6e édition
Java 5 et 6

CLAUDE DELANNOY

Programmer en Java

EYROLLES

Best of EYROLLES

Programmer en Java

Le best-seller de Claude Delannoy, pour une parfaite maîtrise du langage Java

Réédition au format semi-poche de la sixième édition du classique *Programmer en Java* de Claude Delannoy, qui a guidé plus de 40 000 étudiants et professionnels dans l'apprentissage du langage Java.

L'ouvrage vous conduira à une parfaite maîtrise de la programmation orientée objet et des possibilités les plus avancées de Java dans ses versions 5 et 6. Après avoir assimilé la syntaxe de base du langage, vous découvrirez toutes les subtilités de la programmation objet en Java, avant d'aborder la programmation d'applications graphiques à l'aide de la bibliothèque Swing et le développement Web avec les servlets et les JSP.

Chaque notion nouvelle et chaque fonction du langage sont illustrées de programmes complets dont le code source est fourni sur le site *www.editions-eyrolles.com*. L'ouvrage met l'accent sur les nouveautés des versions 5 et 6 de Java Standard Edition (programmation générique, types énumérés, annotations…) et cette sixième édition inclut un nouveau chapitre dédié à l'accès aux bases de données avec JDBC.

Claude Delannoy

Ingénieur informaticien au CNRS, Claude Delannoy possède une grande pratique de la formation continue et de l'enseignement supérieur. Réputés pour la qualité de leur démarche pédagogique, ses ouvrages sur les langages et la programmation totalisent plus de 300 000 exemplaires vendus.

Au sommaire

Code éditeur : G12867
ISBN : 978-2-212-13443-8
Conception **Nord Compo**

www.editions-eyrolles.com

Programmer en Java

AUX ÉDITIONS EYROLLES

Du même auteur

C. Delannoy. – **Exercices en Java.**
N°13358, 3e édition, 2011, 350 pages.

C. Delannoy. – **Programmer en langage C++.**
N°12135, 8e édition, 2011, 820 pages.

C. Delannoy. – **C++ pour les programmeurs C.**
N°12231, 2007, 620 pages.

C. Delannoy. – **Exercices en langage C++.**
N°12201, 3e édition, 2007, 336 pages.

C. Delannoy. – **S'initier à la programmation.**
Avec des exemples en C, C++, C#, Java et PHP.
N°11990, 2008, 360 pages.

Autres ouvrages sur Java/JEE

A. Cogoluègne, T. Templier, J. Dubois, J.-P. Retaillé. – **Spring par la pratique.**
Spring 2.5 et 3.0.
N°12421, 2e édition, 2009, 678 pages.

A. Goncalves. – **Cahier du programmeur Java EE 5 et 6.**
N°12658, 3e édition, 2011, 340 pages.

A. Patricio. – **Java Persistence et Hibernate.**
N°12259, 2008, 390 pages.

Autres ouvrages

H. Bersini, I. Wellesz. – **La programmation orientée objet.**
Cours et exercices en UML 2 avec Java 6, C# 4, C++, Python, PHP 5 et LinQ.
N°12806, 5e édition, 2011, 664 pages.

P. Roques. – **UML 2 par la pratique.**
N°12565, 7e édition, 2009, 396 pages.

R. Goetter. – **CSS avancées.** *Vers HTML 5 et CSS 3.*
N°12826, 2011, 400 pages.

S. Jaber. – **Programmation GWT 2.**
Développer des applications RIA et Ajax avec le Google Web Toolkit.
N°12569, 2010, 484 pages.

A. Brillant. – **XML.** *Cours et exercices*.
N°12691, 2e édition, 2010, 336 pages.

G. Swinnen. – **Apprendre à programmer avec Python.**
N°12708, 2010, 410 pages.

G. Leblanc. – **C# et .NET.** *Versions 1 à 4.*
N°12604, 2009, 910 pages.

EYROLLES

6e édition,
2e tirage 2011

CLAUDE DELANNOY

Programmer en Java

EYROLLES

ÉDITIONS EYROLLES
61, bld Saint-Germain
75240 Paris Cedex 05
www.editions-eyrolles.com

La 6e édition de cet ouvrage a fait l'objet d'un reconditionnement à l'occasion de son 2e tirage (format semi-poche et nouvelle couverture).
Le texte de l'ouvrage reste inchangé par rapport au tirage précédent.

 ISBN : 978-2-212-13443-8

Table des matières

Avant-propos

À qui s'adresse ce livre

Cet ouvrage est destiné à tous ceux qui souhaitent maîtriser la programmation en Java. Il s'adresse à la fois aux étudiants, aux développeurs et aux enseignants en informatique.

Il suppose que le lecteur possède déjà une expérience de la programmation dans un autre langage (Cobol, Pascal, C, C++, Visual Basic, Delphi, PHP, Perl, Python...). En revanche, la connaissance de la programmation orientée objet n'est nullement nécessaire, pas plus que celle de la programmation d'interfaces graphiques ou d'applications Web.

Contenu de l'ouvrage

Les fondements de Java

Les chapitres 1 à 11 sont consacrés aux fondements du langage : types primitifs, opérateurs et expressions, instructions, classes, héritage, tableaux et chaînes de caractères. Les aspects les plus fondamentaux de la programmation orientée objet que sont le polymorphisme, la surdéfinition et la redéfinition des méthodes y sont également étudiés de façon approfondie, aussi bien dans leur puissance que dans leurs limitations.

Tous les aspects du langage sont couverts, y compris ceux qui sont spécifiques à Java, comme les interfaces, les classes internes, les classes anonymes, les exceptions ou les threads. Les

moins usités font généralement l'objet d'un paragraphe intitulé *Informations complémentaires* dont la connaissance n'est pas indispensable à l'étude de la suite de l'ouvrage.

Par ailleurs, le chapitre 21 présente les possibilités de programmation générique introduites récemment. Sa place tardive dans l'ouvrage est surtout justifiée par son lien étroit avec les collections présentées au chapitre 22.

Enfin, le chapitre 25 présente les annotations et les techniques d'introspection.

Les principales API

Le JDK (*Java Developpement Kit*) de Java est livré, en standard, avec différentes bibliothèques, paquetages ou API (*Application Programming Interface*) fournissant de nombreuses classes utilitaires. Les chapitres 12 à 20 et 22 à 24 examinent les API qui correspondent aux besoins les plus universels et qui, à ce titre, peuvent être considérés comme partie intégrante du langage.

Les chapitres 12 à 19 sont consacrés à la programmation d'interfaces graphiques en Java à l'aide de l'API nommée *Swing* : événements et écouteurs ; boutons, cases à cocher et boutons radio ; boîtes de dialogue ; menus ; barres d'outils ; actions abstraites ; événements générés par le clavier, la souris, les fenêtres et la focalisation ; gestionnaires de mise en forme ; affichage de textes et de dessins ; applets. Dans cette partie, l'accent est mis sur les mécanismes fondamentaux qui interviennent en programmation graphique et événementielle.

Le chapitre 20 traite de l'API relative aux entrées-sorties, unifiées à travers la notion de flux. Il intègre un exemple de connexion TCP/IP par sockets.

Le chapitre 22 décrit les principales structures de données qu'on regroupe souvent sous le terme de collection : listes, ensembles, vecteurs dynamiques, queues et tables associatives.

Le chapitre 23 se veut une introduction aux possibilités de programmation côté serveur, offertes par les servlets, les JSP et les JavaBeans. En toute rigueur, il s'agit là, non plus d'API standard de Java, mais de spécifications de JEE (*Java Enterprise Edition*).

Enfin, le chapitre 24 présente l'API standard JDBC permettant d'exploiter des bases de données locales ou distantes.

Pour aller plus loin

Après l'étude de cet ouvrage consacré à ce que l'on pourrait appeler les « bases élargies du langage », le lecteur pourra appréhender aisément l'importante documentation des classes standards Java et de leurs méthodes[1]. Il sera alors parfaitement armé pour développer ses propres applications, aussi complexes et spécialisées soient-elles, notamment les applications côté serveur à base d'EJB ou les applications distribuées, sujets non traités dans cet ouvrage.

1. Par exemple, en consultant le site officiel de java : *http://java.sun.com/reference/docs/*.

Forme de l'ouvrage

L'ouvrage est conçu sous la forme d'un cours. Il expose progressivement les différentes notions fondamentales, en les illustrant systématiquement de programmes complets accompagnés d'un exemple d'exécution.

Pour en faciliter l'assimilation, les fondements du langage sont présentés de façon indépendante de la programmation d'interfaces graphiques, en s'appuyant sur les possibilités qu'offre Java d'écrire des applications à interface *console*.

Dans la partie consacrée à la programmation graphique, les composants sont introduits suffisamment progressivement pour permettre au lecteur de les découvrir en tant qu'utilisateur de logiciel. L'expérience montre en effet, que, pour réaliser une bonne interface graphique, un développeur doit non seulement savoir programmer correctement les composants concernés, mais également bien connaître leur ergonomie.

Outre son caractère didactique, nous avons conçu l'ouvrage d'une manière très structurée pour qu'il puisse être facilement consulté au-delà de la phase d'apprentissage du langage. Dans cet esprit, il est doté d'une table des matières détaillée et d'un index fourni dans lequel les noms de méthodes sont toujours accompagnés du nom de la classe correspondante (il peut y avoir plusieurs classes). Les exemples complets peuvent servir à une remémoration rapide du concept qu'ils illustrent. Des encadrés permettent de retrouver rapidement la syntaxe d'une instruction, ainsi que les règles les plus importantes. Enfin, des annexes fournissent des aide-mémoire faciles à consulter :

- liste des fonctions mathématiques (classe *Math*) ;
- liste des exceptions standards ;
- liste des composants et des en-têtes de leurs méthodes ;
- liste des événements, écouteurs et méthodes correspondantes ;
- liste des classes et interfaces liées aux collections et méthodes correspondantes ;
- outils de professionnalisation des applications (pour la plupart introduits par Java 6).

L'ouvrage, les versions de Java et C++

Si les instructions de base de Java n'ont pratiquement pas évolué depuis sa naissance jusqu'à sa version 5, il n'en va pas de même de ses bibliothèques standards. En particulier, le modèle de gestion des événements a été fortement modifié par la version 1.1. Une nouvelle bibliothèque de composants graphiques, Swing, est apparue dans la version 1.2 de l'édition Standard de Java, renommée à cette occasion J2SE (Java 2 Standard Edition). Après deux nouvelles versions nommées respectivement J2SE 1.3 et J2SE 1.4, Sun a modifié son système de numérotation en introduisant J2SE 5 en 2004, puis Java SE 6 en 2006. Nous parlerons plus simplement de Java 5 et Java 6 pour nous référer à ces deux dernières.

Depuis J2SE 1.2, chaque édition standard de Java complétée par un ensemble de spécifications, nommé J2EE (Java 2 Enterprise Edition) jusqu'à la version 4 et JEE (Java Enterprise Edition) depuis la version 5 ; ces spécifications sont dédiées notamment au développement côté serveur et aux applications réparties[1].

La version standard J2SE 5 a introduit bon nombre de nouveautés fondamentales : types énumérés, types enveloppes, boxing/unboxing automatiques, arguments variables en nombre, boucle *for... each*. Un chapitre de l'ouvrage est consacré aux possibilités de programmation générique. Le chapitre relatif aux collections est prévu pour tenir compte de leur aspect générique, mais aussi pour permettre l'utilisation d'anciens codes.

L'ouvrage prend également en compte les apports de la version Java 6. Notamment, nous présentons le nouveau gestionnaire de mise en forme qu'est *GroupLayout*, ainsi que les fonctionnalités permettant de professionnaliser une application (classe *Desktop*, classe *Console*, action sur la barre des tâches du système). Nous tenons compte des nouvelles possibilités offertes par la classe *File* ainsi que des nouvelles interfaces et classes de collections (*Deque*, *ArayDeque*, *NavigableSet*, *NavigableMap*). En outre, le chapitre 25 présente les annotations (déjà présentes dans Java 5, mais mieux intégrées dans le langage depuis Java 6) et décrit en même temps les possiblités d'introspection qui permettent d'en tirer véritablement profit.

Compte tenu de la popularité du langage C++, nous avons introduit de nombreuses remarques titrées *En C++*. Elles mettent l'accent sur les liens étroits qui existent entre Java et C++, ainsi que sur leurs différences. Elles offriront des passerelles utiles non seulement au programmeur C++ qui apprend ici Java, mais également au lecteur qui, après la maîtrise de Java, souhaitera aborder l'étude de C++[2].

1. Il existe une troisième édition de Java, JME (Java Micro Edition), destinée aux développements d'applications embarquées pour les téléphones mobiles, les assistants personnels et divers appareils électroniques grand public.

2. L'ouvrage *Apprendre le C++*, du même auteur, chez le même éditeur, s'adresse à un public ayant déjà la maîtrise d'un langage tel que Java.

1

Présentation de Java

Après un très bref historique du langage Java montrant dans quel esprit il a été créé, nous en présenterons les principales caractéristiques. Nous verrons tout d'abord que Java est un langage objet et nous exposerons les concepts majeurs de la programmation orientée objet. Puis nous ferons la distinction entre les programmes utilisant une interface console et les programmes utilisant une interface graphique, ce qui nous amènera à parler des possibilités de programmation événementielle qui sont offertes par Java sous la forme de classes standards Enfin, nous montrerons que Java est le premier langage à offrir une portabilité aussi avancée.

1 Petit historique du langage

On peut faire remonter la naissance de Java à 1991. À cette époque, des ingénieurs de chez SUN ont cherché à concevoir un langage applicable à de petits appareils électriques (on parle de *code embarqué*). Pour ce faire, ils se sont fondés sur une syntaxe très proche de celle de C++, en reprenant le concept de machine virtuelle déjà exploité auparavant par le Pascal UCSD. L'idée consistait à traduire d'abord un programme source, non pas directement en langage machine, mais dans un pseudo langage universel, disposant des fonctionnalités communes à toutes les machines. Ce code intermédiaire, dont on dit qu'il est formé de *byte codes*[1], se trouve ainsi compact et portable sur n'importe quelle machine ; il suffit simplement que cette dernière dispose d'un programme approprié (on parle alors de *machine virtuelle*) permettant de l'interpréter dans le langage de la machine concernée.

1. Conformément à la tradition, nous n'avons pas cherché à traduire ce terme.

En fait, ce projet de langage pour code embarqué n'a pas abouti en tant que tel. Mais ces concepts ont été repris en 1995 dans la réalisation du logiciel *HotJava*, un navigateur Web écrit par SUN en Java, et capable d'exécuter des *applets* écrits précisément en *byte codes*.

