o = super.clone();
    } catch(CloneNotSupportedException e) {
      System.err.println("Snake can't clone");
    }
    return o;
  }
  public static void main(String[] args) {
    Snake s = new Snake(5, 'a');
    System.out.println("s = " + s);
    Snake s2 = (Snake)s.clone();
    System.out.println("s2 = " + s2);
    s.increment();
    System.out.println(
      "after s.increment, s2 = " + s2);
  }
} ///:~
```

Un **Snake** est composé d'un ensemble de segments, chacun de type **Snake**. C'est donc une liste chaînée simple. Les segments sont créés récursivement, en décrémentant le premier argument du constructeur pour chaque segment jusqu'à ce qu'on atteigne zéro. Afin de donner à chaque segment une étiquette unique, le deuxième argument, un **char**, est incrémenté pour chaque appel récursif au constructeur.

La méthode **increment()** incrémente récursivement chaque étiquette afin de pouvoir observer les modifications, et **toString()** affiche récursivement chaque étiquette. La sortie est la suivante :

```
s = :a:b:c:d:e
s2 = :a:b:c:d:e
after s.increment, s2 = :a:c:d:e:f
```

Ceci veut dire que seul le premier segment est dupliqué par **Object.clone()**, qui ne réalise donc qu'une copie superficielle. Si on veut dupliquer tout le **Snake** - une copie profonde - il faut réaliser d'autres opérations dans la méthode **clone()** redéfinie.

Typiquement il faudra donc faire un appel à **super.clone()** dans chaque classe dérivée d'une classe cloneable pour s'assurer que toutes les opérations de la classe de base (y compris **Object.-clone()**) soient effectuées. Puis cela sera suivi par un appel explicite à **clone()** pour chaque référence contenue dans l'objet ; sinon ces références seront aliasées sur celles de l'objet original. Le mécanisme est le même que lorsque les constructeurs sont appelés - constructeur de la classe de base d'abord, puis constructeur de la classe dérivée suivante, et ainsi de suite jusqu'au constructeur de la classe dérivée la plus lointaine de la classe de base. La différence est que **clone()** n'est pas un constructeur, il n'y a donc rien qui permette d'automatiser le processus. Il faut s'assurer de le faire soi-même.

Cloner un objet composé

Il se pose un problème quand on essaye de faire une copie profonde d'un objet composé. Il faut faire l'hypothèse que la méthode **clone()** des objets membres va à son tour réaliser une copie profonde de *leurs* références, et ainsi de suite. Il s'agit d'un engagement. Cela veut dire que pour qu'une copie profonde fonctionne il faut soit contrôler tout le code dans toutes les classes, soit en savoir suffisamment sur les classes impliquées dans la copie profonde pour être sûr qu'elles réalisent leur propre copie profonde correctement.

Cet exemple montre ce qu'il faut accomplir pour réaliser une copie profonde d'un objet composé :

```
//: appendixa:DeepCopy.java
// Clonage d'un objet composé.

class DepthReading implements Cloneable {
  private double depth;
  public DepthReading(double depth) {
    this.depth = depth;
  }
  public Object clone() {
    Object o = null;
    try {
      o = super.clone();
    } catch(CloneNotSupportedException e) {
      e.printStackTrace(System.err);
    }
    return o;
  }
}
```

```
class TemperatureReading implements Cloneable {
  private long time;
  private double temperature;
  public TemperatureReading(double temperature) {
    time = System.currentTimeMillis();
    this.temperature = temperature;
  }
  public Object clone() {
    Object o = null;
    try {
      o = super.clone();
    } catch(CloneNotSupportedException e) {
      e.printStackTrace(System.err);
    }
    return o;
  }
}

class OceanReading implements Cloneable {
  private DepthReading depth;
  private TemperatureReading temperature;
  public OceanReading(double tdata, double ddata){
    temperature = new TemperatureReading(tdata);
    depth = new DepthReading(ddata);
  }
  public Object clone() {
    OceanReading o = null;
    try {
      o = (OceanReading)super.clone();
    } catch(CloneNotSupportedException e) {
      e.printStackTrace(System.err);
    }
    // On doit cloner les références :
    o.depth = (DepthReading)o.depth.clone();
    o.temperature =
      (TemperatureReading)o.temperature.clone();
    return o; // Transtypage en Object
  }
}

public class DeepCopy {
  public static void main(String[] args) {
    OceanReading reading =
      new OceanReading(33.9, 100.5);
    // Maintenant on le clone :
    OceanReading r =
      (OceanReading)reading.clone();
```

```
  }
} ///:~
```

DepthReading et **TemperatureReading** sont quasi identiques ; elles ne contiennent toutes les deux que des scalaires. La méthode **clone()** est donc relativement simple : elle appelle **super.-clone()** et renvoie le résultat. Notez que le code de **clone()** des deux classes est identique.

OceanReading est composée d'objets **DepthReading** et **TemperatureReading** ; pour réaliser une copie profonde, sa méthode **clone()** doit donc cloner les références à l'intérieur de **Ocean-Reading**. Pour réaliser ceci, le résultat de **super.clone()** doit être transtypé dans un objet **Ocean-Reading** (afin de pouvoir accéder aux références **depth** et **temperature**).

Copie profonde d'une ArrayList

Reprenons l'exemple **ArrayList** exposé plus tôt dans cette annexe. Cette fois-ci la classe **Int2** est cloneable, on peut donc réaliser une copie profonde de l'**ArrayList** :

```
//: appendixa:AddingClone.java
// Il faut apporter quelques modifications
// pour que vos classes soient cloneables.
import java.util.*;

class Int2 implements Cloneable {
  private int i;
  public Int2(int ii) { i = ii; }
  public void increment() { i++; }
  public String toString() {
    return Integer.toString(i);
  }
  public Object clone() {
    Object o = null;
    try {
      o = super.clone();
    } catch(CloneNotSupportedException e) {
      System.err.println("Int2 can't clone");
    }
    return o;
  }
}

// Une fois qu'elle est cloneable, l'héritage
// ne supprime pas cette propriété :
class Int3 extends Int2 {
  private int j; // Automatiquement dupliqué
  public Int3(int i) { super(i); }
}

public class AddingClone {
```

```
  public static void main(String[] args) {
    Int2 x = new Int2(10);
    Int2 x2 = (Int2)x.clone();
    x2.increment();
    System.out.println(
      "x = " + x + ", x2 = " + x2);
    // Tout objet hérité est aussi cloneable :
    Int3 x3 = new Int3(7);
    x3 = (Int3)x3.clone();

    ArrayList v = new ArrayList();
    for(int i = 0; i < 10; i++ )
      v.add(new Int2(i));
    System.out.println("v: " + v);
    ArrayList v2 = (ArrayList)v.clone();
    // Maintenant on clone chaque élément :
    for(int i = 0; i < v.size(); i++)
      v2.set(i, ((Int2)v2.get(i)).clone());
    // Incrémente tous les éléments de v2 :
    for(Iterator e = v2.iterator();
        e.hasNext(); )
      ((Int2)e.next()).increment();
    // Vérifie si les éléments de v ont été modifiés :
    System.out.println("v: " + v);
    System.out.println("v2: " + v2);
  }
} ///:~
```

Int3 est dérivée de **Int2** et un nouveau membre scalaire **int j** a été ajouté. On pourrait croire qu'il faut redéfinir **clone()** pour être sûr que **j** soit copié, mais ce n'est pas le cas. Lorsque la méthode **clone()** de **Int2** est appelée à la place de la méthode **clone()** de **Int3**, elle appelle **Object.clone()**, qui détermine qu'elle travaille avec un **Int3** et duplique tous les bits de **Int3**. Tant qu'on n'ajoute pas de références qui ont besoin d'être clonées, l'appel à **Object.clone()** réalise toutes les opérations nécessaires au clonage, sans se préoccuper de la profondeur hiérarchique où **clone()** a été définie.

Pour réaliser une copie profonde d'une **ArrayList**, il faut donc la cloner, puis la parcourir et cloner chacun des objets pointés par l'**ArrayList**. Un mécanisme similaire serait nécessaire pour réaliser une copie profonde d'un **HashMap**.

Le reste de l'exemple prouve que le clonage s'est bien passé en montrant qu'une fois cloné, un objet peut être modifié sans que l'objet original n'en soit affecté.

Quand on examine la sérialisation d'objets dans Java (présentée au Chapitre 11), on se rend compte qu'un objet sérialisé puis désérialisé est, en fait, cloné.

Pourquoi alors ne pas utiliser la sérialisation pour réaliser une copie profonde ? Voici un exemple qui compare les deux approches en les chronométrant :

```
//: appendixa:Compete.java
import java.io.*;
```

```
class Thing1 implements Serializable {}
class Thing2 implements Serializable {
  Thing1 o1 = new Thing1();
}

class Thing3 implements Cloneable {
  public Object clone() {
    Object o = null;
    try {
      o = super.clone();
    } catch(CloneNotSupportedException e) {
      System.err.println("Thing3 can't clone");
    }
    return o;
  }
}

class Thing4 implements Cloneable {
  Thing3 o3 = new Thing3();
  public Object clone() {
    Thing4 o = null;
    try {
      o = (Thing4)super.clone();
    } catch(CloneNotSupportedException e) {
      System.err.println("Thing4 can't clone");
    }
    // Clone aussi la donnée membre :
    o.o3 = (Thing3)o3.clone();
    return o;
  }
}

public class Compete {
  static final int SIZE = 5000;
  public static void main(String[] args)
  throws Exception {
    Thing2[] a = new Thing2[SIZE];
    for(int i = 0; i < a.length; i++)
      a[i] = new Thing2();
    Thing4[] b = new Thing4[SIZE];
    for(int i = 0; i < b.length; i++)
      b[i] = new Thing4();
    long t1 = System.currentTimeMillis();
    ByteArrayOutputStream buf =  new ByteArrayOutputStream();
    ObjectOutputStream o =  new ObjectOutputStream(buf);
    for(int i = 0; i < a.length; i++)
```

```
        o.writeObject(a[i]);
      // Récupère les copies:
      ObjectInputStream in =  new ObjectInputStream(
         new ByteArrayInputStream(
           buf.toByteArray()));
      Thing2[] c = new Thing2[SIZE];
      for(int i = 0; i < c.length; i++)
        c[i] = (Thing2)in.readObject();
      long t2 = System.currentTimeMillis();
      System.out.println(
        "Duplication via serialization: " +
        (t2 - t1) + " Milliseconds");
      // Maintenant on tente le clonage :
      t1 = System.currentTimeMillis();
      Thing4[] d = new Thing4[SIZE];
      for(int i = 0; i < d.length; i++)
        d[i] = (Thing4)b[i].clone();
      t2 = System.currentTimeMillis();
      System.out.println(
        "Duplication via cloning: " +
        (t2 - t1) + " Milliseconds");
    }
  } ///:~
```

Thing2 et **Thing4** contiennent des objets membres afin qu'une copie profonde soit nécessaire. Il est intéressant de noter que bien que les classes **Serializable** soient plus faciles à implémenter, elles nécessitent plus de travail pour les copier. Le support du clonage demande plus de travail pour créer la classe, mais la duplication des objets est relativement simple. Les résultats sont édifiants. Voici la sortie obtenue pour trois exécutions :

```
Duplication via serialization: 940 Milliseconds
Duplication via cloning: 50 Milliseconds

Duplication via serialization: 710 Milliseconds
Duplication via cloning: 60 Milliseconds

Duplication via serialization: 770 Milliseconds
Duplication via cloning: 50 Milliseconds
```

Outre la différence significative de temps entre la sérialisation et le clonage, vous noterez aussi que la sérialisation semble beaucoup plus sujette aux variations, tandis que le clonage a tendance à être plus stable.

Supporter le clonage plus bas dans la hiérarchie

Si une nouvelle classe est créée, sa classe de base par défaut est **Object**, qui par défaut n'est pas cloneable (comme vous le verrez dans la section suivante). Tant qu'on n'implémente pas explicitement le clonage, celui-ci ne sera pas disponible. Mais on peut le rajouter à n'importe quel niveau

et la classe sera cloneable à partir de ce niveau dans la hiérarchie, comme ceci :

```
//: appendixa:HorrorFlick.java
// On peut implémenter le Clonage
// à n'importe quel niveau de la hiérarchie.
import java.util.*;

class Person {}
class Hero extends Person {}
class Scientist extends Person
    implements Cloneable {
  public Object clone() {
    try {
      return super.clone();
    } catch(CloneNotSupportedException e) {
      // Ceci ne devrait jamais arriver :
      // la classe est Cloneable !
      throw new InternalError();
    }
  }
}
class MadScientist extends Scientist {}

public class HorrorFlick {
  public static void main(String[] args) {
    Person p = new Person();
    Hero h = new Hero();
    Scientist s = new Scientist();
    MadScientist m = new MadScientist();

    // p = (Person)p.clone(); // Erreur lors de la compilation
    // h = (Hero)h.clone(); // Erreur lors de la compilation
    s = (Scientist)s.clone();
    m = (MadScientist)m.clone();
  }
} ///:~
```

Tant que le clonage n'est pas supporté, le compilateur bloque toute tentative de clonage. Si le clonage est ajouté dans la classe **Scientist**, alors **Scientist** et tous ses descendants sont cloneables.

Pourquoi cet étrange design ?

Si tout ceci vous semble étrange, c'est parce que ça l'est réellement. On peut se demander comment on en est arrivé là. Que se cache-t-il derrière cette conception ?

Originellement, Java a été conçu pour piloter des boîtiers, sans aucune pensée pour l'Internet. Dans un langage générique tel que celui-ci, il semblait sensé que le programmeur soit capable de cloner n'importe quel objet. C'est ainsi que **clone()** a été placée dans la classe de base **Object**, *mais* c'était une méthode **public** afin qu'un objet puisse toujours être cloné. Cela semblait l'approche la

plus flexible, et après tout, quel mal y avait-il à cela ?

Puis, quand Java s'est révélé comme le langage de programmation idéal pour Internet, les choses ont changé. Subitement, des problèmes de sécurité sont apparus, et bien sûr, ces problèmes ont été réglés en utilisant des objets, et on ne voulait pas que n'importe qui soit capable de cloner ces objets de sécurité. Ce qu'on voit donc est une suite de patchs appliqués sur l'arrangement initialement simple : **clone()** est maintenant **protected** dans **Object**. Il faut la redéfinir *et* **implement**er **Cloneable***et* traiter les exceptions.

Il est bon de noter qu'on n'est obligé d'utiliser l'interface **Cloneable** *que* si on fait un appel à la méthode **clone()** de **Object**, puisque cette méthode vérifie lors de l'exécution que la classe implémente **Cloneable**. Mais dans un souci de cohérence (et puisque de toute façon **Cloneable** est vide), il vaut mieux l'implémenter.

Contrôler la clonabilité

On pourrait penser que, pour supprimer le support du clonage, il suffit de rendre **private** la méthode **clone()**, mais ceci ne marchera pas car on ne peut prendre une méthode de la classe de base et la rendre moins accessible dans une classe dérivée. Ce n'est donc pas si simple. Et pourtant, il est essentiel d'être capable de contrôler si un objet peut être cloné ou non. Une classe peut adopter plusieurs attitudes à ce propos :

1. L'indifférence. Rien n'est fait pour supporter le clonage, ce qui signifie que la classe ne peut être clonée, mais qu'une classe dérivée peut implémenter le clonage si elle veut. Ceci ne fonctionne que si **Object.clone()** traite comme il faut tous les champs de la classe.

2. Implémenter **clone()**. Respecter la marche à suivre pour l'implémentation de **Cloneable** et redéfinir **clone()**. Dans la méthode **clone()** redéfinie, appeler **super.clone()** et intercepter toutes les exceptions (afin que la méthode **clone()** redéfinie ne génère pas d'exceptions).

3. Supporter le clonage conditionnellement. Si la classe contient des références sur d'autres objets qui peuvent ou non être cloneables (une classe conteneur, par exemple), la méthode **clone()** peut essayer de cloner tous les objets référencés, et s'ils génèrent des exceptions, relayer ces exceptions au programmeur. Par exemple, prenons le cas d'une sorte d'**ArrayList** qui essayerait de cloner tous les objets qu'elle contient. Quand on écrit une telle **ArrayList**, on ne peut savoir quelle sorte d'objets le programmeur client va pouvoir stocker dans l'**ArrayList**, on ne sait donc pas s'ils peuvent être clonés.

4. Ne pas implémenter **Cloneable** mais redéfinir **clone()** en **protected**, en s'assurant du fonctionnement correct du clonage pour chacun des champs. De cette manière, toute classe dérivée peut redéfinir **clone()** et appeler **super.clone()** pour obtenir le comportement attendu lors du clonage. Cette implémentation peut et doit invoquer **super.clone()** même si cette méthode attend un objet **Cloneable** (elle génère une exception sinon), car personne ne l'invoquera directement sur un objet de la classe. Elle ne sera invoquée qu'à travers une classe dérivée, qui, elle, implémente **Cloneable** si elle veut obtenir le fonctionnement désiré.

5. Tenter de bloquer le clonage en n'implémentant pas **Cloneable** et en redéfinissant **clone()** afin de générer une exception. Ceci ne fonctionne que si toutes les classes dérivées appellent **super.clone()** dans leur redéfinition de **clone()**. Autrement, un programmeur est capable de contourner ce mécanisme.

6. Empêcher le clonage en rendant la classe **final**. Si **clone()** n'a pas été redéfinie par

l'une des classes parentes, alors elle ne peut plus l'être. Si elle a déjà été redéfinie, la redéfinir à nouveau et générer une exception **CloneNotSupportedException**. Rendre la classe **final** est la seule façon d'interdire catégoriquement le clonage. De plus, si on manipule des objets de sécurité ou dans d'autres situations dans lesquelles on veut contrôler le nombre d'objets créés, il faut rendre tous les constructeurs **private** et fournir une ou plusieurs méthodes spéciales pour créer les objets. De cette manière, les méthodes peuvent restreindre le nombre d'objets créés et les conditions dans lesquelles ils sont créés (un cas particulier en est le patron *singleton* présenté dans *Thinking in Patterns with Java*, téléchargeable à *www.BruceEckel.com*).

Voici un exemple qui montre les différentes façons dont le clonage peut être implémenté et interdit plus bas dans la hiérarchie :

```
//: appendixa:CheckCloneable.java
// Vérifie si une référence peut être clonée.

