Jean-Bernard Boichat

4e édition

Apprendre Java et C++ en parallèle

EYROLLES

Apprendre Java et C++ en *parallèle*

CHEZ LE MÊME ÉDITEUR

Ouvrages sur Java

A. TASSO. – **Le livre de Java premier langage (5e édition).**
N°12376, 2008, 520 pages + CD-Rom.

C. DELANNOY. – **Programmer en Java.** *Java 5 et 6.*
N°12326, 2008, 788 pages (format semi-poche).

J. BOUGEAULT. – **Java - La maîtrise.** *Java 5 et 6.*
N°12250, 2008, 550 pages.

A. PATRICIO. – **Java Persistence et Hibernate.**
N°12259, 2008, 364 pages.

C. DELANNOY. – **Exercices en Java.** *Java 5.0.*
N°11989, 2006, 314 pages.

E. PUYBARET. **– Les Cahiers du programmeur Java (3e édition).**
Java 1.4 et 5.0.
N°11916, 2006, 370 pages + CD-Rom.

R. FLEURY. **– Les Cahiers du programmeur Java/XML.**
N°11316, 2004, 218 pages.

P. HAGGAR. – **Mieux programmer en Java.** *68 astuces pour optimiser son code.*
N°9171, 2000, 256 pages.

J.-P. RETAILLÉ. – **Refactoring des applications Java/J2EE.**
N°11577, 2005, 390 pages.

Ouvrages sur C++

C. DELANNOY. –**C++ pour les programmeurs C.**
N°12231, 2007, 602 pages.

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

C. DELANNOY. – **Apprendre le C++.**
N°12135, 2007, 760 pages.

H. SUTTER. – **Mieux programmer en C++.**
N°9224, 2000, 215 pages.

Jean-Bernard Boichat

Apprendre Java et C++ en *parallèle*

4e édition

EYROLLES

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

© Groupe Eyrolles, 2008, ISBN : 978-2-212-12403-3

Pour Ornella, qui m’a soutenu et cajolé durant cet été 2008 pas comme les autres !

À mes chers enfants, Nathalie, Stéphanie et Nicolas.

Table des matières

ANNEXES

Avant-propos

Pourquoi un tel ouvrage ?

Les réponses à cette question sont multiples. Au cours de cette préface, nous essayerons d'y répondre sans trop de philosophie et de débats contradictoires. Nous commencerons par deux raisons évidentes :

- Ces deux langages de programmation, Java et C++, sont très semblables, tout au moins dans leurs syntaxes. Il est ainsi tout à fait possible de reprendre un morceau de code du premier langage et de l'appliquer sans adaptation dans le second.
- Ils sont tous les deux très populaires dans le monde de l'informatique, le premier avec la venue d'Internet, le second comme langage système essentiel.

Pour assurer une progression constante et logique, au travers d'une comparaison directe de ces deux langages, il nous a fallu structurer la présentation du livre en fonction des particularités de chacun. Cela n'a pas été facile et nous avons accepté le défi de tenter une telle expérience. Il a été évidemment impossible de couvrir tous les détails de Java et de C++ car nous avons voulu que cet ouvrage conserve une épaisseur raisonnable.

À qui s'adresse ce livre ?

Nous aimerions dire aux débutants, mais ce ne serait sans doute pas honnête vis-à-vis des experts et des gourous C++, car ce dernier langage est considéré comme l'un des plus difficiles à assimiler et à maîtriser. Nous pensons aussi aux programmeurs Basic ou, mieux encore, à ceux favorisés par leurs connaissances dans des langages plus orientés objet, comme Delphi, le langage Pascal de Borland ou le C#.

Comme autres cas de figure, nous citerons également les programmeurs Java, Smalltalk, C ou C++ traditionnels qui aimeraient s'initier à l'un de ces deux langages essentiels et les ajouter à leur curriculum vitae.

Quel est l'intérêt d'apprendre plusieurs langages ?

Dans le cas précis, comme ces deux langages sont très proches, autant les apprendre en parallèle. Un autre aspect, encore plus essentiel, est l'apprentissage de leurs différences,

de leurs qualités et de leurs défauts, ce qui nous amènera ainsi à programmer en utilisant des techniques variées et réfléchies. Le résultat sera un code beaucoup plus propre, conçu selon une vision plus large des systèmes, de leurs possibilités et de leurs limites.

Pour un programmeur, aussi bien débutant que professionnel, l'apprentissage d'un nouveau langage est avant tout enrichissant. Une fonctionnalité manquante dans un langage l'amènera à une remise en question de ses méthodes. Par exemple, quand un programmeur C++ découvre qu'il n'existe pas de destructeur en Java, il va se poser toute une série de questions. En fin de compte, cette réflexion pourra sans aucun doute le conduire à réviser ses concepts et à écrire ses programmes C++ différemment. De la même manière, un programmeur C qui cherche à apprendre C++ et Smalltalk en parallèle, ce qui est loin d'être une mauvaise idée, va sans doute découvrir de nouveaux termes comme celui de variable de classe et se mettra à douter de ses vieilles habitudes en programmation C. Enfin, un programmeur qui, en guise de premier exercice en Java, décide de transférer une page Web sur son site, va certainement remettre en question ses choix lorsqu'il devra employer tel ou tel langage pour ses applications.

En fin d'ouvrage, dans le dernier chapitre, nous donnerons un aperçu du langage de Microsoft né en 2001, le C#. Nous serons alors surpris de retrouver la plupart des thèmes déjà abordés avec Java et C++ et combien il sera aisé alors d'écrire un premier programme dans ce langage. Le lecteur se rendra compte alors que son bagage informatique est déjà solide.

Dans cet ouvrage, nous parlerons aussi de performances, ainsi que de la manière de vérifier et de tester les programmes. Ces deux aspects font partie de la formation des informaticiens ; ils sont importants dans l'acquisition des méthodes de travail et des connaissances techniques pour concevoir et développer des applications professionnelles.

Quelles versions de Java et de C++ ?

Il faut tout d'abord indiquer que cet ouvrage ne couvre que le bien nommé « pure Java ». Il en va de même pour le C++ que nous appelons plus précisément le « Standard C++ ». Un grand nombre de fonctions intégrées dans le JDK 1.6 (officiellement nommé JDK 6) peuvent être directement comparées à celles disponibles dans le Standard C++. Le code C++ présenté dans cet ouvrage, tout comme le code en Java, devrait de plus pouvoir se compiler et s'exécuter dans plusieurs environnements comme Windows ou Linux.

Les exemples contenus dans ce livre couvrent des domaines variés et parfois complexes. Cependant, le code présenté restera compilable, nous aimerions dire pour l'éternité, et l'acheteur de ce livre n'aura pas besoin d'acquérir une nouvelle édition dans les prochains mois.

Le langage Java a beaucoup évolué ces dernières années, au contraire du C++ qui a atteint une stabilité naturelle. Aujourd'hui, nous parlons de Java 2 (Swing, Beans), Java 5 ou Java 6, qui représentent les versions de 1.2 à 1.4, 1.5 et 1.6 respectivement, des JDK de Sun Microsystems. Certains concepts et constructions de la version 1.1 ont été remplacés par des techniques plus évoluées. Dans cet ouvrage, nous laisserons volontairement de

côté la plupart des méthodes et des classes dépréciées des versions antérieures à la 1.6. Nous mentionnerons aussi quelques nouveautés apparues depuis la version 1.5.

Pourquoi le Standard C++ ?

Pourquoi avons-nous choisi le Standard C++ et non pas le langage C ou le C++ traditionnel ? Il y a en fait plusieurs raisons à cela. La principale concerne les nouvelles bibliothèques qui ont été rajoutées à partir de 1995 et qui font partie du langage au même titre que celles, nombreuses, qui font partie de Java. Comme exemple, nous pourrions citer la classe `string` qui n'existe pas dans le C++ classique. Certaines classes des premières versions du Standard C++ ont été dépréciées et corrigées dans cette édition.

Le Standard C++ est maintenant aussi agréé par les standards ANSI et ISO et représente une base encore plus solide pour un langage qui ne devrait plus beaucoup évoluer selon les dires de son créateur, Bjarne Stroustrup.

Comment est présenté cet ouvrage ?

Lors de la présentation des langages, nous commençons en général par le C++ et poursuivons ensuite avec la partie Java. En effet, pourquoi ne serions-nous pas respectueux avec le plus vieux d'abord ? En cas de difficultés, suivant les connaissances du programmeur, il est tout à fait possible de passer plus rapidement sur un sujet et d'y revenir par la suite. Pour un programmeur Visual Basic, s'initier en Java et C++ va représenter un travail important, car la syntaxe sera toute nouvelle pour lui. Dans ce cas-là, il devra sans doute se concentrer d'abord sur la partie Java et revenir ensuite sur le C++ avec ses particularités et ses variantes plus complexes.

Si nous avions traité l'un ou l'autre de ces langages séparément, la structure de cet ouvrage aurait été totalement autre. Java et C++ possèdent en effet quelques particularités foncièrement différentes qui auraient pu être exposées d'une autre manière. Ici, la difficulté principale réside dans les premiers chapitres où, pour présenter des exemples qui tiennent debout, nous avons souvent besoin de connaître certains aspects du langage qui seront examinés dans les chapitres suivants, et ce tout en traitant en parallèle les deux langages.

Tous les exemples de code présentés dans les différents chapitres ont été compilés séparément et souvent sous différents systèmes d'exploitation. Ils sont bien évidemment présents sur le CD-Rom accompagnant cet ouvrage.

De quel matériel a-t-on besoin ?

Cet ouvrage est conçu pour travailler dans un environnement Windows (XP ou Vista). Dans l'annexe F, nous montrerons qu'il est aussi possible d'utiliser Linux.

Un système équipé d'un processeur cadencé à 450 MHz et de 256 Mo de mémoire vive est suffisant pour assurer une compilation des programmes et garantir un environnement

agréable pour le développement. Le CD-Rom fourni avec cet ouvrage contient tous les outils de développement nécessaires ainsi que NetBeans (voir annexe E) pour Java et C++. Pour pouvoir utiliser correctement NetBeans, un processeur récent et 1 Go de mémoire sont recommandés.

Nous trouverons aussi, sur ce même CD-Rom, un éditeur de texte et la version la plus récente du SDK de la plate-forme Framework .NET de Microsoft qui inclut un compilateur C#.

Pourquoi autant d'exemples et d'exercices ?

À notre avis, il n'y en a jamais assez.

Lorsque nous débutons en programmation, nous commençons par écrire de petits morceaux de programmes. Ensuite, nous passons à des exercices plus complexes, à de petits outils pour gérer notre travail ou même à écrire de petits jeux intelligents. L'essentiel est d'écrire du code, beaucoup de code. Un bon programmeur sera toujours un collectionneur d'exemples, d'essais et de programmes, qu'il pourra rapidement retrouver lorsqu'il en aura besoin.

Pratiquement tous les exercices de ce livre sont proposés dans les deux langages, sauf si cela est explicitement indiqué. Les solutions sont disponibles sur le CD-Rom d'accompagnement. Des `Makefile` (GNU `make`) sont à disposition pour chaque chapitre et permettent de recompiler le code en cas de besoin.

Les exercices ont donné à l'auteur l'occasion de vérifier la consistance et la structure de cet ouvrage. Ils peuvent aussi permettre aux lecteurs de suivre les progrès de leur apprentissage.

Commentaires et suivi de l'ouvrage

L'auteur apprécierait de recevoir des commentaires ou des critiques de son ouvrage à l'adresse électronique suivante :

```
À:      jean-bernard@boichat.ch
Objet:  Java et C++
```

Sur le site Web :

```
http://www.boichat.ch/javacpp/
```

nous pourrons trouver d'éventuelles corrections découvertes dans les exemples et les exercices, ainsi que des remarques pertinentes ou des difficultés rencontrées par certains lecteurs. Nous aurons également à notre disposition des informations sur les nouvelles versions des compilateurs et les nouveaux outils, ainsi qu'une liste de sites et d'ouvrages intéressants et recommandés. Toutes ces informations seront régulièrement mises à jour par l'auteur.

1

L'incontournable Hello world

Nous le retrouvons dans presque tous les ouvrages de programmation. Il vient toujours en tête ! Ce sera aussi pour nous l'occasion de nous familiariser avec l'environnement de développement. Dans ce chapitre, il s'agira tout d'abord d'éditer les fichiers source, à savoir `hello.cpp` et `Hello.java`.

Sur le CD-Rom d'accompagnement figure un éditeur bien pratique pour Windows, que nous pouvons utiliser pour les différentes étapes de la programmation de nos exemples ou exercices : l'éditeur Crimson (voir annexe C). L'environnement de Crimson permet d'éditer les programmes source C++ et Java qui sont des fichiers de texte ASCII, de les compiler et de les exécuter. La compilation et le résultat de l'exécution des programmes peuvent s'afficher dans une fenêtre de travail intégrée à Crimson ; c'est donc l'outil idéal pour cela. Quant aux utilisateurs de Linux, ils ne devraient pas avoir de difficultés à choisir les outils adaptés aux tâches décrites ci-dessous (voir annexe F).

Dans ce chapitre, nous allons aussi introduire le fameux `make` de GNU, qui nous permettra d'automatiser la compilation. Avant de pouvoir utiliser le `make` ainsi que les compilateurs Java et C++ et de les intégrer dans l'éditeur Crimson, il faudra tout d'abord installer ces produits. Ils sont tous disponibles sur le CD-Rom, comme décrit dans l'annexe B. Un autre éditeur que Crimson peut évidemment être utilisé, mais il faudra alors compiler les programmes avec le `make` et les exécuter dans une fenêtre DOS (procédures décrites dans l'annexe B). Un outil Open Source beaucoup plus complet, NetBeans, peut aussi être utilisé (voir les annexes E et F pour son installation et son utilisation à travers des exemples complets).

Nous avons étendu l'exercice traditionnel en ajoutant la date, qui sera imprimée avec notre incontournable « `Hello world` ».

Hello world en C++

La première tâche consiste à introduire le code source dans le fichier `hello.ccp` via l'éditeur Crimson tel que décrit dans l'annexe C.

```
// hello.cpp
#include <ctime>
#include <iostream>

using namespace std;

int main()
{
 time_t maintenant;
 time(&maintenant);

 cout << "Hello world en C++: " << ctime(&maintenant) << endl;
 return 0;
}
```

Avant de passer à la compilation, nous allons examiner ce code dans les détails, en laissant de côté le `main()` et le `cout`, qui seront analysés et comparés entre les deux langages un peu plus loin. Il en va de même pour le :

```
using namespace std;
```

qui identifie un espace de noms, que nous étudierons au chapitre 6. Nous pouvons déjà signaler ici que l'utilisation de `namespace` n'est disponible que dans le Standard C++. Cet emploi est lié à la disparition en Standard C++ des `.h` dans les en-têtes. Si nous avions écrit le code suivant :

```
// hello2.cpp
#include <time.h>
#include <iostream.h>

int main()
{
 time_t maintenant;
 time(&maintenant);

 cout << "Hello world2 en C++: " << ctime(&maintenant) << endl;
 return 0;
}
```

nous aurions obtenu en fait le même résultat, excepté que cette dernière version est plus familière aux programmeurs C et C++ traditionnels ! En le compilant, nous obtiendrions ces remarques de codes dépréciés :

```
C:/MinGW/bin/../lib/gcc/mingw32/3.4.5/../../../../include/c++/3.4.5/backward/
➥iostream.h:31,
                 from hello2.cpp:4:
#warning This file includes at least one deprecated or antiquated header.
```

Le binaire (hello2.exe) serait tout de même généré et correctement exécutable.

Pour en revenir à la première version, les :

```
#include <ctime>
#include <iostream>


using namespace std;
```

vont en effet de pair, et nous les utiliserons tout au long de cet ouvrage. Nous travaillerons presque exclusivement avec les formes sans les .h, c'est-à-dire avec le Standard C++, même si, à de rares occasions, il nous arrivera d'utiliser des .h, c'est-à-dire des fonctions C qui ne sont pas disponibles dans la bibliothèque du Standard C++.

Les #include <ctime> et #include <iostream> sont des fichiers d'en-têtes nécessaires à la définition des classes et des objets utilisés. Si time_t, time et ctime n'étaient pas introduits, la ligne #include <ctime> ne serait pas obligatoire.

Il faut noter que le fichier <ctime> du Standard C++ contient en fait l'include <time.h>, qui est un héritage des fonctions C. Nous reviendrons sur la directive include au chapitre 4, section « Directive include ».

Dans <iostream>, nous avons la définition de cout et de son opérateur <<.

time_t est un type défini dans la bibliothèque ANSI C++ permettant de définir la variable maintenant, qui sera utilisée pour obtenir la date actuelle au travers de time(&maintenant). Nous reviendrons plus loin sur le &, qui doit être ajouté devant la variable car la fonction C time() s'attend à recevoir une adresse. Le résultat sera obtenu au moyen de << ctime (&maintenant), comme nous allons le voir ci-dessous. Le chaînage des opérateurs << est à noter, ce qui permet d'envoyer une série de blocs de données sans devoir les définir sur des lignes séparées.

Le fichier hello.cpp est ensuite compilé par le compilateur C++, accessible par le chemin d'accès PATH décrit dans l'annexe B. Cette compilation se fait de la manière suivante :

```
g++ -o hello.exe hello.cpp
```

Le paramètre –o de g++ indique que le nom hello.exe correspond au fichier binaire compilé à partir des fichiers source inclus dans la ligne de commande (ici, hello.cpp). Avec Linux, nous pourrions utiliser :

```
g++ -o hello hello.cpp
```

Les fichiers hello.exe ou hello sont les exécutables. Par la suite, nous utiliserons systématiquement l'extension .exe, qui est nécessaire sous DOS. Un programme nommé hello.exe pourra aussi s'exécuter sous Linux, qui possède une propriété d'exécutable pour le fichier. Ce n'est pas le cas sous DOS, où les programmes doivent se terminer avec une extension déterminée comme .bat, .com ou .exe. Lorsque nous exécutons le programme hello.exe, on obtient :

```
Hello world en C++: Mon Mar 31 14:55:20 2008
```

Le fichier `hello.exe`, nommé parfois exécutable binaire, contient le code machine que le processeur pourra directement exécuter grâce aux fonctions du système d'exploitation (DOS ou Linux). Il s'exécutera uniquement sur les machines ayant le même système d'exploitation. Le programme `hello.exe` peut aussi ne pas fonctionner sur un PC doté d'un processeur ancien, car le compilateur n'aura sans doute pas assuré de compatibilité avec les modèles antérieurs, par exemple, au Pentium 586 d'Intel. Pour fonctionner sous Linux, avec d'autres systèmes d'exploitation ou d'autres types de configurations matérielles, il faudra recompiler le programme `hello.cpp` sur ce type de machine. Un programme C ou C++ compilable sur différents supports est communément appelé portable. Nous allons voir que cela se passe bien différemment avec Java.

Hello world en Java

Avant tout, une première remarque s'impose concernant le nom des fichiers. Comme le nom d'une classe commence toujours par une majuscule, il est nécessaire de définir le nom du fichier selon le même principe, c'est-à-dire ici `Hello.java`. Faillir à cette règle est une erreur classique, et la plupart des débutants tombent dans le piège. En effet, si on oublie ce principe, cette classe ne pourra être compilée :

```
import java.util.*;

public class Hello {
   public static void main(String[] args) {
      Date aujourdhui;
      aujourdhui = new Date();
      System.out.println("Hello world en Java: " + aujourdhui);
   }
}
```

En Java, le traitement de la date est vraiment très simple. Nous n'avons plus des variables ou des structures compliquées comme en C ou en C++, mais simplement un objet. `aujourdhui` est ainsi un objet de la classe `Date` qui est créé grâce à l'opérateur `new`. Les détails de la méthode `println` sont expliqués plus loin.

La compilation s'effectue ainsi :

```
javac Hello.java
```

et nous l'exécutons par :

```
java Hello
```

ce qui nous donne comme résultat :

```
Hello world en Java: Mon Mar 31 14:55:44 GMT+02:00 2008
```

Nous remarquons ici que nous n'avons pas de `Hello.exe`. En effet, le processus se déroule différemment en Java : après la compilation de `Hello.java`, avec `javac.exe` (qui est, comme pour C++, un exécutable binaire différent sur chaque système d'exploitation), un fichier compilé `Hello.class` est généré. Pour exécuter `Hello.class`, nous utilisons le

programme java.exe, qui est la machine virtuelle de Java. Notons qu'il faut enlever dans la commande l'extension .class. java.exe trouvera alors un point d'entrée main() dans Hello.class et pourra l'exécuter. Si la classe Hello utilise d'autres classes, elles seront chargées en mémoire part la machine virtuelle si nécessaire.

Le GMT (*Greenwitch Mean Time*) pourrait être différent suivant l'installation et la configuration du PC, la langue ou encore la région. Un CET, par exemple, pourrait être présenté : *Central European Time*.

Une fois que Hello.java a été compilé en Hello.class, il est alors aussi possible de l'exécuter sur une autre machine possédant une machine virtuelle Java, mais avec la même version (ici 1.6) ou une version supérieure. Le fichier Hello.class sera aussi exécutable sous Linux avec sa propre machine virtuelle 1.6. Si la machine virtuelle ne trouve pas les ressources nécessaires, elle indiquera le problème. Dans certains cas, il faudra recompiler le code Java avec une version plus ancienne pour le rendre compatible.

Nous constatons donc qu'en Java, le fichier Hello.class ne contient pas de code machine directement exécutable par le processeur, mais du code interprétable par la machine virtuelle de Java. Ce n'est pas le cas du programme C++ hello.exe, qui utilise directement les ressources du système d'exploitation, c'est-à-dire de Windows. L'application hello.exe ne pourra pas être exécutée sous Linux et devra être recompilée (voir les exemples de l'annexe F).

La machine virtuelle Java – JRE

Dans cet ouvrage, nous devons tout de même mentionner la machine virtuelle JRE (*Java Runtime Environment*) bien que nous allons certainement passer la moitié de notre temps à éditer et compiler des programmes Java (l'autre moitié pour le C++).

Sur notre machine de développement, l'exécutable java.exe se trouve dans le répertoire C:\Program Files\Java\jdk1.6.0_06\bin. Si nous examinons le répertoire C:\Program Files\Java, nous découvrirons un second répertoire nommé jre1.6.0_06 ainsi qu'un sous-répertoire bin qui contient également un fichier java.exe.

Nos amis ou clients à qui nous livrerons la classe compilée Hello.class n'auront pas besoin d'installer JDK, le kit de développement, mais uniquement la machine virtuelle, c'est-à-dire JRE. Sun Microsystems met à disposition différentes distributions de JRE, juste pour exécuter notre Hello.class :

```
"C:\Program Files\Java\jre1.6.0_06\bin\java.exe" Hello
Hello world en Java: Wed Jul 30 13:58:09 CEST 2008
```

et ceci depuis le répertoire C:\JavaCpp\EXEMPLES\Chap01.

Le JDK n'est nécessaire que pour la compilation, c'est-à-dire lorsque nous utilisons la commande javac.exe. Sur une même machine, nous pourrions avoir plusieurs JDK et plusieurs JRE dans le répertoire C:\Program Files\Java (voir annexe B, section « Désinstallation des anciennes versions »).

Erreurs de compilation

L'oubli de la déclaration des ressources est une erreur classique. Si nous effaçons la première ligne (`import java.util.*;`) ou si nous la mettons en commentaire (`//` devant), nous générerons l'erreur suivante :

```
javac Hello.java
Hello.java:5: Class Date not found.
    Date aujourdhui;
    ^
Hello.java:6: Class Date not found.
    aujourdhui = new Date();
                     ^
2 errors
```

Il est à noter la manière claire dont le compilateur nous indique la position des erreurs. Ici, il ne trouve pas la classe `Date`. En ajoutant `import java.util.*`, nous indiquons au compilateur d'importer toutes les classes du *package* (paquet) des utilitaires de Java. Au lieu d'importer toutes les classes du package, nous aurions pu écrire :

```
import java.util.Date;
```

Compiler régulièrement est une très bonne habitude en Java et C++, en écrivant de petits morceaux et avant de terminer son code. Le simple oubli d'un point-virgule en C++ (qui indique la fin d'une instruction) peut entraîner une erreur quelques lignes plus loin et faire perdre inutilement du temps précieux.

Notre premier fichier Makefile

Pour compiler nos programmes Java et C++, nous allons utiliser tout au long de cet ouvrage un outil GNU bienvenu : le `make`. Le fichier `Makefile` est le fichier utilisé par défaut lorsque la commande `make` est exécutée sans paramètre. Cette commande est un héritage du monde Unix (Linux) que nous retrouverons aussi avec NetBeans (voir annexe E, section « Configuration pour le C++ et le make »). Le `Makefile` possède une syntaxe très précise basée sur un système de dépendances. Le `make` pourra identifier, en utilisant la date et l'heure, qu'une recompilation ou une action devra être exécutée. Dans le cas de fichiers Java, si un fichier `.java` est plus récent qu'un fichier `.class`, nous devrons considérer qu'un fichier `.class` devra être régénéré. L'outil `make` permet donc d'automatiser ce processus.

Voici notre tout premier exemple de `Makefile`, soit le fichier `MakefilePremier`, dont nous allons expliquer le fonctionnement. Les `Makefile` sont des fichiers texte ASCII que nous pouvons aussi éditer et exécuter sous Windows avec l'éditeur Crimson (voir annexe C) qui va nous simplifier le travail.

```
#
# Notre premier Makefile
#
all:          cpp java
```

```
cpp:          hello.exe hello2.exe
java:         Hello.class

hello.exe:    hello.o
              g++ -o hello.exe hello.o

hello.o:      hello.cpp
              g++ -c hello.cpp

hello2.exe:   hello2.o
              g++ -o hello2.exe hello2.o

hello2.o:     hello2.cpp
              g++ -c hello2.cpp

Hello.class:  Hello.java
              javac Hello.java

clean:
              rm -f *.class *.o *.exe
```

Avant d'exécuter ce `Makefile`, il faudrait s'assurer que tous les objets sont effacés. Dans le cas contraire, nous risquons de n'avoir aucun résultat tangible ou partiel, bien que cela resterait correct : uniquement les fichiers objets (`.o`, `.exe` et `.class`), dont les sources respectives ont été modifiées plus récemment, seront recompilés !

Nous aurions pu fournir un fichier `efface.bat` pour faire ce travail avec les commandes DOS :

```
del *.exe
del *.o
del *.class
```

mais nous avons préféré cette version :

```
make clean
```

L'entrée `clean` (nettoyer) va permettre ici d'effacer tous les fichiers `.class`, `.o` et `.exe` (s'ils existent) afin que le `make` puisse régénérer tous les objets et tous les exécutables. La commande `rm` est l'équivalent Linux de la commande DOS `del` et le paramètre `-f` va forcer l'effacement sans demander une quelconque confirmation. Un fichier `efface.bat` est fourni dans chaque répertoire des exemples et des exercices.

Le `make clean` et sa présence dans le `Makefile` sont importants : NetBeans (voir annexe E) en a besoin pour construire ses projets C++.

Mentionnons ce message que le lecteur pourrait rencontrer lors de son travail :

```
make: *** No rule to make target `clean'.  Stop.
```

Il indique qu'il n'y a pas de règle (`rule`) pour le point d'entrée `clean`. Nous devrons alors le rajouter dans le `Makefile` ou vérifier que la syntaxe du fichier est correcte (espaces et marques de tabulations tout particulièrement).

Enfin un premier make effectif

Pour exécuter le fichier `MakefilePremier`, nous devrons entrer la commande :

```
make -f MakefilePremier
```

En navigant dans le répertoire, nous pourrons constater que les fichiers `.o`, `.exe` et `.class` ont été régénérés. L'erreur sur la recompilation de `hello2.cpp` réapparaîtra évidemment (voir ci-dessus).

Si nous avons un nom de fichier `Makefile`, comme c'est le cas dans tous les chapitres, il nous faudra simplement exécuter :

```
make
```

Lorsque la commande `make` est exécutée, le fichier `Makefile` sera chargé et le point d'entrée `all` exécuté. La commande `make` permet aussi de spécifier un point d'entrée :

```
make -f MakefilePremier java
```

Ici, uniquement le point d'entrée `java` serait activé pour compiler la classe `Hello.class`. Les points d'entrée `Hello.class` ou `hello.exe` sont aussi possibles pour un choix plus sélectif.

`all`, `cpp` et `java` sont des actions à exécuter. `hello.exe`, `hello.o` et `Hello.class` sont des fichiers générés par les compilateurs. Il faut être très attentif avec le format des fichiers `Makefile`, car ils sont extrêmement sensibles à la syntaxe. Après les deux-points (`:`), il est préférable d'avoir un tabulateur (TAB), bien que nous puissions avoir des espaces, mais sur la ligne suivante nous avons toujours des tabulateurs. Suivant la grandeur des variables, nous en aurons un ou plusieurs, mais cela dépend aussi de la présentation que nous avons choisie. Après les `hello.exe:` et `hello.o:`, nous trouvons les dépendances et sur les lignes suivantes les commandes.

À la première exécution du `make`, `cpp` et `java` seront activés, car aucun des `hello.o`, `hello.exe` et `Hello.class` n'existe. En cas d'erreur de compilation dans `hello.cpp`, ni `hello.o` ni `hello.exe` ne seront créés.

Passons maintenant à la partie la plus intéressante : si `hello.cpp` est modifié, sa date sera nouvelle et précédera celle de `hello.o`. Les deux commandes `g++` seront alors exécutées.

Dans ce cas précis, la séparation en deux parties `hello.o` et `hello.exe` n'est pas vraiment nécessaire, puisque nous n'avons qu'un seul fichier `hello.cpp`. La commande suivante suffirait :

```
g++ -o hello.exe hello.cpp
```

Enfin, que se passe-t-il si nous définissons :

```
hello.o:            Hello.java hello.cpp
                    g++ -c hello.cpp
```

`hello.o` sera régénéré si `Hello.java` a changé et même si `hello.cpp` n'a pas été touché. On voit donc que cette dépendance est inutile.

Les trois premières lignes du fichier `MakefilePremier` sont des commentaires. Ils commencent par le caractère # :

```
#
# Notre premier Makefile
#
```

Nous pouvons, par exemple, les utiliser pour éliminer des parties de compilation pendant le développement :

```
all:          cpp #java
```

Ici, uniquement la partie C++ sera compilée.

Nous reviendrons sur les dépendances dues aux fichiers d'en-tête au chapitre 4. Les paramètres `-c` et `-o` y seront expliqués en détail.

Le point d'entrée main()

La fonction `main()` est le point d'entrée de tout programme. Le corps de la fonction située entre les accolades sera exécuté. En Java, il n'est pas possible de définir une fonction indépendante, car elle doit faire partie d'une classe. De plus, elle doit être déclarée `public` et `static`. Nous en comprendrons les raisons plus loin dans cet ouvrage. Contrairement au C++, chaque classe en Java peut posséder son entrée `main()`. Lorsque nous entrons :

```
java Hello
```

la méthode `main()` de la classe `Hello.class` est activée. La classe `Hello` pourrait utiliser d'autres classes possédant aussi un point d'entrée `main()`. Nous verrons plus loin que cette technique peut être employée pour tester chaque classe indépendamment.

Les paramètres de main()

`main()` peut recevoir des paramètres. Lorsque nous entrons une commande DOS telle que :

```
copy hello.cpp hello.bak
```

les fichiers `hello.cpp` et `hello.bak` sont les deux paramètres reçus par la commande `copy`. Nous allons examiner à présent les différences entre les deux langages et la manière de procéder pour tester et acquérir des paramètres. Par exemple, si nous devions programmer la commande DOS `copy` (équivalente à la commande `cp` sous Linux), il faudrait que nous vérifiions les paramètres de la manière qui va suivre.

main() et C++

Le fichier `copy_arg.cpp` peut se présenter sous cette forme :

```
// copy_arg.cpp
#include <iostream>

using namespace std;

int main(int argc, char **argv) {
  if (argc != 3) {
    cerr << "Nombre invalide de paramètres" << endl;
    return -1;
  }

  cout << "argv[0]: " << argv[0] << endl;
  cout << "argv[1]: " << argv[1] << endl;
  cout << "argv[2]: " << argv[2] << endl;
  return 0;
}
```

Ce programme C++ peut se compiler avec :

```
g++ -o copy_arg.exe copy_arg.cpp
```

ou avec un `Makefile` qui se trouve sur le CD-Rom. Nous pouvons ensuite l'exécuter avec :

```
copy_arg hello.cpp hello.bak
```

Il faut absolument joindre ces deux arguments, sous peine de récolter une erreur (nombre invalide de paramètres). Le résultat sera ainsi présenté :

```
argv[0]: copy_arg
argv[1]: hello.cpp
argv[2]: hello.bak
```

Nous savons déjà que `copy_arg.exe` peut être abrégé en `copy_arg` sous DOS, mais pas sous Linux.

main() et Java

Le fichier `CopyArgs.java` est en fait très similaire si nous l'étudions en détail :

```
public class CopyArgs {
  public static void main (String[] args) {

    if (args.length != 2) {
      System.err.println("Nombre invalide de paramètres");
      return;
    }
```

```
    System.out.println("args[0]: " + args[0]);
    System.out.println("args[1]: " + args[1]);
  }
}
```

Ce programme Java peut se compiler avec :

```
javac CopyArgs
```

ou un `Makefile`, combiné avec la compilation de la version C++ ci-dessus, qui se trouve sur le CD-Rom. La classe `CopyArgs.class` compilée sera exécutée avec la machine virtuelle Java de cette manière :

```
java CopyArgs hello.cpp hello.bak
```

Nous obtiendrons le résultat suivant :

```
args[0]: hello.cpp
args[1]: hello.bak
```

Analyse comparative

La première grande différence est le nombre de paramètres retournés, c'est-à-dire un de plus pour la version C++, où nous recevons (dans `argv[0]`) le nom du programme. Les `strings` existent aussi en Standard C++, mais la fonction `main()` est un héritage du C.

L'instruction `if` viendra au chapitre 3, mais la traduction de :

```
if (argc != 3) {
```

serait « si `argc` n'est pas égal à 3 alors… ».

Les différentes sorties à l'écran, `cout` et `cerr` pour C++, ainsi que `out` et `err` en Java, sont utilisées sur les deux canaux de sorties. Il est possible, plus particulièrement sous Linux, de filtrer ces deux sorties pour différencier les cas normaux (`cout`, `out`) des erreurs (`cerr`, `err`). Nous ne donnerons pas ici d'exemples sur l'utilisation de telles techniques, mais nous garderons cependant l'habitude d'utiliser le canal `cerr`/`err` pour les erreurs.

Observons ce qui se passe avec l'instruction :

```
cout << "argv[0]: " << argv[0] << endl;
```

Le premier « `argv[0]:` » est simplement un texte. Le texte entre guillemets est envoyé au travers de l'opérateur `<<` à `cout`. Ce dernier est défini dans le fichier `iostream`, qui se trouve dans la bibliothèque standard (`std`) du Standard C++. Nous reviendrons plus tard sur ces détails.

Les sorties à l'écran (`cout`) pouvant être chaînées, le terme suivant se trouve être `argv[0]`, `argv` étant un pointeur à une liste de pointeurs, c'est-à-dire la raison du `**` dans la déclaration du `main()`. `argv[0]` va pointer sur l'adresse mémoire où figure le premier paramètre, c'est-à-dire le nom de la commande. Si nous écrivions :

```
cout << "Adresse de argv[0]: " << &argv[0] << endl;
```

nous verrions apparaître sous forme hexadécimale l'adresse mémoire où se trouve en fait le texte « copy_arg.exe ». On peut également concevoir argv comme un tableau de chaînes de caractères, le 0 de argv[0] étant un index dans ce tableau.

Enfin, le endl est une opération aussi définie dans <iostream>, qui consiste à envoyer le caractère de saut de ligne, que l'on représente parfois comme le \n ou \010 (valeur octale).

En Java, nous n'avons pas de chaînes de caractères comme en C++, mais un String. Celui-ci est un objet représentant aussi une chaîne de caractères. length est une opération exécutée par le compilateur qui nous permet de recevoir la dimension de la variable args, qui est un tableau de String identifié avec le []. En C et en C++ nous recevons la variable argc. La dimension de args doit être de 2 dans notre exemple, ce qui nous permet d'obtenir les deux paramètres args[0] et args[1]. L'index commence toujours à 0 dans un tableau. L'index maximal est args.length -1.

L'objet out dans la bibliothèque Java System permet de sortir du texte sur la console au moyen de println(). out, défini dans le package java.lang.System, est un objet statique de la classe PrintStream ; println() est une méthode (fonction) de la classe PrintStream. Nous verrons ces concepts plus loin. Il n'est pas nécessaire d'importer le package System avec une instruction import, car il est reconnu par le compilateur.

Le ln de println() est équivalent au endl du C++. Dans cet exemple Java, nous utilisons l'opérateur + pour associer deux strings avant de les envoyer au travers de println(). Il est aussi possible d'utiliser plusieurs print() et le "\n" comme ici :

```
System.out.print("args[0]: ");
System.out.print(args[0]);
System.out.print("\n");
```

Le return est équivalent en C/C++ et en Java. En Java, nous avons un void main() et nous ne devons pas retourner de valeur. Le return peut d'ailleurs être omis si nous atteignons correctement la fin du programme. Avec un int main() en C++, il est nécessaire de retourner une valeur qu'il est judicieux de définir négativement en cas d'erreur.

Toute démarche doit se faire d'une manière logique. Si nous devions écrire les programmes copy_arg.cpp et Copy_arg.java en totalité, il faudrait vérifier l'existence du fichier entré comme premier paramètre (ici, hello.cpp). Ensuite, si le deuxième paramètre était un fichier existant (ici, hello.bak), il faudrait décider si nous acceptons de l'écraser ou non. Nous pourrions alors demander, avec un oui ou un non, si cela est acceptable ou encore définir un autre paramètre (par exemple, –f) pour forcer l'écriture. Nous aurions alors une commande comme :

```
copy -f hello.cpp hello.bak
```

Comme -f serait optionnel, il faudrait traiter les arguments différemment. Enfin, nous pourrions pousser l'analyse à l'extrême et vérifier si l'espace disque est suffisant avant de commencer la copie. En cas d'erreur, il faudrait effacer la mauvaise copie. Si nous n'avions pas la permission d'écrire, cela affecterait vraisemblablement la logique du traitement des erreurs ! Il y a en effet des cas en programmation où nous découvrons qu'un oubli

dans l'analyse ou la conception (*design*, en anglais) peut entraîner une restructuration complète du code !

> **Note**
>
> Que se passe-t-il sous DOS ?
>
> Si nous exécutons cette ligne d'instruction C++ sous DOS :
>
> `cout << "éàöâê" << endl;`
>
> cela pourrait nous surprendre au premier abord. En effet, les caractères sous DOS sont différents. Les lettres et caractères de la langue anglaise vont apparaître correctement, alors que les lettres avec accents poseront quelques difficultés. Mais comme nous exécuterons nos programmes avec Crimson, c'est-à-dire sous Windows, nous ne rencontrerons pas ce type de problème. Nous reviendrons sur ce point plus loin dans cet ouvrage.

Jouer avec Crimson

C'est sans doute le meilleur moment pour retourner dans l'éditeur Crimson (voir annexe C) et pour refaire quelques-unes des démarches et fonctions usuelles :

- Charger un fichier `.java`, le compiler et l'exécuter.
- Charger un fichier `.cpp` , le compiler et l'exécuter.
- Charger un `Makefile` et l'exécuter avec le `make`.

Constater que nous pouvons aussi charger des fichiers `.bat` (`efface.bat` ou `Makefile.bat`) et les exécuter dans Crimson directement et sans fenêtre DOS.

Résumé

À la fin de ce chapitre, nous savons déjà compiler des programmes Java et C++, bien qu'il ne soit pas encore possible d'écrire un programme digne de ce nom. L'outil `make` de GNU permet d'automatiser le processus lorsque des changements dans le code ont été apportés.

Exercices

Toutes les solutions des exercices de cet ouvrage sont disponibles sur le CD-Rom (voir annexe B, section « Installation des exemples et des exercices »). En programmation, il n'y a pas UNE solution, mais plusieurs. Il vaut cependant la peine de s'accrocher et de faire quelques-uns de ces exercices, voire tous, avant de consulter les solutions.

1. Écrire une classe Java `Bonjour` qui nous sortira un :

   ```
   Bonjour et bonne journée
   ```

2. Écrire le `Makefile` pour cette classe et vérifier qu'une compilation est à nouveau exécutée si le fichier `Bonjour.class` est effacé ou bien si le fichier `Bonjour.java` est modifié avec un autre texte de salutations !

3. Écrire une version de `copy_arg.cpp` et de `CopyArgs.java` qui, lorsque aucun paramètre n'est donné, imprime à l'écran un descriptif des paramètres nécessaires aux programmes.
4. Créer un fichier `execHello.bat` pour exécuter les binaires `Bonjour.class` et `bonjour.exe` des exercices 1 et 2 précédents. Ajouter une pause en fin de fichier et exécuter ce dernier deux fois depuis Crimson. Examiner l'effet de la pause et le résultat obtenu en double-cliquant sur le fichier depuis l'explorateur de Windows.

2

La déclaration et l'affectation des variables numériques

Le traitement des variables non numériques sera étudié au chapitre 7. Comme la chaîne de caractères `char*` en C++, qui est une variable essentielle de ce langage, n'existe pas en Java (remplacée par la classe `String`), il nous a semblé nécessaire de traiter séparément et soigneusement ce sujet essentiel.

Déclaration des variables

Une variable est un moyen donné par le langage pour définir et manipuler un objet et sa valeur. Elle correspond en fait à un emplacement dans la mémoire. Si notre programme a besoin de conserver le nombre de personnes invitées le week-end prochain, nous devons définir cette variable de la manière suivante :

```
int nb_personne;
```

Une déclaration est une instruction Java ou C++. Elle doit se terminer par un point-virgule. La première partie de cette instruction est un mot-clé qui correspond soit à un type prédéfini soit à un nom de classe. Dans le cas d'une classe, sa définition doit apparaître avant l'instruction. Cette définition est généralement contenue dans un fichier d'en-tête en C++ (`#include`) ou de package en Java (`import`). Nous reviendrons sur les détails tout au long de ce chapitre.

La deuxième partie de l'instruction représente le nom de la variable. Ce nom doit être unique et ne pas correspondre à un mot-clé défini par le langage. Il doit impérativement commencer par une lettre ou le caractère de soulignement _, mais peut contenir des chiffres et ce même caractère de soulignement. Les lettres doivent faire partie de l'alphabet anglais

et peuvent être en majuscule ou en minuscule. Dans certains langages de programmation comme ADA, la variable `MonNombre` sera identique à `monNombre`. En Java et C++, ces deux variables seront différentes. En C++, il n'est pas souhaitable de commencer un nom de variable ou de fonction par le caractère de soulignement, car les compilateurs l'utilisent souvent pour des variables et fonctions spéciales.

Choix des noms de variables

Un nom de variable ne doit être ni trop long ni trop court. Dans le cas de `nb_personne`, ce nom est presque parfait, car il atteint déjà une grandeur respectable. Un nom trop court comme `n` ou trop long comme `nombre_de_personnes_invitees` n'est pas judicieux. En C++, il est recommandé de n'utiliser que des minuscules, le caractère _ et des chiffres ; en Java, il est souhaitable d'employer des minuscules, des majuscules et des chiffres. Par ailleurs, il serait préférable de toujours ajouter un commentaire dans le code, du style de :

```
int nb_personnes;  // nombre de personnes invitées
int nbPersonnes;   // alternative Java (minuscule au début)
```

Un commentaire tel que :

```
int nb_personnes;  // nombre de personnes
```

n'apportera rien de nouveau.

Une difficulté pourrait apparaître si nous devions définir une autre variable pour le nombre de personnes qui ne sont pas invitées ! Nous pourrions écrire par exemple :

```
int nb_pers_inv;  // nombre de personnes invitées
int nb_pers_abs;  // nombre de personnes non invitées (absentes)
```

Bien que « absent » ne veuille pas dire « pas invité », nous préférons cette écriture à la forme :

```
int nb_pers_pinv;  // nombre de personnes non invitées
```

car la lettre p pourrait être facilement oubliée ou rajoutée et entraîner des erreurs de programmation difficiles à corriger !

Une variable trop longue est aussi difficile à gérer si nous avons plusieurs tests de conditions sur la même ligne :

```
if ((nombre_de_personnes_invitees < 0) || ((nombre_de_personnes_invitees == 13) || .... )
```

Il n'est pas recommandé d'adopter des particularités de langage comme d'ôter toutes les voyelles d'une variable. Ce n'est pas une invention, cela existe ! La variable `nombre_de_personnes_invitees` deviendrait alors `nmbr_d_prsnns_nvts`… Il faut toujours penser à l'éventuel programmeur qui reprendrait un jour ce code.

Pourquoi ne pas utiliser simplement la langue anglaise, dont les mots sont plus courts (`nb_person`) ? C'est beaucoup plus facile, et il suffit de lire une fois un programme en langue allemande pour se rendre compte qu'un style « international » serait bienvenu.

Enfin, utiliser des lettres uniques, comme `i`, `j` ou `k`, est tout à fait approprié dans des instructions de répétitions et de boucles.

Affectation des variables

Dans cet ouvrage, nous avons utilisé les termes d'affectation et « d'assignement ». Ces deux termes sont équivalents, bien que le deuxième ne soit pas vraiment français, mais utilisé dans la littérature anglaise et tout à fait compréhensible.

Les variables sont déclarées et affectées au moyen d'instructions. Si nous écrivons :

```
int valeur1;
valeur1 = 10;
int valeur2 = 20;
```

nous remarquerons que la variable `valeur1` est déclarée et affectée en deux étapes. Après la première instruction, `valeur1` ne sera pas encore initialisée, bien que le compilateur lui ait déjà réservé une zone mémoire.

La forme :

```
int valeur3(30) ;
```

est intéressante, mais seulement acceptée en C++. Nous en verrons la raison lors du traitement et de l'affectation des objets au chapitre 4.

En Java, le code suivant ne compilera pas :

```
public class InitInt {
   public static void main(String[] args) {
      int valeur1;
      System.out.print("Valeur1: ");
      System.out.println(valeur1);
   }
}
```

L'erreur suivante sera reportée :

```
---------- Capture Output ----------
>"C:\Program Files\Java\jdk1.6.0_06\bin\javac.exe" -Xlint -classpath
➥C:\JavaCpp\EXEMPLES\Chap02 InitInt.java
InitInt.java:5: variable valeur1 might not have been initialized
      System.out.println(valeur1);
                         ^
1 error
> Terminated with exit code 1.
```

Nous remarquons donc qu'il n'est pas possible de compiler ce code, alors que cela est tout à fait permis en C++. C'est l'un des grands avantages de Java : son compilateur est très restrictif et force le programmeur qui a une culture C++ à coder correctement son programme. La compilation est simplement refusée ! Le paramètre `-Xlint` (voir le chapitre 14, section «Les types génériques en Java », et l'annexe C, en fin de section « Raccourci ou

favori ») n'est pas nécessaire ici, mais donnerait encore plus d'informations sur des formes ou usages (*warnings*) qui ne suivent pas précisément les spécifications du langage Java.

En C++, c'est donc une bonne habitude d'initialiser les variables au moment de leurs déclarations. Si nous devions vérifier plus tard la valeur d'une variable, nous pourrions avoir des surprises. Dans ce code :

```
int erreur;
// beaucoup plus loin
if (erreur) {
  // code d'erreur
}
```

les caractères `//` nous permettent d'introduire des commentaires qui seront ignorés par les compilateurs. En C++, pour que le code d'erreur soit exécuté, il faut que la variable `erreur` soit différente de 0. Même si les compilateurs mettent généralement les variables non initialisées à 0, ce n'est pas une raison pour ne pas le faire. Nous reviendrons sur les variables booléennes dans le chapitre suivant. Ce type de code est généralement utilisé avec des instructions telles que :

```
erreur = fonction();
```

qui consiste à appeler `function()`, qui nous retournera une valeur qui sera stockée dans la variable `erreur`. Si cette ligne de code venait à disparaître après des corrections un peu trop rapides, nous aurions soit un résultat inattendu, soit une instruction inutile. Dans ce cas précis, un :

```
int erreur = 1;
```

serait judicieux.

`int` est aussi appelé un mot-clé ou mot réservé. En effet, il n'est pas possible par exemple d'utiliser `int` pour un nom de variable. `int` définit le type de variable représentant des nombres entiers positifs et négatifs. Sa zone mémoire lui permet de stocker un nombre dans des limites prédéfinies. Si le nombre est trop grand ou d'une autre forme comme une valeur numérique décimale (avec virgule flottante), il faudra utiliser d'autres types comme :

```
float valeur3 = 1.2;
double valeur4 = 1000000;
```

Un point essentiel au sujet de la valeur d'une variable est à noter : il est possible que des opérations arithmétiques affectent cette valeur et débordent de sa capacité. Il est souvent préférable en cas de doute d'utiliser un `long` au lieu d'un `int`.

Le code Java suivant, qui augmente de 1 la variable `valeur1`, compile parfaitement :

```
public class IntLong {
   public static void main(String[] args) {
      int valeur1 = 2147483647;
      valeur1++;
      System.out.print("Valeur1: ");
```

```
        System.out.println(valeur1);
    }
}
```

mais retournerait une erreur de compilation si nous avions donné la valeur de 2147483648 à la variable :

```
---------- Capture Output ----------
>"C:\Program Files\Java\jdk1.6.0_06\bin\javac.exe" -Xlint -classpath
➥C:\JavaCpp\EXEMPLES\Chap02 IntLong.java
IntLong.java:3: integer number too large: 2147483648
      int valeur1 = 2147483648;
                    ^
1 error
> Terminated with exit code 1.
```

Cependant, si nous compilons et exécutons le code avec la valeur de 2147483647, le résultat pourrait sembler stupéfiant :

```
Valeur1: -2147483648
```

C'est simple à comprendre si nous examinons ce tableau et les valeurs binaires présentées. Il faut déjà se rappeler comment est formé un nombre binaire, et nous prendrons l'exemple d'un octet, c'est-à-dire de 8 bits :

00000000 = 0	00000001 = 1	00000010 = 2	00000011 = 3	00000100 = 4
00000101 = 5	00001000 = 8	00001010 = 10	00010000 = 16	10000000 = 128

Nous voyons que les bits se promènent de droite à gauche et sont simplement des 0 ou 1 en mémoire. Notre 128, dans une représentation signée, est en fait le bit de négation. Sa valeur signée serait alors –127. Si nous faisions ce même exercice en C++ nous pourrions avoir des surprises et ne pas obtenir même résultat suivant les machines et les versions de compilateur. Si nous devions travailler avec de telles valeurs, pour limiter les risques, il suffirait de passer d'`int` en `long`. Voici un exemple de codage des nombres négatifs (complément à 2). Ce n'est pas si simple d'établir une formule pour les personnes qui ont quelques difficultés avec les mathématiques. Ce qu'on peut dire tout simplement : si nous avons un bit 1 tout à gauche, nous avons un chiffre négatif :

```
0111.1111.1111.1111.1111.1111.1111.1111 = 2147483647
1000.0000.0000.0000.0000.0000.0000.0000 = -2147483648
```

Le point permet simplement de séparer les groupes par 4 bits étalés sur 4 octets. L'instruction `valeur1++;` ajoute 1 en mémoire, c'est-à-dire décale à gauche, comme pour le passage de 3 (011) à 4 (100), et la nouvelle valeur binaire correspond à la valeur négative extrême. Nous avons fait le tour, de l'extrême droite à l'extrême gauche, et le bit de négation est positionné ! Dans notre tableau à 8 bits ci-dessus, nous aurions pu ajouter le 11111111, c'est-à-dire –1, le premier nombre négatif.

Dans ce même contexte, le code C++ suivant :

```
// int_long.cpp
#include <iostream>

using namespace std;

int main() {
 int valeur = 2147483648;
 valeur++;

 cout << "Valeur: " << valeur << endl;
 return 0;
}
```

va compiler, mais en donnant un avertissement :

```
---------- Capture Output ----------
> "C:\MinGW\bin\g++.exe" -o int_long int_long.cpp
int_long.cpp: In function `int main()':
int_long.cpp:7: warning: this decimal constant is unsigned only in ISO C90
> Terminated with exit code 0.
```

L'ISO C90, identique à l'ANSI C89 de l'*American National Standards Institute* (ANSI), indique la norme utilisée pour le langage C.

À l'exécution, le résultat est au premier abord surprenant :

```
Valeur: -2147483647 (1000.0000.0000.0000.0000.0000.0000.0001)
```

car il correspond à –2147483648 + 1. Le `int` 2147483648 de l'affectation de la variable `valeur` était bien –2147483648.

Utilisons la calculatrice de Windows en mode scientifique afin de convertir des valeurs décimales en valeurs hexadécimales ou binaires, et de visualiser rapidement la position des bits 0 ou 1.

Contrairement à Java, il faut être beaucoup plus prudent en C++, car les compilateurs ne sont pas tout à fait semblables et les valeurs possibles des entiers (`int`) peuvent être différentes d'une machine à l'autre. La solution est d'utiliser un `long` pour être certain d'avoir une réserve suffisante. Dans le cas présent, c'est un peu particulier puisque le `int` a la dimension d'un `long` sur une machine 32 bits comme Windows. Nous sommes donc aussi dans les limites de capacité d'un `long`.

Encore une fois, si nous connaissons précisément dans quel domaine de valeurs nous travaillons, nous pouvons être plus précis :

```
// int_long2.cpp
#include <iostream>

using namespace std;
```

```
int main()
{
 unsigned long valeur = 2147483648ul;
 valeur++;

 cout << "Valeur: " << valeur << endl;
 return 0;
}
```

Le résultat est attendu. Le bit tout à gauche n'est plus la marque du signe positif (0) ou négatif (1) mais est utilisable pour cette valeur élevée :

```
Valeur: 2147483649
```

Nous avons indiqué clairement que nos valeurs n'étaient jamais négatives (`unsigned` et `ul`).

Transtypage

Le transtypage permet de convertir un type en un autre lorsque c'est nécessaire. C'est valable aussi bien en C++ qu'en Java, bien qu'il faille l'éviter dans la mesure du possible dans ce dernier langage. D'une manière générale, lors de l'écriture d'un programme, nous devrions connaître d'avance les valeurs extrêmes contenues dans les variables. Dans le code C++ suivant :

```
cout << "Valeur: " << (unsigned)valeur << endl;  // donnera 2147483649
cout << "Valeur: " << static_cast<unsigned>(valeur) << endl;
```

les deux instructions sont équivalentes, la deuxième forme étant celle utilisée en Standard C++.

En fait, nous pourrions ignorer les valeurs négatives et utiliser un `unsigned int`, qui nous permettrait de travailler avec des entiers positifs, mais sur une plage de valeurs différentes. Mais c'est loin d'être aussi évident lorsqu'il faut s'adapter aux fonctions ou méthodes mises à disposition par les bibliothèques des langages.

Positionnement des variables

Il y a plusieurs manières de positionner une variable. Pour simplifier, nous n'allons considérer ici que les variables de type primitif. Nous considérerons aussi plus loin les variables de classe et les variables globales.

Toute variable a une portée. En écrivant ce code en C++ :

```
#include <iostream>

using namespace std;

int main() {
  int valeur1 = 1;
  cout << "Valeur1: " << valeur1 << endl;
```

```
  {
    int valeur1 = 2;
    cout << "Valeur1: " << valeur1 << endl;
  }

  cout << "Valeur1: " << valeur1 << endl;
}
```

il nous donnera ce résultat :

```
Valeur1: 1
Valeur1: 2
Valeur1: 1
```

Nous voyons donc que la variable a uniquement une signification à l'intérieur d'un même corps défini entre deux accolades.

Si nous faisons de même en Java :

```
public class Portee {
   public static void main(String[] args) {
      int valeur1 = 1;
      {
        int valeur1 = 2;
      }
   }
}
```

le compilateur est plus restrictif et nous retournera :

```
javac Portee.java
Portee.java:5: Variable 'valeur1' is already defined in this method.
     int valeur1 = 2;
         ^
1 error
*** Error code 1
make: Fatal error: Command failed for target `Portee.class'
```

C'est plus propre, et il y a moins de risques d'erreur. Une même variable en Java ne peut être déclarée qu'une seule fois dans le bloc d'une méthode.

Il y a également d'autres cas où la définition des variables peut s'opérer de manière plus élégante. Lorsque nous avons ce type de code en C++ ou en Java (nous reviendrons au chapitre 3 sur les instructions de conditions et de boucles) :

```
int i = 0;
for (; i < 10; i++) {
  // ...
};
```

et que `i` n'est utilisé qu'à l'intérieur du corps de la boucle `for()`, il est judicieux de le coder autrement :

```
for (int i = 0; i < 10; i++) {
  // ...
};
```

dans le cas où la variable `i` ne serait plus utilisée à l'extérieur de la boucle. Pour le positionnement de la définition des variables, il est souvent préférable de les déplacer en début de corps :

```
int main() {
  int valeur1 = 1;
  int valeur2 = 2;
```

Si la variable `valeur2` n'est utilisée que beaucoup plus loin dans le corps de la fonction, nous pourrions argumenter pour un déplacement de la définition de `valeur2` juste avant son utilisation. Cependant, comme le code à l'intérieur d'une fonction doit se limiter à un certain nombre de lignes (autre recommandation), cette argumentation ne tient plus !

Variables du type pointeur en C++

En C ou en C++, il est possible de définir une variable comme un pointeur, c'est-à-dire une adresse mémoire. Cette dernière correspond à une adresse d'un autre objet existant ou à celle d'un objet alloué avec la fonction C `malloc()` ou l'opérateur `new`, sur lequel nous reviendrons au chapitre 4. Tous les objets créés avec le mot-clé `new` sont des pointeurs. Bien que Java crée des objets avec l'opérateur `new`, il ne possède pas de pointeurs.