Les autres navigateurs Web ont suivi, ce qui a contribué à l'essor du langage qui a beaucoup évolué depuis cette date, sous forme de versions successives : 1.01 et 1.02 en 1996, 1.1 en 98 et 1.2 (finalement rebaptisée Java 2) en 1999, 1.3 en 2000, 1.4 en 2002, 5.0 en 2004 (toujours appelées Java 2). Ainsi parle-t-on du J2SE 1.4 (Java 2 Standard Edition 1.4) basée sur le JDK 1.4 (Java Development Kit 1.4), plus récemment du J2SE5.0 (JDK 5.0) ou encore de Java 5. En revanche, la dernière version s'intitule JSE 6 (le 2 a disparu !), ou plus simplement Java 6[1].

On notera que, au fil des différentes versions, les aspects fondamentaux du langage ont peu changé (ils ont quand même été complétés de façon substancielle par Java 5, notamment par l'introduction de la programmation générique et du remaniement des "collections"). En revanche, les bibliothèques standards (API) ont beaucoup évolué, à la fois par des modifications et par des ajouts. Il en va d'ailleurs de même des ensembles de spécifications accompagnant chaque version standard de Java (J2EE jusqu'à la version 4 et JEE depuis la version 5) .

2 Java et la programmation orientée objet

La P.O.O. (programmation orientée objet) possède de nombreuses vertus universellement reconnues désormais. Notamment, elle ne renie pas la programmation structurée (elle se fonde sur elle), elle contribue à la fiabilité des logiciels et elle facilite la réutilisation de code existant. Elle introduit de nouveaux concepts, en particulier ceux d'objets, d'encapsulation, de classe et d'héritage.

2.1 Les concepts d'objet et d'encapsulation

En programmation structurée, un programme est formé de la réunion de différentes procédures et de différentes structures de données généralement indépendantes de ces procédures.

En P.O.O., un programme met en œuvre différents *objets*. Chaque objet associe des *données* et des *méthodes* agissant exclusivement sur les données de l'objet. Notez que le vocabulaire évolue quelque peu : on parlera de *méthodes* plutôt que de *procédures* ; en revanche, on pourra utiliser indifféremment le mot *données* ou le mot *champ*.

Mais cette association est plus qu'une simple juxtaposition. En effet, dans ce que l'on pourrait qualifier de P.O.O. "pure", on réalise ce que l'on nomme une *encapsulation des données*. Cela signifie qu'il n'est pas possible d'agir directement sur les données d'un objet ; il est nécessaire de passer par ses méthodes, qui jouent ainsi le rôle d'interface obligatoire. On tra-

1. Attention, la documentation de référence de Sun comporte toujours les numéros de version de la forme 1.1, 1.2, 1.3, 1.4, 1.5 (pour Java 5) et 1.6 (pour Java 6).

duit parfois cela en disant que l'appel d'une méthode est en fait l'envoi d'un message à l'objet.

Le grand mérite de l'encapsulation est que, vu de l'extérieur, un objet se caractérise uniquement par les spécifications de ses méthodes, la manière dont sont réellement implantées les données étant sans importance. On décrit souvent une telle situation en disant qu'elle réalise une *abstraction des données* (ce qui exprime bien que les détails concrets d'implémentation sont cachés). À ce propos, on peut remarquer qu'en programmation structurée, une procédure pouvait également être caractérisée (de l'extérieur) par ses spécifications, mais que, faute d'encapsulation, l'abstraction des données n'était pas réalisée.

L'encapsulation des données présente un intérêt manifeste en matière de qualité de logiciel. Elle facilite considérablement la maintenance : une modification éventuelle de la structure des données d'un objet n'a d'incidence que sur l'objet lui-même ; les utilisateurs de l'objet ne seront pas concernés par la teneur de cette modification (ce qui n'était bien sûr pas le cas avec la programmation structurée). De la même manière, l'encapsulation des données facilite grandement la réutilisation d'un objet.

2.2 Le concept de classe

Le concept de classe correspond simplement à la généralisation de la notion de type que l'on rencontre dans les langages classiques. En effet, une classe n'est rien d'autre que la description d'un ensemble d'objets ayant une structure de données commune et disposant des mêmes méthodes. Les objets apparaissent alors comme des variables d'un tel type classe (en P.O.O., on dit aussi qu'un objet est une *instance* de sa classe). Bien entendu, seule la structure est commune, les valeurs des champs étant propres à chaque objet. En revanche, les méthodes sont effectivement communes à l'ensemble des objets d'une même classe.

Lorsque, comme cela arrive parfois dans l'écriture d'interfaces graphiques, on est amené à ne créer qu'un seul objet d'une classe donnée, la distinction entre les notions d'objet et de classe n'est pas toujours très évidente.

En revanche, lorsque l'on dispose de plusieurs objets d'une même classe, le principe d'encapsulation s'appliquera à la classe et non à chacune de ses instances, comme nous le verrons.

2.3 L'héritage

Un autre concept important en P.O.O. est celui d'héritage. Il permet de définir une nouvelle classe à partir d'une classe existante (qu'on réutilise en bloc !), à laquelle on ajoute de nouvelles données et de nouvelles méthodes. La conception de la nouvelle classe, qui hérite des propriétés et des aptitudes de l'ancienne, peut ainsi s'appuyer sur des réalisations antérieures parfaitement au point et les spécialiser à volonté. Comme on peut s'en douter, l'héritage facilite largement la réutilisation de produits existants, d'autant plus qu'il peut être réitéré autant de fois que nécessaire (la classe C peut hériter de B, qui elle-même hérite de A).

Cette technique s'appliquera aussi bien aux classes que vous serez amené à développer qu'aux très nombreuses classes fournies en standard avec Java.

Certains langages, tels C++, offrent la possibilité d'un héritage multiple : une même classe peut hériter simultanément de plusieurs autres. Ce n'est pas le cas de Java, mais nous verrons que la notion d'*interface* permet de traiter plus élégamment les situations correspondantes.

2.4 Le polymorphisme

En Java, comme généralement, en P.O.O., une classe peut "redéfinir" (c'est-à-dire modifier) certaines des méthodes héritées de sa classe de base. Cette possibilité est la clé de ce que l'on nomme le "polymorphisme", c'est-à-dire la possibilité de traiter de la même manière des objets de types différents, pour peu qu'ils soient issus de classes dérivées d'une même classe de base. Plus précisément, on utilise chaque objet comme s'il était de cette classe de base, mais son comportement effectif dépend de sa classe effective (dérivée de cette classe de base), en particulier de la manière dont ses propres méthodes ont été redéfinies. Le polymorphisme permet d'ajouter de nouveaux objets dans un scénario préétabli et, éventuellement, écrit avant d'avoir connaissance du type exact de ces objets.

2.5 Java est presque un pur langage de P.O.O.

Certains langages ont été conçus pour appliquer à la lettre les principes de P.O.O. C'est notamment le cas de Simula, Smalltalk et de Eiffel. Dans ce cas, tout est objet (ou instance de classe) et l'encapsulation des données est absolue. Les procédures sont obligatoirement des méthodes, ce qui revient à dire qu'il n'existe pas de procédures indépendantes, c'est-à-dire susceptibles de s'exécuter indépendamment de tout objet.

D'autres langages, comme Pascal ou C++, ont cherché à appliquer une "philosophie objet" à un langage classique. Les objets y cohabitent alors avec des variables usuelles. Il existe à la fois des méthodes, applicables à un objet, et des procédures indépendantes. À à la limite, on peut réaliser un programme ne comportant aucun objet.

Java se veut un langage de la première catégorie, autrement dit un pur langage de P.O.O. Par nature, un programme s'y trouvera formé d'une classe ou de la réunion de plusieurs classes et il instanciera des objets. Il sera impossible de créer un programme n'utilisant aucune classe. Cependant, il faut apporter quelques nuances qui troublent très légèrement la pureté du langage.

- Java dispose de types dits primitifs pour représenter les entiers, les flottants, les caractères et les booléens. Les variables correspondantes ne sont pas des objets. Certes, la plupart du temps, ces types primitifs seront utilisés pour définir les champs d'une classe, donc finalement d'un objet ; cependant, il y aura des exceptions...
- Une classe pourra comporter des méthodes particulières dites *méthodes de classe* (déclarées avec le mot-clé *static*) qui seront utilisables de façon indépendante d'un objet. Comme ces méthodes peuvent déclarer localement des variables d'un type primitif, on voit qu'on peut

ainsi retrouver les possibilités des procédures ou des fonctions des langages non objet. La seule différence (purement syntaxique) viendra de ce que ces méthodes seront localisées artificiellement dans une classe (on verra qu'il existe une telle méthode nommée *main* jouant le rôle de programme principal). À la limite, on peut concevoir un programme ne comportant aucun objet (mais obligatoirement au moins une classe). C'est d'ailleurs cette particularité que nous exploiterons pour vous exposer les bases du langage, en dehors de tout contexte objet.

- L'encapsulation se trouve naturellement induite par la syntaxe du langage mais elle n'est pas absolue.

3 Java et la programmation événementielle

3.1 Interface console ou interface graphique

Actuellement, on peut distinguer deux grandes catégories de programmes, en se fondant sur leur *interface* avec l'utilisateur, c'est-à-dire sur la manière dont se font les échanges d'informations entre l'utilisateur et le programme :

- les programmes à interface console,
- les programmes à interface graphique.

3.1.1 Les programmes à interface console (ou en ligne de commande)

Historiquement, ce sont les plus anciens. Dans de tels programmes, on fournit des informations à l'écran sous forme de lignes de texte s'affichant séquentiellement, c'est-à-dire les unes à la suite des autres. Pour fournir des informations au programme, l'utilisateur frappe des caractères au clavier (généralement un "écho" apparaît à l'écran).

Entrent dans cette catégorie :

- les programmes fonctionnant sur PC sous DOS ou, plus fréquemment, dans une fenêtre DOS de Windows,
- les programmes fonctionnant sous Unix ou Linux et s'exécutant dans une "fenêtre de commande".

Avec une interface console, c'est le programme qui décide de l'enchaînement des opérations : l'utilisateur est sollicité au moment voulu pour fournir les informations demandées[1].

1. En toute rigueur, les informations fournies peuvent influer sur le déroulement ultérieur du programme ; il n'en reste pas moins que l'interface console n'offre pas à l'utilisateur la sensation d'initiative qu'il trouvera dans une interface graphique.

3.1.2 Les programmes à interface graphique (G.U.I.)

Dans ces programmes, la communication avec l'utilisateur se fait par l'intermédiaire de *composants* tels que les menus déroulants, les menus surgissants, les barres d'outils ou les boîtes de dialogue, ces dernières pouvant renfermer des composants aussi variés que les boutons poussoirs, les cases à cocher, les boutons radio, les boîtes de saisie, les listes déroulantes...

L'utilisateur a l'impression de piloter le programme, qui semble répondre à n'importe laquelle de ses demandes. D'ailleurs, on parle souvent dans ce cas de *programmation événementielle*, expression qui traduit bien le fait que le programme réagit à des évènements provoqués (pour la plupart) par l'utilisateur.

On notera que le terme G.U.I. (*Graphical User Interface*) tend à se généraliser pour désigner ce genre d'interface. Manifestement, il met en avant le fait que, pour permettre ce dialogue, on ne peut plus se contenter d'échanger du texte et qu'il faut effectivement être capable de dessiner, donc d'employer une interface graphique. Il n'en reste pas moins que l'aspect le plus caractéristique de ce type de programme est dans l'aspect événementiel[1].

3.2 Les fenêtres associées à un programme

3.2.1 Cas d'une interface console

L'interface console n'utilise qu'une seule fenêtre (dans certains anciens environnement, la fenêtre n'était même pas visible, car elle occupait tout l'écran). Celle-ci ne possède qu'un petit nombre de fonctionnalités : déplacement, fermeture, parfois changement de taille et défilement.

3.2.2 Cas d'une interface graphique

L'interface graphique utilise une fenêtre principale qui s'ouvre au lancement du programme. Il est possible que d'autres fenêtre apparaissent par la suite : l'exemple classique est celui d'un logiciel de traitement de texte qui manipule différents documents associés chacun à une fenêtre.

L'affichage des informations dans ces fenêtres ne se fait plus séquentiellement. Il est généralement nécessaire de prendre en compte l'aspect "coordonnées". En contrepartie, on peut afficher du texte en n'importe quel emplacement de la fenêtre, utiliser des polices différentes, jouer sur les couleurs, faire des dessins, afficher des images...

3.3 Java et les interfaces

3.3.1 La gestion des interfaces graphiques est intégrée dans Java

Dans la plupart des langages, on dispose d'instructions ou de procédures standards permettant de réaliser les entrées-sorties en mode console.

---

1. Bien entendu, une des "retombées" de l'utilisation d'une interface graphique est que le programme pourra afficher des graphiques, des dessins, des images...

En revanche, les interfaces graphiques doivent être programmées en recourant à des instructions ou à des bibliothèques spécifiques à chaque environnement (par exemple *X11* ou *Motif* sous Unix, *API Windows*, *MFC* ou *Object Windows* sous *Windows*).

L'un des grands mérites de Java est d'intégrer des outils (en fait des classes standards) de gestion des interfaces graphiques. Non seulement on pourra utiliser le même code source pour différents environnements mais, de plus, un programme déjà compilé (*byte codes*) pourra s'exécuter sans modification sur différentes machines.

3.3.2 Applications et applets

À l'origine, Java a été conçu pour réaliser des *applets* s'exécutant dans des pages *Web*. En fait, Java permet d'écrire des programmes indépendants du Web. On parle alors d'*applications* (parfois de "vraies applications").

Les fonctionnalités graphiques à employer sont quasiment les mêmes pour les applets et les applications. D'ailleurs, dans cet ouvrage, nous présenterons l'essentiel de Java en considérant des applications. Un seul chapitre sera nécessaire pour présenter ce qui est spécifique aux applets.

Théoriquement, une applet est faite pour que son code (compilé) soit téléchargé dans une page Web. Autrement dit, il peut sembler indispensable de recourir à un navigateur pour l'exécuter (pas pour la compiler). En fait, quel que soit l'environnement, vous disposerez toujours d'un visualisateur d'applets vous permettant d'exécuter une applet en dehors du Web.

3.3.3 On peut disposer d'une interface console en Java

A priori, Java a été conçu pour développer des applets ou des applications utilisant des interfaces graphiques.

En fait, en plus des fenêtres graphiques qu'elle est amenée à créer, toute application dispose automatiquement d'une fenêtre dans laquelle elle peut (sans y être obligée) réaliser des entrées-sorties en mode console.

Cette possibilité pourra s'avérer très précieuse lors de la phase d'apprentissage du langage. En effet, on pourra commencer à écrire du code, sans avoir à maîtriser les subtilités de la gestion des interfaces graphiques. Cet aspect sera d'autant plus intéressant que, comme on le verra par la suite, Java permet de lancer une application sans que cette dernière ne soit obligée de créer une fenêtre principale.