// Ne peut être clonée car ne redéfinit pas clone() :
class Ordinary {}

// Redéfinit clone, mais n'implémente pas
// Cloneable :
class WrongClone extends Ordinary {
  public Object clone()
      throws CloneNotSupportedException {
    return super.clone(); // Génère une exception
  }
}

// Fait le nécessaire pour le clonage :
class IsCloneable extends Ordinary
    implements Cloneable {
  public Object clone()
      throws CloneNotSupportedException {
    return super.clone();
  }
}

// Interdit le clonage en générant une exception :
class NoMore extends IsCloneable {
  public Object clone()
      throws CloneNotSupportedException {
    throw new CloneNotSupportedException();
  }
}

class TryMore extends NoMore {
  public Object clone()
      throws CloneNotSupportedException {
    // Appelle NoMore.clone(), génère une excpetion :
```

```
    return super.clone();
  }
}

class BackOn extends NoMore {
  private BackOn duplicate(BackOn b) {
    // Crée une copie de b d'une façon ou d'une autre
    // et renvoie cette copie. C'est une copie sans
    // intérêt, juste pour l'exemple :
    return new BackOn();
  }
  public Object clone() {
    // N'appelle pas NoMore.clone() :
    return duplicate(this);
  }
}

// On ne peut dériver cette classe, donc on ne peut
// redéfinir la méthode clone comme dans BackOn:
final class ReallyNoMore extends NoMore {}

public class CheckCloneable {
  static Ordinary tryToClone(Ordinary ord) {
    String id = ord.getClass().getName();
    Ordinary x = null;
    if(ord instanceof Cloneable) {
      try {
        System.out.println("Attempting " + id);
        x = (Ordinary)((IsCloneable)ord).clone();
        System.out.println("Cloned " + id);
      } catch(CloneNotSupportedException e) {
        System.err.println("Could not clone "+id);
      }
    }
    return x;
  }
  public static void main(String[] args) {
    // Transtypage ascendant :
    Ordinary[] ord = {
      new IsCloneable(),
      new WrongClone(),
      new NoMore(),
      new TryMore(),
      new BackOn(),
      new ReallyNoMore(),
    };
    Ordinary x = new Ordinary();
```

```
    // Ceci ne compilera pas, puisque clone()
    // est protected dans Object:
    //! x = (Ordinary)x.clone();
    // tryToClone() vérifie d'abord si
    // une classe implémente Cloneable :
    for(int i = 0; i < ord.length; i++)
      tryToClone(ord[i]);
  }
} ///:~
```

La première classe, **Ordinary**, représente le genre de classes que nous avons rencontré tout au long de ce livre : pas de support du clonage, mais pas de contrôle sur la clonabilité non plus. Mais si on dispose d'une référence sur un objet **Ordinary** qui peut avoir été transtypé à partir d'une classe dérivée, on ne peut savoir s'il est peut être cloné ou non.

La classe **WrongClone** montre une implémentation incorrecte du clonage. Elle redéfinit bien **Object.clone()** et rend la méthode **public**, mais elle n'implémente pas **Cloneable**, donc quand **super.clone()** est appelée (ce qui revient à un appel à **Object.clone()**), une exception **CloneNotSupportedException** est générée et le clonage échoue.

La classe **IsCloneable** effectue toutes les actions nécessaires au clonage : **clone()** est redéfinie et **Cloneable** implémentée. Cependant, cette méthode **clone()** et plusieurs autres qui suivent dans cet exemple n'*interceptent pas***CloneNotSupportedException**, mais la font suivre à l'appelant, qui doit alors l'envelopper dans un bloc try-catch. Dans les méthodes **clone()** typiques il faut intercepter **CloneNotSupportedException** *à l'intérieur* de **clone()** plutôt que de la propager. Cependant dans cet exemple, il est plus intéressant de propager les exceptions.

La classe **NoMore** tente d'interdire le clonage comme les concepteurs de Java pensaient le faire : en générant une exception **CloneNotSupportedException** dans la méthode **clone()** de la classe dérivée. La méthode **clone()** de la classe **TryMore** appelle **super.clone()**, ce qui revient à appeler **NoMore.clone()**, qui génère une exception et empêche donc le clonage.

Mais que se passe-t-il si le programmeur ne respecte pas la chaîne d'appel « recommandée » et n'appelle pas **super.clone()** à l'intérieur de la méthode **clone()** redéfinie ? C'est ce qui se passe dans la classe **BackOn**. Cette classe utilise une méthode séparée **duplicate()** pour créer une copie de l'objet courant et appelle cette méthode dans **clone()***au lieu* d'appeler **super.clone()**. L'exception n'est donc jamais générée et la nouvelle classe est cloneable. La seule solution vraiment sûre est montrée dans **ReallyNoMore**, qui est **final** et ne peut donc être dérivée. Ce qui signifie que si **clone()** génère une exception dans la classe **final**, elle ne peut être modifiée via l'héritage et la prévention du clonage est assurée (on ne peut appeler explicitement **Object.clone()** depuis une classe qui a un niveau arbitraire d'héritage ; on en est limité à appeler **super.clone()**, qui a seulement accès à sa classe parente directe). Implémenter des objets qui traite de sujets relatifs à la sécurité implique donc de rendre ces classes **final**.

La première méthode qu'on voit dans la classe **CheckCloneable** est **tryToClone()**, qui prend n'importe quel objet **Ordinary** et vérifie s'il est cloneable grâce à **instanceof**. Si c'est le cas, il transtype l'objet en **IsCloneable**, appelle **clone()** et retranstype le résultat en **Ordinary**, interceptant toutes les exceptions générées. Remarquez l'utilisation de l'identification dynamique du type (voir Chapitre 12) pour imprimer le nom de la classe afin de suivre le déroulement du programme.

Dans **main()**, différents types d'objets **Ordinary** sont créés et transtypés en **Ordinary** dans la

définition du tableau. Les deux premières lignes de code qui suivent créent un objet **Ordinary** et tentent de le cloner. Cependant ce code ne compile pas car **clone()** est une méthode **protected** dans **Object**. Le reste du code parcourt le tableau et essaye de cloner chaque objet, reportant le succès ou l'échec de l'opération. Le résultat est :

```
Attempting IsCloneable
Cloned IsCloneable
Attempting NoMore
Could not clone NoMore
Attempting TryMore
Could not clone TryMore
Attempting BackOn
Cloned BackOn
Attempting ReallyNoMore
Could not clone ReallyNoMore
```

Donc pour resumer, si on veut qu'une classe soit cloneable, il faut:

1. Implémenter l'interface **Cloneable**.
2. Redéfinir **clone()**.
3. Appeler **super.clone()** depuis la méthode **clone()** de la classe.
4. Intercepter les exceptions à l'intérieur de la méthode **clone()**.

Ceci produira l'effet désiré.

Le constructeur de copie

Le clonage peut sembler un processus compliqué à mettre en oeuvre. On se dit qu'il doit certainement exister une autre alternative, et souvent on envisage (surtout les programmeurs C++) de créer un constructeur spécial dont le travail est de dupliquer un objet. En C++, on l'appelle le *constructeur de copie*. Cela semble a priori la solution la plus évidente, mais en fait elle ne fonctionne pas. Voici un exemple.

```
//: appendixa:CopyConstructor.java
// Un constructeur pour copier un objet du même
// type, dans une tentaive de créer une copie locale.

class FruitQualities {
  private int weight;
  private int color;
  private int firmness;
  private int ripeness;
  private int smell;
  // etc...
  FruitQualities() { // Constucteur par défaut.
    // fait des tas de choses utiles...
  }
  // D'autres constructeurs :
  // ...
```

```
  // Constructeur de copie :
  FruitQualities(FruitQualities f) {
    weight = f.weight;
    color = f.color;
    firmness = f.firmness;
    ripeness = f.ripeness;
    smell = f.smell;
    // etc...
  }
}

class Seed {
  // Membres...
  Seed() { /* Constructeur par défaut */ }
  Seed(Seed s) { /* Constructeur de copie */ }
}

class Fruit {
  private FruitQualities fq;
  private int seeds;
  private Seed[] s;
  Fruit(FruitQualities q, int seedCount) {
    fq = q;
    seeds = seedCount;
    s = new Seed[seeds];
    for(int i = 0; i < seeds; i++)
      s[i] = new Seed();
  }
  // Autres constructeurs :
  // ...
  // Constructeur de copie :
  Fruit(Fruit f) {
    fq = new FruitQualities(f.fq);
    seeds = f.seeds;
    // Appelle le constructeur de copie sur toutes les Seed :
    for(int i = 0; i < seeds; i++)
      s[i] = new Seed(f.s[i]);
    // D'autres activités du constructeur de copie...
  }
  // Pour permettre aux constructeurs dérivés (ou aux
  // autres méthodes) de changer les qualités :
  protected void addQualities(FruitQualities q) {
    fq = q;
  }
  protected FruitQualities getQualities() {
    return fq;
  }
```

```
}

class Tomato extends Fruit {
  Tomato() {
    super(new FruitQualities(), 100);
  }
  Tomato(Tomato t) { // Constructeur de copie.
    super(t); // Transtypage pour le constructeur de copie parent.
    // D'autres activités du constructeur de copie...
  }
}

class ZebraQualities extends FruitQualities {
  private int stripedness;
  ZebraQualities() { // Constructeur par défaut.
    // Fait des tas de choses utiles...
  }
  ZebraQualities(ZebraQualities z) {
    super(z);
    stripedness = z.stripedness;
  }
}

class GreenZebra extends Tomato {
  GreenZebra() {
    addQualities(new ZebraQualities());
  }
  GreenZebra(GreenZebra g) {
    super(g); // Appelle Tomato(Tomato)
    // Restitue les bonnes qualités :
    addQualities(new ZebraQualities());
  }
  void evaluate() {
    ZebraQualities zq =
      (ZebraQualities)getQualities();
    // Utilise les qualités
    // ...
  }
}

public class CopyConstructor {
  public static void ripen(Tomato t) {
    // Utilise le « constructeur de copie » :
    t = new Tomato(t);
    System.out.println("In ripen, t is a " +
      t.getClass().getName());
  }
```

```
  public static void slice(Fruit f) {
    f = new Fruit(f); // Hmmm... est-ce que cela va marcher ?
    System.out.println("In slice, f is a " +
      f.getClass().getName());
  }
  public static void main(String[] args) {
    Tomato tomato = new Tomato();
    ripen(tomato); // OK
    slice(tomato); // OOPS!
    GreenZebra g = new GreenZebra();
    ripen(g); // OOPS!
    slice(g); // OOPS!
    g.evaluate();
  }
} ///:~
```

Ceci semble un peu étrange à première vue. Bien sûr, un fruit a des qualités, mais pourquoi ne pas mettre les données membres représentant ces qualités directement dans la classe **Fruit** ? Deux raisons à cela. La première est qu'on veut pouvoir facilement insérer ou changer les qualités. Notez que **Fruit** possède une méthode **protected addQualities()** qui permet aux classes dérivées de le faire (on pourrait croire que la démarche logique serait d'avoir un constructeur **protected** dans **Fruit** qui accepte un argument **FruitQualities**, mais les constructeurs ne sont pas hérités et ne seraient pas disponibles dans les classes dérivées). En créant une classe séparée pour la qualité des fruits, on dispose d'une plus grande flexibilité, incluant la possibilité de changer les qualités d'un objet **Fruit** pendant sa durée de vie.

La deuxième raison pour laquelle on a décidé de créer une classe **FruitQualities** est dans le cas où on veut ajouter de nouvelles qualités ou en changer le comportement via héritage ou polymorphisme. Notez que pour les **GreenZebra** (qui sont *réellement* un type de tomates - j'en ai cultivé et elles sont fabuleuses), le constructeur appelle **addQualities()** et lui passe un objet **ZebraQualities**, qui est dérivé de **FruitQualities** et peut donc être attaché à la référence **FruitQualities** de la classe de base. Bien sûr, quand **GreenZebra** utilise les **FruitQualities** il doit le transtyper dans le type correct (comme dans **evaluate()**), mais il sait que le type est toujours **ZebraQualities**.

Vous noterez aussi qu'il existe une classe **Seed**, et qu'un **Fruit** (qui par définition porte ses propres graines) name="fnB82">[82]contient un tableau de **Seeds**.

Enfin, vous noterez que chaque classe dispose d'un constructeur de copie, et que chaque constructeur de copie doit s'occuper d'appeler le constructeur de copie de la classe de base et des objets membres pour réaliser une copie profonde. Le constructeur de copie est testé dans la classe **CopyConstructor**. La méthode **ripen()** accepte un argument **Tomato** et réalise une construction de copie afin de dupliquer l'objet :

```
t = new Tomato(t);
```

tandis que **slice( )** accepte un objet **Fruit** plus générique et le duplique aussi :

```
f = new Fruit(f);
```

Ces deux méthodes sont testées avec différents types de **Fruit** dans **main()**. Voici la sortie produite :

```
In ripen, t is a Tomato
In slice, f is a Fruit
In ripen, t is a Tomato
In slice, f is a Fruit
```

C'est là que le problème survient. Après la construction de copie réalisée dans **slice()** sur l'objet **Tomato**, l'objet résultant n'est plus un objet **Tomato**, mais seulement un **Fruit**. Il a perdu toute sa tomaticité. De même, quand on prend une **GreenZebra**, **ripen()** et **slice()** la transforment toutes les deux en **Tomato** et **Fruit**, respectivement. La technique du constructeur de copie ne fonctionne donc pas en Java pour créer une copie locale d'un objet.

Pourquoi cela fonctionne-t-il en C++ et pas en Java ?

Le constructeur de copie est un mécanisme fondamental en C++, puisqu'il permet de créer automatiquement une copie locale d'un objet. Mais l'exemple précédent prouve que cela ne fonctionne pas en Java. Pourquoi ? En Java, toutes les entités manipulées sont des références, tandis qu'en C++ on peut manipuler soit des références sur les objets soit les objets directement. C'est le rôle du constructeur de copie en C++ : prendre un objet et permettre son passage par valeur, donc dupliquer l'objet. Cela fonctionne donc très bien en C++, mais il faut garder présent à l'esprit que ce mécanisme est à proscrire en Java.

Classes en lecture seule

Bien que la copie locale produite par **clone()** donne les résultats escomptés dans les cas appropriés, c'est un exemple où le programmeur (l'auteur de la méthode) est responsable des effets secondaires indésirables de l'aliasing. Que se passe-t-il dans le cas où on construit une bibliothèque tellement générique et utilisée qu'on ne peut supposer qu'elle sera toujours clonée aux bons endroits ? Ou alors, que se passe-t-il si on *veut* permettre l'aliasing dans un souci d'efficacité - afin de prévenir la duplication inutile d'un objet - mais qu'on n'en veut pas les effets secondaires négatifs ?

Une solution est de créer des *objets immuables* appartenant à des classes en lecture seule. On peut définir une classe telle qu'aucune méthode de la classe ne modifie l'état interne de l'objet. Dans une telle classe, l'aliasing n'a aucun impact puisqu'on peut seulement lire son état interne, donc même si plusieurs portions de code utilisent le même objet cela ne pose pas de problèmes.

Par exemple, la bibliothèque standard Java contient des classes « wrapper » pour tous les types fondamentaux. Vous avez peut-être déjà découvert que si on veut stocker un **int** dans un conteneur tel qu'une **ArrayList** (qui n'accepte que des références sur un **Object**), on peut insérer l'**int** dans la classe **Integer** de la bibliothèque standard :

```
//: appendixa:ImmutableInteger.java
// La classe Integer ne peut pas être modifiée.
import java.util.*;

public class ImmutableInteger {
  public static void main(String[] args) {
    ArrayList v = new ArrayList();
    for(int i = 0; i < 10; i++)
      v.add(new Integer(i));
```

```
    // Mais comment changer l'int à
    // l'intérieur de Integer?
  }
} ///:~
```

La classe **Integer** (de même que toutes les classes « wrapper » pour les scalaires) implémentent l'immuabilité d'une manière simple : elles ne possèdent pas de méthodes qui permettent de modifier l'objet.

Si on a besoin d'un objet qui contient un scalaire qui peut être modifié, il faut la créer soi-même. Heureusement, ceci se fait facilement :

```
//: appendixa:MutableInteger.java
// Une classe wrapper modifiable.
import java.util.*;

class IntValue {
  int n;
  IntValue(int x) { n = x; }
  public String toString() {
    return Integer.toString(n);
  }
}

public class MutableInteger {
  public static void main(String[] args) {
    ArrayList v = new ArrayList();
    for(int i = 0; i < 10; i++)
      v.add(new IntValue(i));
    System.out.println(v);
    for(int i = 0; i < v.size(); i++)
      ((IntValue)v.get(i)).n++;
    System.out.println(v);
  }
} ///:~
```

Notez que **n** est amical pour simplifier le codage.

IntValue peut même être encore plus simple si l'initialisation à zéro est acceptable (auquel cas on n'a plus besoin du constructeur) et qu'on n'a pas besoin d'imprimer cet objet (auquel cas on n'a pas besoin de **toString()**) :