Si nous écrivons ce morceau de code en C++ :

```
int nombre1 = 1;
int *pnombre2;
pnombre2 = &nombre1;
*pnombre2 = 2;
```

la variable `nombre1` contiendra initialement la valeur 1. La deuxième instruction définit une variable `pnombre2` de type pointeur sur un entier, ce qui se fait avec l'opérateur `*`. La troisième instruction permet de donner à la variable `pnombre2` l'adresse en mémoire de la variable `nombre1`, ceci avec l'opérateur `&`. Enfin, l'instruction :

```
*pnombre2 = 2;
```

attribue la valeur 2 à la variable entière qui pointe sur `pnombre2`, c'est-à-dire la variable `nombre1`. Si nous écrivons :

```
cout <<  nombre1 << *pnombre2;
```

nous obtiendrons 22 et non pas 12. Si nous entrons par erreur :

```
cout << pnombre2;
```

l'adresse actuelle de la mémoire où se trouve stocké le nombre 2 sera affichée !

Une variable de type pointeur nous permet d'associer plusieurs variables au même objet, qui se trouve à un endroit déterminé en mémoire.

Utilisation des pointeurs

N'associons jamais des variables de type pointeur à des objets qui disparaissent de leur portée comme ici :

```
int main()
{
 int *pcompteur;
 {
  int compteur = 20;
  pcompteur = &compteur;
  compteur = 30;
 }  // le compteur disparaît de sa portée
 *pcompteur = 40;
}
```

Ce code fonctionnera vraisemblablement, car la mémoire adressée par `pcompteur` ne sera sans doute pas encore réutilisée. Cependant, une erreur de ce type peut être difficile à déceler, et il faut donc se méfier. Le code qui se trouve à l'intérieur de la portée va perdre sa variable `compteur`, dont nous avons récupéré l'adresse ! Nous reviendrons sur la lettre `p` de notre variable `pcompteur` au chapitre 4 lorsque nous ferons une autre recommandation au sujet du nom des variables en C++.

Utilisation de malloc() en C++

Il ne faut pas utiliser les fonctions C `malloc()` et `free()` : elles constituent l'un des héritages du langage C que nous pouvons oublier sans grandes difficultés. En effet, les opérateurs `new` et `delete` du langage C++ les remplacent avantageusement.

```
#include <iostream>
#include <cstdlib>

using namespace std;

int main()
{
 int *mon_int;
 mon_int = (int *)malloc(sizeof(int));

 *mon_int = 10;

 cout << *mon_int << endl;
 free(mon_int);
}
```

Nous verrons au chapitre 4 comment utiliser l'opérateur `new`. La fonction C `sizeof()` nous permet de recevoir la dimension du type utilisé, afin d'allouer la mémoire correspondante.

Nous pouvons déterminer le nombre d'octets attribué à chaque type de cette manière :

```
 cout << "sizeof int: " << sizeof(int) << endl;
```

```
cout << "sizeof char: " << sizeof(char) << endl;
cout << "sizeof short: " << sizeof(short) << endl;
cout << "sizeof double: " << sizeof(double) << endl;
cout << "sizeof long: " << sizeof(long) << endl;
```

et qui nous donnera le résultat :

```
sizeof int: 4
sizeof char: 1
sizeof short: 2
sizeof double: 8
sizeof long: 4
```

Ce résultat peut être différent sur un autre système (par exemple Linux). Ce n'est pas le cas en Java où c'est une nécessité d'avoir la même dimension pour chaque type, puisque le code est exécutable sur différentes machines. Nous remarquons ici que, sous Windows, `int` et `long` ont la même dimension.

Variables constantes

Les variables constantes sont définies avec les mots-clés `const` en C++ et `final` en Java. Nous reviendrons plus tard sur leurs utilisations conjointement avec le mot-clé `static` dans le cadre des objets constants de classe.

```
const int constante1 = 10;  // C++
final int constante1 = 10;  // Java
```

L'emploi de variables constantes permet de définir des variables fixes qui ne seront jamais modifiées par le programme. Le compilateur C++ ou Java rejettera la compilation si une nouvelle valeur est affectée à une constante :

```
public class Constante {
   public static void main(String[] args) {
      final int valeur1 = 1;
      valeur1 = 2;
   }
}

javac Constante.java
Constante.java:4: Can't assign a value to a final variable:
➥valeur1
    valeur1 = 2;
    ^
1 error
```

Ces variables constantes peuvent être par exemple les dimensions d'un échiquier, qui resteront toujours de 8 sur 8. Des fonctions de recherche dans des tableaux pourraient être utilisées pour le jeu d'échecs, mais aussi pour d'autres jeux avec des tableaux plus grands. Dans ce dernier cas, elles fonctionneraient également après une nouvelle compilation, à condition d'avoir utilisé ces variables constantes et non des valeurs fixes dans le code.

Très souvent, une variable constante est utilisée pour déterminer des conditions internes qui ne devraient pas changer, ou bien pour tester des limites de système. Le DOS accepte le fameux format 8.3 pour les fichiers (exemple : `autoexec.bat`). Le code C++ suivant :

```
const int dim_nom = 8;
const int dim_ext = 3;
char  nom_fichier[dim_nom + 1];
char  ext_fichier[dim_ext + 1];
```

est tout à fait réalisable avec un +1 pour le 0 de la terminaison de la chaîne de caractères. Nous reviendrons sur la construction des tableaux et des chaînes de caractères au chapitre 5.

Les variables constantes pouvant être modifiées un jour ou l'autre, il est nécessaire de les regrouper dans un fichier d'en-tête (`.h`), si possible commun. Un changement de ces valeurs nécessiterait alors une nouvelle compilation. Les `#define` du C devraient être remplacés par des variables constantes, bien qu'ils soient nombreux dans les bibliothèques de ce langage. Ainsi, nous avons dans la bibliothèque C `math.h` la définition suivante :

```
#define M_PI            3.14159265358979323846
```

Celle-ci pourrait être avantageusement remplacée par :

```
const double m_pi 3.14159265358979323846;
```

Variables globales en C++

Il n'est pas possible de définir une variable globale en Java de la même manière qu'en C++. En Java, elle sera définie soit comme variable de classe, même statique, soit comme variable de travail à l'intérieur d'une méthode. Le code C++ qui suit n'est ainsi pas possible en Java :

```
int variable1 = 1;
const int variable2 = 2;
extern int variable3 = 3;

int main(int argc, char **argv) {
}
```

Nous voyons que ces variables sont définies en dehors de la portée de `main()`. Elles sont donc accessibles par toutes les fonctions du programme. C'est une manière classique de procéder, héritage du langage C, qui n'a pas d'autres mécanismes à sa disposition, tel celui de définir des variables de classe constantes.

Le mot-clé `extern` permet à des modules séparés d'accéder à leurs variables. Si nous compilons un programme de cette manière :

```
gcc -o programme.exe module1.cpp module2.cpp
```

une variable `variable1` définie dans le `module1.cpp` pourra être utilisée dans le `module2.cpp` si elle est déclarée `extern` dans le `module2.cpp`. Sans la déclaration en `extern` de cette variable

ou son existence dans un fichier d'en-tête, le compilateur indiquerait qu'il ne trouve pas de déclaration ou de référence à cette variable.

Dans l'exemple ci-dessus, les trois variables seront initialisées avant que le code du `main()` ne soit exécuté.

Dans la mesure du possible, il faut éviter ce genre de construction distribuée dans plusieurs modules. Une alternative est de regrouper toutes ces variables dans un fichier d'en-tête `.h`. Nous donnons cependant la préférence à des variables de classe statiques, que nous étudierons au chapitre 10.

Fichiers d'en-tête en C++

Nous verrons au chapitre 4 que les fichiers d'en-tête sont principalement utilisés pour la définition des classes C++. Comme pour les fichiers `.cpp`, ce sont des fichiers ASCII avec généralement l'extension `.h`. Ils contiennent des définitions, par exemple des constantes, qui peuvent être utilisées par différentes sources `.cpp`. Nous apprendrons au chapitre 6 qu'ils peuvent aussi être développés pour nos propres bibliothèques de classe C++ ou de fonctions C. Dans ce cas-là, le compilateur devra utiliser la directive **–I** pour identifier la localisation de ces fichiers dans un répertoire prédéfini. Pour l'instant, il nous suffit de savoir que le compilateur sait identifier par lui-même l'emplacement des fichiers d'en-tête du système, par exemple `iostream` ou `cmath`.

Opérations et opérateurs

Après nous être familiarisés à la déclaration et à l'affectation des variables numériques, nous allons étudier les opérateurs mis à disposition par les deux langages, ainsi que certaines fonctions faisant partie des différentes bibliothèques C, C++ ou Java.

Les microprocesseurs, ou plus généralement les processeurs des ordinateurs, sont capables aujourd'hui de calculer différentes fonctions mathématiques. Il y a vingt ou trente ans, une simple multiplication n'était pas encore possible, et tout devait être programmé par le logiciel. Mais ce n'est pas très important en première analyse. Si nous écrivons en C++ :

```
// math.cpp
#include <iostream>
#include <cmath>

using namespace std;

int main()
{
 double a = 2.0;
 double b = 3.0;
 double c = a * b;
 cout << "Multiplication: " << c << endl;
```

```
 c = sin(a)*cos(b) + sin(b)*cos(a);  // sin(a + b)
 cout << "Sinus(a + b): " << c << endl;

 c = pow(a, b);
 cout << "a à la puissance b: " << c << endl;

 return 0;
}
```

qui fournit comme résultat :

```
Multiplication: 6
Sinus(a + b): -0.958924
a à la puissance b: 8
```

Nous découvrons toute une série d'opérations. Nous constatons que nous avons à la fois des opérateurs et des fonctions de la bibliothèque du langage C (`<math.h>`). `<cmath>` est le fichier d'en-tête à utiliser pour la bibliothèque du Standard C++ ; `<math.h>` est en fait inclus dans `<cmath>`. Le point important est de savoir où trouver ces fonctions (qui sont ici `sin()`, `cos()` et `pow()`) : dans un manuel de référence, un outil de développement sous Windows ou des pages de manuel (`man`) sous Linux. Cependant, nous pouvons toujours lire directement le fichier `math.h`, qui se trouve, par exemple pour l'installation de GNU, dans le répertoire d'installation du compilateur :

```
..\Mingw32\i386-mingw32\include\math.h
```

et où nous pourrions trouver ceci :

```
extern double atan _PARAMS((double));
extern double cos _PARAMS((double));
extern double sin _PARAMS((double));
extern double tan _PARAMS((double));
extern double tanh _PARAMS((double));

extern double log _PARAMS((double));
extern double log10 _PARAMS((double));
extern double pow _PARAMS((double, double));
extern double sqrt _PARAMS((double));
```

ce qui est aussi une manière de vérifier les paramètres que nous passons à la fonction. Nous reviendrons plus loin sur la définition et l'utilisation des fonctions C++ ou des méthodes en Java. Pour l'instant, nous oublierons le `_PARAMS` et nous lirons simplement un `double cos(double)`. Cela nous indique qu'une variable de type `double` est demandée par la fonction `cos()` et que le résultat nous est retourné comme un `double`.

La classe Java Math

La classe Java `Math` possède un grand nombre de fonctions mathématiques utiles aux scientifiques, économistes ou autres statisticiens. En C++, nous utilisons des fonctions C.

Ici, nous allons travailler avec des variables ou méthodes statiques, ce qui est en fait équivalent aux fonctions C.

Voici une liste non exhaustive de quelques méthodes de la classe `Math` :

```
public static final double PI;
public static double sin(double a);
public static double cos(double a);
public static double pow(double a, double b);
public static long round(double a);
```

ainsi qu'un programme en Java similaire au précédent en C++ :

```
public class MathTest {
   public static void main(String[] args) {
     double a = 2.0;
     double b = 3.0;
     double c = a * b;

     System.out.println("Multiplication: " + c);

     c = Math.sin(a)*Math.cos(b) + Math.sin(b)*Math.cos(a);  // sin(a + b)
     System.out.println("Sinus(a + b): " + c);

     c = Math.pow(a, b);
     System.out.println("a à la puissance b: " + c);
   }
}
```

Le résultat peut paraître surprenant :

```
Multiplication: 6.0
Sinus(a + b): -0.9589242746631385
a à la puissance b: 8.0
```

car il nous donne une meilleure précision qu'en C++. Si nous voulions obtenir le même résultat en C++, il faudrait ajouter les deux lignes suivantes à l'endroit correct :

```
#include <iomanip>
cout << "Sinus(a + b): " << setprecision(16) << c << endl;
```

Le `setprecision(16)` est envoyé au `cout` (le canal de sortie) avant la variable `double c` pour lui indiquer que nous aimerions plus de chiffres après la virgule (6 étant le défaut de `cout`). Le dernier chiffre décimal est aussi différent, mais c'est vraisemblablement dû à la manière dont le `double` est arrondi.

Les opérateurs traditionnels

À propos des opérateurs arithmétiques traditionnels (`+`, `-`, `*` et `/`), il faut revenir sur l'opérateur de division (`/`) et son compagnon, le reste de la division (`%`). Si nous divisons

un nombre en virgule flottante et que nous restons dans ce type, il est évident que la division se fera convenablement.

```
double  d1 = 5;
double  d2 = 2;
int     i1 = 5;
int     i2 = 2;

double  d3 = d1/d2;   // 2.5
int     i3 = i1/i2;   // 2
int     i4 = i1%i2;   // 1

d3 = d1%d2;  // erreur de compilation
```

Ces instructions sont communément utilisées, mais chacune dans leur domaine particulier. Pour compter une population ou pour en faire des moyennes, les entiers peuvent être utilisés. Pour des applications bancaires ou statistiques précises, les `double` sont nécessaires.

Char et byte

Un programmeur C ou C++ qui entre dans le monde Java sera surpris en découvrant le `char` de Java à 16 bits. En C++, une déclaration telle que :

```
char mon_char = 'c';
```

qui est aussi correcte en Java, représente un bon vieux caractère de 8 bits parmi le jeu des caractères ASCII. Si nous retournons un peu plus en arrière, nous nous souviendrons même des débuts des communications sur modem, quand nous ne transmettions que 7 bits aux États-Unis, car nos voisins outre-atlantiques n'ont pas de caractères accentués. De très anciens programmeurs se rappelleront aussi des difficultés avec le bit 7 (à gauche), où il fallait parfois écrire en C un :

```
unsigned char mon_char;
```

afin d'utiliser certaines procédures de la bibliothèque C qui ne fonctionnaient pas correctement. `unsigned` veut dire ici que `mon_char` peut être considéré comme un nombre entre 0 et 255, et non pas entre -128 et 127. Si, en Java, nous voulons absolument travailler avec une variable à 8 bits, nous utiliserons le *byte* (octet) :

```
byte mon_byte = (byte)'b';
```

Le transtypage est nécessaire, sinon le compilateur retourne une erreur.

Mais passons sur les détails et ne soyons pas trop surpris avec la représentation Java à 16 bits, à savoir l' « **Unicode** » dont le site Web, `http://www.unicode.org`, nous donnera plus de détails sur un travail de définition qui se poursuit encore. Un caractère **Unicode** permet de représenter une lettre dans n'importe quel langage : 65 536 possibilités nous sont ainsi offertes pour coder toutes les lettres de tous les alphabets de la planète. Nous pouvons déjà mentionner que le `String` de Java est une suite de caractères **Unicode** de 16 bits.

En C++, nous n'avons donc que 8 bits. Le jeu de caractères ASCII/ANSI est codé sur un octet et n'offre que 255 combinaisons. Les 255 caractères utilisés sous Windows sont en fait les 255 premiers caractères de l'**Unicode**. Une notation hexadécimale permet de spécifier un caractère particulier qui ne peut être sélectionné au clavier. L'instruction suivante en Java :

```
char unicar = '\u0044';  // au lieu de = 'D';
```

n'a pas vraiment de sens puisqu'elle correspond à la lettre majuscule D. Terminons par la table des caractères spéciaux que nous utilisons régulièrement en C++ et en Java :

Code d'échappement	Unicode en Java	Description
\b	\u0008	Effacement arrière
\t	\u0009	Tabulation horizontale
\n	\u000A	Saut de ligne
\r	\u000D	Retour de chariot
\"	\u0022	Guillemet
\'	\u0027	Apostrophe
\\	\u005C	Antislash
\f	\u000C	Saut de page
\DDD	\u00HH	Caractère octal (ex: '\101' = 'a')
	\uHHHH	Caractère unicode

Nous définirons volontiers, en C++, une variable constante pour représenter un de ces caractères :

```
const char nl = '\n';
```

Intervalles des types entiers en Java

Les types entiers ont des intervalles fixes en Java :

- **byte** : de –128 à 127 inclus ;
- **short** : de –32 768 à 32 767 inclus ;
- **int** : de –2 147 483 648 à 2 147 483 647 inclus ;
- **long** : de –9 223 372 036 854 775 808 à 9 223 372 036 854 775 807 inclus ;
- **char** : de '\u0000' à '\uffff' inclus, c'est-à-dire de 0 à 65 535.

Ceci est une obligation pour garantir la portabilité entre machines.

En C++, `int` peut être défini sur 16 ou 32 bits. Cela dépendra à la fois de la machine et du système d'exploitation. Lorsque le programmeur a des doutes sur la capacité d'un `int`, il peut toujours choisir `long` ou `double` comme alternative.

Règles de priorité des opérateurs

Dans ce chapitre, il faut mentionner les règles de priorité nécessaires en mathématiques et qui existent en général dans tous les langages. En Java et C++, une instruction telle que :

```
7*3+2
```

nous donnera 23 et non pas 35 : la multiplication a la priorité sur l'addition. Les opérateurs `*` ou `/` ont la priorité sur l'addition et la soustraction. Bien que les programmeurs connaissent ces règles, il est tout de même plus élégant et compréhensible d'écrire ce code comme suit :

```
(7*3)+2
```

Cette écriture est propre et n'entraînera aucune différence à l'exécution. Cependant, nous pouvons forcer l'ordre des opérations et écrire :

```
7*(3+2)
```

Dans la réalité, ces nombres seront des variables.

Une diversion sur le cin (entrée)

Comme nous l'avons affirmé dans la préface, il est difficile d'élaborer des exercices sans utiliser un certain nombre de concepts qui seront présentés plus loin dans cet ouvrage. Afin que nos petits exercices tiennent la route, il faut tout de même pouvoir entrer des données *via* la console. En C++, le travail se fait d'une manière relativement simple :

```
// cin.cpp
#include <iostream>
#include <string>

using namespace std;

int main()
{
  string texte = "";
  int nombre = 0;

  cout << "Entre un texte: ";
  cin  >> texte;
  cout << "\nLe texte: " << texte << endl;
  cout << "Entre un nombre: ";
  cin  >> nombre;
  cout << "\nLe nombre: " << nombre << endl;
}
```

Nous dirons simplement que le `cin` est l'opposé du `cout` : il permet d'obtenir des données entrées par l'opérateur sur la console. Nous y reviendrons plus en détail dans le chapitre consacré aux entrées et sorties. De plus, l'opérateur >> est capable de gérer différents types d'objets et de stocker l'information correctement. Si, dans le programme ci-dessus, nous entrons un texte pour le nombre, un 0 sera transféré dans la variable nombre. Le `cin` est en fait directement lié au `stdin` du langage C qui correspond au flux d'entrée.

Au sujet du `string`, nous devons pour l'instant le considérer comme un objet capable de stocker une chaîne de caractères.

En passant en Java, cela se gâte :

```
import java.io.*;

public class Cin {
  public static void main(String[] args) {
    try {
      String texte = "";
      int nombre = 0;

      BufferedReader stdin = new BufferedReader(new InputStreamReader(System.in));

      System.out.print("Entre un texte: ");
      texte = stdin.readLine();
      System.out.println("\nLe texte: " + texte);

      System.out.print("Entre un nombre: ");
      nombre = Integer.parseInt(stdin.readLine());
      System.out.println("\nLe nombre: " + nombre);
    }
    catch(IOException ioe) {}
  }
}
```

Ce n'est pas encore possible ici de passer au travers de toutes ces nouvelles fonctionnalités. Disons simplement que les classes `BufferedReader` et `InputStreamReader` sont définies dans le paquet `java.io`. Par analogie au C++, nous dirons que l'instruction :

```
BufferedReader stdin = new BufferedReader(new InputStreamReader(System.in));
```

correspond au `cin` du langage C++. Les deux instructions suivantes :

```
texte  = stdin.readLine();
nombre = Integer.parseInt(stdin.readLine());
```

sont en fait équivalentes au >> du C++ sur un `string` et sur un entier. Nous verrons que `parseInt()` est une méthode statique similaire à une fonction C pour réaliser la conversion nécessaire. Si le nombre était un `double`, il faudrait écrire :

```
double fortune = 0;
fortune = Double.parseDouble(stdin.readLine());
```

Nous oublierons le `try` et le `catch` pour l'instant, mais nous pouvons aussi essayer d'entrer des lettres à la place du nombre pour constater le problème suivant :

```
Entre un texte:
Le texte: aaa
Entre un nombre: java.lang.NumberFormatException: aaa
        at java.lang.Integer.parseInt(Compiled Code)
        at java.lang.Integer.parseInt(Integer.java:458)
        at Cin.main(Cin.java:16)
Exception in thread "main"
```

Nous verrons comment traiter et régler ce problème plus loin lorsque nous aborderons les exceptions.

Les opérateurs d'affectation composés

Un certain nombre d'opérateurs composés sont applicables dans les deux langages, Java et C++. Leur présentation se fera au travers de quelques exemples.

Les trois instructions suivantes sont équivalentes :

```
nombre = nombre + 1;
nombre += 1;
nombre++;
```

La deuxième forme est généralement préférée à la première pour une incrémentation différente de 1 ; pour une incrémentation égale à 1, la troisième serait de mise. Ces formes simplifient aussi bien l'écriture que la lecture.

Quant aux deux instructions :

```
nombre--;     //préférable
nombre -= 1;
```

elles sont aussi identiques.

Nous avons également la possibilité de combiner des opérations comme suit :

```
int nombre = 4;
nombre *= 50;    // devient 200
nombre /= 49;    // devient 4 et non 4.0816
nombre %= 3;     // devient 1
```

Ces opérations sont tout à fait correctes, mais rarement utilisées.

Mais revenons sur les opérateurs `++` et `--`, que l'on retrouve très souvent et qui peuvent aussi précéder la variable :

```
int nombre1 = 1;
int nombre2 = nombre1++;   // nombre2 = 1   nombre1 = 2
nombre2 = ++nombre1;       // nombre2 = 3   nombre1 = 3
```

Dans la deuxième instruction, l'opération ++ se fera avant d'assigner la valeur du `nombre1` à la variable `nombre2`. Dans la dernière, c'est l'inverse : nous transférons `nombre1` dans `nombre2` et ensuite nous incrémentons `nombre2`.

Mais que se passe-t-il avec :

```
--nombre2 *= ++nombre2;
```

Bien difficile de le savoir. Il faut donc essayer !

```
#include <iostream>

using namespace std;

int main()
{
 int nombre2 = 0;
 cin >> nombre2;
 --nombre2 *= ++nombre2;
 cout << "nombre2: " << nombre2 << endl;
 return 0;
}
```

On se rendra compte que le résultat est simplement :

```
nombre2 = nombre2 * nombre2;
```

car les opérations ++ et -- ont la préséance sur *=.

En Java, si nous essayons de compiler une telle forme, nous obtiendrons :

```
Quiz.java:17: Invalid left hand side of assignment.
      (--nombre2) *= (++nombre2);
       ^
1 error
```

En Java, des opérateurs tels que -- ne peuvent être exécutés sur la partie gauche de l'instruction, mais seulement à droite. Le compilateur est plus restrictif que celui du C++, et il a raison de rejeter cette forme. L'instruction suivante en C++ :

```
un_nombre++ = 2;
```

est acceptée. Elle n'a en fait aucun sens, car la valeur 2 est de toute manière affectée à `un_nombre`, et ceci après l'opération ++. Il faudrait donc garder une forme simple et penser aux lecteurs et correcteurs potentiels de ce code.

Les opérations binaires

Les opérateurs binaires permettent de travailler directement sur les bits qui composent des nombres entiers. Les opérateurs binaires en C++ et Java sont les suivants : `&` (et), `¦` (ou), `^` (ou exclusif), `~` (non), `>>` (déplacement vers la droite) et `<<` (déplacement vers la

gauche). Nous allons étudier à présent quelques opérations binaires que les programmeurs utilisent, par exemple, dans des applications liées aux circuits logiques en électronique.

Imaginons un carrefour de circulation routière comportant huit feux de signalisation, qui pourront être rouges ou verts. Nous n'allons pas considérer le passage à l'orange, que nous pourrions traiter comme une transition. Définissons les bits comme suit :

- 0 : signal au vert ;
- 1 : signal au rouge.

En outre, nous n'allons pas définir huit variables, mais une seule contenant 8 bits.

Nous utiliserons les types `short int` en C++ et `byte` en Java.

La combinaison suivante **11111110** signifie donc que le premier feu (bit 0) est au vert et tous les autres au rouge. De la même manière, avec la combinaison **01111111**, le bit de gauche (bit 7) nous indique que le huitième feu est au vert.

Nous allons à présent effectuer quelques opérations sur ces feux de signalisation, tout d'abord en C++ :

```
// feux.cpp
#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
 short int feux  = 0xFE;
 short int nfeux = feux;

 cout << "Nos feux: " << hex << nfeux << endl;

 nfeux = feux | 0x01;    // (A)
 cout << "Nos feux: " << hex << nfeux << endl;
 nfeux = feux ^ 0x03;    // (B)
 cout << "Nos feux: " << hex << nfeux << endl;
 nfeux = (~feux) & 0xFF; // (C)
 cout << "Nos feux: " << hex << nfeux << endl;

 cout << "Le feu: " << hex << (feux & 0xA0) << endl;  // (D)
 return 0;
}
```

Nous utilisons ici de nouvelles formes **0xFE** ou **0x01**, qui sont une manière en C++ et en Java de représenter des nombres dans une forme hexadécimale. Un nombre hexadécimal est une représentation d'un nombre en base 16 (de 0 à 15) en utilisant la notation suivante : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E et F. Les lettres minuscules (de a à f) sont aussi acceptées. Un nombre hexadécimal est en fait un nombre binaire de 4 bits (de 0000 à 1111). Le nombre 0xFE représente donc 11111110, c'est-à-dire tous les feux allumés (au rouge) sauf le

premier. Le nombre 0x01 (00000001) nous indique que seul le premier feu est au rouge. `nfeux` est une copie de la valeur d'origine.

Le tableau ci-dessous devrait vous aider à comprendre les explications qui vont suivre.

OR, **XOR** et **AND** sont des opérateurs binaires qui agissent sur deux composants. **NOT**, en revanche, est un opérateur unaire. Les deux tableaux suivants (les tables de vérité et un exemple) devraient nous aider :

OR	0	1	XOR	0	1	AND	0	1
0	0	1	**0**	0	1	**0**	0	0
1	1	1	**1**	1	0	**1**	0	1

Composant 1	Composant 2	Opérateur	Résultat
0110	1100	OR	1110
0110	1100	XOR	1010
0110	1100	AND	0100
0110		NOT	1001

Revenons au code précédent :

- Ligne (A) : l'opérateur | (**OR – ou inclusif**) permet d'ajouter des bits, qu'ils soient ou non déjà positionnés. Le premier feu 0x01 (00000001) deviendra rouge, même s'il l'était déjà.
- Ligne (B) : l'opérateur ^ (**XOR – ou exclusif)** va mettre les bits à 1 (rouge) s'ils sont différents. Avec 0x03 (00000011), nous allons mettre au rouge le premier feu, alors que le deuxième deviendra vert. Le résultat sera 0XFD, c'est-à-dire 11111101.
- Ligne (C) : l'opérateur ~ **(NOT**) va nous inverser tous les feux. Malheureusement, l'opération d'inversion va aussi toucher les autres bits du `short int`, et il est alors nécessaire de masquer le résultat avec l'opérateur `&` (**AND**) sur les 8 bits qui nous intéressent (0xFF, 11111111).
- Ligne (D) : cette dernière opération (**AND – et**) permet de masquer un ou plusieurs bits afin de nous indiquer s'ils sont ou non positionnés. Le cas 0xA0 (10100000) pourrait être utilisé, par exemple pour contrôler des feux de passages piétons.

Nous allons passer à un exemple beaucoup plus bref en Java, car la représentation de nombres de ce type n'est pas aussi aisée qu'en C++.

```
public class Feux {
   public static void main(String[] args) {
      byte feux = 0x03;  // 00000011
      System.out.println("Feux:  " + Integer.toBinaryString(feux));
      feux <<= 2;        // 00001100
      System.out.println("Feux:  " + Integer.toBinaryString(feux));
```

```
        feux <<= 4;         // 01000000 + négatif
        System.out.println("Feux:  " + Integer.toBinaryString(feux));
        feux >>= 6;         // 00000001 + négatif
        System.out.println("Feux:  " + Integer.toBinaryString(feux));

        int ifeux = 0x03;  // 00000011
        ifeux >>= 1;        // 00000001
        System.out.println("IFeux: " + Integer.toBinaryString(ifeux));
    }
}
```

`Integer.toBinaryString()` est une méthode statique, dont nous comprendrons plus tard le mécanisme, qui nous permet d'obtenir une représentation binaire du résultat. En exécutant ce code, nous obtiendrons :

```
Feux:  11
Feux:  1100
Feux:  11111111111111111111111111000000
Feux:  11111111111111111111111111111111
IFeux: 1
```

Ici, nous avons volontairement laissé de côté les opérateurs OR, XOR et AND pour utiliser les opérateurs de décalage. Le premier décalage à gauche se fait avec l'opérateur `<<=`, qui est en fait équivalent à :

```
feux = feux << 2;
```

qui est d'une lecture plus simple. Nous aurions donc les deux premiers feux (0 et 1) qui passeraient au vert et les deux suivants (2 et 3), au rouge. Le décalage suivant de 4 bits nous donne une première difficulté. Le décalage sur le bit 8 avec l'opérateur <<= rend la variable négative et perd en fait le bit. La raison de ces nombreux bits 1 vient d'`Integer.toBinaryString( )`, qui nous donne une représentation sur un `Integer` (4×8 bits) alors que nous avions un `byte` sur 16 bits.

En décalant à nouveau à droite, nous n'allons pas récupérer le bit perdu, et la variable restera négative. Le dernier exemple nous montre que le bit 0 est perdu en décalant à droite et n'est pas récupéré sur la droite. Il faut enfin noter que si nous écrivons :

```
ifeux = 0xFF;   //255
```

nous avons bien la valeur de 255 (11111111). Si nous voulions faire des rotations de feux de signalisation il faudrait définir une autre logique et revenir certainement à nos opérateurs logiques **OR**, **XOR** et **AND**.

Typedef et énumération en C++

Le mot-clé `typedef` permet en C++ de définir de nouveaux types de données, ce qui n'est pas possible en Java. Si nous écrivons :

```
#include <iostream>
```

```
using namespace std;

int main() {
  typedef int Entier;

  Entier ent = 1;
  cout << ent << endl;
}
```

nous définissons `Entier` comme un nouveau type. Il s'agit en fait simplement d'un nouveau synonyme (alias) qui remplace `int` et qui s'avère vraiment inutile dans ce cas précis. Le mot-clé `typedef` n'a de sens que pour simplifier des déclarations parfois compliquées. En conséquence, un listing de code de programme pour Windows est souvent surprenant :

```
WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int) {
}
```

car il donne en fait l'impression de travailler avec un autre langage, ce qui demande donc un certain temps avant de se familiariser. La définition de **LPSTR**, par exemple, tirée du Visual C++ de Microsoft, est la suivante : *Pointer to a null-terminated string of 8-bit Windows (ANSI) characters*. En fait, ce n'est rien de plus qu'un banal `char *`, une chaîne de caractères que nous étudierons au chapitre 5.

Les énumérations en Java sont enfin disponibles depuis la version 1.5 du JDK et nous y reviendrons au chapitre 6. Nous verrons aussi une autre manière de faire, en groupant des constantes dans une interface, à la fin du chapitre 13.

En C++, une énumération permet de décrire, sous un même chapeau d'un nom défini avec le mot-clé `enum`, une liste de noms qui seront affectés avec une valeur prédéfinie. Dans les deux exemples qui suivent :

```
enum position1 { vide, blanc, noir, exterieur };
enum position {
  vide = 0, blanc = 1, noir = 2, exterieur = -1
};
```

la seule différence portera sur le deuxième `exterieur`, qui aura pour valeur –1, au lieu de la valeur 3 allouée automatiquement par le compilateur. Ce dernier commencera toujours par la valeur 0, si elle n'est pas choisie explicitement. Cette énumération pourrait être avantageusement utilisée dans le jeu d'Othello, que nous rencontrerons à plusieurs occasions dans cet ouvrage.

Naturellement, il aurait été tout à fait possible d'écrire ceci :

```
const int vide = 0;
const int blanc = 1;
const int noir = 2;
const exterieur = -1;
```

Mais si nous écrivons à présent :

```
position pos = vide;
pos = 2;
```

le compilateur refusera cette dernière construction, qu'il faudrait remplacer par :

```
pos = noir;
```

Cette manière de définir des variables peut donc avoir certains avantages évidents.

Résumé

Dans ce long chapitre, nous avons donc abordé la manipulation des variables numériques et de quelques opérations arithmétiques et binaires, tout en mettant l'accent sur les écueils et les erreurs grossières de programmation à éviter. Ces opérations devraient permettre aux programmeurs d'élaborer des calculs complexes et variés.

Exercices

Tous les exercices présentés dans cet ouvrage seront programmés dans les deux langages.

1. Quelles instructions du langage devons-nous utiliser, sans passer par des fonctions de bibliothèque, pour arrondir volontairement un nombre décimal de la manière suivante : par exemple, 5,500 est arrondi à 5 et 5,501 à 6 ?
2. Afficher la valeur de π !
3. Calculer le reste d'une division par 2 sur un entier, sans utiliser les opérateurs `/` et `%`, mais en travaillant sur la valeur binaire.
4. Écrire un programme qui va produire une division par 0 et constater les effets.
5. Je possède à la banque une fortune de 12 245,20 € qui me rapporte 5,25 % d'intérêts mais sur lesquels je dois payer 15 % d'impôt sur le rendement et 0,02 % sur la fortune. Écrire le programme pour qu'il accepte ces données de la console et donner un ensemble de valeurs significatives pour tester la justesse du programme.
6. Reprendre le code `int_long2.cpp` et écrire un nouveau code intitulé `int_long3.cpp` qui multiplie par 2 le résultat final. Expliquer la valeur retournée : étrange, mais correcte.

3

Et si on contrôlait l'exécution ?

Contrôler l'exécution à l'aide d'instructions conditionnelles est la clé de la programmation. Que ce soit pour tester des variables ou des résultats de fonctions avec l'instruction `if` ou encore répéter une opération avec `for`, le principe est le même en Java comme en C++. Tout d'abord, nous allons dire un mot sur la forme et la présentation du code, c'est-à-dire à l'endroit où les instructions devraient apparaître dans le texte, ainsi que la position des accolades pour du code regroupé dans des blocs conditionnels.

Recommandations pour la forme

Ces recommandations ne sont pas nécessairement des règles à suivre, mais plutôt des conseils sur la syntaxe à adopter. Il suffit souvent de recevoir du code de l'extérieur, d'autres programmeurs, pour se rendre compte que c'est un aspect capital. Un code mal écrit et mal documenté donne toujours une mauvaise impression et entraîne souvent une perte de temps lorsqu'il s'agit de le comprendre, en vue de le corriger, de le modifier ou de le perfectionner. C'est une sorte de frustration. Certes, nous aurions pu ou dû accepter un consensus préalable, car il y a souvent des divergences entre les programmeurs. La plupart des entreprises ou sociétés informatiques définissent des *Design Rules* (règles de conception) afin de pallier ces divergences d'approche, d'écriture et de style entre les auteurs des programmes. Il n'y a alors pratiquement plus de discussion sur le sujet, sauf sur des aspects plus subjectifs comme les commentaires dans le code. Nous allons simplement commencer par un exemple en C++ :

```
int main(int argc, char **argv) {
  int variable = 0;
```

```
variable = .....;  // (appel de fonction)

// commentaire

if (variable == 0) {  // commentaire court
  for (int i = 0; i < 10; i++) {
    if (i == 5) {
      ...
    }

    // commentaire
    ......
  }
}
```

En examinant ce morceau de code, voici ce que nous préconisons :

- Le corps du code principal (`main()`) doit apparaître avec un ou deux espaces après le début de la ligne.
- Seulement un espace entre les différentes parties des instructions.
- Jamais d'espace après les parenthèses ouvrante et fermante `()`, les virgules `,` ou les points-virgules `;`.
- Deux espaces avant les commentaires courts qui commencent avec des `//`.
- Deux espaces ou plus pour les nouveaux blocs définis avec deux accolades. La première accolade doit se trouver sur la première ligne du bloc et la deuxième doit être alignée verticalement avec le début de l'instruction. Autre alternative qui apparaît plus souvent en C++ :

```
if (variable == 0)
{
   ....
}
```

- Pas de tabulateurs dans le code source. Certains éditeurs de texte acceptent les deux, mais c'est souvent catastrophique lors de l'impression sur papier ou du transfert dans d'autres traitements de texte (conversion du TAB en huit espaces, par exemple).
- Un alignement des variables et de la documentation donnera plus d'allure aux programmes :

```
int    nombre;      // texte
double mon_compte;  // texte
```

Opérateurs de condition

L'instruction `if` (si) permet de tester une condition et d'accepter une ou plusieurs instructions (dans un corps de bloc délimité par des `{}`) si cette condition est remplie. L'instruction

`else` (sinon) est utilisée, si nécessaire, pour ajouter d'autres instructions lorsque la première condition n'est pas acceptée.

Nous allons prendre un exemple en C++ de deux nombres entiers entrés par l'utilisateur. Le programme nous retournera une indication de comparaison. Si la première condition n'est pas valable, alors la deuxième partie du code sera exécutée.

```
// iftest.cpp
#include <iostream>

using namespace std;

int main(int argc, char **argv) {
  int nombre1 = 0;
  int nombre2 = 0;

  cout << "Entre un premier nombre: ";
  cin >> nombre1;
  cout << "Entre un deuxième nombre: ";
  cin >> nombre2;

  if (nombre1 > nombre2) {
    cout << nombre1 << " est plus grand que " << nombre2 << endl;
  }
  else {
    cout << nombre1 << " est plus petit ou égal à " << nombre2 << endl;
 }

 return 0;
}
```

En Java, nous ajouterons un test d'égalité en plus.

```
import java.io.*;

public class IfTest {
  public static void main(String[] args) {
    try {
      int nombre1 = 0;
      int nombre2 = 0;

      BufferedReader stdin = new BufferedReader(new InputStreamReader(System.in));

      System.out.print("Entre un premier nombre: ");
      nombre1 = Integer.parseInt(stdin.readLine());
      System.out.println(nombre1);
      System.out.print("Entre un second nombre: ");
      nombre2 = Integer.parseInt(stdin.readLine());
      System.out.println(nombre2);

      if (nombre1 > nombre2) {
        System.out.println(nombre1 + " est plus grand que " + nombre2);
```

```
        }
        else {
          if (nombre1 < nombre2) {
            System.out.println(nombre1 + " est plus petit que " + nombre2);
          }
          else {
            System.out.println(nombre1 + " est egal a " +  nombre2);
          }
        }
      }
      catch(IOException ioe) {}
    }
}
```

L'instruction `if` et son opérateur `>` :

```
if (nombre1 > nombre2) {
```

signifie que nous testons si `nombre1` est plus grand que `nombre2`. Il y a d'autres séquences possibles comme `<` (inférieur à), `>=` (supérieur ou égal à), `<=` (inférieur ou égal à) ou encore `!=` (non égal à, différent de). L'instruction `if` ci-dessus est équivalente à :

```
if (nombre2 <= nombre1) {
```

c'est-à-dire « si le `nombre2` est inférieur ou égal au `nombre1` »…

Et si c'était faux (False)

Lorsqu'en C++ nous désirons tester l'égalité entre deux variables ou entre une variable et une valeur constante, il peut nous arriver d'écrire l'instruction suivante :

```
if (variable = constante) {
```

alors que nous voulions en fait écrire :

```
if (variable == constante) {
```

En revanche, en Java nous aurons :

```
Mif.java:5: Incompatible type for if. Can't convert int to boolean.
     if (variable = constante) {
     ^
1 error
```

Ici, `variable` est un `int`, et le résultat est un `int`. Java n'accepte que des variables de type `boolean` pour l'instruction `if`, et c'est une bonne chose. En fait, le programmeur voulait certainement utiliser l'opérateur `==` ! En Java, nous pouvons utiliser des variables de type `boolean` d'une manière beaucoup plus systématique, puisque uniquement des booléens sont acceptés pour les tests de condition. Le code Java suivant correspond à des situations que nous devrions rencontrer régulièrement :

```
boolean vrai = true;
//... code qui peut modifier la variable vrai
if (vrai) {
```

```
  System.out.println("C'est vrai");
}
```

Une autre construction composée possible et assez courante en C++ est celle-ci :

```
if ((variable = fonction()) < 0) {
```

L'affectation de la variable par l'appel d'une fonction, ainsi que le test, se fait dans la même instruction. Nous pensons cependant que la forme suivante est préférable :

```
variable = fonction();
if (variable < 0) {
```

Une autre forme encore plus simplifiée est aussi utilisée en C++ :

```
if (variable) {
```

Les règles de l'opérateur `if` en C++ s'appliquent sur la valeur de la variable. Si la valeur de la variable est 0, le code ne sera pas exécuté ; sinon, dans tous les autres cas, le code sera exécuté, même si ce n'est pas forcément 1. Un commentaire tel que :

```
int resultat = 0;  // initialise le résultat à faux (false)
```

est donc tout à fait justifié. Malheureusement, la plupart des fonctions C utilisent le retour de fonction avec un 0 pour indiquer que l'opération ou le test a passé, alors qu'une valeur négative, généralement –1, est utilisée pour indiquer une erreur.

L'opérateur logique NOT

Il peut s'appliquer à tous les opérateurs logiques que nous venons de voir dans les deux langages. Une opération telle que :

```
if (i !> 2) {    //identique à <=
```

signifie « si `i` n'est pas plus grand que 2 ». Cette forme sera prise en compte suivant la condition à tester et peut améliorer la lecture. Nous verrons qu'elle est souvent utilisée lorsque plusieurs conditions doivent être combinées. Les codes C++ et Java suivants sont suffisamment explicites et ne demandent pas d'analyse particulière :

```
int i = 1;
if (!(i == 2)) {
  cout << "i n'est pas égal à 2" << endl;
}

int i = 0;
if (!(i == 2)) {
  System.out.println("i n'est pas égal à 2");
}
```

Préconisation du bool en C++

Le Standard C++ a apporté un nouveau type, le `bool`. Ce type peut être associé à deux valeurs possibles, `true` (vrai) et `false` (faux). Nous pensons qu'il peut être utilisé systématiquement pour du nouveau code, car il améliorera la lisibilité. Dans ce morceau de code :

```
#include <iostream>

using namespace std;

int main()
{
  bool resultat = false;
  cout << "Resultat: " << resultat << endl;  // 0

  resultat = true;
  cout << "Resultat: " << resultat << endl;  // 1

  resultat = -2;                             // à éviter
  cout << "Resultat: " << resultat << endl;  // 1

  resultat = true;                           // 0
  resultat++;                                // Jamais !
  cout << "Resultat: " << resultat << endl;  // 1
}
```

nous remarquons que le type `bool` ne nécessite pas de fichier d'en-tête particulier. Il est donc intégré au compilateur, mais n'existe ni en C ni dans les compilateurs plus anciens. L'affectation de la variable avec la valeur de –2 entraîne une conversion automatique en 1 (`false`). L'opérateur `++` n'a aucun sens sur un booléen et pourrait disparaître des nouveaux compilateurs sans grande perte. L'opérateur `--` est d'ailleurs déjà refusé sur une variable de type `bool`.

Les boucles for, while et do

Les trois structures de contrôle `for`, `while` et `do` permettent de répéter des instructions. Bien que la dernière soit la moins utilisée, il est tout à fait possible d'utiliser n'importe laquelle des trois pour faire un travail équivalent. Cependant, il convient de choisir la forme qui fournira le code le plus simple et la présentation la plus adéquate, sans nécessairement penser à des considérations d'optimisation.

Avant de passer à notre exemple comparatif, regardons tout d'abord une forme tout à fait particulière de la boucle `for` éternelle :

```
for (;;) {
  ..... instructions
  if (...) {
    break;
  }
}
```

Cette forme est très souvent utilisée. C'est une boucle infinie ! Lorsque la condition `if ()` sera atteinte, correspondant à une certaine valeur, un événement extérieur ou encore à un moment déterminé, le `break` nous permettra de stopper le processus et de se retrouver à la sortie du bloc de l'instruction `for (;;)`.

Passons à présent à notre exemple comparatif des trois boucles au moyen de ce même exercice : imprimer les trois lettres `a`, `b` et `c` au moyen d'une boucle et d'une instruction séparée pour chaque lettre et répéter ceci avec les trois instructions `for()`, `while()` et `do`. Le résultat sera `abcabcabc`. Faire une variante du code dans la version Java qui va enchaîner les caractères avant de les imprimer.

En C++ :

```
// abc.cpp
#include <iostream>

using namespace std;

int main(int argc, char **argv) {
  for (int i = 0; i < 3; i++) {
    cout << (char)('a' + i);
  }

  int j = 3;
  while (j != 0) {
    cout << (char)('d' - j--);
  }

  char lettre = 'a';
  do {
    cout << lettre;
    lettre++;
  } while (lettre != 'd');

  cout << endl;
  return 0;
}
```

Il y a d'autres constructions possibles, mais la plus tordue reste celle du `while()` ! Lorsque nous prenons la première lettre, nous obtenons bien un `'a'` avec le `'d' - 3`. Comme le `--` est exécuté avant, à cause des règles de priorité des opérateurs, il est nécessaire de mettre un `'d'`. Nous pourrions très bien jouer avec un `'z'` et calculer les autres valeurs pour que cela fonctionne. Tout ceci pour affirmer que nous pouvons toujours écrire du code inutilement illisible (ce qui est le cas ici), mais correct !

Dans la version Java :

```
public class Abc {
 public static void main(String[] args) {
   String abc = "";

   for (int i = 0; i < 3; i++) {
     abc += (char)('a' + i);
   }

   int j = 3;
```

```
    while (j != 0) {
      abc += (char)('d' - j--);
    }

    char lettre = 'a';
    do {
      abc += lettre;
      lettre++;
    } while (lettre != 'd');

    System.out.println(abc);
  }
}
```

nous retrouvons la même structure et la même syntaxe. Cependant, nous y construisons un `String` qui n'est envoyé à la console qu'au dernier moment. En fait, le `String abc` pourrait être réutilisé à d'autres fins et, par exemple, être conservé en interne dans la classe.

Revenons à la construction particulière du `(char)`, qu'on retrouve dans les deux langages. La forme `(char)` est un transtypage.

```
abc += (char)('a' + i);
```

Sans le `(char)`, la boucle `for ()` nous donnerait ce résultat :

```
979899
```

c'est-à-dire les valeurs entières décimales de `'a'` (`97`), `'b'` (`98`) et `'c'` (`99`).

En exécutant `('a' + i)` nous obtenons la valeur décimale de la lettre dans la table ASCII. Nous comprenons enfin comment ce processus fonctionne, avec une astuce en Java, où le caractère ASCII est converti avec l'opérateur `+=` de la classe `String`. Lors de l'analyse des performances, plus loin dans cet ouvrage, nous montrerons que l'utilisation du `String` de cette manière est loin d'être efficace.

Pour résumer, la forme `for()` est vraiment la plus simple. Il faudrait toujours écrire du code en pensant qu'un autre programmeur pourrait modifier et corriger ce code. Il est évident que le `while()` peut très bien être accepté à condition d'inverser la logique. La forme `do {} while` est rarement utilisée. Le code suivant :

```
int k = 0;
while (k < 3) {
  cout << (char)('a' + k);
  k++;
}
```

est très clair, bien que la version suivante apparaisse plus souvent :

```
int k = 0;
while (k < 3) {
  cout << (char)('a' + k++);
}
```

Les boucles for en Java à partir du JDK 1.5

La boucle `for` en Java a été améliorée et étendue à partir du JDK 1.5. Nous y reviendrons à plusieurs occasions dans ce livre, mais voici un premier exemple avec les 7 chiffres d'un tirage de l'Euro Millions :

```
public class EuroMillions {
  public static void main(String[] args) {
    int[] tirage = {3, 7, 13, 23, 32, 1, 2 };

    for (int i = 0; i < tirage.length; i++) {
      System.out.print(tirage[i] + " ");
    }
    System.out.println();

    for (int numero:tirage) {
      System.out.print(numero + " ");
    }
    System.out.println();
  }
}
```

En exécutant le programme, nous recevrons deux fois le même tirage du vendredi de la semaine :

```
3 7 13 23 32 1 2
3 7 13 23 32 1 2
```

Cette nouvelle forme `for (int numero:tirage)` est nettement plus simple. C'est bien le caractère : et le caractère ;. Nous retirons du début à la fin chaque entier déposé dans `numero` depuis le tableau `tirage`. L'ancienne forme `tirage[i]` est aussi plus risquée si la valeur de `i` est négative ou plus grande que 7 ici.

Tester plusieurs conditions

Il est tout à fait possible de tester plusieurs conditions à l'intérieur de ces trois formes ou d'exécuter plusieurs instructions dans le `for()`. Il est aussi possible à l'intérieur du corps, comme pour l'exemple du `for (;;)`, de s'arrêter pour d'autres raisons. Si de tels cas peuvent arriver, il est essentiel d'y ajouter les commentaires appropriés ou d'affecter correctement les variables qui seront utilisées après la boucle.

Ceci ET cela

Prenons un exemple en C++ : considérons que l'école est obligatoire pour un enfant âgé d'au moins 6 ans, mais de moins de 17 ans. Cela peut se traduire par :

```
// ecole.cpp
#include <iostream>
```

```
using namespace std;

int main(int argc, char **argv) {
  int age = 0;
  bool balecole = false;

  cout << "Entre ton âge: ";
  cin >> age;

  if ((age >= 6) && (age < 17)) {
    cout << "Tu dois aller à l'école mon fils !" << endl;
    balecole = true;
  }
  else {
    cout << "Tu peux rester à la maison !" << endl;
  }

  return 0;
}
```

Cet exemple nous montre comment s'utilise l'opérateur de condition `&&` (`AND`, et), qui va nous permettre de tester deux conditions. L'opérateur `>=` (plus grand ou égal) va aussi inclure la valeur 6, alors que pour l'opérateur `<`, nous limiterons l'accès aux jeunes de moins de 17 ans. Le `balecole` (besoin d'aller à l'école) est ici pour expliquer comment ajouter une variable qui pourrait être réutilisée plus loin dans le code.

Le moment est venu de faire une remarque essentielle sur les deux opérateurs `&&` et `&`, qu'il ne faut pas prendre l'un pour l'autre. Si nous écrivons :

```
int valeur1 = 1;
int valeur2 = 2;
cout << "& : :" << (valeur1 & valeur2) << endl;   // 0
cout << "&&: :" << (valeur1 && valeur2) << endl;  // 1
```

nous obtiendrons des résultats différents. Ce problème n'existe pas en Java, qui travaille avec un booléen pour les opérateurs de condition. Comme nous l'avons déjà mentionné, il vaudrait mieux travailler avec des variables `bool`, qui sont apparues dans le Standard C++.

Optimisation dans le cas de conditions multiples

Il faut toujours penser à ce qui se passe réellement dans le code. Dans l'instruction :

```
if ((age >= 6) && (age < 17)) {
```

la condition (`age >= 6`) va être testée avant (`age < 17`). Si l'âge est inférieur à 6, la deuxième condition ne sera jamais exécutée, puisque la première n'est pas vraie. En revanche, si nous écrivons :

```
if ((age < 17) && (age >= 6)) {
```

ceci va nous paraître moins lisible, mais la première instruction (`age < 17`) va d'abord être exécutée. Bien que cette forme soit moins évidente, il faut considérer le cas où elle serait utilisée des milliers, voire plusieurs millions de fois ! Comme la plupart des individus sont âgés de plus de 16 ans, la deuxième partie (`age >= 6`) sera exécutée plus rarement. Donc cette dernière forme est plus avantageuse !

Ceci OU cela

Autre exemple : aujourd'hui, il y a aussi des invités de tout âge qui iront à l'école, car c'est la visite annuelle pour certains parents et amis. Ces invités seront donc acceptés à l'école indépendamment de leur âge. Voici à présent l'exercice en Java :

```
import java.util.*;
public class Ecole {
 public static void main(String[] args) {
   int age = 17;
   boolean bvisiteur = true;

   if ((bvisiteur) || ((age < 17) && (age >= 6))) {
     System.out.println("Tu vas à l'école");
   }
   else {
     System.out.println("Tu ne vas pas à l'école");
   }
 }
}
```

Afin de simplifier le code, nous avons laissé de côté l'entrée des données sur la console. Pour tester d'autres conditions, il faudra changer la valeur des variables directement dans le code. `bvisiteur` sera testé en premier. Si celui-ci est vrai, les conditions sur l'âge ne seront pas vérifiées. En revanche, dans le cas contraire, en raison de l'opérateur `||` (`OR`, ou), la deuxième partie :

```
((age < 17) && (age >= 6))
```

sera exécutée selon la combinaison déjà utilisée dans l'exercice C++. Les parenthèses nous aident à visionner correctement le code.

Éviter les tests de conditions multiples compliqués

Une instruction telle que :

```
if (!((!bvisiteur) && (age != 12))) {
```

devrait être évitée. Nous remarquons que nous vérifions si l'âge n'est pas de 12 ans (`!=`) : cette dernière construction est tout à fait raisonnable. Le `(!bvisiteur)` est également acceptable : la condition sera vraie si `bvisiteur` est faux. Cependant, nous inversons encore une fois le résultat, et ce code devient alors simplement illisible ! Nous laisserons au lecteur le soin d'imaginer une solution plus propre !

Plusieurs sélections avec switch

L'instruction `switch` est parfaite ! Elle combine efficacité et présentation. Elle n'a qu'un seul défaut : elle ne s'applique qu'aux types `char` et `int`. Le point de départ doit être donc une valeur ou un caractère. Au lieu d'utiliser une série d'`if`, `else`, `if`, il suffit d'ajouter un nouveau choix. Considérons la commande suivante :

```
cmd -l -g -s
```

Les différents paramètres, précédés du signe moins, convention très utilisée, sont des options de la commande `cmd`. Voici le code C++ :

```
// cmd.cpp
#include <iostream>

using namespace std;

int main(int argc, char **argv) {
  for (int i = 1; i < argc; i++) {
    if (argv[i][0] != '-') {
      cerr << "Caractère - du paramètre manquant" << endl;
      return -1;
    }
    else {
      if (argv[i][2] != 0) {
        cerr << "Paramètre invalide (trop long)" << endl;
        return -1;
      }

      switch (argv[i][1]) {
        case 'l':
          cout << "Paramètre l présent" << endl;
          break;
        case 'g':
          cout << "Paramètre g présent" << endl;
          break;
        case 's':
          cout << "Paramètre s présent" << endl;
          break;
        default:
          cerr << "Paramètre invalide" << endl;
          return -1;
      }
    }
  }

  return 0;
}
```

Le traitement des tableaux viendra au chapitre 5. Cependant, nous pensons que le lecteur ne devrait pas avoir trop de difficultés à comprendre ce code un peu plus élaboré. Considérons-le comme une introduction plus directe aux tableaux !

`argv` est un tableau de chaîne de caractères. `argv[1]` contient dans notre cas la chaîne `"-l"`. `argv[1][0]` contient le signe `"-"`, `argv[1][1]` la lettre `"l"` et `argv[1][2]` le chiffre `"0"`, qui indique que c'est la fin de la chaîne. C'est ensuite une question de structure pour contrôler les cas d'erreur, comme un signe `"-"` isolé sans lettre ou encore une lettre invalide.

Voici le code Java à présent :

```
import java.util.*;

public class Cmd {
 public static void main(String[] args) {
   for (int i = 0; i < args.length; i++) {
     if (args[i].charAt(0) != '-') {
       System.err.println("Caractère - du paramètre manquant");
       return;
     }
     else {
       if (args[i].length() != 2) {
         System.err.println("Paramètre invalide");
         return;
       }
       switch (args[i].charAt(1)) {
         case 'l':
           System.out.println("Paramètre l présent");
           break;
         case 'g':
           System.out.println("Paramètre g présent");
           break;
         case 's':
           System.out.println("Paramètre s présent");
           break;
         default:
           System.err.println("Paramètre invalide");
           return;
       }
     }
   }
 }
}
```

Nous avons déjà analysé les différences concernant les paramètres `argc`, `argv` et `args` au chapitre 1. Au sujet du `switch()`, le thème de cette partie, il n'y a pas de mystère : c'est tout à fait équivalent en C++ et Java.

La différence se situe au niveau des variables `argv` et `args`. En Java, il faut utiliser le `charAt()` pour obtenir un caractère dans une position déterminée, car nous travaillons avec un `String`. En C/C++, cela se fait plus simplement.

Nous notons aussi une différence de conception entre les deux versions. En C++, après avoir testé `argv[i][0]`, nous passons directement à `argv[i][2]`, qui devrait être 0. Cependant, nous n'avons pas encore testé `argv[i][1]`, qui pourrait être 0 si nous n'entrons que "-" sans lettre. En revanche, le langage C++ accepte tout de même de lire sans problème en dehors des limites. C'est là une des grandes faiblesses de C++. Le cas particulier d'un "-" unique sortira en fait avec le choix `default`, ce qui signifie : tous les autres cas.

Si nous avions utilisé la même logique en Java avec une instruction équivalente à :

```
if (args[i].charAt(2) != 0) {
```

nous aurions obtenu une erreur pendant l'exécution :

```
java Cmd -o
Exception in thread "main" java.lang.StringIndexOutOfBoundsException: String index
➥out of range: 2
        at java.lang.String.charAt(Compiled Code)
        at Cmd.main(Compiled Code)
```

L'index de `charAt()` doit être plus petit que `args[i].length()` !