On pourra aussi utiliser une fenêtre console lors de la mise au point d'un programme pour y afficher différentes informations de traçage du code.

Enfin, vous disposerez automatiquement d'une fenêtre console si vous lancez une applet depuis le visualisateur d'applet.

4 Java et la portabilité

Dans la plupart des langages, on dit qu'un programme est portable car un même code source peut être exploité dans des environnements différents moyennant simplement une nouvelle compilation.

En Java, la portabilité va plus loin. En effet, comme nous l'avons évoqué précédemment, la compilation d'un code source produit, non pas des instructions machine, mais un code intermédiaire formé de *byte codes*. D'une part, ce code est exactement le même, quel que soit le compilateur et l'environnement concernés. D'autre part, ces *byte codes* sont exécutables dans toute implémentation disposant du logiciel d'interprétation nommé *machine virtuelle*[1] ou, parfois, *système d'exécution Java*.

De surcroît, Java définit exactement les caractéristiques des types primitifs servant à représenter les caractères, les entiers et les flottants. Cela concerne non seulement la taille de l'emplacement mémoire, mais aussi le comportement arithmétique correspondant. Ainsi, quelle que soit la machine, une valeur de type *float* (réel) aura exactement même taille, mêmes limites et même précision. Java est ainsi le premier langage qui assure qu'un même programme, exécuté dans des environnements différents, fournira les mêmes résultats[2].

1. JVM (abréviation de *Java Virtual Machine*).

2. À l'erreur de représentation près (qui, dans des calculs complexes, peut quand même avoir une incidence importante !).

2

Généralités

Ce chapitre constitue une première approche d'un programme Java, fondée sur quelques exemples commentés. Vous y verrez, de manière informelle pour l'instant, comment s'expriment les instructions de base (déclaration, affectation, écriture...), ainsi que deux structures fondamentales (boucle avec compteur et choix). Cela nous permettra par la suite d'illustrer certaines notions par des programmes complets, compréhensibles avant même que nous n'ayons effectué une étude détaillée des instructions correspondantes.

Nous dégagerons ensuite quelques règles générales concernant l'écriture d'un programme. Enfin, nous montrerons comment mettre en œuvre un programme Java, de sa saisie à son exécution, ce qui vous permettra de vous familiariser avec votre propre environnement de développement.

Notez que nous exploiterons ici les possibilités de simplification présentées au chapitre précédent. D'une part, nous nous limiterons à des programmes utilisant une interface de type console ; d'autre part, nous ne ferons pas intervenir d'objets. Autrement dit, ce chapitre se bornera à vous montrer comment s'expriment en Java des concepts que vous avez déjà rencontrés dans d'autres langages (C, C++, Visual Basic, C#, PHP...).

1 Premier exemple de programme Java

Voici un exemple très simple de programme qui se contente d'afficher dans la fenêtre console le texte : "Mon premier programme Java".

```
public class PremProg
{ public static void main (String args[])
  { System.out.println ("Mon premier programme Java") ;
  }
}
```

Mon premier programme Java

1.1 Structure générale du programme

Vous constatez que, globalement, sa structure se présente ainsi[1] :

```
public class PremProg
{  .....
}
```

Elle correspond théoriquement à la définition d'une classe nommée *PremProg*. La première ligne identifie cette classe ; elle est suivie d'un bloc, c'est-à-dire d'instructions délimitées par des accolades { et } qui définissent le contenu de cette classe. Ici, cette dernière est réduite à la seule définition d'une "méthode" particulière nommée *main* :

```
public static void main (String [] args)
{ System.out.println ("Mon premier programme Java") ;
}
```

Là encore, une première ligne identifie la méthode ; elle est suivie d'un bloc ({ ..... }) qui en fournit les différentes instructions.

Pour l'instant, vous pouvez vous contenter d'utiliser un tel canevas, sans vraiment connaître les notions de classe et de méthode. Il vous suffit simplement de placer dans le bloc le plus interne les instructions de votre choix, comme vous le feriez dans le programme principal (ou la fonction principale) d'un autre langage.

Simplement, afin d'utiliser dès maintenant le vocabulaire approprié, nous parlerons de la méthode *main* de notre programme formé ici d'une seule classe nommée *PremProg*.

Informations complémentaires

Si vous souhaitez en savoir un peu plus, voici quelques indications supplémentaires que vous retrouverez lorsque nous étudierons les classes.

1 Le mot-clé *static* précise que la méthode *main* de la classe *PremProg* n'est pas liée à une instance (objet) particulière de la classe. C'est ce qui fait de cette méthode l'équivalent d'une procédure ou d'une fonction usuelle des autres langages. En outre, comme

1. Contrairement à ce qui se produit en C++, on ne trouve pas de point-virgule à la fin de la définition de la classe.

elle porte le nom *main*, il s'agit de la fonction principale, c'est-à-dire de l'équivalent de la fonction *main* du C ou du programme principal de Pascal.

2 Le paramètre *String[] args* de la fonction *main* permet de récupérer des arguments transmis au programme au moment de son lancement. On peut lancer un programme sans fournir d'arguments, mais l'indication *String args[]* est obligatoire (en C/C++, on trouve des paramètres similaires dans la fonction *main*, mais ils sont facultatifs).

   Vous pouvez indifféremment écrire *String[] args* ou *String args[]*. Vous verrez plus tard que ce paramètre *args* est un tableau d'objets de type *String*, servant à représenter des chaînes de caractères. Comme pour tout paramètre d'une fonction, son nom peut être choisi librement ; vous pourriez tout aussi bien utiliser *infos*, *valeurs*, *param*... Toutefois, la tradition veut qu'on utilise plutôt *args*.

3 Le mot-clé *public* dans *public class PremProg* sert à définir les droits d'accès des autres classes (en fait de leurs méthodes) à la classe *PremProg*. Comme manifestement, aucune autre classe n'a besoin de *PremProg*, le mot-clé *public* pourrait être omis. Cependant, comme cela pourrait vous conduire à prendre de mauvaises habitudes en ce qui concerne l'organisation de vos fichiers source, nous vous conseillons de le conserver (au moins pour l'instant).

4 Le mot-clé *public* dans *public static void main* est obligatoire pour que votre programme puisse s'exécuter. Ici, il ne s'agit plus véritablement d'un problème de droit d'accès, mais plutôt d'une convention qui permet à la machine virtuelle d'accéder à la méthode *main*. Notez que vous pouvez inverser l'ordre des mots-clés *public* et *static* en écrivant *static public void main*.

1.2 Contenu du programme

Notre programme comporte ici une seule instruction :

```
System.out.println ("Mon premier programme Java") ;
```

Si vous aviez simplement trouvé

```
println ("Mon premier programme Java") ;
```

les choses vous auraient probablement paru assez intuitives, le mot *println* apparaissant comme l'abréviation de *print line* (affichage suivi d'un changement de ligne).

Pour l'instant, vous pouvez vous contenter de considérer que *System.out.println* correspond à une méthode d'affichage dans la fenêtre console, méthode à laquelle on mentionne un texte à afficher sous forme d'une constante chaîne usuelle (entre guillemets, comme dans la plupart des langages).

Il existe également une méthode *System.out.print* qui fait la même chose, avec cette seule différence qu'elle ne provoque pas de changement de ligne après affichage. Ainsi, l'unique instruction de notre programme pourrait être (artificiellement) remplacée par :

```
System.out.print ("Mon premier programme ") ;
System.out.println ("Java") ;
```

Informations complémentaires

Nous aurons bientôt l'occasion de voir comment afficher autre chose que des chaînes constantes. Sachez que la notation *System.out.println* fait également appel aux notions d'objet et de méthode. Plus précisément, *System* désigne une classe dans laquelle se trouve défini un champ donnée *out*, représentant la fenêtre console. Ici encore, ce champ possède l'attribut *static*, ce qui signifie qu'il existe indépendamment de tout objet de type *System*. C'est pourquoi on le désigne par *System.out* (alors qu'un champ non statique serait repéré par un nom d'objet et non plus par un nom de classe). Enfin, la méthode *println* est une méthode (classique, cette fois) de la classe dont est issu l'objet *out* (il s'agit de la classe *PrintStream*). La notation *System.out.println* représente l'appel de la méthode *println* associée à l'objet *System.out*.

Par ailleurs, le JDK 5.0 a introduit une méthode *printf* permettant de "formater" l'affichage des informations, à la manière de l'instruction *printf* du langage C.

2 Exécution d'un programme Java

Pour mettre en œuvre notre précédent programme, il faut bien sûr le saisir et le sauvegarder dans un fichier. Ici, ce dernier devra impérativement se nommer *PremProg.java.* En effet, nous verrons que, quel que soit l'environnement concerné, le code source d'une classe publique[1] doit toujours se trouver dans un fichier portant le même nom et possédant l'extension *java*.

Ensuite, on procède à la compilation de ce fichier source. Rappelons que celle-ci produit non pas du code machine, mais un code intermédiaire formé de *bytecodes*. Si la compilation s'est bien déroulée, on obtiendra un fichier portant le même nom que le fichier source et l'extension *class*, donc ici *PremProg.class*. On pourra lancer l'exécution des *byte codes* ainsi obtenus par l'intermédiaire de la machine virtuelle Java. Bien entendu, on pourra exécuter autant de fois qu'on le voudra un même programme, sans avoir besoin de le recompiler.

La démarche à employer pour procéder à ces différentes étapes dépend tout naturellement de l'environnement de développement avec lequel on travaille. S'il s'agit du JDK[2] de SUN, on compilera avec la commande :

```
javac PremProg.java
```

On exécutera avec la commande suivante (attention à ne pas mentionner d'extension à la suite du nom du programme) :

```
java PremProg
```

1. Certes, comme il a été dit précédemment, nous ne sommes pas obligés de déclarer notre classe publique. Toutefois, cette possibilité est déconseillée pour l'instant.

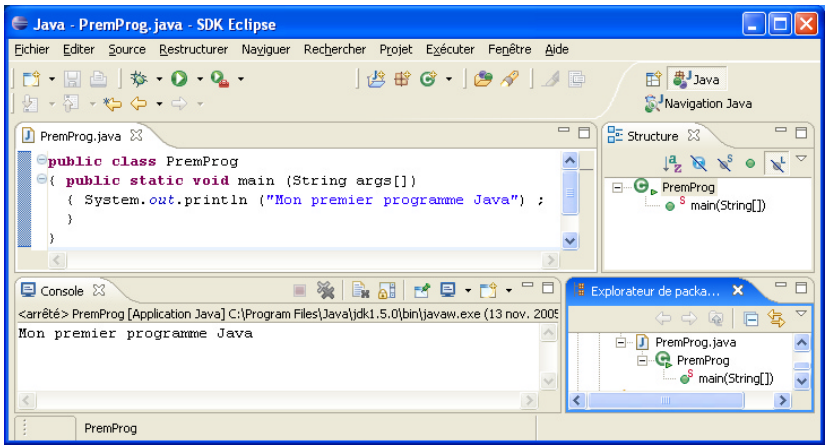
2. *Java Developpment Kit* : kit de développement Java.

À la suite de cette dernière commande, on obtiendra les résultats dans la même fenêtre, qui ressemblera donc à ceci (en fait, les commandes seront probablement précédées d'un "prompt") :

```
javac PremProg.java
java PremProg
Mon premier programme Java
```

Exemple d'exécution du programme PremProg *(1)*

Avec un environnement de développement "intégré", on sera amené à utiliser des menus pour commander ces deux étapes. Le lancement de l'exécution créera une fenêtre console qui ressemblera à ceci (ici, nous avons employé le produit Eclipse 3.1) :



Exemple d'exécution du programme PremProg *(2)*

Précautions

Voici quelques indications concernant quelques problèmes que vous pouvez rencontrer.

1 Certains environnements intégrés peuvent générer plus ou moins automatiquement du code, ou tout au moins un squelette à compléter. Si vous exploitez ces possibilités, vous risquez de rencontrer des instructions dont nous n'avons pas encore parlé. Dans ce cas, le plus simple est de les éliminer. Cela concerne tout particulièrement une instruction d'attribution de classe à un "paquetage", de la forme :

```
package xxxxx ;
```

La conserver pourrait vous imposer des contraintes sur le répertoire (dossier) contenant votre fichier source.

2 Prenez bien soin de respecter la casse (majuscules/minuscules) dans les noms de fichier. Une erreur de casse abouti au même comportement qu'une erreur de nom. Attention : le comportement de Windows peut être déroutant. En effet, supposons que vous ayez d'abord enregistré votre programme dans un fichier *Premprog* (au lieu de *PremProg*). Après avoir découvert votre erreur, vous chercherez probablement à créer un nouveau fichier (par une commande du type *Save as*) avec le bon nom *PremProg*. Dans ce cas, Windows vous signalera que ce fichier existe déjà. En fait, si vous demandez à le remplacer, il prendra bien en compte le nouveau nom. Autrement dit, tout se passe comme si la casse n'était pas significative pour Windows, mais il l'utilise quand même dans le nom effectivement attribué au fichier.

3 Si vous transférez des fichiers d'un environnement à un autre, il se peut qu'en cours de route, vous passiez de noms de fichiers longs (nombre de caractères quelconque, nombre d'extensions quelconque et de longueur quelconque[1]) à des noms de fichiers courts (8 caractères maximum et une seule extension de 3 caractères maximum). Dans ce cas, vous perdrez obligatoirement l'extension *java*. Il vous faudra penser à la restaurer avant compilation.

4 Certains environnements intégrés ferment automatiquement la fenêtre console lorsque le programme a fini de s'exécuter. Dans ce cas, le programme précédent laissera peu de traces de son passage. Vous pourrez vous arranger pour qu'il ne s'arrête pas tout de suite, en lui ajoutant une instruction de lecture au clavier, comme vous apprendrez à le faire au paragraphe 4.

Informations complémentaires

Pour l'instant, nous pouvons nous permettre de confondre la notion de programme avec celle de classe. Plus tard, vous verrez que lorsque vous demandez à la machine virtuelle d'exécuter un fichier *xxxx.class*, elle y recherche une fonction publique de nom *main*. Si elle la trouve, elle l'exécute ; dans le cas contraire, elle indique une erreur.