```
class IntValue { int n; }
```

La recherche de l'élément et son transtypage par la suite est un peu lourd et maladroit, mais c'est une particularité de **ArrayList** et non de **IntValue**.

Créer des classes en lecture seule

Il est possible de créer ses propres classes en lecture seule. Voici un exemple :

```
//: appendixa:Immutable1.java
// Des objets qu'on ne peut modifier
// ne craignent pas l'aliasing.

public class Immutable1 {
  private int data;
  public Immutable1(int initVal) {
    data = initVal;
  }
  public int read() { return data; }
  public boolean nonzero() { return data != 0; }
  public Immutable1 quadruple() {
    return new Immutable1(data * 4);
  }
  static void f(Immutable1 i1) {
    Immutable1 quad = i1.quadruple();
    System.out.println("i1 = " + i1.read());
    System.out.println("quad = " + quad.read());
  }
  public static void main(String[] args) {
    Immutable1 x = new Immutable1(47);
    System.out.println("x = " + x.read());
    f(x);
    System.out.println("x = " + x.read());
  }
} ///:~
```

Toutes les données sont **private**, et aucune méthode **public** ne modifie les données. En effet, la méthode qui semble modifier l'objet, **quadruple()**, crée en fait un nouvel objet **Immutable1** sans modifier l'objet original.

La méthode **f()** accepte un objet **Immutable1** et effectue diverses opérations avec, et la sortie de **main()** démontre que **x** ne subit aucun changement. Ainsi, l'objet **x** peut être aliasé autant qu'on le veut sans risque puisque la classe **Immutable1** a été conçue afin de guarantir que les objets ne puissent être modifiés.

L'inconvénient de l'immuabilité

Créer une classe immuable semble à première vue une solution élégante. Cependant, dès qu'on a besoin de modifier un objet de ce nouveau type, il faut supporter le coût supplémentaire de la création d'un nouvel objet, ce qui implique aussi un passage plus fréquent du ramasse-miettes. Cela n'est pas un problème pour certaines classes, mais cela est trop coûteux pour certaines autres (telles que la classes **String**).

La solution est de créer une classe compagnon qui, elle, *peut* être modifiée. Ainsi, quand on effectue beaucoup de modifications, on peut basculer sur la classe compagnon modifiable et revenir à la classe immuable une fois qu'on en a terminé.

L'exemple ci-dessus peut être modifié pour montrer ce mécanisme :

```
//: appendixa:Immutable2.java
// Une classe compagnon pour modifier
// des objets immuables.

class Mutable {
  private int data;
  public Mutable(int initVal) {
    data = initVal;
  }
  public Mutable add(int x) {
    data += x;
    return this;
  }
  public Mutable multiply(int x) {
    data *= x;
    return this;
  }
  public Immutable2 makeImmutable2() {
    return new Immutable2(data);
  }
}

public class Immutable2 {
  private int data;
  public Immutable2(int initVal) {
    data = initVal;
  }
  public int read() { return data; }
  public boolean nonzero() { return data != 0; }
  public Immutable2 add(int x) {
    return new Immutable2(data + x);
  }
  public Immutable2 multiply(int x) {
    return new Immutable2(data * x);
  }
  public Mutable makeMutable() {
    return new Mutable(data);
  }
  public static Immutable2 modify1(Immutable2 y){
    Immutable2 val = y.add(12);
    val = val.multiply(3);
    val = val.add(11);
    val = val.multiply(2);
    return val;
  }
  // Ceci produit le même résultat :
  public static Immutable2 modify2(Immutable2 y){
```

```
    Mutable m = y.makeMutable();
    m.add(12).multiply(3).add(11).multiply(2);
    return m.makeImmutable2();
  }
  public static void main(String[] args) {
    Immutable2 i2 = new Immutable2(47);
    Immutable2 r1 = modify1(i2);
    Immutable2 r2 = modify2(i2);
    System.out.println("i2 = " + i2.read());
    System.out.println("r1 = " + r1.read());
    System.out.println("r2 = " + r2.read());
  }
} ///:~
```

Immutable2 contient des méthodes qui, comme précédemment, préservent l'immuabilité des objets en créant de nouveaux objets dès qu'une modification est demandée. Ce sont les méthodes **add()** et **multiply()**. La classe compagnon est appelée **Mutable**, et possède aussi des méthodes **add()** et **multiply()**, mais ces méthodes modifient l'objet **Mutable** au lieu d'en créer un nouveau. De plus, **Mutable** possède une méthode qui utilise ses données pour créer un objet **Immutable2** et vice-versa.

Les deux méthodes static **modify1()** et **modify2()** montrent deux approches différentes pour arriver au même résultat. Dans **modify1()**, tout est réalisé dans la classe **Immutable2** et donc quatre nouveaux objets **Immutable2** sont créés au cours du processus (et chaque fois que **val** est réassignée, l'instance précédente est récupérée par le ramasse-miettes).

Dans la méthode **modify2()**, on peut voir que la première action réalisée est de prendre l'objet **Immutable2 y** et d'en produire une forme **Mutable** (c'est comme si on appelait **clone()** vue précédemment, mais cette fois un différent type d'objet est créé). L'objet **Mutable** est alors utilisé pour réaliser un grand nombre d'opérations *sans* nécessiter la création de nombreux objets. Puis il est retransformé en objet **Immutable2**. On n'a donc créé que deux nouveaux objets (l'objet **Mutable** et le résultat **Immutable2**) au lieu de quatre.

Cette approche est donc sensée quand :

1. On a besoin d'objets immuables et
2. On a besoin de faire beaucoup de modifications sur ces objets ou
3. Il est prohibitif de créer de nouveaux objets immuables.

Chaines immuables

Examinons le code suivant:

```
//: appendixa:Stringer.java
public class stringer{
disparait static String upcase(string s) {
```

```
    return s.toUpperCase().
  }
  public static void main(String[] args) {
    String q = new String(" howdy ");
    System.out.println(q) ; // howdy
    String qq = upcase(qq) ; // HOWDY
    System.out.println(q) ; //howdy
  }
} ///:~
```

Quand **q**est passé à **upcase()** c'est en fait une copie de la reference sur **q**. L'objet auquel cette référence est connectée reste dans la même localisation physique. Les références sont copiées quand elles sont passées en argument.

En regardant la definition de **upcase()**, on peut voir que la référence passée en argument porte le nom **s**, et qu'elle existe seulement pendant que le corps de **upcase()** est exécuté. Quand **upcase()** se termine, la référence locale **s** disparaît. **Upcase()** renvoie le résultat, qui est la chaîne originale avec tous ses caractères en majuscules. Bien sûr, elle renvoie en fait une référence sur le resultat. Mais la référence qu'elle renvoie porte sur un nouvel objet, et l'original **q** est laissé inchanger.

Comment cela se fait-il?

Constantes implicites

Si on écrit :

```
String s = " asdf ";
String x = Stringer.upcase(s);
```

Est-ce qu'on veut réellement que la méthode **upcase()** modifie l'argument? En général, non, car pour le lecteur du code, un argument est une information fournie à la méthode et non quelque chose qui puisse être modifié. C'est une garantie importante, car elle rend le code plus facile à lire et à comprendre.

En C++, cette garantie a été jugée suffisamment importante pour justifier l'introduction d'un mot-clef special, **const**, pour permettre au programmeur de s'assurer qu'une référence (pointeur ou référence C++) ne pouvait être utilisée pour modifier l'objet original. Mais le programmeur C++ devait alors faire attention et se rappeler d'utiliser **const** de partout. Cela peut être déroutant et facile à oublier.

Surcharge de l'opérateur « + » et les StringBuffer

Les objets de la classe **String** sont conçu pour être immuables, en utilisant les techniques montrées précédemment. Lorsqu'on exeamine la documaentation en ligne de la classe **String** (qui est résumée un peu plus loin dans cette annexe), on se rend compte que chaque méthode de la classe qui semble modifier un objet **String** crée et renvoie un nouvel objet de la classe contenant la modi-

fication; **la String** original reste inchangée. Il n'y a donc pas de fonctionnalité en Java telle que le const du C++ pour s'assurer de l'immuabilité des objets lors de la compilation. Si on en a besoin, il faut l'implémenter soi-même, comme le fait la classe **String**.

Comme les objets **String** sont immuables, on peut aliaser un objet **String** autant de fois qu'on veut. Puisqu'il est en lecture seule, un référence ne peut rien modifier qui puisse affecter les autres références. Un objet en lecture seule résoud donc élégamment le problème de l'aliasing.

Il semble également possible de gérer tous les cas dans lequel on a besoin de modifier un objet en créant unr nouvelle version de l'objet avec les modifications, comme le fait la classe **String**. Ce n'est toutefois pas efficace pour certaines opérations. C'est le cas de l'opérateur « + » qui a été surchargé pour les objets **String.** Surchargé veut dire qu'un sens supplémentaire lui a été attribué s'il est utilisé avec une classe particulière (les opérateur « + » et « += » pour les **String** sont les seuls opérateurs surchargés en java, et JAVA ne permet pas au programmeur d'en surcharger d'autre) [83].

Quand il est utilisé avec des objets **String**, l'opérateur « + » permet de concaténer des **String**:

```
String s = "abc" + foo + "def" + integer.toString(47);
```

on peut imaginer comment ceci *pourrait* être implémeté: la **String** « abc » pourrait avoir une méthode **append()** qui créerait un nouvel objet **String** contenant la chaîne « abc » concaténée avec le contenu de foo. Le nouvel objet **String** devrait alors créer une nouvelle **String** à laquelle serait ajoutée la chaîne « def », et ainsi de suite.

Cela marcherait bien sûr, mais cela nécessiterait la création d'un grand nombre d'objets **String** rien que pour assembler la nouvelle **String** et donc on se retrouverait avec un ensemble d'objets **String** intermédiaires qui devraient être réclamés par le ramasse-miettes. Je suspecte les concepteurs de Java d'avoir te,té cette approche en premier (une leçon de la conception logiciel – on ne sait réellement rien d'un système tant qu'on n'a pas essaye de la coder et qu'on ne dispose pas d'un système fonctionnel). Je suspecte aussi qu'ils ont decouvert que les performances étaient inacceptables.

La solution consiste en une classe compagnon modifiable similaire à celle exposée précédemment. Pour la classe **String** , cette classe compagnon es tappelée **StringBuffer**, et le compilateur crée automatiquement un objet **StringBuffer** pour évaluer certaines expressions, en particulier quand les opérateurs surchargés « + » et « += » sont utilisés avec des objets **String**. L'exemple suivant montre ce qui se passe :

```
//: appendixa: ImmutableStrings.java
// Démonstration de la classe StringBuffer.
public class ImmutableStrings {
public static void main(String[ ] args) {
String foo = « foo »;
String s = « abc » + foo + « def » + Integer.toString(47);
System.out.println(s);
// L' "équivalent " en utilisant la classe StringBuffer :
```

```
StringBuffer sb = new StringBuffer(« abc »); //Crée une string!
sb.append(foo);
sb.append(« def »); //Crée une String!
sb.append(Integer.toString(47));
System.out.println »sb »;
}
} ///:~
```

Durant la création de la **String s**, le compilateur fait a peu pres l'équivalent du code suivant qui utilise **sb** : une **StringBuffer** est crée et **append()** est appelée pour ajouter les nouveaux caractères directement dans l'objet **StringBuffer** (plutôt que de créer de nouvelles copies à chaque fois). Bien que ceci soit beaucoup plus efficace, il est bon de noter que chaque fois qu'on crée une chaîne de caractères quotée telle que « abc » ou « def », le compilatuer les transforme en objets **String** . Il peut donc y avoir plus d'objets crées que ce qu'on pourrait croire, en dépit de l'efficacité permise par la classe **StringBuffer**.

Les classes String et StringBuffer

Voici un aperçu des méthodes disponibles pour les deux classes **String** et **StringBuffer** afin de se faire une idée de la manière dont elles interagissent. Ces tables ne référencent pas toutes les méthodes disponibles, mais seulement celles qui sont importantes pour notre sujet. Les méthodes surchargées sont résumées sur une ligne.

Methode	Arguments, Surcharges	Utilisation
Constructeur	Surchargés : Defaut, **String**, **StringBuffer**, tableau de **char**, tableau de **byte**.	Création d'objets **String**.
length()		Nombre de cractères dans la **String.**
charAt()	**Int** Index	Le n-ième caractère dans **String.**
GetChars(), getBytes()	Le debut et la fin de la zone à copier, le tableau dans lequel efectuer la copie, un index dans le tableau de destination.	Copie des **chars** ou des **bytes** dans un tableau externe.
ToCharArray()		Produit un **char[ ]** contenant les caractères de la **String.**
Equals(), equalsIgnoreCase()	Une **String** avec laquelle comparer.	Un test d'égalité sur le contenu de deux **String.**
compareTo()	Une **String** avec laquelle comparer	Le resultat est négatif, zéro ou positif suivant la comparaison léxicographique de la **String** et

		l'argument. Majuscules et minuscules sont différentiées !
regionMatches()	Déplacement dans la **String**, une autre **String** ainsi que son déplacement et la longueur à comparer. Surchargé : ajoute l'insensiblisation à la casse.	Résultat **boolean** indiquant si les régions correspondent.
startsWith()	La String avec laquelle la **String** pourrait débuter. Surchargé : ajoute un déplacement pour l'argument.	Résultat **boolean** indiquant si la **String** débute avec l'argument.
endsWith()	La **String** qui pourrait être un suffixe de cette **String**.	Résultat **boolean** indiquant si l'argument est un suffixe de la **String**.
Indexof(), lastIndexOf()	Surchargés : char, char et index de départ, **String**, **String** et index de départ.	Renvoie -1 si l'argument n'est pas trouvé dans la **String**, sinon renvoie l'index où l'argument commence. **LastIndexOf()** recherche en partant de la fin.
substring()	Surchargé : index de départ, index de départ et index de fin.	Renvoie un nouvel objet **String** contenant l'ensemble des caractères spécifiés.
concat()	La **String** à concaténer.	Renvoie un nouvel objet **String** contenant les caractères de l'objets **String** initial suivis des caractères de l'argument.
replace()	L'ancien caractère à rechercher, le nouveau caractère avec lequel le remplacer.	Renvoie un nouvel objet **String** avec les remplacements effectués. Utilise l'ancienne **String** si aucune correspondance n'est trouvée.
ToLowerCase(), toUpperCase()		Renvoie un nouvel objet **String** avec la casse de tous les carctères modifiée. Utilise l'ancienne **String** si aucun changement n'a été effectué.
trim()		Renvoie un nouvel objet **String** avec les espaces enlevés à chaque extrémité. Utilise l'ancienne **String** si aucun changement n'a été effectué.
valueOf()	Surchargés : Objetc, char [ ], char [ ] et déplacement et compte, boolean, char, int ,	Renvoie une **String** contenant un représentation sous forme de chaîne de l'argument.

	long, float, double.	
intern()		Produit une et une seule référence **String** pour chaque séquence unique de caractères.

On peut voir que chaque méthode de la classe **String** renvoie un nouvel objet **String** quand il est nécessaire d'en modifier le contenu. Remarquez ausii que si le contenu n'a pas besoin d'être modifié la méthode renverra juste une référence sur la **String** originale. Cela permet d'économiser sur le stockage et le surcout d'une création.

Et voivi la classe **Stringbuffer** :

Méthode	Arguments, Surcharges	Utlisation
Constructeur	Surchargés : défaut, longueur du buffer à créer, **String** à partir de laquelle créer.	Crée un nouvel objet **Stringbuffer**.
toString()		Crée un objet **String** à partir de l'objet **Stringbuffer**.
length()		Nombre de caractères dans l'objet **Stringbuffer**.
capacity()		Renvoie l'espace actullement alloué.
ensure-capacity()	Entier indiquant la capacité désirée.	Fait que l'objet **Stringbuffer** alloue au minimun un certain espace.
setLength()	Entier indiquant la nouvelle longueur de la chaîne de caractères dans le buffer.	Tronque ou accroit la chaîne de caractères. Si acroissement, complète avec des caractères nulls.
charAt()	Entier indiquant la localisation de l'élément désiré et la nouvelle valeur char de cet élément.	Modifie la valeur à un endroit précis.
getChar()	Le début et la fin à partir de laquelle copier, le tableau dans lequel copier, un index dans le tableau de destination.	Copie des **chars** dans un tableau extérieur. Il n'existe pas de **getBytes()** comme dans la classe **string**.
append()	Surchargés : **Object, Sring, char [ ], char[ ]** avec déplacement et longueur, **boolean, char, int, long, float, double**.	L'argument est converti en chaîne de caractères et concaténé à la fin du buffer courant, en augmentant la taille du buffer si nécessaire.
insert()	Surchargés, chacuns avec un premier argument contenant le	Le deuxième argument est converti en chaîne de caractères

	déplacement auquel on débute l'insertion : **Object, String, char [ ], boolean, char, int, long, float, double.**	et inséré dans le buffer courant au déplacement spécifié. La taille du buffer est augmentée si nécessaire.
reverse()		L'ordre des caractères du buffer est inversé.

La méthode la plus fréquemment utilisée est **append()**, qui est utilisée par le compilateur quand il évalue des expressions **String** contenant les opérateurs « + » et « += ». La méthode **insert()** a une forme similaire, et les deux méthodes réalisent des modifications significatives sur le buffer au lieu de créer de nouveaux objets.

Les Strings sont spéciales

La classe **String** n'est donc pas une simple classe dans Java. La classe String est spéciale à bien des égards, en particulier parce que c'est une classe intégrée et fondamentale dans Java. Il y a le fait qu'une chaîne de caractères entre quotes est convertie en **String** par le compilateur et les opérateurs « + » et « += ». Dans cette annexe vous avez pu voir d'autres spécificités : l'immuabilité précautionneusement assurée par la classe compagnon **StringBuffer** et d'autres particularités magiques du compilateur.

Résumé

Puisque dans Java tout est référence, et puisque chaque objet est crée dans le segment et réclamée par la ramasse-miettes seulement quand il n'est plus utilisé, la façon de manipuler les objets change, spécialement lors du passage et du retour d'ojets. Par exemple, en C ou en C++, si on veut initialiser un endroit de stockage dans une méthode, il faut demander à l'utilisateur de passer l'adresse de cet endroit de stockage à la méthode, ou alors il faut se mettre d'accord sur qui a la responsabilité de détruire cet espace de stockage. L'interface et la comprehension de telles méthodes sont donc bien plus compliquées. Mais en Java, on n'a pas à se soucier de savoir si un objet existera toujours lorsqu'on en aura besoin, puisque tout est déjà gére pour nous. On peut ne créer un objet que quand on en a besoin, et pas avant, sans se soucier de déléguer les responsabilités sur cet objet : on passe simplement une référence. Quelquefois la simplification que cela engendre passe inaperçue, d'autres fois elle est flagrante.

Les inconvénients de cette magie sous-jacente sont doubles :

1. Il y a toujours une pénalité d'efficacité pour cette gestion supplémetaire de la mémoire (bien qu 'elle puisse être relativement faible), et il y a toujours une petite incertitude sur le temps d'exécution (puisque le ramasse-meittes paut être obligé d'entrer en action si la quantité de mémoire disponible n'est pas suffisante). Pour la plupart des applications, les bénéfices compensent largement les inconvénients, et les parties de l'application critiques quand au temps peuvent êter écrite en utilisant des méthodes **natives** (voir annex B).

2. aliasing : on peut se retrouver accidentellement avec deux références sur le même objet, ce qui est un problème si les deux références sont supposées pointer sur deux objets *distincts*. Cela demande de faire un peu plus attention, et, si nécessaire, **clone()** l'objet pour empêcher toute modification intempestive de l'objet par l'autre référence. Cependant on peut

supporter l'aliasing pour l'efficacité qu'il procure en évitant cet inconvénient en créant des objets immuables dont les opérations renvoient un nouvel objet du même type ou d'un type différend, mais ne changent pas l'objet original afin que toute référence liée à cet objet ne voie pas de changements.

Certaines personnes insinuent que la conception du clonage a été baclée en Java, et pour ne pas s'embêter avec, implémentent leur propre version du clonage, sans jamais appeler la méthode **Object.clone()**, éliminant ainsi le besoin d'implémenter **Cloneable** et d'intercepter l'exception **CloneNotSupportedException**. Ceci est certainement une approche raisonnable et comme **clone()** est très peu implémentée dans la bibliothèque standard Java, elle est ausii relativement sûre. Mais tant qu'on appelle pas **Object.clone()** on n'a pas besoin d'implémenter **Cloneable** ou d'intercepter l'exception, donc cela semble acceptable aussi.

Exercices

Les solutions aux exercices sélectionnés peuvent être trouvées dans le document électronique *The Thinking in Java Annotated Solution Guide*, disponible pour une faible somme sur *www.BruceEckel.com*.

1. Démontrer un second niveau d'aliasing. Créer une méthode qui crée une référence sur un objet mais ne modifie pas cet objet. Par contre, la méthode appelle une seconde méthode, lui passant la référence, et cette seconde méthode modifie l'objet.
2. Créer une classe **myString** contenant un objet **String** qu'on initialisera dans le constructeur en utilisant l'argument du constructeur. Ajouter une méthode **toString()** et une méthode **concatenate()** qui ajoute un objet **String** à la chaîne interne. Implémenter **clone()** dans **myString**. Créer deux méthodes **static** acceptant chacune une référence **myString x** comme argument et appellant **x.concatenate("test")**, mais la seconde méthode appelle **clone()** d'abord. Tester les deux méthodes et montrer les différents effets.
3. Créer une classe **Battery** contenant un **int** qui est un numéro de batterie (comme identifiant unique). La rendre cloneable et lui donner une méthode **toString()**. Créer maintenant une classe **Toy** qui contienne un tableau de **Battery** et une méthode **toString()** qui affiche toutes les **Battery**. Ecrire une méthode **clone()** pour la classe **Toy** qui clone automatiquement tous ses objets **Battery**. Tester cette méthode en clonant **Toy** et en affichant le résultat.
4. Changer **CheckCloneable.java** afin que toutes les méthodes **clone()** interceptent l'exception **CloneNotSupportedException** plutôt que de la faire suivre à l'appelant.
5. En utilisant la technique de la classe compagnon modifiable, créer une classe immuable contenant un **int**, un **double** et un tableau de **char**.
6. Modifier **Compete.java** pour ajouter de nouveaux objets membres aux classes **Thing2** et **Thing4**, et voyez si vous pouvez déterminer comment le minutage varie avec la complexité - si c'est une simple relation linéaire ou si cela semble plus compliqué.
7. Modifier **Compete.java** pour ajouter de nouveaux objets membres aux classes **Thing2** et **Thing4**, et voyez si vous pouvez déterminer comment le minutage varie avec la complexité - si c'est une simple relation linéaire ou si cela semble plus compliqué.
8. Hériter d'une **ArrayList** et faire que sa méthode **clone()** réalise une opération de copie profonde.

[79]En C, qui généralement gère des données de petite taille, le passage d'arguments par dé-

faut se fait par valeur. C++ a dû suivre cette norme, mais avec les objets, le passage par valeur n'est pas la façon de faire la plus efficace. De plus, coder une classer afin qu'elle supporte le passage par valeur en C++ est un gros casse-tête.

[80]Ce n'est pas l'orthographe correcte du mot clonable (N.d.T. : en français comme en anglais), mais c'est de cette manière qu'il est utilisé dans la bibliothèque Java ; j'ai donc décidé de l'utiliser ainsi ici aussi, dans l'espoir de réduire la confusion.

[81]On peut apparemment créer un simple contre-exemple de cette affirmation, comme ceci :