En Java, il faut distinguer ces deux formes :

```
args.length
args[i].length()
```

La première donne la grandeur du tableau, alors que la seconde indique la dimension de la chaîne pour un `String` dans le tableau. Nous verrons plus loin que `length` est intégré dans le compilateur et que `length()` est une méthode de classe.

Une dernière remarque : nous ne contrôlons pas le nombre d'arguments ou la répétition du même paramètre. Si aucun paramètre n'est spécifié, la commande est acceptée. En d'autres termes, cela signifie que tous les paramètres sont optionnels.

L'infâme goto

Il est vraiment infâme et devrait être interdit ! Un programmeur de culture Basic ou assembleur devra apprendre à s'en passer. Il est accepté en C++ et rejeté en Java. Nous n'en donnerons pas la syntaxe, car cela pourrait donner de mauvaises idées au programmeur.

L'instruction `goto` autorise un transfert direct à un endroit spécifique du code. Ce qui est plus grave encore est la prolifération possible de multiples points de transfert, dont nous n'arriverions plus à identifier l'origine et la raison. Un programmeur Basic devrait s'amuser à programmer sans instructions `goto` ! Il verrait alors apparaître un code structuré qu'il serait enfin possible de documenter d'une manière simplifiée et de modifier sans grandes difficultés.

Résumé

Nous avons appris ici les structures et les instructions fondamentales que nous pouvons retrouver dans la plupart des langages de programmation. Nous sommes à présent préparés pour passer à des concepts plus spécifiques et plus sérieux que sont les classes et les objets.

Exercices

1. Modifier le programme qui teste les paramètres de la commande `cmd -l -g -s` afin qu'il accepte un format plus étendu du style `cmd -lgs ou cmd -l -gs`. Les instructions `if`, `for` et `switch` devraient apparaître.
2. Tester si un entier est pair avec l'opérateur `&`. Si l'entier est pair, indiquer s'il est aussi divisible par quatre.

4

On fait ses classes

Certains diront que c'est un peu tôt pour présenter les classes. Mais le problème se pose ici différemment puisqu'il s'agit d'une comparaison directe entre le C++ et Java. Dès le premier exemple Java présenté dans cet ouvrage, nous avions déjà rencontré un objet, notre fameux `String[] args`. En C++, nous avons le `char *` pour les chaînes de caractères et les tableaux multidimensionnels. Nous pouvons même utiliser la structure `struct`, qui est héritée du langage C, et écrire des programmes complets avec un compilateur C++, sans jamais instancier un objet d'une classe (voir définitions de ces termes page suivante). Nous reviendrons sur ces aspects dans les chapitres suivants.

Notre première classe en C++

Nous allons écrire une classe `Personne`, à partir de laquelle nous pourrons créer des objets C++ qui nous permettront de conserver et d'obtenir les données suivantes :

- le nom de famille d'une personne ;
- son prénom ;
- son année de naissance sous la forme d'un nombre entier.

Définition de la classe Personne

La première opération consiste à définir la classe `Personne`. Comme elle sera réutilisée plus tard, nous allons la définir dans un module à part, avec un fichier d'en-tête séparé. Ce fichier d'en-tête nous permettra d'utiliser cette classe dans d'autres parties de code et d'applications. Nous verrons, plus loin dans ce chapitre, comment compiler les différents

modules, lorsque nous aborderons le `Makefile` pour construire une application utilisant notre classe `Personne`. Voici donc notre première définition de classe en C++ :

```
// Personne.h : définition de la classe Personne
#include <string>

class Personne {
  private:
    std::string nom;
    std::string prenom;
    int         annee;

  public:
    Personne(std::string leNom, std::string lePrenom, std::string lAnnee);
    void un_test();
};
```

Nous n'allons pas trop philosopher sur l'utilisation prématurée de l'objet `string`, qui fait partie de la bibliothèque Standard C++ et qui est défini par la directive :

```
#include <string>
```

Avec cette directive, nous allons accéder et inclure le fichier `string` dont le chemin d'accès est retrouvé par le compilateur (voir la section « Directive include » plus loin dans ce chapitre).

L'utilisation de chaînes de caractères C (`char *`) aurait été également possible, mais aurait rendu notre présentation encore plus lourde.

Mais revenons à la définition de notre classe qui a la forme :

```
class Nom_de_la_classe { … };
```

À l'intérieur des accolades, nous trouvons la définition des objets et des méthodes de la classe. C'est sans doute le bon moment de donner quelques définitions :

- **Classe** – Entité générique qui permet d'engendrer des objets analogues.
- **Objet** – Instance d'une classe. Un objet sera composé d'attributs et de méthodes. Les attributs des objets peuvent être eux-mêmes des objets (`nom` et `prenom`) ou des attributs de type primitif (`annee` est un entier). `nom` et `prenom` sont des instances de la classe `string` de la bibliothèque du Standard C++. La forme `std::string`, dans le fichier de définition `Personne.h`, est en relation avec l'utilisation des espaces de noms pour le Standard C++ (`std`). Nous verrons au chapitre 6 la raison de l'utilisation de cette forme. Nous pouvons considérer pour l'instant que `std::string` est identique à `string`.
- **Instance** – Terme indiquant qu'un objet est créé à partir d'une classe. `nom` est une instance de la classe `string`. Objets et instances sont souvent utilisés pour exprimer la même chose.
- **Méthode** – Fonction ou procédure unique attribuée à cette classe permettant d'accéder aux données de l'objet ou de modifier son comportement en changeant ses attributs.

- **Constructeur** – Méthode qui permet d'initialiser l'objet. Elle est appelée à la création de l'instance.
- **Destructeur** – Méthode appelée à la destruction de l'objet afin d'effacer ou de libérer certaines ressources encore actives qui ont été utilisées pendant une partie ou toute la durée de vie de l'objet.

Présentons ensuite les deux mots-clés, `private` et `public`. `public` nous indique que les objets ou les méthodes de la classe sont directement accessibles. Nous verrons des exemples plus loin. `private` est utilisé pour cacher des objets à l'intérieur de la classe et ne les rendre accessibles qu'au moyen de méthodes de classe. Les trois attributs de la classe `Personne` (`nom`, `prenom` et `annee`) sont privés, ce qui constitue un bon départ en programmation orientée objet. Le caractère : de `private` nous indique le début de la partie privée de la classe. Ce bloc se termine ici au `public:` suivant. Il est tout à fait possible de mélanger les parties publiques et privées et d'en définir plusieurs. C'est cependant une bonne habitude de les regrouper comme c'est le cas dans notre définition de classe. C'est propre et clair.

Définition des objets d'une classe

Tous les attributs devraient être privés (`private`) ; c'est l'un des fondements de la programmation orientée objet.

Un attribut déclaré `public` sera accessible et modifiable par n'importe qui, ce qui signifie qu'il n'y aura aucun contrôle. En revanche, s'il est privé, seules les méthodes de la classe auront un droit d'accès. Celui-ci pourra donc se faire au travers de méthodes définies une fois pour toute (API), et le programmeur aura ainsi un contrôle total du code de sa classe. Il pourra même changer cette dernière, la définition de ces attributs, et ceci avec la seule contrainte extérieure de garder inchangée la définition des méthodes publiques de sa classe.

Un seul constructeur

Pour ce premier exercice, nous ne définirons qu'un seul constructeur et qu'une seule méthode. Le constructeur :

```
Personne(std::string leNom, std::string lePrenom, std::string lAnnee);
```

permet au programmeur de créer un objet de cette classe. Nous le verrons dans l'exemple ci-dessous. Un constructeur ne peut retourner de valeur, ce qui signifie que nous ne pouvons jamais vérifier le résultat de la création d'objet directement. Nous traiterons les détails plus loin dans cet ouvrage. Dans notre exemple de constructeur, nous avons choisi trois paramètres. C'est un choix que nous avons fait. Si dans la conception du programme nous avions déterminé que l'année de naissance n'était pas primordiale, nous l'aurions laissée de côté.

Une seule méthode

La méthode que nous avons choisie ne peut être plus simple : elle n'a ni paramètre ni valeur de retour (indiquée par `void`).

```
void un_test();
```

Nous en comprendrons la raison très rapidement.

Nom et définition des classes

Il est recommandé de définir un nom de classe commençant par une majuscule.

En C++, au contraire de Java, il n'est pas nécessaire d'écrire systématiquement des classes. Il est tout à fait possible d'élaborer un programme complet en C++ sans construire de classes, mais en utilisant simplement les outils des bibliothèques C ou C++ du langage. Cependant, dès que nous pensons réutiliser ce code à d'autres fins, il est judicieux de penser à élaborer de nouvelles classes, plus ou moins globales et génériques, et de les collectionner dans une bibliothèque, comme nous le verrons au chapitre 7.

Code de la classe Personne

Dès que la classe est définie, il s'agit d'écrire le code nécessaire qui nous permettra de créer des instances de cette classe et d'utiliser ses méthodes. La forme `"Personnes.h"` de la directive `#include` ci-dessous nous indique que le fichier `Personnes.h`, qui sera inclus lors de la compilation, se trouve dans le répertoire courant, c'est-à-dire celui du fichier `Personne.cpp` (voir la section « Directive include » plus loin dans ce chapitre).

```
// Personne.cpp
#include "Personne.h"

#include <iostream>
#include <sstream>

using namespace std;

Personne::Personne(string leNom, string lePrenom, string lAnnee) {
   nom    = leNom;
   prenom = lePrenom;

   //flux pour la conversion d'un string
   istringstream iss(lAnnee);
   iss >> annee;
}

void Personne::unTest() {
   cout << "Nom et prénom: " << nom << " " << prenom << endl;
   cout << "Année de naissance: " << annee << endl;
}
```

La première remarque concerne l'opérateur `::`, qu'on nomme opérateur de portée. Le nom précédant `::` nous indique le nom de la classe. `Personne::Personne(...)` représente le constructeur et `Personne::un_test()` une méthode de la classe `Personne`.

Nous découvrons ici le code nécessaire pour le constructeur de `Personne` et de la méthode `un_test()`, qui nous permettra de vérifier notre première classe. Les deux directives `include` sont nécessaires pour obtenir les définitions de l'objet `cout`, que nous avons vu dans le premier chapitre, et de la classe `istringstream`. Nous reviendrons plus loin sur cette dernière, qui fait partie de ces classes essentielles du Standard C++ qu'il s'agit de maîtriser. Ce qu'il faut retenir ici est que les instructions :

```
istringstream iss(lAnnee);
iss >> annee;
```

nous permettent d'extraire du `string lAnnee` sa valeur entière, car `annee` est un entier (`int`). Nous déposons notre `string` dans l'objet `iss` qui possède un opérateur d'extraction `>>` dans un entier, afin de nous retourner sa valeur (voir chapitre 9 pour plus de détails).

Nous aurions pu choisir de définir notre constructeur de cette manière :

```
Personne(string leNom, string lePrenom, int lAnnee);
```

et la conversion n'aurait pas été nécessaire. Cependant, nous pouvons considérer que les trois variables (nom, prénom et année de naissance) seront définies par une interface utilisateur qui passera les paramètres en tant que `string`. Ceci est plus une question de choix au cours de la conception, mais rien ne nous empêcherait de définir un deuxième constructeur. Le langage C++, comme Java d'ailleurs, nous le permet. Il faudrait alors ajouter la ligne ci-dessus dans `Personne.h`, ainsi que le code suivant dans le fichier `Personne.cpp` :

```
Personne::Personne(string leNom, string lePrenom, int lAnnee) {
   nom    = leNom;
   prenom = lePrenom;
   lannee = lAnnee;
}
```

Une autre forme plus concise est aussi disponible pour l'initialisation des objets (attributs) de la classe :

```
Personne::Personne(string leNom, string lePrenom, int lAnnee)
         :nom(leNom), prenom(lePrenom), annee(lAnnee) {
}
```

Ici, les variables privées qui suivent directement le caractère : seront affectées avec les valeurs définies entre parenthèses. Ces dernières doivent faire partie des paramètres du constructeur.

La méthode `un_test()` n'a besoin ni de recevoir de paramètre ni de retourner un quelconque résultat. Elle est juste là pour sortir sur la console le contenu des attributs de cette classe.

Enfin, nous passons au programme de test de notre classe :

```
// fichier: TestPersonne.cpp
// test de la classe Personne
#include "Personne.h"
```

```
int main()
{
   Personne haddock("Haddock", "Capitaine", "1907");
   haddock.un_test();
   return 0;
}
```

Ce code viendra dans un fichier séparé, `TestPersonne.cpp`.

Le `#include` est essentiel, car il nous permettra d'accéder à la définition de la classe `Personne` et de sa méthode `un_test()`. Au premier abord, l'instruction qui correspond à créer une instance de la classe `Personne` :

```
Personne haddock("Haddock", "Capitaine", "1907");
```

peut nous paraître étrange. `Personne` est le nom de la classe, `haddock` est la variable, suivie des trois paramètres exigés par le constructeur. Une instruction telle que :

```
int valeur(10);
```

sur une variable de type primitif est aussi possible en C++. Nous allons d'ailleurs très rapidement nous familiariser avec cette forme, qui n'est pas permise en Java.

À l'exécution du programme, nous obtiendrons finalement :

```
Nom et prénom: Haddock Capitaine
Année de naissance: 1907
```

Ceci correspond au résultat attendu de la méthode `test()` appliqué à l'instance `haddock` de la classe `Personne`. Enfin, et par analogie à la forme de l'instanciation en Java, il nous faut présenter ici une autre manière essentielle de créer une instance en C++ :

```
Personne *ppersonne;
ppersonne = new Personne("Haddock", "Capitaine", "1907");
ppersonne->un_test();
delete ppersonne;
```

Ceci se fait de la même manière qu'une variable de type pointeur, que nous avons découvert au chapitre 2.

Le caractère `*` indique que nous avons affaire à une variable du type pointeur, car l'opérateur `new` nous retournera un pointeur en mémoire où se trouve effectivement l'objet. En C++, il y a donc deux formes pour la création d'objets, ce qui n'est pas le cas en Java. Le `->` est nécessaire pour indiquer au compilateur que `ppersonne` est un pointeur. Si le `delete` n'était pas présent, l'objet resterait en mémoire et ne serait jamais libéré. Nous reviendrons sur les détails en fin de chapitre, après la partie consacrée à Java.

Directive include

Jusqu'à présent, nous avons rencontré trois formes de directive `include`, dont la dernière au chapitre 1 :

```
#include "Personne.h"
```

```
#include <iostream>
#include <time.h>
```

La première, avec les `""`, va inclure le fichier `Personne.h` lors de la compilation g++ de `Personne.cpp` en recherchant le fichier dans le répertoire courant. Si nous avions écrit :

```
#include "perso/Personne.h"
```

il aurait fallu créer un sous-répertoire `perso` dans le répertoire de travail (le caractère `\` convient aussi, mais le caractère `/` est préférable pour la compatibilité avec Linux).

Les `iostream` et `time.h` indiquent des références à des fichiers d'en-tête du C++ et du C respectivement. C'est le compilateur qui retrouvera ces références. Pour notre installation, ils se trouvent dans les répertoires suivants :

- `C:\MinGW\include\c++\3.4.5` ;
- `C:\MinGW\include`.

Nous pouvons évidemment charger ces fichiers dans Crimson et comprendre comment ils sont construits. Le lecteur y découvrira sans doute les directives `#define` et `#ifndef` qui permettent d'introduire un mécanisme pour n'inclure qu'une seule fois d'autres fichiers d'en-tête, déjà inclus lors d'`includes` multiples. Nous y reviendrons à la fin du chapitre 6, section « Fichiers d'en-tête multiples en C++ ».

Ceci concerne évidemment la compilation des objets `.o`. Cependant, lors de la génération de l'exécutable, nous aurons un même mécanisme, transparent pour le programmeur, qui va inclure cette fois-ci du code exécutable. Le `cout` ci-dessus, dans le code `unTest()` du fichier `Personne.cpp`, aura besoin du fichier `C:\MinGW\include\c++\3.4.5\string` pour la génération de `Personne.o`. Le fichier final exécutable `TestPersonne.exe` aura besoin de plusieurs morceaux de code binaire disponibles dans une ou plusieurs bibliothèques `.a` que nous retrouvons dans le répertoire du compilateur g++ : `C:\MinGW\lib`.

Au chapitre 7, section « Utilisation de notre bibliothèque C++ », nous décrirons le paramètre de compilation `-I`. Celui-ci nous permettra de compiler nos codes source en recherchant nos fichiers d'en-tête `.h` dans les répertoires spécifiés par ce `-I`. Nous avons un même mécanisme pour des bibliothèques privées avec le `-L`. C'est aussi dans ce même chapitre que nous verrons comment créer nos propres bibliothèques `.a`.

Commentaires et documentation des classes

Il n'est pas nécessaire d'ajouter trop de commentaires dans les fichiers `.cpp` correspondant à l'implémentation des classes. Cependant, si, par exemple pour des raisons d'optimisation, le code devient plus complexe, il est conseillé de décrire plus en détail ce qui pourrait sembler difficile à déchiffrer pour un programmeur qui devrait reprendre ce code.

Cependant, l'endroit correct et idéal pour la documentation reste le fichier d'en-tête. Celui-ci peut aussi être livré séparément avec une bibliothèque précompilée. Nous allons donner ici un exemple de documentation possible pour notre classe `Personne`. Le fichier `Personne.h` pourrait se présenter comme ceci :

```
#include <string>

class Personne {
  private:
    std::string nom;     // nom de la personne
    std::string prenom;  // prénom de la personne
    int         annee;   // année de naissance de la personne

  public:
    Personne(std::string leNom, std::string lePrenom, std::string lAnnee);
    // Description :
    //    Constructeur
    //    Aucun contrôle sur le contenu des variables n'est
    //    exécuté.
    //
    // Paramètres :
    //    leNom    - le nom de la personne
    //    lePrenom - le prénom de la personne
    //    lAnnee   - l'année de naissance de la personne
    //

    Personne(char *unNom, char *unPrenom, char *uneAnnee);
    // Description :
    //    Constructeur
    //    Avec des char* classiques
    //
    // Paramètres :
    //    unNom    - le nom de la personne
    //    unPrenom - le prénom de la personne
    //    uneAnnee - l'année de naissance de la personne
    //

    Personne(std::string leNom, std::string lePrenom, int lAnnee);
    // Description :
    //    Constructeur
    //    Aucun contrôle sur le contenu des variables n'est
    //    exécuté.
    //
    // Paramètres :
    //    leNom - le nom de la personne
    //    lePrenom - le prénom de la personne
    //    lAnnee - l'année de naissance de la personne
    //

    void unTest();
    // Description :
    //    Affiche sur la console tous les attributs de la classe.
    //    Uniquement utilisé à des fins de test.
    //
    // Paramètres :
    //    Aucun
```

```
    //
    // Valeur de retour :
    //    Aucune
    //
};
```

Notons déjà quelques points de détail importants :

- Documenter les variables privées n'est en fait utile que pour les développeurs de la classe, car les utilisateurs n'ont pas directement accès à ces attributs.
- Comme le constructeur ne peut retourner de valeur, c'est-à-dire de résultat, une description de la valeur de retour n'est pas nécessaire. Nous touchons déjà à un sujet délicat, sur lequel nous reviendrons, qui est de savoir comment traiter les cas d'erreur dans un constructeur. Comme règle de base, nous dirons simplement que le constructeur ne devrait pas allouer de ressources pouvant entraîner des difficultés.

La première remarque qui nous saute aux yeux est le contenu de ce code, qui ne contient en fait que des définitions, et ceci à des fins d'analyse et de conception. Nous pouvons très bien distribuer cette définition sans coder l'implémentation, simplement pour vérification (*design review*). Cette définition de classe est-elle suffisante à nos besoins ? Certainement pas ! Il faudrait y ajouter un certain nombre de méthodes plus évoluées et se poser toute une série de questions comme celle de savoir ce que voudrait bien dire un âge négatif ou un nom et un prénom vide.

Ce qu'il faut retenir avant tout ici, c'est de savoir si les commentaires sont suffisants pour un programmeur qui va utiliser cette classe. Si ce but est atteint, c'est très bien, tout simplement !

Un Makefile évolué

L'exemple ci-dessus de la classe `Personne` et de son programme de test est plus complexe que celui du premier chapitre, dans lequel le `make` de GNU a été présenté. Nous pouvons maintenant passer à un exemple nous montrant d'autres aspects d'un `Makefile`.

```
all: testp.exe

testp.exe:          Personne.o TestPersonne.o
                    g++ -o testp.exe Personne.o TestPersonne.o

Personne.o:         Personne.cpp Personne.h
                    g++ -c Personne.cpp

TestPersonne.o:     TestPersonne.cpp Personne.h Personne.h
                    g++ -c TestPersonne.cpp
```

Attention tout d'abord aux tabulateurs et aux espaces : nous rappellerons, par exemple, qu'entre `testp.exe` et `Personne.o`, il n'y a pas d'espace, seulement un ou plusieurs tabulateurs.

En revanche, entre `Personne.o` et `TestPersonne.o`, il y a un espace et non un tabulateur. En cas de doute, consultons un `Makefile` sur le CD-Rom fourni avec cet ouvrage.

Le fichier binaire et exécutable `testp.exe` est formé de deux composants, `Personne.o` et `TestPersonne.o`, qui sont compilés séparément puis liés (option `–o`). Lorsque `Test Personne.o`, qui contient le point d'entrée `main()`, est lié avec `Personne.o`, il est possible que le compilateur découvre une ressource manquante. Celle-ci provient en général de la bibliothèque standard du compilateur et non pas de la classe `Personne`, car le fichier d'entête `Personne.h` est inclus dans le fichier `TestPersonne.cpp`, et les ressources de cette classe ont déjà été vérifiées par la première phase de compilation.

Ce qui est important de noter ici, c'est la dépendance supplémentaire sur `Personne.h` que nous avons rajoutée. Si ce dernier fichier est modifié, la dépendance se répercutera sur les trois parties, et tous les composants seront à nouveau compilés. Il est donc essentiel de vérifier soigneusement son `Makefile` ainsi que la présence correcte de tabulateurs, comme nous l'avons vu au chapitre 1.

`Personne.o` est donc la première partie exécutée par le Makefile. Notons que l'option `-c` demande au compilateur de produire un fichier binaire `.o` à partir du code source (par exemple, `Personne.cpp` deviendra `Personne.o`). `Personne.o` représente du code binaire, mais il n'est pas exécutable. Ce n'est que le résultat de la première phase de la compilation, et il lui manque par exemple toutes les fonctions de la bibliothèque, ainsi que le code nécessaire au démarrage du programme.

L'option `-o` est suivie du nom que nous voulons donner au programme exécutable. Le compilateur devra lier ses différents composants et lui ajouter les références externes comme les fonctions de la bibliothèque des **iostreams**. Le compilateur saura en fait lui-même trouver ces bibliothèques avec le chemin d'accès `..\lib\` à partir de son répertoire d'installation (`bin`) sur le disque.

Si nous écrivons :

```
g++ -o testp.exe Personne.o testPersonne.cpp
```

le compilateur commencera par compiler `testPersonne.cpp`, mais ne produira pas de `test Personne.o` car il sera directement intégré dans `testp.exe`. Cependant, il est préférable de passer par différentes phases, qui peuvent être spécifiées par des dépendances dans un `Makefile`.

Notre première classe en Java

Nous allons à présent faire le même exercice en Java avec la classe `Personne` que nous venons de créer en C++. Nous constaterons qu'il existe beaucoup de similitudes avec le C++. Cependant, en Java, il n'y a pas de fichiers séparés pour la définition et le code de la classe, car ces deux composants ne peuvent être dissociés et doivent se trouver dans le même fichier `.java`.

Comme en C++, il est recommandé de définir un nom de classe commençant par une majuscule. Le nom des fichiers `.h` et `.cpp` suivra cette règle. Dans la mesure du possible, nous ne définirons qu'une seule classe par fichier d'en-tête `.h`. Nous avions au chapitre 1 deux fichiers source : `hello.cpp` et `Hello.java`. Le `h` de `hello.cpp` était en minuscule, mais ce n'était pas le cas de `Hello.java`. On se rappellera de plus que le nom de la classe et le nom du fichier doivent être identiques, incluant les majuscules et minuscules correctes.

Voici donc notre code Java correspondant à notre classe `Personne` précédemment définie :

```
public class Personne {
 private String nom;
 private String prenom;
 private int annee;

 Personne(String lenom, String leprenom, String lannee) {
   nom = lenom;
   prenom = leprenom;
   annee = Integer.parseInt(lannee);
 }

 Personne(String lenom, String leprenom, int lannee) {
   nom = lenom;
   prenom = leprenom;
   annee = lannee;
 }

 void un_test() {
   System.out.println("Nom et prénom: " + nom + " " + prenom);
   System.out.println("Année de naissance: " + annee);
 }

 public static void main(String[] args) {
   Personne nom1 = new Personne("Haddock", "Capitaine", "1907");
   Personne nom2 = new Personne("Kaddock", "Kaptain", 1897);
   nom1.un_test();
   nom2.un_test();
 }
}
```

Nous retrouvons les mêmes mots-clés qu'en C++ pour les permissions d'accès : `private` et `public`. Nous avons aussi laissé les deux constructeurs. Tout d'abord, il nous faut revenir sur une construction très particulière :

```
annee = Integer.parseInt(lannee);
```

qui peut sembler complexe. `lannee` est un `String`, et il s'agit de le convertir en `int`. `parseInt` est une méthode statique de la classe `Integer` qui fera la conversion.

Dans la partie `main()`, nous constatons que les deux directives `new` sont nécessaires en Java. Il n'y a pas d'autre alternative comme en C++. Nous analyserons les détails un peu plus loin dans ce chapitre.

À l'exécution du programme, nous obtiendrons :

```
java Personne
Nom et prénom: Haddock Capitaine
Année de naissance: 1907
Nom et prénom: Kaddock Kaptain
Année de naissance: 1897
```

Apparaît ici une différence essentielle avec la version C++. En effet, le :

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

nous permet de tester directement notre classe.

Si nous voulions tester la classe séparément, comme nous l'avons fait avec `TestPersonne` en C++, il nous faudrait écrire une classe séparée, `TestPersonne`, qui serait programmée de cette manière :

```
public class TestPersonne {
 public static void main(String[] args) {
   Personne nom1 = new Personne("Paddock", "Capitaine", "1907");
   Personne nom2 = new Personne("Maddock", "Kaptain", 1897);
   nom1.un_test();
   nom2.un_test();
 }
}
```

Pour pouvoir exécuter `java TestPersonne`, il est nécessaire, sans autres complications pour l'instant, d'avoir la classe `Personne.class` définie dans le même répertoire !

Tester les classes

Il est tout à fait possible de laisser un :

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

dans toutes les classes afin de pouvoir les tester séparément. Une application, du style de `TestPersonne`, mais employant plusieurs classes, ne va pas contrôler ni utiliser toutes les méthodes et cas d'erreurs implémentés dans la classe. Alors, pourquoi ne pas garder un `main()` dans la classe, avec une bonne logique de test, et éventuellement l'effacer lors de la préparation finale ou de la livraison du produit ?

Commentaires et documentation des classes

Lorsque nous parlons de documentation en Java, nous pensons immédiatement à `javadoc`, un produit de Sun Microsystems. La première fois qu'un programmeur exécute `javadoc`, il est simplement émerveillé ! Essayons donc avec notre première classe `Personne.java`. Le premier conseil à donner est de copier notre fichier `Personne.java` dans un nouveau répertoire vide et d'exécuter :

```
javadoc Personne.java
```

La première surprise vient du nombre de fichiers générés, à savoir une feuille de style `.css` et de nombreux documents `.html` ! Pour constater le résultat, il suffit de charger le document `index.html` avec Internet Explorer ou Netscape. Une merveille, cette petite classe `Personne` !

Par analogie avec notre documentation de classe en C++, il est aussi possible d'utiliser cette documentation afin de vérifier la conception de notre classe `Personne.java`. Dans ce cas, il est souhaitable de générer aussi la partie privée de la classe. Cela se fait avec l'option `private` :

```
javadoc -doc doc -private Personne.java
```

L'option `-doc` permet de spécifier le répertoire dans lequel seront générés les fichiers `.html`. Pour simplifier ou régénérer rapidement la documentation, nous avons créé un fichier `genJavadoc.bat`.

`javadoc` est un outil puissant qu'il est possible de configurer à nos besoins, afin de décrire notre classe tout aussi bien que nous l'avons fait pour notre classe C++. Au premier abord, cette nouvelle version peut nous paraître étrange, mais après l'avoir « javadocée », il en sera tout autrement dès que nous l'aurons visionnée avec notre navigateur Web favori.

```
/**
 * La classe <code>Personne</code> est notre première classe en Java.
 * D'une simplicité extrême, elle est juste là comme introduction.
 * Il est possible de l'utiliser de cette manière :
 * <p><blockquote><pre>
 *   Personne nom1 = new Personne("Paddock", "Capitaine", "1907");
 *   nom1.un_test();
 *
 * </pre></blockquote>
 *
 * @author  J-B.Boichat
 * @version 1.0, 18/07/08
 */
public class Personne {
 private String nom;
 private String prenom;
 private int annee;

 /**
  * Alloue une nouvelle <code>Personne</code>.
  * L'année est un String.
  *
  * @param  leNom     Le nom de la personne.
  * @param  lePrenom  Le prénom de la personne.
  * @param  lAnnee    L'année de naissance de la personne.
  */

 public Personne(String leNom, String lePrenom, String lAnnee) {
   nom    = leNom;
```

```
    prenom = lePrenom;
    annee  = Integer.parseInt(lAnnee);
  }

  /**
   * Alloue une nouvelle <code>Personne</code>.
   * L'année est un entier.
   *
   * @param  leNom     Le nom de la personne.
   * @param  lePrenom  Le prénom de la personne.
   * @param  lAnnee    L'année de naissance de la personne.
   */

  public Personne(String leNom, String lePrenom, int lAnnee) {
    nom    = leNom;
    prenom = lePrenom;
    annee  = lAnnee;
  }

  /**
   * Affiche sur la console tous les attributs de la classe.
   * Uniquement utilisé à des fins de test.
   *
   * @return     Aucun
   */

  public void un_test() {
    System.out.println("Nom et prenom: " + nom + " " + prenom);
    System.out.println("Année de naissance: " + annee);
  }

  public static void main(String[] args) {
    Personne nom1 = new Personne("Haddock", "Capitaine", "1907");
    nom1.un_test();
  }
}
```

Notons tout d'abord la séquence des `/** .*/`. `javadoc` est capable d'interpréter des commandes à l'intérieur d'une séquence commençant par `/**` et qui se termine par `*/`. Les `*` sur les lignes intérieures sont ignorés par `javadoc`.

Nous découvrons de plus toute une série de mots-clés (`<code>`, `</code>`, `<p>`, `<blockquote>`, `<pre>`, `</pre>` ou encore `</blockquote>`) qui permettent d'encastrer du code HTML. Ceci permet de rendre la documentation plus lisible, car ils apparaîtront de toute manière en HTML. Il y a aussi d'autres codes comme `@author`, `@version`, `@param` ou `@return` qui sont interprétés par `javadoc`. `@param` est utilisé pour définir les paramètres d'entrée des méthodes et du constructeur et `@return` pour la valeur de retour.

Nous montrons ici une partie du code généré en HTML pour notre classe :

Classe Personne

```
java.lang.Object
  |
  +--Personne
public class Personne
extends java.lang.Object
```

La classe `Personne` est notre première classe en Java. D'une simplicité extrême, elle est juste là comme introduction !

Il est possible de l'utiliser de cette manière :

```
Personne nom1 = new Personne("Paddock", "Capitaine", "1907");
nom1.un_test();
```

La syntaxe communément utilisée pour Firefox ou Internet Explorer est la suivante (nous pouvons aussi double-cliquer sur le document `index.html` dans l'explorateur de Windows) :

```
file:///C:/JavaCpp/EXEMPLES/Chap04/doc/index.html
```

`file:///` au lieu de `http://` indique que le document est accessible localement sur notre machine. Nous verrons qu'il nous est possible de naviguer entre les attributs ou méthodes. Tout en haut de ce document `index.html`, nous aurons des liens renvoyant à d'autres documents générés par `javadoc` : le lien `Index` est particulièrement intéressant.

La documentation de `javadoc` avec ses nombreuses options se trouve sur le CD-Rom d'accompagnement ; elle est installée avec les outils du JDK de Sun Microsystems, comme décrit dans l'annexe B. L'outil de développement NetBeans (voir annexe E) possède des instruments puissants pour préparer l'édition de la documentation (génération automatique de modèle incluant les paramètres des méthodes de classe).

Création des objets

Nous allons montrer à présent les différences importantes entre Java et C++ lors de la création des objets. Le langage Java a simplifié au maximum le processus, alors que le C++ a plusieurs façons de procéder et requiert beaucoup plus de précautions.

Lorsque nous écrivons en Java :

```
String nom = new String("mon_nom_de_famille");
System.out.println(nom);
```

la variable `nom` après l'exécution de la première instruction contient un objet instancié de la classe `String` qui renferme notre texte `"mon_nom_de_famille"`. Le `println` de l'instruction suivante est légitime car la classe `String` possède une méthode `toString()` qui va rendre l'opération possible au travers de la méthode `println()`. Cependant, si nous écrivions :

```
String nom = "mon_nom_de_famille";
```

à la manière C/C++, cela fonctionnerait aussi, car le compilateur accepte cette forme particulière, mais uniquement pour cette classe.

Nous allons examiner les équivalences possibles en C++ :

```
string nom  = "mon_nom_de_famille";
cout "nom" endl;
```

ou bien

```
string *pnom;
pnom = new string("mon_nom_de_famille");
cout "*pnom" endl;
delete nom;
```

En Java, tous les objets sont construits avec le `new` : ce n'est pas le cas en C++. Un objet C++ peut être créé sur la pile (*stack*) et disparaîtra lorsque nous sortirons du corps de la méthode (ici l'accolade fermante `}`).

En Java et en C++, l'utilisation de `new` fonctionne de la même manière, et les objets seront créés dans le tas (*heap*). Cependant, dans tous les cas, même sans opérateur `new` en C++, le constructeur sera appelé avec les paramètres nécessaires.

En Java, il n'y a pas de destructeur, donc pas de `delete`. Le `delete` en C++ s'applique sur une adresse d'objet. Le `delete` sera responsable de l'effacement des ressources au moyen du destructeur, qui sera aussi appelé si l'objet est dans la pile. Les objets Java sont automatiquement effacés par le récupérateur de mémoire (l'effaceur ou ramasse-miettes ou *garbage collector*) lorsqu'ils ont perdu toutes références à des variables utilisées.

En C++, si on oublie un `delete`, c'est dramatique. Si nous écrivons en C++ :

```
string *pnom;
pnom = new string("mon_nom_de_famille");
pnom = new string("mon_nouveau_nom_de_famille");
```

un objet est à nouveau créé avec la dernière instruction, et le pointeur de l'ancienne mémoire est perdu. Cette zone mémoire restera allouée jusqu'à l'achèvement du programme. Comme celui-ci peut très bien être un processus permanent, il faut donc absolument effacer l'objet avant de réutiliser la variable `pnom` pour sauvegarder l'adresse d'un nouvel objet. Sauvegarder cette adresse est essentiel, car il faudra, à un moment ou à un autre, effacer cette zone mémoire du tas.

En C++, il est recommandé d'identifier clairement les variables situées sur le tas de celles présentes sur la pile, en utilisant par exemple un nom de variable commençant par `p` (pointeur). Une variable C++ commençant par un `p` pourra signifier que nous avons affaire à un objet alloué avec un `new`, qui devra être effacé avec un `delete` en fin de procédure ou dans un destructeur pour des attributs d'objet ou de classe (variables statiques).

Make, javac et redondance

Il y a une fonctionnalité de `javac` qu'il nous faut absolument mentionner. Dans notre exemple précédent, si nous modifions le code de `Personne.java` et exécutons la compilation sur la classe de test seulement :

```
javac Test_personne.java
```

nous allons constater que la classe `Personne.class` sera aussi régénérée. `javac` est assez intelligent pour se comporter en fait comme un `make` qui sait déterminer, grâce aux dates de modification des fichiers, si une nouvelle compilation est nécessaire. Il apparaît donc une certaine redondance avec le `make` de GNU, qui reste tout de même beaucoup plus puissant.

Nos classes Java dans un paquet .jar

Nous reviendrons sur les bibliothèques de plusieurs classes Java regroupées dans un `.jar` et le `CLASSPATH` au chapitre 7.

Nous allons tout de même vous présenter un exemple simple d'une petite application regroupée dans un fichier `.jar` et comment l'exécuter. Nous verrons dans l'annexe E que NetBeans construit ses applications de cette manière.

Dans le répertoire `Chap04` des exemples, nous avons ajouté les deux fichiers suivants : `DansUnJar.bat` et `monManifest.txt` avec leur contenu respectif :

```
@rem fichier DansUnJar.bat
del personnes.jar

jar cvf personnes.jar Personne.class TestPersonne.class
java -classpath personnes.jar TestPersonne
java -classpath personnes.jar Personne

jar umf monManifest.txt personnes.jar
java -jar personnes.jar
```

et

```
Main-Class: TestPersonne
```

Le `@rem`, que nous utilisons ici pour la première fois, nous permet d'ajouter un commentaire dans un fichier `.bat`, ce qui est très pratique.

Le premier `jar` :

```
jar cvf personnes.jar Personne.class TestPersonne.class
```

permet de construire une archive compressée de nos deux classes :

```
manifest ajouté
ajout : Personne.class (entrée = 1285) (sortie = 693) (46% compressés)
ajout : TestPersonne.class (entrée = 499) (sortie = 334) (33% compressés)
```

Le `manifest` contient les informations sur l'archive : ici, un fichier par défaut sera généré. Il ne contient pas de point d'entrée et nous serons alors obligés d'exécuter notre programme sous la forme :

```
 java -classpath personnes.jar TestPersonne
```

Le point d'entrée `main()` de la classe `TestPersonne` sera activé. La classe `TestPersonne` est contenue dans l'archive `personnes.jar` avec les autres classes (ici, `Personne` uniquement).

Si nous n'avions pas de `classpath`, c'est-à-dire :

```
java TestPersonne
```

nous devrions alors copier les fichiers `Personne.class` et `TestPersonne.class` dans le répertoire courant. L'utilisation de fichiers `.jar` s'avère nécessaire lorsqu'il y a un grand nombre de classes ou d'autres fichiers venant d'ailleurs (pas d'exemples donnés dans cet ouvrage).

La forme

```
java -classpath personnes.jar Personne
```

fonctionne aussi mais, cette fois, le `main()` de la classe `Personne` est utilisé et la classe `TestPersonne`, dans l'archive `.jar`, ne sert à rien. Comme nous le reverrons à plusieurs occasions, un « petit » `main()` dans chaque classe pourrait s'avérer utile pour une vérification rapide de notre application.

La forme

```
jar umf monManifest.txt personnes.jar
```

est intéressante : elle ajoute à notre fichier `personnes.jar` une option du `manifest` dans l'archive, que nous avons introduite dans le fichier `monManifest.txt` ci-dessus, c'est-à-dire la définition du point d'entrée `main()`.

L'exécution avec

```
java -jar personnes.jar
```

va fonctionner, car le point d'entrée est défini dans le `manifest` de notre archive (les détails et options de notre fichier `personnes.jar`).

Pour plus d'informations et d'autres options, il faudra consulter la documentation de Java.

Comment étendre notre classe Personne ?

Au travers de l'élaboration de ces deux classes, nous nous sentons plus à l'aise et avons le sentiment d'acquérir une meilleure maîtrise du sujet. Cependant, ces deux classes restent très primitives et même inutilisables. Essayons maintenant d'imaginer comment étendre cette classe `Personne` afin qu'elle puisse acquérir plus de consistance. Pour ce faire, nous allons introduire de nouvelles données (attributs) ainsi que des fonctionnalités supplémentaires (méthodes), que nous allons définir comme suit :

- Cette personne travaille dans une entreprise qui lui a attribué un numéro de téléphone unique à quatre chiffres.
- Cette entreprise multinationale rémunère en dollars ses employés, mais ces derniers reçoivent leur salaire mensuel dans la monnaie du pays où ils travaillent. Le taux de change peut être modifié à tout instant.

- Tout salarié reçoit au minimum vingt jours de vacances par année, mais avec un jour de plus tous les cinq ans à partir de vingt-cinq ans. Un employé de 50 ans aura donc vingt-six jours de vacances. Le nombre de jours de vacances restant peut être transféré l'année suivante.

Pour traiter ce genre de problèmes, c'est-à-dire ajouter les méthodes et les attributs nécessaires à notre classe, nous voyons tout de suite les difficultés. À ce stade de nos connaissances, nous allons buter sur un certain nombre d'aspects ou de constructions que nous ne saurons pas comment traduire en programmation. En voici une petite liste :

- Nous ne savons pas encore traiter les tableaux. Un tableau à deux dimensions avec le nom et le taux de change est vraisemblable.
- Lorsque les données ont été originellement acquises, comme le nom, le prénom et la date de naissance, nous pouvons nous attendre à ce qu'elles soient stockées sur un support permanent, comme dans des fichiers ou une base de données. Le format des données devrait être tel qu'aussi bien la classe C++ que la classe Java puissent y accéder.
- Pour l'attribution ou l'acquisition des données, ainsi que le traitement des erreurs, il nous faudra maîtriser la conception et la réalisation des méthodes selon la spécification des paramètres nécessaires.

Ainsi, avant de passer à ce dernier point, qui consiste avant tout à étendre notre classe `Personne`, nous allons consacrer les prochains chapitres aux tableaux et aux entrées-sorties afin d'acquérir les connaissances suffisantes pour pouvoir exécuter ce travail proprement.

Diversion sur les structures C

Les structures (`struct`) sont un vieil héritage du langage C. Cependant, dès qu'un programmeur est confronté à d'anciennes techniques de programmation ou de code existant, il est obligé de les maîtriser et de les utiliser. Par conséquent, nous n'allons pas interdire ici l'utilisation des structures C, mais plutôt conseiller une approche plus orientée objet avec les classes C++.

La programmation Windows constitue un exemple de cas où le débutant ne devra surtout pas s'effrayer devant l'utilisation à toutes les sauces de `typedef` (nouvelle définition de type comme nous l'avons vu au chapitre 2) et de structures. Pour montrer la différence entre une structure C et une classe, nous avons choisi l'exemple d'une structure de données contenant un texte que nous pourrions dessiner à l'intérieur d'un rectangle d'une certaine dimension :

```
// Structure.cpp
#include <string>
#include <iostream>

using namespace std;

struct Rectangle1 {
```

```
 string texte;
 int    hauteur;
 int    largeur;
};

class Rectangle2 {
 public:
   string texte;
   int    hauteur;
   int    largeur;
};

int main(int argc, char* argv[])
{
 Rectangle1 mon_rec1;
 Rectangle2 mon_rec2;
 Rectangle1 *pmon_rec3;
 pmon_rec3 = new Rectangle1;

 mon_rec1.texte = mon_rec2.texte = pmon_rec3->texte = "Salut";
 mon_rec1.hauteur = mon_rec2.hauteur = pmon_rec3->hauteur = 10;
 mon_rec1.largeur = mon_rec2.largeur = pmon_rec3->largeur = 100;

 cout << mon_rec1.texte << ": " << mon_rec1.hauteur;
 cout << " sur " << mon_rec1.largeur << endl;
 cout << mon_rec2.texte << ": " << mon_rec2.hauteur;
 cout << " sur " << mon_rec2.largeur << endl;
 cout << pmon_rec3->texte << ": " << pmon_rec3->hauteur;
 cout << " sur " << pmon_rec3->largeur << endl;

 return 0;
}
```

Nous commencerons par la classe `Rectangle2` où le mot-clé `public` va permettre d'accéder aux attributs comme dans une structure C. Dans le paragraphe consacré à la définition des objets d'une classe en C++, nous avons affirmé qu'en programmation objet, il est déconseillé de rendre les attributs publics : pourtant, c'est malheureusement ce que nous faisons ici. Avec cette construction, il est alors possible d'accéder directement à l'attribut lorsque nous possédons une instance de cette classe. C'est le cas par exemple de `mon_rec2.texte`. C'est exactement la même forme qu'une structure C où tous les attributs sont publics.

Nous remarquons qu'il est aussi possible d'utiliser l'opérateur `new` pour des structures, alors qu'un programmeur C utilisera plutôt `malloc()`. Notons aussi l'affectation des attributs au moyen de l'opérateur `=` répété trois fois sur la même ligne : cette construction est tout à fait raisonnable.

La directive `struct` en programmation C est essentielle car elle permet de structurer des données sous un même chapeau. En programmation C++, tout comme en Java qui ne possède pas de structure, nous définirons une classe avec les recommandations et les

méthodes d'accès appropriées. Cependant, aucune raison ne nous empêche d'utiliser des structures comme attributs de classe pour rendre notre code plus soigné.

Résumé

Nous sommes encore loin de maîtriser tous les concepts de classes et d'objets en Java et en C++. Avant d'y revenir plus en détail, il nous faut étudier d'autres aspects plus généraux, afin de progresser par étapes dans l'apprentissage de ces deux langages.

Exercices

1. Créer une classe nommée Dette qui va conserver une dette initiale en début d'année, un taux d'intérêt applicable sur cette somme et un remboursement mensuel fixe.
2. Écrire une méthode qui nous affiche la dette restante après une année sans préciser les centimes. Documenter la classe comme décrit dans ce chapitre.

5

On enchaîne avec les tableaux

Définir, utiliser et manipuler des tableaux (*arrays*) constitue l'un des domaines essentiels de tout langage de programmation. Lorsque nous parlons de tableaux, nous pensons plus généralement à une présentation multicolonnes ou à une feuille de calcul. Cependant, le tableau le plus simple reste encore celui composé d'une seule ligne de plusieurs éléments. Ces derniers peuvent être aussi bien composés de nombres que de toutes sortes d'objets de différents types. Un mot, une phrase, voire un texte entier sont aussi des tableaux, dans le sens qu'ils peuvent représenter une zone continue en mémoire, dont nous connaissons l'adresse de départ et dont chaque caractère peut être accessible individuellement par un index.

Voici une petite liste d'exemples de tableaux :

```
int tableau1[10];
long[] tableau2 = new long[10];
char tableau[8][8];
```

ainsi que ces deux chaînes de caractères, respectivement en C++ et en Java :

```
char *message1 = "Mon premier message";
String message2 = new String("Mon premier message");
```

Certaines de ces constructions sont possibles dans les deux langages, tandis que d'autres subiront quelques changements. En Java, la conception des tableaux a été optimalisée par des contraintes et des améliorations apportées au langage, tandis qu'en C++, de nouvelles classes ont amélioré la programmation par rapport au langage C ou C++ classique. Il existe aussi d'autres formes de collections d'objets comme les `vector` et `list`, qui seront

traitées plus loin, au chapitre 14. Ces dernières classes vont nous permettre de simplifier la création et l'accès à des tableaux de dimensions variables.

Tableaux d'entiers

Deux aspects essentiels doivent être abordés lors de la création de tableaux : leur définition et leur initialisation. Prenons l'exemple d'un tableau contenant les cinq plus petits nombres premiers. Nous devrons d'abord créer le tableau, avant de déposer une valeur dans chaque élément. Si nous écrivons en C++:

```
// premier1.cpp
#include <iostream>

using namespace std;

const int dim =5;

int main() {
  int premier[dim];

  for (int i = 0; i <= dim; i++) {
    cout << premier[i] << " ";
  }
  cout << endl;

  return 0;
}
```

et son équivalent Java :

```
public class Premier1 {
  static final int dim = 5;

  public static void main(String[] args) {
    int[] premier = new int[dim];

    for (int i = 0; i <= dim; i++) {
      System.out.print(premier[i] + " ");
    }
    System.out.println("");
  }
}
```

nous constatons que si les tableaux sont bien créés, ils ne sont cependant pas initialisés par le programme. Le résultat du programme C++ sera :

```
39124524 2143490060 4198431 4276224 4276228 39124520
```

alors que celui de Java, bien différent, nous retourne une erreur :

```
0 0 0 0 0 java.lang.ArrayIndexOutOfBoundsException
```

```
at Premier1.main(Compiled Code)
Exception in thread "main"
```

Commençons par l'erreur en Java. L'instruction `premier[i]` permet d'accéder à l'élément `i` du tableau. L'exception `ArrayIndexOutOfBoundsException` nous indique un dépassement de l'index en dehors des limites : nous avons écrit `i <= dim`, alors qu'il fallait écrire `i < dim`. Comme l'index `i` commence à 0, que nous allons jusqu'à 5 (sixième élément du tableau `premier`) et que Java est beaucoup mieux construit que C++, il nous reporte une erreur. Ce n'est pas le cas en C++, où même une écriture dans ce sixième octet est possible et pourrait s'avérer catastrophique.

Notons également l'initialisation à 0 en Java, alors qu'en C++ nous avons vraiment n'importe quoi (`39124524 …`). En Java, même si chaque élément n'est pas assigné à une valeur par le programme, le `new int[dim]` initialisera chaque élément à 0. Ceci est valable pour les types primitifs. En Java, la forme C++ sans le `new`, `int premier[dim]`, n'est pas possible et donnerait une erreur à la compilation. Nous remarquons, d'une manière générale, que le langage Java est plus précis et nous empêche de créer des erreurs qui peuvent être extrêmement difficiles à déceler.

La forme en Java :

```
int[] premier = new int[dim];
```

avec les `[]` suivant directement le type, est la forme usuellement pratiquée et correspondant en C++ à :

```
int premier[dim];
```

La forme :

```
int premier[] = new int[dim];
```

est tout aussi correcte en Java, mais sans doute moins parlante. Le code Java :

```
static final int dim = 5;
```

représente l'équivalent en C++ de :

```
const int dim = 5;
```

qui est dans notre cas une constante globale, car définie avant le `main()`. En Java, ce n'est pas possible, et une variable constante doit être une variable de classe. `final` signifie que la variable `dim` ne peut être modifiée ; `static` indique que toutes les instances de la classe `Premier1` utiliseront la même variable. Si nous voulions à présent initialiser le tableau avec nos nombres premiers, les codes C++ et Java se présenteraient ainsi :

```
// premier2.cpp
#include <iostream>

using namespace std;

const int dim =5;
```

```
int main() {
  int premier[5] = {1, 2, 3, 5, 7};

  for (int i = 0; i < dim; i++) {
    cout << premier[i] << " ";
  }
  cout << endl;

  return 0;
}
```

```
public class Premier2 {
  public static void main(String[] args) {
    int[] premier = {1, 2, 3, 5, 7 };

    for (int i = 0; i < premier.length; i++) {
      System.out.print(premier[i] + " ");
    }
    System.out.println("");
  }
}
```

Et cela avec le même résultat :

```
1 2 3 5 7
```

En C++, les deux formes `int premier[5]` et `int premier[ ]` sont possibles. Ce n'est pas le cas en Java, où uniquement le `[ ]` est permis. En Java, cette forme d'initialisation sans new est acceptée par le compilateur.

Nous retrouvons en Java le `length` que nous avions déjà rencontré au chapitre 1. Il nous donne la dimension du tableau.

Copie de tableau d'entiers en Java

Lorsque nous copions des tableaux, opération qui se rencontre souvent, nous devons toujours vérifier que notre code est correct et ne travaille pas en fait sur le tableau d'origine. En Java, si nous écrivons :

```
public class Premier3 {
  public static  void main(String[] args) {
    int[] premier1 = {1, 2, 3, 5, 7 };
    int[] premier2;

    premier2 = premier1;
    premier2[3] = 11;

    for (int i = 0; i < premier1.length; i++) {
      System.out.print(premier1[i] + " ");
    }
    System.out.println("");
  }
}
```

nous aurons le résultat :

```
1 2 3 11 7
```

Nous constatons que le tableau `premier2` est en fait le tableau `premier1`. L'instruction `premier2[3]` permet de vérifier que nous accédons bien au tableau `premier1`. Si nous voulions avoir deux tableaux distincts initialisés avec le même contenu, il nous faudrait procéder comme suit :

```
public class Premier4 {
  public static void main(String[] args) {
    int[] premier1 = {1, 2, 3, 5, 7 };
    int[] premier2 = new int[5];

    for (int i = 0; i < premier1.length; i++) {
      premier2[i] = premier1[i];
    }
  }
}
```

Ce dernier code est applicable en C++ pour la copie de tableaux, à condition d'avoir pour la déclaration de `premier2` :

```
int premier2[5];
```

Tableau dynamique en C++

Cette partie étant d'un niveau de complexité plus élevé, il est tout à fait envisageable de passer au sujet suivant et d'y revenir à une autre occasion.

Il est possible et parfois nécessaire de créer des tableaux dynamiques en C++. Voici une manière de procéder :

```
// premier3.cpp
#include <iostream>

using namespace std;

const int dim = 5;

int main()
{
 int premier[5] = {1, 2, 3, 5, 7};
 int **ppremier1;
 int **ppremier2;
 int i = 0;

 ppremier1 = (int **)new int[dim];
 ppremier2 = ppremier1;
```

```
  for (i = 0; i < dim; i++) {
    ppremier1[i] = new int(0);
    *ppremier1[i] = premier[i];
  }

  for (i = 0; i < dim; i++) {
    cout << *ppremier2[i] << " ";
    delete ppremier2[i];
  }
  cout << endl;

  delete[] ppremier1;
  return 0;
}
```

Ce petit exercice nous montre comment copier un tableau présent sur la pile (*stack*) dans un tableau alloué dynamiquement.

La première remarque qui s'impose concerne la variable `ppremier2`, qui est là uniquement pour montrer comment acquérir l'adresse d'un tableau existant. `ppremier2` va simplement pointer sur le tableau dynamique `ppremier1`. Le terme dynamique est utilisé pour indiquer que le tableau est alloué avec l'opérateur `new` ; autrement dit, le tableau sera présent dans le tas (*heap*) et devra être libéré lorsque nous ne l'utiliserons plus.

Nous savons déjà qu'un `int *nombre` représente un pointeur à un nombre. Ici, la présence d'un double `**` indique un pointeur à un tableau. La première opération consiste à allouer de la mémoire nécessaire pour contenir cinq pointeurs. Cela se fait au moyen du `new int[dim]`. Le transtypage (`int **`) a été nécessaire avec ce compilateur, car le programme, qui pourtant acceptait de compiler, refusait d'exécuter cette instruction alors que d'autres compilateurs produisaient du code parfaitement exécutable !

La deuxième opération consiste à allouer un entier, individuellement, pour chaque composant du tableau :

```
ppremier1[i] = new int(0);
```

Nous profitons de la boucle pour copier chaque élément :

```
*ppremier1[i] = premier[i];
```

et remarquer la notation avec le pointeur (`*`) et le nom de la variable commençant par un premier `p`. C'est bien un pointeur au i^e élément dans le tableau.

Effacer les ressources doit d'abord se faire sur les éléments, et ensuite sur le tableau de pointeurs. Pour ce dernier, la présence du `[]` suivant le `delete` permet d'indiquer que nous avons un tableau à effacer et non pas un seul élément.

Tableaux multidimensionnels

Si nous devons programmer le jeu d'échecs ou le jeu d'Othello, dont nous verrons les règles et les détails au chapitre 22, il nous faut travailler avec un tableau de 8 par 8. Par exemple :

```
int jeu[8][8];
```

Le coin supérieur gauche de l'échiquier sera référencé par `jeu[0][0]` et l'inférieur droit par `jeu[7][7]`. Durant la conception du jeu, nous pourrons nous rendre compte que nous devons continuellement tester les bords pour savoir si un déplacement est possible. Il est donc avantageux d'ajouter une case supplémentaire de chaque côté. Au jeu d'échecs, si nous travaillons avec un tableau de déplacement possible pour le cavalier (2,1), il faut donc vraisemblablement étendre le tableau à 12 par 12. Nous aurons donc :

```
int othello[10][10];
int echec[12][12];
```

Avec l'exemple qui suit, écrit en Java et en C++, nous allons montrer comment initialiser un tableau pour le jeu d'Othello avec les quatre pions d'origine et les conditions suivantes :

- –1 : bord extérieur ;
- 0 : position libre ;
- 1 : pion noir ;
- 2 : pion blanc.

Le jeu d'Othello en Java

```
public class Othello1 {
  static final int dim = 10;

  int[][] othello = new int[dim][dim];

  public Othello1() {
    int i = 0;  // position horizontale
    int j = 0;  // position verticale

    for (i = 0; i < dim; i++) {  // bord à -1
      othello[i][0] = -1;
      othello[i][dim-1] = -1;
      othello[0][i] = -1;
      othello[dim-1][i] = -1;
    }

    for (j = 1; j < dim-1; j++) {  // intérieur vide
      for (i = 1; i < dim-1; i++) {
        othello[i][j] = 0;
      }
    }
```

```
    othello[4][5] = 1;  // blanc
    othello[5][4] = 1;  // blanc
    othello[4][4] = 2;  // noir
    othello[5][5] = 2;  // noir
  }

  public void test1() {
    int i = 0;  // position horizontale
    int j = 0;  // position verticale

    for (j = 0; j < dim; j++) {
      for (i = 0; i < dim; i++) {
        if (othello[i][j] >= 0) System.out.print(" ");
        System.out.print(othello[i][j]);
      }
      System.out.println();
    }
  }

  public static void main(String[] args) {
   Othello1 monjeu = new Othello1();
   monjeu.test1();
  }
}
```

Le jeu d'Othello en C++

```
// othello1.cpp
#include <iostream>

using namespace std;

class Othello1 {
  private:
    const static int dim = 10;  // 8 x 8 plus les bords
    int othello[dim][dim];      // le jeu

  public:
    Othello1();
    void un_test();
};

Othello1::Othello1()
{
  int i = 0;  // position horizontale
  int j = 0;  // position verticale

  for (i = 0; i < dim; i++) {  // bord à -1
    othello[i][0] = -1;
```

```
    othello[i][dim-1] = -1;
    othello[0][i] = -1;
    othello[dim-1][i] = -1;
  }

  for (j = 1; j < dim-1; j++) {  // intérieur vide
    for (i = 1; i < dim-1; i++) {
      othello[i][j] = 0;
    }
  }

  othello[4][5] = 1;  // blanc
  othello[5][4] = 1;  // blanc
  othello[4][4] = 2;  // noir
  othello[5][5] = 2;  // noir
}

void Othello1::un_test()
{
 int i = 0;  // position horizontale
 int j = 0;  // position verticale

 for (j = 0; j < dim; j++) {
   for (i = 0; i < dim; i++) {
     if (othello[i][j] >= 0) cout << " ";
     cout << othello[i][j] << " ";
   }
   cout << endl;
 }
}

int main()
{
 Othello1 lejeu;
 lejeu.un_test();
}
```

Le résultat est identique pour les deux exemples :

```
-1-1-1-1-1-1-1-1-1-1
-1 0 0 0 0 0 0 0 0-1
-1 0 0 0 0 0 0 0 0-1
-1 0 0 0 0 0 0 0 0-1
-1 0 0 0 2 1 0 0 0-1
-1 0 0 0 1 2 0 0 0-1
-1 0 0 0 0 0 0 0 0-1
-1 0 0 0 0 0 0 0 0-1
-1 0 0 0 0 0 0 0 0-1
-1-1-1-1-1-1-1-1-1-1
```

Devant ce résultat, cela nous donnera certainement l'envie de programmer ce jeu et d'aller un peu plus loin. C'est ce que nous ferons à plusieurs occasions dans cet ouvrage,

et plus particulièrement au chapitre 20. En effet, Othello est un jeu tout à fait abordable pour un débutant et un excellent exercice de conception de programme. Sa seule vraie difficulté est d'atteindre un niveau de jeu acceptable, ce qui est loin d'être évident.