3 Quelques instructions de base

L'exemple du paragraphe 1 a permis de présenter le canevas général à utiliser pour écrire un programme en Java. Voici maintenant un exemple un peu plus important, accompagné de ce que son exécution afficherait dans la fenêtre console :

1. Malgré tout, le nombre total de caractères ne doit pas excéder 255.

```
public class Exemple
{
  public static void main (String [] args)
  {
    int n ;
    double x ;
    n = 5 ;
    x = 2*n + 1.5 ;

    System.out.println ("n = " + n) ;
    System.out.println ("x = " + x) ;
    double y ;
    y = n * x + 12 ;
    System.out.println ("valeur de y : " + y) ;
  }
}
```

```
n = 5
x = 11.5
valeur de y : 69.5
```

Exemple de programme Java

Bien entendu, nous avons utilisé le même canevas que précédemment avec un autre nom de classe (ici *Exemple*) :

```
public class Exemple
{ public static void main (String[] args)
  {  .....
  }
}
```

Les deux premières instructions de notre fonction *main* sont des déclarations classiques :

```
int n ;
double x ;
```

La première précise que la variable *n* est de type *int*, c'est-à-dire qu'elle est destinée à contenir des nombres entiers (relatifs). Comme la plupart des langages, Java dispose de plusieurs types entiers. De la même manière, la seconde instruction précise que *x* est une variable de type *double*, c'est-à-dire destinée à contenir des nombres flottants en "double précision" (approximation de nombres réels). Nous verrons que Java dispose de deux types de flottants, le second se nommant *float* (nous ne l'avons pas utilisé ici car il aurait fait intervenir des problèmes de conversion des constantes flottantes).

Comme dans la plupart des langages modernes, les déclarations sont obligatoires en Java. Cependant, il n'est pas nécessaire qu'elles soient regroupées en début de programme (comme cela est le cas en C ou en Pascal) ; il suffit simplement qu'une variable ait été déclarée avant d'être utilisée.

Les instructions suivantes sont des affectations classiques :

```
n = 5 ;
x = 2*n + 1.5 ;
```

Les deux instructions suivantes font appel à la fonction *System.out.println* déjà entrevue au paragraphe 1 :

```
System.out.println ("n = " + n) ;
System.out.println ("x = " + x) ;
```

Mais cette fois, vous constatez que son argument ne se limite plus à une simple constante chaîne. En Java, l'expression *"n = "* + *n* est interprétée comme la *concaténation* de la chaîne constante *"n = "* avec le résultat de la conversion en chaîne de la valeur de la variable *n*. Une telle conversion fournit en fait la suite de caractères correspondant à l'écriture du nombre en décimal.

La même remarque s'applique à l'expression *"x = "* + *x*. Nous verrons que l'opérateur + possède une propriété intéressante : dès que l'un de ses deux opérandes est de type chaîne, l'autre est converti en chaîne.

La suite du programme est classique. On y note simplement une déclaration (tardive) de la variable *y*. Elle est autorisée à ce niveau car *y* n'a pas été utilisée dans les instructions précédentes.

Remarques

1 Aucun objet n'apparaît dans ce programme. Les variables *n*, *x* et *y* sont analogues aux variables qu'on rencontre dans les autres langages. En fait, seule la présence artificielle de la classe *Exemple* distingue ce programme d'un programme C.

2 Si, connaissant le C, vous essayez de remplacer les déclarations de type *double* par des déclarations de type *float*, vous serez certainement surpris de découvrir une erreur de compilation. Cela provient d'une part de ce que les constantes flottantes sont implicitement de type *double*, d'autre part de ce que Java refuse la conversion implicite de *double* en *float*.

3 En Java, il n'est pas aussi facile que dans les autres langages d'agir sur la manière dont les nombres sont convertis en chaînes, donc affichés (gabarit, nombre de chiffres significatifs, notation exponentielle ou flottante...). Bien entendu, il reste toujours possible de développer des outils dans ce sens.

4 Avec une instruction telle que :

```
System.out.println ("resultats = ", a + b*x) ;
```

on affichera à la suite du texte "resultats = ", la valeur de *a* suivie de celle de *b*x*. Pour obtenir celle de l'expression *a* + *b*x*, il faudra procéder ainsi :

```
System.out.println ("resultats = ", (a + b*x) ) ;
```

4 Lecture d'informations au clavier

Java est avant tout destiné à développer des applications ou des applets utilisant des interfaces graphiques. Mais comme nous l'avons déjà signalé, la programmation des interfaces graphiques nécessite de nombreuses connaissances, y compris celles relatives à la programmation orientée objet. Pour faciliter l'apprentissage du langage, il est de loin préférable de commencer par réaliser des programmes travaillant en mode console.

4.1 Présentation d'une classe de lecture au clavier

Comme nous l'avons vu précédemment, l'affichage dans la fenêtre console ne présente pas de difficultés puisqu'il suffit de recourir à l'une des fonctions *System.out.println* ou *System.out.print*. Malheureusement, Java ne prévoit rien de comparable pour la lecture au clavier.

En fait, il est toujours possible de développer une petite classe offrant les services de base que sont la lecture d'un entier, d'un flottant ou d'un caractère. Vous trouverez une telle classe sous le nom *Clavier.java* parmi les fichiers source disponibles en téléchargement sur le site www.editions-eyrolles.com, ainsi que sa liste complète en annexe B. Il n'est pas nécessaire de chercher à en comprendre le fonctionnement pour l'instant. Il vous suffit de savoir qu'elle contient des fonctions[1] de lecture au clavier, parmi lesquelles :

- *Clavier.lireInt()* fournit en résultat une valeur entière lue au clavier,
- *Clavier.lireDouble()* fournit en résultat une valeur de type *double* lue au clavier.

Ainsi, voici comment nous pourrions demander à l'utilisateur de fournir un nombre entier qu'on place dans la variable *nb* :

```
int nb ;
   .....
System.out.print ("donnez un nombre entier : ") ;
nb = Clavier.lireInt() ;   // () obligatoires pour une fonction sans arguments
```

Nous utiliserons les possibilités de cette classe *Clavier* dans notre prochain exemple de programme, au paragraphe 4.

Notez que nous nous sommes limités à la lecture d'une seule valeur par ligne. D'autre part, si l'utilisateur fournit une réponse incorrecte (par exemple 45é ou 3.5 pour un *int*, ou encore 4.25.3 ou 2.3à2 pour un *double*), nous avons prévu que le programme s'interrompe avec le message : **** Erreur de donnee ****.

1. Ici encore, nous parlons de fonctions, alors qu'il s'agit en réalité de méthodes statiques de la classe *Clavier*.

4.2 Utilisation de cette classe

Pour pouvoir utiliser cette classe *Clavier* au sein d'un de vos programmes, vous disposez de plusieurs solutions. Pendant la phase d'apprentissage du langage, la démarche la plus simple consiste à :

- recopier le fichier source *Clavier.java* dans le même répertoire que celui où se trouve le programme l'utilisant,
- compiler une seule fois ce fichier.

Par la suite, la classe *Clavier.class* sera automatiquement utilisée dès que vous compilerez une autre classe y faisant appel.

Avec certains environnements intégrés, vous aurez peut-être besoin de mentionner cette classe *Clavier.java* au sein d'un fichier projet. En revanche, il ne sera plus nécessaire qu'elle figure dans le même répertoire que le programme l'utilisant.

Remarque

Comme vous le verrez dans le chapitre relatif aux classes, vous pourrez également utiliser la classe *Clavier* en la collant à la suite de votre fichier source, de manière à obtenir deux classes dans un même fichier. Dans ce cas, toutefois, il vous faudra supprimer le mot-clé *public* de la ligne *public Class Clavier.*

4.3 Boucles et choix

Voici maintenant un exemple de programme comportant, en plus des instructions de base déjà rencontrées, une structure de choix et une structure de boucle. Il calcule les racines carrées de 5 valeurs fournies en données. Les lectures au clavier sont réalisées en utilisant la fonction *Clavier.lireInt()* de la classe *Clavier* dont nous avons parlé précédemment.

```
// Calcul de racines carrees
// La classe Racines utilise la classe Clavier
public class Racines
{ public static void main (String[] args)
  { final int NFOIS = 5 ;
    int i ;
    double x ;
    double racx ;

    System.out.println ("Bonjour") ;
    System.out.println ("Je vais vous calculer " + NFOIS + " racines carrees") ;

    for (i=0 ; i<NFOIS ; i++)
      { System.out.print ("Donnez un nombre : ") ;
        x  = Clavier.lireDouble ()  ;
```

```
        if (x < 0.0)
          System.out.println (x + " ne possede pas de racine carree") ;
         else
          { racx = Math.sqrt(x) ;
            System.out.println (x + " a pour racine carree : "  + racx) ;
          }
      }
    System.out.println ("Travail termine - Au revoir") ;
  }
}
```

```
Je vais vous calculer 5 racines carrees
Donnez un nombre : 16
16.0 a pour racine carree : 4.0
Donnez un nombre : 2
2.0 a pour racine carree : 1.4142135623730951
Donnez un nombre : -9
-9.0 ne possede pas de racine carree
Donnez un nombre : 5.25
5.25 a pour racine carree : 2.29128784747792
Donnez un nombre : 2.25
2.25 a pour racine carree : 1.5
Travail termine - Au revoir
```

Exemple d'un programme de calcul de racines carrées

Les deux premières lignes commencent par // ; ce sont des commentaires.

La première instruction de la fonction *main* est une déclaration particulière :

```
final int NFOIS = 5 ;
```

Elle comporte une initialisation dont le rôle est intuitif : placer la valeur 5 dans *NFOIS*, avant le début de l'exécution. Quant au mot-clé *final*, il précise simplement que la valeur de la variable correspondante ne peut pas être modifiée au cours de l'exécution. En définitive, *NFOIS* est l'équivalent de ce qu'on appelle une constante symbolique dans certains langages.

Les autres déclarations ne posent pas de problème, pas plus que les deux appels de *System.out.println*. Notez simplement la concaténation de trois chaînes dans le deuxième de ces appels (*"Je vais vous calculer " + NFOIS + " racines carrees"*).

Pour faire une répétition : l'instruction *for*

En Java comme dans la plupart des langages, il existe plusieurs façons d'effectuer une répétition. Ici, nous avons utilisé l'instruction *for* que les connaisseurs du C n'auront aucun mal à interpréter :

```
for (i=0 ; i<NFOIS ; i++)
```

Son rôle est de répéter le bloc (délimité par des accolades { et }) figurant à sa suite, en respectant les consignes suivantes :

- avant de commencer cette répétition, réaliser :

```
i = 0
```

- avant chaque nouvelle exécution du bloc (tour de boucle), examiner la condition :

```
i<NFOIS
```

  Si elle est satisfaite, exécuter le bloc indiqué, sinon passer à l'instruction suivant ce bloc ;

- à la fin de chaque exécution du bloc, réaliser :

```
i++
```

  Comme C, Java dispose d'une notation d'incrémentation. Ici, *i++* est équivalente à *i = i+1*.

Pour faire des choix : l'instruction *if*

Les lignes :

```
if (x < 0.0)
  System.out.println (x + " ne possede pas de racine carree") ;
 else
  { racx = Math.sqrt(x) ;
    System.out.println (x + " a pour racine carree : "  + racx) ;
  }
```

constituent une instruction de choix basée sur la condition *x <0.0*. Si cette dernière est vraie, on exécute l'instruction suivante :

```
System.out.println (x + " ne possede pas de racine carree") ;
```

Si elle est fausse, on exécute l'instruction suivant le mot *else*, c'est-à-dire ici le bloc :

```
{ racx = Math.sqrt(x) ;
  System.out.println (x + " a pour racine carree : "  + racx) ;
 }
```

Notez l'appel de la fonction[1] *Math.sqrt* qui fournit une valeur de type *double*, correspondant à la racine carrée de la valeur (de type *double*) qu'on lui fournit en argument.

Les différentes sortes d'instructions

Comme les autres langages, Java distingue les instructions de déclaration (fournissant des informations au compilateur pour qu'il mène à bien sa traduction) et les instructions exécutables (dont la traduction fournit des instructions en code machine, ou plutôt ici en *byte codes*). Cependant, nous verrons que la liberté offerte dans l'emplacement des déclarations conduit à les rendre partiellement exécutables.

Quant aux (vraies) instructions exécutables, nous verrons qu'on peut les classer selon trois catégories :

1. Ici encore, il s'agit en fait d'une méthode statique de la classe *Math*.

- les *instructions simples*, obligatoirement terminées par un point-virgule[1],
- les *instructions de structuration* telles que *if* ou *for*,
- des *blocs*, délimités par { et }.

Les deux dernières ont une définition "récursive" puisqu'elles peuvent contenir, à leur tour, n'importe laquelle des trois formes.

Lorsque nous parlerons d'instruction sans précisions supplémentaires, il pourra s'agir de n'importe laquelle des trois formes ci-dessus.

En C++

La syntaxe de Java est fondée sur celle de C++, ce qui fait que le précédent programme est facilement compréhensible pour un programmeur C ou C++. Il existe cependant quelques petites différences. Ici, déjà, vous constatez que le mot-clé *const* a été remplacé par *final*.

Remarque

Rappelons que certains environnements ferment automatiquement la fenêtre console à la fin de l'exécution du programme. Pour continuer à la voir, il vous suffit d'ajouter, avant la fin de votre programme, une instruction :

```
Clavier.lireInt() ;
```

Votre programme ne s'interrompra pas tant que vous n'aurez pas fourni une valeur entière (quelconque).

5 Règles générales d'écriture

Ce paragraphe expose un certain nombre de règles générales intervenant dans l'écriture d'un programme Java. Nous aborderons en particulier ce qu'on nomme les *identificateurs* et les *mots-clés*, le *format libre* dans lequel sont écrites les instructions, l'usage des *séparateurs* et des *commentaires*. Enfin, nous vous dirons un mot d'*Unicode*, le code universel utilisé par Java pour représenter les caractères.

5.1 Les identificateurs

Dans un langage de programmation, un *identificateur* est une suite de caractères servant à désigner les différentes entités manipulées par un programme : variables, fonctions, classes, objets...

1. Notez la différence avec certains langages comme Pascal, dans lequel le point-virgule est un séparateur d'instructions.

En Java comme dans la plupart des autres langages, un identificateur est formé de lettres ou de chiffres, le premier caractère étant obligatoirement une lettre. Les lettres comprennent les majuscules A-Z et les minuscules a-z, ainsi que le caractère "souligné" (_) et le caractère $[1]. Voici quelques identificateurs corrects :

```
ligne    Clavier    valeur_5    _total    _56
```

Aucune limitation ne pèse sur le nombre de caractères, qui sont tous significatifs (en C, seuls les 32 premiers l'étaient).

Notez bien que, comme en C (et contrairement à Pascal), on distingue les majuscules des minuscules. Ainsi, *Ligne* et *ligne* désignent deux identificateurs différents ; il en va de même pour *PremProg* et *Premprog*.