```
public class Cloneit implements Cloneable {
  public static void main (String[] args)
  throws CloneNotSupportedException {
    Cloneit a = new Cloneit();
    Cloneit b = (Cloneit)a.clone();
  }
}
```

Toutefois, ceci ne marche que parce que **main( )** est une méthode de **Cloneit** et donc a la permission d'appeler la méthode **protected clone()** de la classe de base. Si on l'appelle depuis une autre classe, la compilation générera des erreurs.

[82]L'avocat excepté, qui a été reclassifié en « aliment gras ».

[83]C++ permet au programmeur de surcharger les opérateurs comme il l'entend. Comme ceci se révèle souvent être un processus compliqué (voir le Chapitre 10 de *Thinking in C++, 2nd edition*, Prentice-Hall, 2000), les concepteurs de Java ont jugé une telle fonctionnalité « non souhaitable » qui ne devait pas donc pas être intégrée dans Java. Elle n'était apparemment pas si indésirable que cela, puisqu'eux-mêmes l'ont finalement utilisée, et ironiquement, la surcharge d'opérateurs devrait être bien plus facile à utiliser en Java qu'en C++. On peut le voir dans Python (voir www.python.org) qui dispose d'un ramasse-miettes et d'un mécanisme de surcharge d'opérateurs très simple à mettre en oeuvre.

[84]Doug Lea, qui m'a été d'une aide précieuse sur ce sujet, m'a suggéré ceci, me disant qu'il créait simplement une fonction **duplicate()** pour chaque classe.

Annexe B - L'Interface Java Natif (JNI)

Le contenu de cette annexe a été fourni et est utilisé avec la permission d'Andrea Provaglio (www.AndreaProvaglio.com).

Le langage Java et son API standard sont suffisamment riches pour écrire des applications complètes. Mais dans certains cas on doit appeler du code non-Java ; par exemple, pour accéder à des fonctionnalités spécifiques du système d'exploitation, s'interfacer avec des matériels particuliers, réutiliser une base de code non-Java existante, ou implémenter des parties de codes à contraintes temps réel fortes.

S'interfacer avec du code non-Java nécessite un support dédié dans le compilateur et dans la machine virtuelle, ainsi que des outils supplémentaires pour associer le code Java au code non-Java. La solution standard fournie par JavaSoft pour appeler du code non-Java est appelée la *Java Native Interface*, qui sera introduite dans cette annexe. Ceci n'est pas un traitement en profondeur, et dans certains cas vous êtes supposés avoir une connaissance partielle des concepts et techniques concernés.

JNI est une interface de programmation assez riche qui permet d'appeler des méthodes natives depuis une application Java. Elle a été ajoutée dans Java 1.1, en maintenant un certain degré de compatibilité avec son équivalent en Java 1.0 : la native method interface (NMI). NMI a des caractéristiques de conception qui la rendent impropre à l'utilisation sur certaines machines virtuelles. Pour cette raison, les versions ultérieures du langage pourraient ne plus supporter NMI, et elle ne sera pas couverte ici.

Actuellement, JNI est conçue pour s'interfacer uniquement avec les méthodes natives écrites en C ou C++. En utilisant JNI, vos méthodes natives peuvent :

- créer, inspecter et modifier des objets Java (y compris les tableaux et les **String**s),
- appeler des méthodes Java,
- intercepter *[catch]* et générer *[throw]* des exceptions,
- charger des classes et obtenir des informations sur les classes,
- effectuer des contrôles lors de l'exécution (*run-time type checking*).

De ce fait, pratiquement tout ce qu'on peut faire avec des classes et des objets en Java ordinaire, on peut aussi le faire dans des méthodes natives.

Appeler une méthode native

Nous allons commencer avec un exemple simple : un programme Java qui appelle une méthode native, qui à son tour appelle la fonction **printf( )** de la bibliothèque C standard.

La première opération consiste à écrire le code Java qui déclare une méthode native et ses arguments :

```
//: appendixb:ShowMessage.java
public class ShowMessage {
  privatenative void ShowMessage(String msg);
  static {
```

```
System.loadLibrary("MsgImpl");
  // Astuce Linux, si vous ne pouvez pas
  // positionner le chemin des bibliothèques
  // dans votre environnement :
  // System.load(
  //"/home/bruce/tij2/appendixb/UseObjImpl.so");
}
 public static void main(String[] args) {
ShowMessage app = new ShowMessage();
app.ShowMessage("Generated with JNI");
}
} ///:~
```

La déclaration de la méthode native est suivie par un bloc **static** qui appelle **System.loadLibrary( )** (qu'on pourrait appeler à n'importe quel moment, mais ce style est plus clair). **System.loadLibrary( )** charge une DLL en mémoire et se lie à elle. La DLL doit être dans votre chemin d'accès aux bibliothèques système (*system library path*). L'extension de nom de fichier est automatiquement ajoutée par la JVM dépendant de la plateforme.

Dans le code ci-dessus on peut voir également un appel à la méthode **System.load( )** , qui est mis en commentaire. Le chemin spécifié ici est un chemin absolu, plutôt que dépendant d'une variable d'environnement. Utiliser une variable d'environnement est naturellement une meilleure solution et est plus portable, mais si vous n'y arrivez pas vous pouvez mettre en commentaire l'appel à **loadLibrary( )** et enlever la mise en commentaire de l'appel à la méthode System.load(), en ajustant le chemin de votre propre répertoire.

Le générateur d'entête *[header file generator]* : javah

Maintenant, compilez votre fichier source Java et lancez **javah** sur le fichier **.class** résultant, en précisant le paramètre **-jni** (ceci est déjà fait dans le makefile inclus dans la fourniture du code source pour ce livre) :

```
javah -jni ShowMessage
```

javah lit la classe Java et pour chaque déclaration de méthode native il génère un prototype de fonction dans un fichier d'entête C ou C++. Voici le résultat : le fichier source **ShowMessage.h** (légèrement modifié pour l'inclure dans ce livre) :

```
/* DO NOT EDIT THIS FILE
- it is machine generated */
#include <jni.h>
/* Header for class ShowMessage */

#ifndef _Included_ShowMessage
#define _Included_ShowMessage
#ifdef __cplusplus
extern "C" {
#endif
/*
```

```
 * Class:     ShowMessage
 * Method:    ShowMessage
 * Signature: (Ljava/lang/String;)V
 */
JNIEXPORT void JNICALL
Java_ShowMessage_ShowMessage
(JNIEnv *, jobject, jstring);

#ifdef __cplusplus
}
#endif
#endif
```

Comme on peut le voir par la directive préprocesseur **#ifdef __cplusplus** , ce fichier peut être compilé aussi bien par un compilateur C que C++. La première directive **#include** inclut **jni.h**, un fichier d'entête qui, entre autres choses, définit les types qu'on peut voir utilisés dans le reste du fichier. **JNIEXPORT** et **JNICALL** sont des macros qui une fois étendues génèrent les directives spécifiques aux différentes plateformes. **JNIEnv**, **jobject** et **jstring** sont des définitions de types de données, qui seront expliquées par la suite.

Les conventions de nommage *[name mangling]*et les signatures de fonctions

JNI impose une convention de nommage (appelée *name mangling*) aux méthodes natives. Ceci est important car cela fait partie du mécanisme par lequel la machine virtuelle lie les appels Java aux méthodes natives. Fondamentalement, toutes les méthodes natives commencent par le mot "Java", suivi par le nom de la méthode native. Le caractère sous-tiret est utilisé comme séparateur. Si la méthode java native est surchargée, alors la signature de fonction est ajoutée au nom également ; on peut voir la signature native dans les commentaires précédant le prototype. Pour plus d'-informations sur les conventions de nommage, se référer à la documentation de JNI.

Implémenter votre DLL

Arrivé ici, il ne reste plus qu'à écrire un fichier source C ou C++ qui inclut le fichier d'entête généré par **javah** et implémenter la méthode native, puis le compiler et générer une bibliothèque dynamique. Cette opération dépend de la plateforme. Le code ci-dessous est compilé et lié dans un fichier appelé **MsgImpl.dll** pour Windows ou **MsgImpl.so** pourUnix/Linux (le makefile fourni avec les listings du code contient les commandes pour faire ceci ; il est disponible sur le CD ROM fourni avec ce livre, ou en téléchargement libre sur *www.BruceEckel.com*) :

```
//: appendixb:MsgImpl.cpp
//# Testé avec VC++ & BC++. Le chemin d'include
//# doit être adapté pour trouver les en-têtes
//# JNI. Voir le makefile de ce chapitre
//# (dans le code source téléchargeable)
//# pour exemple.
#include <jni.h>
#include <stdio.h>
#include "ShowMessage.h"
```

```
extern "C" JNIEXPORT> void JNICALL
Java_ShowMessage_ShowMessage(JNIEnv* env,
jobject, jstring jMsg) {
  const char* msg=env->GetStringUTFChars(jMsg,0);
printf("Thinking in Java, JNI: %s\n", msg);
env->ReleaseStringUTFChars(jMsg, msg);
///:~
```

Les arguments qui sont passés à la méthode native permettent de revenir dans Java. Le premier, de type **JNIEnv**, contient tous les points d'ancrage *[hooks]* (nous allons revenir là-dessus dans la section suivante). Le deuxième argument a une signification différente selon le type de la méthode. Pour des méthodes non-**static** comme dans l'exemple ci-dessus, le deuxième argument est l'équivalent du pointeur "this" en C++ et similaire au **this** en Java : c'est une référence à l'objet qui a appelé la méthode native. Pour les méthodes **static**, c'est une référence à l'objet **Class** où la méthode est implémentée.

Les arguments restants représentent les objets java passés dans l'appel de la méthode native. Les scalaires sont aussi passés de cette façon, mais en les passant par valeur.

Dans les sections suivantes nous allons expliquer ce code en regardant de quelle façon on accède à la JVM et comment on la contrôle depuis l'intérieur d'une méthode native.

Accéder à des fonctions JNI : l'argument JNIEnv

Les programmeurs utilisent les fonctions JNI pour interagir avec la JVM depuis une méthode native. Comme on l'a vu dans l'exemple ci-dessus, chaque méthode native JNI reçoit un argument spécial comme premier paramètre : l'argument JNIEnv, qui est un pointeur vers une structure de données JNI spéciale de type **JNIEnv_**. Un élément de la structure de données JNI est un pointeur vers un tableau généré par la JVM. Chaque élément de ce tableau est un pointeur vers une fonction JNI. Les fonctions JNI peuvent être appelée depuis la méthode native en déréférençant ces pointeurs (c'est plus simple qu'il n'y paraît). Chaque JVM fournit sa propre implémentation des fonctions JNI, mais leurs adresses seront toujours à des décalages prédéfinis.

A l'aide de l'argument **JNIEnv**, le programmeur a accès à un large éventail de fonctions. Ces fonctions peuvent être regoupées selon les catégories suivantes :

- obtenir une information de version,
- effectuer des opérations sur les classes et les objets,
- gérer des références globales et locales à des objets Java,
- accéder à des champs d'instances et à des champs statiques *[instance fields and static fields],*
- appeler des méthodes d'instances et des méthodes statiques,
- effectuer des opérations sur des chaînes et des tableaux,
- générer et gérer des exceptions Java.

Il existe de nombreuses fonctions JNI qui ne seront pas couvertes ici. A la place, je vais montrer le principe d'utilisation de ces fonctions. Pour des informations plus détaillées, consultez la

documentation JNI de votre compilateur.

Si on jette un coup d'oeil au fichier d'entête **jni.h** , on voit qu'à l'intérieur de la directive pré-processeur **#ifdef __cplusplus**, la structure **JNIEnv_** est définie comme une classe quand elle est compilée par un compilateur C++. Cette classe contient un certain nombre de fonctions inline qui permettent d'accéder aux fonctions JNI à l'aide d'une syntaxe simple et familière. Par exemple, la ligne de code C++ de l'exemple précédent :

```
env->ReleaseStringUTFChars(jMsg, msg);
```

pourrait également être appelée en C comme ceci :

```
(*env)->ReleaseStringUTFChars(env, jMsg, msg);
```

On notera que le style C est (naturellement) plus compliqué ; on doit utiliser un double déréférencement du pointeur **env**, et on doit aussi passer ce même pointeur comme premier paramètre de l'appel à la fonction JNI. Les exemples de cette annexe utilisent le style C++.

Accéder à des chaînes Java

Comme exemple d'accès à une fonction JNI, observez le code dans **MsgImpl.cpp**. Ici, l'argument **JNIEnv env** est utilisé pour accéder à une **String** Java. Les **String**s Java sont au format Unicode, donc si on en reçoit une et qu'on veut la passer à une fonction non-Unicode (**printf( )**, par exemple), il faut d'abord la convertir en caractères ASCII avec la fonction JNI **GetStringUTFChars( )**. Cette fonction prend une **String** Java et la convertir en caractères UTF-8 (ceux-ci font 8 bits pour contenir de valeurs ASCII ou 16 bits pour contenir de l'Unicode ; si le contenu de la chaîne d'origine était composée uniquement d'ASCII, la chaîne résultante sera également de l'ASCII).

GetStringUTFChars( )> est une des fonctions membres de **JNIEnv**. Pour accéder à la fonction JNI, on utilise la syntaxe typique C++ pour appeler une fonction membre à l'aide d'un pointeur. On utilise la forme ci-dessus pour appeler l'ensemble des fonctions JNI.

Passer et utiliser des objets Java

Dans l'exemple précédent nous avons passé une String à la méthode native. On peut aussi passer des objets Java de sa propre création à une méthode native. A l'intérieur de sa méthode native, on peut accéder aux champs *[fields]* et aux méthodes de l'objet reçu.

Pour passer des objets, on utilise la syntaxe Java normale quand on déclare la méthode native. Dans l'exemple ci-dessous, **MyJavaClass** a un champ **public** et une méthode **public**. La classe **UseObjects** déclare une méthode native qui prend un objet de la classe **MyJavaClass**. Pour voir si la méthode native manipule son argument, le champ **public** de l'argument est positionné, la méthode native est appelée, et enfin la valeur du champ **public** est imprimée.