Il y a très peu de choses à dire car le code est suffisamment explicite. La manière de définir un tableau à deux dimensions est similaire au cas d'un tableau à une seule dimension, le `[ ]` devenant simplement `[ ][ ]`. Lors de l'initialisation du tableau, il faut travailler avec deux index, l'un sur la position verticale, l'autre sur l'horizontale. En analysant le code, nous pourrions constater certaines redondances et remarquer que certaines cases sont assignées deux fois, mais cela au profit d'un code plus simple et plus lisible.

La manière de procéder pour un programmeur C peut paraître saugrenue. Nous voulions montrer comment utiliser un tableau multidimensionnel, et nous commençons par écrire une classe. En fait, il aurait été tout à fait possible d'écrire cette ébauche de programme en C classique, avec un tableau global. En Java, nous n'avons pas le choix, et, en fin de compte, la solution C++ d'écrire une classe est élégante et orientée objet. Est-ce vraiment plus difficile ? Nous ne le pensons pas, surtout si nous avons l'intention d'écrire le jeu complet par la suite.

Le cavalier du jeu d'échecs

Mais revenons au jeu d'échecs, avec notre cavalier qui peut se déplacer de 2 et 1 dans huit cases différentes autour de sa position actuelle. Le code suivant nous montre comment initialiser un tableau avec ces huit paires de valeurs.

```
int cavalier[8][2] = {  // C++
   {1, 2},   {2, 1},   {2, -1}, {1, -2},
   {-1, -2}, {-2, -1}, {-2, 1}, {-1, 2},
 };

 int cavalier[ ][ ] = {  // Java
   {1, 2},   {2, 1},   {2, -1}, {1, -2},
   {-1, -2}, {-2, -1}, {-2, 1}, {-1, 2},
 };
```

En Java, si nous le désirons, il est possible d'obtenir la dimension de chaque composant de cette manière :

```
cavalier.length;        //résultat : 8
cavalier[0].length;     //résultat : 2
```

Chaînes de caractères en C

Une chaîne de caractères en C et C++ est en fait un tableau de caractères. Lorsque nous avons besoin d'un texte ou d'un message, nous pouvons simplement utiliser un pointeur, dont nous voyons la forme ici :

```
// message.cpp
#include <iostream>
#include <cstring>

using namespace std;

int main()
{
  char *message1 = "Mon message";
  cout << message1 << endl;

  int longueur   = strlen(message1);
  char message2[longueur + 1];

  strcpy(message2, message1);
  message2[4] = 'M';
  cout << message2 << endl;

  for (int i = 0; i < longueur; i++) {
    cout << message2[i];
  }
  cout << endl;

  char *message3;
  message3 = new char[longueur + 2];

  strcpy(message3, message1);
  message3[longueur] = 'r';
  message3[longueur + 1] = 0;
  cout << message3 << endl;

  delete[] message3;
  return 0;
}
```

Et le résultat correspondant :

```
Mon message
Mon Message
Mon Message
Mon messager
```

Ce type de code, plus proche du C que du C++, se rencontre encore très souvent de nos jours. Ce n'est certainement pas la meilleure manière de procéder, mais il est cependant essentiel de maîtriser ces différentes constructions. La variable `message1` est un pointeur à un texte fixe en mémoire que nous pouvons utiliser dans différentes fonctions C ou C++. Le pointeur reste et restera à la mode en C++, en particulier comme paramètre pour le `cout`.

Les deux `cout` suivants sont équivalents, bien que l'utilisation de la variable `bonjour` ne soit pratique que si elle est réutilisée :

```
char *bonjour = "Bonjour";
cout << bonjour;
cout << "Bonjour";
```

Les fonctions C `strlen()` et `strcpy()` sont encore beaucoup utilisées, mais il faut rester très prudent. Si nous écrivons :

```
char *message1 = "1234";
char message2[3];
strcpy(message2, message1);
```

le `message1` est copié dans un tampon trop court. `message2` doit avoir au moins une longueur de 5, c'est-à-dire 4 plus un zéro qui détermine la fin du texte.

Dans le programme précédent, nous voyons aussi comment accéder à des caractères individuels dans la chaîne pour les modifier ou pour sortir les différents textes sur la console.

La manière d'utiliser `message2` comme pointeur est en fait reconnue par le compilateur, et les deux instructions suivantes sont équivalentes :

```
cout << message2 << endl;
cout << &message2[0] << endl;
```

elles nous fournissent l'adresse du premier caractère de la chaîne. `cout` s'arrêtera au moment où le caractère 0 sera trouvé dans la chaîne. Si nous ajoutions une instruction comme celle-ci avant le `cout` :

```
message2[1] = 0;
cout << message2 << endl;
```

seule la lettre initiale `'M'` apparaîtrait !

Pour terminer, nous avons ajouté une allocation dynamique, avec le `new`, d'un message qui sera effacé de la mémoire, avec le `delete[]`, lorsqu'il ne sera plus utilisé. La longueur du tampon de `message3` devra être suffisante pour contenir un caractère supplémentaire, car nous y ajouterons la lettre `'r'` du messager. Ce dernier caractère vient se placer sur le `0` terminant la copie précédente avec le `strcpy()`. Enfin, il faut remettre un `0` à l'index suivant.

Le fichier d'en-tête `cstring` contient la définition des fonctions C `strcpy()` et `strlen()`, ainsi que bien d'autres fonctions mises à disposition, comme ajouter des chaînes les unes derrière les autres ou rechercher des séquences de caractères. Le problème, cependant, reste le même : le risque de définir un tampon trop court ou de travailler avec un index en dehors des limites !

Les String de Java

Il n'y a pas de chaînes de caractères en Java. Cependant, les deux instructions suivantes sont identiques et correctes. La première forme n'est acceptée que pour les `String`, et la deuxième correspond à la forme standard de la création d'objet. `String` est bien une classe et non un type primitif comme `int`, `char` ou `long`.

```
String message1 = "Mon message";
String message1 = new String("Mon message");
```

Le programme C++ que nous venons de voir, `message.cpp`, ne peut être transcrit directement en Java. En effet, il n'est pas possible d'accéder à un caractère avec son index, tel que `message2[i]`. Nous allons en analyser les détails :

```
public class Message1 {
  public static void main(String[] args) {
     String message1 = new String("Mon message");
     System.out.println(message1);
     System.out.println(message1.charAt(4));

     String message2 = message1.substring(0, 4) +
       'M' + message1.substring(5, message1.length()) + 'r';
     System.out.println(message2);

     StringBuffer messageb3 = new StringBuffer(message1);
     messageb3.setCharAt(4, 'M');
     messageb3.append('r');
     System.out.println(messageb3);
  }
}
```

La méthode `charAt(index)` de la classe `String` permet d'accéder à un caractère individuellement. De plus, il n'y a pas de méthodes pour modifier quoi que ce soit à l'intérieur d'un `String` en Java. Les objets de cette classe sont immuables, comme un `const string` en C++.

De plus, la classe `String` va vérifier la position de l'index. Si nous avions malencontreusement écrit ceci :

```
System.out.println(message1.charAt(message1.length()));
```

nous aurions reçu alors un :

```
java.lang.StringIndexOutOfBoundsException: String index out of range: 11
at java.lang.String.charAt(Compiled Code)
at Message1.main(Message1.java:5)
```

qui nous indique que nous essayons d'accéder à un élément en dehors des limites du `String` (`StringIndexOutOfBoundsException`). C'est propre et beaucoup moins dangereux qu'en C++. Il faut noter que nous accédons ici à la position où se situait notre 0 terminant une chaîne de caractères en C/C++. En Java, ce n'est pas nécessaire, car il possède un autre mécanisme.

Si nous voulions changer le `m` minuscule en `M` majuscule, il nous faudrait, par exemple, recomposer le `String` en utilisant la méthode `substring(index1, index2)`. Cette dernière est très particulière : `index1` est la position de départ, alors qu'`index2` est la position dans la chaîne du premier caractère que nous ne voulons plus copier !

En revanche, la classe `StringBuffer` est modifiable (*mutable*). Non seulement nous pouvons modifier son contenu (ici, avec `setCharAt(4, 'M')`, qui va nous positionner notre fameux

M majuscule), mais nous pouvons aussi étendre notre chaîne en lui ajoutant des caractères. La mémoire sera réallouée automatiquement, sans devoir passer par du code aussi complexe que celui-ci en C++ :

```
char *texte1 = "Un deux trois ";
char *texte2 = "quatre cinq six";
char texte1_2[strlen(texte1) + strlen(texte2) + 1];
strcpy(texte1_2, texte1);
strcpy(texte1_2 + strlen(texte1), texte2);
```

où nous utilisons des fonctions C pour recalculer l'espace nécessaire. La dernière instruction est assez particulière. Le `texte2` est copié dans le nouveau tableau `texte1_2` depuis la position du `0` généré par le `strcpy()` précédent. L'instruction :

```
texte1_2 + strlen(texte1)
```

est simplement le calcul de la position d'une adresse mémoire. Nous verrons, au chapitre 9, que nous avons d'autres outils à disposition en C++ pour former et combiner des chaînes de caractères variables. Et ceci en plus de la classe `string`, que nous allons étudier à présent.

Les string du C++, un nouvel atout

La classe `string` est apparue récemment en C++. C'est l'une des classes essentielles du Standard C++. Pendant des années, les programmeurs ont utilisé des tableaux de caractères, comme nous venons de le voir, ont créé leurs propres classes `String` ou ont utilisé des produits commerciaux. L'utilisation de la classe `string` va sans doute se généraliser, et c'est avant tout au bénéfice de la simplicité et de la réutilisation du code. Nous allons reprendre à présent une partie des fonctionnalités que nous venons d'analyser au travers de ce code :

```
// strings.cpp
#include <iostream>
#include <string>

using namespace std;

int main()
{
 string message1 = "Mon message";
 cout << "1: " << message1 << endl;
 cout << "2: " << message1[4] << endl;
 cout << "3: " << message1[100] << endl;  //indéfini

 string message2;
 message2 = message1;
 message2[4] = 'M';
 message2.append("r");
```

```
    cout << "4: " << message1 << endl;
    cout << "5: " << message2 << endl;

    string *pmessage1;
    string *pmessage2;
    pmessage1 = new string("Mon message");
    pmessage2 = pmessage1;
    (*pmessage2)[5]  = 'a';
    cout << "6: " << *pmessage2 << endl;
    delete pmessage1;
    return 0;
}
```

Et le résultat donne :

```
1: Mon message
2: m
3: ?
4: Mon message
5: Mon Messager
6: Mon massage
```

Le constructeur du `string` accepte un pointeur à un tableau de caractères, et l'opérateur `[ ]` permet d'accéder à des caractères individuels. Nous voyons qu'il est toujours possible d'utiliser un index (notre `100`) totalement en dehors des limites, et que le résultat obtenu reste indéfini (`3: ?`), sans autres mécanismes de détection d'erreur. L'analyse de l'instruction :

```
message2 = message1;
```

est essentielle. Le contenu du message1 est copié dans le `message2`. Les modifications apportées ensuite à `message2`, c'est-à-dire la lettre `M` à l'index 4 et l'ajout de la lettre `r` en fin de `string` avec la réallocation automatique de la mémoire, ne touchent en rien au `message1`. En revanche, si nous écrivions en Java :

```
StringBuffer message1 = new StringBuffer("Mon message");
StringBuffer message2;
message2 = message1;
message2.setCharAt(4, 'M');
System.out.println(message1);  // résultat : Mon Message
```

`message1` et `message2` seraient les mêmes objets : ils utilisent la même référence. Cependant, avec le code suivant, nous procédons bien à une copie :

```
StringBuffer message1 = new StringBuffer("Mon message");
StringBuffer message2 = new StringBuffer(message1.toString());
message2.setCharAt(4, 'M');
System.out.println(message1);  // résultat : Mon message
```

Le `message1` d'origine ne sera pas modifié. Le `toString()` qui retourne un `String` est nécessaire, car le constructeur de `StringBuffer` n'accepte pas de `StringBuffer` comme paramètre.

Ensuite, il faut revenir au code C++ ci-dessus (`strings.cpp`), où nous avons utilisé un `string` dynamique `pmessage1` et alloué avec `new` un nouvel objet. `pmessage2` est non pas un deuxième

objet, mais une adresse qui est assignée à `pmessage1`. En modifiant le contenu de `pmessage2`, nous touchons en fait au `string` dynamique `pmessage1`. Il existe donc une similitude avec le code Java ci-dessus, mais il est très important de maîtriser les différences. L'instruction :

```
cout << *pmessage2 << endl;
```

est bien correcte. Si nous omettons le caractère `*`, nous recevrons l'adresse mémoire en hexadécimal où se trouve l'objet `*pmessage1` de la classe `string` ! Cela pourrait être utilisé à des fins de test, pour vérifier que nous travaillons bien sur la bonne zone mémoire.

Enfin, nous noterons encore une fois dans le code qui suit :

```
// strings2.cpp
#include <iostream>
#include <string>

using namespace std;

int main()
{
 string *pmessage1;
 string *pmessage2;
 pmessage1 = new string("Mon premier message");
 pmessage2 = pmessage1;
 pmessage1 = new string("Mon second message");

 cout << "pmessage1: " << *pmessage1 << endl;  //Mon second message
 cout << "pmessage2: " << *pmessage2 << endl;  //Mon premier message

 delete pmessage1;
 delete pmessage2;

 return 0;
}
```

qu'il faut rester très prudent afin d'éviter des fuites de mémoire. Bien que ce code soit un peu farfelu, il est tout à fait correct. Nous avons deux `new` et deux `delete`. Est-ce vraiment juste ? Le second `new` se fait aussi sur la variable `pmessage1`, mais sa première valeur a été sauvée dans `pmessage2`. Le `delete pmessage2;` va donc bien nous effacer la ressource du premier message. Ce code est aussi un avertissement de ce qu'un programmeur C++ ne devrait jamais faire. Java ne nous le permettrait pas, et c'est tout à son honneur.

Les méthodes des classes String (Java) et string (C++)

Lors du traitement des chaînes de caractères en Java ou en C++, le programmeur doit avant tout consulter la documentation (API) pour rechercher une fonction, c'est-à-dire une méthode, dont il a besoin. La qualité d'un programmeur est souvent en relation directe avec sa facilité de naviguer dans la documentation pour y découvrir ce qu'il recherche. Une autre possibilité, qui se révèle aussi très efficace, est de collectionner de nombreux

exemples dans une hiérarchie simple où des outils de recherches composées, comme `egrep` sous Linux, pourraient beaucoup apporter (voir fin de l'annexe B).

Certaines méthodes seront évidemment disponibles dans un langage et pas dans l'autre. Il faudra souvent faire un choix : soit privilégier la performance mais ce au prix de longues heures de programmation, soit préférer une programmation rapide au détriment de la performance. Nous allons présenter ici un petit aperçu de quelques méthodes à disposition, puis nous en donnerons quelques variantes en exercices. Nous commencerons par du code en Java :

```
public class TestString {
  public static void main(String[] args) {
    String texte1 = "  abcd1234abcd5678abcd   ";
    System.out.println("[" + texte1 + "]");

    texte1 = texte1.trim();
    System.out.println("[" + texte1 + "]");

    int index = texte1.indexOf("bcd", texte1.indexOf("bcd") + 3);
    String texte2 = texte1.substring(0, index) + texte1.substring(index
    ➥+ 3).toUpperCase();
    System.out.println("[" + texte2 + "]");
  }
}
```

qui fournit comme résultat :

```
[  abcd1234abcd5678abcd   ]
[abcd1234abcd5678abcd]
[abcd1234a5678ABCD]
```

Le `trim()` nous efface les espaces au début et à la fin, ce qui peut s'avérer pratique pour un traitement de texte. Les deux `indexOf()` vont nous retourner l'index du départ du deuxième `"bcd"`. Enfin, nous allons extraire, avec `substring()`, tout ce qui vient avant le deuxième `"bcd"` et ensuite y ajouter tout ce qui suit ce même `"bcd"`, mais en le convertissant en majuscules avec la méthode `toUpperCase()`. Nous y trouvons donc notre bonheur, tout en nous demandant à quoi cela peut bien servir !

Nous allons à présent passer à la classe `string` du C++ et montrer aussi quelques possibilités d'utilisation de ces méthodes :

```
// teststring.cpp
#include <iostream>
#include <string>

using namespace std;

int main()
{
  string aref = "<a href=\"mailto:Jean-Bernard@Boichat.ch\">";
```

```
  for (int i = 0; i < aref.length(); i++) {
    if ((aref[i] >= 'A') && (aref[i] <= 'Z')) {
      aref[i] += 32;
    }
  }

  int index1 = aref.find("href=\"");
  if (index1 >= 0) {
    index1 += 6;
    int index2 = aref.find("\"", index1);
    if (index2 >= 0) {
      string =  aref.substr(index1, index2 - index1);

      if (urlstr.find("mailto:") == 0) {
        string emaila = urlstr.substr(7);
        int indexa = emaila.find("@");
        if (indexa <= 0) {
          cout << "Adresse email invalide: " << emaila << endl;
        }
        else {
          cout << "Adresse email: " << emaila << endl;
        }
      }
    }
  }
}
```

Et le résultat produit :

```
Adresse email: jean-bernard@boichat.ch
```

qui correspond à l'extraction d'une adresse e-mail dans un morceau (un très petit morceau) d'une page Web.

Dans notre adresse URL, `<a href=\"mailto:Jean-Bernard@Boichat.ch\">`, nous remarquons les codes d'échappement `\"` qui sont nécessaires pour affecter le `string` dans notre code. Nous avons bien un seul caractère `"` qui va délimiter ce que nous recherchons, à savoir une adresse URL.

La première boucle `for()` est une conversion de toutes les majuscules en minuscules. Pour du texte, en particulier dans une page Web, cette conversion ne serait pas très pertinente, mais ici il est raisonnable de ne garder que les minuscules pour ces adresses URL, dont les majuscules sont converties en minuscules de toute manière. Cette manière de procéder, sans utiliser de méthodes de classe, est tout à fait envisageable lorsque aucune méthode correspondante n'existe ou lorsque nous voulons gagner en efficacité. Dans la boucle `for()`, nous pourrions aussi ajouter d'autres fonctionnalités telles que celle de rechercher un caractère particulier ou un cas d'erreur. Nous verrons, dans le chapitre 15 sur les performances, que tout appel de fonction est une perte sensible en performance. Notons aussi que nous utilisons l'opérateur `[]` de la classe `string`, qui se révèle en fait similaire à la méthode `charAt()` en Java. Nous reviendrons sur ces détails dans les chapitres suivants.

Le reste du code correspond plutôt à une présentation de quelques méthodes de la classe `string` qu'à un code propre et complet. De nombreux cas d'erreurs sont traités, mais il reste encore du travail. La première recherche :

```
int index1 = aref.find("href=\"");
```

nous donne l'index du `href="` dans le `string`. Dans un document HTML, il y a d'autres codes, mais comme nous cherchons d'abord une adresse e-mail, cela semble justifié. La deuxième recherche (`aref.find("\"", index1)`) possède cette fois un second argument, l'`index1`, afin de rechercher la position du dernier `"` à partir de la position `index1 += 6`. Nous utilisons la méthode `substr()` avec un ou deux paramètres. Nous nous souvenons de la méthode Java `substring()`, dont le deuxième paramètre était un index. Ici, ce n'est pas une position, mais une longueur.

Le `string urlstr` est un état intermédiaire. Il contient ici `mailto:jean-bernard@boichat.ch`. Nous aurions pu avoir, par exemple, `http://www.w3.org/` et transformer notre code en un traitement complet d'une page Web. L'une des premières tâches serait d'éliminer les valeurs fixes comme 6, 7, `href` ou `mailto` par des constantes ou des dictionnaires. Enfin, nous remarquerons que ce travail, très concret cette fois-ci, n'est en fait pas bien différent de notre exemple Java, `TestString`, qui nous paraissait totalement abstrait et inutile.

Résumé

Nous progressons, nous le sentons, mais beaucoup reste à faire. Nos chaînes de caractères sont encore malheureusement beaucoup trop statiques. Au chapitre 14, nous reviendrons sur le traitement des listes dynamiques (`list` et `vector`, par exemple) qui sont essentielles pour l'élaboration d'applications contenant des objets persistants et modifiables.

Exercices

1. Définir un tableau qui nous permet, dans le jeu d'Othello, d'identifier les huit directions possibles à partir d'une position déterminée.
2. Pour le jeu d'Othello, calculer le nombre de positions prises par chaque couleur dans le tableau entier. Ce nombre correspond au score.
3. Écrire, d'une manière simple, le jeu de la vie avec un petit tableau et quelques exemples pour vérifier la logique. Nous pouvons trouver une description du jeu de la vie écrite par J.Conway dans *Scientific American* d'octobre 1970. Les règles sont les suivantes :
   - Une cellule continue à vivre si elle a deux ou trois cellules voisines.
   - Une cellule meurt d'isolement si elle a moins de deux voisines.
   - Une cellule meurt d'étouffement si elle a plus de trois voisines.
   - Une cellule naît dans une case vide si trois cases voisines exactement sont occupées.

4. En Java, utiliser les méthodes suivantes de la classe `String` :
   - `boolean startsWith(String a)` : commence avec le `String a`.
   - `boolean endsWith(String b)` : termine avec le `String b`.
   - `int compareTo(String c)` : comparaison avec le `String c`.
   - `int compareToIgnoreCase(String d)` : comparaison avec le `String d` indépendamment des majuscules ou des minuscules.
   - `String toLowerCase(String e)` : convertit le `String e` en minuscules.
5. En C++, utiliser les méthodes suivantes de la classe `string` :
   - `insert(int pos1, string a)` : insère le `string a` après la position `pos1`.
   - `insert(int pos2, int nombre, char car)` : insère le caractère `car` le `nombre` de fois à la position `pos2`.
   - `insert(int pos3, string b, int pos4, int dim1)` : insère à la position `pos3` les `dim1` caractères du `string b` commençant à la position `pos4`.
   - `erase(int pos5, int dim2)` : efface `dim2` caractères depuis la position `pos5`.
   - `replace(int pos6, int dim3, string c, int pos7)` : remplace depuis la position `pos6` et pour `dim3` caractères par le contenu du `string c` commençant à la position `pos7`.

6

De la méthode dans nos fonctions

Fonctions et méthodes

Durant la conception d'un programme, nous sommes tous amenés à regrouper à l'intérieur d'un fragment de code un certain nombre d'instructions dont nous avons identifié qu'elles se répéteront régulièrement. Suivant le langage, nous les appelons procédures, fonctions ou méthodes. Ces dernières peuvent recevoir ou non des paramètres et produire un résultat sous différentes formes. Dans les deux listes qui suivent, nous allons résumer les différentes possibilités que nous retrouvons généralement dans ce que nous appelons les entrées et les sorties. Nous commencerons par les données initiales reçues ou utilisées par une de ces fonctions :

- arguments passés à la fonction ;
- paramètres stockés quelque part en mémoire, comme des attributs de classe ;
- paramètres disponibles ou accessibles à l'extérieur du programme comme le disque, un serveur de base de données SQL, le clavier ou encore Internet.

Quant au résultat, nous pourrons le retrouver aussi sous différentes formes. S'il s'agit d'un calcul arithmétique élémentaire, on s'imagine qu'un simple retour de fonction peut se révéler suffisant. Cependant, la fonction elle-même peut produire une multitude d'actions et de résultats directement ou indirectement visibles, par exemple :

- un retour de la fonction (une seule valeur ou objet en Java et C++) ;
- un retour par des arguments passés à la fonction ;

- un résultat stocké quelque part en mémoire, comme dans des attributs d'objet ou de classe ;
- une action à l'extérieur comme l'écran, un fichier sur le disque, un serveur SQL, un équipement hardware ou encore Internet.

Nous verrons, au travers de quelques exemples, ces différents cas et la manière de les programmer correctement. Il est essentiel de décrire ces détails dans la documentation de l'API.

API : Application Programming Interface

L'API est le terme généralement utilisé pour définir les interfaces dont le programmeur dispose pour construire ses applications. Par exemple, toutes les classes délivrées avec le JDK (*Java Development Kit*) de Sun Microsystems font partie de l'API de Java, et ceci toujours pour une version déterminée. Lorsque nous parlons d'API, nous entendons par-là la définition exacte des fonctions, des classes et des méthodes. La documentation peut être un simple fichier d'en-tête, un document ou, encore mieux, un fichier HTML ou un fichier d'aide sous Windows.

Il arrive aussi que l'utilisation et la transformation de variables temporaires privées puissent affecter le comportement ultérieur d'autres fonctions. Il faut être très prudent lors de l'allocation des ressources dans un constructeur ou dans une méthode. Nous introduisons inévitablement des dépendances qui devront plus tard être considérées. Si nous avons une méthode pour lire une ligne de texte depuis un fichier, il aura bien fallu appeler une première fonction pour ouvrir ce fichier. Mais que se passerait-il si cette dernière n'avait jamais été appelée ?

Une procédure, qui est un terme plus généralement utilisé en langage Pascal, est appelée fonction en C et méthode en C++ et en Java. En C, elle ne sera pas attachée à une classe ou à un objet. Cependant, il n'y a pas de grandes différences de syntaxe entre une fonction C et une méthode C++. Nous le verrons plus explicitement au travers de quelques exemples pratiques. En Java, toute méthode doit être définie impérativement à l'intérieur du code d'une classe. Un programmeur C++ studieux pourrait très bien définir toutes ces fonctions C au départ comme méthodes de classe C++. Nous commencerons par un exemple de fonction C très simple et l'étendrons avec des méthodes C++ et Java.

splice() de Perl

Nous avons choisi ici la commande `splice()` du langage Perl comme premier exercice. Nous la développerons en C, C++ et Java. Perl5 est la dernière version d'un langage qui fut originellement conçu pour l'écriture de scripts et d'outils sous Linux. Il est devenu plus récemment l'un des langages essentiels pour les scripts **CGI** (*Common Gateway Interface*) des serveurs Web. Nous trouverons dans l'annexe G un certain nombre de références sur Perl. Afin de montrer les effets de la commande `splice()`, nous allons présenter ce morceau de code Perl :

```
#!/usr/local/bin/perl
my @str = ("H", "e", "l", "l", "o", "1", "2", "3", "4", "5");
print "@str\n";
splice(@str, 3, 3);
print "@str\n";
splice(@str, 3, 2);
print "@str\n";
splice(@str, 3, 1);
print "@str\n";
```

et son résultat :

```
H e l l o 1 2 3 4 5
H e l 2 3 4 5
H e l 4 5
H e l 5
```

`splice()` reçoit comme premier argument une chaîne de caractères. La variable `@str` contiendra une chaîne équivalente à un `string` en C++. À l'origine, elle contiendra `"Hello12345"`. `splice()` va effacer un certain nombre de caractères définis par le dernier argument et à partir d'une position déterminée par le second argument. Si l'index est en dehors des limites, la chaîne ne sera pas touchée, et si le nombre de caractères à effacer dépasse le maximum possible, le reste de la chaîne sera tout de même coupé.

splice() comme fonction C

Retour par arguments

Dans le code qui suit, nous avons choisi de passer un pointeur à une chaîne de caractères C classique. Le contenu de la chaîne sera modifié, comme c'est d'ailleurs le cas en Perl, mais aucune réallocation de mémoire ne sera apportée. Ceci aurait été cependant possible puisque, en principe, au moins un caractère va disparaître. Voici donc notre première version :

```
void splice1(char *pstr, int pos, int longueur) {
  int dimension = 0;  //longueur de la chaîne

  while (pstr[dimension] != 0)  dimension++;

  if ((pos < 0) || (longueur <= 0) || (pos >= dimension))  return;

  int i = pos;
  for (i = pos; i < dimension - longueur; i++) {
    pstr[i] = pstr[i + longueur];
  }
  pstr[i] = 0;
}
```

Le premier calcul est de déterminer la longueur de la chaîne, car elle sera utilisée à plus d'une reprise.

L'instruction à l'intérieur `while()` est en fait équivalente à un `strlen()` de la bibliothèque C qui nous calcule la longueur d'une chaîne de caractères. La première instruction `if` détermine tous les cas d'erreur. Ici, aucune indication d'erreur ne sera reportée, et la chaîne sera retournée intacte. Dans la boucle `for()` nous déplaçons un ou plusieurs caractères, ce qui correspond à notre fonction Perl `splice()`. Il faut absolument vérifier que nous ne dépassons pas la limite droite du tableau. Le dernier `str[i]` est suffisant pour marquer la fin de la chaîne dans le tampon `str` puisque le 0 indiquera la fin de la nouvelle chaîne de caractères « splicée » !

Accès à des données répétitives

Si une fonction utilise plus d'une fois une donnée qui doit être recalculée, il est avantageux de la conserver localement dans une variable. Certains programmeurs affirmeront, avec raison d'ailleurs, qu'une variable isolée et inutilisée peut être intéressante pour vérifier le code pendant la phase de test. Il est alors possible de s'arrêter après l'exécution de l'instruction avec un débogueur.

Retour de fonction

Avant de passer à la phase de test, nous allons directement proposer une autre version de `splice()` qui cette fois-ci ne va pas toucher la chaîne d'origine, mais qui va en créer une nouvelle. La fonction va donc nous retourner le résultat, qui sera constitué d'un pointeur sur une chaîne de caractères nouvellement allouée. L'application utilisant cette fonction devra tester le résultat afin d'effacer cette nouvelle ressource ou de choisir l'action nécessaire en cas d'erreur. Voici donc cette nouvelle version :

```
char *splice2(const char *pstr, int pos, int longueur) {
  int dimension = 0;  //longueur de la chaîne

  while (pstr[dimension] != 0)  dimension++;

  if ((pos < 0) || (longueur <= 0) || (pos >= dimension))  return 0;

  int max = dimension - longueur;
  if ((pos + longueur) > dimension)  max = pos;

  int i = 0;
  int j = 0;
  char *pnew_str = new char[max + 1];
  for (; i < max; i++) {
    if (i == pos) j += longueur;
    pnew_str[i] = pstr[j++];
  }
  pnew_str[i] = 0;

  return  pnew_str;
}
```

Outre le retour avec un `char *`, nous constatons ici une forme assez particulière :

```
const char *pstr
```

forme qui n'existe pas en Java ni sur les anciens compilateurs C. Si nous avions utilisé cette construction dans la version `splice1()`, nous aurions obtenu une erreur de compilation pour les deux instructions `str[i] = ...`. En effet, le `const` va être utilisé par le compilateur pour vérifier que la variable n'est pas touchée. Nous reviendrons dans ce même chapitre sur les différentes variantes d'utilisation et du passage des variables en C++.

La logique de la fonction `splice2()` est légèrement différente, car nous devons calculer soigneusement la dimension de la nouvelle chaîne avant son allocation. La variable `max` doit être adaptée dans le cas où la longueur de la partie à effacer serait supérieure à la dimension du tableau. Ce cas-là est aussi vérifié. Le nouveau tableau doit toujours avoir un octet supplémentaire pour stocker le 0 de la terminaison. La variable `j` est essentielle, car elle nous permet de sauter sur la partie que nous voulons couper.

Recommandation pour des programmeurs C potentiels

À l'exception du cas où un programmeur se doit d'écrire des programmes C sur d'anciennes machines, il n'y a pas ou peu d'avantages à vouloir absolument programmer en C ou à vouloir débuter en C avant de passer à C++, ne serait-ce que pour des raisons de simplicité. Un débutant peut très bien écrire de petits programmes en utilisant les nouvelles ressources du langage C++, mais sans nécessairement écrire de nouvelles classes. C'est en fait ce que nous faisons ici !

Comment utiliser nos fonctions C ?

Nous avons présenté deux fonctions C, que nous devons à présent intégrer dans un fichier source que nous définirons commun pour les deux : `csplice.cpp`. Nous pouvons compiler ces deux fonctions dans un fichier `csplice.o`, que nous allons réutiliser très rapidement :

```
g++ -c csplice.cpp
```

Le lecteur pourra retrouver ces fichiers et le `Makefile` sur le CD-Rom.

Cependant, il nous faut d'abord revenir sur un point de détail dans le cas où ces deux fonctions seraient développées préalablement dans le même fichier source. Si nous écrivons le code avec cette forme :

```
#include <iostream.h>

void splice1(char *str, int pos, int longeur) {
  //.............
}

char *splice2(const char *str, int pos, int longeur) {
  //.....................
}
```

```
int main() {
  // utilisation de splice1() et splice2()
}
```

nous n'aurions aucune difficulté. En revanche, si nous inversions le code ainsi :

```
#include <iostream.h>

int main() {
  // utilisation de splice1() et splice2()
}

void splice1(char *str, int pos, int longeur) {
  //.............
}

char *splice2(const char *str, int pos, int longeur) {
  //...................
}
```

le compilateur retournerait les erreurs :

```
"splice1.cpp", line 12: Error: The function splice2 must have a prototype.
```

car ces deux fonctions ne sont pas définies et ne le seront que plus loin. Il faut donc définir un prototype pour ces fonctions, de cette manière :

```
#include <iostream.h>

void  splice1(char *str, int pos, int longeur);
char *splice2(const char *str, int pos, int longeur);

int main() {
  // utilisation de splice1() et splice2()
}

void splice1(char *str, int pos, int longeur) {
  //.............
}

char *splice2(const char *str, int pos, int longeur) {
  //...................
}
```

Il s'agit à présent de tester soigneusement notre code. Ceci est une nécessité absolue, car le code ci-dessus est relativement pointu pour le traitement des codes d'erreurs et de débordement. Pour ce faire, nous allons intégrer ces deux fonctions dans un programme de test, de la manière suivante :

```
// splice1.cpp
#include <iostream>
```

```
using namespace std;

void  splice1(char *pstr, int pos, int longueur);
char *splice2(const char *pstr, int pos, int longueur);

int main() {
  //char *pmon_string = "0123456789";
  char pmon_string[11];
  char *pmon_nouveau;
  strcpy(pmon_string, "0123456789");
  cout << " 0: " <<pmon_string << endl;

  splice1(pmon_string, 3, 2);
  cout << "11: " << pmon_string << endl;
  if ((pmon_nouveau = splice2(pmon_string, 3, 2)) == 0)  return 0;
  cout << "12: " << pmon_nouveau << endl;
  delete pmon_nouveau;

  splice1(pmon_string, 0, 2);
  cout << "21: " << pmon_string << endl;
  if ((pmon_nouveau = splice2(pmon_string, 0, 2)) == 0)  return 0;
  cout << "22: " << pmon_nouveau << endl;
  delete pmon_nouveau;

  splice1(pmon_string, 3, 100);
  cout << "31: " << pmon_string << endl;
  if ((pmon_nouveau = splice2(pmon_string, 3, 100)) == 0)  return 0;
  cout << "32: " << pmon_nouveau << endl;
  delete pmon_nouveau;
}

void splice1(char *pstr, int pos, int longueur) {
  // code ci-dessus
}

char *splice2(const char *pstr, int pos, int longueur) {
  // code ci-dessus
}
```

Le résultat peut sembler incongru à première vue, mais demeure cependant tout à fait correct :

```
0: 0123456789
11: 01256789
12: 012789
21: 256789
22: 6789
31: 256
```

Il faut d'abord revenir sur la première partie du code, qui a donné des sueurs froides à l'auteur. Le code :

```
char pmon_string[11];
strcpy(pmon_string, "0123456789");
```

pourrait en fait être remplacé par :

```
char *pmon_string = "0123456789";
```

Mais le compilateur que nous avons utilisé semble protéger la location mémoire pour cette dernière forme et plante la machine sur la fonction `strcpy()`. Ce n'est pas le cas avec d'autres compilateurs, comme celui de **Borland (C++Builder)**, qui accepte les deux formes. Le `strcpy()` est une fonction C que nous pourrions avantageusement remplacer par des méthodes de la classe **iostream**, que nous verrons plus loin dans cet ouvrage.

Notre chaîne de caractères d'origine étant `0123456789`, le premier résultat `01256789` correspond bien à l'effacement de deux caractères à partir de la position 3. Le second résultat `012789` correspond aussi à l'effacement de deux caractères à partir de la position 3, mais à partir du résultat précédent. Cependant, pour le test 21 (test 2, `splice1()`), nous n'utilisons pas le précédent résultat car `splice2()` a préservé le contenu de la chaîne. Enfin, le dernier test 32 n'apparaîtra pas, car l'index 3 se trouve à présent en dehors du tableau, et une nouvelle chaîne ne sera pas retournée.

Nos fonctions C dans un module séparé

Si nous écrivons le code ci-dessous :

```
// splice2.cpp
#include <iostream>

using namespace std;

extern void  splice1(char *pstr, int pos, int longueur);
extern char *splice2(const char *pstr, int pos, int longueur);

int main() {
  char pmon_string[27];
  char *pmon_nouveau;
  strcpy(pmon_string, "aBcDeFgHiJkLmNoPqRsTuVwXyZ");
  cout << "a: " <<pmon_string << endl;

  splice1(pmon_string, 16, 6);
  cout << "b: " << pmon_string << endl;
  if ((pmon_nouveau = splice2(pmon_string, 0, 10)) == 0)  return 0;
  cout << "c: " << pmon_nouveau << endl;
  delete pmon_nouveau;
}
```

cela nous indique que nos deux fonctions `extern splice1()` et `splice2()` doivent se trouver dans un module séparé. Le `Makefile` va nous aider à mieux comprendre la composition et la création de cette application :

```
splice2.exe:     csplice.o splice2.o
                 g++ -o splice2.exe splice2.o csplice.o

splice2.o:       splice2.cpp
                 g++ -c splice2.cpp

csplice.o:       csplice.cpp
                 g++ -c csplice.cpp
```

L'exécutable `splice2.exe` est composé de deux modules, `splice2.o` et `csplice.o`, qui sont compilés séparément. Il est important d'ajouter les dépendances, car si nous modifions le code de `csplice.cpp`, la recompilation doit s'effectuer correctement. Si nous changions la définition des fonctions, il faudrait aussi modifier les deux lignes d'`extern` dans le code de `splice2.cpp`. Ce n'est donc pas la meilleure manière de faire, et il faudrait transférer ces deux lignes de code dans un fichier d'en-tête commun aux deux fichiers `.cpp`. Nous ferons ce travail comme exercice.

Les arguments de méthodes en C++

Cette introduction aux fonctions du langage C était sans aucun doute un passage obligé. Nous allons à présent réutiliser notre fonction Perl `splice()` et l'intégrer dans une classe C++ que nous allons tout simplement nommer `Perl1`. Nous allons présenter les différentes variantes du passage des paramètres en C++. Nous allons commencer par la manière traditionnelle, le passage par référence, qui n'existe pas en langage C.

Passage par référence (classe C++ Perl1)

Nous allons procéder par étape, afin d'analyser les différentes manières en C++ de traiter le passage d'arguments ainsi que le retour du résultat de la méthode. Il est tout à fait raisonnable de créer une méthode `splice1()` et d'utiliser cette fois-ci la classe `string` du Standard C++. Nous commencerons donc par la définition de la classe `Perl1` dans le fichier d'en-tête `perl1.h` :

```
// Perl1.h
#include <string>
class Perl1 {
  public:
    void splice1(string &str, int pos, int longueur = 1);
};
```

Presque trop simple pour être juste ! Une classe sans constructeur ni attribut ne peut être plus déshabillée ! Ce qui est important ici, c'est l'apparition d'une référence (`&str`) et de l'utilisation d'un paramètre par défaut (`int longueur = 1`). Nous en comprendrons très rapidement l'utilisation, après avoir développé le code de la méthode :

```
// Perl1.cpp
#include "Perl1.h"
```

```
using namespace std;

void Perl1::splice1(string &str, int pos, int longueur) {
    int posend = pos + longueur;
    int dimension = str.length();

    if ((pos < 0) || (longueur <= 0) || (pos >= dimension))  return;

    if ((longueur + pos) > dimension) {
      str = str.substr(pos, dimension);
    }
    else {
      str = str.substr(0, pos) + str.substr(posend, dimension);
    }
}
```

Comme notre méthode `splice1()` ne retourne pas de valeur, nous devons nous imaginer que le résultat sera disponible dans la variable `str`. C'est effectivement le cas, car nous utilisons une référence. L'ancienne chaîne de caractères sera donc perdue comme dans le langage Perl. La méthode `splice1()` modifiera directement le contenu de `str`. Nous allons le vérifier à présent avec le code suivant, qui nous permet de tester la classe :

```
// Perltest1.cpp
#include <iostream>
#include <string>
#include "Perl1.h"

using namespace std;

int main() {
  string mon_string("0123456789abcdef");
  Perl1 mon_perl;

  cout << " 0          : " << mon_string << endl;

  mon_perl.splice1(mon_string, 3, 2);
  cout << " 1   (3, 2) : " << mon_string << endl;

  mon_perl.splice1(mon_string, 0, 3);
  cout << " 2   (0, 3) : " << mon_string << endl;

  mon_perl.splice1(mon_string, 100, 3);
  cout << " 3 (100, 4) : " << mon_string << endl;

  mon_perl.splice1(mon_string, 4, 100);
  cout << " 4 (4, 100) : " << mon_string << endl;

  mon_perl.splice1(mon_string, 2);
  cout << " 5 (2)      : " << mon_string << endl;
}
```

Nous voyons aussi dans ce code l'utilisation de la méthode `splice1()` avec uniquement deux paramètres. Dans le fichier d'en-tête nous avions :

```
public void splice1(string &str, int pos, int longueur = 1);
```

ce qui indique que, si le troisième paramètre est omis, il sera automatiquement initialisé à 1. C'est une forme très pratique, uniquement possible en C++, sur laquelle nous allons revenir. Nous pouvons constater les différents résultats qui sortent de ce programme de test :

```
0             :  0123456789abcdef
1    (3, 2)   :  01256789abcdef
2    (0, 3)   :  56789abcdef
3    (100, 4) :  56789abcdef
4    (4, 100) :  9abcdef
5    (2)      :  9acdef
```

et qui correspondent à la spécification définie préalablement. Un test complet aurait dû inclure plus de cas limites comme des entiers négatifs. Cette version avec un passage par référence est sans doute notre préférée et la plus propre. Nous le comprendrons mieux après avoir examiné les deux autres variantes, par valeur et par pointeur.

Pour le dernier appel, `mon_perl.splice1(mon_string, 2)`, nous noterons que le paramètre `longueur = 1` est automatiquement ajouté. Il est possible d'en définir plusieurs de ce type, mais ils doivent impérativement se trouver en fin de liste. Une construction telle que :

```
void  Perl1::fonction(int va = 1, int vb, int vc = -1);
```

n'est pas possible. Il faudrait déplacer le paramètre `vb` devant `va`.

Paramètres déclarés comme const

Dans l'exemple qui précède, pourrions-nous utiliser la forme suivante ?

```
void Perl1::splice1(const string &str, int pos, int longueur)
```

Eh bien, non ! La directive `const` indique que la variable `str` ne sera pas touchée par la méthode `splice1()`. Pour s'en convaincre, il suffit de modifier les fichiers `Perl1.h` et `Perl1.cpp` et d'essayer de recompiler le code. Les deux lignes suivantes :

```
str = str.substr(pos, dimension);
str = str.substr(0, pos) + str.substr(posend, dimension);
```

nous retourneraient une erreur, puisque le `string str` est modifié.

Passage par valeur (classe C++ Perl2)

Essayons à présent une version sans référence (`&`). Cette forme est appelée passage par valeur. Nous inclurons ici les trois parties du code, c'est-à-dire la définition, le code de la classe et le test dans le même fichier `Perl2.cpp`, ceci uniquement pour des raisons de présentation. Il n'est pas obligatoire de nommer la méthode `splice2()` avec un 2. Nous

avons fait ce choix pour la différencier plus aisément par rapport à la précédente, qui se nomme en fait `Perl1::splice1()`. Nous aurions fort bien pu mettre les deux méthodes dans une seule classe. Voici donc le code avec un passage par valeur :

```
// Perl2.cpp
#include <iostream>
#include <string>

using namespace std;

class Perl2 {
  public:
    void splice2(string str, int pos, int longeur = 1);
};

void Perl2::splice2(string str, int pos, int longueur) {
    int posend = pos + longueur;
    int dimension = str.length();

    if ((pos < 0) || (longueur <= 0) || (pos >= dimension))  return;

    if ((longueur + pos) > dimension) {
      str = str.substr(pos, dimension);
    }
    else {
      str = str.substr(0, pos) + str.substr(posend, dimension);
    }
}

int main() {
  string mon_string("0123456789abcdef");
  Perl2 mon_perl;

  cout << " 0          : " << mon_string << endl;

  mon_perl.splice2(mon_string, 3, 2);
  cout << " 1   (3, 2) : " << mon_string << endl;
}
```

Si nous exécutons ce code, nous remarquons que `str` n'a pas été touché. En effet, dans ce cas-là, la méthode `splice2()` va copier `str` sur la pile et travailler sur celle-ci. Au retour, ce travail sera perdu, comme nous le voyons :

```
 0          : 0123456789abcdef
 1    (3, 2) : 0123456789abcdef
```

Cette solution n'est pas possible, et il nous faut, cette fois-ci, recevoir le `string` comme résultat de la méthode. C'est ce que nous ferons dans la version `Perl2a` ci-dessous :

```
// Perl2a.cpp
#include <iostream>
#include <string>
```

```
using namespace std;

class Perl2 {
  public:
    string splice2(string str, int pos, int longeur = 1);
};

string Perl2::splice2(string str, int pos, int longueur) {
    int posend = pos + longueur;
    int dimension = str.length();

    if ((pos < 0) || (longueur <= 0) || (pos >= dimension))  return str;

    if ((longueur + pos) > dimension) {
      return str.substr(pos, dimension);
    }
    else {
      return str.substr(0, pos) + str.substr(posend, dimension);
    }
}

int main() {
  string mon_string("0123456789abcdef");
  Perl2 mon_perl;

  cout << " 0          : " << mon_string << endl;

  mon_string = mon_perl.splice2(mon_string, 3, 2);
  cout << " 1   (3, 2) : " << mon_string << endl;
}
```

dont le résultat, cette fois-ci, nous satisfait :

```
 0          : 0123456789abcdef
 1    (3, 2) : 01256789abcdef
```

Le problème principal concerne la copie de tous les paramètres passés comme arguments par valeur. Si ces `string` sont importants, nous pouvons nous imaginer le travail fourni, qui peut s'avérer inutile dans cette situation, pour produire une seconde copie en mémoire. Cela affectera les performances, et il faut donc éviter cette construction, en général sans intérêt.

const et passage par valeur

Le morceau de code suivant pourrait sembler stupide en première lecture, mais demeure essentiel pour les programmeurs débutants :

```
// untest.cpp
#include <iostream>
```

```
using namespace std;

void untest(const int var1, int var2) {
  int v1 = var1;
  int v2 = var2;

  v1 = 100;
  v2 = 200;

  // var1 = 10;  impossible
  var2 = 20;
  // … etc.
}
int main() {
  int variable1 = 1;
  int variable2 = 2;

  untest(variable1, variable2);
  cout << "variable2: " << variable2 << endl;
}
```

Et son résultat :

```
variable2: 2
```

De ce code, nous pouvons tirer les conclusions suivantes :

- Le paramètre `var2` étant passé par valeur, il restera inchangé au retour de la fonction `untest()`.
- Il est donc inutile de le copier dans une nouvelle variable `v2` sur la pile, sauf si la méthode `untest()` doit réutiliser la valeur initiale à plusieurs reprises.
- Le `const` est inutile, puisque la variable `var1` est passée par valeur et restera inchangée pour la partie appelante. Elle obligerait inutilement la copie dans une autre variable (`v1`), si nous désirions l'utiliser comme variable de travail.

Passage par pointeur (classe C++ Perl3)

C'est la forme classique C que nous avons déjà traitée, mais sur laquelle nous allons revenir rapidement pour terminer cette partie :

```
// Perl3.cpp
#include <iostream>
#include <string>

using namespace std;

class Perl3 {
  public:
    void splice3(string *str, int pos, int longeur = 1);
};
```

```
void Perl3::splice3(string *str, int pos, int longueur) {
    int posend = pos + longueur;
    int dimension = str->length();

    if ((pos < 0) || (longueur <= 0) || (pos >= dimension))  return;

    if ((longueur + pos) > dimension) {
      *str = str->substr(pos, dimension);
    }
    else {
      *str = str->substr(0, pos) + str->substr(posend, dimension);
    }
}

int main() {
  string mon_string("0123456789abcdef");
  Perl3 mon_perl;

  cout << " 0          : " << mon_string << endl;

  mon_perl.splice3(&mon_string, 3, 2);
  cout << " 1   (3, 2) : " << mon_string << endl;
}
```

Par rapport à la version `Perl2`, il n'y a qu'une seule différence dans le `main()`, notre `&` devant le `mon_string`. En effet, `splice3()` s'attend à recevoir un pointeur. En revanche, dans le code de `splice3()`, c'est un peu plus « sauvage ». Un programmeur C y retrouvera ses habitudes avec ces bons vieux `*` et `->`. Un programmeur Java débutant en C++ va sans doute être horripilé par cette construction. Comme, dans ce cas-ci, il n'y a aucune nécessité d'utiliser cette construction, nous aimerions conseiller, voire contraindre, les programmeurs à utiliser le passage par référence (`Perl1::splice1()`).

Le suffixe const pour une méthode C++

Il est possible d'ajouter le suffixe `const` à la fin de la déclaration d'une méthode. Nous allons étudier sa signification au travers de cet exemple :

```
class UneClasse {
  private:
    int data;

  public:
    void inspecte() const {
      // data = 1;  impossible
    }

    void fait_quelque_chose() /* const impossible */ {
      data = 1;
    }
};
```

Nous promettons que la méthode `inspecte()` ne modifiera sous aucun prétexte l'état de l'objet, c'est-à-dire ici l'attribut `data`. En revanche, la méthode `fait_quelque_chose()` peut changer l'état de l'objet. Si nous remettions le `data = 1;` nous obtiendrions du compilateur :

```
const.cpp: In method `void UneClasse::inspecte() const':
const.cpp:13: assignment of member `UneClasse::data'
              in read-only structure
```

ce qui est justifié et propre. À présent, si nous voulons utiliser des méthodes au travers d'objets de cette classe, nous pourrions utiliser une fonction écrite ainsi :

```
void ma_fonction(UneClasse& uc1, const UneClasse& uc2) {
  uc1.inspecte();
  uc2.inspecte();
  uc1.fait_quelque_chose();
  // uc2.fait_quelque_chose();  impossible
}
```

Nous ne pouvons pas laisser la dernière instruction, car le compilateur nous retournerait :

```
const.cpp: In function `void ma_fonction(UneClasse &, const UneClasse &)':
const.cpp:26: passing `const UneClasse' as `this' argument of `void
➥UneClasse::fait_quelque_chose()' discards qualifiers
```

Lors de la définition de `ma_fonction()`, nous désirons que l'objet `uc2` reste constant, c'est-à-dire inchangé. Un objet déclaré non constant, comme `uc1`, pourra utiliser n'importe quel type de méthode publique de la classe `UneClasse`.

C'est un aspect essentiel contribuant à une bonne spécification des paramètres d'une fonction ou d'une méthode. De très nombreux objets peuvent être passés comme constants à des méthodes. Nous aurions pu écrire simplement, sans penser trop loin :

```
void fait_quelque_chose() const {}
```

Le compilateur nous aurait alors forcé à regarder notre code plus sérieusement. Il peut aussi arriver que le simple rajout d'un suffixe `const` nous permette de compiler notre code qui refusait systématiquement certaines constructions !

Toute cette discussion est aussi applicable à des pointeurs que nous pouvons déclarer `constant` alors que cela n'a aucun sens pour des paramètres passés par valeur.

Comme nous l'avons vu ci-dessus, l'utilisation du mot-clé `const` pour les méthodes et fonctions C++ doit faire l'objet d'une attention particulière. En d'autres termes, il n'est pas recommandé d'ignorer systématiquement la directive `const` simplement par paresse. Un emploi courant de passage de paramètres par référence constante est la marque d'une programmation C++ professionnelle !

Fonctions et méthodes inline en C++

Une fonction peut être déclarée `inline` (en ligne) :

```
inline int une_fonction(const int nombre1, const int nombre2) {
  // … code de la fonction
}
```

Le mot-clé `inline` est destiné au compilateur afin de lui demander de remplacer l'appel de la fonction par du code en ligne. Chaque fois que nous utiliserons `une_function()`, le code sera inséré par le compilateur, ce qui permettra d'économiser un appel de fonction. Cela peut représenter un gain important de performance. Pour une classe, si nous écrivons le code d'une méthode faisant partie de la définition de la classe, comme suit :

```
class UneClasse {
  public:
    int une_methode(const int nombre1, const int nombre2) {
       // … code de la fonction
  }
};
```

la méthode `une_methode()` sera automatiquement `inline`. Ceci veut dire que chaque fois que cette méthode sera utilisée, pour un objet de la classe `UneClasse`, le compilateur insérera le code en ligne. Si nous définissons notre classe ainsi :

```
class UneClasse {
  public:
    int une_methode(const int nombre1, const int nombre2);
};

int UneClasse::une_methode(const int nombre1, const int nombre2) {
     // … code de la fonction
}
```

la méthode restera normale, avec un appel de méthode. Il y a évidemment une perte en performance, mais en contrepartie un gain en dimension de programme puisque le code ne sera pas recopié chaque fois. Nous rappelons que la définition de la classe, et non le code de la classe, est en général incluse dans un fichier d'en-tête.

Toutefois, si nous souhaitons que le compilateur recopie le code à chaque fois, nous devons employer la forme suivante :

```
class UneClasse {
  public:
    int une_methode(const int nombre1, const int nombre2);
};

inline int UneClasse::une_methode(const int nombre1, const int nombre2) {
     // … code de la fonction
}
```

Bruce Eckel, par exemple, dans son ouvrage *Thinking in C++*, consacre un chapitre entier à ce sujet, ainsi qu'aux aspects de performance des compilateurs et à l'écriture de ces derniers. Si la méthode est très souvent utilisée et ne contient que quelques lignes de code, il est tout à fait raisonnable de la définir `inline`.

Utilisation des énumérations avec des méthodes C++

Au chapitre 2, pour le C++ seulement, nous avons déjà étudié les énumérations, au travers du mot-clé `enum`, que nous pouvons utiliser judicieusement dans une classe et dans des appels de fonctions ou de méthodes. Voici deux utilisations possibles des énumérations, comme l'illustre cette petite classe abrégée, qui devrait être définie dans un fichier d'en-tête séparé :

```
// methode_enum.cpp
class EnumTest {
  public:
    enum ENUM_TEST_RES { ENUM_TEST_OK, ENUM_TEST_ERR };

    enum ENUM_OPTION { ENUM_OPTION_DEC, ENUM_OPTION_OCT, ENUM_OPTION_HEX };

    ENUM_TEST_RES calcule(const char *pinput,
                          const char *poutput, ENUM_OPTION enum_opt) {
      // code de la méthode
      return ENUM_TEST_OK;
    }
};
int main() {
  EnumTest et;
  char *pentree = "123456";
  char *psortie;

  EnumTest::ENUM_TEST_RES res = et.calcule(pentree, psortie,
  ➥EnumTest::ENUM_OPTION_DEC);

  switch(res) {
    case EnumTest::ENUM_TEST_OK:
      //....
      break;
    case EnumTest::ENUM_TEST_ERR:
      //....
      break;
    default:
      //... erreur
      break;
  }
  return 0;
}
```

La déclaration de l'énumération doit être publique si nous voulons accéder à des définitions comme `EnumTest::ENUM_TEST_OK`. Il est tout à fait possible d'utiliser des énumérations privées, pour ne les utiliser qu'à l'intérieur de la classe. Définir des énumérations globales, c'est-à-dire en dehors d'une classe et dans un fichier d'en-tête, est également envisageable, mais à déconseiller, au risque de nous trouver avec une multitude de noms. On retrouve parfois, dans la littérature, le terme de pollution dans ce contexte.

Dans cet exemple, nous utilisons une énumération à la fois pour passer un paramètre et pour retourner un résultat. La méthode `calcule()` prendra l'action nécessaire, suivant que nous travaillons en décimal, en octal ou en hexadécimal. On ne sait pas pourquoi ici, mais on s'imagine que le résultat viendra dans une chaîne de caractères allouée dynamiquement sur le pointeur `psortie`. Retourner un résultat avec un `enum` est très souvent utilisé en lieu et place d'un entier, qui contient parfois n'importe quoi.