Informations complémentaires

Bien qu'elles ne soient nullement imposées par le langage, certaines règles sont traditionnellement utilisées dans le choix des identificateurs d'un programme Java. Ainsi, les noms de variables et les noms de fonctions sont écrits en minuscules, sauf s'ils sont formés de la juxtaposition de plusieurs mots, auquel cas chaque mot sauf le premier comporte une majuscule, par exemple : *valeur*, *nombreValeurs*, *tauxEmprunt, nombreReponsesExactes*. Les noms de classes suivent la même règle, mais leur première lettre est écrite en majuscules : *Clavier*, *PremProg*. Les noms de constantes sont écrits entièrement en majuscules. Notez que ces règles permettent de savoir que *System* est une classe et que *out* n'en est pas une.

5.2 Les mots-clés

Certains mots-clés sont réservés par le langage à un usage bien défini et ne peuvent pas être utilisés comme identificateurs. En voici la liste, par ordre alphabétique :

abstract	*assert*	*boolean*	*break*	*byte*
case	*catch*	*char*	*class*	*const*
continue	*default*	*do*	*double*	*else*
extends	*final*	*finally*	*float*	*for*
goto	*if*	*implements*	*import*	*instanceof*
int	*interface*	*long*	*native*	*new*
null	*package*	*private*	*protected*	*public*
return	*short*	*static*	*super*	*switch*
synchronized	*this*	*throw*	*throws*	*transient*
try	*void*	*volatile*	*while*	

1. Il est conseillé d'éviter de l'employer car il est utilisé par Java dans ses mécanismes internes.

5.3 Les séparateurs

Dans notre langue écrite, les mots sont séparés par un espace, un signe de ponctuation ou une fin de ligne.

Il en va quasiment de même en Java. Ainsi, dans un programme, deux identificateurs successifs entre lesquels la syntaxe n'impose aucun signe particulier[1] doivent impérativement être séparés soit par un espace, soit par une fin de ligne. En revanche, dès que la syntaxe impose un séparateur quelconque, il n'est pas nécessaire de prévoir d'espaces supplémentaires, bien qu'en pratique cela améliore la lisibilité du programme.

Ainsi, vous ne pourrez pas remplacer :

```
int x,y
```

par

```
intx,y
```

En revanche, vous pourrez écrire indifféremment :

```
int n,compte,p,total,valeur ;
```

ou, plus lisiblement :

```
int n, compte, p, total, valeur ;
```

voire :

```
int n,
    compte,
    p,
    total,
    valeur ;
```

5.4 Le format libre

Comme Pascal ou C, Java autorise une mise en page parfaitement libre. En particulier, une instruction peut s'étendre sur un nombre quelconque de lignes, et une même ligne peut comporter autant d'instructions que voulu. Les fins de ligne ne jouent pas de rôle particulier, si ce n'est celui de séparateur, au même titre qu'un espace[2] (un identificateur ne pouvant être coupé en deux par une fin de ligne.

Bien entendu, cette liberté de mise en page possède des contreparties. Si l'on n'y prend gare, le risque existe d'aboutir à des programmes peu lisibles. À simple titre d'exemple, voici comment pourraît être (mal) présenté le programme du paragraphe 4 :

---

1. Comme : : , = ; * ( ) [ ] { } + - / < > & |.

2. Sauf dans les constantes chaînes telles que "Bonjour monsieur", où les fins de ligne sont interdites. De telles constantes doivent impérativement être écrites sur une même ligne.

```
// Calcul de racines carrees
// La classe Racines utilise la classe Clavier
public class Racines1  { public
static void main (String[]
args)  { final int NFOIS = 5 ;  int i ; double
  x ;    double racx ;    System.out.println (
"Bonjour"
 ) ;  System.out.println ("Je vais vous calculer " +
  NFOIS  + " racines carrees") ;    for (i=0 ; i
 <NFOIS ; i++)  { System.out.print ("Donnez un nombre : ") ;  x  =
 Clavier.lireDouble  ()  ;  if (x < 0.0)  System.out.println (x +
  " ne possede pas de racine carree")
; else  { racx = Math.sqrt(x) ;  System.out.println (
x + " a pour racine carree : "  + racx) ; } }System.out.println
("Travail termine - Au revoir") ;   }}
```

Exemple de programme mal présenté

5.5 Les commentaires

Comme tout langage évolué, Java autorise la présence de commentaires dans les programmes source. Ces textes explicatifs destinés aux lecteurs du programme n'ont aucune incidence sur sa compilation.

Java dispose de deux formes de commentaires :

- les commentaires usuels,
- les commentaires de fin de ligne,

5.5.1 Les commentaires usuels

Ce sont ceux utilisés en langage C. Ils sont formés de caractères quelconques placés entre les caractères /* et */. Ils peuvent apparaître à tout endroit du programme où un espace est autorisé. En général, cependant, on se limitera à des emplacements propices à une bonne lisibilité du programme. En voici quelques exemples :

```
/* programme de calcul de racines carrees */

/* commentaire s'etendant
   sur plusieurs lignes
   de programme
*/

/* ============================================================ *
 *                UN TITRE MIS EN VALEUR                        *
 * ============================================================ */

int i ;         /* compteur de boucle */
float x;        /* nombre dont on cherche la racine */
float racx ;    /* pour la racine carre de x */
```

5.5.2 Les commentaires de fin de ligne

Ce sont ceux de C++. Ils sont introduits par le double caractère //. Tout le texte suivant jusqu'à la fin de la ligne est considéré comme un commentaire. Cette nouvelle possibilité n'apporte qu'un surcroît de confort et de sécurité. En effet, une ligne telle que :

```
System.out.println ("bonjour) ;  // formule de politesse
```

peut toujours être écrite ainsi :

```
System.out.println ("bonjour) ;  /* formule de politesse */
```

Vous pouvez mêler les deux formules. Notez que dans :

```
/* partie1 // partie2 */ partie3
```

le commentaire ouvert par /* ne se termine qu'au prochain */ ; donc *partie1* et *partie2* sont des commentaires, tandis que *partie3* est considéré comme appartenant aux instructions. De même, dans :

```
partie1 // partie2 /* partie3 */ partie4
```

le commentaire introduit par // s'étend jusqu'à la fin de la ligne. Il couvre donc *partie2*, *partie3* et *partie4*.

Remarque

Si l'on utilise systématiquement le commentaire de fin de ligne, on peut alors faire appel à /* et */ pour inhiber un ensemble d'instructions (contenant éventuellement des commentaires) en phase de mise au point.

Informations complémentaires

On dit souvent que Java possède une troisième forme de commentaires dits de documentation. Il s'agit en fait d'un cas particulier de commentaires usuels, puisqu'ils commencent par /** et qu'ils se terminent par */. Leur seul intérêt est de pouvoir être extraits automatiquement par des programmes utilitaires de création de documentation tels que Javadoc.

5.6 Emploi du code Unicode dans le programme source

La plupart des langages vous ont habitué à écrire un programme en recourant à un nombre de caractères relativement limité. La plupart du temps, vous disposez en effet des majuscules, des minuscules et des chiffres, mais pas nécessairement des caractères accentués. Ou bien vous pouvez saisir les caractères accentués dans une implémentation, mais ils apparaissent différemment dans une autre... Ces limitations proviennent de ce que, dans bon nombre de pays dont les pays occidentaux, les caractères sont codés sur un seul octet. En outre, le code utilisé n'est pas universel. Il s'agit le plus souvent d'une variante locale du code ASCII international à 7 bits (auquel n'appartiennent pas nos caractères accentués). Par exemple, Windows utilise le code ASCII/ANSI Latin 1.

Les concepteurs de Java ont cherché à atténuer ces limitations en utilisant le codage Unicode. Basé sur 2 octets, il offre 65536 combinaisons qui permettent de représenter la plupart des symboles utilisés dans le monde[1]. Cependant, pour l'instant, vous devez toujours saisir votre programme avec un éditeur générant des caractères codés sur un octet. C'est pourquoi Java a prévu que, avant d'effectuer sa traduction, le compilateur commence par traduire votre fichier source en Unicode.

Cette propriété aura des conséquences intéressantes dans le cas des constantes caractères que nous étudierons au prochain chapitre. Si, dans une implémentation donnée, vous parvenez à entrer un certain caractère (par exemple é), vous êtes certain qu'il sera représenté de façon portable dans les *byte codes* générés par le compilateur. Quant à votre programme source, il ne sera pas plus portable que celui d'un autre langage et sa traduction dans différents environnements pourra toujours conduire à certaines différences.

En fait, Java a prévu une notation permettant d'introduire directement dans le source un des 65536 caractères Unicode. Ainsi, n'importe où dans le source, vous pouvez utiliser la notation suivante, dans laquelle *xxxx* représente 4 chiffres hexadécimaux :

\uxxxx

Cette notation désigne simplement le caractère ayant comme code Unicode la valeur *xxxx*. Cette possibilité pourra s'avérer pratique pour introduire dans un programme une constante caractère n'existant pas dans l'implémentation où se fait la saisie, mais dont on sait qu'elle existe dans l'implémentation où le programme sera amené à s'exécuter.

Informations complémentaires

Au paragraphe 5.1, nous avons donné quelques règles concernant l'écriture des identificateurs et nous avons défini ce qu'on appelait une "lettre". En toute rigueur, de nombreux caractères Unicode sont des lettres. C'est notamment le cas de tous nos caractères accentués. S'ils existent dans l'implémentation où vous saisissez votre programme source, rien ne vous empêche de déclarer par exemple :

```
int têteListe ;      // correct
```

Vous pouvez aussi utiliser la notation *\uxxxx* dans un identificateur. Mais cette possibilité peu lisible n'a guère d'intérêt : n'oubliez pas, en effet, que les symboles utilisés pour écrire un identificateur disparaissent après compilation.

D'autre part, depuis sa version JDK 5.0, Java permet d'utiliser un codage Unicode élargi. Il faut alors distinguer le codage usuel dit BMP (Basic Multilingual Plan) qui utilise 16 bits (un *char*) d'une part, et le codage élargi qui quant à lui utilise 21 bits et nécessite un *int*.

1. Mais pas tous !

3

Les types primitifs de Java

Dans les chapitres précédents, nous avons rencontré les types *int* et *double*. Java dispose d'un certain nombre de types de base dits *primitifs*, permettant de manipuler des entiers, des flottants, des caractères et des booléens. Ce sont les seuls types du langage qui ne sont pas des classes. Mais ils seront utilisés pour définir les champs de données de toutes les classes que vous serez amenés à créer.

Avant de les étudier en détail, nous effectuerons un bref rappel concernant la manière dont l'information est représentée dans un ordinateur et la notion de type qui en découle.

1 La notion de type

La mémoire centrale est un ensemble de "positions binaires" nommées bits. Les bits sont regroupés en octets (8 bits), et chaque octet est repéré par ce qu'on nomme son adresse.

Compte tenu de sa technologie (actuelle !), l'ordinateur ne sait représenter et traiter que des informations exprimées sous forme binaire. Toute information, quelle que soit sa nature, devra être codée sous cette forme. Dans ces conditions, on voit qu'il ne suffit pas de connaître le contenu d'un emplacement de la mémoire (d'un ou de plusieurs octets) pour pouvoir lui attribuer une signification. Il faut également connaître la manière dont l'information qu'il contient a été codée. Il en va de même en général pour traiter cette information. Par exemple, pour additionner deux valeurs, il faudra savoir quel codage a été employé afin de pouvoir mettre en œuvre les bonnes instructions[1] en langage machine.

1. Par exemple, on ne fait pas appel aux mêmes circuits électroniques pour additionner deux nombres codés sous forme entière et deux nombres codés sous forme flottante.

D'une manière générale, la notion de type, telle qu'elle existe dans les langages évolués, sert (entre autres choses) à régler les problèmes que nous venons d'évoquer.

Les types primitifs de Java se répartissent en quatre grandes catégories selon la nature des informations qu'ils permettent de représenter :

- nombres entiers,
- nombres flottants,
- caractères,
- booléens.

Remarque

Le JDK 5.0 a introduit la notion de type enuméré que nous étudierons au chapitre 10. Nous verrons qu'il repose sur la notion de classe et qu'il ne constitue donc pas un type primitif.

2 Les types entiers

Ils servent à représenter des nombres entiers relatifs.

2.1 Représentation mémoire

Un bit est réservé au signe (0 pour positif et 1 pour négatif). Les autres servent à représenter :

- la valeur absolue du nombre pour les positifs,
- ce que l'on nomme le complément à deux du nombre, pour les négatifs.

Examinons cela plus en détail, dans le cas de nombres représentés sur 16 bits (ce qui correspondra finalement au type *short* de Java).

2.1.1 Cas d'un nombre positif

Sa valeur absolue est donc écrite en base 2, à la suite du bit de signe. Voici quelques exemples de codages de nombres. À gauche, le nombre en décimal ; au centre, le codage binaire correspondant ; à droite, le même codage exprimé en hexadécimal :

```
    1          0000000000000001          0001
    2          0000000000000010          0002
    3          0000000000000011          0003
   16          0000000000010000          00F0
  127          0000000001111111          007F
  255          0000000011111111          00FF
```

2.1.2 Cas d'un nombre négatif

Sa valeur absolue est codée suivant ce que l'on nomme la technique du *complément à deux*. Pour ce faire, cette valeur est d'abord exprimée en base 2, puis tous les bits sont inversés (1 devient 0 et 0 devient 1) ; enfin, on ajoute une unité au résultat. Voici quelques exemples (avec la même présentation que précédemment) :

```
  -1          1111111111111111          FFFF
  -2          1111111111111110          FFFE
  -3          1111111111111101          FFFD
  -4          1111111111111100          FFFC
 -16          1111111111110000          FFF0
-256          1111111100000000          FF00
```

Remarques

1 Le nombre 0 est codé d'une seule manière (0000000000000000).

2 Si l'on ajoute 1 au plus grand nombre positif (ici 0111111111111111, soit 7FFF en hexadécimal ou 32768 en décimal) et que l'on ne tient pas compte de la dernière retenue (ou, ce qui revient au même, si l'on ne considère que les 16 derniers bits du résultat), on obtient... le plus petit nombre négatif possible (ici 1111111111111111, soit FFFF en hexadécimal ou -32767 en décimal). C'est ce qui explique le phénomène de "modulo" bien connu de l'arithmétique entière, les dépassements de capacité n'étant jamais signalés, quel que soit le langage considéré.

2.2 Les différents types d'entiers

Java dispose de quatre types entiers, correspondant chacun à des emplacements mémoire de taille différente, donc à des limitations différentes. Le tableau suivant en récapitule leurs caractéristiques. Notez l'existence de constantes prédéfinies de la forme *Integer.MAX_VALUE* qui fournissent les différentes limites.