```
//: appendixb:UseObjects.java
class MyJavaClass {
  public int aValue;
  public void divByTwo() { aValue /= 2; }
}
public class UseObjects {
  private native void
```

```
changeObject(MyJavaClass obj);
  static {
System.loadLibrary("UseObjImpl");
   // Astuce Linux, si vous ne pouvez pas
   // positionner le chemin des bibliothèques
   // dans votre environnement :
   // System.load(
   //"/home/bruce/tij2/appendixb/UseObjImpl.so");
}
  public static void main(String[] args) {
UseObjects app = new UseObjects();
MyJavaClass anObj = new MyJavaClass();
anObj.aValue = 2;
app.changeObject(anObj);
System.out.println("Java: " + anObj.aValue);
}
} ///:~
```

Après avoir compilé le code et exécuté **javah**, on peut implémenter la méthode native. Dans l'exemple ci-dessous, après avoir obtenu les identificateurs du champ et de la méthode, on y accède à l'aide de fonctions JNI.

```
//: appendixb:UseObjImpl.cpp
//# Testé avec VC++ & BC++. Le chemin d'include
//# doit être adapté pour trouver les en-têtes
//# JNI. Voir le makefile de ce chapitre
//# (dans le code source téléchargeable)
//# pour exemple.

#include <jni.h>
extern "C" JNIEXPORT void JNICALL
Java_UseObjects_changeObject(
JNIEnv* env, jobject, jobject obj) {
jclass cls = env->GetObjectClass(obj);
jfieldID fid = env->GetFieldID(
cls, "aValue", "I");
jmethodID mid = env->GetMethodID(
cls, "divByTwo", "()V");
  int value = env->GetIntField(obj, fid);
printf("Native: %d\n", value);
env->SetIntField(obj, fid, 6);
env->CallVoidMethod(obj, mid);
value = env->GetIntField(obj, fid);
printf("Native: %d\n", value);
} ///:~
```

Ignorant l'équivalent de "this", la fonction C++ reçoit un jobject, qui est l'aspect natif de la référence à l'objet Java que nous passons depuis le code Java. Nous lisons simplement **aValue**, l'im-

primons, changeons la valeur, appelons la méthode **divByTwo()** de l'objet, et imprimons la valeur à nouveau.

Pour accéder à un champ ou une méthode Java, on doit d'abord obtenir son identificateur en utilisant **GetFieldID( )** pour les champs et **GetMethodID( )** pour les méthodes. Ces fonctions prennent la classe, une chaîne contenant le nom de l'élément, et une chaîne donnant le type de l'-information : le type de donnée du champ, ou l'information de signature d'une méthode (des détails peuvent être trouvés dans la documentation de JNI). Ces fonctions retournent un identificateur qu'on doit utiliser pour accéder à l'élément. Cette approche pourrait paraître tordue, mais notre méthode n'a aucune connaissance de la disposition interne de l'objet Java. Au lieu de cela, il doit accéder aux champs et méthodes à travers les index renvoyés par la JVM. Ceci permet à différentes JVMs d'implémenter différentes dispositions des objets sans impact sur vos méthodes natives.

Si on exécute le programme Java, on verra que l'objet qui est passé depuis le côté Java est manipulé par notre méthode native. Mais qu'est ce qui est exactement passé ? Un pointeur ou une référence Java? Et que fait le ramasse-miettes pendant l'appel à des méthodes natives ?

Le ramasse-miettes continue à travailler pendant l'exécution de la méthode native, mais il est garanti que vos objets ne seront pas réclamés par le ramasse-miettes durant l'appel à une méthode native. Pour assurer ceci, des *références locales* sont créées auparavant, et détruites juste après l'appel à la méthode native. Puisque leur durée de vie englobe l'appel, on sait que les objets seront valables tout au long de l'appel à la méthode native.

Comme ces références sont créées et ensuite détruites à chaque fois que la fonction est appelée, on ne peut pas faire des copies locales dans les méthodes natives, dans des variables **static**. Si on veut une référence qui dure tout le temps des appels de fonctions, on a besoin d'une référence globale. Les références globales ne sont pas créées par la JVM, mais le programmeur peut créer une référence globale à partir d'une locale en appelant des fonctions JNI spécifiques. Lorsqu'on crée une référence globale, on devient responsable de la durée de vie de l'objet référencé. La référence globale (et l'objet qu'il référence) sera en mémoire jusqu'à ce que le programmeur libère explicitement la référence avec la fonction JNI appropriée. C'est similaire à **malloc()** et **free()** en C.

JNI et les exceptions Java

Avec JNI, les exceptions Java peuvent être générées, interceptées, imprimées et retransmises exactement comme dans un programme Java. Mais c'est au programmeur d'appeler des fonctions JNI dédiées pour traiter les exceptions. Voici les fonctions JNI pour gérer les exceptions :

- **Throw( )** Émet un objet exception existant. Utilisé dans les méthodes natives pour réémettre une exception.
- **ThrowNew( )** Génère un nouvel objet exception et l'émet.
- **ExceptionOccurred( )** Détermine si une exception a été émise et pas encore effacée *[cleared]*.
- **ExceptionDescribe( )** Imprime une exception et la trace de la pile *[stack trace]*.
- **ExceptionClear( )>** Efface une exception en cours.
- **FatalError( )** Lève *[raises]* une erreur fatale. Ne revient pas *[does not return]*.

Parmi celles-ci, on ne peut pas ignorer **ExceptionOccurred( )** et **ExceptionClear( )**. La plupart des fonctions JNI peuvent générer des exceptions, et comme le langage ne dispose pas de

construction équivalente à un bloc try Java, il faut appeler **ExceptionOccurred( )** après chaque appel à une fonction JNI pour voir si une exception a été émise. Si on détecte une exception, on peut choisir de la traiter (et peut-être la réémettre). Il faut cependant s'assurer qu'une exception est effacée en final. Ceci peut être fait dans notre fonction en utilisant **ExceptionClear( )** ou dans une autre fonction si l'exception est réémise, mais cela doit être fait.

Il faut s'assurer que l'exception est effacée, sinon les résultats seront imprévisibles si on appelle une fonction JNI alors qu'une exception est en cours. Il y a peu de fonctions JNI qu'on peut appeler sans risque durant une exception ; parmi celles-ci, évidemment, il y a toutes les fonctions de traitement des exceptions.

JNI et le threading

Puisque Java est un langage multithread, plusieurs threads peuvent appeler une méthode native en même temps (la méthode native peut être suspendue au milieu de son traitement lorsqu'un second thread l'appelle). C'est au programmeur de garantir que l'appel natif est "thread-safe" ; c'est à dire qu'il ne modifie pas des données partagées de façon non contrôlée. Fondamentalement, il y a deux possibilités : déclarer la méthode native **synchronized**, ou implémenter une autre stratégie à l'intérieur de la méthode native pour assurer une manipulation correcte des données concurrentes.

De plus, on ne devrait jamais passer le pointeur JNIEnv à travers des threads, car la structure interne sur laquelle il pointe est allouée thread par thread et contient des informations qui n'ont de sens que dans ce thread particulier.

Utiliser une base de code préexistantes

La façon la plus simple d'implémenter des méthodes JNI natives est de commencer à écrire des prototypes de méthodes natives dans une classe Java, de compiler cette classe, et exécuter le fichier **.class** avec **javah**. Mais si on possède une large base de code préexistante qu'on désire appeler depuis Java ? Renommer toutes les fonctions de notre DLL pour les faire correspondre aux conventions du JNI name mangling n'est pas une solution viable. La meilleure approche est d'écrire une DLL d'encapsulation "à l'extérieur" de votre base de code d'origine. Le code Java appelle des fonctions de cette nouvelle DLL, qui à son tour appelle les fonctions de votre DLL d'origine. Cette solution n'est pas uniquement un contournement ; dans la plupart des cas on doit le faire de toutes façons parce qu'on doit appeler des fonctions JNI sur les références des objets avant de pouvoir les utiliser.

Information complémentaire

Vous pouvez trouver des éléments d'introduction, y compris un exemple C (plutôt que C++) et une discussion des problèmes Microsoft, dans l'Annexe A de la première édition de ce livre, que vos pouvez trouver sur le CD ROM fourni avec ce livre, ou en téléchargement libre sur *www.BruceEckel.com*. Des informations plus complètes sont disponibles sur java.sun.com (dans le moteur de recherche, sélectionnez "training & tutorials" avec "native methods" comme mots-clés). Le chapitre 11 de *Core Java 2*, *Volume II*, par Horstmann & Cornell (Prentice-Hall, 2000) donne une excellente description des méthodes natives.

Annexe C - Conseils pour une programation stylée en java

Cette annexe contient des conseils destinés à vous aider à structurer vos programmes et écrire le code source de ceux-ci.

Bien entendu, ce ne sont que des suggestions, en aucun cas des règles à suivre à la lettre. L'idée ici est de s'en inspirer, tout en se rappelant que certaines situations imposent de faire une entorse aux règles.

Conception

1. **L'élégance est toujours récompensée.** C'est vrai que cela peut prendre un peu plus de temps pour arriver à une solution élégante du problème considéré, mais les bénéfices en sont tout de suite apparents (même si personne ne peut les quantifier réellement) lorsque le programme marche du premier coup et s'adapte facilement à de nouvelles situations plutôt que de devoir se battre avec pendant des heures, des jours et des mois. Non seulement le programme est plus facile à écrire et à débugguer, mais il est aussi plus facile à comprendre et à maintenir, et c'est là que réside sa valeur financière. Ce point demande de l'expérience pour être complètement assimilé, parce qu'on est tenté de se dire que la recherche d'un code élégant fait baisser la productivité. Se précipiter ne peut que vous ralentir.

2. **D'abord le faire marcher, ensuite l'optimiser.** Ceci est vrai même si on est certain qu'une portion de code est réellement importante et sera un goulot d'étranglement pour le système. D'abord faire fonctionner le système avec une conception aussi simple que possible. Ensuite le profiler s'il n'est pas assez rapide : on se rendra compte la plupart du temps que le problème ne se situe pas là où on pensait qu'il serait. Il faut préserver son temps pour les choses réellement importantes.

3. **Se rappeler le principe « Diviser pour mieux régner ».** Si le problème considéré est trop embrouillé, il faut essayer d'imaginer quelles opérations ferait le programme s'il existait une « entité magique » qui prendrait en charge les parties confuses. Cette entité est un objet - il suffit d'écrire le code qui utilise l'objet, puis analyser cet objet et encapsuler *ses* parties confuses dans d'autres objets, et ainsi de suite.

4. **Séparer le créateur de la classe de l'utilisateur de la classe (le** *programmeur client*).L'utilisateur de la classe est le « client », il n'a pas besoin et ne veut pas savoir ce qui se passe dans les coulisses de la classe. Le créateur de la classe doit être un expert en conception et écrire la classe de manière à ce qu'elle puisse être utilisée même par le plus novice des programmeurs, tout en remplissant efficacement son rôle dans le fonctionnement de l'application. L'utilisation d'une bibliothèque ne sera aisée que si elle est transparente.

5. **Quand on crée une classe, essayer de choisir des noms explicites afin de rendre les commentaires inutiles.** Le but est de présenter une interface conceptuellement simple au programmeur client. Ne pas hésiter à surcharger une méthode si cela est nécessaire pour proposer une interface intuitive et facile à utiliser.

6. **L'analyse et la conception doivent fournir, au minimum, les classes du système, leur interface publique et leurs relations avec les autres classes, en particulier les**

classes de base. Si la méthodologie de conception produit plus de renseignements que cela, il faut se demander si tout ce qui est produit a de la valeur pendant toute la durée du vie du programme. Si ce n'est pas le cas, les maintenir sera coûteux. Les membres d'une équipe de développement tendent à « oublier » de maintenir tout ce qui ne contribue pas à leur productivité ; ceci est un fait que beaucoup de méthodologie de conception négligent.

7. **Tout automatiser.** D'abord écrire le code de test (avant d'écrire la classe), et le garder avec la classe. Automatiser l'exécution des tests avec un Makefile ou un outil similaire. De cette manière, chaque changement peut être automatiquement vérifié en exécutant les tests, et les erreurs seront immédiatement détectées. La présence du filet de sécurité que constitue la suite de tests vous donnera plus d'audace pour effectuer des changements radicaux quand vous en découvrirez le besoin. Se rappeler que l'amélioration la plus importante dans les langages de programmation vient des tests intégrés fournis par la vérification de type, la gestion des exceptions, etc. mais que ces tests ne sont pas exhaustifs. Il est de votre responsabilité de fournir un système robuste en créant des tests qui vérifient les fonctionnalités spécifiques à votre classe ou votre programme.

8. **D'abord écrire le code de test (avant d'écrire la classe) afin de vérifier que la conception de la classe est complète.** Si on ne peut écrire le code de test, c'est qu'on ne sait pas ce à quoi la classe ressemble. De plus, l'écriture du code de test montrera souvent de nouvelles fonctionnalités ou contraintes requises dans la classe - ces fonctionnalités ou contraintes n'apparaissent pas toujours dans la phase d'analyse et de conception. Les tests fournissent aussi des exemples de code qui montrent comment utiliser la classe.

9. **Tous les problèmes de conception logicielle peuvent être simplifiés en introduisant un niveau supplémentaire dans l'abstraction.** Cette règle fondamentale de l'ingénierie logicielle [85] constitue la base de l'abstraction, la fonctionnalité première de la programmation orientée objet.

10. **Toute abstraction doit avoir une signification** (à mettre en parallèle avec la règle no 9)**.** Cette signification peut être aussi simple que « mettre du code fréquemment utilisé dans une méthode ». Si on ajoute des abstractions (abstraction, encapsulation, etc.) qui n'ont pas de sens, cela est aussi futile que de ne pas en rajouter.

11. **Rendre les classes aussi atomiques que possible.** Donner à chaque classe un but simple et précis. Si les classes ou la conception du système gagnent en complexité, diviser les classes complexes en classes plus simples. L'indicateur le plus évident est bien sûr la taille : si une classe est grosse, il y a des chances qu'elle en fasse trop et devrait être découpée.

Les indices suggérant la reconception d'une classe sont :

1. Une instruction switch compliquée : utiliser le polymorphisme ;
2. Un grand nombre de méthodes couvrant un large éventail d'opérations différentes : utiliser plusieurs classes ;
3. Un grand nombre de variables membres couvrant des caractéristiques fondamentalement différentes : utiliser plusieurs classes.

12. **Eviter les longues listes d'arguments.** Les appels de méthode deviennent alors difficiles à écrire, lire et maintenir. A la place, essayer de déplacer la méthode dans une classe plus appropriée, et / ou passer des objets en tant qu'arguments.

13. **Ne pas se répéter.** Si une portion de code est récurrente dans plusieurs méthodes dans des classes dérivées, mettre ce code dans une méthode dans la classe de base et l'appeler depuis les méthodes des classes dérivées. Non seulement on gagne en taille de code, mais de plus les changements seront immédiatement effectifs. De plus, la découverte de ces parties de code commun peut ajouter des fonctionnalités intéressantes à l'interface de la classe.

14. **Eviter les instructions** *switch* **ou les** *if-else* **enchaînés.** Ceci est typiquement un indicateur de code *vérificateur de type*, ce qui implique qu'on choisit le code à exécuter suivant une information concernant le type (le type exact peut ne pas être évident à première vue). On peut généralement remplacer ce genre de code avec l'héritage et le polymorphisme ; un appel à une méthode polymorphique se charge de la vérification de type pour vous, et permet une extensibilité plus sûre et plus facile.

15. **Du point de vue de la conception, rechercher et séparer les choses qui changent des choses qui restent les mêmes.** C'est à dire, trouver les éléments du système qu'on pourrait vouloir changer sans forcer une reconception, puis encapsuler ces éléments dans des classes. Ce concept est largement développé dans *Thinking in Patterns with Java*, téléchargeable à *www.BruceEckel.com*.

16. **Ne pas étendre les fonctionnalités fondamentales dans les sous-classes.** Si un élément de l'interface est essentiel dans une classe il doit se trouver dans la classe de base, et non pas rajouté par dérivation. Si on ajoute des méthodes par héritage, la conception est peut-être à revoir.

17. **Moins c'est mieux.** Partir d'une interface minimale pour une classe, aussi restreinte et simple que possible pour résoudre le problème, mais ne pas essayer d'anticiper toutes les façons dont la classe *pourrait* être utilisée. Au fur et à mesure que la classe sera utilisée, on découvrira de nouvelles fonctionnalités à inclure dans l'interface. Cependant, une fois qu'une classe est utilisée, son interface ne peut être réduite sans perturber le code des classes clientes. S'il y a besoin d'ajouter de nouvelles méthodes, cela ne pose pas de problèmes : on ne force qu'une recompilation du code. Mais même si de nouvelles méthodes remplacent les fonctionnalités d'anciennes méthodes, il ne faut pas toucher à l'interface (on peut toujours combiner les fonctionnalités dans l'implémentation sous-jacente si on veut). Si on veut étendre l'interface d'une méthode existante en ajoutant des arguments, créer une méthode surchargée avec les nouveaux arguments : de cette manière les appels à la méthode existante n'en seront pas affectés.