Utilisation des énumérations en Java

Les énumérations en Java sont apparues à partir du JDK 1.5. Sans trop rentrer dans les détails, nous donnerons quelques exemples et commencerons par un cas simple, une liste de fruits :

```
public class EnumFruits1 {
  public enum Fruits { poire, pomme, fraise, mirabelle, prune };

  public static void main(String[] args) {
    for (Fruits unFruit : Fruits.values()) {
      System.out.println(unFruit);
    }

    System.out.println();
    System.out.println(Fruits.pomme);
  }
}
```

Maintenant, si nous exécutons ce code :

```
poire
pomme
fraise
mirabelle
prune

pomme
```

Nous voyons que les différentes valeurs de l'énumération sont prises comme des `Strings`. La boucle `for` utilise aussi la forme étendue du JDK 1.5 que nous avons vu au chapitre 3.

Nous pouvons étendre notre exemple de cette manière :

```
public class EnumFruits2 {
  public enum Fruits {
    poire, pomme, fraise;

    public String toString() {
      switch (this) {
         case poire:
            return ("Poire");
          case pomme:
```

```
            return ("Pomme");
          case fraise:
            return ("Fraise");
      }
      return "Fruit inconnu";
    }
  }

  public static void main(String[] args) {
    for (Fruits unFruit : Fruits.values()) {
      System.out.println(unFruit);
    }

    if (unFruit == Fruits.pomme) {
      System.out.println("--> Nous avons vraiment une pomme");
    }
  }
}
```

Le résultat sera présenté ainsi avec nos fruits en majuscules :

```
Poire
Pomme
--> Nous avons vraiment une pomme
Fraise
```

En fait c'est équivalent à une classe interne. Pour chaque fruit, la méthode `toString()` sera appelée afin de nous présenter nos fruits dans la forme désirée : ici avec une majuscule devant.

Les arguments de méthodes en Java

Le langage Java, au contraire de C++, n'a pas trois méthodes de passage d'arguments (par valeur, par pointeur ou par référence), mais qu'une seule, que nous appellerons par valeur bien qu'il faille préciser les détails plus loin. Nous pouvons aussi choisir, comme pour C++, de retourner le résultat soit avec un argument soit avec un retour de fonctions. Nous allons présenter ici deux variantes :

- `Perl1` : similaire à l'implémentation du langage **Perl**, où la chaîne de caractères n'est pas préservée.
- `Perl2` : retour de la méthode avec un nouveau `String`, sans toucher à l'original.

splice() avec retour par l'argument (classe Java Perl1)

```
class Perl1 {
  public void splice(StringBuffer str, int pos, int longueur) {
    int posfin = pos + longueur;
    int dimension = str.length();
```

```
    if ((pos < 0) || (longueur <= 0) || (pos >= dimension))  return;

    if ((longueur + pos) > dimension)  posfin = dimension;

    str.delete(pos, posfin);
  }

  public static void main(String[] args) {
    Perl1 myperl = new Perl1();
    StringBuffer str = new StringBuffer("0123456789");
    System.out.println(str);

    if (args.length > 0) {
      int pos  = Integer.parseInt(args[0]);
      int larg = Integer.parseInt(args[1]);
      myperl.splice(str, pos, larg);
      System.out.println(str);
    }
    else {
      myperl.splice(str, 3, 2);
      System.out.println(str);
      myperl.splice(str, 0, 2);
      System.out.println(str);
      myperl.splice(str, 3, 100);
      System.out.println(str);
    }
  }
}
```

Durant la phase de développement du programme, il est essentiel de tester toutes les variations possibles. Un bon programmeur n'est pas nécessairement un bon testeur ! Certains cas limites sont souvent oubliés. Nos trois petits tests sont bons, mais rien de plus ! Le résultat :

```
0123456789
01256789
256789
256
```

est devenu correct alors qu'à l'origine nous avions utilisé une `dimension - 1`. L'API du `delete()` nous indique bien : à l'exclusion de `posfin` ! En effet, si nous avions écrit :

```
if ((longueur + pos) > dimension)  posfin = dimension - 1;
```

au lieu de :

```
if ((longueur + pos) > dimension)  posfin = dimension;
```

nous aurions alors reçu le résultat suivant :

```
0123456789
01256789
256789
2569
```

qui nous semble correct au premier coup d'œil ! L'introduction de différents choix par l'utilisateur augmente les possibilités de combinaisons des tests, souvent sans le vouloir, alors que des algorithmes systématiques de test n'aboutiraient pas forcément à toutes ces possibilités. L'entrée suivante, avec le code `dimension - 1`, nous montrerait immédiatement l'erreur :

```
java Perl1 0 1000
0123456789
9
```

Il nous faut à présent passer aux détails du code. Nous commencerons par l'utilisation de `StringBuffer`, qui est mutable (modifiable), au contraire de `String`. Il n'y a pas de méthode telle que `delete` dans la classe `String`. Nous pourrions nous poser la question de savoir pourquoi nous n'utilisons pas simplement :

```
str.delete(pos, posfin);
```

sans vérifier la validité des arguments. La réponse est simple ! Une petite erreur d'index entraînerait une exception et un résultat inattendu. Dans notre exemple, nous avons accepté le consensus de **Perl**, c'est-à-dire ne pas toucher à la chaîne si la position ou la longueur du morceau à couper est incorrecte.

splice() avec retour de méthode (classe Java Perl2)

Nous passons à présent à la version qui nous retourne un nouveau `String` laissant inchangée la chaîne d'origine.

```
class Perl2 {
  public String splice(String str, int pos, int longueur) {
    int posend = pos + longueur;
    int dimension = str.length();

    if ((pos < 0) || (longueur <= 0) || (pos >= dimension)) return str;

    if ((longueur + pos) > dimension) return str.substring(pos, dimension);

    return str.substring(0, pos) + str.substring(posend, dimension);
  }

  public static void main(String[] args) {
    Perl2 myperl = new Perl2();
    String str = new String("0123456789");
    System.out.println(str);

    if (args.length > 0) {
      int pos  = Integer.parseInt(args[0]);
      int larg = Integer.parseInt(args[1]);
      System.out.println(myperl.splice(str, pos, larg));
    }
    else {
```

```
        System.out.println(myperl.splice(str, 3, 2));
        System.out.println(myperl.splice(str, 0, 2));
        System.out.println(myperl.splice(str, 3, 100));
      }
    }
}
```

Le résultat sera évidemment différent de celui de la classe `Perl1` puisque la chaîne de caractères initiale est toujours utilisée :

```
0123456789
01256789
23456789
3456789
```

Dans ce genre de code, il est essentiel de vérifier l'API de la méthode Java `substring()`. Nous ne pouvons pas la comparer à la méthode `substr()` du C++, dont le deuxième paramètre est une dimension et non une position comme en Java. Il est donc judicieux de nommer correctement le nom des variables, comme ici avec `pos`, `pos2` ou `posfin`. Le programmeur pourrait se faire un petit schéma afin d'analyser son code.

Java : argument par référence ou par valeur ?

C'est une question qui revient souvent, principalement pour les programmeurs qui ont une culture C++. Nous allons entrer dans les détails après avoir examiné ce morceau de code :

```
class RefOuVal {
  public void monTest(StringBuffer monstr, int monnb) {
    monnb = 9999;
    monstr.append("_un_appendice_" + monnb);
    System.out.println("Pendant: " + monstr + " " + monnb);
    monstr = new StringBuffer("Vraiment_0123456789");
    System.out.println("Pendant: " + monstr + " " + monnb);
  }

  public static void main(String[] args) {
    RefOuVal unobjet = new RefOuVal();

    StringBuffer unstr = new StringBuffer("0123456789");
    int leNombre = 10;

    System.out.println("Avant: " + unstr + " " + leNombre);
    unobjet.monTest(unstr, leNombre);
    System.out.println("Après: " + unstr + " " + leNombre);
  }
}
```

et bien évidemment le résultat :

```
Avant: 0123456789 10
Pendant: 0123456789_un_appendice_9999 9999
```

```
Pendant: Vraiment_0123456789 9999
Après: 0123456789_un_appendice_9999 10
```

Commençons par le plus simple ! La variable `leNombre` de type primitif `int` est passée par valeur à la méthode `monTest()`. À l'origine, elle possède la valeur de 10 et ne sera pas affectée par les modifications dans la méthode (affectation à 9999). Ainsi, une méthode Java peut jouer avec n'importe quel type d'argument de type primitif.

En ce qui concerne la variable `unstr`, qui est un objet de la classe `StringBuffer`, elle est aussi passée par valeur, mais pour la référence de l'objet. Comme `StringBuffer` est mutable, le contenu sera bien modifié comme le montre le résultat. Cependant, le dernier :

```
monstr = new StringBuffer("Vraiment_0123456789");
```

ne va pas toucher le contenu original. `monstr` est en fait une copie de l'adresse de la référence. À la fin de la méthode, ce nouvel objet sera envoyé dans le ramasse-miettes (*garbage collector*). Cette dernière construction serait d'ailleurs contraire à une programmation orientée objet.

Enfin, cette forme est aussi possible :

```
public void monTest(final StringBuffer monstr, int monnb) {
```

mais entraînerait une erreur de compilation :

```
RefOuVal.java:6: Can't assign a second value to a blank final variable: monstr
    monstr = new StringBuffer("Vraiment_0123456789");
    ^
1 error
```

Le seul avantage de ce code serait de forcer le programmeur à utiliser d'autres constructions et de ne pas créer un objet dont il pourrait penser qu'il sera réutilisé après avoir été modifié.

Pour résumer, nous dirons que :

- Les arguments de type primitif sont passés par valeur.
- Les autres types d'argument sont passés par valeur pour la référence de l'objet, mais leur contenu peut être modifié si ce dernier est mutable.

Les espaces de noms en C++

Lorsque nous avons rencontré pour la première fois l'espace de noms :

```
using namespace std;
```

c'était bien évidemment au chapitre 1, où nous avons montré clairement notre intention de ne pas écrire du C ou du C++ traditionnel. Le `std` indique ici que les noms que nous allons utiliser se trouvent dans l'espace de noms de la bibliothèque Standard C++.

Le Standard C++ a introduit un mécanisme permettant de prévenir la collision des noms utilisés au travers du système. Un programmeur C a sans doute rencontré ce problème en

utilisant par erreur des noms déjà définis globalement ou dans des fichiers d'en-tête. En passant en C++, en éliminant par exemple les variables globales, la situation s'est améliorée, mais pas totalement, et ce notamment dans le cas de gros projets informatiques. L'ancien programmeur C aura remplacé ses variables globales par des attributs de classe et ses fonctions par des méthodes de classe. Il accédait autrefois à des variables globales, alors que maintenant, en bon programmeur C++, il n'accédera même plus à un attribut public d'une classe, mais l'aura protégé pour forcer l'accès au travers d'une ou de plusieurs méthodes de cette classe. Dans son dernier ouvrage de référence, Bjarne Stroustrup consacre un chapitre entier sur les espaces de noms et les diverses techniques d'accès de plusieurs espaces de noms. Dans le présent ouvrage, nous ne donnerons qu'un aperçu rapide, dans le but d'en comprendre le mécanisme et l'accès.

Utilisation classique du namespace

Prenons pour premier exemple un programme traditionnel :

```
// namespace1.cpp
#include <cstdio>

using namespace std;

void main() {
  printf("J'efface le fichier test.txt\n");
  remove("test.txt");
}
```

Les deux fonctions `printf()` et `remove()` sont de bonnes vieilles fonctions C qui sont en fait définies dans le fichier d'en-tête `stdio.h`, bien connu des programmeurs C. Nous savons déjà que `printf()` peut être avantageusement remplacé par un `cout <<`. La fonction `remove()` est une de ces fameuses fonctions C qui n'existent pas dans `iostream`. Comme elle entraîne souvent des problèmes de compatibilité, elle est parfois remplacée par `unlink()`, qui permet d'effacer un fichier. Le `<cstdio>` est l'illustration de comment procéder à présent. Cette forme nous permet d'accéder indirectement aux fonctions des anciennes bibliothèques C, qui sont toujours accessibles et dont la plupart sont certifiées ANSI ou ISO. Si nous avions écrit nos deux instructions du `main()` de cette manière :

```
std::printf("J'efface le fichier test.txt\n");
std::remove("test.txt");
```

notre code aurait fonctionné pareillement. La notation `::` permet d'accéder à un membre dans un espace de noms déterminé. Ici, l'espace de noms `std`, utilisé tout au long de notre ouvrage, est spécifié formellement, bien que cela ne soit pas nécessaire, car nous l'avons déjà défini avec `using namespace std;`, notre espace de noms par défaut.

Conflit de nom

À présent, nous aimerions définir une fonction nommée `remove()` ainsi :

```
// namespace2.cpp
#include <cstdio>

using namespace std;

void remove(const char *pfile) {
  printf("J'efface ici le fichier test.txt\n");
  remove("test.txt");
}

void main() {
  printf("J'efface le fichier test.txt\n");
  remove("test.txt");
}
```

Si nous essayons de compiler ce code, nous voyons bien la difficulté :

```
g++ -c namespace2.cpp
namespace2.cpp: In function `void remove(const char *)':
namespace2.cpp:7: new declaration `void remove(const char *)'
/usr/include/stdio.h:226: ambiguates old declaration
                        `int remove(const char *)'
```

Nous avons donc bien un conflit de nom entre la méthode `remove()` disponible dans `<cstdio>` et celle que nous aimerions définir. Il nous faut soit changer de nom soit utiliser un autre mécanisme.

Comment définir un espace de noms

Pour corriger le code précédent, nous sommes obligés de définir notre nouvelle fonction dans un espace de noms séparé. Cela se fera de cette façon :

```
// namespace3.cpp
#include <cstdio>
#include <iostream>

using namespace std;

namespace MaLibrairie {
  void remove(const char *pfile) {
    printf("J'efface ici le fichier test.txt\n");
    std::remove("test.txt");
  }

  class UneClasse {
    private:
      int valeur;

    public:
      static int attribut;
```

```
      UneClasse(int une_valeur) : valeur(une_valeur) {}

      void printf() {
        std::printf("Bonjour d'UneClasse: ");
        cout << attribut << " : " << valeur << endl;
      }
  };
}
int MaLibrairie::UneClasse::attribut = 0;

int main() {
  printf("J'efface le fichier test.txt\n");
  MaLibrairie::remove("test.txt");

  MaLibrairie::UneClasse::attribut = 10;
  MaLibrairie::UneClasse maclasse(20);
  maclasse.printf();
}
```

Dans `main()`, nous notons la manière d'accéder à `remove()` dans l'espace de noms `MaLibrairie` avec l'opérateur de portée `::`. Dans notre code `remove()` la forme :

```
std::remove("test.txt");
```

est essentielle. Sans le `std::`, la fonction `remove()` de l'espace de noms `MaLibrairie` serait appelée, et ceci dans une boucle récursive infinie. Nous avons ajouté notre classe `UneClasse` dans cet espace de noms. L'instruction :

```
int MaLibrairie::UneClasse::attribut = 0;
```

est fondamentale, sinon la variable statique ne serait pas initialisée. Nous en verrons d'autres exemples au chapitre 10. Ce qui est important ici, c'est de comprendre la syntaxe et le mécanisme. Un exercice supplémentaire devrait suffire pour s'y familiariser. Nous présentons tout de même le résultat du programme ci-dessus :

```
J'efface le fichier test.txt
J'efface ici le fichier test.txt
Bonjour d'UneClasse: 10 : 20
```

qui est attendu. Bien évidemment, si le fichier `test.txt` existe, il sera effacé. Dans le cas contraire, rien ne se passera, car l'existence ou le résultat de l'opération n'est pas vérifié. Pour s'en convaincre, nous pouvons créer un fichier `test.txt`, exécuter `namespace3.exe` et vérifier que `test.txt` a disparu.

Fichiers d'en-tête et namespace

Nous avons vu au chapitre 4 la forme que devait prendre le fichier d'en-tête `Personne.h` :

```
// Personne.h : définition de la classe Personne
#include <string>
```

```
class Personne {
  private:
    std::string nom;     // nom de la personne
    std::string prenom;  // prénom de la personne
    int         annee;   // année de naissance de la personne
```

et plus loin dans le code associé de `Personne.cpp` :

```
// Personne.cpp
#include "Personne.h"

#include <iostream>
#include <sstream>

using namespace std;

Personne::Personne(string leNom, string lePrenom, string lAnnee) {
   ……
```

C'est la règle conventionnelle du Standard C++ que nous adopterons : ne jamais utiliser de directive `namespace` dans un fichier d'en-tête `.h`.

C'est au programmeur, lors de la saisie de son code (c'est-à-dire des fichiers `.cpp`) de définir son espace de noms. Pour plus d'informations, consultons par exemple le site suivant : `http://www.gotw.ca/publications/migrating_to_namespaces.htm`

Fichiers d'en-tête multiples en C++

Si nous consultons le fichier d'en-tête `cstdio` de la distribution C++ de GNU, qui se trouve dans le répertoire `include\g++`, nous pourrions y découvrir ceci :

```
// The -*- C++ -*- standard I/O header.
// This file is part of the GNU ANSI C++ Library.
#ifndef __CSTDIO__
#define __CSTDIO__
#include <stdio.h>
#endif
```

Le fait que nous retrouvions le fichier `<stdio.h>` n'est pas vraiment surprenant, puisque nous aimerions réutiliser ces bonnes vieilles fonctions C. Ce qui est vraiment nouveau ici, ce sont ces différentes directives `#ifndef`, `#define` et `#endif`.

Le `#ifndef` indique que si la définition `__CSTDIO__` existe déjà tout le code entre `#ifndef` et `#endif` sera ignoré. Si notre code possédait les trois directives suivantes :

```
#include <cstdio>
#include <cstdio>
#include <cstdio>
```

cela signifierait que `<stdio.h>` ne serait inclus qu'une seule fois. Ceci est important, car des redéfinitions de nom pourraient créer des complications. Certains fichiers d'en-tête

créent des définitions qui seront réutilisées. L'exemple ci-dessus peut paraître stupide, mais si nous avions ceci :

```
#include <headera>
#include <headerb>
#include <headerc>
```

il pourrait se révéler que les trois `<header.>` contiennent eux-mêmes, chacun, le fichier d'en-tête `<cstdio>` ! Nous n'aurons jamais de telles combinaisons dans cet ouvrage, qui ne contient que des exemples et exercices simples.

Si nous avions un fichier de définition de classe nommé `MaClasse1A.h`, nous aimerions conseiller d'inclure systématiquement ces trois directives :

```
#ifndef __MACLASSE1A__
#define __MACLASSE1A__
// ..... le fichier d'en-tête
#endif
```

Nous mettons tout en majuscule et ajoutons un __ devant et derrière. À la première apparition du fichier d'en-tête `MaClasse1A.h`, `__MACLASSE1A__` sera défini, et le code ne sera plus ajouté à nouveau en cas d'inclusions multiples. Nous reviendrons sur la compilation conditionnelle au chapitre 9.

Résumé

Dans ce chapitre, nous avons exposé les différentes manières de passer des arguments à des méthodes de classe. Les nombreux exemples donnés nous ont sans doute permis de nous y retrouver parmi ces nuances que nous appelons passage par valeur, par référence ou encore, en C++, par pointeur. Nous avons aussi rapidement passé en revue le mécanisme du namespace. Le lecteur devrait maintenant posséder un meilleur bagage pour développer et étendre ces futures classes.

Exercices

1. De la même manière que notre module `csplice.cpp` et l'exemple `splice2.cpp`, écrire une fonction `mamult()` qui nous multiplie deux `int` et nous retourne le résultat en `double` à la fois comme retour et comme arguments de fonction (pour les deux cas : par pointeur et par référence). Vérifier le résultat et écrire cette fois-ci un fichier d'en-tête et un module séparé `mamult.o`. Ne pas oublier ce dernier dans le `Makefile`.
2. En Java, montrer qu'en passant comme argument un tableau d'entiers (type primitif `int[ ]`) ne contenant qu'un seul élément il est possible de modifier sa valeur (classe `ArgChaine`).
3. Écrire, en C++, une classe `MonString` qui ne possède que trois méthodes publiques qui vont exécuter, toutes les trois, le même travail. Deux `string` C++ seront passés comme paramètres, une fois par référence, une fois par valeur, et enfin par pointeur.

Le deuxième `string` sera ajouté au premier, à une position déterminée et pour une certaine longueur. Cette longueur est toujours connue. Si la position n'est pas définie, elle se fera à la fin et sera donc un paramètre par défaut. En cas d'erreur quelconque, le premier `string` est retourné inchangé. Comme il y a plusieurs implémentations possibles, expliquer ses choix. Exemple : strings `"abcdef"` et `"12345"` avec la position 3 et la longueur 4, résultat : `"abc1234def"`.

4. Modifier le programme `namespace3.cpp` en définissant un nouvel espace de noms qui contiendra la fonction `printf()`. Cette dernière fonction sera appelée en lieu et place du `printf()` actuel, afin d'inclure le message entre crochets ([...]), message qui sera imprimé avec le `printf()` inclus dans `<cstdio>`. On se rendra compte que le mixage de `cout` et de `printf()` n'est pas vraiment bienvenu !

7

Notre code dans des bibliothèques

L'une des grandes différences entre les langages Java et C++ concerne la manière dont les programmes sont construits et compilés. Lorsque nous examinons un programme Java, nous découvrons par exemple :

```
import java.util.Date;
```

tandis qu'en C++ nous avons :

```
#include <iostream>
```

En Java, ceci implique que des ressources, c'est-à-dire des classes, sont à disposition grâce à la directive `import`. Le compilateur pourra alors contrôler que le programme à compiler utilise correctement ces ressources.

En C++, c'est presque équivalent, bien que la compilation se fasse en deux phases :

```
g++ -c hello.cpp
g++ -o hello.exe hello.o
```

Dans la première phase, seul le fichier d'en-tête, ici `iostream`, est utilisé pour le contrôle de la syntaxe et la compilation. Ensuite, lors de la seconde étape, le compilateur va lier le fichier binaire `hello.o` avec les bibliothèques du système, dont le compilateur sait, dans le cas présent, où trouver toutes les ressources.

Lors du développement de grands projets informatiques, il se révèle souvent essentiel de partager les tâches, afin de construire des modules séparés qui seront inclus dans des bibliothèques. Ces modules pourront aussi être vérifiés et maintenus séparément, mais

également réutilisés et distribués pour des produits différents. Le code sera donc organisé en composants disponibles dans une ou plusieurs bibliothèques.

C'est ce que nous allons voir dans ce chapitre en présentant un certain nombre de mécanismes que nous avons en fait déjà utilisés implicitement.

Les extensions .jar, .a et .dll

Jusqu'à présent, dans cet ouvrage, nous avons rencontré de nombreux fichiers comportant les extensions suivantes :

- `.java` pour les fichiers source Java.
- `.class` pour les classes Java, qui sont des compilations avec `javac` de fichiers `.java` et qui sont directement exécutables par la machine virtuelle `java`, si elle possède un point d'entrée statique `main()`.
- `.h` pour les fichiers d'en-tête C ou C++ qui contiennent des définitions de fonctions C ou de classes C++. Nous inclurons aussi les fichiers `#include` sans le `.h`, par exemple `<iostream>` qui sont en fait des fichiers dont l'extension `.h` a été masquée.
- `.cpp` pour du code C++, c'est-à-dire le code des programmes ou des classes.
- `.o` pour les fichiers objets C et C++ compilés par la première phase du compilateur C++ (par exemple le g++) avant qu'ils ne soient liés avec le code des bibliothèques pour former des exécutables `.exe`.
- `.exe` pour des fichiers binaires exécutables par le système d'exploitation. Il faut rappeler que l'extension `.exe` n'est pas nécessaire sous Linux.

> **Note**
>
> Nous aurions pu également mentionner les extensions `.c`, utilisées pour des fichiers source en langage C, ou encore les extensions `.C` et `.H`, utilisées plus souvent sous Linux en C++ pour les différencier des `.h` et `.c`.

Nous allons à présent rencontrer d'autres catégories de fichiers qui sont, par exemple :

- `.jar` pour des fichiers d'archives Java qui sont composés d'un ensemble de fichiers `.class` compressés.
- `.a` pour des fichiers d'archives C ou C++ qui pourront être inclus dans l'exécutable `.exe` durant la deuxième étape de la compilation C ou C++.
- `.dll` pour des fichiers de bibliothèques dynamiques que nous rencontrons sous Windows. Ils sont chargés durant l'exécution du programme et font partie de la programmation Windows que nous aborderons au chapitre 21, consacré à JNI (*Java Native Interface*). Nous ne parlerons pas des bibliothèques dynamiques sous Linux, qui n'ont pas l'extension `.dll`.

Les packages en Java

La création de paquets Java présentée ici n'est pas évidente. Pour des projets informatiques d'une certaine dimension, il est évident que des outils comme Crimson ou le `make` ne sont pas appropriés. NetBeans, dans l'annexe E, le fait très bien et automatiquement. À la fin de cette section, nous vous proposerons un récapitulatif des fichiers `.bat` dans le répertoire des exemples, afin de comprendre les différentes étapes de la création de paquets et des tests associés.

À la fin du chapitre 4, nous avons examiné un certain nombre de fonctionnalités nous permettant d'étendre notre classe `Personne`. Afin de comprendre le mécanisme de création des packages, nous avons pris l'exemple de trois classes :

- La classe `Personne` contenant les informations générales sur le salarié.
- La classe `Salaire` avec son montant et la monnaie dans laquelle elle sera versée.
- La classe `ConversionMonnaie`, qui calcule le taux de conversion. Cette dernière pourrait se faire, par exemple, par le biais d'Internet ou d'une base de données.

Nous allons considérer que les classes `Personne` et `Salaire` sont publiques, c'est-à-dire qu'elles pourraient être utilisées par n'importe quelle application future. La classe `ConversionMonnaie` étant tout à fait particulière, elle ne sera pas ouverte au public ! Cette dernière ne sera donc utilisée que par les classes `Personne` et `Salaire`. C'est ici que nous introduisons le concept d'accès package.

Une grande partie du code de la classe `Personne` ne nécessite aucun éclaircissement :

```
package monpaquet;

public class Personne {
  private String  nom;
  private String  prenom;
  private int     annee;
  private Salaire salaire;

  public Personne(String leNom, String lePrenom, String lAnnee) {
    nom     = leNom;
    prenom  = lePrenom;
    annee   = Integer.parseInt(lAnnee);
    salaire = new Salaire(0, "Dollar");
  }

  public void setSalaire(int unSalaire, String uneMonnaie) {
    salaire = new Salaire(unSalaire, uneMonnaie);
  }

  public void un_test() {
    System.out.println("Nom et prenom: " + nom + " " + prenom);
    System.out.println("Annee de naissance: " + annee);
    System.out.println("Salaire annuel: " + salaire.salaireAnnuel()
                                          + " " + salaire.getMonnaie());
  }
```

```
  public static void main(String[] args) {
    Personne unePersonne = new Personne("Dupont", "Justin", "1960");
    unePersonne.setSalaire(5000, "Euro");
    unePersonne.un_test();

    ConversionMonnaie uneConversion = new ConversionMonnaie("Euro");
    System.out.println("Taux: " + uneConversion.tauxConversion());
  }
}
```

si ce n'est la première instruction avec le mot-clé `package`, qui doit absolument se trouver au début du code :

```
package monpaquet;
```

Cette instruction indique que la classe `Personne` fait partie du `package` nommé `monpaquet`. La classe `Personne` utilise aussi une nouvelle classe nommée `Salaire` que nous allons aussi intégrer dans le même paquet :

```
package monpaquet;

public class Salaire {
  private int     montant;
  private String  monnaie;
  private double  bonus;

  public Salaire(int leMontant, String laMonnaie) {
    montant = leMontant;
    monnaie = laMonnaie;
    bonus   = 1.10;           //10 % de bonus annuel
  }

  public int salaireAnnuel() {
    ConversionMonnaie conversion = new ConversionMonnaie(monnaie);
    double tauxDeChange = conversion.tauxConversion();
    return (int)(12 * tauxDeChange * montant * bonus);
  }

  public String getMonnaie() {
    return monnaie;
  }
}
```

Afin de tester nos classes `Personne` et `Salaire`, nous avons introduit une méthode `salaire Annuel()` dans cette dernière classe `Salaire`. Comme le salaire défini en dollars sera payé dans la monnaie du pays, il nous faudra le calculer en fonction du taux de change. C'est pour cette raison que nous avons introduit en troisième classe notre classe `Conversion Monnaie` avec sa méthode `tauxConversion()` :

```
package monpaquet;

class ConversionMonnaie {
   private String monnaie;

   public ConversionMonnaie(String laMonnaie) {
     monnaie = laMonnaie;
   }

   public double tauxConversion() {
     //Accès sur Internet ou une base de données pour obtenir le taux
     // à partir du nom de la monnaie. Sera 1 pour le dollar.

     return 1.2;  //valeur de test
   }
}
```

Elle sera aussi dans notre `package monpaquet`, mais sans un accès `public`. En fait, seule la classe `Salaire` va l'utiliser. L'introduction du bonus dans la classe `Salaire` n'est ici que pour compliquer un peu le jeu, et non pas pour envenimer le climat salarial par des revendications syndicales.

Compiler les classes de notre package

Il s'agit à présent de compiler nos classes, `ConversionMonnaie`, `Salaire.java` et `Personne.java`. Si nous les compilions d'une manière traditionnelle, nous obtiendrions ce type d'erreurs :

```
Salaire.java:15: Class monpaquet.ConversionMonnaie not found.
ConversionMonnaie conversion = new ConversionMonnaie(monnaie);
Personne.java:7: Class monpaquet.Salaire not found.
 private Salaire salaire;
```

En fait, deux points essentiels doivent être réglés :

- Ces trois classes compilées, déposées quelque part, doivent faire partie d'un répertoire nommé `monpaquet`.
- Le chemin d'accès (CLASSPATH) doit être défini pour le compilateur pour identifier où sont les différentes classes utilisées.

Pour ce faire, et pour nous simplifier la tâche, nous avons intégré ces paramètres dans un `Makefile` nommé `MakefileMonPaquet` :

```
MCLASSPATH = C:/JavaCpp/EXEMPLES/Chap07/devPaquet
MDIR = C:/JavaCpp/EXEMPLES/Chap07/devPaquet

all:    java

java:   ConversionMonnaie.class Salaire.class Personne.class


ConversionMonnaie.class:  ConversionMonnaie.java
```

```
                        javac -d $(MDIR) -classpath $(MCLASSPATH)
                        ➥ConversionMonnaie.java

Salaire.class:   Salaire.java
                 javac -d $(MDIR) -classpath $(MCLASSPATH) Salaire.java

Personne.class:  Personne.java
                 javac -d $(MDIR) -classpath $(MCLASSPATH) Personne.java
```

`MCLASSPATH` est ici une variable de notre fichier `make` qui est utilisée pour le `classpath` (voir ci-dessous). Le `MDIR` indique le sous-répertoire où seront stockés nos trois fichiers Java compilés (`.class`) :

```
C:\JavaCpp\EXEMPLES\Chap07\devPaquet\monpaquet
```

Il faut vraiment se méfier des `/` ou des `\`. Il faudra parfois s'y reprendre à plusieurs fois car, suivant les outils utilisés (`make` ou `javac`), ils ne fonctionneront pas correctement. Le premier réflexe, en cas d'erreur de compilation, est de les modifier et d'essayer à nouveau.

Pour compiler ce `Makefile`, nous pourrons le faire avec Crimson ou directement :

```
make -f MakefileMonPaquet
```

Ce `Makefile` compilera aussi bien `Personne.java` que `Salaire.java` ou encore `ConversionMonnaie.java`, car ces classes sont interdépendantes, et nous savons déjà que `javac` est assez intelligent pour compiler les trois fichiers.

Par exemple, pour le fichier `Salaire.java`, le fichier `Salaire.class` sera déposé dans le sous-répertoire `devPaquet/monpaquet` : la première partie venant du `-d` et la seconde du `package monpaquet`. `Salaire.java` aura besoin de la classe `ConversionMonnaie`, déjà compilée et accessible grâce au chemin d'accès du `classpath` (voir ci-dessous).

Le `Makefile` ci-dessus est plus dans la logique C++ de cet ouvrage, car les compilateurs C++ n'ont pas cette facilité. Il faut noter l'utilisation des `/` dans `C:/JavaCpp/EXEMPLES/Chap07`, qui viennent d'une notation Linux et qui sont nécessaires dans le `make`. Sous Windows, les deux notations sont acceptées.

Pour finir, nous allons tester nos trois classes en revenant dans le répertoire `C:\JavaCpp\EXEMPLES\Chap07` et créer une classe `TestMonPaquet` :

```
import monpaquet;

public class TestMonPaquet {
 public static void main(String[] args) {
   Personne unePersonne = new Personne("Dupont", "Justin", "1960");
   unePersonne.setSalaire(5000, "Euros");
   unePersonne.un_test();

   Salaire salaire = new Salaire(1000, "Euros");
   System.out.println("Salaire annuel: " + salaire.salaireAnnuel() +
                                          " " + salaire.getMonnaie());
 }
}
```

Le programme compilera et s'exécutera correctement seulement en utilisant cette forme (incluse dans le `Makefile` standard) :

```
javac -classpath C:/JavaCpp/EXEMPLES/Chap07/devPaquet TestMonPaquet.java

"C:\Program Files\Java\jdk1.6.0_06\bin\java.exe" -classpath .;"C:\JavaCpp\EXEMPLES\
➥Chap07\devPaquet" TestMonPaquet
```

Avec le résultat :

```
Nom et prenom: Dupont Justin
Annee de naissance: 1960
Salaire annuel: 79200 Euros
Salaire annuel: 15840 Euros
```

Les classes utilisées sont disponibles dans le répertoire `monpaquet` du sous-répertoire `devPaquet`.

Pour le vérifier, nous pouvons renommer le répertoire `monpaquet` :

```
C:\JavaCpp\EXEMPLES\Chap07>move monpaquet lepaquet
C:\JavaCpp\EXEMPLES\Chap07>java TestMonPaquet
java.lang.NoClassDefFoundError: monpaquet/Personne
Exception in thread "main"
```

`TestMonPaquet` n'a pas trouvé la classe `Personne`.

La variable d'environnement CLASSPATH

Nous l'avons déjà rencontrée lors de la compilation précédente en utilisant `javac -class path` ou encore avec :

```
"C:\Program Files\Java\jdk1.6.0_06\bin\java.exe" -classpath .;"C:\JavaCpp\EXEMPLES\
➥Chap07\devPaquet" TestMonPaquet
```

La variable d'environnement `CLASSPATH` est composée de un ou de plusieurs chemins d'accès, séparés par un point-virgule (`;`) ou le caractère deux-points (`:`), suivant les systèmes d'exploitation, pour identifier les classes utilisées par l'application. Le caractère `;` doit être utilisé sous Windows et le caractère `:` sous Linux. Le point (`.`) n'indique pas la fin de la phrase ou de la ligne, mais le répertoire courant.

Nous donnerons un exemple pour l'exécution (`java.exe`). Dans le cas de la classe `TestMon Paquet`, nous savons déjà que le `main()` doit être présent dans le fichier `TestMonPaquet.java`. Ce n'est pas le cas ici, mais cette classe de test pourrait très bien utiliser d'autres classes présentes dans ce même répertoire (par le caractère `.`). Les autres classes qui sont nécessaires lors de l'exécution seront retrouvées dans le sous-répertoire `devPaquet\monpaquet` (`devPaquet` au travers du `classpath` et `monpaquet` au travers de l'`import`).

Nous aurions très bien pu définir un sous-ensemble `p1` :

```
import monpaquet.p1;
```

Il aurait alors fallu définir un :

```
package monpaquet.p1;
```

en s'assurant que tout était construit correctement dans un répertoire :

```
C:/JavaCpp/EXEMPLES/Chap07/devPaquet/monpaquet/p1
```

Nos classes dans un fichier d'archive .jar

Nous pouvons à présent introduire nos trois classes dans un fichier d'archive `.jar` de la manière définie dans le fichier `devJar.bat` dans le répertoire `devPaquet` :

```
jar cf monpaquet.jar monpaquet/*.class
copy monpaquet.jar ..
pause
```

`jar.exe` est un outil distribué avec le JDK de Java qui permet de compresser des classes Java compilées dans une archive.

Signer et vérifier un fichier .jar

Nous signalons au passage que les fichiers `.jar` peuvent être signés à des fins de vérification. Si nous exécutons la commande :

```
jarsigner -verify monpaquet.jar
```

nous obtiendrons :

```
jar is unsigned. (signatures missing or not parsable)
```

car notre fichier `monpaquet.jar` ne contient pas de signature. `jarsigner` est un outil disponible avec la distribution de Java de Sun Microsystems. `keytool`, un autre outil de cette même distribution, permet de créer une clé à des fins d'authentification sécurisée. Nous n'en dirons pas plus sur ce sujet.

Test avec le fichier monpaquet.jar

Nous avons copié ci-dessus, avec `devJar.bat`, le paquet `monpaquet.jar`. Il est maintenant possible de procéder différemment (`TestMonPaquetJar.bat`) :

```
"C:\Program Files\Java\jdk1.6.0_06\bin\java.exe" -classpath .;monpaquet.jar
➥TestMonPaquet
```

Les classes `Personne` et `Salaire` utilisées par `TestMonPaquet` sont incluses dans l'archive `monpaquet.jar`.

Un fichier `.jar` peut être ouvert avec WinZip ou 7-Zip (voir annexe B) :

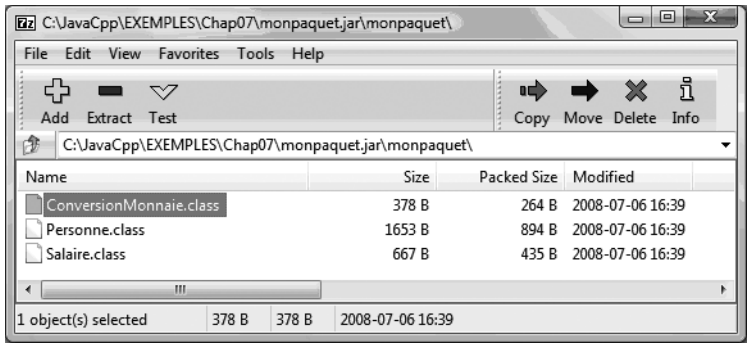

Figure 7-1
Visualisation du contenu d'un fichier .jar avec 7-Zip

Résumé des différentes étapes avec les fichiers .bat

En examinant les fichiers `.bat` disponibles dans le répertoire `EXEMPLES\Chap07`, nous comprendrons mieux les étapes de la construction d'un paquet Java (`package`) :

- `efface.bat` (voir figure 7-2) – Efface tous les fichiers d'objets créés par le `make`.
- `makefile.bat` (voir figure 7-3) – Effectue les trois étapes suivantes à la suite :
  - `make -f MakefileMonPaquet`
  - `make -f Makefile`
  - `devJar.bat` (dans le répertoire `devPaquet`)
- `MakeFileMonPaquet` – `Makefile` pour générer les trois classes Java du paquet dans le répertoire `devPaquet`.
- `Makefile` – Le traditionnel `make` pour ce chapitre, mais sans la génération du paquet Java.
- `devJar.bat` (voir figure 7-4) – Création du `.jar` qui sera utilisé pour exécuter `TestMonPaquetJar.bat`.
- `TestMonPaquetJar.bat` (voir figure 7-5) – Test de la classe `TestMonPaquet` avec le paquet `monpaquet.jar`.

Nous savons qu'il est possible d'exécuter un fichier `.bat` dans Crimson. La figure 7-2 présente l'exécution du fichier `efface.bat` :

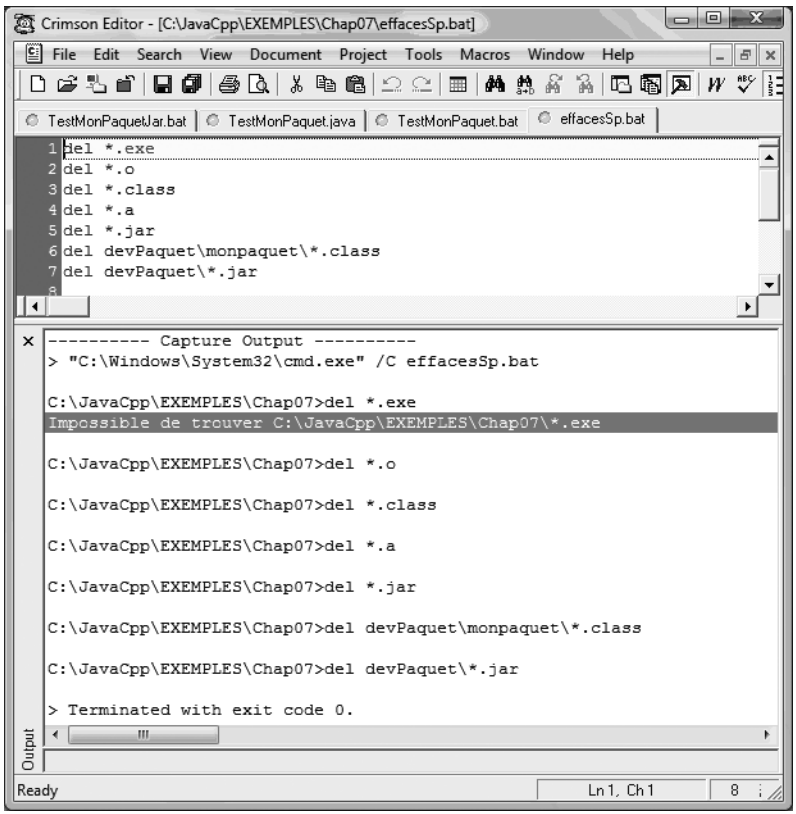

Figure 7-2

Effacement de tous les objets d'un répertoire d'exemple

Le `Makefile.bat` génère toutes les étapes de compilation du chapitre (si tous les objets ont été préalablement effacés) :

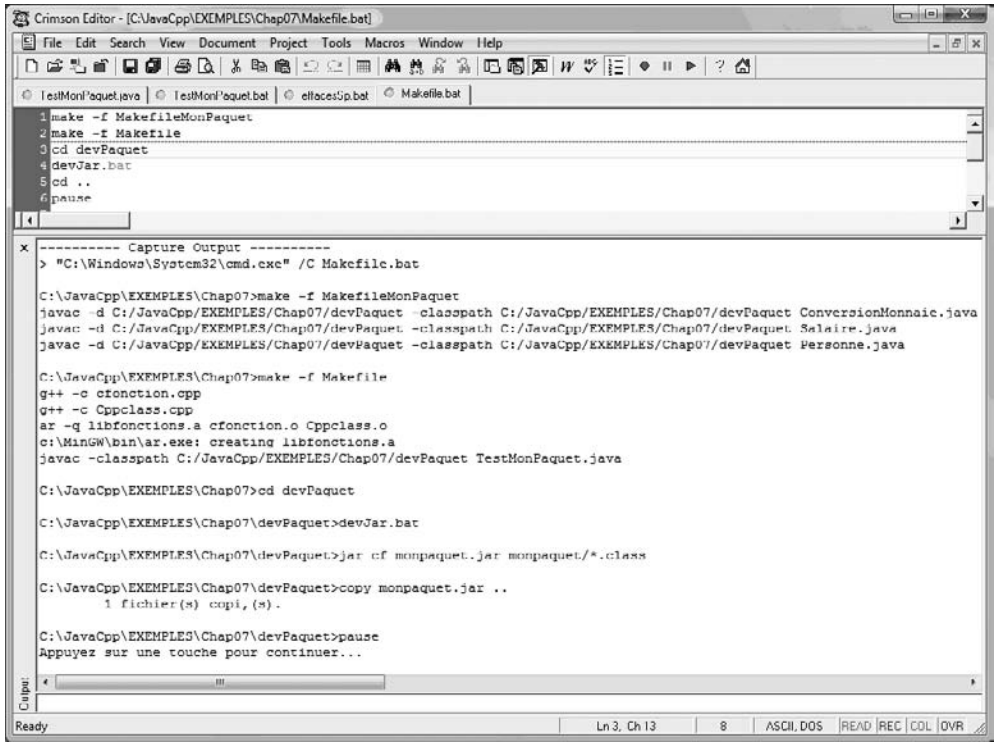

Figure 7-3

Recompilation du chapitre entier

Ici, il faudra entrer un `Retour` dans la fenêtre des résultats, pour terminer le processus à cause du `PAUSE` dans le fichier `.bat`.

Le fichier suivant, `devJar.bat`, se trouve dans le répertoire `devPaquet` et génère le fichier `monpaquet.jar` en le copiant à l'endroit correct :

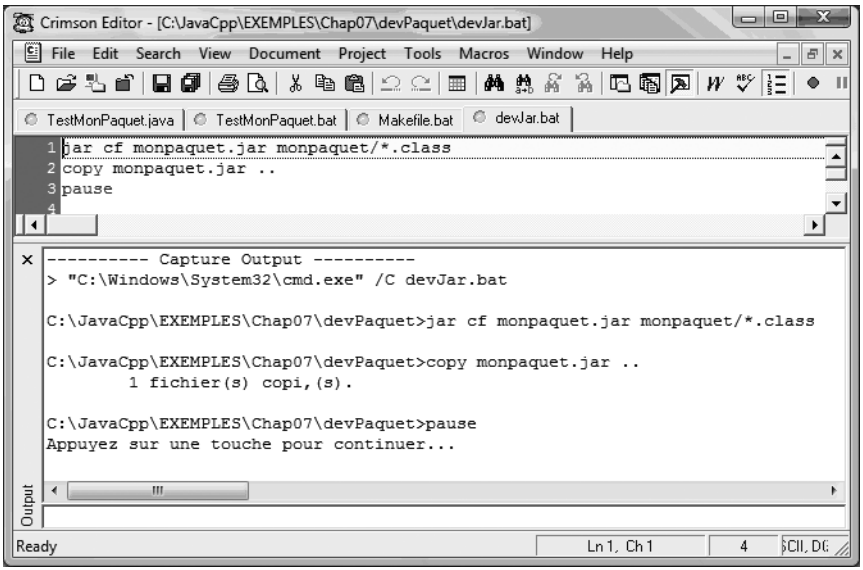

Figure 7-4

Création du fichier monpaquet.jar

Finalement, nous n'aurons pas de surprise avec `TestMonPaquetJar.bat` :

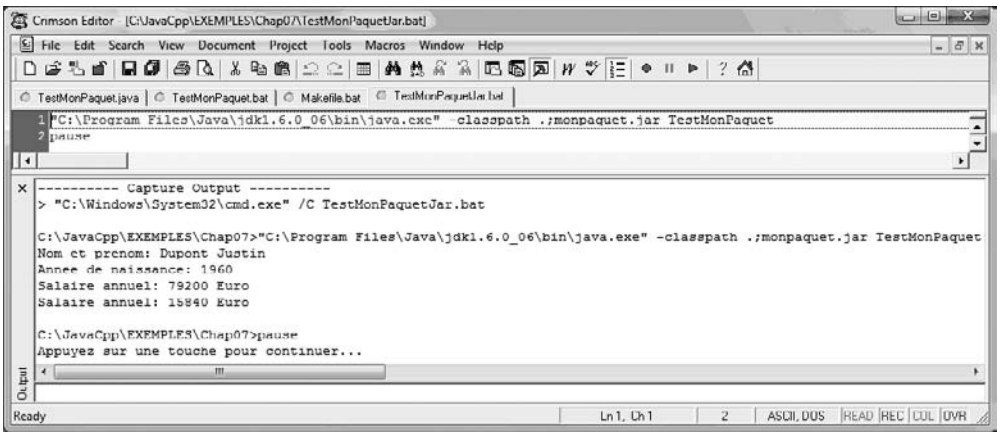

Figure 7-5

Le résultat final attendu

Les constructions de bibliothèques C et C++

Pour illustrer la création de bibliothèques en C ou C++, nous aurions pu choisir le même exemple qu'en Java, mais cela ne nous aurait rien apporté de plus. Nous devons en effet traiter le cas d'une fonction C séparément, et notre classe `ConversionMonnaie` avec un accès package n'a en fait aucune équivalence en C++.

Nous allons montrer ici le mécanisme de la construction de bibliothèques statiques en C++. Par statique, nous voulons dire que, lors de la compilation, des bibliothèques extérieures seront ajoutées au programme inclus dans le fichier binaire et non pas chargées pendant l'exécution comme nous le ferions avec des fichiers DLL.

Création d'une bibliothèque statique en C++

Pour ce faire, nous allons créer une fonction C et une classe C++, créer une bibliothèque pour ces deux dernières et les réutiliser dans un programme de test. Nous allons commencer par la définition de la fonction C et de la classe C++ dans deux fichiers d'en-tête séparés :

```
// cfunction.h
extern int cfonction(int, int);
```

et :

```
// Cppclass.h
class Cppclass {
  public:
    int fonction(int num1, int num2);
};
```

Ces fichiers d'en-tête sont essentiels car ils seront réutilisables par n'importe quel programmeur qui voudra réutiliser ces classes. Nous remarquerons incidemment que, dans `cfunction.h`, le nom des variables n'est en fait pas nécessaire. Nous passons à présent à l'implémentation du code :

```
// cfonction.cpp
#include "cfonction.h"

int cfonction(int num1, int num2) {
  return num1 * num2;
}
```

et :

```
// Cppclass.cpp
#include "Cppclass.h"

int Cppclass::fonction(int num1, int num2) {
  return num1 + num2;
}
```

Ces deux petits modules correspondent en fait à deux fonctions qui multiplient et qui additionnent deux nombres. La compilation de ces deux morceaux, `cfonction.o` et `Cppclass.o`, ne va pas causer de difficultés, car les quatre fichiers se trouvent dans le même répertoire. Il faut noter l'utilisation des "…" pour les fichiers d'en-tête, au lieu des <…> classiques. Ceci indique qu'ils sont présents sur le répertoire courant. Sinon, il faudra utiliser le paramètre `-I` lors de la compilation. Nous verrons les détails en fin de chapitre. Dans le `Makefile` nous aurons :

```
cfonction.o:    cfonction.h cfonction.cpp
                g++ -c cfonction.cpp

Cppclass.o:     Cppclass.h Cppclass.cpp
                g++ -c Cppclass.cpp
```

Il s'agit à présent de créer une bibliothèque qui va se faire ainsi :

```
libfonctions.a:   cfonction.o Cppclass.o
                  ar -q libfonctions.a cfonction.o Cppclass.o
```

`ar` veut dire archive et `-q` quick (archive rapide). Le nom de l'archive doit commencer par `lib` et avoir l'extension `.a`. Nous en comprendrons rapidement les raisons. Nous aurons donc un fichier d'archives `libfonctions.a` composé de `cfonction.o` et de `Cppclass.o`. Il est ensuite possible d'utiliser à nouveau `ar` avec le paramètre `–t` :

```
ar -t libfonctions.a
```

pour vérifier que notre bibliothèque contient bien notre nouvelle fonction C `cfonction()` et notre classe `Cppclass`.

Utilisation de notre bibliothèque C++

Afin d'utiliser cette bibliothèque, nous avons volontairement créé un sous-répertoire `TestFonctions` afin de vérifier l'accès aux fichiers d'en-tête et à la nouvelle bibliothèque. Nous allons écrire un petit programme dans le sous-répertoire `TestFonctions` :

```
// TestFonctions.cpp
#include <iostream>
#include <cfonction.h>
#include <cppclass.h>

using namespace std;

int main() {
  cout << cfonction(2,10) << endl;
  Cppclass ma_classe;
  cout << ma_classe.fonction(2,10) << endl;
}
```

qui va pouvoir nous permettre de vérifier nos deux fonctions et leurs inclusions dans le fichier binaire `TestFonctions.exe`. Nous percevons immédiatement la difficulté que vont rencontrer nos deux fichiers d'en-tête `cfonction.h` et `cppclass.h`. Le compilateur doit non

seulement trouver la définition de la fonction `cfonction()` et de la classe `Cppclass`, mais aussi ajouter le code de notre bibliothèque lors de la phase finale de la compilation. Cela va se faire de cette manière dans un `Makefile` du sous-répertoire `TestFonctions` :

```
INC = C:/JavaCpp/EXEMPLES/Chap07
LIB = C:/JavaCpp/EXEMPLES/Chap07

all:        TestFonctions.exe

TestFonctions.exe:  TestFonctions.o
                    g++ -I$(INC) -L$(LIB) -o TestFonctions.exe
                    ➥TestFonctions.o -lfonctions

TestFonctions.o:    TestFonctions.cpp
                    g++ -I$(INC) -c TestFonctions.cpp
```

Ce `Makefile` débute par la définition de deux variables, `INC` et `LIB`, et il faut noter la syntaxe pour y accéder : `$(INC)` et `$(LIB)`. Nous voyons apparaître deux nouvelles options de compilation :

- `-I.` – Définit le ou les chemins d'accès où le compilateur va rechercher les fichiers d'en-tête lors de la compilation du code source. Dans cet ouvrage, nous avons toujours utilisé des fichiers d'en-tête qui se trouvaient sur le répertoire de travail ou qui étaient connus par le compilateur (par exemple, `<iostream>` et `<string>`). Dans ces deux cas, le `-I` n'est pas nécessaire, et un `-I.` serait redondant (`.` pour le répertoire courant).
- `-L.` – Définit le ou les chemins d'accès où le compilateur va rechercher la ou les bibliothèques spécifiées par le paramètre `-l`. Ces bibliothèques doivent être liées avec les autres composants pour construire le fichier exécutable. La classe `string`, par exemple, fait partie de la bibliothèque du Standard C++, dont il n'est pas nécessaire de donner le chemin d'accès avec `-L`, car le compilateur C++ sait la retrouver.

La déclaration `-lfonctions`, dans la deuxième étape de la compilation, indique que le nom de la bibliothèque est en fait `libfonctions.a`. C'est une convention, `lib` et `.a` étant en fait rajoutés par le compilateur.

Les deux variables `INC` et `LIB` doivent être évidemment adaptées suivant l'installation sur le disque dur. Si plusieurs chemins d'accès sont nécessaires pour les fichiers d'en-tête ou les bibliothèques, il est possible de les chaîner. Si nous écrivions :

```
g++ -o application.exe -Ichemin1 -Ichemin2 -Lchemin4
                       -Lchemin3 application.cpp -leux -leui
```

ceci indiquerait, sans doute, que certains en-têtes utilisés dans `application.cpp` sont définis dans `chemin1` et `chemin2` et que les bibliothèques `libeux.a` et `libeui.a` sont disponibles dans les `chemin3` et `chemin4`. Il peut aussi arriver que le code d'`application.cpp` n'utilise que des classes et fonctions définies dans la bibliothèque `libeux.a`. Cela signifierait alors que le code de `libeux.a` nécessite d'autres fonctions disponibles dans `libeui.a`. Ce genre de dépendance devrait être défini dans la documentation de l'API.

Nous verrons au chapitre 21 comment construire une bibliothèque dll (Windows).

Résumé

Les extensions de fichier `.jar` en Java, `.a` et `.dll` en C/C++ nous indiquent la présence de bibliothèques. Nous avons appris dans ce chapitre aussi bien à les construire qu'à y accéder. En Java, le chemin d'accès doit être spécifié avec `CLASSPATH`.

Exercices

1. Créer, en Java, une classe `Banque` qui conserve notre fortune et retourne la valeur du compte après un certain nombre d'années et un taux donné. Créer une classe `Magasin`, qui possède elle-même un compte pouvant être alimenté soit par le client directement (prépaiement) soit par la banque, à condition que les achats dépassent l'état du compte. Introduire ces deux classes, dignes du commerce électronique, dans un package `monpaquet` et dans un fichier `.jar`. Écrire le `Makefile`, les procédures habituelles et un programme de test.
2. Reprendre la classe `Banque` ci-dessus, la convertir en une classe C++, et l'archiver dans une bibliothèque statique. Puis écrire un programme de test. Ne pas introduire les commandes de compilation dans un `Makefile`, mais les exécuter à la main, en donnant une brève explication pour chaque étape.

8

À quelques exceptions près

Une division par zéro devrait se rencontrer de manière exceptionnelle dans l'exécution d'un programme. De même, un accès dans un tableau en dehors des limites avec un index négatif ou plus grand que la dimension allouée doit être tout aussi exceptionnel et considéré plutôt comme une erreur de programmation. Quant à la lecture d'un fichier qui n'existe pas, cela fait partie de ces domaines de la conception des logiciels pour lesquels les explications sont multiples. L'opérateur a entré un nom de fichier incorrect ou bien a essayé d'accéder à des ressources du système qui ont disparu. Certains cas sont invraisemblables, et la seule alternative est de réinstaller un logiciel ou le système d'exploitation en entier !

La difficulté dans cet ouvrage est de maintenir un parallélisme entre les langages C++ et Java, afin de couvrir les sujets nécessaires à notre apprentissage. Le traitement des exceptions n'a été introduit que tardivement en C++, dans des versions récentes, alors qu'il fait partie intégrante des concepts de base du langage Java. Il n'y a, par exemple, aucune exception générée par les fonctions de la bibliothèque C, partie intégrante du langage C++. En C++, que nous utilisions des fonctions C ou des méthodes de classe de la bibliothèque **iostream** des entrées-sorties, le traitement se fait en général en utilisant les valeurs de retour de ces mêmes fonctions ou méthodes. À l'inverse, comme nous le verrons dans le chapitre suivant sur les entrées et sorties, une simple ouverture d'un fichier qui n'existe pas se traite en Java au moyen d'une exception.

Au contraire du C++, Java est né avec les exceptions

Dans les chapitres précédents, nous avons déjà rencontré un certain nombre d'exceptions qui font partie des toutes premières erreurs commises par un programmeur débutant en Java. Des erreurs de compilation lui indiquent qu'il faut traiter correctement les exceptions de

la même manière que, par exemple, le passage de paramètres pour les méthodes. Lors des premiers tests en Java, d'autres exceptions nous dévoilent qu'il faut étendre notre code pour vérifier ces cas particuliers.

Ce programme en Java :

```
public class Exception1 {
   public static void main(String[] args) {
     System.out.println(args[0].charAt(1));
   }
}
```

est évidemment incomplet. Il essaie d'accéder au deuxième caractère du premier paramètre passé au programme et peut produire deux types d'erreurs :

1. `ArrayIndexOutOfBoundsException` dans le cas où aucun paramètre n'est passé au programme ;
2. `StringIndexOutOfBoundsException` dans le cas où uniquement une lettre serait entrée comme premier paramètre.