Type	Taille (octets)	Valeur minimale	Valeur maximale
byte	1	-128 (Byte.MIN_VALUE)	127 (Byte.MAX_VALUE)
short	2	-32 768 (Short.MIN_VALUE)	32 767 (Short.MAX_VALUE)
int	4	-2 147 483 648 (Integer.MIN_VALUE)	2 147 483 647 (Integer.MAX_VALUE)
long	8	-9 223 372 036 854 775 808 (Long.MIN_VALUE)	9 223 372 036 854 775 807 (Long.MAX_VALUE)

Les caractéristiques des quatre types entiers de Java

En C++

On trouve trois types entiers *short*, *int*, *long*. Mais contrairement à ce qui se passe en Java, leur taille n'est pas entièrement définie par le langage et peut donc varier d'une implémentation à une autre. Par exemple, le type *int* peut être codé sur 2 octets sur une machine, 4 octets sur une autre... Le type *byte* n'existe pas en C++ (mais on peut utiliser le type *signed char*).

2.3 Notation des constantes entières

La façon la plus naturelle d'introduire une constante entière dans un programme est de l'écrire sous forme décimale usuelle, avec ou sans signe, comme dans ces exemples :

```
+5478   57    -278
```

Mais on peut aussi utiliser une notation hexadécimale ou octale. Pour la notation hexadécimale, on utilise les 10 chiffres (0 à 9), ainsi que les 6 lettres a, b, c, d, e et f (en majuscules ou en minuscules) ; on fait précéder la valeur de 0x ou 0X. Par exemple :

0x1A correspond à la valeur décimale 26 (16+10).

Pour la notation octale, on fait précéder du chiffre 0 la valeur exprimée en base 8. Par exemple :

014 correspond à la valeur décimale 12,
037 correspond à la valeur décimale 31.

Une constante entière est de l'un des types *int* ou *long*, suivant sa valeur. En principe, on peut penser que cet aspect a peu d'importance. Nous verrons cependant au chapitre 4 qu'il aura une légère incidence sur la validité de certaines expressions.

On peut forcer une constante à être du type *long* en faisant suivre sa valeur de la lettre *l* ou *L*, comme dans 25L (25 serait de type *int*).

Le compilateur rejetera toute constante ayant une valeur supérieure à la capacité du type *long* (dans certains autres langages, on pouvait obtenir une valeur erronée, sans en être prévenu).

3 Les types flottants

3.1 Les différents types et leur représentation en mémoire

Les types flottants permettent de représenter, de manière approchée, une partie des nombres réels. Pour ce faire, ils s'inspirent de la notation dite scientifique (ou exponentielle) bien connue qu'on trouve dans $1.5\ 10^{22}$ ou $0.472\ 10^{-8}$; dans une telle notation, on nomme *mantisses* les quantités telles que 1.5 ou 0.472 et *exposants* les quantités telles que 22 ou -8.

Plus précisément, un nombre réel sera représenté en flottant en déterminant un signe *s* (+1 ou -1) et deux quantités M (mantisse) et E (exposant) telles que la valeur

$$s\ M\ .\ 2^E$$

représente une approximation de ce nombre.

Java prévoit deux types de flottants correspondant chacun à des emplacements mémoire de tailles différentes : *float* et *double*. La connaissance des caractéristiques exactes du système de codage n'est généralement pas indispensable[1]. En revanche, il est important de noter que de telles représentations sont caractérisées par deux éléments :

- *la précision* : lors du codage d'un nombre décimal quelconque dans un type flottant, il est nécessaire de ne conserver qu'un nombre fini de bits. Or la plupart des nombres s'exprimant avec un nombre limité de décimales ne peuvent pas s'exprimer de façon exacte dans un tel codage. On est donc obligé de se limiter à une représentation approchée en faisant ce qu'on nomme une erreur de troncature.
- *le domaine couvert*, c'est-à-dire l'ensemble des nombres représentables à l'erreur de troncature près.

Le tableau suivant récapitule les caractéristiques des types *float* et *double*. Notez ici encore l'existence de constantes prédéfinies de la forme *Float.MAX_VALUE* qui fournissent les différentes limites.

Type	Taille (octets)	Précision (chiffres significatifs)	Valeur absolue minimale	Valeur absolue maximale
float	4	7	1.40239846E-45 (Float.MIN_VALUE)	3.40282347E38 (Float.MAX_VALUE)
double	8	15	4.9406564584124654E-324 (Double.MIN_VALUE)	1.7976931348623l6E308 (Double.MAX_VALUE)

Les caractéristiques des types flottants en Java

Une estimation de l'erreur relative de troncature est fournie dans la colonne Précision sous la forme d'un nombre de chiffres significatifs. Il s'agit d'une approximation décimale plus parlante que son équivalent (exact) en nombre de bits significatifs.

En C++

On trouve trois types flottants *float*, *double* et *long double*. Ici encore, contrairement à ce qui se passe en Java, leur taille, donc a fortiori leurs caractéristiques dépendent de l'implémentation.

1. Sauf lorsque l'on doit faire une analyse fine des erreurs de calcul.

Remarque

Java impose l'utilisation des conventions IEEE 754 pour représenter les nombres flottants. Cela implique qu'il existe des motifs binaires particuliers permettant de représenter une valeur infinie (positive ou négative), ainsi qu'une valeur non représentable (comme le résultat de la division de 0 par 0). Nous y reviendrons au chapitre suivant. D'autre part, il existe deux représentations de zéro (0+ et 0-) mais ce point n'a aucune incidence sur les calculs ou sur les comparaisons ; en particulier, ces deux valeurs sont bien strictement égales à 0.f (*float*) ou 0. (*double*).

3.2 Notation des constantes flottantes

Comme dans la plupart des langages, les constantes flottantes peuvent s'écrire indifféremment suivant l'une des deux notations :

- décimale,
- exponentielle.

La notation décimale doit comporter obligatoirement un point (correspondant à notre virgule). La partie entière ou la partie décimale peut être omise (mais bien sûr, il ne faut pas omettre les deux en même temps !). En voici quelques exemples corrects :

```
12.43    -0.38     -.38     4.     .27
```

En revanche, la constante *47* serait considérée comme entière et non comme flottante. En pratique, ce fait aura peu d'importance, compte tenu des conversions automatiques mises en place par le compilateur (et dont nous parlerons dans le chapitre suivant).

La notation exponentielle utilise la lettre e (ou E) pour introduire un exposant entier (puissance de 10), avec ou sans signe. La mantisse peut être n'importe quel nombre décimal ou entier (le point peut être absent dès que l'on utilise un exposant). Voici quelques exemples corrects (les exemples d'une même ligne étant équivalents) :

```
   4.25E4          4.25e+4          42.5E3
54.27E-32        542.7E-33        5427e-34
   48e13            48.e13          48.0E13
```

Par défaut, toutes les constantes sont créées par le compilateur dans le type *double*. Cela implique qu'une simple affectation telle que la suivante pose problème :

```
float x ;
   .....
x = 12.5 ;    // erreur de compilation : on ne peut pas convertir
              //   implicitement de double en float
```

Il est toutefois possible d'imposer à une constante flottante d'être du type *float*, en faisant suivre son écriture de la lettre F (ou f). Cela permet de gagner un peu de place en mémoire, en contrepartie d'une éventuelle perte de précision. On peut aussi régler le problème évoqué ci-dessus, en procédant ainsi :

```
float x ;
   .....
x = 12.5f ;    // cette fois, 12.5f est bien de type float
```

4 Le type caractère

4.1 Généralités

Comme la plupart des langages, Java permet de manipuler des caractères. Mais il offre l'originalité de les représenter en mémoire sur deux octets en utilisant le code universel Unicode dont nous avons parlé au chapitre précédent.

Parmi les 65536 combinaisons qu'offre Unicode, on retrouve les 128 possibilités du code ASCII restreint (7 bits) et même les 256 possibilités du code ASCII/ANSI (Latin 1) utilisé par Windows. Celles-ci s'obtiennent simplement en complétant le code ASCII par un un premier octet nul. Cela signifie donc qu'en Java comme dans les autres langages, la notion de caractère dépasse celle de caractère imprimable, c'est-à-dire auquel on peut associer un graphisme.

Une variable de type caractère se déclare en utilisant le mot-clé *char* comme dans :

```
char c1, c2 ;    // c1 et c2 sont deux variables de type caractère
```

4.2 Écriture des constantes de type caractère

On peut noter une constante caractère de façon classique entre apostrophes, par exemple :

```
'a'    'E'    'é'    '+'
```

Bien entendu, cette notation n'est utilisable que si le caractère est disponible dans l'implémentation considérée, qu'il dispose d'un graphisme et d'une combinaison de touche permettant de le saisir avec un éditeur[1].

1. Rappelons que tous les caractères de votre programme source (y compris ceux placés entre apostrophes) sont codés sur un octet lors de la saisie ; ce n'est qu'avant la compilation qu'ils sont convertis en Unicode.

Certains caractères ne disposant pas de graphisme possèdent une notation conventionnelle utilisant le caractère \[1] ; dans cette catégorie, on trouve également quelques caractères qui, bien que disposant d'un graphisme, jouent un rôle particulier de délimiteur qui leur interdit la notation classique. En voici la liste :

Notation	code Unicode (hexadécimal)	Abréviation usuelle	Signification
\b	0008	BS (Backspace)	Retour arrière
\t	0009	HT (Horizontal tabulation)	Tabulation horizontale
\n	000a	LF (Line feed)	Saut de ligne
\f	000c	FF (Form feed)	Saut de page
\r	000d	CR (Cariage return)	Retour chariot
\"	0022		
\'	0027		
\\	005c		

Les caractères disposant d'une notation spéciale

De plus, on peut définir une constante caractère en fournissant directement son code en hexadécimal sous la forme suivante, déjà rencontrée au paragraphe 5.6 du chapitre 2 :

\uxxxx

Rappelons en effet que vous saisissez votre programme source avec un éditeur local codant les caractères sur un octet. Le jeu de caractères d'une implémentation donnée est donc très limité. Le mérite de la notation précédente est de vous permettre d'entrer le code d'un caractère inconnu de votre éditeur. Bien entendu, cela ne préjuge nullement de l'usage qu'en fera le programme par la suite[2].

Remarques

1 Les constantes caractère ayant un code compris entre 0 et 255 peuvent être exprimées en octal sous la forme suivante, dans laquelle *ooo* représentent un nombre à trois chiffres en base 8 (0 à 7) :

```
'\ooo'
```

1. Ne pas confondre cette instruction avec celle employée pour introduire un caractère par son code Unicode.

2. Par exemple, si le programme affiche un tel caractère à l'écran, celui-ci sera à nouveau converti d'Unicode dans le codage local (sur un octet) de la machine d'exécution (qui n'est pas nécessairement celle où s'est faite la saisie). Si la conversion n'est pas possible, on obtiendra simplement l'affichage d'un point d'interrogation.

2 Dans un programme source, distinguez bien les caractères qui disparaîtront après compilation (comme ceux qui interviennent dans un identificateur) de ceux qui subsisteront (comme ceux qui apparaissent dans une constante caractère).

Informations complémentaires

Si vous souhaitez voir comment certains des 65536 caractères Unicode s'affichent dans votre implémentation, vous pouvez adapter le programme suivant. Ce dernier affiche tous les caractères dont le code est compris entre *codeDeb* et *codeFin*. Sachez que lorsqu'un caractère n'est pas connu de l'implémentation, il doit s'afficher sous la forme d'un point d'interrogation. Notez que ce programme fait intervenir des aspects qui ne seront exposés que plus tard (conversions de *char* en *int*, utilisation d'un entier pour initialiser une variable de type *char*).

```
public class TestUni
{  static void main (String[] args)
  { final char carDeb = 200, carFin = 300 ;
    char c ;
    for (c=carDeb ; c<carFin ; c++)
      { System.out.print ((int)c + "-") ;
        System.out.print (c + " ") ;
      }
  }
}
```

```
200-╩ 201-╦ 202-╠ 203-═ 204-╬ 205-╧ 206-╨ 207-╤ 208-╥ 209-╙ 210-╘ 211-╒ 212-╓ 21
3-╫ 214-╪ 215-┘ 216-┌ 217-█ 218-▄ 219-▌ 220-▐ 221-▀ 222-α 223-ß 224-Γ 225-π 226-
Σ 227-σ 228-µ 229-τ 230-Φ 231-Θ 232-Ω 233-δ 234-∞ 235-φ 236-ε 237-∩ 238-≡ 239-±
240-≥ 241-≤ 242-⌠ 243-⌡ 244-÷ 245-≈ 246-° 247-∙ 248-· 249-√ 250-ⁿ 251-² 252-■ 25
3- 254- 255- 256-? 257-? 258-? 259-? 260-? 261-? 262-? 263-? 264-? 265-? 266-
? 267-? 268-? 269-? 270-? 271-? 272-? 273-? 274-? 275-? 276-? 277-? 278-? 279-?
280-? 281-? 282-? 283-? 284-? 285-? 286-? 287-? 288-? 289-? 290-? 291-? 292-? 29
3-? 294-? 295-? 296-? 297-? 298-? 299-?
```

Affichage des caractères de codes donnés

En C++

C++ représente les caractères sur un seul octet en utilisant un code dépendant de l'implémentation. En outre, il existe une équivalence entre octet et caractère sur laquelle s'appuient bon nombre de programmes.

5 Le type booléen

Ce type sert à représenter une valeur logique du type *vrai/faux*. Il n'existe pas dans tous les langages car il n'est pas aussi indispensable que les autres. En effet, on peut souvent se contenter d'expressions booléennes du genre de celles qu'on utilise dans une instruction *if* :

```
if (n<p) .....    //  n<p est une expression booléenne valant vrai ou faux
```

En Java, on peut disposer de variables de ce type, ce qui permettra des affectations telles que :

```
boolean ordonne ;   // déclaration d'une variable de type booléen
   .....
ordonne = n<p ;     // ordonne reçoit la valeur de l'expression booléenne n<p
```

Les deux constantes du type booléen se notent *true* et *false*.

6 Initialisation et constantes

6.1 Initialisation d'une variable

Une variable peut recevoir une valeur initiale au moment de sa déclaration, comme dans :

```
int n = 15 ;
```

Cette instruction joue le même rôle que :

```
int n ;
n = 15 ;
```

Rien n'empêche que la valeur de *n* n'évolue par la suite.