18. **Relire la hiérarchie de classes à voix haute pour s'assurer de sa logique.** Les relations se lisent « est-une » entre classe de base et classe dérivées, et « a-un » entre classe et objet membre.

19. **Se demander si on a besoin de surtyper jusqu'au type de base avant de choisir entre héritage et composition.** Préférer la composition (objets membres) à l'héritage si on n'a pas besoin du transtypage ascendant. Cela permet d'éliminer le besoin d'avoir de nombreux types de base. Si l'héritage est utilisé, les utilisateurs croiront qu'ils sont supposés surtyper.

20. **Utiliser des données membres pour le stockage des valeurs, et des redéfinitions de fonctions pour des modifications de comportement.** Autrement dit, une classe qui utilise des variables d'état en conjonction avec des méthodes qui modifient leur comportement suivant la valeur de ces variables devrait être repensée afin d'exprimer les différences de comportement au sein de classes dérivées et de méthodes redéfinies.

21. **Utiliser la surcharge.** Une méthode ne doit pas se baser sur la valeur d'un argument pour choisir quelle portion de code exécuter. Si le cas se présente, il faut créer deux (voire plus) méthodes surchargées.

22. **Utiliser les hiérarchies d'exceptions** - préférablement dérivées des classes spécifiques appropriées de la hiérarchie standard d'exceptions de Java. La personne capturant les exceptions peut alors capturer les types spécifiques d'exceptions, suivies du type de base. Si de nouvelles exceptions dérivées sont ajoutées, le code client continuera de capturer les exceptions à travers le type de base.

23. **Un simple agrégat peut suffire pour accomplir le travail.** Dans un avion, le confort des passagers est assuré par des éléments totalement indépendants : siège, air conditionné, vidéo, etc. et pourtant ils sont nombreux dans un avion. Faut-il créer des membres privés et construire une nouvelle interface ? Non - dans ce cas, les éléments font partie de l'interface publique ; il faut donc créer des objets membres **public**s. Ces objets ont leur propre implémentation privée, qui reste sûre. Un agrégat n'est pas une solution fréquemment rencontrée, mais cela arrive.

24. **Se placer du point de vue du programmeur client et de la personne maintenant le code.** Concevoir les classes afin qu'elles soient aussi évidentes que possibles à utiliser. Anticiper le type de changements qui seront effectués, et concevoir les classes afin de faciliter l'introduction de ces changements.

25. **Surveiller qu'on n'est pas victime du « syndrome de l'objet géant ».** Ce syndrome afflige souvent les programmeurs procéduraux nouvellement convertis à la POO qui se retrouvent à écrire des programmes procéduraux en les encapsulant dans un ou deux énormes objets. A l'exception des environnements de développement, les objets représentent des concepts dans l'application et non l'application elle-même.

26. **Si on doit incorporer une horreur au système, au moins localiser cette horreur à l'intérieur d'une classe.**

27. **Si on doit incorporer une partie non portable, créer une abstraction pour ce service et le localiser à l'intérieur d'une classe.** Ce niveau supplémentaire d'abstraction empêche la non-portabilité de s'étendre au reste du programme (cet idiome est repris par l'image du *Pont*).

28. **Les objets ne doivent pas seulement contenir des données.** Ils doivent aussi avoir des comportements bien définis (occasionnellement, des « objets de données » peuvent être appropriés, mais seulement s'ils sont utilisés pour empaqueter et transporter un groupe d'articles là où un conteneur plus général ne répondrait pas aux besoins).

29. **Privilégier la composition lorsqu'on crée de nouvelles classes à partir de classes existantes.** L'héritage ne doit être utilisé que s'il est requis par la conception. Si l'héritage est utilisé là où la composition aurait suffi, la modélisation deviendra inutilement compliquée.

30. **Utiliser l'héritage et la redéfinition de méthodes pour exprimer les différences de comportement, et des variables pour exprimer des variations dans l'état.** Un exemple extrême de ce qu'il ne faut pas faire est de dériver différentes classes pour représenter des couleurs plutôt qu'utiliser un champ « couleur ».

31. **Se méfier de la *variance*.** Deux objets sémantiquement différents peuvent avoir des actions ou des responsabilités identiques, et il est tentant de faire de l'une une sous-classe de l'autre pour bénéficier de l'héritage. Ceci est appelé la *variance*, mais il n'y a aucune raison

d'instaurer une relation superclasse / classe dérivée là où il n'y en a pas. Une meilleure solution est de créer une classe de base générique qui produise une interface pour les deux classes dérivées - cela nécessite un peu plus de place, mais on bénéficie toujours de l'héritage, et on va probablement faire une découverte importante concernant la modélisation.

32. **Eviter les *limitations* introduites durant un héritage.** Les conceptions les plus claires ajoutent de nouvelles capacités à des classes dérivées. Les conceptions suspectes enlèvent des capacités durant l'héritage sans en ajouter de nouvelles. Mais les règles sont faites pour être transgressées, et si on travaille avec une ancienne bibliothèque de classes, il peut être plus efficace de restreindre une classe existante dans la classe dérivée que de restructurer la hiérarchie afin que la nouvelle classe s'intègre là où elle devrait, c'est à dire au dessus de la vieille classe.

33. **Utiliser les patrons de conception (design patterns) pour éliminer les « fonctionnalités non cachées ».** C'est à dire que si un seul objet de la classe doit être créé, ne pas livrer la classe telle quelle à l'application avec un commentaire du genre : « Ne créer qu'un seul de ces objets. » ; il faut l'encapsuler dans un singleton. Si le programme principal comporte trop de code embrouillé destiné à créer les objets de l'application, rechercher un modèle de création, par exemple une méthode propriétaire dans laquelle on pourrait encapsuler la création de ces objets. Eliminer les « fonctionnalités non cachées » rendra le code non seulement plus facile à comprendre et maintenir, mais il le blindera aussi contre les mainteneurs bien intentionnés qui viendront après.

34. **Eviter la « paralysie analytique ».** Se rappeler qu'il faut avancer dans un projet avant de tout savoir, et que parfois le meilleur moyen d'en apprendre sur certains facteurs inconnus est de passer à l'étape suivante plutôt que d'essayer de les imaginer. On ne peut connaître la solution tant qu'on ne l'a pas. Java possède des pare-feux intégrés ; laissez-les travailler pour vous. Les erreurs introduites dans une classe ou un ensemble de classes ne peuvent détruire l'intégrité du système dans son ensemble.

35. **Faire une relecture lorsqu'on pense avoir une bonne analyse, conception ou implémentation.** Demander à une personne extérieure au groupe - pas obligatoirement un consultant, cela peut très bien être quelqu'un d'un autre groupe de l'entreprise. Faire examiner le travail accompli par un oeil neuf peut révéler des problèmes durant une phase où il est plus facile de les corriger, et largement compenser le temps et l'argent « perdus » pendant le processus de relecture.

Implémentation

1. **D'une manière générale, suivre les conventions de codage de Sun.** Celles-ci sont disponibles à *java.sun.com/docs/codeconv/index.html* (le code fourni dans ce livre respecte ces conventions du mieux que j'ai pu). Elles sont utilisées dans la majeure partie du code à laquelle la majorité des programmeurs Java seront exposés. Si vous décidez de vous en tenir obstinément à vos propres conventions de codage, vous rendrez la tâche plus ardue au lecteur. Quelles que soient les conventions de codage retenues, s'assurer qu'elles sont respectées dans tout le projet. Il existe un outil gratuit de reformatage de code Java disponible à *home.wtal.de/software-solutions/jindent/*.

2. **Quelles que soient les conventions de codage utilisées, cela change tout de les standardiser au sein de l'équipe (ou même mieux, au niveau de l'entreprise).** Cela devrait même aller jusqu'au point où tout le monde devrait accepter de voir son style de codage

modifié s'il ne se conforme pas aux règles de codage en vigueur. La standardisation permet de passer moins de temps à analyser la forme du code afin de se concentrer sur son sens.

3. **Suivre les règles standard de capitalisation.** Capitaliser la première lettre des noms de classe. La première lettre des variables, méthodes et des objets (références) doit être une minuscule. Les identifiants doivent être formés de mots collés ensemble, et la première lettre des mots intermédiaires doit être capitalisée. Ainsi : **CeciEstUnNomDeClasse**, **ceciEstUnNomDeMethodeOuDeVariable**. Capitaliser *toutes* les lettres des identifiants déclarés comme **static final** et initialisés par une valeur constante lors de leur déclaration. Cela indique que ce sont des constantes dès la phase de compilation. **Les packages constituent un cas à part** - ils doivent être écrits en minuscules, même pour les mots intermédiaires. Les extensions de domaine (com, org, net, edu, etc.) doivent aussi être écrits en minuscules (ceci a changé entre Java 1.1 et Java 2).

4. **Ne pas créer des noms « décorés » de données membres privées.** On voit souvent utiliser des préfixes constitués d'underscores et de caractères. La notation hongroise en est le pire exemple, où on préfixe le nom de variable par son type, son utilisation, sa localisation, etc., comme si on écrivait en assembleur ; et le compilateur ne fournit aucune assistance supplémentaire pour cela. Ces notations sont confuses, difficiles à lire, à mettre en oeuvre et à maintenir. Laisser les classes et les packages s'occuper de la portée des noms.

5. **Suivre une « forme canonique »** quand on crée une classe pour un usage générique. Inclure les définitions pour **equals()**, **hashCode()**, **toString()**, **clone()** (implémentation de **Cloneable**), et implémenter **Comparable** et **Serializable**.

6. **Utiliser les conventions de nommage « get », « set » et « is » de JavaBeans** pour les méthodes qui lisent et changent les variables **private**, même si on ne pense pas réaliser un JavaBean au moment présent. Non seulement cela facilitera l'utilisation de la classe comme un Bean, mais ce sont des noms standards pour ce genre de méthode qui seront donc plus facilement comprises par le lecteur.

7. **Pour chaque classe créée, inclure une méthode** *static public test()* **qui contienne du code testant cette classe.** On n'a pas besoin d'enlever le code de test pour utiliser la classe dans un projet, et on peut facilement relancer les tests après chaque changement effectué dans la classe. Ce code fournit aussi des exemples sur l'utilisation de la classe.

8. **Il arrive qu'on ait besoin de dériver une classe afin d'accéder à ses données** *protected***.**Ceci peut conduire à percevoir un besoin pour de nombreux types de base. Si on n'a pas besoin de surtyper, il suffit de dériver une nouvelle classe pour accéder aux accès protégés, puis de faire de cette classe un objet membre à l'intérieur des classes qui en ont besoin, plutôt que dériver à nouveau la classe de base.

9. **Eviter l'utilisation de méthodes** *final* **juste pour des questions d'efficacité.**Utiliser **final** seulement si le programme marche, mais pas assez rapidement, et qu'un profilage a montré que l'invocation d'une méthode est le goulot d'étranglement.

10. **Si deux classes sont associées d'une certaine manière (telles que les conteneurs et les itérateurs), essayer de faire de l'une une classe interne à l'autre.** Non seulement cela fait ressortir l'association entre les classes, mais cela permet de réutiliser le nom de la classe à l'intérieur du même package en l'incorporant dans une autre classe. La bibliothèque des conteneurs Java réalise cela en définissant une classe interne **Iterator** à l'intérieur de chaque classe conteneur, fournissant ainsi une interface commune aux conteneurs. Utiliser une classe interne permet aussi une implémentation **private**. Le bénéfice de la classe interne est

donc de cacher l'implémentation en plus de renforcer l'association de classes et de prévenir de la pollution de l'espace de noms.

11. **Quand on remarque que certaines classes sont liées entre elles, réfléchir aux gains de codage et de maintenance réalisés si on en faisait des classes internes l'une à l'autre.** L'utilisation de classes internes ne va pas casser l'association entre les classes, mais au contraire rendre cette liaison plus explicite et plus pratique.

12. **C'est pure folie que de vouloir optimiser trop prématurément.** En particulier, ne pas s'embêter à écrire (ou éviter) des méthodes natives, rendre des méthodes **final**, ou stresser du code pour le rendre efficace dans les premières phases de construction du système. Le but premier est de valider la conception, sauf si la conception spécifie une certaine efficacité.

13. **Restreindre autant que faire se peut les portées afin que la visibilité et la durée de vie des objets soient la plus faible possible.** Cela réduit les chances d'utiliser un objet dans un mauvais contexte et la possibilité d'ignorer un bug difficile à détecter. Par exemple, supposons qu'on dispose d'un conteneur et d'une portion de code qui itère en son sein. Si on copie ce code pour l'utiliser avec un nouveau conteneur, il se peut qu'on en arrive à utiliser la taille de l'ancien conteneur comme borne supérieure du nouveau. Cependant, si l'ancien conteneur est hors de portée, l'erreur sera reportée lors de la compilation.

14. **Utiliser les conteneurs de la bibliothèque Java standard.** Devenir compétent dans leur utilisation garantit un gain spectaculaire dans la productivité. Préférer les **ArrayList** pour les séquences, les **HashSet** pour les sets, les **HashMap** pour les tableaux associatifs, et les **LinkedList** pour les piles (plutôt que les **Stack**) et les queues.

15. **Pour qu'un programme soit robuste, chaque composant doit l'être.** Utiliser tout l'arsenal d'outils fournis par Java : contrôle d'accès, exceptions, vérification de types, etc. dans chaque classe créée. Ainsi on peut passer sans crainte au niveau d'abstraction suivant lorsqu'on construit le système.

16. **Préférer les erreurs de compilation aux erreurs d'exécution.** Essayer de gérer une erreur aussi près de son point d'origine que possible. Mieux vaut traiter une erreur quand elle arrive que générer une exception. Capturer les exceptions dans le gestionnaire d'exceptions le plus proche qui possède assez d'informations pour les traiter. Faire ce qu'on peut avec l'exception au niveau courant ; si cela ne suffit pas, relancer l'exception.

17. **Eviter les longues définitions de méthodes.** Les méthodes doivent être des unités brèves et fonctionnelles qui décrivent et implémentent une petite part de l'interface d'une classe. Une méthode longue et compliquée est difficile et chère à maintenir, et essaye probablement d'en faire trop par elle-même. Une telle méthode doit, au minimum, être découpée en plusieurs méthodes. Cela peut aussi être le signe qu'il faudrait créer une nouvelle classe. De plus, les petites méthodes encouragent leur réutilisation à l'intérieur de la classe (quelquefois les méthodes sont grosses, mais elles doivent quand même ne réaliser qu'une seule opération).

18. **Rester « aussi private que possible ».** Une fois rendu public un aspect de la bibliothèque (une méthode, une classe, une variable), on ne peut plus l'enlever. Si on le fait, on prend le risque de ruiner le code existant de quelqu'un, le forçant à le réécrire ou même à revoir sa conception. Si on ne publie que ce qu'on doit, on peut changer tout le reste en toute impunité ; et comme la modélisation est sujette à changements, ceci est une facilité de développement à ne pas négliger. De cette façon, les changements dans l'implémentation auront

un impact minimal sur les classes dérivées. La privatisation est spécialement importante lorsqu'on traite du multithreading - seuls les champs **private** peuvent être protégés contre un accès non **synchronized**.

19. **Utiliser les commentaires sans restrictions, et utiliser la syntaxe de documentation de** *javadoc* **pour produire la documentation du programme.** Cependant, les commentaires doivent ajouter du sens au code ; les commentaires qui ne font que reprendre ce que le code exprime clairement sont ennuyeux. Notez que le niveau de détails typique des noms des classes et des méthodes de Java réduit le besoin de commentaires.

20. **Eviter l'utilisation des « nombres magiques »** - qui sont des nombres codés en dur dans le code. C'est un cauchemar si on a besoin de les changer, on ne sait jamais si « 100 » représente « la taille du tableau » ou « quelque chose dans son intégralité ». A la place, créer une constante avec un nom explicite et utiliser cette constante dans le programme. Cela rend le programme plus facile à comprendre et bien plus facile à maintenir.