Nous constatons que cette manière de procéder est propre et qu'elle oblige le programmeur à introduire le code nécessaire pour tester non seulement l'existence d'éventuels paramètres, mais aussi du contenu, comme sa dimension dans ce cas précis. Nous verrons plus loin les différentes possibilités offertes au programmeur.

En revanche, dans cette version C++, il n'y aura pas d'exception générée :

```
#include <iostream>

using namespace std;

int main(int argc, char **argv) {
  cout << argv[1][1] << endl;
}
```

Le programme soit sortira n'importe quel caractère, soit produira une opération non conforme et s'arrêtera brutalement : cela dépendra des compilateurs.

Utilisation des exceptions en Java

Dans cet ouvrage, nous ne traiterons que les exceptions de Java qui héritent de la classe `Exception`, qui elle-même est une sous-classe de `Throwable`. La classe `Exception` possède deux branches principales (sous-classes) :

1. `RuntimeException`, qui traite en principe des problèmes liés à la programmation. C'est le cas de nos `ArrayIndexOutOfBoundsException` et `StringIndexOutOfBoundsException` ci-dessus ou encore lors de l'emploi d'un pointeur `null`.
2. Les autres exceptions comme `IOException`, lorsque nous essayons de lire un fichier inexistant sur le disque.

Il est difficile d'établir des règles précises, mais, globalement, nous dirons que dans le cas de `RuntimeException` le code devrait être adapté pour que ce type d'erreur n'apparaisse jamais. Durant la phase de développement, le programmeur devrait s'assurer, en exécutant des tests adéquats, que tous les cas limites ont été vérifiés. Pour ces derniers, le code devrait être adapté judicieusement. Le code que nous avons vu précédemment ne produira plus d'erreur si nous l'écrivons ainsi :

```
public class Exception1 {
   public static void main(String[] args) {
     if ((args.length > 0) && (args[0].length() > 1)) {
       System.out.println(args[0].charAt(1));
     }
   }
}
```

Pour les autres exceptions, par exemple pour un fichier manquant, l'exception doit être capturée et le code nécessaire pour ce cas particulier doit être introduit.

Capture des exceptions

Généralement, un programmeur débutant en Java est surpris par l'apparition de ces erreurs de compilation mentionnant le traitement des exceptions. Au chapitre 2, nous avions introduit l'exemple suivant, que nous avons ici volontairement raccourci et modifié :

```
import java.io.*;

public class Cin1 {
  public static void main(String[] args) {
    int nombre = 0;

    BufferedReader stdin = new BufferedReader(newInputStreamReader(System.in));
    System.out.print("Entre un nombre: ");
    nombre = Integer.parseInt(stdin.readLine());
    System.out.println("\nLe nombre: " + nombre);
  }
}
```

Si nous compilons ce code, nous obtiendrons ceci :

```
Cin1.java:10: Exception java.io.IOException must be caught, or it must be declared in
➥the throws clause of this method.
      nombre = Integer.parseInt(stdin.readLine());
                                                ^
1 error
```

Pour comprendre la raison de cette erreur de compilation, il nous faut regarder la définition de `readline()` dans la classe `BufferedReader` :

```
public String readLine() throws IOException
```

Celle-ci nous indique qu'une `IOException` peut être retournée et que le programmeur doit absolument la prendre en considération. Le code correct peut être écrit ainsi :

```
import java.io.*;

public class Cin2 {
  public static void main(String[] args) {
    int nombre = 0;

    BufferedReader stdin = new BufferedReader(new InputStreamReader(System.in));
    System.out.print("Entre un nombre: ");

    try {
      nombre = Integer.parseInt(stdin.readLine());
    }
    catch(IOException ioe) {
      nombre = -1;
    }

    System.out.println("\nLe nombre: " + nombre);
  }
}
```

Nous rencontrons ici la séquence du `try` et du `catch()`. Celle-ci signifie que si l'une des exceptions de la classe `IOException` se produit dans la méthode `readline()`, alors le code de la partie `catch()` sera exécuté. Le lecteur peut consulter, dans l'API du JDK de Sun Microsystems, les nombreuses exceptions de la classe `IOException` et constater qu'une erreur dans le domaine de l'entrée à la console avec la méthode `readline()` est plutôt du domaine de l'invraisemblable et que l'instruction :

```
catch(IOException ioe) {
  nombre = -1;
}
```

nous donnera un résultat de –1, ce qui nous fait penser à un style de retour en C ou C++. Nous pourrions cependant laisser la variable `nombre` à 0. Ce serait sans doute plus raisonnable car le programme accepte aussi des entrées négatives. Nous entrerons d'ailleurs dans ces considérations, que nous pourrions presque considérer comme philosophiques, dans les prochaines pages.

Ignorer les exceptions

Il est tout à fait acceptable, dans le cas précédent, de vouloir ignorer les exceptions en les laissant passer au niveau supérieur. En revanche, ce ne serait pas le cas lors de la lecture d'un fichier auquel le programmeur s'attendrait à pouvoir accéder. Il faudrait alors traiter l'erreur avec du logiciel approprié pour contourner ce cas précis ou l'indiquer à l'extérieur. Le code suivant :

```
import java.io.*;
```

```
public class Cin3 {
  public static void main(String[] args) throws IOException {
    int nombre = 0;

    BufferedReader stdin = new BufferedReader(new InputStreamReader(System.in));
    System.out.print("Entre un nombre: ");

    nombre = Integer.parseInt(stdin.readLine());

    System.out.println("\nLe nombre: " + nombre);
  }
}
```

va compiler et fonctionner normalement. En cas d'erreur, tout à fait improbable, nous aurions un message directement sur la console. Cependant, la version `try catch()`, pour ce cas précis, est sans doute plus judicieuse.

À présent, continuons notre petit jeu avec ceci :

```
Entre un nombre: oo
Exception in thread "main" java.lang.NumberFormatException: oo
        at java.lang.Integer.parseInt(Compiled Code)
        at java.lang.Integer.parseInt(Compiled Code)
        at Cin3.main(Compiled Code)
```

Un nouveau problème se pose, bien plus grave que le précédent. Nous n'entrons pas des chiffres, mais des lettres ! Un vrai jeu télévisé ! Nous devons aussi constater que la même erreur se produira si nous ajoutons des lettres après des chiffres. Un programmeur connaît vraisemblablement la fonction `atoi()`, qui, elle, accepte aussi des lettres en fin de chaîne de caractères et qui peut être parfois bien pratique. Ici, nous n'avons aucune chance, car la méthode Java `Integer.parseInt()` est très restrictive. Le lecteur doit en fait se méfier et s'y reprendre à deux fois, car la ligne de code :

```
nombre = Integer.parseInt(stdin.readLine());
```

peut générer deux exceptions, une sur le `BufferedReader.readline()` et une sur le `Integer.parseInt()`.

Nous avons déjà examiné la première, et nous pouvons consulter la documentation de l'API de Java pour la seconde et découvrir cette définition :

```
public static int parseInt(String s) throws NumberFormatException
```

c'est-à-dire aussi avec un `throws`. Cependant, le compilateur nous a tout de même permis de compiler ce code, au contraire du cas précédent avec `IOException`. La raison en est simple : la classe `NumberFormatException` hérite de `RuntimeException`, qui n'a pas besoin d'être capturé. Le problème ici est donc clair : notre exception de la classe `RuntimeException` est en fait plus importante à nos yeux que l'`IOException` d'une entrée à la console, erreur tout à fait improbable. Bien que la séquence `try` et `catch()` pour notre `NumberFormatException` ne soit pas requise par le compilateur, nous allons tout de même l'introduire pour traiter notre cas d'erreur, tout à fait probable ici. Voici donc notre code :

```
import java.io.*;

public class Cin4 {
  public static void main(String[] args) throws IOException {
    int nombre = 0;

    BufferedReader stdin = new BufferedReader(new InputStreamReader(System.in));
    System.out.print("Entre un nombre: ");

    try {
      nombre = Integer.parseInt(stdin.readLine());
      System.out.println("\nLe nombre: " + nombre);
    }
    catch (NumberFormatException nfe) {
      System.out.println("\nL'entree ne contient pas que des chiffres !");
    }
  }
}
```

que nous pourrions modifier, en guise d'exercice, afin de demander à l'utilisateur, en cas d'erreur, de retaper son entrée avec des chiffres seulement. Une autre variante pourrait consister à analyser et à trier le `String` au retour de `stdin.readLine()` afin qu'il ne possède que des chiffres à l'entrée d'`Integer.parseInt`.

Une construction telle que :

```
catch (RuntimeException nfe) {
        System.out.println(nfe);
}
```

est tout à fait possible pour identifier quelle sous-classe de `RuntimeException` doit être capturée avant de terminer le code.

Plusieurs exceptions en une seule fois

Reprenons le même exercice, mais cette fois-ci en produisant une division de 1 000 sur le chiffre entré à la console. Il faudra donc considérer, en plus des entrées invalides, la division par zéro, d'où les deux `catch()` à la suite :

```
import java.io.*;

public class Cin5 {
  public static void main(String[] args) throws IOException {
    int nombre = 0;

    BufferedReader stdin = new BufferedReader(new InputStreamReader(System.in));
    System.out.print("Entre un nombre: ");

    try {
      nombre = 1000/Integer.parseInt(stdin.readLine());
      System.out.println("\nLe nombre: " + nombre);
```

```
    }
    catch (NumberFormatException nfe) {
      System.out.println("\nL'entree ne contient pas que des chiffres !");
    }
    catch (ArithmeticException ae) {
      System.out.println("\nL'entree ne peut être 0 !");
    }
  }
}
```

La division par zéro est traitée par la machine virtuelle Java à l'aide de l'exception `ArithmeticException`.

En général, il n'est ni conseillé ni logique de capturer une exception héritant de `RuntimeException`, car nous avons en principe toujours des solutions à ces types de traitements d'erreur, qui doivent être analysés pendant la phase de conception du programme et de ses classes.

Dans le cas précis ci-dessus, le programmeur devrait décider de lui-même, avant de laisser la machine virtuelle générer un `ArithmeticException`. Un code tel que :

```
int entree = Integer.parseInt(stdin.readLine());
if (entree == 0 ) {
  //quelque chose
}
nombre = 1000/entree;
```

pourrait remplacer la séquence `try` et `catch()` de notre `ArithmeticException`.

Lors de l'analyse de ce code, nous pourrions découvrir qu'il n'est pas nécessaire d'utiliser des entiers, car nous aimerions peut-être dessiner un graphique où nous aurions alors besoin de précision. En utilisant le `double`, il faudrait modifier deux lignes de ce code :

```
double nombre = 0;
nombre = 1000/Double.parseDouble(stdin.readLine());
```

et le résultat serait :

```
Entre un nombre: 0
Le nombre: Infinity
```

Nous n'avons donc plus le problème de l'exception `ArithmeticException`.

Lancement d'une exception

Si nous voulons nous-mêmes, dans notre code, générer une exception, deux possibilités nous sont offertes : soit lancer une exception à partir d'une classe existante, soit en créer une nouvelle comme nous le verrons ci-dessous. Mais pourquoi ne pas reprendre notre exercice précédent en l'adaptant à nos besoins ?

Nous aimerions définir une méthode qui nous retourne un `int` avec la valeur entrée à la console et qui nous génère une exception `EOFException` lorsque n'importe quelle erreur se

produit. `EOF` (*End Of File*) identifie en fait la fin du fichier, ce qui est plus ou moins raisonnable pour notre cas. Avant de coder la méthode proprement dite, il est tout à fait possible de définir ces paramètres :

```
public int recoitEntree() throws EOFException { }
```

Cela nous indique que l'utilisateur de cette fonction devra soit inclure une séquence de `try` et `catch()` pour l'exception `EOFException`, soit propager plus loin cette exception, comme nous l'avons vu précédemment. Mais passons maintenant au code :

```
import java.io.*;

public class Cin6 {
  public static void main(String[] args) {
    Cin6 monCin = new Cin6();

    for (;;) {
      try {
        System.out.print("Entre un nombre: ");
        System.out.println("Le nombre: " + monCin.recoitEntree());
      }
      catch (EOFException eofe) {
        break;
      }
    }

    System.out.print("Cin6 termine correctement");
  }

  public int recoitEntree() throws EOFException {
    int nombre = 0;

    BufferedReader stdin = new BufferedReader(new InputStreamReader(System.in));
    for (;;) {
      try {
        return Integer.parseInt(stdin.readLine());
      }
      catch (NumberFormatException nfe) {
        throw new EOFException();
      }
      catch (IOException ioe) {
        throw new EOFException();
      }
    }
  }
}
```

Et nous constatons une nouvelle forme d'instruction :

```
throw new EOFException();
```

Cette instruction va nous lancer une exception. `throw` accepte un objet de la classe `EOFException`, qui hérite elle-même d'`IOException`, d'`Exception` et, enfin, tout en bas, de la base des exceptions, `Throwable`. En d'autres termes, `throw` fonctionnera sur un `Throwable`.

Pour illustrer l'utilisation de `throw`, nous avons écrit une méthode séparée et remanié le code de manière significative. Nous constatons que les deux exceptions possibles, `IOException` et `NumberFormatException`, sont en fait converties en une seule, `EOFException`. Enfin, la méthode `recoitEntree()` ne contient pas de `System.out`.

Recommandation pour l'écriture de méthodes réutilisables

Le programmeur doit toujours s'imaginer l'utilisation de ces méthodes dans différents contextes ou interfaces. Des instructions telles que `System.out`, qui assume le fait qu'une console est attachée à l'application, devraient se trouver à l'extérieur. Dans un système, il y a beaucoup de processus de contrôle qui travaillent en arrière-plan.

Retour avec -1 comme en C ou C++

Un programmeur C++ qui programme régulièrement ces fonctions C ou méthodes C++ fera sans doute cette remarque : nous pourrions retourner un `-1` et ne pas lancer l'exception. En effet, ce peut être un débat intéressant, mais difficile.

Il nous apparaît cependant important d'indiquer qu'il faut avant tout rechercher la simplicité. Le traitement des erreurs peut se révéler délicat, et des raccourcis sont parfois nécessaires pour éviter soit trop de code (plus il y a de code, plus il y a de test, et donc besoin de temps mais aussi de risques d'erreurs supplémentaires) soit du code qui ralentit considérablement l'application.

Mais revenons au `-1`, qui dans notre cas est possible et fonctionne ! Si nous devions retourner un `String`, nous pourrions définir un `null` comme retour ou une chaîne vide. Si nous voulions absolument retourner un `-1`, le `String` devrait alors être passé en argument. Mais un `String` n'est pas mutable, et nous devrions retourner un `StringBuffer`. Pourquoi pas !

Il nous semble tout de même que notre séquence `try catch()` reste simple et élégante !

Création de nouvelles exceptions

Pour traiter le cas précédent, nous allons nous amuser un peu en créant une classe « exceptionnelle », car nous n'avons pas été heureux avec le nom précédemment choisi, notre `EOFException` ! Voilà pour le cadre. Pour la définition, nous choisirons une exception nommée `FindEntreeException`. Comme nous le remarquons, il est d'usage d'ajouter `Exception` dans le nom (à la fin) de notre nouvelle classe. Nous n'avons pas encore traité de l'héritage des classes, mais ce n'est pas bien complexe et il n'y aura pas besoin de faire semblant de comprendre !

Voici donc notre nouveau code, similaire au précédent :

```
import java.io.*;
class FindEntreeException extends IOException {
  public FindEntreeException() {}
  public FindEntreeException(String message) {
    super(message);
  }
}
public class Cin7 {
  public static void main(String[] args) {
    Cin7 monCin = new Cin7();

    for (;;) {
      try {
        System.out.print("Entre un nombre: ");
        System.out.println("Le nombre: " + monCin.recoitEntree());
      }
      catch (FindEntreeException fe) {
        System.out.println(fe);
        break;
      }
    }

    System.out.println("Cin6 termine correctement");
  }

  public int recoitEntree() throws FindEntreeException {
    int nombre = 0;

    BufferedReader stdin = new BufferedReader(new InputStreamReader(System.in));
    for (;;) {
      try {
        return Integer.parseInt(stdin.readLine());
      }
      catch (NumberFormatException nfe) {
        throw new FindEntreeException("Erreur1");
      }
      catch (IOException ioe) {
        throw new FindEntreeException("Erreur2");
      }
    }
  }
}
```

Notre nouvelle classe d'exception `FindEntreeException` hérite (`extends`) donc d'`IOException`. C'est une bonne pratique d'ajouter un deuxième constructeur, car nous pourrons alors définir un message pour une meilleure compréhension de l'exception et de son origine. La méthode `super(message)` est nécessaire car elle permet d'appeler la classe de base (ou

une précédente) afin de sauver le message, c'est-à-dire un attribut qui sera réutilisé lors d'un éventuel `System.out.println(fe);`. Nous allons constater que le message `Erreur1` est ainsi correctement transporté.

Si nous exécutons le code, nous aurons par exemple :

```
java Cin7
Entre un nombre: 101
Le nombre: 101
Entre un nombre: a
FindEntreeException: Erreur1
Cin7 termine correctement
```

Nettoyage à l'aide de finally

Il peut parfois être nécessaire de nettoyer certaines ressources, car nous n'avons pas en Java de destructeur comme en C++. Dans une séquence `try catch()`, un des deux blocs sera exécuté. Il nous faut donc un mécanisme permettant d'exécuter ce travail qui peut se révéler essentiel comme lors de l'utilisation des classes `AWT` (interface utilisateur). Ce mécanisme se fait à l'aide de l'instruction (appelée généralement clause) `finally`. En voici les détails dans un exemple :

```
public class Finalement {
  public static void main(String[] args) {
    Finalement monFin = new Finalement();

    monFin.methode1();
    monFin.methode2();

    System.out.print("Finalement termine correctement");
  }

  public void methode1() {
    try {
      Integer.parseInt("oooooh");
    }
    catch (NumberFormatException nfe) {
      System.out.println("Finalement1: erreur capturée");
    }
    finally {
      System.out.println("Finalement1");
    }
  }

  public void methode2() throws NumberFormatException {
    try {
      Integer.parseInt("aaaaah");
    }
    finally {
      System.out.println("Finalement2");
```

```
    }
  }
}
```

Et son résultat :

```
Finalement1: erreur capturée
Finalement1
Finalement2
Exception in thread "main" java.lang.NumberFormatException: aaaaah
        at java.lang.Integer.parseInt(Compiled Code)
        at java.lang.Integer.parseInt(Compiled Code)
        at Finalement .methode2(Compiled Code)
        at Finalement.main(Compiled Code)
```

Ce résultat peut paraître étrange à première vue, mais tout à fait correct et attendu. Les deux instructions :

```
      Integer.parseInt("oooooh");
      nteger.parseInt("aaaaah");
```

vont simplement créer un `NumberFormatException`, et aucun retour de paramètre n'est nécessaire pour ce petit exercice expliquant la clause `finally`.

`Integer.parseInt()` va donc produire une erreur que nous capturons. Ensuite, le code de la clause est aussi exécuté. Cela explique donc nos deux messages `Finalement1` produits par la `methode1()`.

Pour la `methode2()`, c'est différent, car nous avons défini un `throws` :

```
  public void methode2() throws NumberFormatException {
```

Ceci va relancer l'exception au niveau supérieur. La forme du `try` dans la `methode2()` n'est là en fait que pour activer le code du `finally`.

Le message de l'exception `aaaaah` sur la console vient du fait que nous avons ignoré le `NumberFormatException` dans le `main()`. Le message `"Finalement termine correctement"` n'apparaît donc pas.

Utilisation des exceptions en C++

Le traitement des exceptions dans le langage C++ ne faisait pas partie des premières versions des compilateurs. Ce n'est que plus tard qu'il est apparu, alors que de son côté la bibliothèque `iostream` (les entrées-sorties) avait déjà subi quelques modifications, mais toujours sans utilisation des exceptions. En Java, il n'est pas possible d'ignorer le mécanisme des exceptions lors d'une lecture de fichier, alors qu'en C++ un programmeur pourra même coder toute sa vie en ignorant les exceptions.

Dans cet ouvrage, nous n'allons donc pas trop nous étendre sur ce sujet, en particulier sur la hiérarchie de la classe `exception` dans la bibliothèque du Standard C++, car il pourrait être considéré comme un sujet à part entière. Nous allons simplement en expliquer le

mécanisme par quelques exemples. Notre travail sera d'ailleurs facilité par la partie Java présentée ci-dessus, à condition que le lecteur n'ait pas sauté ce passage à la manière d'un infâme `goto !`

Un programmeur Java qui maîtrise et utilise correctement les exceptions pourrait sans aucun doute adopter la même approche pour le développement de ces classes et applications C++. C'est un peu le but que l'auteur s'est fixé dans cette partie.

Un exemple sans exceptions

À l'inverse de notre présentation Java, nous serons ici plus directs et débuterons par un exemple plus concret, celui d'un distributeur automatique de billets, appelé Bancomat. Lors du retrait, nous allons considérer deux cas d'erreurs particuliers : il n'y a plus assez de billets dans l'appareil, et il n'y a plus assez d'argent sur le compte. Dans les deux cas, même si l'appareil pouvait retourner quelques billets, rien ne serait délivré par le Bancomat. Nous pourrions imaginer la classe suivante :

```
#include <iostream>
#include <string>

using namespace std;

class Bancomat1 {
 private:
   string utilisateur;
   int soldem;     // solde de l'appareil
   int soldeutil;  // solde de l'utilisateur

 public:
   void set_solde_machine(int lasoldem) {
     soldem  = lasoldem;
   }

   void set_utilisateur(const string lutilisateur) {
     utilisateur  = lutilisateur;
   }

   void set_solde_utilisateur(int lasoldeutil) {
     soldeutil  = lasoldeutil;
   }

   bool retrait_possible(int nb_billets);
};

bool Bancomat1::retrait_possible(int nb_billets) {
  if (((soldeutil - (100*nb_billets)) < 0) ||
      ((soldem - (100*nb_billets)) < 0))  return false;
  return true;
}
```

```
int main() {
  Bancomat1 labanque;

  labanque.set_solde_machine(1000);
  labanque.set_utilisateur("Tintin");
  labanque.set_solde_utilisateur(500);

  cout << "Test1: " << labanque.retrait_possible(4)  << endl;
  cout << "Test2: " << labanque.retrait_possible(6)  << endl;
  cout << "Test3: " << labanque.retrait_possible(11) << endl;
}
```

Toutes les méthodes `inline` que nous avons introduites devraient en fait être reliées à l'appareil pour obtenir les données, et à l'extérieur pour vérifier éventuellement le compte de l'utilisateur. L'identification de celui-ci ne sera pas utilisée dans nos exemples. Cependant, nous l'avons tout de même gardé pour montrer que si une extension de cet exercice était apportée, il deviendrait l'un des objets essentiels dans la conception du programme complet.

Dans cette version, la méthode :

```
bool retrait_possible(int nb_billets);
```

nous retournerait simplement un vrai ou faux :

```
Test1: 1
Test2: 0
Test3: 0
```

où il n'y a pas de possibilités de différencier l'erreur. Il faudrait alors retourner par exemple un `-1` ou `-2` pour chaque cas.

Un exemple avec exceptions

Nous allons donc reprendre l'exemple précédent, mais sans passer par les différentes phases que nous avons rencontrées en Java, et donner une solution qui pourrait presque être considérée comme définitive. Dans la première partie de ce code apparaissent deux nouvelles classes, `SoldeUtilisateurException` et `SoldeMachineException`, que nous allons utiliser pour lancer (`throw`) des exceptions :

```
#include <iostream>
#include <string>

using namespace std;

class SoldeUtilisateurException {
  private:
    int solde_du_compte;

  public:
```

```
    SoldeUtilisateurException(int un_solde) {
      solde_du_compte = un_solde;
    }
    int getsolde() {
      return solde_du_compte;
    }
};

class SoldeMachineException {
  private:
    int solde_billets;

  public:
    SoldeMachineException(int solde_machine) {
      solde_billets = solde_machine/100;
    }

    int getsolde() {
      return solde_billets;
    }
};

class Bancomat2 {
 private:
   string utilisateur;
   int soldem;     //solde de l'appareil
   int soldeutil;  //solde de l'utilisateur

 public:
   void set_solde_machine(int lasoldem) {
     soldem  = lasoldem;
   }

   void set_utilisateur(const string lutilisateur) {
     utilisateur  = lutilisateur;
   }

   void set_solde_utilisateur(int lasoldeutil) {
     soldeutil  = lasoldeutil;
   }

   void retrait_possible(int nb_billets);

   void retrait(int nb_billets);
};

void Bancomat2::retrait_possible(int nb_billets) {
  if ((soldeutil - (100*nb_billets)) < 0)  throw
                                    SoldeUtilisateurException(soldeutil);
  if ((soldem - (100*nb_billets)) < 0)  throw
                                    SoldeMachineException(soldem);
```

```
}

void Bancomat2::retrait(int nb_billets) {
  try {
    retrait_possible(nb_billets);
    cout << "Retirez vos " << nb_billets << " billets ! Merci !" << endl;
  }
  catch (SoldeUtilisateurException sue) {
    cerr << "Vous n'avez plus que " << sue.getsolde() <<
                                    " euros sur votre compte !" << endl;
  }
  catch (SoldeMachineException sme) {
    cerr << "Il n'y a plus que " << sme.getsolde() <<
                                    " billets dans la machine !" << endl;
  }
}

int main() {
  Bancomat2 labanque;

  labanque.set_solde_machine(1000);
  labanque.set_utilisateur("Tintin");
  labanque.set_solde_utilisateur(500);

  labanque.retrait(4);
  labanque.retrait(6);
  labanque.set_solde_utilisateur(2000);
  labanque.retrait(11);
}
```

Afin de mieux comprendre le code ci-dessus, nous pouvons déjà examiner le résultat, qui se présentera ainsi :

```
Retirez vos 4 billets ! Merci !
Vous n'avez plus que 500 euros sur votre compte !
Il n'y a plus que 10 billets dans la machine !
```

Nous avons aussi ajouté le mot `Exception` à la fin du nom de nos deux classes, `SoldeUtilisateurException` et `SoldeMachineException`. Ce n'est pas l'usage en C++, comme en Java, mais c'est tout de même plus lisible. Il faut noter que la forme Java :

```
throw new SoldeUtilisateurException(soldeutil);
```

n'est pas possible en C++. Nous avons d'ailleurs constaté que le programme compilait sans message d'erreur, mais ne fonctionnait simplement pas correctement. Il faut vraiment appeler le constructeur directement !

Dans la partie `main()`, nous aurions pu écrire ceci :

```
try {
  labanque.retrait_possible(4);
  labanque.retrait_possible(6);
```

```
  labanque.set_solde_utilisateur(2000);
  labanque.retrait_possible(11);
}
catch (SoldeUtilisateurException sue) {
  cerr << "Vous n'avez plus que " << sue.getsolde() << " sur votre compte !" << endl;
}
catch (SoldeMachineException sme) {
  cerr << "Il n'y a plus que " << sme.getsolde() << " billets dans la machine !"
  ➥<< endl;
}
```

et le résultat aurait été simplement :

```
Vous n'avez plus que 500 sur votre compte !
```

C'est propre et bien écrit. Si nous avions écrit le code suivant :

```
Bancomat2 labanque;
labanque.set_solde_machine(1000);
labanque.set_utilisateur("Tintin");
labanque.set_solde_utilisateur(500);
labanque.retrait_possible(4);
cout << "Retrait 4 a passé" << endl;
labanque.retrait_possible(6);
cout << "Retrait 6 a passé" << endl;
labanque.retrait_possible(11);
cout << "Retrait 6 a passé" << endl;
```

nous aurions alors obtenu comme résultat :

```
Retrait 4 a passé
abnormal program termination
```

Le compilateur C++ accepte de compiler ce code où aucune séquence de `try` ni de `catch()` n'est présente. Il faut donc reconnaître que Java est tout de même plus propre. Autant dire qu'aucune piste nous est donnée en C++ et que la présence de séquences d'un ou de plusieurs `try` et `catch()` va nous aider considérablement dans le traitement des erreurs.

Propager les exceptions

Dans l'exemple qui suit, pour lequel nous avons volontairement laissé de côté la définition et une partie du code de la classe `Bancomat3`, nous allons voir comment relancer une exception au niveau supérieur :

```
void Bancomat3::retrait(int nb_billets) {
  try {
    retrait_possible(nb_billets);
    cout << "Retirez vos " << nb_billets << " billets ! Merci !" << endl;
  }
  catch (SoldeUtilisateurException sue) {
    cerr << "Vous n'avez plus que " << sue.getsolde() <<
                                    " euros sur votre compte !" << endl;
```

```
  }
  catch (SoldeMachineException sme) {
    cerr << "Il n'y a plus que " << sme.getsolde() << " billets dans la machine !"
    ➥<< endl;
    throw;
  }
}

int main() {
  Bancomat3 labanque;

  labanque.set_solde_machine(1000);
  labanque.set_utilisateur("Tintin");
  labanque.set_solde_utilisateur(500);

  labanque.retrait(4);
  labanque.retrait(6);
  labanque.set_solde_utilisateur(2000);
  try {
    labanque.retrait(11);
  }
  catch (SoldeMachineException sme) {
    cerr << "Il n'y a vraiment plus que " << sme.getsolde() <<
                                    " billets dans la machine !" << endl;
    throw;
  }
}
```

En exécutant ce code, nous obtiendrons :

```
Retirez vos 4 billets ! Merci !
Vous n'avez plus que 500 euros sur votre compte !
Il n'y a plus que 10 billets dans la machine !
Il n'y a vraiment plus que 10 billets dans la machine !
abnormal program termination
```

Nous voyons ainsi comment, avec un simple `throw`, propager l'exception `SoldeMachine Exception` dans la partie `main()`. Lorsqu'une exception surviendra dans la méthode `retrait()`, elle sera passée au programme principal, qui va lui-même la propager au système d'exploitation, qui va nous retourner notre :

```
abnormal program termination
```

Sans la séquence `try` et `catch()` du `main()`, nous aurions eu le même résultat, sans le :

```
Il n'y a vraiment plus que 10 billets dans la machine !
```

Dans ce cas présent, un `throw` dans le `main()` n'est vraiment pas nécessaire, puisque nous contrôlons cette erreur et devrions en accepter les conséquences avec du code approprié.

Exception dans la bibliothèque Standard C++

Une hiérarchie de classe a aussi été définie dans la bibliothèque du Standard C++. Nous donnerons un exemple simple qui permettra éventuellement aux lecteurs de l'utiliser dans leurs applications futures. Nous insisterons à nouveau sur le fait que la bibliothèque Standard C++ est relativement récente et qu'il existe différentes versions de l'implémentation de cette bibliothèque. L'exemple suivant compilera très bien sous C++ Builder (Borland) sans l'en-tête `<stdexcept>`.

Nous rappellerons qu'en Java le mécanisme des exceptions est beaucoup plus restrictif, forçant le programmeur à introduire des séquences `try` et `catch()`, alors que les compilateurs C++ ne sont pas aussi limitatifs. Dans l'exemple qui suit, nous constaterons que la classe `string` du Standard C++ peut générer dans certains cas une exception :

```
// STLex.cpp
#include <string>
#include <iostream>
#include <stdexcept>

using namespace std;

int main(int argc, char* argv[])
{
  string mon_str = "ABC";
  try {
    char  mon_char = mon_str.at(10);
  }
  catch (const out_of_range &oer) {
    cerr << "Exception: out_of_range" << endl;
  }
  cout << "Fin de test 1" << endl;

  try {
    char  mon_char = mon_str.at(10);
  }
  catch (logic_error &le) {
    cerr << le.what() << endl;
  }
  cout << "Fin de test 2" << endl;

  try {
    char  mon_char = mon_str.at(10);
  }
  catch (exception &e) {
    cerr << e.what() << endl;
  }
  cout << "Fin de test 3" << endl;
  return 0;
}
```

L'instruction `mon_str.at(10);` accède donc au dixième octet d'une chaîne de caractères qui n'en contient que 3. La méthode `at()` de la classe `string` vérifie que l'index ne dépasse pas les limites possibles et va lancer une exception `out_of_range` si c'est le cas.

Si nous consultons le fichier d'en-tête `stdexcept`, nous découvrirons :

```
class out_of_range : public logic_error {
```

et :

```
class logic_error : public exception {
```

Nous voyons que `out_of_range` hérite de la classe `logic_error`, et ce dernier d'`exception`. L'héritage de classe sera traité au chapitre 12. Cela nous a donné l'idée de montrer ces trois manières de faire et leurs résultats :

```
Exception: out_of_range
Fin de test 1
pos >= length ()
Fin de test 2
pos >= length ()
Fin de test 3
```

Il serait donc tout à fait possible de traiter plusieurs exceptions et de les relancer par exemple sous une autre forme avec un `throw`.

Généraliser les exceptions en C++ comme en Java ?

Nous ne le pensons pas. Il y a trop de fonctions C ou de bibliothèques C++, comme les `iostream`, qui n'utilisent pas ce mécanisme. Cependant, le programmeur C++ doit être sensibilisé à ce problème et traiter correctement et systématiquement les cas d'erreurs. Toutes les méthodes de classe qui allouent ou utilisent des ressources pouvant provoquer des erreurs ou des exceptions pourraient retourner un booléen indiquant si une erreur s'est produite. Les classes elles-mêmes pourraient conserver des attributs indiquant par exemple un code d'erreur et un texte descriptif. Ces derniers pourraient être retournés aux applications qui aimeraient connaître les détails sur le problème.

Résumé

Ce chapitre, beaucoup plus délicat dans sa partie C++, sera certainement assimilé après écriture d'un certain nombre d'exercices comparatifs. Comme le mécanisme des exceptions en Java a été défini dès la création du langage, il ne pose pas de difficultés : il force le programmeur Java à traiter et à utiliser les exceptions. En C++ en revanche, c'est beaucoup moins aisé et sujet à discussion.

Exercices

1. Modifier la classe `Cin4` pour qu'elle demande à l'opérateur de répéter son entrée si celle-ci ne contient pas que des chiffres. Constater ce qui se passe si le nombre est trop grand.
2. Écrire la classe et l'exemple `Bancomat2` en Java.
3. Développer une classe C++ nommée `Arithmetic` avec une méthode `divise()` qui génère une exception `DivisionParZeroException`.

9

Entrées et sorties

Avec ce chapitre, nous entamons une partie beaucoup plus sérieuse, qui va nous permettre d'écrire de vrais programmes. Jusqu'à maintenant, toutes les entrées se faisaient sur le clavier et toutes les sorties sur la console. Nous allons à présent examiner comment lire et écrire des fichiers qui se trouvent sur le disque local. Ces fichiers peuvent contenir du texte que nous pourrions lire avec un éditeur traditionnel, des documents avec un format particulier ou encore des fichiers totalement binaires comme des exécutables ou des images, qui sont traités par des programmes spécifiques. Nous irons même lire, en Java, un document HTML sur un site Web qui pourrait se situer à l'extérieur de notre environnement. Nous allons examiner, en première partie de ce chapitre, un cas bien particulier : un fichier texte délimité.

Ce sujet va aussi nous permettre d'entrer dans la jungle des classes C++ et Java qui traitent des entrées et sorties. Un débutant en C++ ou en Java risque de se retrouver perdu s'il essaie de considérer chaque classe indépendamment et s'il essaie de comprendre son usage et son utilité. En Java notamment, c'est encore plus complexe, car parfois plusieurs classes doivent être associées en parallèle. Un exercice pourrait consister à écrire un exemple par classe, mais cela prendrait vite la place d'un ouvrage substantiel. Le meilleur moyen de s'en sortir est encore de se poser les deux questions suivantes :

- Est-ce que nous lisons ou nous écrivons ?
- Est-ce un fichier binaire ou simplement du texte ?

Lorsque nous avons répondu à ces deux questions, il suffit alors de consulter le sommaire de ce chapitre, de sauter au paragraphe approprié et d'examiner le code associé. Cette manière de faire peut sembler bizarre, mais elle a le mérite d'être pragmatique et efficace pour s'y retrouver dans cette jungle.

Avant de passer aux cas d'étude, nous allons présenter dans ce tableau les différents exemples que nous avons choisis de traiter :

Tableau 9-1 Les combinaisons de lecture et d'écriture

	Lecture texte	Lecture binaire	Écriture texte	Écriture binaire
C++	Lecture d'un fichier Access délimité.	Commande Linux `strings` : extraction de chaînes visibles.	Information structurée au format XML.	100 octets binaires au hasard.
Java	Lecture d'un fichier Access délimité.	Chargement d'une sauvegarde du jeu d'Othello.	Sauvegarde d'une partie du jeu d'Othello.	Sauvegarde d'une partie du jeu d'Othello.

Comme extra nous avons ajouté la lecture en Java d'une page Web (HTML). Une page Web peut aussi être considérée comme un fichier texte. Cependant, elle n'est pas accessible sur le disque de notre PC, mais sur Internet.

Du texte délimité à partir de Microsoft Access

Access est le fameux outil de bases de données Microsoft. Si nous ne le possédons pas, ce n'est pas très important, car le fichier texte que nous allons créer peut être construit à la main.

Au chapitre 4, nous avons créé notre première classe nommée `Personne`, qui contenait un nom, un prénom et une année de naissance. Nous allons donc créer une table dans Access qui contient ces trois champs. Nous pouvons prendre, par exemple, `bd1.mdb` comme nom de base de données, `Personnes` comme nom de table et `Nom`, `Prenom` et `Annee_Naissance` comme noms de champs. `Nom` et `Prenom` sont des champs texte alors que `Annee_Naissance` sera de type numérique. Nous définirons sur cette table une clé primaire nommée `Numero`. Finalement, nous entrerons les enregistrements suivants :

Tableau 9-2 Notre table Microsoft Access

Nom	Prenom	Annee_Naissance
Haddock	Capitaine	1907
Kaddock	Kaptain	1897

Lorsque la table aura été enregistrée, nous allons l'exporter (Fichier > Enregistrer sous > Exporter) en tant que fichier texte avec le nom `Personnes.txt` après avoir choisi le format délimité avec le point-virgule comme séparateur. Le fichier texte peut être lu par un éditeur de texte traditionnel comme Crimson :

```
1;"Haddock";"Capitaine";1907
2;"Kaddock";"Kaptain";1897
```

Ce qui est intéressant ici, c'est de constater que le fichier `Personnes.txt` peut aussi être créé et modifié par un traitement de texte ou par un programme, comme nous allons le

voir ici. Le nouveau fichier `Personnes.txt` peut très bien être importé après modification dans **Access** de Microsoft, ce qui peut se faire avec le menu Importer (Fichier > Données externes > Importer). Il est évident que cette méthode n'est pas appropriée s'il fallait changer continuellement des données. Il faudrait alors travailler directement avec un serveur SQL ou une interface ODBC, ce que nous verrons au chapitre 20.

Lecture de fichiers texte en C++

Nous allons utiliser à présent la classe `ifstream` pour lire notre fichier `Personnes.txt`. Cette classe fait partie de la bibliothèque `iostream` du C++ traditionnel. Nous aurions pu utiliser des fonctions C comme `open()` ou `fopen()`, mais elles ne supportent pas les opérateurs << ou >>, qui sont devenus les outils traditionnels du C++. Nous allons donc les laisser aux oubliettes, comme d'ailleurs toutes les fonctions C qui peuvent être avantageusement remplacées par des méthodes de classe de la bibliothèque du Standard C++, lorsque celles-ci sont disponibles. Nous allons directement écrire le code pour exécuter cette tâche, puis passer aux explications. Cette méthode de présentation est plus directe et se retrouvera tout au long de cet ouvrage.

```
// lecture_texte.cpp
#include <string>
#include <iostream>
#include <fstream>

using namespace std;

class Lecture_texte {
 private:
   string nom_fichier;

 public:
   Lecture_texte(const string le_fichier);
   void lecture();
};

Lecture_texte::Lecture_texte(const string le_fichier)
              : nom_fichier(le_fichier) {
}

void Lecture_texte::lecture() {
  ifstream ifichier(nom_fichier.c_str());

  if (!ifichier) {
    cerr << "Impossible d'ouvrir le fichier " << nom_fichier << endl;
    return;
  }

  string une_ligne;
  int numero_de_ligne = 1;
```

```
  while (ifichier >> une_ligne) {
    cout << numero_de_ligne << ":  " << une_ligne << endl;
    numero_de_ligne++;
  }
}

int main() {
  Lecture_texte les_personnes("Personnes.txt");
  les_personnes.lecture();
}
```

Le `<fstream>` fait son apparition. C'est dans ce fichier d'en-tête que la classe `ifstream` est définie. Le constructeur d'`ifstream` s'attend à recevoir un pointeur à une chaîne de caractères correspondant au nom du fichier. Ce dernier pourrait contenir le chemin d'accès complet du fichier. La méthode `c_str()` de la classe `string` fait le travail. La construction :

```
if (!ifichier) {
```

est un peu particulière. Elle détermine en fait si le fichier `Personnes.txt` est effectivement accessible. La boucle :

```
while (ifichier >> une_ligne) {
```

nous permet de lire ligne par ligne notre fichier `Personnes.txt`. Une ligne de texte dans un fichier se termine par un `\n` ou `\r\n`, mais ces derniers ne sont pas copiés dans le `string une_ligne`, car ils sont filtrés par l'opérateur `>>`. Lorsque la fin du fichier est atteinte, la boucle se terminera.

> **Note**
>
> `\n` (octal 012) termine en général une ligne de texte édité ou construit sous Linux.
>
> La séquence `\r \n` (octal 015 et 012) est applicable dans notre cas, car le fichier est préparé et enregistré sous DOS.

Le programme ci-dessus nous sortira le résultat suivant :

```
1:  1;"Haddock";"Capitaine";1907
2:  2;"Kaddock";"Kaptain";1897
```

qui sera identique au programme Java qui va suivre. Il est important de noter que si une ligne vide est ajoutée en fin de fichier, elle sera traitée comme un enregistrement vide par Access de Microsoft en cas d'importation. Le `7` de `1897` devra être le dernier caractère du fichier.

La méthode getline()

Que se passerait-il avec le programme précédent si les données contenaient des espaces ? Ou, en d'autres mots, si notre fichier `Personne.txt` contenait les données suivantes :

```
1;"Haddock";"Le Capitaine";1907
2;"Kaddock";"Kaptain";1897
```

Il suffit de l'essayer pour constater que le résultat est loin de nous satisfaire :

```
1:  1;"Haddock";"Le
2:  Capitaine";1907
3:  2;"Kaddock";"Kaptain";1897
```

Le problème vient de l'instruction suivante :

```
while (ifichier >> une_ligne) {
```

Si une espace ou autre tabulateur est découvert dans le fichier, l'opérateur >> stoppera son transfert. Le nom de la variable n'est vraiment pas approprié, et `une_ligne` devrait plutôt être nommée `un_mot`. En fait, les points-virgules (;) ne sont pas considérés comme des séparateurs par l'opérateur >> de la classe `istream`. Pour que notre programme fonctionne avec `"Le Capitaine"`, il nous faudrait modifier le code comme suit :

```
// lecture_texte2.cpp
#include <string>
#include <iostream>
#include <fstream>

using namespace std;

class Lecture_texte {
 private:
   string nom_fichier;

 public:
   Lecture_texte(const string le_fichier);
   void lecture();
};

Lecture_texte::Lecture_texte(const string le_fichier)
              : nom_fichier(le_fichier) {
}

void Lecture_texte::lecture() {
  ifstream ifichier(nom_fichier.c_str());

  if (!ifichier) {
    cerr << "Impossible d'ouvrir le fichier " << nom_fichier << endl;
    return;
  }

  char une_ligne[1024];
  int numero_de_ligne = 1;

  while (ifichier.getline(une_ligne, 1024)) {
    cout << numero_de_ligne << ":  " << une_ligne << endl;
    numero_de_ligne++;
  }
}
```

```
int main() {
  Lecture_texte les_personnes("Personnes2.txt");
  les_personnes.lecture();
}
```

Nos données sont lues du fichier `Personnes2.txt` et le résultat sera correct :

```
1:  1;"Haddock";"Le Capitaine";1907
2:  2;"Kaddock";"Kaptain";1897
```

La différence vient de notre :

```
while (ifichier.getline(une_ligne, 1024))
```

qui va lire une ligne entière jusqu'à un maximum de 1 024 caractères. Le `getline()` va en fait s'arrêter dès qu'il trouve un `\n` ou un `\r`, qui correspond à la fin de la ligne. Le `while()` se terminera aussi dès que la fin du fichier sera atteinte.

Pour terminer, il nous faut absolument indiquer une autre construction :

```
char un_char;
ifichier.get(un_char);
cout << "Un caractère: " << un_char << endl;
```

`get()` est une méthode de la classe `istream` (classe de base d'`ifstream`) qui peut être utile dans des situations où il se révèle nécessaire de lire caractère par caractère. Nous l'utiliserons dans l'exercice 2 avec son équivalent `put()` pour l'écriture dans la classe `ostream` (classe de base d'`ofstream`) :

```
ofstream ofichier(…);
char onechar = …;
ofichier.put(onechar);
```

Lecture de fichiers texte en Java

Comme pour le programme ci-dessus, qui lit notre fichier délimité `Personne.txt` exporté depuis Microsoft Access, la version Java (`LectureTexte.java`) s'écrira de cette manière :

```
import java.io.*;

public class LectureTexte {
 private String nom_fichier;

 public LectureTexte(String le_fichier) {
   nom_fichier = le_fichier;
 }

 public void lecture() throws Exception {
   String une_ligne;
   BufferedReader in = new BufferedReader(new FileReader(nom_fichier));
   int numero_de_ligne = 1;
```

```
    for (;;) {
      une_ligne = in.readLine();
      if (une_ligne == null)  break;
      System.out.println(numero_de_ligne + ":  " + une_ligne);
      numero_de_ligne++;
    }
  }

  public static void main(String[] args) throws Exception {
    LectureTexte les_personnes = new LectureTexte("Personnes.txt");
    les_personnes.lecture();
  }
}
```

et nous obtenons le même résultat qu'en C++.

La première remarque doit se faire sur ce `throws Exception`, car nous avons ici décidé de rejeter toutes les exceptions qui peuvent être générées dans la méthode `lecture()`. Le code serait cependant plus robuste si nous retournions, par exemple, un `boolean` de la méthode `lecture()`.

La classe `BufferedReader` possède une méthode `readline()` qui permet de lire un tampon de lecture ligne par ligne, de la même manière que notre exemple en C++. Le `Buffered Reader` est associé à un fichier ouvert en lecture au travers de la classe `FileReader`. Une question se pose immédiatement : que se passe-t-il si notre fichier `Personnes.txt` n'existe pas ? Lorsque le constructeur de `FileReader` ne peut ouvrir ce fichier, il va générer une exception nommée `FileNotFoundException`. Ici, nous utilisons un mécanisme particulier pour ignorer les exceptions : la combinaison `throws Exception`. Ces exceptions seront ignorées non pas à l'exécution, mais seulement dans le code où les séquences de `try` et `catch` ne sont pas nécessaires.

Utilisation de la variable separatorChar

Nous devons aussi dire quelques mots sur le caractère de séparation des répertoires. Ce petit programme :

```
import java.io.*;
public class FichierSeparateur {
  public static void main(String[] args) throws Exception {
    File mon_fichier = new File("Personnes.txt");

    System.out.println("Chemin: " + mon_fichier.getAbsolutePath());
    System.out.println("Séparateur: " + File.separatorChar);
  }
}
```

nous donnera :

```
Chemin: E:\JavaCpp\EXEMPLES\Ch06\Personnes.txt
Séparateur: \
```

Cela nous indique la location du fichier sur le disque. Le séparateur sera différent sous Linux (/), et attention : la variable statique `File.separatorChar` doit être absolument utilisée si nous devons accéder ou rechercher un répertoire dans le chemin complet et bénéficier d'un programme qui fonctionne aussi sous Linux.

Si nous déplaçons notre fichier `Personne.txt` dans un sous-répertoire `xxx`, les deux formes suivantes sont acceptables aussi bien sous Windows que sous Linux :

```
File mon_fichier = new File("xxx/Personnes.txt");
File mon_fichier = new File("xxx\\Personnes.txt");
```

Les deux `\\` sont nécessaires, car le premier est le caractère d'échappement pour le suivant. Si nous voulions rester totalement portables entre différents systèmes, une instruction telle que :

```
String fichier_complet = System.getProperty("user.dir")
                         + File.separator + "Personnes.txt";
```

serait souhaitable. La propriété `user.dir`, qui est reconnue par toutes les machines virtuelles de Java, nous donne le répertoire courant.

Lecture de fichiers sur Internet en Java

Lire des fichiers sur Internet en C++ ne sera pas traité dans cet ouvrage car accéder à Internet en C++ nécessite des bibliothèques spécialisées qui sont spécifiques à la machine et au système d'exploitation. En Java, cette fonctionnalité est extrêmement simple. Lire un fichier sur le disque local ou sur Internet est tout à fait similaire.

Pour cet exercice, nous avons choisi un fichier HTML très compact, que nous avons nommé `mon_test.html` et installé sur un serveur Web Apache (`http://www.apache.org`) sur notre machine. L'adresse URL est donc `http://localhost/mon_test.html` et apparaît de cette manière avec Firefox ou Microsoft Internet Explorer :

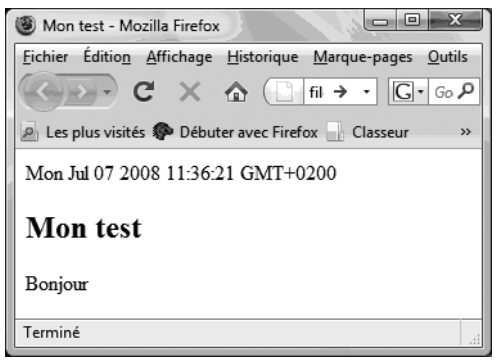

Figure 9-1

Page Web mon_test.html avec Firefox

```
Sun Oct 31 11:18:35 UTC+0100 1999
Mon test
Bonjour
```

Pour ceux qui ne désirent pas installer un serveur Apache, il est tout à fait possible de spécifier une autre adresse URL en utilisant la forme :

```
file:///C:/JavaCpp/EXEMPLES/Chap09/mon_test.html
```

sous Firefox qui correspond à un fichier sur le disque local. Le code Java simplifié (`LectureUrl.java`) se présente ainsi :

```
import java.net.*;
import java.io.*;

public class LectureUrl {
  public static void main(String[] args) throws Exception {
    URL apache_local = new URL("http://localhost/mon_test.html");

    BufferedReader in = new BufferedReader(new
                          InputStreamReader(apache_local.openStream()));

    String inputLine;
    while ((inputLine = in.readLine()) != null) {
      System.out.println(inputLine);
      in.close();
    }
}
```

et le résultat sera le suivant :

```
<html>
<head><title>Mon test</title></head>
<body>
<script language="JavaScript">document.write(new  Date());</script>
<h2>Mon test</h2>
Bonjour
</body>
</html>
```

Si nous utilisons la forme fichier, il nous faudra ajouter sous Windows une série de `\\`, car le caractère `\` doit être entré avec son caractère d'échappement :

```
URL apache_local = new URL("file:\\\\\\C:\\JavaCpp\\EXEMPLES \\Ch09\\mon_test.html");
```

En ce qui concerne le code, nous retrouvons la classe `BufferedReader`, que nous utilisons cette fois-ci sur un `InputStreamReader`. Cette dernière classe va nous offrir une transparence complète au niveau du protocole HTTP, protocole nécessaire pour lire un document sur le réseau Internet.

Dans l'annexe E, nous présentons NetBeans, environnement qui permet de développer des applications Web. Le serveur http Apache Tomcat est intégré à NetBeans.

Lecture de fichier binaire en C++

Il nous est parfois nécessaire de devoir extraire d'un fichier binaire, par exemple d'un programme exécutable, la partie visible de son contenu, c'est-à-dire tout ce qui correspond à du texte bien visible (caractères ASCII entre l'espace et le caractère binaire 127 (0x7F)). Un outil Linux, **strings**, existe d'ailleurs et s'avère souvent utilisé avec d'autres filtres pour rechercher des chaînes de caractères bien particuliers.

Pour ce faire, nous allons lire le fichier par bloc et ne sortir sur la console que les parties qui excèdent cinq caractères. Chaque séquence de caractères visibles sera précédée de sa position dans le fichier, ceci entre parenthèses et chaque fois sur une nouvelle ligne.

```
// strings.cpp
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

int main(int argc, char **argv) {
 if (argc != 2) {
   cerr << "Nombre d'arguments invalides" << endl;
   cerr << "Strings Fichier" << endl;
   return -1;
 }

 ifstream infile(argv[1], ios::in|ios::binary);
 if (!infile) {
   cout << "Fichier d'entrée n'existe pas";
   return -1;
 }

 const int bloc   = 512;  // dim bloc lu
 int   avant_bloc = 0;    // dim * n lu
 int   ncinq_char = 0;    // nombre de premiers chars lus
 char  cinq_char[6];      // cinq premiers chars

 cinq_char[5] = 0;        // toujours 5 imprimés

 char tampon[bloc];
 int  octets_lus;
 for (;;) {                      // lecture par bloc
   infile.read(tampon, bloc);
   octets_lus = infile.gcount();
   for (int i = 0; i < octets_lus; i++) {
     if ((tampon[i] >= ' ') && (tampon[i] < 127)) {
       if (ncinq_char < 5) {
         cinq_char[ncinq_char] = tampon[i];
         ncinq_char++;
       }
```

```
        else {
          if (ncinq_char == 5) {
            cout << (avant_bloc + i - 5) << ": " << cinq_char << tampon[i];
            ncinq_char++;
          }
          else  cout << tampon[i];
        }
      }
      else {  // char binaire
        if (ncinq_char > 5) {
          cout << endl;
        }
        ncinq_char = 0;
      }
    }
    if (octets_lus < bloc)  break;   // dernier bloc
    avant_bloc += bloc;              // avance compteur relatif
  }

  infile.close();
}
```

La constante `bloc` a pour valeur 512, et ce n'est pas par hasard. Lorsque nous devons lire des fichiers binaires, il n'y a pas de restrictions ni d'obligations de lire les octets l'un après l'autre. L'opérateur >> en C++ de la classe `ifstream` ou le `readline()` en Java, que nous avons vus tous deux précédemment, possèdent en fait un test interne sur chaque caractère pour identifier une fin de ligne. Ici, ce n'est pas nécessaire, et nous choisirons des dimensions de blocs suffisamment grandes. Si nous avons choisi une dimension de 512 et non pas de 100, de 1 000, de 511 ou de 513, cela vient sans doute de vieilles habitudes de l'auteur, qui considère encore qu'il est avantageux de choisir une dimension équivalant ou correspondant à un nombre entier de blocs sur un secteur du disque. À un moment ou à un autre, les primitives du système devront bien accéder au disque ou à son cache ! Ce qui est important ici, c'est le nombre d'appels de `infile.read()` qui se fera selon cette formule :

```
(dimension du fichier)/(dimension du bloc (512 ici)) + 1
```

Si nous lisons le fichier octet par octet, nous multiplierons ce nombre par 512. C'est de cette manière que nous obtiendrons les plus mauvaises performances. Choisir des blocs correspondant à la dimension du fichier n'est pas non plus un choix sensé, car certains fichiers pourraient dépasser les capacités de mémoire vive ou virtuelle. Si nous devions travailler avec des fichiers plus importants, il serait avantageux d'augmenter le bloc à 4 096, par exemple, ce qui n'entamerait pas trop les ressources en mémoire du programme. Cette discussion correspond aux calculs de performance que nous effectuerons au chapitre 15.

Nous lisons donc par blocs de 512, ce qui limite le nombre d'accès aux primitives qui vont finalement accéder au disque, à travers le système d'exploitation, et au contrôleur du disque. Ensuite, chaque caractère sera lu de la mémoire et donc vérifié plus rapidement. Le tampon `cinq_char` garde une suite consécutive de caractères qui sont présentés à l'écran

lorsque le sixième apparaît. Dans le cas contraire, nous remettons le compteur `ncinq_char` à zéro et recommençons. Cette manière de faire nous permet de sortir des séquences de caractères même s'ils se trouvent à cheval sur deux blocs de 512 octets. Sinon il faudrait avoir deux blocs `tampon` de travail, garder leurs index respectifs et aller de l'un à l'autre.

Écriture d'un fichier binaire en C++

Nous avons choisi ici un exemple un peu particulier, guère vraisemblable, mais qui nous permettra d'utiliser des fonctions pouvant poser problème avec d'autres compilateurs : c'est en effet un aspect important qu'un programmeur C++ doit absolument maîtriser. Nous allons écrire un fichier binaire de 100 octets avec des valeurs totalement aléatoires.

```
// ecritbin.cpp
#include <iostream>
#include <fstream>
#include <ctime>
#include <cstdlib>

using namespace std;

const int dim = 100;

int main(int argc, char* argv[])
{
  ofstream outfile("au_hasard.bin", ios::out | ios::binary);
  if (!outfile) {
    cerr << "Fichier de sortie ne peut être créé" << endl;
    return false;
  }

  char tampon[dim] ; // nos valeurs aléatoires
  srand(time(NULL));

  for (int i = 0; i < dim; i++) {
    tampon[i] = (char)((rand() * 256)/RAND_MAX);  // entre 0 et 255
    //tampon[i] = (char)rand(256);  // entre 0 et 255 certains compilateurs
  }

  outfile.write(tampon, dim) << endl;
  outfile.close();

  return 0;
}
```

La fonction C `srand()` va définir un point de départ pour le générateur de nombres aléatoires assuré par la fonction `rand()`. Pour ce faire, nous utiliserons l'heure actuelle en secondes, qui nous fournira des résultats différents à chaque utilisation du programme. Le fichier `au_hasard.bin` va contenir 100 octets de données aléatoires. La fonction `rand()` peut avoir différentes implémentations suivant les compilateurs : il vaut donc mieux consulter

attentivement la documentation ou plus simplement faire un test préalable. Nous comprendrons ainsi ce que nous retourne `rand()`, puisqu'il faut ensuite le multiplier par `256/RAND_MAX`. Nous verrons les détails un peu plus loin.

Sous Windows, nous pouvons associer le `.bin` à un éditeur hexadécimal pour visionner le résultat. Les trois indicateurs `ios` doivent être inclus (avec l'opérateur logique OU (`|`)) pour indiquer que :

- `ios::out` – fichier d'écriture ;
- `ios::binary` – fichier binaire.