Comme en Java les déclarations peuvent apparaître à n'importe quel emplacement du programme, on ne peut plus les distinguer complètement des instructions exécutables. En particulier, on peut initialiser une variable avec une expression autre qu'une constante. Voici un exemple :

```
int n = 10 ;
   .....
int p = 2*n      // OK : p reçoit la valeur 20 (si la valeur de n n'a pas
                 // changé depuis son initialisation)
```

Un autre exemple :

```
int n ;
   .....
n = Clavier.lireInt() ;
int p = 2 * n ;         // OK - la valeur initiale de p ne sera toutefois
                        // connue qu'au moment de l'exécution
```

6.2 Cas des variables non initialisées

En Java, une variable n'ayant pas encore reçu de valeur ne peut pas être utilisée, sous peine d'aboutir à une erreur de compilation. En voici deux exemples :

```
int n ;
System.out.println ("n = " + n ) ;   // erreur de compilation : la valeur
                                      // de n n'est pas définie ici


int n ;
int p = n+3 ; // erreur de compilation : la valeur de n n'est pas encore définie
n = 12 ;      // elle l'est ici (trop tard)
```

Contrairement à la plupart des autres langages, Java est capable de détecter toutes les situations de variables non initialisées, comme le montre cet exemple :

```
int n ;
if (...) n = 30 ;
p = 2*n ;   // erreur de compilation : n n'est pas initialisée dans tous les cas
```

Ici, le compilateur s'est aperçu de ce que la variable *n* pouvait ne pas recevoir de valeur dans certains cas. En revanche, les instructions suivantes seront acceptées :

```
int n ;
if (...) n = 30 ;
    else n = 10 ;
p = 2*n ;   // OK : n a obligatoirement reçu l'une des valeurs 30 ou 10
```

Informations complémentaires

Nous verrons que ce que nous nommons pour l'instant *variable* est en fait une variable locale à la méthode *main*. Nous rencontrerons d'autres sortes de variables, en particulier les champs des objets. Contrairement aux variables locales évoquées ici, ces champs seront soumis à une initialisation implicite par défaut ; le risque de champ non défini n'existera donc pas. D'autre part, nous verrons qu'il existe des variables ou des champs d'un type autre que primitif, à savoir objet ou tableau.

6.3 Constantes et expressions constantes

6.3.1 Le mot-clé final

Java permet de déclarer que la valeur d'une variable ne doit pas être modifiée pendant l'exécution du programme. Par exemple, avec :

```
final int n = 20 ;
```

on déclare la variable *n* de type *int*, de valeur initiale 20. De plus, toute tentative ultérieure de modification de la valeur de *n* sera rejetée par le compilateur :

```
n = n + 5 ;              // erreur : n a été déclarée final
n = Clavier.lireInt() ;  // idem
```

D'une manière générale, le mot-clé *final* peut être utilisé quelle que soit l'expression d'initialisation de la variable :

```
int p ;
   .....
p = Clavier.lireInt() ;
final int n = 2 * p ;     // OK, bien que la valeur de n ne soit connue
                          // qu'à l'exécution
   .....
n++ ;                     // erreur de oompilation : n est déclarée final
```

6.3.2 Notion d'expression constante

Les deux expressions utilisées pour initialiser les variables *n* de nos deux exemples précédents sont de nature différente. Dans le premier cas, il s'agit de la valeur 20, bien définie et connue dès la compilation. Dans le second cas, il s'agit d'une expression (*2*p*) dont la valeur, connue qu'à l'exécution, peut différer d'une exécution à l'autre.

Vous constatez que *final* ne fait pas la distinction. On ne peut donc pas dire que *final* sert à déclarer ce que l'on nomme des constantes symboliques dans certains autres langages. En fait, *final* demande simplement que la valeur d'une variable n'évolue plus après avoir été fixée (nous verrons qu'il n'est même pas nécessaire que cette première valeur soit définie dans la déclaration).

En revanche, nous rencontrerons des situations où il est néssaire de distinguer les deux sortes d'expressions. On parlera d'*expression constante* pour désigner une expression calculable par le compilateur.

Lorsque *final* est utilisé conjointement avec une expression constante, on retrouve la notion usuelle de définition de constante symbolique, autrement dit d'un symbole dont la valeur est parfaitement connue du compilateur. En Java, il est d'usage d'écrire les constantes symboliques en majuscules :

```
final int NOMBRE = 20 ;
   .....
final int LIMITE = 2 * NOMBRE + 3 ;
```

En C++

En C++, on peut déclarer des constantes symboliques à l'aide du mot-clé *const*. Les valeurs correspondantes doivent obligatoirement être des expressions constantes. En revanche, la protection contre la modification accidentelle de telles valeurs est beaucoup moins efficace qu'en Java.

6.3.3 L'initialisation d'une variable final peut être différée

En théorie, vous n'êtes pas obligé d'initialiser une variable déclarée *final* lors de sa déclaration. En effet, Java demande simplement qu'une variable déclarée *final* ne reçoive qu'une seule fois une valeur. Voyez ces exemples :

```
final int n ;              // OK, même si n n'a pas (encore) reçu de valeur
   .....
n = Clavier.lireInt() ;    // première affectation de n : OK
   .....
n++ ;                      // nouvelle affectation de n : erreur de compilation


final int n ;
   .....
if (...) n = 10 ;                  // OK
    else n = 20
```

Cependant, nous ne recommandons de recourir le moins possible à une telle démarche qui nuit fortement à la lisibilité des programmes et de toujours chercher à initialiser une variable le plus près possible de l'endroit où elle est définie.

4

Les opérateurs et les expressions

Java est l'un des langages les plus fournis en opérateurs. Il dispose en effet des opérateurs classiques (*arithmétiques*, *relationnels*, *logiques*) ou moins classiques (*manipulations de bits*), mais aussi d'un important éventail d'opérateurs originaux *d'affectation* et *d'incrémentation*. Nous verrons en effet que ces actions sont réalisées par des opérateurs.

Ce chapitre étudie tous les opérateurs de Java, ainsi que les règles de priorité et de conversion de type qui interviennent dans les évaluations des expressions.

1 Originalité des notions d'opérateur et d'expression

Dans la plupart des langages, on trouve, comme en Java :

- des *expressions* formées (entre autres) à l'aide d'opérateurs,
- des *instructions* pouvant éventuellement faire intervenir des expressions, comme l'instruction d'affectation :

```
y = a * x + b ;
```

Mais généralement, dans les langages autres que Java (ou C++), l'expression possède une valeur mais ne réalise aucune action, en particulier aucune affectation d'une valeur à une variable. Au contraire, l'affectation y réalise une affectation d'une valeur à une variable mais

ne possède pas de valeur. Les notions de valeur et d'action sont parfaitement disjointes. En Java, il en va différemment puisque :

- les (nouveaux) opérateurs d'incrémentation pourront non seulement intervenir au sein d'une expression (laquelle, au bout du compte, possédera une valeur), mais aussi agir sur le contenu de variables. Ainsi, l'expression (comme nous le verrons, il s'agit bien d'une expression en Java) :

  ```
  ++i
  ```

  réalisera une action, à savoir augmenter la valeur de *i* de 1 ; en même temps, elle aura une valeur, celle de *i* après incrémentation.

- une affectation apparemment classique telle que :

  ```
  i = 5
  ```

  pourra, à son tour, être considérée comme une expression (ici, de valeur 5). D'ailleurs, en Java, l'affectation (=) est un opérateur. Par exemple, la notation suivante :

  ```
  k = i = 5
  ```

  représente une expression (ce n'est pas encore une instruction – nous y reviendrons). Elle sera interprétée comme :

  ```
  k = (i = 5)
  ```

  Autrement dit, elle affectera à *i* la valeur 5 puis elle affectera à *k* la valeur de l'expression *i = 5*, c'est-à-dire 5.

En fait, en Java, les notions d'expression et d'instruction sont étroitement liées puisque la principale instruction de ce langage est... une expression terminée par un point-virgule. On la nomme souvent *instruction expression*. Voici des exemples de telles instructions qui reprennent les expressions prédédentes :

```
++i ;
i = 5 ;
k = i = 5 ;
```

Les deux premières ont l'allure d'une affectation telle qu'on la rencontre classiquement dans la plupart des autres langages. Notez que, dans les deux cas, il y a évaluation d'une expression (*++i* ou *i=5*) dont la valeur est finalement inutilisée. Dans le dernier cas, la valeur de l'expression *i=5*, c'est-à-dire 5, est à son tour affectée à *k* ; par contre, la valeur finale de l'expression complète est, là encore, inutilisée.

Remarque

Un appel de fonction (méthode) est aussi une expression. S'il est suivi d'un point-virgule, cela devient une instruction expression : si la méthode fournit une valeur, elle est alors ignorée. Examinons ces exemples :

```
Clavier.lireInt() ;    // lit une valeur au clavier, sans l'utiliser
Math.sqrt(5) ;         // OK mais sans intérêt
```

Lorsqu'une méthode ne fournit pas de valeur, on ne peut pas l'utiliser dans une expression, mais on peut toujours l'employer dans une instruction expression. C'est ce que l'on fait couramment dans :

```
System.out.println ("bonjour") ;
```

2 Les opérateurs arithmétiques

2.1 Présentation des opérateurs

Comme tous les langages, Java dispose d'opérateurs classiques *binaires* (c'est-à-dire portant sur deux opérandes), à savoir l'addition (+), la soustraction (-), la multiplication (*) et la division (/), ainsi que de deux opérateurs *unaires* (c'est-à-dire ne portant que sur un seul opérande) correspondant à l'opposé noté - (comme dans *-n* ou dans *-x+y*) et à l'identité noté + (comme dans *+a* ou dans *+(b-a)*).

Les opérateurs binaires ne sont a priori définis que pour deux opérandes ayant le même type[1] parmi *int*, *long*, *float* ou *double* et ils fournissent un résultat de même type que leurs opérandes. Mais nous verrons au paragraphe 3 que par le jeu des conversions implicites, le compilateur saura leur donner une signification lorsqu'ils porteront :

- soit sur des opérandes de types différents,
- soit sur des opérandes de type *byte*, *char* ou *short*.

De plus, il existe un opérateur de modulo noté % qui peut porter sur des entiers ou sur des flottants. Avec des entiers, il fournit le reste de la division entière de son premier opérande par son second. Par exemple :

11%4 vaut 3,

23%6 vaut 5,

-11%3 vaut -2,

-11%-3 vaut -2

Avec des flottants, il fonctionne de manière similaire[2] :

12.5%3.5 vaut environ 3,

-15.2%7.5 vaut environ -0.2.

---

1. En machine, il n'existe, par exemple, que des additions de deux entiers de même taille ou de flottants de même taille. Il n'existe pas d'addition d'un entier et d'un flottant ou de deux flottants de taille différente.

2. Mais peu de langages disposent d'un opérateur modulo pour les flottants.

Notez bien qu'en Java, le quotient de deux entiers fournit un entier. Ainsi 5/2 vaut 2 ; en revanche, le quotient de deux flottants (noté lui aussi /) est bien un flottant (5.0/2.0 vaut bien approximativement 2.5).

Remarque

Il n'existe pas d'opérateur d'élévation à la puissance. Il faut faire appel soit à des produits successifs pour des puissances entières pas trop grandes (par exemple, on calculera x^3 comme x*x*x), soit à la fonction *Math.pow*[1].

En C++

En C/C++, l'opérateur % n'est parfaitement défini que pour des entiers positifs ; il ne peut pas porter sur des flottants.

2.2 Les priorités relatives des opérateurs

Lorsque plusieurs opérateurs apparaissent dans une même expression, il est nécessaire de savoir dans quel ordre ils sont mis en jeu. En Java comme dans les autres langages, les règles sont celles de l'algèbre traditionnelle (du moins en ce qui concerne les opérateurs arithmétiques dont nous parlons ici).

Les opérateurs unaires + et - ont la priorité la plus élevée. On trouve ensuite, à un même niveau, les opérateurs *, / et %. Au dernier niveau, apparaissent les opérateurs binaires + et -.

En cas de priorités identiques, les calculs s'effectuent de gauche à droite. On dit que l'on a affaire à une *associativité de gauche à droite*[2].

Des parenthèses permettent d'outrepasser ces règles de priorité, en forçant le calcul préalable de l'expression qu'elles contiennent. Notez qu'elles peuvent également être employées pour assurer une meilleure lisibilité d'une expression.

Voici quelques exemples dans lesquels l'expression de droite, où ont été introduites des parenthèses superflues, montre dans quel ordre s'effectuent les calculs (les deux expressions proposées conduisent donc aux mêmes résultats) :

```
a + b * c                    a + ( b * c )
a * b + c % d                ( a * b ) + ( c % d )
- c % d                      ( - c ) % d
- a + c % d                  ( - a ) + ( c % d )
- a / - b + c                ( ( - a ) / ( - b ) ) + c
- a / - ( b + c )            ( - a ) / ( - ( b + c ) )
```

1. En toute rigueur, il s'agit de la fonction statique *pow* de la classe *Math* (de même que *System.println* désigne la fonction statique *println* de la classe *System*).

2. Nous verrons que quelques opérateurs (non arithmétiques) utilisent une associativité de droite à gauche.

2.3 Comportement en cas d'exception

Comme dans tous les langages, il existe des circonstances où un opérateur ne peut pas fournir un résultat correct, soit parce que le type utilisé ne permet pas de le représenter, soit parce que le calcul est tout simplement impossible. On parle dans ce cas de situation d'exception[1].

2.3.1 Cas des entiers

Comme nous l'avons déjà signalé, le dépassement de capacité n'est jamais détecté. On se contente de conserver les bits les moins significatifs du résultat. Nous en avons rencontré quelques exemples.

En revanche, la division par zéro (par / ou par %) conduit à une erreur d'exécution. Plus précisément, il y a déclenchement de ce que l'on nomme une *exception* de type *ArithmecticException*. Nous apprendrons au chapitre 10 qu'il est possible d'intercepter une telle exception. Si nous le faisons pas, nous aboutissons simplement à l'arrêt de l'exécution du programme, avec un message de ce type en fenêtre console :

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Test.main(Test.java:9)
```

2.3.2 Cas des flottants

En Java, aucune opération sur les flottants ne conduit à un arrêt de l'exécution (pas même une division par zéro !). En effet, comme nous l'avons dit, les flottants sont codés en respectant les conventions IEEE 754. Celles-ci imposent qu'il existe un motif particulier pour représenter l'infini positif, l'infini négatif et une valeur non calculable. En Java, ces valeurs peuvent même être affichées ou affectées directement à des variables. Examinons ce petit exemple :

```
public class IEEE
{ public static void main (String args[])
  { float x = 1e30f ;
    float y ;
    y = x*x ;
    System.out.println (x + " a pour carre : " + y) ;

    float zero = 0.f ;                    // division flottante par zero
    float z = y/zero ;
    System.out.println (y + " divise par 0 = " + z) ;
    y = 15 ;
    System.out.println (y + " divise par 0 = " + z) ;
```

1. Ce terme doit être distingué de celui utilisé dans la "gestion des exceptions", même si, dans certains cas, il y a bien un lien entre les deux notions.