21. **Quand on crée des constructeurs, réfléchir aux exceptions.** Dans le meilleur des cas, le constructeur ne fera rien qui générera une exception. Dans le meilleur des cas suivant, la classe sera uniquement composée et dérivée de classes robustes, et aucun nettoyage ne sera nécessaire si une exception est générée. Sinon, il faut nettoyer les classes composées à l'intérieur d'une clause **finally**. Si un constructeur échoue dans la création, l'action appropriée est de générer un exception afin que l'appelant ne continue pas aveuglément en pensant que l'objet a été créé correctement.

22. **Si la classe nécessite un nettoyage lorsque le programmeur client en a fini avec l'objet, placer la portion de code de nettoyage dans une seule méthode bien définie** - avec un nom explicite tel que **cleanup()** qui suggère clairement sa fonction. De plus, utiliser un flag **boolean** dans la classe pour indiquer si l'objet a été nettoyé afin que **finalize()** puisse vérifier la « condition de destruction » (cf Chapitre 4).

23. **La seule responsabilité qui incombe à** *finalize()* **est de vérifier la « condition de destruction » d'un objet pour le débuggage**(cf Chapitre 4). Certains cas spéciaux nécessitent de libérer de la mémoire qui autrement ne serait pas restituée par le ramasse-miettes. Comme il existe une possibilité pour que le ramasse-miettes ne soit pas appelé pour un objet, on ne peut utiliser **finalize()** pour effectuer le nettoyage nécessaire. Pour cela il faut créer sa propre méthode de nettoyage. Dans la méthode **finalize()** de la classe, vérifier que l'objet a été nettoyé, et génèrer une exception dérivée de **RuntimeException** si cela n'est pas le cas, afin d'indiquer une erreur de programmation. Avant de s'appuyer sur un tel dispositif, s'assurer que **finalize()** fonctionne sur le système considéré (un appel à **System.gc()** peut être nécessaire pour s'assurer de ce fonctionnement).

24. **Si un objet doit être nettoyé (autrement que par le ramasse-miettes) à l'intérieur d'une portée particulière, utiliser l'approche suivante :** initialiser l'objet et, en cas de succès, entrer immédiatement dans un block **try** avec une clause **finally** qui s'occupe du nettoyage.

25. **Lors de la redéfinition de** *finalize()* **dans un héritage, ne pas oublier d'appeler** *super.finalize()*(ceci n'est pas nécessaire si **Object** est la classe parente immédiate). Un appel à **super.finalize()** doit être la *dernière* instruction de la méthode **finalize()** redéfinie plutôt que la première, afin de s'assurer que les composants de la classe de base soient toujours valides si on en a besoin.

26. **Lors de la création d'un conteneur d'objets de taille fixe, les transférer dans un**

tableau - surtout si on retourne le conteneur depuis une méthode. De cette manière on bénéficie de la vérification de types du tableau lors de la compilation, et le récipiendiaire du tableau n'a pas besoin de transtyper les objets du tableau pour les utiliser. Notez que la classe de base de la bibliothèque de conteneurs, **java.util.Collection**, possède deux méthodes **toArray()** pour accomplir ceci.

27. **Préférer les** *interface***s aux classes** *abstract***.**Si on sait que quelque chose va être une classe de base, il faut en faire une **interface**, et ne la changer en classe **abstract** que si on est obligé d'y inclure des définitions de méthodes et des variables membres. Une **interface** parle de ce que le client veut faire, tandis qu'une classe a tendance à se focaliser sur (ou autorise) les détails de l'implémentation.

28. **A l'intérieur des constructeurs, ne faire que ce qui est nécessaire pour mettre l'objet dans un état stable.** Eviter autant que faire se peut l'appel à d'autres méthodes (les méthodes **final** exceptées), car ces méthodes peuvent être redéfinies par quelqu'un d'autre et produire des résultats inattendus durant la construction (se référer au chapitre 7 pour plus de détails). Des constructeurs plus petits et plus simples ont moins de chances de générer des exceptions ou de causer des problèmes.

29. **Afin d'éviter une expérience hautement frustrante, s'assurer qu'il n'existe qu'une classe non packagée de chaque nom dans tout le classpath.** Autrement, le compilateur peut trouver l'autre classe de même nom d'abord, et renvoyer des messages d'erreur qui n'ont aucun sens. Si un problème de classpath est suspecté, rechercher les fichiers **.class** avec le même nom à partir de chacun des points de départ spécifiés dans le classpath, l'idéal étant de mettre toutes les classes dans des packages.

30. **Surveiller les surcharges accidentelles.** Si on essaye de redéfinir une méthode de la classe de base et qu'on se trompe dans l'orthographe de la méthode, on se retrouve avec une nouvelle méthode au lieu d'une méthode existante redéfinie. Cependant, ceci est parfaitement légal, et donc ni le compilateur ni le système d'exécution ne signaleront d'erreur - le code ne fonctionnera pas correctement, c'est tout.

31. **Surveiller les surcharges accidentelles.** Si on essaye de redéfinir une méthode de la classe de base et qu'on se trompe dans l'orthographe de la méthode, on se retrouve avec une nouvelle méthode au lieu d'une méthode existante redéfinie. Cependant, ceci est parfaitement légal, et donc ni le compilateur ni le système d'exécution ne signaleront d'erreur - le code ne fonctionnera pas correctement, c'est tout.

32. **Surveiller les surcharges accidentelles.** Si on essaye de redéfinir une méthode de la classe de base et qu'on se trompe dans l'orthographe de la méthode, on se retrouve avec une nouvelle méthode au lieu d'une méthode existante redéfinie. Cependant, ceci est parfaitement légal, et donc ni le compilateur ni le système d'exécution ne signaleront d'erreur - le code ne fonctionnera pas correctement, c'est tout.

[85]Qui m'a été expliquée par Andrew Koenig.

Annexe D - Resources

Logicielles

Le JDK de *java.sun.com*. Même si vous choisissez d'utiliser un autre environnement de dé-

veloppement, il est toujours bon d'avoir le JDK sous la main au cas où vous vous heurteriez à ce qui pourrait être une erreur du compilateur. Le JDK est la référence ; et s'il y a un bug, il a de fortes chances d'être rapidement repéré.

La documentation HTML de Java de *java.sun.com*. Je n'ai jamais trouvé de livre de référence sur la bibliothèque standard Java qui soit à jour ou complet. Bien que la documentation HTML de Sun soit parsemée de petits bugs et quelquefois un peu laconique, au moins on y trouve toutes les classes et méthodes. Souvent on se sent moins enclin à utiliser une ressource online plutôt qu'un livre imprimé, mais cela vaut généralement le coup de surmonter cette appréhension et de consulter d'abord la documentation HTML, afin de se faire au moins une idée générale. Si cela ne suffit pas, alors on peut toujours se tourner vers une ressource plus traditionnelle.

Livres

Thinking in Java, 1st Edition. **Disponible sur le CD ROM fourni avec ce livre ou téléchargeable gratuitement depuis** *www.BruceEckel.com* **sous forme HTML pleinement indexée avec colorisation de syntaxe. Contient des informations moins récentes ou jugées pas assez pertinentes pour demeurer dans la deuxième édition.**

Core Java 2, de Horstmann & Cornell, Volume I-Fundamentals (Prentice-Hall, 1999). Volume II-Advanced Features, 2000. Enorme, détaillé, et le premier endroit où je vais lorsque je cherche une réponse. Le livre que je recommande une fois terminé *Thinking in Java* et que vos besoins sont plus conséquents.

Java in a Nutshell: A Desktop Quick Reference, 2nd Edition, de David Flanagan (O'Reilly, 1997). Un résumé condensé de la documentation Java online. Personnellement, je préfère parcourir la documentation online de *java.sun.com*, en particulier parce qu'elle est si souvent mise à jour. Cependant, nombreux sont ceux qui préfèrent une documentation imprimée et celle-ci convient parfaitement ; de plus elle contient plus de commentaires que les documents online.

The Java Class Libraries: An Annotated Reference, de Patrick Chan et Rosanna Lee (Addison-Wesley, 1997). Ce que la référence online *aurait dû* être : posséder assez de descriptions pour être utilisable. Un des relecteurs techniques de *Thinking in Java* a dit : « Si je ne devais avoir qu'un livre sur Java, ce serait celui-ci (en plus du tien, bien sûr) ». Je ne suis pas aussi enthousiaste que lui. Il est gros, il est cher, et la qualité de ses exemples ne me satisfait pas. *Mais* c'est un endroit où regarder quand on est coincé et il semble avoir plus de profondeur (et une taille plus conséquente) que *Java in a Nutshell*.

Java Network Programming, de Elliotte Rusty Harold (O'Reilly, 1997). Je n'ai commencé à comprendre la programmation réseau en Java qu'après avoir lu ce livre. J'ai aussi trouvé que son site web, Café au Lait, offrait un point de vue stimulant, pertinent, à jour et non influencé sur les développements de Java. Ses mises à jour fréquentes suivent l'actualité de Java. Se reporter à *metalab.unc.edu/javafaq/*.

JDBC Database Access with Java, de Hamilton, Cattell & Fisher (Addison-Wesley, 1997). Si vous n'y connaissez rien en SQL et aux bases de données, ce livre vous en fournira les premiers éléments. Il contient aussi une « référence annotée » ainsi que des détails de l'API (encore une fois, de ce que l'API était lors de la sortie du livre). L'inconvénient de ce livre, comme tous ceux de *The Java Series* (« Les SEULS Livres Approuvés par JavaSoft »), est qu'ils sont édulcorés en ce sens qu'ils ne disent que du bien de Java - vous ne trouverez aucune critique dans cette collection.

Java Programming with CORBA, de Andreas Vogel & Keith Duddy (John Wiley & Sons,

1997). Un sujet sérieusement traité avec des exemples de code pour trois ORBs Java (Visibroker, Orbix, Joe).

Design Patterns, de Gamma, Helm, Johnson & Vlissides (Addison-Wesley, 1995). Le livre qui a lancé le concept de patrons dans la programmation.

Practical Algorithms for Programmers, de Binstock & Rex (Addison-Wesley, 1995). Les algorithmes sont en C, ils sont donc facilement transposables en Java. Chaque exemple est minutieusement expliqué.

Analyse & conception

Extreme Programming Explained, de Kent Beck (Addison-Wesley, 2000). J'*adore* ce livre. Bien que j'aie tendance à utiliser une approche radicale, j'ai toujours cru qu'il devait exister une manière différente et plus efficace de mener à bien un projet, et je crois que XP s'en approche drôlement. Le seul livre qui m'ait fait autant d'effet est *PeopleWare* (décrit ci-après), qui parle principalement de l'environnement et traite de la culture d'entreprise. *Extreme Programming Explained* parle de programmation, et remet les choses, y compris les dernières « trouvailles », à leur place. Il va même jusqu'à dire que les images sont OK tant qu'on ne passe pas trop de temps avec elles et qu'on les jette à terme (vous remarquerez que ce livre ne porte pas la mention « approuvé par UML » sur sa couverture). J'ai vu des gens choisir une entreprise en se basant seulement sur le fait de savoir si celle-ci utilisait XP. Petit livre, petits chapitres, agréable à lire, profond quant à la reflexion qu'il provoque. Vous vous imaginez dans une telle atmosphère de travail et cela vous amène des visions d'un monde entièrement nouveau.

UML Distilled, 2nd Edition, de Martin Fowler (Addison-Wesley, 2000). La première rencontre avec UML est souvent rebutante à cause de tous les diagrammes et les détails. Selon Fowler, la plupart de ces détails ne sont pas nécessaires, et il tranche dedans pour aller à l'essentiel. Pour la plupart des projets, vous n'avez besoin de connaître que quelques diagrammes outils, et le but de Fowler est d'arriver à une bonne conception plutôt que de s'ennuyer avec tous les détails pour y arriver. Un petit livre agréable et intéressant ; le premier à lire si vous avez besoin de vous plonger dans UML.

UML Toolkit, de Hans-Erik Eriksson & Magnus Penker, (John Wiley & Sons, 1997). Explique UML et comment l'utiliser, avec un exemple appliqué en Java. Un CD ROM contient le code Java du livre et une version allégée de Rational Rose. Une excellente introduction sur UML et comment l'utiliser pour construire un système réel.

The Unified Software Development Process, de Ivar Jacobsen, Grady Booch, et James Rumbaugh (Addison-Wesley, 1999). Je m'étais attendu à détester ce livre. Il semblait avoir tous les défauts d'un texte universitaire ennuyeux. Je fus agréablement surpris - seules quelques parties du livre contiennent des explications qui font penser que les concepts ne sont pas clairs pour les auteurs. La majeure partie du livre n'est pas seulement claire, mais agréable à lire. Et le mieux, c'est que la méthodologie présentée n'est pas dénuée de sens pratique. Ce n'est pas la Programmation Extreme (en particulier elle ne présente pas la même lucidité quant aux tests), mais elle fait partie du mastodonte UML - même si vous n'arrivez pas à faire adopter XP, la plupart des gens ont adhéré au « mouvement UML » (sans tenir compte de leur niveau *réel* d'expérience) et vous devriez pouvoir faire adopter cette méthode. Je pense que ce livre est le vaisseau amiral de UML, et celui que vous devriez lire après *UML Distilled* de Fowler quand vous aurez besoin de plus de détails.

Avant d'opter pour une méthode, il est bon de se renseigner auprès de personnes qui n'essayent pas d'en vendre une. Il est facile d'adopter une méthode sans réellement comprendre ce qu'on

en attend ou ce qu'elle va faire pour nous. « D'autres l'utilisent, ce qui fait une bonne raison de l'adopter ». Cependant, les hommes peuvent montrer une bizarrerie psychologique : s'ils pensent que quelque chose va solutionner leurs problèmes, ils l'essayeront (c'est l'expérimentation, ce qui est une bonne chose). Mais si cela ne résoud pas leurs problèmes, ils peuvent redoubler d'efforts et annoncer à voix haute quelle découverte merveilleuse ils ont fait (c'est un mensonge, ce qui est une vilaine chose). Le raisonnement fait ici est que si vous arrivez à attirer d'autres personnes dans le même bateau, vous ne vous retrouverez pas tout seul, même s'il ne va nulle part (ou coule).

Cela ne veut pas dire que toutes les méthodologies se terminent en cul-de-sac, mais que vous devez vous préparer pour vous maintenir dans le mode expérimentation (« Cela ne marche pas, essayons quelque chose d'autre »), sans rentrer dans le mode mensonge (« Non, ce n'est pas réellement un problème. Tout marche, on n'a pas besoin de changer »). Je pense que les livres suivants, à lire *avant* de choisir une méthodologie, vous aident dans cette préparation.

Software Creativity, de Robert Glass (Prentice-Hall, 1995). Ceci est le meilleur livre que j'aie vu qui discute des perspectives du problème de la méthodologie. C'est un recueil de courts essais et articles que Glass a écrit et quelquefois récupéré (P. J. Plauger est l'un des contributeurs), reflétant ses années d'étude et de réflexion sur le sujet. Ils sont distrayants et juste assez longs pour expliquer ce qu'ils veulent faire passer ; il ne vous égare pas ni ne vous ennuie. Glass ne vous mène pas en bateau non plus, il y a des centaines de références à d'autres articles et études. Tous les programmeurs et les directeurs devraient lire ce livre avant de s'aventurer dans la jungle des méthodologies.

Software Runaways: Monumental Software Disasters, de Robert Glass (Prentice-Hall, 1997). Le point fort de ce livre est qu'il met en lumière ce dont personne ne parle : combien de projets non seulement échouent, mais échouent spectaculairement. La plupart d'entre nous pensent encore : « Cela ne peut pas m'arriver » (ou « Cela ne peut pas *encore* m'arriver »), et je pense que cela nous met en position de faiblesse. En se rappelant que les choses peuvent toujours mal se passer, on se retrouve en bien meilleure position pour les contrôler.

Peopleware, **2nd Edition**, de Tom Demarco et Timothy Lister (Dorset House, 1999). Bien que les auteurs proviennent du monde du développement logiciel, ce livre traite des projets et des équipes en général. Mais il se concentre sur les *gens* et leurs besoins, plutôt que sur la technologie et ses besoins. Les auteurs parlent de la création d'un environnement où les gens seront heureux et productifs, plutôt que des décisions et des règles que ces gens devraient suivre pour être les composants bien huilés d'une machine. C'est cette dernière attitude qui, selon moi, fait sourire et acquiescer les programmeurs quand la méthode XYZ est adoptée et qu'ils continuent à travailler de la même manière qu'ils l'ont toujours fait.

Complexity, de M. Mitchell Waldrop (Simon & Schuster, 1992). Ce livre rapporte la mise en relations d'un groupe de scientifiques de différentes spécialités à Santa Fe, New Mexico, pour discuter de problèmes que leurs disciplines ne pouvaient résoudre individuellement (le marché de la bourse en économie, la création originelle de la vie en biologie, pourquoi les gens agissent ainsi qu'ils le font en sociologie, etc...). En combinant la physique, l'économie, la chimie, les mathématiques, l'informatique, la sociologie et d'autres matières, une approche multidisciplinaire envers ces problèmes se développe. Mais plus important, une nouvelle façon de *penser* ces problèmes ultra-complexes émerge : loin du déterminisme mathématique et de l'illusion qu'on peut écrire une équation qui prédise tous les comportements, mais basée sur l'*observation* à la recherche d'un motif et la tentative de recréer ce motif par tous les moyens possibles. (Le livre rapporte, par exemple, l'émergence des algorithmes génétiques.) Je pense que cette manière de penser est utile lorsqu'on observe les façons de gérer des projets logiciels de plus en plus complexes.