Nous trouvons la définition de ces bits dans le fichier d'en-tête `streambuf.h`, qui est lui-même inclus dans `iostream`. Les `out`, `binary` et autres `noreplace` sont des énumérations publiques de la classe `ios`, et il y a parfois des variantes selon les implémentations de la bibliothèque du Standard C++ et les compilateurs.

Dans le code précédent, le fichier `au_hasard.bin` est écrit à chaque exécution du programme. Si nous ne voulions pas le remplacer, il faudrait s'assurer au préalable qu'il existe bien.

La classe `ofstream` est extrêmement simple à utiliser. De la même manière qu'`ifstream`, le `(!outfile)` nous permet de vérifier si l'ouverture du fichier a été faite correctement. Le `rand(256)` nous retourne un entier entre 0 et 255, c'est-à-dire une des valeurs possibles pour représenter un octet. Notre tampon de 100 caractères est donc du type `char`. Certains nous diront qu'un `unsigned char` (0 à 255) aurait été plus approprié qu'un char (–128 à 127). Cependant, le transtypage fonctionne correctement, et, pour être convaincus, nous pouvons le vérifier avec un éditeur hexadécimal, qui devrait nous montrer qu'il y a bien des octets avec le bit 7 correctement positionné. Le `rand(256)` devrait d'ailleurs nous allouer environ la moitié de l'ensemble !

La méthode `write()` de la classe `ofstream` a besoin ici de deux paramètres, le `tampon` (un `char *`) et le nombre d'octets à écrire (`dim`). C'est une bonne habitude de fermer correctement le `stream` avec un `close()` en fin d'opération.

Compilation conditionnelle

Comme nous venons de le voir dans l'exemple ci-dessus, il y a parfois des circonstances où une compilation conditionnelle s'avère nécessaire. Ce peut être un problème de plate-forme (DOS ou Linux (Unix)) ou de compilateur ne supportant pas certaines fonctions, ou les supportant différemment. Le morceau de code suivant devrait nous aider à pouvoir contourner ce genre de difficulté avec des compilations conditionnelles.

```
// define.cpp
#ifdef GNU
#define GNU_UNIX
#endif

#ifdef UNIX
#define GNU_UNIX
#endif
```

```
#include <iostream>

using namespace std;

int main(int argc, char* argv[])
{
#ifdef GNU
    cout << "Avec GNU" << endl;
#endif

#ifdef GNU_UNIX
    cout << "Avec GNU et Unix" << endl;
#endif

#ifndef DOS
    cout << "Sans DOS" << endl;
#endif

  cout << "Avec tous" << endl;
  return 0;
}
```

Si nous compilons ce code sans options de compilation, c'est-à-dire avec :

```
g++ -o define.exe define.cpp
```

nous obtiendrons :

```
Sans DOS
Avec tous
```

Dans ce code, nous découvrons les directives de compilation conditionnelle. Le message « Sans DOS » apparaît car la constante du préprocesseur n'a pas été définie. `#ifndef` veut donc dire : si DOS n'est pas défini, alors je compile le code jusqu'au prochain `#endif`, qui indique la fin d'une condition de compilation.

Si nous voulons à présent compiler avec la constante GNU, il faudra compiler le programme de cette manière :

```
g++ -DGNU -o define.exe define.cpp
```

ou bien :

```
g++ -DGNU -c define.o define.cpp
g++ -DGNU -o define.exe define.o
```

Ce qui nous donnera :

```
Avec GNU
Avec GNU et Unix
Sans DOS
Avec tous
```

Le `#define GNU_UNIX` permet de définir une nouvelle constante à l'intérieur du code source. Cela permet de ne pas avoir à recopier le même code pour des conditions différentes.

> **Note**
>
> Des constantes de compilations sont parfois utilisées pour du code de test durant les phases de développement. Lorsque le produit final est délivré, il suffit alors d'ajouter ou d'enlever la constante dans un `Makefile`.
>
> ```
> define.o:  define.cpp
> g++ $BTEST -c define.cpp
> ```

Nous avons déjà parlé de l'utilisation des `#define` au chapitre 6 pour la manipulation de fichiers d'en-tête multiples.

Écriture d'un fichier binaire en Java

Pour cette partie, nous allons revenir à notre jeu d'Othello (voir chapitres 5 et 20), dans lequel nous allons à présent sauvegarder et recharger une partie en cours (fichiers source `SauveOthello.java` et `ChargeOthello.java`).

Au chapitre 5, nous avions défini notre échiquier comme un `int jeu[10][10]`, avec quatre possibilités : 0 pour libre, 1 pour noir, 2 pour blanc et –1 pour les bords. Comme nous ne nous intéressons qu'à l'intérieur, un simple fichier de 64 octets (8×8) sera suffisant, et nous n'avons pas à traiter la valeur de –1 sur un octet de 8 bits.

Nous commencerons par l'écriture de notre fichier binaire, que nous allons nommer `othello.bin` :

```
import java.io.*;

public class SauveOthello {
  private String fichier;

  private static final int dim = 10;
  private int[][] othello = new int[dim][dim];

  public SauveOthello(String fichier_de_sauvetage) {
    fichier = fichier_de_sauvetage;
    for (int j = 1; j < dim-1; j++) {  // intérieur vide
      for (int i = 1; i < dim-1; i++) {
        othello[i][j] = 0;
      }
    }

    othello[4][5] = 1;  // blanc
    othello[5][4] = 1;  // blanc
    othello[4][4] = 2;  // noir
    othello[5][5] = 2;  // noir
```

```
  }

  public boolean sauve() {
    File outFile = new File(fichier);

    try {
      FileOutputStream out = new FileOutputStream(outFile);
      for (int j = 1; j < dim-1; j++) {
        for (int i = 1; i < dim-1; i++) {
          out.write(othello[i][j]);
        }
      }
      out.close();
    }
    catch (IOException e) {
      return false;
    }
    return true;
  }

  public static void main(String[] args) {
    SauveOthello mon_sauvetage = new SauveOthello("othello.bin");
    if (mon_sauvetage.sauve()) {
      System.out.println("Sauvegarde effectué");
    }
    else {
      System.out.println("Erreur de sauvegarde");
    }
  }
}
```

La première partie de ce code nous est déjà familière. Nous positionnons nos quatre pions de départ (voir chapitre 20 pour les règles du jeu) afin de vérifier le résultat produit. Pour ce qui est de l'écriture, nous allons essayer d'éclairer le lecteur sur la manière de procéder pour savoir quelles classes utiliser. Quelque part, il nous faut rechercher une méthode `write()`. Celle-ci doit pouvoir écrire des octets et non pas du texte ou des objets de taille variable.

`FileOutputStream` est la classe la plus élémentaire pour envoyer des données binaires dans un fichier au travers d'un flux. Elle possède des méthodes `write()` pour envoyer des octets, mais aussi des entiers. Elle reçoit aussi le nom du fichier de sortie comme paramètre de constructeur. `othello` étant un tableau d'entier, le fichier aura donc un format binaire déterminé par le type `int` du langage Java. Si nous devions lire `othello.bin` depuis un autre langage ou avec d'autres classes, nous pourrions rencontrer des difficultés. La séquence `try` et `catch()` devrait nous sortir les erreurs. Dans notre cas, il y a peu d'erreurs possibles. Si nous voulions provoquer une erreur, nous pourrions alors essayer d'écrire dans un fichier tel que `Z:\othello.bin`. Il faudrait alors modifier le code pour vérifier le type d'erreur et définir une action spécifique pour chaque type. Si nous écrivions :

```
    catch (IOException e) {
      System.out.println(e);
      return false;
    }
```

et vérifiions le résultat avec `Z:\othello.bin`, nous obtiendrions ceci :

```
java.io.FileNotFoundException: Z:\othello.bin (Le chemin d'accès spécifié est
➥introuvable)
Erreur de sauvegarde
```

Un bon traitement des erreurs nécessiterait donc un peu plus de code.

Lecture d'un fichier binaire en Java

Nous terminons avec la relecture (`ChargeOthello.java`) de notre fichier binaire `othello.bin` précédemment sauvegardé :

```
import java.io.*;

public class ChargeOthello {
  private String fichier;

  private static final int dim = 10;
  private int[][] othello = new int[dim][dim];

  public ChargeOthello(String fichier_de_sauvetage) {
    fichier = fichier_de_sauvetage;
  }

  public boolean charge() {
    File inFile = new File(fichier);

    try {
      FileInputStream in = new FileInputStream(inFile);
      for (int j = 1; j < dim-1; j++) {
        for (int i = 1; i < dim-1; i++) {
          othello[i][j] = in.read();
        }
      }
      in.close();
    }
    catch (IOException e) {
      return false;
    }
    return true;
  }

  public void test() {
    int i = 0;  // position horizontale
    int j = 0;  // position verticale
```

```
    for (j = 0; j < dim; j++) {
      for (i = 0; i < dim; i++) {
        if (othello[i][j] >= 0) System.out.print(" ");
        System.out.print(othello[i][j]);
      }
      System.out.println();
    }
  }

  public static void main(String[] args) {
    ChargeOthello mon_sauvetage = new ChargeOthello("othello.bin");
    if (mon_sauvetage.charge()) {
      System.out.println("Chargement effectué");
      mon_sauvetage.test();
    }
    else {
      System.out.println("Erreur de chargement");
    }
  }
}
```

Ici, nous faisons l'inverse : nous utilisons la classe `FileInputStream` et la méthode `read()`. Il n'y a aucun contrôle du format du fichier. Nous assumons le fait que le fichier contienne des entiers et possède le nombre correct de caractères. Il n'y a, à nouveau, aucun véritable traitement des erreurs. Pour vérifier que le programme lit correctement, il faudrait sans doute réutiliser la classe précédente et vérifier un plus grand nombre de possibilités que nos quatre premiers pions !

Comme les fichiers de sauvegarde du jeu d'Othello peuvent être réutilisés afin de tester notre jeu pendant la construction du programme, il est évident que l'utilisation d'un fichier binaire n'est pas intéressante. Il serait donc avantageux d'utiliser un format texte simple, afin de pouvoir modifier les sauvegardes avec un éditeur traditionnel.

Un format tel que :

```
VVVVVVVV
VVVVVVVV      V - position vide
VVVVBNVV
VVNNNVVV      B - pion blanc
VVVNBVVV
VVVVVVVV      N - pion noir
VVVVVVVV
VVVVVVVV
```

se comprendrait de lui-même. C'est ce que nous allons aborder à présent.

Écriture d'un fichier texte en Java

Nous allons continuer de jouer avec notre exemple d'Othello, représenté ci-dessous dans le code source `WriteOthello.java`, où les noirs viennent de jouer.

Le fichier produit, `othello.txt`, sera cette fois-ci un fichier de type texte traditionnel :

```
import java.io.*;

public class WriteOthello {
  private String nomFichier;

  private static final int dim = 10;
  private int[][] othello = new int[dim][dim];

  public WriteOthello(String fichierDeSauvetage) {
    nomFichier = fichierDeSauvetage;

    for (int j = 1; j < dim-1; j++) {  // intérieur vide
      for (int i = 1; i < dim-1; i++) {
        othello[i][j] = 0;
      }
    }

    othello[3][4] = 2;  // noir joue

    othello[4][4] = 2;  // noir
    othello[5][4] = 2;  // noir
    othello[4][5] = 2;  // noir
    othello[5][5] = 1;  // blanc
  }

  public boolean sauve() {
    try {
      PrintWriter out = new PrintWriter(new FileWriter(nomFichier));

      char pion;
      StringBuffer tampon = new StringBuffer("12345678");
      for (int j = 0; j < 8; j++) {
        for (int i = 0; i < 8; i++) {
          pion = 'V';  // vide
          if (othello[i + 1][j + 1] == 1)  pion = 'B';  // blanc
          else  if (othello[i + 1][j + 1] == 2)  pion = 'N';  // noir
          tampon.setCharAt(i, pion);
        }
        out.println(tampon);
      }

      out.close();
    }
    catch(IOException ioe) {
```

```
      return false;
    }
    return true;
  }

  public static void main(String[] args) {
    String fichierDeSauvetage = "othello.txt";
    WriteOthello mon_sauvetage = new WriteOthello("othello.txt");

    if (mon_sauvetage.sauve()) {
      System.out.println("Sauvegarde effectué dans " + fichierDeSauvetage);
    }
    else {
      System.out.println("Erreur de sauvegarde dans " + fichierDeSauvetage);
    }
  }
}
```

Nous rencontrons ici une nouvelle classe, `PrintWriter`. Cette classe nous permet d'écrire dans le fichier de la même manière que nous le faisons pour une sortie à l'écran avec notre `println()` habituel.

Lors de l'écriture binaire en Java, nous avions utilisé `FileOutputStream`. Ici, c'est presque pareil avec `PrintWriter`, mais `write()` est remplacé par `println()`. Cette dernière méthode nous écrira une chaîne de caractères terminée par une nouvelle ligne. Notre variable `tampon` possède une réserve pour 8 octets, qui seront initialisés, avant l'écriture, avec le contenu d'une ligne de notre tableau `othello`. Les lettres employées, `V`, `B` et `N`, remplaceront des valeurs binaires 0, 1 et 2, illisibles avec un éditeur.

Nous rappellerons que `StringBuffer` est mutable, au contraire de la classe `String`. Nous pourrions aussi construire notre chaîne avec cette dernière, mais cela serait moins efficace, puisqu'il faudrait composer le `String` avec une série de `+` (objet `String` régénéré à chaque fois).

Écriture d'un fichier texte en C++

Nous allons profiter de cette occasion pour présenter le XML, qui va nous permettre d'enregistrer des données dans un fichier texte traditionnel dans le cas de données structurées telles que des bases de données. Ce sera à la fois un exercice de style en C++, pouvant être adapté sans difficulté en Java, mais aussi une introduction essentielle pour cette technologie de plus en plus utilisée dans le commerce électronique.

Le XML pour l'information structurée

Le XML, ou *e**X**tensible **M**arkup **L**anguage*, a été conçu pour remédier aux insuffisances du HTML, ***H**yper **T**ext **M**arkup **L**anguage*, dans l'exploitation des informations structurées. Tout comme le HTML, le XML découle du SGML, ou ***S**tandard **G**eneral **M**arkup **L**anguage*,

qui a été développé pour maintenir une structure et un contenu standard pour des documents électroniques. Sur le site `http://www.w3.org` du World Wide Web Consortium, nous trouverons les recommandations pour les technologies du Web et les spécifications du XML.

Le XML a hérité du HTML, mais sans garder ses défauts. Il reste aussi beaucoup plus simple que son aïeul, le SGML. La structure hiérarchique d'un document est propre et l'information intégrée entre des balises symétriques, claires et concises. Le balisage XML permet une identification rapide du contenu des données.

Comme exemple, nous allons reprendre notre classe `Personne`, que nous avions développée au chapitre 4, mais sans l'année de naissance. Nous garderons les deux premiers attributs, le nom et le prénom. Le document XML devrait se présenter ainsi :

```
<?xml version="1.0"?>
<carnet>
  <personne>
     <nom>Haddock</nom>
     <prenom>Capitaine</prenom>
  </personne>
  <personne>
     <nom>Boichat</nom>
     <prenom>Jean-Bernard</prenom>
  </personne>
</carnet>
```

Les balises `personne`, `nom` et `prenom` sont similaires à celle du HTML. Dans notre `carnet`, nous avons deux `personne`, qui possèdent chacune un `nom` et un `prenom`. L'oubli du `é` est volontaire pour des raisons de programmation. Contrairement au HTML, la fin de balise, comme `</personne>`, est essentielle.

Sur la deuxième ligne, il n'y a pas de DTD, c'est-à-dire de déclaration de document type. Elle n'est pas obligatoire en XML, car notre document est bien formé, rigoureux et explicite. Un processeur XML n'aura aucune difficulté pour le traiter.

Écriture du fichier XML

L'exemple ci-dessous est simplifié à l'extrême et devait être en fait intégré à la classe `Personne` du chapitre 4, avec une méthode que nous pourrions appeler par exemple `write_XML()`. Nous avons ici créé une classe nommée `WriteXML`, qui va générer un fichier `personne.xml` sur le répertoire courant ::

```
// write_xml.cpp
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

class WriteXML {
```

```
  private:
    ofstream outfile;
    string avecBalise(const string balise, const string valeur) {
      return "     <" + balise + ">" + valeur +  "</" + balise + ">";
  }

  public:
    bool write_debut(const char *fichier) {
      outfile.open(fichier, ios::out);

      if (!outfile.is_open()) {
        return false;
      }

      outfile << "<?xml version=\"1.0\"?>" << endl;
      outfile << "<carnet>" << endl;
    }

    void write_record(const char *nom, const char *prenom) {
      outfile << "  <personne>" << endl;
      outfile << avecBalise("nom", nom) << endl;
      outfile << avecBalise("prenom", prenom) << endl;
      outfile << "  </personne>" << endl;
    }

    void write_fin() {
      outfile << "</carnet>" << endl;
      outfile.close();
    }
};
int main() {
  WriteXML mon_xml;

  if (!mon_xml.write_debut("personne.xml")) {
    cout << "Ne peut pas écrire dans le fichier personne.xml" << endl;
    return -1;
  }

  mon_xml.write_record("Haddock", "Capitaine");
  mon_xml.write_record("Boichat", "Jean-Bernard");
  mon_xml.write_fin();

  cout << "Fichier personne.xml généré" << endl;
}
```

Dans ce code, nous utilisons la méthode `open()` de la classe `ofstream`. Elle nous retourne un `void`, et il est donc essentiel d'appeler la méthode `bool is_open()` de cette même classe pour vérifier si le fichier a effectivement été créé et s'il est bien accessible pour y écrire nos données. Bien que le traitement des erreurs d'écriture soit réduit au minimum, nous fermons tout de même le fichier avec la méthode `close()`. L'emploi de l'opérateur `<<` de

la classe `ofstream` et du manipulateur `endl`, pour écrire le caractère de fin de ligne, relève vraiment d'une écriture élégante. Nous écrivons dans un fichier texte comme sur une console. La méthode privée `avecBalise()` nous permet d'exécuter le travail répétitif d'une manière simplifiée, sans une réflexion trop approfondie, car le contenu même d'un objet `personne` pourrait aussi y être intégré. Il nous faut aussi mentionner ici qu'il est tout à fait possible d'inclure, en XML, plusieurs objets d'un même type dans une structure. `personne` pourrait contenir plusieurs blocs de prénoms : ce serait d'ailleurs absolument nécessaire si nous voulions, par exemple, inclure la liste des enfants de cette même personne.

Après avoir exécuté le programme ci-dessus, qui crée le fichier `personne.xml`, nous pourrions le visualiser, par exemple avec Microsoft Internet Explorer, qui va filtrer ce document pour nous retourner ceci :

```
<?xml version="1.0" ?>
- <carnet>
- <personne>
  <nom>Haddock</nom>
  <prenom>Capitaine</prenom>
  </personne>
- <personne>
  <nom>Boichat</nom>
  <prenom>Jean-Bernard</prenom>
  </personne>
  </carnet>
```

Accès des répertoires sur le disque

De nombreux outils informatiques accèdent aux répertoires et aux fichiers d'un disque. Il est donc essentiel de couvrir ce sujet. Lorsque nous accédons au répertoire d'un disque et utilisons la commande `dir` sous DOS ou `ls -l` sous Linux (voir annexe C), le résultat nous est présenté sous forme de liste de fichiers ou de répertoires, et nous obtenons par exemple des données sur la protection, la dimension ou encore la date des fichiers. Les deux morceaux de code suivants peuvent être le départ d'un grand nombre de petits programmes qui peuvent être conçus plus particulièrement dans le cadre d'exercices de programmation. Nous pourrions nous imaginer des programmes de sauvegarde incrémentiel, de statistique, de compression ou encore de contrôle de versions (plusieurs versions du même code dans le cadre de projets informatiques, de développement ou de maintenance).

Lecture d'un répertoire en C++

Le code suivant utilise un certain nombre de fonctions de la bibliothèque C pour accéder au répertoire d'un disque :

```
// listdir.cpp
#include <sys/types.h>
#include <sys/stat.h>
#include <dirent.h>
```

```
#include <ctime>

#include <iostream>

using namespace std;
int main() {
 DIR    *pdir;
 struct dirent *pdirent;
 struct stat statbuf;
 if ((pdir = opendir(".")) == NULL) {
   cout << "Ne peut ouvrir le répertoire ." << endl;
   return -1;
 }

 while ((pdirent = readdir(pdir)) != NULL) {
   cout << "Fichier: " << pdirent->d_name << endl;
   if (stat(pdirent->d_name, &statbuf) != 0)  continue;
   if (statbuf.st_mode & S_IWRITE) {
      cout << "Nous avons la permission d'écrire" << endl;
   }
   else {
      cout << "Nous n'avons pas la permission d'écrire" << endl;
   }
   if (statbuf.st_mode & S_IFDIR) {
      cout << "C'est un répertoire" << endl;
   }
   cout << "La lettre du disque: " << (char)('A'+statbuf.st_dev) << endl;
   cout << "La dimension en octets: " << statbuf.st_size << endl;
   cout << "La dernière fois qu'il fut ouvert: "
        << ctime(&statbuf.st_ctime) << endl;
 }
 closedir(pdir);
 return 0;
}
```

Il est vraiment parfait, ce petit programme ! Nous montrerons ici une partie du résultat :

```
Fichier: Lecture_texte.java
Nous avons la permission d'écrire
La lettre du disque: E
La dimension en octets: 684
La dernière fois qu'il fut ouvert: Thu Nov 11 18:27:32 1999
Fichier: strings.cpp
Nous avons la permission d'écrire
La lettre du disque: E
La dimension en octets: 1473
La dernière fois qu'il fut ouvert: Tue Nov 16 19:06:10 1999
```

Les trois premiers fichiers d'en-tête, `<sys/types.h>`, `<sys/stat.h>` et `<dirent.h>`, nous indiquent que nous utilisons des fonctions C. Cela signifie que ce programme pourrait rencontrer des difficultés en cours de compilation ou d'utilisation sur une autre plate-forme, ou bien

en utilisant, sous Windows, des outils tels que C++ Builder de Borland ou Visual C++ de Microsoft. Certaines adaptations pourront donc être nécessaires.

La fonction C `opendir()` nous permet d'accéder au répertoire courant qui est spécifié avec la notation `"."`. Nous aurions pu donner le chemin complet, en respectant la notation `/` ou `\` suivant le système d'exploitation. `opendir()` est défini dans `dirent.h` sur le répertoire `i386-mingw32\include` du compilateur et retourne un pointeur à une structure `DIR` définie dans ce même fichier d'en-tête. Nous conseillerons, comme ici, de vérifier les cas d'erreur.

La partie intéressante commence avec `readdir()`, qui nous permet d'obtenir un premier niveau d'information d'un fichier dans la structure `dirent`. Chaque fois que nous rappelons `readdir()`, jusqu'à l'obtention du `NULL`, nous recevons l'information du prochain fichier. Ici, nous n'utilisons que le nom du fichier obtenu avec `d_name`, car les autres données de la structure `pdirent` ne nous apporteraient rien. Avec celui-ci, nous pouvons utiliser une autre fonction C, beaucoup plus intéressante, `stat()` :

```
stat(pdirent->d_name, &statbuf)
```

Il faut être attentif à la manière de passer l'adresse de `statbuf` à la fonction `stat()`. Enfin, nous utilisons un certain nombre de données disponibles dans la structure `stat` pour nous retourner l'information telle qu'elle nous apparaît à la console. Il faudra se méfier de certains de ces attributs, comme `st_dev`, qui peuvent avoir une autre signification sur un autre système d'exploitation.

Le fichier d'en-tête `stat.h`, dans le répertoire `386-mingw32\include\sys`, doit absolument être consulté pour vérifier les définitions utilisées. `S_IFDIR`, par exemple, nous permettrait d'exécuter une nouvelle recherche récursive sur les sous-répertoires, ce que nous ferons d'ailleurs en exercice.

Lecture d'un répertoire en Java

Nous allons à présent faire le même travail en Java, c'est-à-dire obtenir la liste de tous les fichiers d'un répertoire. Nous verrons que la création d'une liste de fichiers est nettement plus élégante et flexible. Voici donc le code que nous allons commenter :

```
import java.io.*;
import java.util.Date;

class MonFileFilter implements FilenameFilter {
    public boolean accept(File dir, String name) {
        String tempname = name.toLowerCase();
        return (tempname.endsWith ("doc") || tempname.endsWith ("txt"));
    }
}

public class ListDir {
  private String repertoire;

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

```
    if ( args.length != 0 ) {
      ListDir dr2 = new ListDir(args[0]);
    }
    else {
      ListDir dr2 = new ListDir("E:\\Mes Documents");
    }
  }

  public ListDir(String leRepertoire) {
    repertoire = leRepertoire;

    if (leRepertoire.charAt(leRepertoire.length() - 1) !=  File.separatorChar) {
      repertoire += File.separatorChar;
    }
    File monRep = new File(repertoire);

    if (monRep.exists()) {
      String lesFichiers[] = monRep.list(new MonFileFilter());
      int nombre = lesFichiers.length;
      System.out.println("Nombre de fichiers: " + lesFichiers.length);

      for (int i = 0; i < nombre; i++) {
        File leFichier = new File(repertoire + lesFichiers[i]);
        Date dernModif = new Date(leFichier.lastModified());
        System.out.println(lesFichiers[i] + " " + dernModif);
      }
    }
    else {
      System.out.println("Le répertoire " + repertoire + " n'existe pas");
    }
  }
}
```

Le résultat apparaîtra alors ainsi :

```
Nombre de fichiers: 3
document1.doc Sun Apr 25 11:23:08 GMT+02:00 1999
document2.doc Wed May 26 21:21:56 GMT+02:00 1999
readme.txt Sun Jun 06 18:28:04 GMT+02:00 1999
```

Dans cette version Java, au contraire de la version C++, aucune discussion de portabilité n'est à prévoir : le fonctionnement de ce code est garanti sur toutes les plates-formes. Comme ce code est de plus bien structuré, dans une classe, il nous donne tout de suite une meilleure impression.

Le `main()` est traditionnel ; il contient un répertoire par défaut, `E:\\Mes Documents`, si aucun n'est spécifié comme argument à l'exécution du programme. L'utilisation du `File.separatorChar` est essentielle si nous voulons que le programme soit portable, car il vérifiera et ajoutera le caractère `\` ou `/` si nécessaire.

Lire la liste des fichiers se fait à l'aide de la classe `File` et la méthode `list()`. Cette dernière reçoit un paramètre tout à fait particulier, et d'une puissance extraordinaire, un objet d'une classe héritée de `FilenameFilter`. Nous allons très vite comprendre l'utilisation de ce filtre. En effet, la méthode `accept()`, de la classe `MonFileFilter`, qui hérite de `FilenameFilter`, va être exécutée à l'appel de `monRep.list()`. Après une conversion locale du nom du fichier en minuscules pour accepter tous les `.txt`, `.Txt`, `.doc` ou autres `.DoC`, nous retournons un booléen au cas où le fichier posséderait une extension `.txt` ou `.doc`.

Au retour de `monRep.list()`, la variable `lesFichiers` contiendra un tableau avec tous les fichiers désirés. Nous passerons au travers du tableau `lesFichiers`, non sans avoir consulté au passage la documentation de la classe `File` ; celle-ci nous donnera l'usage de méthode `lastModified()`, qui nous retourne un `long`, valeur que la classe `Date` va accepter comme paramètre de constructeur.

Nous devrons nous demander comment notre programme fonctionne, puisque la méthode `exists()` de la classe `File` teste si le fichier existe alors que nous désirons contrôler l'existence du répertoire. C'est en effet correct, car nous avons toujours un `File.separatorChar` en fin de fichier.

Les flux en mémoire (C++)

Les streams en C++ ont été utilisés jusqu'à présent pour connecter des entrées et sorties sur des fichiers. Nous allons maintenant continuer notre tour d'horizon et parler de la fonction `sprintf()` de la bibliothèque C et des classes `istringstream` et `ostringstream`.

Afin de ne pas prendre les unes pour les autres, nous commencerons donc par définir deux catégories pour ces classes :

- la classe `istringstream` pour extraire des caractères ;
- la classe `ostringstream` pour composer des caractères.

Avant de passer à la description et à l'utilisation de ces deux classes, il est essentiel de revenir en arrière, dans le temps, et de se demander comment les programmeurs C se débrouillaient avant.

sprintf() de la bibliothèque C

Un programmeur C qui utilise régulièrement les fonctions C telles qu'`atoi()` (conversion chaîne de caractères à un `int`) ou `atof()` (conversion à un `float`) pour extraire des caractères ou `sprintf()` pour composer de nouvelles chaînes reconnaîtra rapidement ce type de code :

```
// sprintf.cpp
#include <iostream>
#include <cstdio>
#include <cstdlib>

using namespace std;
int const protect = 10;
```

```
int main()
{
  unsigned char protection1[protect];
  char tampon[25];
  unsigned char protection2[protect];
  int i;
  for (i = 0; i < protect; i++) {
    protection1[i] = 255;
    protection2[i] = 255;
  }
  sprintf(tampon, "Bonjour monsieur %s Salut %d\n",
                  "ABCDEFG", atoi("1234"));
  cout << tampon;
  cout << "12345678901234567890123456789012345" << endl;

  for (i = 0; i < protect; i++) {
    if (protection1[i] != 255) {
      cout << "Erreur dans protection1 à l'index " << i << "("
           << protection1[i] << ")" << endl;
    }
    if (protection2[i] != 255) {
      cout << "Erreur dans protection2 à l'index " << i << "("
           << protection1[i] << ")" << endl;
    }
  }
}
```

et son résultat :

```
Bonjour monsieur ABCDEFG Salut 1234
12345678901234567890123456789012345
Erreur dans protection1 à l'index 0(2)
Erreur dans protection1 à l'index 1(3)
Erreur dans protection1 à l'index 2(4)
Erreur dans protection1 à l'index 3(
)
Erreur dans protection1 à l'index 4(
```

Ce morceau de code pourrait sembler vraiment particulier à première vue, mais il n'est ici que pour montrer les problèmes potentiels de fonctions C comme `sprintf()`. Les fichiers d'en-tête `cstdio` et `cstdlib` sont nécessaires pour disposer des fonctions C que sont `sprintf()` et `atoi()`.

Nous avons encastré la zone `tampon` entre deux blocs de chaîne de caractères, `protection1` et `protection2`, qui sont initialisés avec des octets de bits à 1 (255) afin de vérifier si le programme est écrit dans ces zones. C'est effectivement le cas, car le 25 pour la dimension de notre `tampon` est beaucoup trop petit. Le résultat nous montre clairement les caractères `234` bien visibles et qui ont débordé. Ce n'est pas très important de comprendre plus en détail le positionnement exact car il est évidemment dépendant de la machine et du compilateur. La corruption pourrait être beaucoup plus vicieuse, comme créer des erreurs à d'autres endroits de la mémoire, du système ou du programme et ne pas se reproduire à

chaque exécution ! La solution parfaite à ce problème serait en fait de défendre les tampons `protection1` et `protection2` avec des moyens hardware, afin de protéger la mémoire contre l'écriture et de générer des exceptions par le logiciel.

L'`atoi("1234")` n'est ici que pour montrer son utilisation, la chaîne de caractères `1234` étant convertie en un entier qui sera repris par le `%d` du `sprintf()`. Pour les détails de `sprintf()`, il faudrait consulter la documentation. Nous retiendrons cependant la première partie, `tampon`, qui recevra le résultat du formatage. Dans la deuxième partie, nous avons des parties fixes et variables, `"Bonjour monsieur %s Salut %d\n"`. Comme nous avons deux parties variables, `%s` (pour une chaîne de caractères) et `%d` (pour un nombre décimal), nous obtiendrons aussi deux parties, qui doivent absolument apparaître, les `ABCDEFG` et `atoi("1234")`.

Les fonctions C `sprintf()` et son équivalent `printf()` pour la sortie à l'écran possèdent de nombreuses fonctions de formatage comme celle-ci :

```
sprintf(tampon,"AA%10.4fBB", 1.2);   // résultat : AA    1.2000BB
```

Le `%10.4f` nous indique un nombre avec dix positions et quatre chiffres après la virgule. Nous comprenons donc les espaces après les deux premiers `AA`. Cependant, elles peuvent être avantageusement remplacées par des fonctionnalités équivalentes, qui sont à disposition dans les classes `strstream` ou `stringstream`.

istringstream et ostringstream

Pour illustrer l'utilisation de ces deux classes, nous allons présenter un exemple simplifié qui comporte quelques variantes concrètes et utilisables sans grandes difficultés :

```
// FluxMem.cpp
#include <iostream>
#include <sstream>
#include <string>

using namespace std;

int main(int argc, char **argv) {
    // Lit une ligne de la console :
    cout << "Entrer deux chiffres en HEXA (ex: 12 A0): ";
    string input;
    getline(cin, input);

    // Décode l'entrée
    istringstream is(input);
    int c1;
    int c2;

    is.setf(ios::hex, ios::basefield); // on travaille en hexadécimal
    is >> c1 >> c2;
    cout << "test1: " << c1 <<endl;
    cout << "test2: " << c2 <<endl;
```

```
    int resultat = c1 * c2;

    // Affiche le résultat
    ostringstream os;
    os << "Le resultat est " << resultat << endl;

    cout << os.str();

    // Empêche la fenêtre DOS de se fermer
    cout << "Entrer retour";
    getline(cin, input);
    return 0;
}
```

Après avoir compilé `FluxMem.cpp` avec g++ ou le `make` dans l'éditeur Crimson, il faudra l'exécuter sans capture (voir annexe C) ou alors avec `FluxMem.exe` depuis l'explorateur de Windows et dans le répertoire `C:\JavaCpp\EXEMPLES\Chap09`.

Figure 9-2
Exécution de FluxMem.exe

L'instruction :

```
is >> c1 >> c2;
```

fonctionne, car nous avons un espace entre les deux chiffres et l'`istringstream` possède par défaut l'espace comme séparateur pour l'opérateur >>.

Les instructions suivantes sont pratiques :

```
cout << "test1: " << c1 <<endl;
cout << "test2: " << c2 <<endl;
```

dans le cas où nous n'avons pas de débogueur pour faire du pas à pas (voir annexe E, section « Présentation de NetBeans »). Toute la partie du `ostringstream` fait davantage partie ici d'un gadget, car le résultat aurait pu être présenté plus simplement : nous envoyons tout dans le stream `os` avant de le récupérer avec `os.str()`.

L'exemple ci-dessus pourrait être étendu à l'infini (avec ou sans `sstream`), par exemple pour écrire un petit calculateur !

Un exemple complet avec divers formatages

Après notre exemple simplifié à l'extrême, nous allons présenter quelques fonctions de formatage qui se trouvent à disposition dans la bibliothèque des `iostream` et `sstream`. Au fil des exemples suivants, nous allons découvrir toute une série d'options de formatage possibles.

Dans ce long exemple, nous avons encore gardé notre mécanisme de protection, seulement pour nous convaincre qu'il faut oublier nos anciennes habitudes de programmeur C. Il faut noter ici que ce code est extrêmement délicat et qu'il faudrait utiliser des outils comme PurifyPlus (Rational Software) pour vérifier si tous les tampons sont correctement initialisés et qu'il n'y a pas de fuite de mémoire.

```
// strstr.cpp
#include <iostream>
#include <sstream>
#include <string>

using namespace std;

int main(int argc, char* argv[])
{
  ostringstream sortie1;
  sortie1 << "Bonjour Monsieur ";
  sortie1 << "1234567890abcdefg" << ends;
  cout << "Test1: " << sortie1.str() << endl;

  ostringstream sortie3;
  sortie3 << "12345" << ends;
  sortie3 << "67890" << ends;
  cout << "Test2: " << sortie3.str() << endl;

  stringstream entree_sortie1;
  entree_sortie1 << "abcd\n1234" << ends;

  cout << "Test3: " << entree_sortie1.str() << endl;

  stringstream entree_sortie2;
  int nombre1 = 10;
  int nombre2 = 20;
  string un_string1 = "Bonjour";
  entree_sortie2 << nombre1 << "." << nombre2 << un_string1 << ends;
  cout << "Test4: " << entree_sortie2.str() << endl;

  double double1;
  entree_sortie2 >> double1;
  cout << "Test5: " << double1 << endl;

  string un_string2;
  entree_sortie2 >> un_string2;
  cout << "Test6: " << un_string2 << endl;
```

```
  ostringstream sortie4;
  sortie4.fill('0');
  sortie4.setf(ios::right, ios::adjustfield);
  sortie4.width(10);
  double double2 = 1.234;
  sortie4 << double2;
  sortie4 << "|";
  sortie4.setf(ios::left, ios::adjustfield);
  sortie4.width(8);
  sortie4.precision(3);
  sortie4 << double2;
  sortie4 << "|";
  int nombre5 = 31;
  sortie4 << hex << nombre5 << ":" << oct << nombre5 << ends;

  cout << "Test7: " << "[" << sortie4.str() << "]" << endl;

  return 0;
}
```

Ce code nous donnera le résultat suivant sans entrer de valeurs comme arguments :

```
Test1: Bonjour Monsieur 1234567890abcdefg
Test2: 12345
Test3: abcd
1234
Test4: 10.20Bonjour
Test5: 10.2
Test6: Bonjour
Test7: [000001.234|1.230000|1f:37
```

Le `Test1` utilise un `ostringstream` dynamique. La mémoire sera allouée automatiquement, et `ends` terminera l'entrée sur le flux. La méthode `str()` va nous retourner la chaîne de caractères.

Le `Test2` est très similaire au `Test1`, mais le premier `ends` va fermer le flux : le `67890` est perdu.

Dans le cas des `Test3`, `Test4`, `Test5` et `Test6`, il est intéressant de noter l'utilisation de `stringstream` au lieu de `ostringstream`.

Remarquons que notre chaîne de caractères `"abcd\n1234"` est sur deux lignes, ce qui explique le résultat du `Test3` sur deux lignes. Dans `entree_sortie1`, nous pourrions avoir un fichier texte complet.

Dans le `Test4`, nous montrons ce qui a été effectivement stocké dans `entree_sortie2`.

Dans les `Test5` et `Test6`, nous faisons l'extraction dans l'autre sens. Nous voyons que nous trichons en quelque sorte, puisque nous traitons un `double` et un `String` : `double1` et `un_string2`. Si nous avions un autre format ou une entrée totalement aléatoire et confuse, nous devrions traiter les erreurs correctement.

Le Test7, qui est en effet un concentré d'options de formatage, nous sortira le résultat suivant, composé de trois morceaux :

```
[000001.234|1.230000|1f:37]
```

Les deux premières parties sont composées du nombre 1.234, qui est une fois aligné à droite avec setf(ios::right, ios::adjustfield) et l'autre fois à gauche. Nous contrôlons le remplissage, à gauche ou à droite, grâce à fill('0'), mais le caractère 0 pourrait très bien être remplacé par autre chose. La largeur des champs, qui vaut 10 et 8, est modifiable avant l'opérateur << par la méthode width(). Le deuxième nombre a-t-il perdu des chiffres ? Non, car 1.23 représente une précision de trois chiffres, indépendamment du nombre de chiffres après la virgule. Enfin, le 1f:37 représente la nombre décimal 33 dans les formats hexadécimal et octal.

Le printf Java du JDK 1.5

La fonction printf() est une fonction bien connue des programmeurs C. Nous avons écrit un petit morceau de code en C :

```
// printfC.c          //.c et non .cpp
#include <stdio.h>   //du C pas du C++

main() {
  int  nombre = 17;
  char *text  = "hexa";

  printf("Le nombre %d est %x en %s\n", nombre, nombre, text);
}
```

Le résultat nous donne :

```
Le nombre 17 est 11 en hexa
```

Les %d, %x et %s (décimal, hexadécimal et string) vont prendre et convertir les données des trois variables qui suivent la partie "…" !

Dans le Makefile de ce chapitre, nous avons ajouté une entrée c et un appel au compilateur C de cette manière :

```
gcc -o printfC.exe printfC.c
```

En revanche, en éditant avec Crimson le fichier printF.c, nous pourrons tout aussi bien le compiler avec le compilateur C++, c'est-à-dire g++ dans le menu.

Depuis la version 1.5 de Java, nous pouvons écrire le même programme qui donnera le même résultat :

```
import java.io.*;

class Printf  {
  public static void main (String args[]) {
    int   nombre = 17;
```

```
    String text  = "hexa";

    System.out.printf("Le nombre %d est %x en %s\n", nombre, nombre, text);
  }
}
```

istrstream et ostrstream

Si le lecteur rencontre ces anciennes formes dépréciées (éditions précédentes de cet ouvrage), il devra les convertir avec les `stringstream`. Lors de la compilation de ces anciennes classes avec g++ (version récente), les messages d'erreur nous aiderons dans ce travail. Si le langage C++ avait été développé dès le départ avec une bibliothèque standard et une classe `String` comme en Java, nous n'aurions peut-être jamais inventé les `strstream` ni entendu parler d'eux. Nous allons maintenant présenter d'autres exemples variés que nous devrions rencontrer dans la programmation de tous les jours.

Formatage en Java

Passons à présent au formatage et à la conversion en Java, car nous nous trouvons dans le même contexte : c'est en effet l'une des premières difficultés rencontrées par le programmeur. Nous allons reprendre ici un certain nombre de formes déjà utilisées, que nous utilisons par exemple pour la conversion des arguments reçus par le `main()`. C'est une sorte de synthèse à titre comparatif puisqu'en C++ les flux seront utilisés en général.

La méthode `println()` de la classe `System.out` appliquée à l'opération suivante :

```
int valeur1 = 8;
double valeur2 = 1423.35 * 0.033;
System.out.println(valeur1 + ":" + valeur2);
```

et qui nous sortira le résultat suivant :

```
8:46.970549999999996
```

ne nous donnera certainement pas satisfaction. Ce résultat ne correspondra sans doute pas à celui désiré, avec uniquement deux décimales imprimées, des 0 ou des espaces à gauche et un alignement précis du style :

```
00008:00046.97
```

Ce format peut être obtenu, d'une manière extrêmement simple, grâce au code suivant :

```
DecimalFormat df1 = new DecimalFormat("00000");
DecimalFormat df2 = new DecimalFormat("00000.00");
System.out.println(df1.format(valeur1) + ":" + df2.format(valeur2));
```

La classe `DecimalFormat`, d'une simplicité déconcertante, nous permet d'obtenir le résultat désiré. Les 0 indiquent la position des chiffres, ces derniers pouvant prendre toutes les valeurs, y compris 0. Il faut être très attentif aux résultats dans les cas où nous dépasserions

les limites indiquées. Il faut vraiment vérifier si cela correspond à notre spécification, car nous pourrions être surpris :

```
System.out.println(df2.format(7654321.999));
```

Dans ce cas, nous sommes en mesure de nous poser deux questions :

1. Que se passe-t-il avec le .999 car nous avons choisi deux décimales ?
2. Notre maximum spécifié dans `df2` est en principe 99999.99. Que va-t-il se passer ?

Le résultat ne nous surprendra qu'à moitié :

```
7654322.00
```

Filtrer du texte en Java avec StringTokenizer et StreamTokenizer

Un *token*, en anglais, signifie une marque. En effet, lorsque nous voulons analyser, par exemple, une chaîne de caractères, nous pourrions utiliser une ou plusieurs marques pour identifier, filtrer ou chercher des mots ou des parties de mots. L'une des premières applications est l'extraction de données. Il est souvent plus facile d'utiliser `StringTokenizer` au lieu de se lancer dans du code complexe utilisant les méthodes de la classe `String`, ou dans une recherche caractère par caractère.

Outre la classe `java.util.StringTokenizer`, qui utilise comme entrée un `String`, la bibliothèque Java possède une autre classe, le `java.io.StreamTokenizer`, qui a lui besoin d'un `InputStream` comme entrée. Cet `InputStream` peut représenter un fichier ou encore correspondre à une entrée de données directement sur la console. C'est ce cas-là que nous avons choisi dans l'exemple qui suit. Nous commencerons par trois exemples de filtres avec la classe `StringTokenizer` :

```
import java.util.StringTokenizer;
import java.io.*;

class MonTokenizer {
  public static void main(String[] args) {
    StringTokenizer st = new StringTokenizer("Nous faisons un premier test");
    while (st.hasMoreTokens()) {
      System.out.println(st.nextToken());
    }
    st = new StringTokenizer("Nous\tfaisons  un\r \n \r deuxième \ntest");
    while (st.hasMoreTokens()) {
      System.out.println(st.nextToken());
    }
    st = new StringTokenizer("Ma;base;de;donnée", ";");
    while (st.hasMoreTokens()) {
      System.out.println(st.nextToken());
    }
    Reader lecteur = new BufferedReader(new InputStreamReader(System.in));
```

```
    StreamTokenizer streamt = new StreamTokenizer(lecteur);
    try {
      while (streamt.nextToken() != StreamTokenizer.TT_EOF) {
        switch (streamt.ttype) {
          case StreamTokenizer.TT_NUMBER:
             System.out.println("Un nombre: " + streamt.nval);
             break;
          case StreamTokenizer.TT_WORD:
             System.out.println("Un mot: " + streamt.sval);
             break;
          default:
             System.out.println("Un autre type: " + (char)streamt.ttype);
             break;
        }
      }
    }
    catch (IOException e) {
       System.out.println("IOException: " + e);
    }
  }
}
```

Les deux premiers constructeurs de `StringTokenizer` ne reçoivent pas de second paramètre : cela signifie que le défaut sera utilisé. Celui-ci correspond à l'espace, au tabulateur et aux nouvelles lignes (PC et Linux inclus). Le résultat de cette partie :

```
Nous
faisons
un
premier
test
Nous
faisons
un
deuxième
test
```

nous montre que les espaces et autres caractères de séparations multiples sont bien traités. Pour le deuxième objet, si nous avions utilisé la forme :

```
st = new StringTokenizer("Nous\tfaisons  un\r \n \r deuxième \ntest", "x\n\r\t ");
```

nous aurions obtenu le même résultat, sauf que le mot `deuxième` aurait été coupé en deux en perdant le `x` puisque nous l'aurions traité comme une marque (*token*).

Le troisième objet de `StringTokenizer` utilise cette fois-ci une séparation avec le caractère `;`. C'est le cas classique d'une base de données :

```
Ma
base
de
données
```

qui pourrait être par exemple exportée depuis Microsoft Access. Nous noterons que la classe `StringTokenizer` possède les méthodes `hasMoreTokens()` et `nextToken()`, qui nous permettent de nous promener de morceau à morceau.

Nous terminons notre exemple avec le `StreamTokenizer`, dont nous avons initialisé le `Reader` depuis la console (`System.in`). Au contraire de `StringTokenizer`, nous ne pouvons pas définir les caractères qui seront utilisés comme délimiteur. La classe `StreamTokenizer` va nous retourner le type du morceau qu'il a filtré. Après avoir réceptionné le prochain morceau avec `nextToken()`, nous vérifions son type avec `ttype`. Ici, nous avons vérifié `TT_NUMBER` et `TT_WORD` spécialement à titre d'exercice. Afin de ne pas toujours entrer les mêmes caractères, nous avons introduit le texte suivant dans le fichier `test.txt` :

```
abcd 1234 xyz
unTabSuit       aprèsLeTab
ligneVide_avant
```

et entré la ligne de commande :

```
type test.txt | java MonTokenizer
```

Nous avons obtenu pour la dernière partie du programme le résultat :

```
Un mot: abcd
Un nombre: 1234.0
Un mot: xyz
Un mot: unTabSuit
Un mot: aprèsLeTab
Un mot: ligneVide
Un autre type: _
Un mot: avant
```

avec le cas particulier du caractère _. C'est relativement puissant, mais encore très loin d'une « tokenisation » pour une chaîne de caractères telle que `"<a href="mailto:jean-bernard@boichat.ch">Email at home</a>"`.

Résumé

Nous savons à présent maîtriser la lecture et l'écriture de fichiers. Un grand nombre d'applications utilisent des données stockées sur le disque, les analysent, les transforment et peuvent aussi les sauvegarder à nouveau pour une utilisation future. Nous avons vu aussi comment former et filtrer des chaînes de caractères en mémoire, en utilisant des techniques et opérateurs similaires à la lecture et à l'écriture de fichiers sur le disque.

Exercices

Les quatre premiers exercices seront codés en Java et C++. Nous avons écrit un certain nombre d'exercices, mais avec la tentation d'en écrire beaucoup plus. Ce chapitre est non seulement l'un des plus importants, mais aussi l'un des plus intéressants en ce qui

concerne les possibilités d'écriture de petits programmes d'apprentissage. Un résultat sous forme de fichier est toujours plus concret et motivant, donnant au programmeur le sentiment d'avoir vraiment créé quelque chose.

1. Écrire un programme de copie de fichier binaire. Prendre l'exécutable compilé en C++ comme exemple et vérifier que sa copie est identique en l'exécutant ou en utilisant l'exercice 3.
2. Écrire un programme qui lit un fichier texte Linux (respectivement DOS) et le convertit en fichier texte DOS (respectivement Linux).
3. Écrire un programme de comparaison de deux fichiers binaires. Si les fichiers sont différents, donner l'index et la valeur du premier octet différent.
4. Même exercice que le 3, mais pour un fichier texte avec le numéro et la position de la première ligne différente.
5. Sauver en C++ une partie d'Othello en format texte simple, comme nous l'avons fait précédemment en Java dans la classe `WriteOthello`.
6. À partir du programme `listdir.cpp` en C++ et de la classe `ListDir` en Java, écrire dans les deux langages une classe `ListeRepertoires` qui va pouvoir présenter une liste de tous les fichiers dans le répertoire sélectionné et ses sous-répertoires. Il s'agit d'un exercice d'une certaine complexité qui pourrait même être étendu ou réutilisé pour l'écriture d'outils de maintenance ou de statistique.

10

Variations sur un thème de classe

Le constructeur par défaut en C++

Lorsque nous créons une instance d'une classe, nous utilisons le code disponible dans le constructeur. Ce dernier peut être surchargé. Si nous définissons une classe `Personne`, nous pourrions imaginer créer des instances de cette classe de ces trois manières :

```
Personne personne1("Nom1", "Prénom1");
Personne personne2("Nom2");
Personne personne3(15286);
```

dont il faudrait définir le code suivant dans la définition de la classe :

```
Personne(const char *pnom, const char *pprenom);
Personne(const char *pnom);
Personne(int numero);
```

Au lieu d'un `const char *pnom`, nous pourrions aussi utiliser un `const string` ou encore un `const &string`. Le `numero` pourrait être passé différemment, comme un `string`. Pour l'instant, dans ce contexte particulier, ce n'est pas très important.

Ces trois formes nous indiquent qu'il y a au moins trois constructeurs surchargés et que notre classe doit pouvoir conserver le nom, le prénom et un code d'identification. Si le premier constructeur peut nous sembler suffisant, les autres formes doivent nous rendre attentifs. Ces dernières nous indiquent que, probablement, le prénom peut être omis et qu'un numéro d'employé ou autre code d'identification devrait permettre de rechercher les données avec un autre mécanisme. Cependant, aucune information ne nous est donnée : par exemple, le prénom doit-il être spécifié plus tard au travers d'une méthode,

et comment sera utilisé ce fameux numéro ? Si quelqu'un venait à nous dire que, dans les trois cas, une base de données sera consultée, nous pourrions alors imaginer les cas suivants :

- Pour le constructeur (`"Nom1", "Prénom1"`), une nouvelle entrée dans la base de données sera créée avec une identification qui correspond à ce fameux numéro. Sinon les données existantes de la base de données seront attribuées aux différents attributs de la classe `Personne`.
- Pour le cas (`"Nom2"`), le constructeur n'acceptera la requête que si le prénom est unique pour ce nom dans la base de données. Si nous n'avons pas encore d'entrée, la requête sera rejetée.
- La dernière forme du constructeur sera aussi acceptée si le numéro est déjà alloué.

Il y a bien sûr d'autres cas de figure tout aussi acceptables. Un aspect essentiel à toutes ces considérations est le traitement des erreurs. Il faudra prévoir soit une exception générée dans le constructeur et capturée par l'application, soit un code d'erreur vérifié par toutes les méthodes qui pourraient accéder à ces objets, car ces derniers ont pu ne pas être initialisés correctement.

Enfin, nous devons nous poser cette question essentielle : que se passerait-t-il si nous écrivions ceci :

```
Personne personne4;
```

Cette forme ne sera acceptée que si aucun constructeur n'est défini ! Dans le cas précis de nos trois exemples de constructeur de la classe `Personne`, il est inutile, en principe, de prévoir un constructeur par défaut. Cette dernière ne doit pas être confondue avec des constructions telles que :

```
Personne(0);
Personne("");
```

Dans la première construction, le pointeur nul serait le 0, et, dans la seconde, le nom vide de la personne. Ce sont des cas qui ne devraient jamais se produire ou sinon qui devraient être à contrôler, explicitement, dans le code du constructeur affecté.

Nous pouvons aussi éviter l'utilisation d'un constructeur par défaut, en utilisant cette forme :

```
class Personne {
  private:
    Personne() {
    }
};

int main() {
  Personne p1;
}
```

Si nous essayions de compiler ce code, nous obtiendrions :

```
personne.cpp: In function `int main()':
personne.cpp:3: `Personne::Personne()' is private
personne.cpp:8: within this context
```

Le programmeur comprendra qu'il y a un problème et devra adapter son code. La forme suivante peut nous entraîner des complications :

```
Personne personne[100];
```

Si nous écrivions :

```
class Personne {
  public:
    Personne(const char *pnom, const char *prenom) {}
};

int main() {
  Personne personnes[100];
}
```

nous aurions une erreur de compilation :

```
personne.cpp: In function `int main()':
personne.cpp:8: no matching function for call to `Personne::Personne ()'
personne.cpp:3: candidates are:
Personne::Personne(const char *, const char *)
personne.cpp:4: Personne::Personne(const Personne &)
```

Cela nous indique que le constructeur par défaut, `Personne() {}`, doit absolument être codé. C'est ce que nous ferons comme exercice.

Il existe également une autre construction dont il faut se méfier :

```
Personne(int numero = 0) {}
```

Il s'agit de la troisième forme de constructeur que nous avions définie en début de chapitre, mais avec un paramètre par défaut. Dans ce cas, si nous avions :

```
class Personne {
  public:
    Personne(int numero = 0) {}
};

int main() {
  Personne personnes[100];
}
```

les 100 objets auraient alors le même numéro, et ce n'est certainement pas ce que nous désirons. Pour la forme :

```
Personne(const char *pnom, const char *pprenom = 0);
```

ce serait différent. Cela pourrait signifier que nous ne connaissons que le nom de la personne et pas encore son prénom. L'instruction :

```
Personne personne("Haddock");
```

serait acceptée. La forme :

```
Personne(const char *pnom);
```

pourrait alors être omise. Enfin, il nous faut parler d'une autre façon de programmer, lorsque plusieurs constructeurs sont nécessaires et que le code doit être répété. Cela consiste à écrire une méthode, privée en général, qui contiendra la partie commune, principalement lorsqu'elle renferme du code relativement long. Cette manière de faire est aussi applicable en Java, sans l'utilisation de paramètres par défaut :

```
class Personne {
  private:
    code_commun(.....) {
      // ......
    }

  public:
    Personne(......) {  // constructeur 1
      // code de préparation
      code_commun(.....);
    }

    Personne(..........) {  // constructeur 2
      // code de préparation
      code_commun(.....);
    }
};
```

Nous répéterons encore une fois qu'il faut limiter l'allocation de ressources dans les constructeurs, car elle pourrait nécessiter un traitement d'erreurs, si bien que le constructeur ne retournerait jamais de résultat directement.

Nous remarquons en fait que le langage C++ met à notre disposition toute une série de techniques pour résoudre nos problèmes. Nous constatons ici qu'il est en fait plus facile de maîtriser ces techniques que de définir une conception correcte et propre, dans le cas présent, de notre classe `Personne`. Nous avons vraiment le sentiment d'un flou parfait et que tout reste à faire. Une analyse plus profonde des fonctionnalités de notre classe est absolument nécessaire. Savons-nous vraiment ce que nous voulons ? Sommes-nous certains que cela correspondra aux exigences du produit que nous voulons distribuer à nos clients ?

Définir le ou les constructeurs d'une classe, ainsi que l'initialisation correcte des variables privées, c'est-à-dire des attributs de cet objet, est donc essentiel dans la conception de logiciels programmés en Java et C++.

Le constructeur de copie en C++

Après le constructeur par défaut, il nous faut considérer le constructeur de copie. Si nous écrivions :

```
Personne personne1("Nom1", "Prénom1");
Personne personne2(personne1);
```

le nouvel objet `personne2` devrait être créé avec les mêmes attributs que le premier. Nous allons voir à présent comment traiter correctement ce problème, bien que nous puissions nous poser la question de l'utilité de produire une deuxième copie d'un objet de la classe `Personne`. Ce serait différent dans le cas d'une classe représentant une partie du jeu d'échecs, dans laquelle nous pourrions essayer une combinaison particulière ou certaines variations sur la partie courante, sans toucher à l'original.

Le constructeur de copie par défaut

Dans l'exemple qui suit, nous avons ajouté un attribut `annee_naissance` à une classe nommée `PersonneA`. Cet attribut ne sera affecté que par une méthode séparée, en dehors du constructeur. Nous allons ensuite créer une seconde instance de cette classe `PersonneA`, en utilisant un constructeur de copie :

```
// personnea.cpp
#include <iostream>
#include <string>

using namespace std;

class PersonneA {
  private:
    string nom;
    string prenom;
    int annee_naissance;

  public:
    PersonneA(const char *plenom, const char *pleprenom) {
      nom = plenom;
      prenom = pleprenom;
      annee_naissance = -1;
    }

    void set_annee(int lannee) {
      annee_naissance = lannee;
    }

    void test() {
      cout << nom << " " << prenom << " : " << annee_naissance << endl;
    }
};
```

```
int main() {
  PersonneA personne1("LeRoi", "Louis");
  personne1.set_annee(1978);
  personne1.test();

  PersonneA personne2(personne1);
  personne2.test();
  return 0;
}
```

Si nous exécutons ce programme, nous obtenons :