```
    float x1 = Float.POSITIVE_INFINITY ;    // +infini
    float x2 = Float.NEGATIVE_INFINITY ;    // -infini
    z = x1/x2 ;
    System.out.println (x1 + "/" + x2 + " = " + z) ;
  }
}
```

```
1.0E30 a pour carre : Infinity
Infinity divise par 0 = Infinity
15.0 divise par 0 = Infinity
Infinity/-Infinity = NaN
```

Exemple d'exploitation des conventions IEEE 754

Comme vous le constatez, les trois motifs dont nous avons parlé provoquent l'affichage de *Infinity*, *-Infinity* et *NaN* (abréviation de *Not a Number*). Les constantes correspondantes se nomment *Float.POSITIVE_INFINITY* (*Double.POSITIVE_INFINITY* pour le type *double*), *Float.NEGATIVE_INFINITY* (*Double.NEGATIVE_INFINITY*) et *Float.NaN* (*Double.NaN*).

En C++

En C/C++, le comportement en cas d'exception dépend de l'implémentation.

3 Les conversions implicites dans les expressions

3.1 Notion d'expression mixte

Comme nous l'avons dit, les opérateurs arithmétiques ne sont définis que lorsque leurs deux opérandes sont de même type. Mais vous pouvez écrire des *expressions mixtes* dans lesquelles interviennent des opérandes de types différents. Voici un exemple d'expression correcte, dans laquelle *n* et *p* sont supposées être de type *int*, tandis que *x* est supposée être de type *float* :

```
n * x + p
```

Dans ce cas, le compilateur sait, compte tenu des règles de priorité, qu'il doit d'abord effectuer le produit *n*x*. Pour que ce soit possible, il va mettre en place des instructions[1] de conversion de la valeur de *n* dans le type *float* (car on considère que ce type *float* permet de

1. Attention, le compilateur ne peut que prévoir les instructions de conversion (qui seront donc exécutées en même temps que les autres instructions du programme) ; il ne peut pas effectuer lui-même la conversion d'une valeur que généralement il ne peut pas connaître.

représenter à peu près convenablement une valeur entière, l'inverse étant naturellement faux). Au bout du compte, la multiplication portera sur deux opérandes de type *float* et elle fournira un résultat de type *float*.

Pour l'addition, on se retrouve à nouveau en présence de deux opérandes de types différents (*float* et *int*). Le même mécanisme sera mis en place, et le résultat final sera de type *float*.

3.2 Les conversions d'ajustement de type

Une conversion telle que *int -> float* se nomme une conversion d'ajustement de type. Elle ne peut se faire que suivant une hiérarchie qui permet de ne pas dénaturer la valeur initiale[1], à savoir :

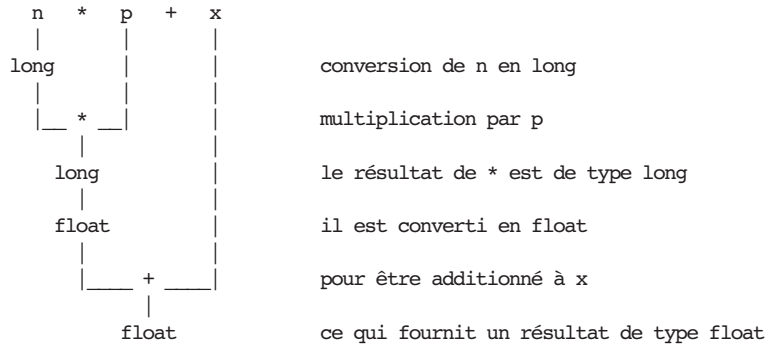
int -> long -> float -> double

On peut bien sûr convertir directement un *int* en *double* ; en revanche, on ne pourra pas convertir un *double* en *float* ou en *int*.

Notez que le choix des conversions à mettre en œuvre est effectué en considérant un à un les opérandes concernés et non pas l'expression de façon globale. Par exemple, si *n* est de type *int*, *p* de type *long* et *x* de type *float*, l'expression :

n * p + x

sera évaluée suivant ce schéma :



3.3 Les promotions numériques

Les conversions d'ajustement de type ne suffisent pas à régler tous les cas. En effet, comme nous l'avons déjà dit, les opérateurs numériques ne sont pas définis pour les types *byte*, *char* et *short*.

En fait, Java prévoit tout simplement que toute valeur de l'un de ces deux types apparaissant dans une expression est d'abord convertie en *int*, et cela sans considérer les types des éven-

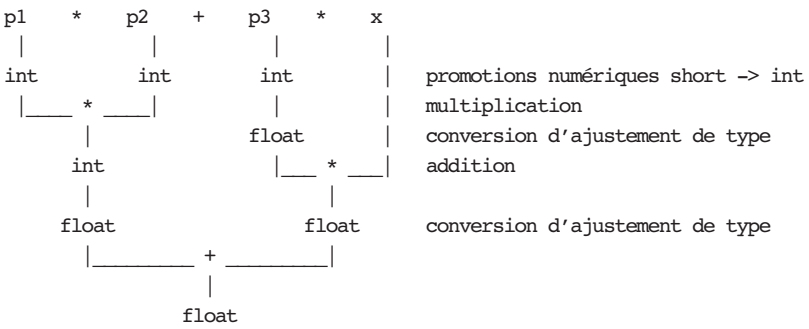
1. On dit parfois que de telles conversions respectent l'*intégrité des données*.

tuels autres opérandes. On parle alors de *promotions numériques* (ou encore de *conversions systématiques*).

Par exemple, si *p1*, *p2* et *p3* sont de type *short* et *x* de type *float*, l'expression :

```
p1 * p2 + p3 * x
```

est évaluée comme l'indique ce schéma :



Notez bien que les valeurs des trois variables de type *short* sont d'abord soumises à la promotion numérique *short -> int* ; ensuite, on applique les mêmes règles que précédemment.

3.4 Conséquences des règles de conversion

La mise en place de conversions automatiques conduit parfois à des situations inattendues, comme le montre ce petit exemple :

```
public class Conver
{ public static void main (String args[])
  { byte b1 = 50, b2 = 100 ;
    int n ;
    n = b1 * b2 ; // b1 et b2 sont convertis en  int, avant qu'on en fasse le
                  // produit (en int) ; le resultat aurait depasse la capacite
                  // du type byte, mais il est correct[1]
    System.out.println (b1 + "*" + b2 + " = " + n) ;
    int n1 = 100000, n2 = 200000 ;
    long p ;
    p = n1*n2 ;   // le produit est calcule en int, il conduit a un dépassement
                  // le resultat (errone) est affecte a p
    System.out.println (n1 + "*" + n2 + " = " + p) ;
  }
}
```

1. En revanche, on verra plus loin qu'un problème se poserait si l'on voulait affecter ce résultat à une variable de type *byte*.

```
50*100 = 5000
100000*200000 = -1474836480
```

Conséquences des règles de conversion

Si, dans la première affectation (*n = b1*b2*), l'expression *b1*b2* avait été calculée dans le type *byte*, sa valeur n'aurait pas été représentable. Mais, compte tenu des règles de promotions numériques, elle a été calculée dans le type *int*.

En revanche, dans la seconde affectation (*p = n1*n2*), l'expression *n1*n2* est calculée dans le type *int* (le type de *p* n'ayant aucune influence sur le calcul à ce niveau). Le résultat théorique dépasse la capacité du type *int* ; dans ce cas, comme dans la plupart des langages, Java n'en conserve que les bits les moins significatifs, ce qui fournit un résultat faux ; c'est ce dernier qui est ensuite converti en *long*.

3.5 Le cas du type *char*

En Java, une variable de type caractère peut intervenir dans une expression arithmétique car il est prévu une *promotion numérique* de *char* en *int*. A priori, vous pouvez vous interroger sur la signification d'une telle conversion. En fait, il ne s'agit que d'une question de point de vue. En effet, une valeur de type caractère peut être considérée de deux façons :

- comme le caractère concerné : a, Z, fin de ligne...,
- comme le code de ce caractère, c'est-à-dire un motif de 16 bits ; or à ce dernier on peut toujours faire correspondre un nombre entier, à savoir le nombre qui, codé en binaire, fournit le motif en question ; par exemple, en Java qui utilise Unicode, le caractère E est représenté par le motif binaire 00000000 01000101, auquel on peut faire correspondre le nombre 69.

Voici quelques exemples d'évaluation d'expressions, dans lesquels on suppose que *c1* et *c2* sont de type *char*, tandis que *n* est de type *int*.

Exemple 1

```
 c1   +    1
 |         |
int        |            promotion numérique char -> int
 |___ + ___|
      |
     int
```

L'expression *c1+1* fournit donc un résultat de type *int*, correspondant à la valeur du code du caractère contenu dans *c1* augmenté d'une unité.

Exemple 2

```
 c1   -    c2
 |         |
int       int          promotions numériques char -> int
 |___ _ ___|
      |
     int
```

Ici, bien que les deux opérandes soient de type *char*, il y a quand même conversion préalable de leurs valeurs en *int* (promotions numériques). Avec *c1 = 'E'* et *c2 = 'A'*, l'expression *c1-c2* vaudra 4[1].

Remarques

1 La conversion de *char* (16 bits) en *int* (32 bits) se fait en complétant le motif binaire initial par 16 bits à zéro. On ne tient pas compte d'un éventuel bit de signe (comme ce serait le cas en C). Autrement dit, les valeurs entières ainsi obtenues sont comprises dans l'intervalle 0 à 65 535 et non dans l'intervalle -32768 à 32767.

2 Ici, nous nous limitons à des expressions faisant intervenir des caractères, sans préjuger de l'usage qui en est fait. Or, autant on n'a aucune raison de vouloir employer *c1*c2* comme une valeur de type caractère, autant on sera tenté de le faire pour *c1+1*, ne serait-ce qu'en écrivant *c2 = c1+1*. Nous verrons plus loin qu'une telle affectation ne pourra se faire qu'en indiquant explicitement la conversion de *c1+1* en *char*, en écrivant :

```
c1 = (char(c1+1)) ;
```

4 Les opérateurs relationnels

4.1 Présentation générale

Comme tout langage, Java permet de comparer des expressions à l'aide d'opérateurs classiques de comparaison. En voici un exemple :

```
2 * a > b + 5
```

Le résultat est une valeur booléenne ayant l'une des deux valeurs *true* ou *false*. On pourra :

- l'utiliser dans une instruction *if*, comme dans :

```
if (2 * a > b + 5) ...
```

1. En Unicode, comme en ASCII, les codes des lettres majuscules sont consécutifs.

- l'affecter à une variable booléenne :

```
boolean ok ;
   .....
ok = 2 * a > b + 5
```

- le faire intervenir dans une expression booléenne plus complexe.

Comme les opérateurs arithmétiques, les opérateurs relationnels ne sont théoriquement définis que pour des opérandes de même type, parmi *int*, *long*, *float* ou *double*. Mais ils soumettent eux aussi leurs opérandes aux conversions implicites (promotions numériques et ajustement de type), de sorte qu'au bout du compte ils pourront porter sur des opérandes de types quelconques, y compris *byte*, *short* ou *char.*

Voici la liste des opérateurs relationnels existant en Java. Remarquez bien la notation (==) de l'opérateur d'égalité, le signe = étant réservé aux affectations :

Opérateur	Signification
<	inférieur à
<=	inférieur ou égal à
>	supérieur à
>=	supérieur ou égal à
==	égal à
!=	différent de

Les opérateurs relationnels

Il faut savoir que les quatre premiers opérateurs (<, <=, >, >=) sont de même priorité. Les deux derniers (== et !=) possèdent également la même priorité, mais celle-ci est inférieure à celle des précédents. D'autre part, ces opérateurs relationnels sont moins prioritaires que les opérateurs arithmétiques. Cela permet souvent d'éviter certaines parenthèses dans des expressions.

Ainsi :

x + y < a + 2

est équivalent à :

(x + y) < (a + 2)

C++ En C++

En C++, il n'existe pas d'expressions booléennes (mais on y trouve des variables booléennes !). C++ dispose des mêmes opérateurs de comparaison qu'en Java, mais ils fournissent un résultat de type entier (0 ou 1).

4.2 Cas particulier des valeurs Infinity et NaN

Nous avons vu au paragraphe 2.3.2 que Java représente les valeurs flottantes en respectant les conventions IEEE 754 qui imposent l'existence de motifs particuliers tels que *Float.POSITIVE_INFINITY*, *Float.NEGATIVE_INFINITY, Float.NaN*...

Comme on peut s'y attendre, les valeurs représentant l'infini sont comparables à n'importe quelles valeurs de type flottant. Par exemple, *Float.POSITIVE_INFINITY* est supérieure à toute valeur finie de type *float* ou *double* :

```
double x = 5e20 ;
if (x < Double.POSITIVE_INFINITY) ...   // vrai
if (x < Float.POSITIVE_INFINITY) ...    // vrai car Float.POSITIVE_INFINITY est
                // convertie en double, ce qui fournit Double.POSITIVE_INFINITY
```

En revanche, la comparaison de *NaN* avec une autre valeur fournit toujours un résultat faux. Par exemple, considérez :

```
float x ;
x = Float.NaN ;
y = 2 ;
if (x<y) ...   // faux
if (y<=x) ...   // faux
```

La relation $x<y$ est fausse, mais la relation $y<=x$ l'est aussi !

4.3 Cas des caractères

Compte tenu des règles de conversion, une comparaison peut porter sur deux caractères. Bien entendu, les comparaisons d'égalité ou d'inégalité ne posent pas de problème particulier. Par exemple, *c1 == c2* sera vrai si *c1* et *c2* ont la même valeur, c'est-à-dire si *c1* et *c2* contiennent des caractères de même code, donc si *c1* et *c2* contiennent le même caractère, De même, *c1 == 'e'* sera vrai si le code de *c1* est égal au code de *'e'*, donc si *c1* contient le caractère *e*.

En revanche, pour les comparaisons d'inégalité, le résultat fera intervenir le codage des caractères concernés. Rappelons que Java impose Unicode, ce qui implique les relations suivantes :

'0' < '1' < '2' < ... < '9' < 'A' <'B' < 'C' < ...< 'Z' < 'a' < 'b' < 'c' < ... <'z'

4.4 Cas particulier des opérateurs == et !=

En plus des situations évoquées précédemment (expressions numériques ou de type caractère), ces opérateurs peuvent s'appliquer à des valeurs de type booléen. L'expression suivante a un sens :

```
a < b == c < d
```

Compte tenu des priorités relatives des opérateurs concernés, elle sera interprétée comme :

```
(a < b) == (c < d)
```

Elle sera vraie si les deux comparaisons $a<b$ et $c<d$ sont soit toutes les deux vraies, soit toutes les deux fausses.