Python

Learning Python, de Mark Lutz and David Ascher (O'Reilly, 1999). Une bonne introduction à ce qui est rapidement devenu mon langage favori, un excellent compagnon à Java. Le livre inclut une introduction à JPython, qui permet de combiner Java et Python dans un unique programme (l'interpréteur JPython est compilé en pur bytecode Java, il n'y a donc rien de spécial à ajouter pour réaliser cela). Cette union de langages promet de grandes possibilités.

La liste de mes livres

Listés par ordre de publication. Tous ne sont pas actuellement disponibles.

Computer Interfacing with Pascal & C, (Auto-publié dans la rubrique Eisys, 1988. Disponible seulement sur *www.BruceEckel.com*). Une introduction à l'électronique au temps où le CP/M était encore le roi et le DOS un parvenu. J'utilisais des langages de haut niveau et souvent le port parallèle de l'ordinateur pour piloter divers projets électroniques. Adapté de mes chroniques dans le premier et le meilleur des magazines pour lesquels j'ai écrit, *Micro Cornucopia* (pour paraphraser Larry O'Brien, éditeur au long cours de *Software Development Magazine* : le meilleur magazine informatique jamais publié - ils avaient même des plans pour construire un robot dans un pot de fleurs !). Hélas, Micro C s'est arrêté bien avant que l'Internet n'apparaisse. Créer ce livre fut une expérience de publication extrêmement satisfaisante.

Using C++, (Osborne/McGraw-Hill, 1989). Un des premiers livres sur le C++. Epuisé et remplacé par sa deuxième édition, renommée *C++ Inside & Out*.

C++ Inside & Out, (Osborne/McGraw-Hill, 1993). Comme indiqué, il s'agit en fait de la deuxième édition de **Using C++**. Le C++ dans ce livre est raisonnablement précis, mais il date de 1992 et *Thinking in C++* est destiné à le remplacer. Vous pouvez trouver plus de renseignements sur ce livre et télécharger le code source sur *www.BruceEckel.com*.

Thinking in C++, 1st Edition, (Prentice-Hall, 1995).

Thinking in C++, 2nd Edition, Volume 1, **(Prentice-Hall, 2000). Téléchargeable sur** *www.BruceEckel.com*.

Black Belt C++, the Master's Collection, Bruce Eckel, éditeur (M&T Books, 1994). Epuisé. Un recueil de chapitres par divers experts C++, basés sur leurs présentations dans le cycle C++ lors de la Conférence sur le Développement Logiciel, que je présidais. La couverture de ce livre m'a convaincu d'obtenir le contrôle de toutes les futures couvertures.

Thinking in Java, 1st Edition, (Prentice-Hall, 1998). La première édition de ce livre a gagné la *Software Development Magazine* Productivity Award, le *Java Developer's Journal* Editor's Choice Award, et le *JavaWorld Reader's Choice Award for best book*. Téléchargeable sur *www.BruceEckel.com*.

Annexe D – ressources

Logicielles

Le JDK de *java.sun.com*. Même si vous choisissez d'utiliser un autre environnement de développement, il est toujours bon d'avoir le JDK sous la main au cas où vous vous heurteriez à ce qui pourrait être une erreur du compilateur. Le JDK est la référence ; et s'il y a un bug, il a de fortes chances d'être rapidement repéré.

La documentation HTML de Java de *java.sun.com*. Je n'ai jamais trouvé de livre de référence sur la bibliothèque standard Java qui soit à jour ou complet. Bien que la documentation HTML de Sun soit parsemée de petits bugs et quelquefois un peu laconique, au moins on y trouve toutes les classes et méthodes. Souvent on se sent moins enclin à utiliser une ressource online plutôt qu'un livre imprimé, mais cela vaut généralement le coup de surmonter cette appréhension et de consulter d'abord la documentation HTML, afin de se faire au moins une idée générale. Si cela ne suffit pas, alors on peut toujours se tourner vers une ressource plus traditionnelle.

Livres

*Thinking in Java, 1st Edition***. Disponible sur le CD ROM fourni avec ce livre ou téléchargeable gratuitement depuis** *www.BruceEckel.com* **sous forme HTML pleinement indexée avec colorisation de syntaxe. Contient des informations moins récentes ou jugées pas assez pertinentes pour demeurer dans la deuxième édition.**

Core Java 2, de Horstmann & Cornell, Volume I-Fundamentals (Prentice-Hall, 1999). Volume II-Advanced Features, 2000. Enorme, détaillé, et le premier endroit où je vais lorsque je cherche une réponse. Le livre que je recommande une fois terminé *Thinking in Java* et que vos besoins sont plus conséquents.

Java in a Nutshell: A Desktop Quick Reference, 2nd Edition, de David Flanagan (O'Reilly, 1997). Un résumé condensé de la documentation Java online. Personnellement, je préfère parcourir la documentation online de *java.sun.com*, en particulier parce qu'elle est si souvent mise à jour. Cependant, nombreux sont ceux qui préfèrent une documentation imprimée et celle-ci convient parfaitement ; de plus elle contient plus de commentaires que les documents online.

The Java Class Libraries: An Annotated Reference, de Patrick Chan et Rosanna Lee (Addison-Wesley, 1997). Ce que la référence online *aurait dû* être : posséder assez de descriptions pour être utilisable. Un des relecteurs techniques de *Thinking in Java* a dit : « Si je ne devais avoir qu'un livre sur Java, ce serait celui-ci (en plus du tien, bien sûr) ». Je ne suis pas aussi enthousiaste que lui. Il est gros, il est cher, et la qualité de ses exemples ne me satisfait pas. *Mais* c'est un endroit où regarder quand on est coincé et il semble avoir plus de profondeur (et une taille plus conséquente) que *Java in a Nutshell*.

Java Network Programming, de Elliotte Rusty Harold (O'Reilly, 1997). Je n'ai commencé à comprendre la programmation réseau en Java qu'après avoir lu ce livre. J'ai aussi trouvé que son site web, Café au Lait, offrait un point de vue stimulant, pertinent, à jour et non influencé sur les développements de Java. Ses mises à jour fréquentes suivent l'actualité de Java. Se reporter à *metalab.unc.edu/javafaq/*.

JDBC Database Access with Java, de Hamilton, Cattell & Fisher (Addison-Wesley, 1997). Si vous n'y connaissez rien en SQL et aux bases de données, ce livre vous en fournira les premiers éléments. Il contient aussi une « référence annotée » ainsi que des détails de l'API (encore une fois, de ce que l'API était lors de la sortie du livre). L'inconvénient de ce livre, comme tous ceux de *The Java Series* (« Les SEULS Livres Approuvés par JavaSoft »), est qu'ils sont édulcorés en ce sens qu'ils ne disent que du bien de Java - vous ne trouverez aucune critique dans cette collection.

Java Programming with CORBA, de Andreas Vogel & Keith Duddy (John Wiley & Sons, 1997). Un sujet sérieusement traité avec des exemples de code pour trois ORBs Java (Visibroker, Orbix, Joe).

Design Patterns, de Gamma, Helm, Johnson & Vlissides (Addison-Wesley, 1995). Le livre qui a lancé le concept de patrons dans la programmation.

Practical Algorithms for Programmers, de Binstock & Rex (Addison-Wesley, 1995). Les algorithmes sont en C, ils sont donc facilement transposables en Java. Chaque exemple est minutieusement expliqué.

Analyse & conception

Extreme Programming Explained, de Kent Beck (Addison-Wesley, 2000). J'*adore* ce livre. Bien que j'aie tendance à utiliser une approche radicale, j'ai toujours cru qu'il devait exister une manière différente et plus efficace de mener à bien un projet, et je crois que XP s'en approche drôlement. Le seul livre qui m'ait fait autant d'effet est *PeopleWare* (décrit ci-après), qui parle principalement de l'environnement et traite de la culture d'entreprise. *Extreme Programming Explained* parle de programmation, et remet les choses, y compris les dernières « trouvailles », à leur place. Il va même jusqu'à dire que les images sont OK tant qu'on ne passe pas trop de temps avec elles et qu'on les jette à terme (vous remarquerez que ce livre ne porte pas la mention « approuvé par UML » sur sa couverture). J'ai vu des gens choisir une entreprise en se basant seulement sur le fait de savoir si celle-ci utilisait XP. Petit livre, petits chapitres, agréable à lire, profond quant à la reflexion qu'il provoque. Vous vous imaginez dans une telle atmosphère de travail et cela vous amène des visions d'un monde entièrement nouveau.

UML Distilled, 2nd Edition, de Martin Fowler (Addison-Wesley, 2000). La première rencontre avec UML est souvent rebutante à cause de tous les diagrammes et les détails. Selon Fowler, la plupart de ces détails ne sont pas nécessaires, et il tranche dedans pour aller à l'essentiel. Pour la plupart des projets, vous n'avez besoin de connaître que quelques diagrammes outils, et le but de Fowler est d'arriver à une bonne conception plutôt que de s'ennuyer avec tous les détails pour y arriver. Un petit livre agréable et intéressant ; le premier à lire si vous avez besoin de vous plonger dans UML.

UML Toolkit, de Hans-Erik Eriksson & Magnus Penker, (John Wiley & Sons, 1997). Explique UML et comment l'utiliser, avec un exemple appliqué en Java. Un CD ROM contient le code Java du livre et une version allégée de Rational Rose. Une excellente introduction sur UML et comment l'utiliser pour construire un système réel.

The Unified Software Development Process, de Ivar Jacobsen, Grady Booch, et James Rumbaugh (Addison-Wesley, 1999). Je m'étais attendu à détester ce livre. Il semblait avoir tous les défauts d'un texte universitaire ennuyeux. Je fus agréablement surpris - seules quelques parties du livre contiennent des explications qui font penser que les concepts ne sont pas clairs pour les auteurs. La majeure partie du livre n'est pas seulement claire, mais agréable à lire. Et le mieux, c'est que la méthodologie présentée n'est pas dénuée de sens pratique. Ce n'est pas la Programmation

Extreme (en particulier elle ne présente pas la même lucidité quant aux tests), mais elle fait partie du mastodonte UML - même si vous n'arrivez pas à faire adopter XP, la plupart des gens ont adhéré au « mouvement UML » (sans tenir compte de leur niveau *réel* d'expérience) et vous devriez pouvoir faire adopter cette méthode. Je pense que ce livre est le vaisseau amiral de UML, et celui que vous devriez lire après *UML Distilled* de Fowler quand vous aurez besoin de plus de détails.

Avant d'opter pour une méthode, il est bon de se renseigner auprès de personnes qui n'essayent pas d'en vendre une. Il est facile d'adopter une méthode sans réellement comprendre ce qu'on en attend ou ce qu'elle va faire pour nous. « D'autres l'utilisent, ce qui fait une bonne raison de l'adopter ». Cependant, les hommes peuvent montrer une bizarrerie psychologique : s'ils pensent que quelque chose va solutionner leurs problèmes, ils l'essayeront (c'est l'expérimentation, ce qui est une bonne chose). Mais si cela ne résoud pas leurs problèmes, ils peuvent redoubler d'efforts et annoncer à voix haute quelle découverte merveilleuse ils ont fait (c'est un mensonge, ce qui est une vilaine chose). Le raisonnement fait ici est que si vous arrivez à attirer d'autres personnes dans le même bateau, vous ne vous retrouverez pas tout seul, même s'il ne va nulle part (ou coule).

Cela ne veut pas dire que toutes les méthodologies se terminent en cul-de-sac, mais que vous devez vous préparer pour vous maintenir dans le mode expérimentation (« Cela ne marche pas, essayons quelque chose d'autre »), sans rentrer dans le mode mensonge (« Non, ce n'est pas réellement un problème. Tout marche, on n'a pas besoin de changer »). Je pense que les livres suivants, à lire *avant* de choisir une méthodologie, vous aident dans cette préparation.

Software Creativity, de Robert Glass (Prentice-Hall, 1995). Ceci est le meilleur livre que j'aie vu qui discute des perspectives du problème de la méthodologie. C'est un recueil de courts essais et articles que Glass a écrit et quelquefois récupéré (P. J. Plauger est l'un des contributeurs), reflétant ses années d'étude et de réflexion sur le sujet. Ils sont distrayants et juste assez longs pour expliquer ce qu'ils veulent faire passer ; il ne vous égare pas ni ne vous ennuie. Glass ne vous mène pas en bateau non plus, il y a des centaines de références à d'autres articles et études. Tous les programmeurs et les directeurs devraient lire ce livre avant de s'aventurer dans la jungle des méthodologies.

Software Runaways: Monumental Software Disasters, de Robert Glass (Prentice-Hall, 1997). Le point fort de ce livre est qu'il met en lumière ce dont personne ne parle : combien de projets non seulement échouent, mais échouent spectaculairement. La plupart d'entre nous pensent encore : « Cela ne peut pas m'arriver » (ou « Cela ne peut pas *encore* m'arriver »), et je pense que cela nous met en position de faiblesse. En se rappelant que les choses peuvent toujours mal se passer, on se retrouve en bien meilleure position pour les contrôler.

Peopleware*,* 2nd Edition, de Tom Demarco et Timothy Lister (Dorset House, 1999). Bien que les auteurs proviennent du monde du développement logiciel, ce livre traite des projets et des équipes en général. Mais il se concentre sur les *gens* et leurs besoins, plutôt que sur la technologie et ses besoins. Les auteurs parlent de la création d'un environnement où les gens seront heureux et productifs, plutôt que des décisions et des règles que ces gens devraient suivre pour être les composants bien huilés d'une machine. C'est cette dernière attitude qui, selon moi, fait sourire et acquiescer les programmeurs quand la méthode XYZ est adoptée et qu'ils continuent à travailler de la même manière qu'ils l'ont toujours fait.

Complexity, de M. Mitchell Waldrop (Simon & Schuster, 1992). Ce livre rapporte la mise en relations d'un groupe de scientifiques de différentes spécialités à Santa Fe, New Mexico, pour discuter de problèmes que leurs disciplines ne pouvaient résoudre individuellement (le marché de la bourse en économie, la création originelle de la vie en biologie, pourquoi les gens agissent ainsi qu'ils le font en sociologie, etc...). En combinant la physique, l'économie, la chimie, les mathématiques, l'informatique, la sociologie et d'autres matières, une approche multidisciplinaire envers ces

problèmes se développe. Mais plus important, une nouvelle façon de *penser* ces problèmes ultra-complexes émerge : loin du déterminisme mathématique et de l'illusion qu'on peut écrire une équation qui prédise tous les comportements, mais basée sur l'*observation* à la recherche d'un motif et la tentative de recréer ce motif par tous les moyens possibles. (Le livre rapporte, par exemple, l'émergence des algorithmes génétiques.) Je pense que cette manière de penser est utile lorsqu'on observe les façons de gérer des projets logiciels de plus en plus complexes.

Python

Learning Python, de Mark Lutz and David Ascher (O'Reilly, 1999). Une bonne introduction à ce qui est rapidement devenu mon langage favori, un excellent compagnon à Java. Le livre inclut une introduction à JPython, qui permet de combiner Java et Python dans un unique programme (l'interpréteur JPython est compilé en pur bytecode Java, il n'y a donc rien de spécial à ajouter pour réaliser cela). Cette union de langages promet de grandes possibilités.

La liste de mes livres

Listés par ordre de publication. Tous ne sont pas actuellement disponibles.

Computer Interfacing with Pascal & C, (Auto-publié dans la rubrique Eisys, 1988. Disponible seulement sur *www.BruceEckel.com*). Une introduction à l'électronique au temps où le CP/M était encore le roi et le DOS un parvenu. J'utilisais des langages de haut niveau et souvent le port parallèle de l'ordinateur pour piloter divers projets électroniques. Adapté de mes chroniques dans le premier et le meilleur des magazines pour lesquels j'ai écrit, *Micro Cornucopia* (pour paraphraser Larry O'Brien, éditeur au long cours de *Software Development Magazine* : le meilleur magazine informatique jamais publié - ils avaient même des plans pour construire un robot dans un pot de fleurs !). Hélas, Micro C s'est arrêté bien avant que l'Internet n'apparaisse. Créer ce livre fut une expérience de publication extrêmement satisfaisante.

Using C++, (Osborne/McGraw-Hill, 1989). Un des premiers livres sur le C++. Epuisé et remplacé par sa deuxième édition, renommée *C++ Inside & Out*.

C++ Inside & Out, (Osborne/McGraw-Hill, 1993). Comme indiqué, il s'agit en fait de la deuxième édition de **Using C++**. Le C++ dans ce livre est raisonnablement précis, mais il date de 1992 et *Thinking in C++* est destiné à le remplacer. Vous pouvez trouver plus de renseignements sur ce livre et télécharger le code source sur *www.BruceEckel.com*.

Thinking in C++, 1st Edition, (Prentice-Hall, 1995).

Thinking in C++, 2nd Edition, Volume 1, **(Prentice-Hall, 2000). Téléchargeable sur** *www.BruceEckel.com*.

Black Belt C++, the Master's Collection, Bruce Eckel, éditeur (M&T Books, 1994). Epuisé. Un recueil de chapitres par divers experts C++, basés sur leurs présentations dans le cycle C++ lors de la Conférence sur le Développement Logiciel, que je présidais. La couverture de ce livre m'a convaincu d'obtenir le contrôle de toutes les futures couvertures.

Thinking in Java, 1st Edition, (Prentice-Hall, 1998). La première édition de ce livre a gagné la *Software Development Magazine* Productivity Award, le *Java Developer's Journal* Editor's Choice Award, et le *JavaWorld Reader's Choice Award for best book*. Téléchargeable sur *www.BruceEckel.com*.