```
LeRoi Louis : 1978
LeRoi Louis : 1978
```

Le constructeur de copie par défaut sera généré par le compilateur et exécuté, puisque nous ne l'avons pas codé, comme nous le verrons plus loin. Le résultat de l'instruction :

```
PersonneA personne2(personne1);
```

sera une copie de toutes les valeurs des attributs dans le nouvel objet. Nous aurons donc deux objets différents, car `personne2` est non pas un pointeur, mais bien un objet séparé dont les caractéristiques sont les mêmes. Un père donne parfois son prénom à son fils, mais l'année de naissance ne peut être identique dans ce cas précis.

Ce qui est important ici, c'est l'instruction :

```
annee_naissance = -1;
```

dans le constructeur général. Avec `-1`, nous indiquons par exemple que l'année de naissance n'a pas été spécifiée, car nous pourrions avoir un autre constructeur avec ce paramètre, ou que cette donnée devra être indiquée plus tard, au travers d'une méthode comme `set_annee()`. Dans le cas d'un constructeur de copie, la valeur `-1` pour l'année de naissance est tout à fait justifiée. Nous pourrions imaginer d'autres variantes, et nous le verrons en exercice, après la présentation de la forme de ce constructeur.

La méthode `set_annee()` pourrait être beaucoup plus complexe, comme accéder à une base de données ou calculer d'autres paramètres. Elle pourrait être définie comme suit :

```
bool set_annee(int lannee) { .... }
```

Le résultat de l'opération pourrait aussi être retourné, ce qui n'est pas le cas pour un constructeur. Utiliser des méthodes de ce type peut apporter une simplification au traitement des erreurs et aux contraintes de notre design.

La forme du constructeur de copie

Il est donc préférable et conseillé de toujours définir un constructeur de copie. Il aura, dans notre cas, la forme suivante :

```
inline Personne::Personne(const Personne &pers)
                :nom(pers.nom), prenom(pers.prenom) { };
```

Cette forme se rencontre généralement, car elle est simple et efficace. Dans ce cas, l'utilisation d'`inline` est correcte, mais ne serait pas nécessaire si nous avions défini le code dans le fichier d'en-tête (automatiquement `inline`).

Comme pour le constructeur par défaut, il est aussi tout à fait possible de fermer la porte à un constructeur de copie. Si nous voulons que le compilateur rejette l'instruction :

```
Personne personne2(personne1);
```

afin d'empêcher le programmeur d'utiliser cette forme pour une raison ou pour une autre, nous pourrions alors définir ceci :

```
private Personne::Personne(const Personne &pers);
```

qui provoquerait une erreur de compilation.

Nous reprenons à présent notre exercice précédent, la classe `PersonneA`, que nous avons analysée pour le constructeur par défaut et que nous allons adapter avec un constructeur de copie :

```
// personneb.cpp
#include <iostream>
#include <string>

using namespace std;

class PersonneB {
  private:
    string nom;
    string prenom;
    int annee_naissance;

  public:
    PersonneB(const char *plenom, const char *pleprenom) {
      nom = plenom;
      prenom = pleprenom;
      annee_naissance = -1;
    }

    PersonneB(const PersonneB &une_autre_personne) {
      nom = une_autre_personne.nom;
      prenom = une_autre_personne.prenom;
      annee_naissance = -1;
    }

    void set_annee(int lannee) {
      annee_naissance = lannee;
    }

    void test() {
      cout << nom << " " << prenom << " : " << annee_naissance << endl;
    }
```

```
};

int main() {
  PersonneB personne1("LeRoi", "Louis");
  personne1.set_annee(1978);
  personne1.test();

  PersonneB personne2(personne1);
  personne2.test();
  return 0;
}
```

Ici, chaque attribut est copié un à un. Nous voyons donc la différence comme pour :

```
nom = une_autre_personne.nom;
```

`une_autre_personne.nom` est bien un attribut privé de l'objet d'origine, mais notre code possède les droits d'accéder à des attributs de cette même classe. La différence réside dans notre instruction :

```
annee_naissance = -1;
```

qui indique que l'année de naissance n'a pas encore été spécifiée. Nous le voyons avec le résultat, qui nous satisfait cette fois-ci :

```
LeRoi Louis : 1978
LeRoi Louis : -1
```

C'est évidemment un choix de conception qui variera de cas en cas. Nous ne pourrions faire de même avec une copie du jeu d'échecs et un attribut qui nous indiquerait que le joueur est en position d'échec, c'est-à-dire qu'il n'aurait pas le droit à n'importe quel déplacement. Nous devons soigneusement analyser quels attributs doivent être copiés ou modifiés, comme le cas d'un compteur du nombre d'instance de cette classe.

Ne pas confondre constructeur et affectation

Pour terminer, il nous faut revenir sur les deux formes suivantes :

```
Personne personne2(personne1);
personne2 = personne1;
```

Elles représentent un constructeur et une affectation. Ces deux formes sont très souvent confondues et prises l'une pour l'autre, principalement pour le résultat attendu. Nous avons déjà analysé la première forme dans ce chapitre, alors que la deuxième, celle de l'affectation, sera traitée au début du chapitre suivant.

Nous espérons que le lecteur se sera rendu compte d'un problème sérieux. Comment peut-on utiliser un constructeur de copie pour cette classe `Personne` qui va affecter à deux objets différents le même nom et le même prénom ? Nous savons que ce cas est tout à fait vraisemblable et n'a rien à voir avec cette partie théorique et essentielle du langage. Notre classe `Personne` est très loin d'une élaboration complète et sans erreurs de conception.

Les outils et les constructions à disposition dans un langage nous permettent en fait de vérifier la conception de nos classes et de nos applications. Que des personnes puissent avoir le même nom et le même prénom a en fait une relation directe avec les contraintes et besoins de l'application ou du produit. Cependant, les ressources du langage peuvent nous aider à découvrir les faiblesses de la spécification du système et à corriger, à modifier ou à mieux préciser ces dernières.

Le constructeur par défaut en Java

Il y a quelques différences significatives quand on passe du C++ au Java. La complexité du C++ provient principalement du fait que les attributs de classes peuvent être de différents types. Nous avons vu par exemple la difficulté du constructeur de copie avec des attributs pointeurs. La première remarque en Java va porter sur la construction :

```
String unNom;
```

Si nous écrivons la classe suivante, pour utiliser la méthode `toLowerCase()`, qui consiste à convertir en minuscules un objet `String` qui n'a pas été initialisé :

```
class Test {
  public static void main (String[] args) {
    String unNom;

    unNom.toLowerCase();
  }
}
```

et que nous essayons de compiler ce programme, nous obtiendrons ceci :

```
Test.java:5: Variable unNom may not have been initialized.
  unNom.toLowerCase();
  ^
1 error
```

C'est propre et clair. La variable doit absolument être initialisée. Ce mécanisme de protection n'existe pas en C++, qui, de plus, aurait deux formes à disposition :

```
string un_nom;
string *pun_nom;
```

De ce point de vue, ces deux langages sont très différents ! Si nous écrivons à présent cette classe de test :

```
class Test {
  private String unNom;
  private boolean flag;
  private int nombre;

  public void unTest() {
    unNom.toLowerCase();
    if (flag == true);
```

```
    if (nombre == 1);
  }

  public static void main (String[] args) {
    Test test = new Test();
    test.unTest();
  }
}
```

nous y découvrons que les variables `unNom`, `flag` et `nombre` ne sont pas initialisées. Le compilateur ne peut le voir, et c'est seulement à l'exécution que nous découvrirons le problème :

```
Exception in thread "main" java.lang.NullPointerException
    at Test.unTest(Compiled Code)
    at Test.main(Compiled Code)
```

Cela est dû au fait qu'un constructeur par défaut sera exécuté selon les règles suivantes :

- les attributs numériques de la classe prendront la valeur `0` ;
- les booléens recevront la valeur `false` ;
- les variables objets seront initialisées à `null`.

`unNom` sera donc `null` et provoquera notre `java.lang.NullPointerException`.

Il y a plusieurs solutions à ce problème spécifique, comme :

```
if (unNom != null) {
   unNom.toLowerCase();
}
```

Nous dirons que cette solution ne nous plaît pas ! Nous prêcherions pour un meilleur design et l'écriture d'un constructeur. Pour la classe `Personne` et les attributs `nom` et `prenom`, il faudrait définir une règle. Un `String` vide pourrait indiquer que l'attribut n'est pas connu, pourrait ne jamais être identifié ou devrait absolument être attribué dans une phase ultérieure.

Nous pouvons revenir à notre classe `Test` et présenter cette fois-ci une implémentation sans constructeur par défaut :

```
class Test {
  private String unNom;
  private boolean flag;
  private int nombre;

  public Test() {
    unNom = "";
    flag = true;
    nombre = -1;
  }

  public void unTest() {
```

```
    unNom.toLowerCase();
    if (flag == true) {
       // quelque chose
    };
    if (nombre == 1){
       // quelque chose
    };
  }

  public static void main (String[] args) {
    Test test = new Test();
    test.unTest();
  }
}
```

Il serait évidemment bienvenu d'écrire dans le code une documentation complète pour la valeur de ces attributs, qui pourraient alors apparaître dans les documents HTML produits par `javadoc`, que nous avons utilisés au chapitre 4.

Le constructeur de copie en Java

Nous ne parlons pas de constructeur de copie par défaut en Java, car il n'existe pas. Dans l'exemple qui suit :

```
class ObjetCopiable {
  private String objet;
  private int valeur;

  public ObjetCopiable(String nom, int prix) {
    objet  = nom;
    valeur = prix;
    System.out.println(objet + " : " + valeur);
  }

  public ObjetCopiable(ObjetCopiable oc) {
    objet  = oc.objet;
    valeur = (int)(oc.valeur * 0.9);
    System.out.println(objet + " : " + valeur);
  }

  public static void main (String[] args) {
    ObjetCopiable co1 = new ObjetCopiable("Statuette", 1000);
    ObjetCopiable co2 = new ObjetCopiable(co1);
  }
}
```

Nous voyons qu'il nous faut définir un constructeur spécifique. `co2` sera bien un autre objet, une deuxième statuette à laquelle nous décidons d'attribuer une valeur diminuée de 10 %, puisqu'elle n'est plus unique. Dans la réalité, nous pourrions augmenter le prix de la deuxième si la première se vend rapidement ou encore diminuer fortement leurs valeurs

si elles restent invendues pendant une période trop longue. Le mot de la fin serait sans doute de devoir conserver le prix de revient de nos objets, en tenant compte de la location du magasin et de l'amortissement des étagères !

Les variables et méthodes statiques d'une classe

Nous avons déjà rencontré le mot-clé `static` en C++ pour les variables globales. Il nous faut à présent le considérer dans le contexte des classes Java et C++. Dans les deux langages, nous avons la même définition, bien que sa syntaxe diffère légèrement. Le mot-clé `static` désigne des variables ou des méthodes qui appartiennent à toutes les instances d'une classe déterminée.

Une variable statique pourra être modifiée par toutes les instances de sa classe, à condition qu'elle ne soit pas constante (`const` en C++ ou `final` en Java). Comme nous préférons garder les variables privées, nous pourrons accéder à une variable statique avec une méthode statique et publique. Une méthode statique ne peut accéder à des variables de classe traditionnelles, car une instance de cette classe doit être utilisée.

Nous aimons bien le terme d'attribut de classe pour désigner ces variables statiques. Ce terme vient du langage Smalltalk. Nous allons voir les détails et la syntaxe au travers de deux exemples en Java et C++.

Nous tirons un numéro

L'exemple que nous avons choisi va nous permettre de tirer un numéro entre 1 et 10 et de le réserver. Chaque instance de la classe aura donc un numéro unique entre 1 et 10. Ce numéro pourra être libéré lorsque l'instance disparaît. Si plus de dix instances sont créées, les nouvelles recevront le numéro 0, jusqu'au moment où l'une des dix réservées sera libérée. Nous ne pousserons pas l'exercice jusqu'à attribuer un numéro libéré au premier objet auquel le numéro 0 avait été attribué. Seuls les nouveaux venus seront éventuellement des veinards.

Comme nous demanderons régulièrement le nombre de numéros alloués, la méthode pour faire ce travail devra être rapide, sans toutefois rendre publics les attributs de la classe.

En C++

Nous passons immédiatement à la présentation du code pour notre problème, qui semble avoir été bien défini :

```
// tirenumero.cpp
#include <iostream>
#include <string>

using namespace std;

class TireNumero {
```

```
private:
  static int  nombre_sorti;
  static bool *ptirage;
  int quel_numero;

public:
  TireNumero() {
    if (ptirage == 0 ) { // premier usage
      ptirage = new bool[10];
      for (int i = 1; i < 10; i++) {
        *(ptirage + i) = false;
      }
      quel_numero = 1;
      *ptirage = true;
      nombre_sorti = 1;
      return;
    }
    if (nombre_sorti == 10) {
      quel_numero = 0;
    }
    else {
      nombre_sorti++;
      for (int i = 0; i < 10; i++) {
        if (*(ptirage + i) == false) {
          *(ptirage + i) = true;
          quel_numero = i + 1;
          break;
        }
      }
    }
  }

  ~TireNumero() {
    if (quel_numero != 0) {
      nombre_sorti--;
      *(ptirage + quel_numero - 1) = false;
      if (nombre_sorti == 0) {
        delete ptirage;
        ptirage = 0;
        cout << "Tous les numéros sont rentrés" << endl;
      }
    }
  }

  int get_numero() {
    return quel_numero;
  }

  static int get_nombre_sorti() {
    return nombre_sorti;
  }
```

```
    static void message1() {
      cout << "Nombre sorti: " << get_nombre_sorti() << endl;
    }

    void message2() {
      cout << "Nombre actuel: " << get_numero() << endl;
    }
};
int  TireNumero::nombre_sorti = 0;
bool *TireNumero::ptirage = 0;

int main() {
  TireNumero *ptn;

  TireNumero tn1;
  TireNumero::message1();
  tn1.message2();

  TireNumero tn2;
  TireNumero::message1();
  tn2.message2();

  ptn = new TireNumero();
  TireNumero::message1();
  ptn->message2();
  delete ptn;
  TireNumero::message1();

  ptn = new TireNumero();
  TireNumero ttn[20];

  TireNumero::message1();
  ttn[19].message2();

  delete ptn;
  TireNumero tn3;
  tn3.message1();
  tn3.message2();
  ttn[5].message2();

  cout << "Fin des tests" << endl;
}
```

La première remarque concerne le compilateur. Il nous indiquera immédiatement si les mots-clés sont correctement positionnés ou si nous accédons par exemple à la variable `quel_numero` à partir d'une méthode statique, ce qui est impossible.

La partie nouvelle consiste ici en l'initialisation des variables statiques :

```
int  TireNumero::nombre_sorti = 0;
bool *TireNumero::ptirage = 0;
```

Elles ne doivent apparaître qu'une seule fois dans le code et devraient se trouver dans le code de la classe, car, à nouveau pour des raisons de présentation, tout le code se trouve dans le même fichier. L'utilisation d'un tableau dynamique `ptirage` rend le code plus complexe, mais plus flexible. Nous pourrions, dynamiquement, attribuer un plus grand nombre de numéros à disposition.

Les variables `nombre_sorti` et `ptirage` sont donc statiques. Le grand travail se fait principalement dans le constructeur, avec l'allocation mémoire de `ptirage`. Nous retrouverons ce dernier dans le destructeur, où toutes les ressources devraient être effacées. Le résultat :

```
Nombre sorti: 1
Nombre actuel: 1
Nombre sorti: 2
Nombre actuel: 2
Nombre sorti: 3
Nombre actuel: 3
Nombre sorti: 2
Nombre sorti: 10
Nombre actuel: 0
Nombre sorti: 10
Nombre actuel: 3
Nombre actuel: 9
Fin des tests
Tous les numéros sont rentrés
```

devrait nous convaincre que le programme semble fonctionner correctement. Nous avons une longue liste de tests, où les `delete` permettent de récupérer des numéros. Le `ttn[5]` est le sixième composant dans le tableau `ttn`. Comme nous avons déjà trois autres numéros alloués, le dernier `9` est vraisemblablement correct. Le dernier `tn3.message1();` est volontaire, car il est possible d'appliquer une méthode statique à un objet de cette classe.

La position dans le tableau commence à 0 et nos numéros à 1. Cela explique les `+1` et `-1` dans notre code. Les méthodes `message1()` et `message2()` ne sont là que pour les tests et vérifier le comportement des méthodes et variables statiques.

En Java

Nous présentons d'abord le code Java, avant de passer aux détails et aux différences, relativement importantes, par rapport à la version C++ :

```
public class TireNumero {
  private static int nombreSorti;
  private static boolean tirage[];

  public int quelNumero;

  public TireNumero() {
    if (tirage == null) {
      tirage = new boolean[10];
    }
```

```
  if (nombreSorti == 10) {
    quelNumero = 0;
  }
  else {
    nombreSorti++;
    for (int i = 0; i < 10; i++) {
      if (tirage[i] == false) {
        tirage[i] = true;
        quelNumero = i + 1;
        break;
      }
    }
  }
}

public void dispose() {
  if (quelNumero != 0) {
    nombreSorti--;
    tirage[quelNumero - 1] = false;
  }
}

public int getNumero() {
  return quelNumero;
}

static int getNombreSorti() {
  return nombreSorti;
}

static void message1() {
  System.out.println("Nombre sorti: " + getNombreSorti());
}

void message2() {
  System.out.println("Nombre actuel: " + getNumero());
}

public static void main(String[] args) {
  TireNumero tn1 = new TireNumero();
  TireNumero tn2 = new TireNumero();
  TireNumero tn3 = new TireNumero();

  tn1.message2();
  tn3.message2();
  TireNumero.message1();

  tn2.dispose();
  tn1 = new TireNumero();  // !!!
  tn1.message2();
  message1();
```

```
    TireNumero ttn[] = new TireNumero[10];

    for (int i = 0; i < 10; i++) {
      ttn[i] = new TireNumero();
    }
    ttn[4].message2();
    message1();

    for (int i = 0; i < 10; i++) {
      ttn[i].dispose();
    }

    tn1.dispose();
    tn2.dispose();   // !!!
    tn3.dispose();
    ttn[9].message1();
  }
}
```

Nous avons modifié quelques fonctionnalités que nous avions certainement appréciées dans la version C++. En fait, reprendre le design d'une application développée dans un langage et l'appliquer directement dans un autre n'est pas toujours la meilleure solution. Nous le verrons très rapidement avec la libération de nos objets.

Nous savons déjà qu'il n'y a pas de destructeur en Java. Pour libérer un numéro alloué, nous avons introduit une méthode `dispose()` pour remplacer le destructeur en C++.

Si nous examinons le résultat :

```
Nombre actuel: 1
Nombre actuel: 3
Nombre sorti: 3
Nombre actuel: 2
Nombre sorti: 3
Nombre actuel: 8
Nombre sorti: 10
Nombre sorti: 0
```

il nous semble correct. La valeur 8 retournée vient de notre `ttn[4].message2();`, c'est-à-dire du cinquième élément du tableau. Nous dirons que nous avons plus ou moins alloué trois objets, puisque le code :

```
tn2.dispose();
tn1 = new TireNumero();  // !!!
```

a été mal écrit volontairement. Nous aurions dû utiliser `dispose()` sur l'objet `tn1`, et non `tn2`. Après la deuxième instruction, l'ancien `tn1` partira dans le ramasse-miettes. En fait, nous avons le même problème en C++ avec des pointeurs si nous oublions un `delete`.

Le dernier 0 peut être considéré comme un accident, car nous avons juste appelé la méthode `dispose()` le bon nombre de fois. Nous n'avons pas vérifié par exemple qu'elle

pourrait être appelée deux fois ou oubliée. Ce dernier cas est beaucoup plus complexe et se résout de manière différente en C++.

Il n'y a pas de grand mystère dans ce code, sinon qu'il ne faut pas oublier l'allocation des dix éléments du tableau tirage avant de les utiliser. Les autres constructions sont similaires aux commentaires que nous avons faits dans la version C++. La dernière instruction :

```
ttn[9].message1();
```

est tout à fait correcte. Nous pouvons aussi appliquer une méthode statique sur un objet de sa classe, et l'objet `ttn[9]` existe encore, bien que nous eussions dû éviter toute utilisation d'objets après l'appel à la méthode `dispose()`.

finalize() en Java

Nous devons mentionner qu'il existe tout de même une méthode `finalize()`, que nous pouvons ajouter à n'importe quelle classe. `finalize()` sera appelée par le ramasse-miettes.

La méthode statique `System.runFinalizersOnExit(true)` peut être exécutée pour s'assurer que les `finalize()` seront appelées avant la terminaison du programme. Nous aimerions cependant suggérer, comme dans l'exemple de la classe `TireNumero` précédente, de ne pas recourir à ce type de construction. Nous pourrions par exemple conseiller de mieux tester la classe et de modifier la méthode `dispose()` pour vérifier un double appel. Un tableau avec un état plus précis de l'objet semble nécessaire, c'est-à-dire un design modifié par rapport à la version C++ !

Un dernier exemple en Java

L'exemple que nous allons montrer pour terminer est le genre de code que le programmeur peut et doit écrire lorsqu'il se pose un certain nombre de questions sur le comportement d'un langage. Écrire le programme, sortir les résultats sur la console et faire une analyse est une démarche conseillée. Nous commençons par le code :

```
public class UnMessage {
  private String le_message;

  UnMessage() {
    le_message = "";
    System.out.println("Constructeur par défaut de la classe UnMessage");
  }

  UnMessage(String msg) {
    le_message = msg;
    System.out.println("Constructeur de la classe UnMessage avec " + msg);
  }

  public String toString() {
    return "Le message: " + le_message;
  }
```

```
  public static void main(String[] args) {
    UnMessage[] ma1 = new UnMessage[2];
    UnMessage[] ma2 = { new UnMessage(), new UnMessage("Salut") };

    ma1[0] = new UnMessage("Bonjour");
    ma1[1] = new UnMessage("Bonsoir");

    for (int i = 0; i < 2; i++) {
      System.out.println(ma1[i] + " et " + ma2[i]);
    }
  }
}
```

Nous poursuivons par le résultat :

```
Constructeur par defaut de la classe UnMessage
Constructeur de la classe UnMessage avec Salut
Constructeur de la classe UnMessage avec Bonjour
Constructeur de la classe UnMessage avec Bonsoir
Le message: Bonjour et Le message:
Le message: Bonsoir et Le message: Salut
```

Et nous terminons par une analyse :

- Avec `ma1`, nous allouons d'abord l'espace pour les deux pointeurs. `ma[0]` et `ma[1]` seront `null` au départ.
- Avec `ma2`, nous faisons les deux : la définition et l'affectation.
- Dans le constructeur par défaut, nous avons notre `le_message = "";` qui est nécessaire. Sinon `le_message` serait et resterait `null` et pourrait créer des erreurs inattendues.
- `toString()` est défini dans la classe `Object` et doit être redéfini avec la même signature (`public String ()`). Elle permettra de rendre possible l'exécution correcte de `println()`, qui utilise `toString()` pour faire le travail. La déclaration de la méthode `toString` dans une classe est essentielle. Elle sera souvent utilisée à des fins de tests.

Résumé

Le langage C++, plus encore que Java, offre de nombreuses possibilités pour construire des objets, ce qui nécessite une progression systématique dans notre apprentissage. La boucle ne sera bouclée qu'en fin de chapitre suivant.

Exercices

1. Créer en C++ une classe `Personnet` pour qu'elle accepte la construction `Personne1 personnes[100];`. Chaque objet de cette classe sera associé à un numéro entre 1 000 et 1 099, ceci en utilisant une variable de classe statique.

2. Créer en C++ une classe `Dessin` qui possède un nom et une position (`x`, `y`) sur l'écran. Définir un constructeur de copie qui va placer le nouveau dessin à une position (`x + 2`, `y + 1`). Si le nom du dessin d'origine est `Oeuvre1`, le nouveau dessin se nommera `Oeuvre1.a`.

11

Manipuler des objets en Java et C++

L'opérateur = ou un exercice d'affectation

Nous avons choisi volontairement d'améliorer nos connaissances, en procédant par petits pas, et d'attendre le plus longtemps possible avant de passer à cette partie, plus délicate mais essentielle. Dans les chapitres précédents, nous avons déjà créé de nombreux objets. Nous nous sommes posé toute une série de questions, mais nous avons soigneusement gardé celle-ci afin de conclure cette partie de l'ouvrage, consacrée à la création et à l'instanciation de classes traditionnelles. Les réponses à la question suivante :

Comment se comporte l'opérateur = ?

devraient nous permettre de bien mieux maîtriser ces deux langages.

Commençons en Java

Dans le petit exemple qui suit, nous allons créer une nouvelle classe nommée `MaClasse` et instancier un certain nombre d'objets pour exposer le sujet.

```
class MaClasse {
   private int compteur;
   private String nom;

   public MaClasse(int valeur, String identif) {
      compteur = valeur;
      nom = identif;
   }
```

```
   public String toString() {
      return nom + ": " + compteur;
   }

   public void setData(int valeur, String identif) {
      compteur = valeur;
      nom = identif;
   }
};

public class Assignement {
  public static void main(String args[]) {
     int a = 1;
     int b = 2;
     b = a;
     a = 3;
     System.out.println("a et b: " + a + " " + b);

     String stra = "str1";
     String strb = "str2";
     strb = stra;
     stra = "str3";
     System.out.println("stra et strb: " + stra + " " + strb);

     String nom1 = "Nom1";
     String nom2 = "Nom2";
     String nom3 = "Nom3";

     MaClasse mca = new MaClasse(1, nom1);
     MaClasse mcb = new MaClasse(2, nom2);
     mcb = mca;

     if (mca.equals(mcb)) {
       System.out.println("Les deux objets sont les mêmes!");
     }

     mca.setData(3, nom3);
     nom3 = "Salut";
     System.out.println("mca et mcb: ");
     System.out.println(mca);
     System.out.println(mcb);
  }
}
```

Nous présentons ici la première partie du résultat :

```
a et b: 3 1
stra et strb: str3 str1
```

qui est tout à fait attendue, si bien que le lecteur pourrait se demander où nous voulons en venir ! Nous avons défini deux entiers, `a` et `b`, avec des valeurs différentes. Ce qu'il est important de noter au sujet des instructions :

```
b = a;
a = 3;
```

c'est que nous allons seulement attribuer de nouvelles valeurs à ces deux objets : l'objet `b` ne va pas devenir l'objet `a`. En donnant une nouvelle valeur de `3` à l'entier `a`, nous constatons que nous touchons bien l'objet désiré. Il en va de même pour les `String stra` et `strb`, où le résultat est identique. Cependant, si nous examinons la deuxième partie du résultat :

```
Les deux objets sont les mêmes!
mca et mcb:
Nom3: 3
Nom3: 3
```

nous sommes peut-être étonnés ! L'instruction `mca.equals(mcb)` utilise la méthode `equals()` de la classe `Object`, qui implémente un test d'équivalence le plus discriminatif qui existe ! Les deux objets sont donc identiques, et cela implique que l'objet original `mcb` est perdu et se trouve déjà dans le ramasse-miettes (*garbage collection*). La méthode `setData()` de `MaClasse` va bien affecter `mcb`, puisque c'est le même objet.

Si nous écrivions en C++ :

```
int *pnombre1 = new int(1);
int *pnombre2 = new int(2);
pnombre2 = pnombre1;
```

ce serait tout à fait équivalent, où `pnombre1` et `pnombre2` pointeraient à la même adresse mémoire, donc au même objet. Nous devons nous rappeler qu'il n'y a pas de ramasse-miettes et qu'ici nous avons en fait oublié le :

```
delete pnombre2;
```

avant la dernière instruction.

Poursuivons en C++

La transition est assurée. Nous allons reprendre, en C++, notre classe Java avec deux variantes possibles. L'attribut `compteur` sera pour la classe `MaClasse1` un attribut entier traditionnel, alors que pour la classe `MaClasse2`, nous aurons un pointeur à un entier. Puisque l'objet `pcompteur` est alloué dynamiquement par le constructeur, nous sommes donc obligés d'introduire un destructeur pour `MaClasse2`. Assez de théorie, il nous faut présenter maintenant le code :

```
// assignement1.cpp
#include <iostream>
#include <string>

using namespace std;
```

```
class MaClasse1 {
   private:
     int compteur;

   public:
      MaClasse1(int valeur):compteur(valeur) {};
      int getCompteur() {
         return compteur;
      }

      void setCompteur(int valeur) {
         compteur = valeur;
      }
};

class MaClasse2 {
   private:
     int *pcompteur;

   public:
      MaClasse2(int valeur) {
         pcompteur = new int(valeur);
      };

      ~MaClasse2() {
         delete pcompteur;
      }

      int getCompteur() {
         return *pcompteur;
      }

      void setCompteur(int valeur) {
         *pcompteur = valeur;
      }
};

int main() {
  int a = 1;
  int b = 2;
  b = a;
  a = 3;
  cout << "a et b: " << a << " " << b << endl;

  string stra = "str1";
  string strb = "str2";
  strb = stra;
  stra = "str3";
  cout << "stra et strb: " << stra << " " << strb << endl;
```

```
  MaClasse1 mc1a(1);
  MaClasse1 mc1b(2);
  mc1b = mc1a;
  mc1a.setCompteur(3);
  cout << "mc1a et mc1b: " << mc1a.getCompteur() << " "
                           << mc1b.getCompteur() << endl;

  MaClasse2 mc2a(1);
  MaClasse2 mc2b(2);
  mc2b = mc2a;
  mc2a.setCompteur(3);
  cout << "mc2a et mc2b: " << mc2a.getCompteur() << " "
                           << mc2b.getCompteur() << endl;

  MaClasse2 mc2d(2);
  {
    MaClasse2 mc2c(1);
    mc2d = mc2c;
    mc2c.setCompteur(3);
    cout << "mc2c: " << mc2c.getCompteur() << endl;
  }
  cout << "mc2d: " << mc2d.getCompteur() << endl;
}
```

Le résultat global est identique à la version Java :

```
a et b: 3 1
stra et strb: str3 str1
mc1a et mc1b: 3 1
mc2a et mc2b: 3 3
mc2c: 3
mc2d: 4325492
```

Pour les entiers `a` et `b`, ainsi que pour les deux `string` C++ `stra` et `strb`, nous obtenons le même résultat qu'en Java. Nous aurions été surpris du contraire. Nos objets restent tels qu'ils étaient définis au départ, et nous ne changeons que leurs valeurs.

Les objets `mc1a` et `mc1b` sont des instances de la classe `MaClasse1`, et nous devons nous demander ce qui se passe avec l'instruction :

```
mc1b = mc1a;
```

Le langage C++, à l'inverse de Java, qui a simplifié la procédure en travaillant en fait comme des pointeurs, va chercher une définition pour l'opérateur `=`. Nous allons voir, dans un instant, que cela est tout à fait possible, mais, ici, le compilateur va exécuter un assignement par défaut. Il va en fait copier le contenu de l'attribut `compteur` de l'objet `mc1a` de la classe `MaClasse1`, c'est-à-dire sa valeur, dans `compteur` de `mc1b`. À l'instruction suivante :

```
mc1a.setCompteur(3);
```

nous remettons bien le compteur de `mca` à une nouvelle valeur. Où cela se gâte, c'est dans la troisième partie, avec l'utilisation d'un pointeur `pcompteur` dans la classe `maClasse2`. L'instruction :

```
mc2b = mc2a;
```

va écraser le pointeur `pcompteur` dans l'objet `mc2b` de la classe `MyClasse2`, et la mémoire qui se trouvait à cet endroit ne sera pas effacée. Lorsque nous déposons une nouvelle valeur dans `mc1a` avec `mc1a.setCompteur(3)`, nous touchons aussi l'objet `mc1b`. Comme `MyClasse2` possède un constructeur, il sera appelé deux fois pour le même objet ! Une alternative dans le destructeur de la classe `MaClasse2` serait :

```
~MaClasse2() {
   if (pcompteur != 0) {
      delete pcompteur;
      pcompteur = 0;
   }
}
```

Dans la dernière partie du code, nous procédons exactement au même exercice, mais l'objet `mc2c` est alloué et libéré à l'intérieur d'une nouvelle portée. Le résultat `4325492`, qui serait sans doute différent sur une autre machine, montre bien qu'il y a un problème ! L'alternative ci-dessus serait inutile et ne résoudrait le problème qu'à moitié. Il faudrait connaître combien d'instances utilise `pcompteur`.

Créer un opérateur =

La surcharge de l'opérateur `=` est essentielle pour des classes du type précédent, où il n'est pas possible de laisser le compilateur utiliser l'assignement par défaut. Nous allons construire à présent une nouvelle classe `MaClasse`, dont l'opérateur `=` se définit ainsi :

```
MaClasse& operator= (const MaClasse& original) {
```

Il reçoit bien une référence et retourne une référence de la même classe. Voici donc le code complet et la partie de test dans le `main()` :

```
// assignement2.cpp
#include <iostream>
#include <string>

using namespace std;

class MaClasse {
   private:
     int *pcompteur;

   public:
      MaClasse(int valeur) {
         pcompteur = new int(valeur);
      };
```

```
      MaClasse& operator= (const MaClasse& original) {
         if (this == &original) {
           cout << "Même objet" << endl;
           return *this;
         }
         cout << "Cela roule" << endl;
         delete pcompteur;
         pcompteur = new int(*original.pcompteur);
         return *this;
      }

      ~MaClasse() {
         delete pcompteur;
      }

      int getCompteur() {
         return *pcompteur;
      }

      void setCompteur(int valeur) {
         *pcompteur = valeur;
      }
};

int main() {
  MaClasse mca(1);
  MaClasse mcb(2);
  mcb = mca;
  mca.setCompteur(3);
  cout << "mca et mcb: " << mca.getCompteur() << " "
                         << mcb.getCompteur() << endl;
  MaClasse mcd(2);

  {
    MaClasse mcc(1);
    mcd = mcc;
    mcc.setCompteur(3);
    cout << "mcc: " << mcc.getCompteur() << endl;
  }
  cout << "mcd: " << mcd.getCompteur() << endl;

  mcd = mcd;
  cout << "mcd: " << mcd.getCompteur() << endl;
}
```

Ce dernier programme nous donnera le résultat suivant :

```
Ca roule
mca et mcb: 3 1
Ca roule
mcc: 3
```

```
mcd: 1
Même objet
mcd: 1
```

Un des aspects essentiels dans ce code est vraisemblablement le :

```
return *this;
```

qui nous retournerait un pointeur à une instance de l'objet. Il ne faudrait pas l'oublier, car il nous permettrait des constructions chaînées telles que :

```
mca = mcb = mcd;
```

L'attribut `pcompteur`, comme défini dans notre classe `MaClasse`, n'apporte rien. Il est ici pour montrer la manière de copier notre attribut. Nous avons évidemment besoin d'un destructeur. Lorsque nous avons :

```
mcd = mcc;
```

la méthode `operateur=`, car nous pouvons effectivement l'assimiler à une méthode, sera appelée sur l'objet `mcd`. Comme `pcompteur` est un objet dynamique, il faut commencer par l'effacer. Ensuite, l'instruction :

```
pcompteur = new int(*original.pcompteur);
```

va nous créer un nouvel objet `pcompteur` avec la valeur contenue dans l'objet `mcc`. Le code :

```
mcd = mcd;
```

peut paraître étrange, mais s'avère essentiel dans cette partie :

```
if (this == &original) {
  cout << "Même objet" << endl;
  return *this;
}
```

Il nous faut effectivement vérifier si nous avons affaire au même objet. Dans ce cas, nous retournerons simplement l'adresse du même objet, et il ne se passera rien du tout. Nous pourrions donner des exemples de cas précis où des accidents sont possibles si ce test d'égalité n'est pas exécuté.

Ici, nous n'avons qu'un seul objet, bien particulier. Si nous avions d'autres attributs, comme nous n'avons plus d'assignement par défaut, il faudrait aussi les copier. Certains de ces attributs pourraient être définis différemment. Ce dernier point est une question de conception et ne s'applique en général qu'au constructeur de copie que nous avons examiné au chapitre précédent.

L'incontournable classe string en C++

Dans tous les ouvrages dignes de ce nom, nous trouvons toujours un exercice d'implémentation d'une classe `string`. Nous savons qu'il existe la classe `string` de la bibliothèque

standard du C++, mais nous allons tout de même créer notre propre classe, `Mon_string`, qui utilisera elle-même un `string` standard dynamique, dont nous libérerons l'espace mémoire utilisé à cette occasion. Cette classe pourrait être utilisée comme modèle pour des débutants ou comme répétition générale après une absence prolongée de pratique de ce langage.

Nous en profiterons aussi pour reprendre un certain nombre de concepts et points essentiels qui concernent la création d'une classe en C++. Dans ce qui va suivre, la partie essentielle reste le destructeur. Si nous avons vu que Java n'a pas besoin de destructeur, nous allons revoir ici son importance en C++, car il permet de libérer toutes les ressources utilisées pendant la durée de vie de l'objet. La classe qui suit est aussi loin d'être complète :

```
// monstring.cpp
#include <iostream>
#include <string>

using namespace std;

class Mon_string {
  private:
    string *ple_string;

 public:
    Mon_string(const string  le_string);
    Mon_string(const char *pchaine = 0);
    Mon_string(const Mon_string &mon_string);
    Mon_string& operator=(const char *pchaine);
    ~Mon_string();
};

Mon_string::Mon_string(const string le_string) {
   ple_string = new string(le_string);
   cout << "Constructeur1 pour " << *ple_string << endl;
}

Mon_string::Mon_string(const char *pchaine) {
   ple_string = new string(pchaine);
   cout << "Constructeur2 pour " << *ple_string << endl;
}

Mon_string::Mon_string(const Mon_string &mon_string) {
   ple_string = new string(*mon_string.ple_string);
   cout << "Constructeur3 pour " << *ple_string << endl;
}

Mon_string::~Mon_string() {
   cout << "Destructeur pour " << *ple_string << endl;
   if (ple_string != NULL) {
     delete ple_string;
   }
```

```
}

Mon_string& Mon_string::operator=(const char *pchaine) {
   cout << "Operateur = avant " << *ple_string << endl;
   delete ple_string;
   ple_string = new string(pchaine);

   cout << "Operateur = après " << *ple_string << endl;
   return *this;
}

int main() {
   Mon_string un_string("Début des tests");

   Mon_string *pnom_str = new Mon_string("Salut");
   *pnom_str = "Bonjour";
   delete pnom_str;

   {
     Mon_string mon_str(string("Hello"));
     mon_str = "Good morning";
   }

   cout << "Fin du programme" << endl;
}
```

Nous allons tout de suite passer au travers des différentes parties de ce code, pour mieux comprendre pourquoi nous obtenons le résultat suivant :

```
Constructeur2 pour Debut des tests
Constructeur2 pour Salut
Operateur = avant Salut
Operateur = après Bonjour
Destructeur pour Bonjour
Constructeur1 pour Hello
Operateur = avant Hello
Operateur = après Good morning
Destructeur pour Good morning
Fin du programme
Destructeur pour Debut des tests
```

Pour simplifier, nous avons d'abord inclus, dans le même fichier, les trois parties distinctes du code, qui sont :

- la définition de la classe, en principe dans un fichier d'en-tête nommé, par exemple, `Mon_string.h` ;
- le code de la classe, généralement dans un fichier nommé `Mon_string.cpp`, c'est-à-dire toutes les méthodes commençant par `Mon_string::` ;

- la partie du `main()` qui devrait se retrouver dans un troisième fichier et qui pourrait être tout aussi bien un programme de test complet qu'une classe ou qu'une application utilisant d'autres classes.

La première chose que nous pouvons analyser dans cet exercice est l'identification des objets et à quel moment ils sont initialisés et libérés. L'objet `un_string` existe sur le tas. Il n'a pas été créé avec l'opérateur `new` et ne sera libéré que tout à la fin, même après notre dernier message. Nous pouvons même affirmer dans ce cas-là que le destructeur est appelé par la dernière accolade fermée de `main()` !

Dans cet exercice, nous avons aussi utilisé des chaînes de caractères du style C. Il est tout à fait acceptable de garder ce type de variable, même en Standard C++, au lieu d'utiliser systématiquement des `string` C++.

La partie intérieure entre les accolades `{}` s'amuse avec l'objet `mon_str`. Nous remarquons que le premier constructeur est utilisé, puisque son unique paramètre est un `string`. Juste après que `"Good morning"` a remplacé `"Hello"`, le destructeur est appelé, puisque nous sortons de la portée.

Nous retrouvons à nouveau la forme particulière de l'opérateur `=`, que nous avons déjà analysé dans ce chapitre. Sa fonction principale est d'effacer l'ancien objet, `"Hello"`, et de le remplacer par le nouveau, `"Good morning"`. Cette forme apparaît très souvent dans le code de classe utilisant l'opérateur `=`, et le programmeur s'y habituera très vite après quelques exercices. Enfin, il doit aussi retourner une référence `String&` pour d'éventuels chaînages. Nous devons faire remarquer que l'opérateur `=` se fait non sur le même type, mais sur un `const char *`. Dans le cas de l'implémentation de l'opérateur :

```
Mon_string & operator= (const Mon_string & original)
```

il faudrait aussi vérifier que les deux objets ne soient pas les mêmes. Au niveau des tests de cette classe, nous devrons nous assurer que les objets `"Hello"` et `"Salut"` ont bien disparu et ont été effacés correctement. Le destructeur devra faire proprement son travail. Il est responsable de l'effacement de toutes les ressources allouées dynamiquement avec l'opérateur `new` durant la durée de vie de l'objet.

Il faut aussi remarquer que si nous avions fait l'exercice avec un `string` au lieu d'un `string*`, toutes les opérations de nettoyage auraient été exécutées par la classe `string`. Le destructeur de `Mon_string` n'aurait pas été nécessaire, ainsi que les `delete` et `new` pour l'opérateur `=`. Un simple `le_string = chaine;` aurait suffi. En fait, le mécanisme décrit ci-dessus se retrouve dans la classe `string`. Le destructeur de la classe `string` lui-même aurait été appelé pour les deux objets restants, `"Bonjour"` et `"Good morning"`.

Recommandation d'ordre des méthodes

Dans le code de la classe `Mon_string` qui précède, nous trouvons les constructeurs, le destructeur et les méthodes. C'est un ordre conseillé aussi en Java, pour une lecture conviviale. Les méthodes `private` devraient être regroupées dans un bloc, après la partie `public`. En C++, l'ordre devrait être le même dans le fichier d'en-tête.

Certaines personnes groupent les méthodes par ordre alphabétique. Nous préférons choisir un ordre par groupe de fonctions ou d'importance. Par exemple, les méthodes de test pourraient se retrouver en fin de code.

Retour à la source

Après ce petit détour essentiel à la création de notre propre classe `string` en C++, il nous faut répéter, une fois encore, la construction particulière en Java :

```
String nom = new String("mon_nom");
nom = "ton_nom";
```

La deuxième forme est en fait, exceptionnellement, acceptée et intégrée dans le compilateur.

Nous rappellerons l'équivalent en C++ :

```
string nom = "mon_nom";
nom = "ton_nom";
```

Le clonage d'objet en Java

Nous avons donc vu que la forme en Java :

```
Personne p1 = new Personne(...);
Personne p2 = p1;
```

nous créait un alias. `p2` est bien le même objet. Si nous utilisons une méthode de `p2` pour modifier un attribut de `p2`, l'objet `p1` sera aussi affecté. Il y a cependant des situations où nous aimerions vraiment copier la brebis Dolly, afin de créer une copie conforme, un clone. Ceci peut se faire de cette manière :

```
Personne P2 = (Personne)p1.clone();
```

La machine virtuelle de Java alloue de la mémoire pour recopier l'objet courant dans une nouvelle instance. La méthode `clone()`, disponible dans la classe `Objet`, va copier bit à bit notre objet. Nous allons en comprendre le mécanisme au travers de ce petit exemple :

```
public class Clown implements Cloneable {
 private String nom;
 private String prenom;
 private int annee;

 public Clown(String leNom, String lePrenom, String lAnnee) {
   nom    = leNom;
   prenom = lePrenom;
   annee  = Integer.parseInt(lAnnee);
 }

 public void setAnnee(int lannee) {
   annee = lannee;
 }
```

```
  public void unTest() {
    System.out.print("Nom et prenom: " + nom + " " + prenom);
    System.out.println(" : " + annee);
  }

  public static void main(String[] args) {
    Clown nom1 = new Clown("Haddock", "Capitaine", "1907");
    Clown nom2;
    Clown nom3;

    nom2 = nom1;

    try {
      nom3 = (Clown)nom1.clone();
      if (!nom3.equals(nom1)) {
        System.out.println("Les objets nom1 et nom3 sont différents");
      }

      nom2.setAnnee(1917);
      nom3.setAnnee(1927);

      nom1.unTest();
      nom2.unTest();
      nom3.unTest();

      if (nom2.equals(nom1)) {
        System.out.println("Les objets nom1 et nom2 sont égaux");
      }
    }
    catch (CloneNotSupportedException cnse) {
      System.out.println("clone() exception" + cnse);
    }
  }
}
```

Nous oublierons pour l'instant `implements Cloneable`, que nous verrons au chapitre 13, car, sans cette construction et bien que le code compile correctement, nous aurions une exception `CloneNotSupportedException`. `clone()` est en fait une méthode protégée de la classe de base `Objet`, et cette construction nous permettra d'accéder à la méthode `clone()`. Si nous examinons le résultat :

```
Les objets nom1 et nom3 sont différents
Nom et prenom: Haddock Capitaine : 1917
Nom et prenom: Haddock Capitaine : 1917
Nom et prenom: Haddock Capitaine : 1927
Object nom1 et nom2 sont égaux
```

nous voyons bien que les deux objets `nom1` et `nom3` sont différents. Nous avons changé l'année de l'objet `nom3` sans toucher à `nom1`. Ce n'est pas le cas de `nom2`, car c'est un alias de `nom1`.

Nous pouvons donc utiliser l'un ou l'autre nom pour accéder au même objet. L'instruction `nom2.setAnnee(1917)` nous donne aussi une confirmation de nos affirmations.

Il est évident que nous allons retrouver le même problème en C++, où parfois nous pourrions copier des références à d'autres objets qui seraient des attributs de cette classe. Il est cependant possible de définir son propre clonage, comme en C++, mais il faudra utiliser une interface. Comme ce sujet n'a pas encore été traité, nous en donnerons un exemple en fin de chapitre 13.

Surcharge d'opérateurs en C++, et nos amis friend

C'est vraiment le dernier moment pour présenter la surcharge des opérateurs, bien que nous l'ayons déjà considérée pour l'opérateur `=` en C++. Mais le cas se pose aussi pour les autres opérateurs, dont nous verrons qu'ils sont souvent directement associés au mot-clé `friend`. Ce mot-clé n'a rien à voir avec la série télévisée ni avec Java, mais reste un sujet essentiel à traiter en C++.

Surcharge d'opérateur

Dans le code qui va suivre, nous allons présenter en C++ deux classes, `EntierA` et `EntierB`, sur lesquelles nous aimerions définir une opération d'addition. Nous utiliserons une méthode, et non un opérateur, pour montrer en fait la manière de faire en Java. En effet, si nous définissions ces deux classes en Java et que nous écrivions ceci :

```
EntierA ea = new EntierA(1);
EntierA ea = new EntierA(1);
```

il serait absolument impossible d'écrire :

```
ea = ea + eb;
```

En Java, l'opérateur `+` n'est disponible que pour les types prédéfinis, comme `int`, `double` et autres. Mais avant de passer à la définition de l'opérateur `+` en C++, présentons le code équivalent d'une addition au travers d'une méthode `add()` :

```
// amis1.cpp
#include <iostream>

using namespace std;

class EntierB;
class EntierA {
  private:
    int entiera;

  public:
    EntierA(int nombre):entiera(nombre) {}
    int add(const EntierB& un_entierb);
    int get() const { return entiera; }
```

```
};

class EntierB {
  private:
    int entierb;

  public:
    EntierB(int nombre):entierb(nombre) {}
    int add(const EntierA& un_entiera);
    int get() const { return entierb; }
};

int EntierA::add(const EntierB& un_entierb) {
  entiera = entiera + un_entierb.get();
  return entiera;
}

int EntierB::add(const EntierA& un_entiera) {
  entierb = entierb + un_entiera.get();
  return entierb;
}

int main() {
  EntierA ea(10);  // ea est 10
  EntierB eb(20);  // eb est 20

  ea.add(eb);      // ea devient 30 (10 + 20)
  eb.add(ea);      // eb devient 50 (30 + 20)

  cout << ea.get() << " " << eb.get() << endl;
}
```

La déclaration simplifiée et avancée (*forward declaration*) avec `class EntierB;` au début du code peut paraître étrange, mais elle est nécessaire pour permettre la compilation de la classe `EntierA` qui l'utilise dans la méthode `add()`. C'est un truc à connaître lorsque les définitions de classe sont ainsi enchevêtrées. Il y a peu d'autres choses à dire sur ce code, sinon que nous utilisons certaines constructions évidentes dont il faut se rappeler les conséquences. Dans la méthode `add(const EntierB& un_entierb)` de la classe `EntierA`, nous avons :

```
entiera = entiera + un_entierb.get();
```

et le `get()` est effectivement nécessaire, car nous ne pouvons pas accéder à l'attribut `entierb`, qui est privé. Il faudrait vraiment être très ami pour obtenir ce privilège et pouvoir écrire directement :

```
entiera = entiera + un_entierb.entierb;
```

La seule manière dans ce cas serait de rendre `entierb` public, et nous savons déjà que ce n'est pas du tout la bonne solution. La construction :

```
entiera = entiera + un_entierb;
```

ou

```
entiera += un_entierb;
```

serait possible en C++ à condition de définir l'opérateur `+` et `+=`, ce que nous verrons plus loin. Enfin, la construction :

```
cout << ea.add(eb);
```

est possible, car un `int` retourné est accepté par l'`ostream`. Une forme telle que :

```
cout << ea;
```

ne serait pas possible sans définir l'opérateur `<<` pour la classe `EntierA`. Le résultat du code précédent :

```
30 50
```

est la somme de 10 et 20 à laquelle nous ajoutons encore 20 afin d'utiliser la méthode `add()` de la deuxième classe `EntierB`.

Pas de surcharge d'opérateur en Java

Il n'y a pas de surcharge d'opérateur en Java. Certains s'en plaindront, d'autres s'en passeront sans autres commentaires. L'implémentation des classes `EntierA` et `EntierB` est relativement directe :

```
class EntierA {
  private int entiera;

  public EntierA(int nombre) {
    entiera = nombre;
  }

  public int add(EntierB unEntierb) {
    entiera = entiera + unEntierb.getEntierb();
    return entiera;
  }
  public int getEntiera() {
    return entiera;
  }
}

class EntierB {
  private int entierb;

  public EntierB(int nombre) {
    entierb = nombre;
```

```
  }

  public int add(EntierA unEntiera){
    entierb = entierb + unEntiera.getEntiera();
    return entierb;
  }

  public int getEntierb() { return entierb; }
}

public class Amis {
  public static void main(String[] args) {
    EntierA ea = new EntierA(10);
    EntierB eb = new EntierB(20);

    System.out.println(ea.add(eb) + " " + eb.add(ea));
  }
}
```

Le mécanisme de déclaration avancée pour `EntierA` n'est pas nécessaire en Java, et nous retrouvons les `new`, qui n'étaient pas requis en C++ pour des objets sur le tas. Nous avons deviné qu'une surcharge de l'opérateur << n'existait pas en Java, tout en reconnaissant que le chaînage des << en C++ est bien pratique.

Les friend, ces amis qui nous donnent l'accès

Nous ferons ici un passage rapide, mais obligé, sur la présentation du mot-clé `friend`. Ce dernier, comme son nom l'indique, ami, va permettre à nos classes d'obtenir l'accès à une autre classe ou fonction. Il va obtenir en particulier le privilège d'accéder à des variables privées ou protégées. Nous présentons à présent le code de nos classes `EntierA` et `EntierB` en employant `friend` :

```
// amis2.cpp
#include <iostream>

using namespace std;

class EntierB;

class EntierA {
  friend EntierA operator+(const EntierA& un_entiera, const EntierB& un_entierb);

  friend EntierA operator+(const EntierA& un_entiera, const int nombre) {
      return EntierA(un_entiera.entiera + nombre);
  }

  friend ostream& operator<<(ostream& os, const EntierA& un_entier) {
    return os << un_entier.entiera;
  }
```

```
  private:
    int entiera;

  public:
    EntierA(int nombre):entiera(nombre) {}
};

class EntierB {
  friend int operator+(const int nombre, const EntierB& un_entierb) {
    return un_entierb.entierb + nombre;
  }

  private:
    int entierb;

  public:
    EntierB(int nombre):entierb(nombre) {}
};

EntierA operator+(const EntierA& un_entiera, const EntierB& un_entierb) {
  return EntierA(un_entiera.entiera + un_entierb);
}

int main() {
  EntierA ea(10);
  EntierB eb(20);

  ea = ea + 70 + eb;
  // ea = eb + ea;
  cout << ea << endl;
}
```

Pour comprendre ce code d'une manière plus directe, nous commencerons par le `70 + eb`. À gauche nous avons un `int`, et à droite, un `EntierB`. Si nous voulions remplacer l'opérateur + par une méthode, nous ne le pourrions pas, car le résultat, qui sera un entier, n'a rien à voir avec une méthode de classe dont il faudrait définir un objet. Cette addition pourrait être remplacée par une fonction C telle que :

```
int res = add(int nombre, const EntierB& un_entierb);
```

Comme cette fonction accède à une variable privée, nous avons besoin d'un `friend`, afin de pouvoir accéder à l'attribut `entierb` privé. Si nous voulions nous passer du `friend`, il faudrait réintroduire une méthode `get()` sur la classe `EntierB`.

Après avoir exécuté `70 + eb`, nous passons le résultat à `ea +` (résultat). Cette fois-ci, l'opérateur :

```
friend EntierA operator+(const EntierA& un_entiera, const int nombre)
```

sera utilisé avec le `int` à droite. Nous pourrions aussi écrire ceci :

```
ea = (ea + 70 ) + eb;
```

car l'opérateur :

```
friend EntierA operator+(const EntierA& un_entiera, const EntierB& un_entierb);
```

est aussi défini. Il faut toujours lire à gauche et à droite. Nous n'avons bien entendu pas couvert toutes les possibilités, puisque la ligne de code :

```
// ea = eb + ea;
```

serait rejetée avec :

```
g++ -c amis2.cpp
amis2.cpp: In function `int main()':
amis2.cpp:50: no match for `EntierB & + EntierA &'
amis2.cpp:40: candidates are: class EntierA operator
                    ➥+(const EntierA &, const EntierB &)
amis2.cpp:12:  class EntierA operator +(const EntierA &, int)
amis2.cpp:28:  int operator +(int, const EntierB &)
make: *** [amis2.o] Error 1
```

Enfin, le :

```
friend ostream& operator<<(ostream& os, const EntierA& un_entier)
```

est une forme très souvent utilisée afin de définir l'opérateur << sur ces classes. Ce `friend` va être remplacé par l'emploi d'une méthode `toString()` publique, comme c'est le cas en Java.

Il n'est pas nécessaire d'avoir systématiquement des `friend` pour toutes les surcharges et redéfinitions d'opérateurs. Les `friend` sont en fait des fonctions définies en dehors de la classe. L'écriture de classes contenant des amis est très particulière. Il faut parfois beaucoup de temps pour découvrir une erreur et comprendre vraiment ce qui se passe. Le lecteur peut s'amuser, par exemple, à effacer un `friend` ou à remplacer une référence par un passage par valeur ! Bon courage !

Dans le fichier `amis2.cpp` qui précède, nous avons à nouveau un assemblage des trois composants du code, qui vont du fichier d'en-tête à l'application, elle-même définie par le `main()`.

Amis : un exemple plus complet

Dans l'exemple qui va suivre, nous allons montrer la nécessité de l'utilisation des « amis » pour un certain nombre d'opérateurs.

```
// Vitesse1.cpp
#include <iostream>

using namespace std;

class Vitesse1 {
  private:
    double x, y;
```

```
  public:
    Vitesse1(double le_x = 0, double le_y = 0) : x(le_x), y(le_y) {}
    Vitesse1(const Vitesse1 &vit) : x(vit.x), y(vit.y) {}
    Vitesse1& operator=(const Vitesse1& vit) {
      x = vit.x;
      y = vit.y;
    }

    friend Vitesse1 operator*(Vitesse1& une_vit, double facteur);
    friend Vitesse1 operator*(double facteur, Vitesse1& une_vit);
    friend Vitesse1 operator*(Vitesse1& une_vit1, Vitesse1& une_vit2);

    friend ostream& operator<<(ostream& os, const Vitesse1& vit) {
      return os << "Direction x:y = " << vit.x << ":" << vit.y;
    }
};

Vitesse1 operator*(Vitesse1& la_vit, double facteur) {
  Vitesse1 une_vit(la_vit);
  une_vit.x *= facteur;
  une_vit.y *= facteur;
  return une_vit;
}

Vitesse1 operator*(double facteur, Vitesse1& la_vit) {
  Vitesse1 une_vit(la_vit);
  une_vit.x *= facteur;
  une_vit.y *= facteur;
  return une_vit;
}

Vitesse1 operator*(Vitesse1& la_vit1, Vitesse1& la_vit2) {
  Vitesse1 une_vit(la_vit1);
  une_vit.x *= la_vit2.x;
  une_vit.y *= la_vit2.y;
  return une_vit;
}

int main() {
  Vitesse1 vit(10, 20);
  cout << vit << endl;

  vit = vit * 1.2;
  cout << vit << endl;

  vit = 0.8 * vit;
  cout << vit << endl;

  vit = vit * vit;
  cout << vit << endl;
```

```
    return 0;
}
```

Et le résultat, qui nous semble correct :

```
Direction x:y = 10:20
Direction x:y = 12:24
Direction x:y = 9.6:19.2
Direction x:y = 92.16:368.64
```

La manière la plus directe de comprendre ce code est d'examiner la partie principale du programme, le `main()`. Si nous remplacions `Vitesse1 vit(10, 20);` par `double vit = 10.1;`, nous n'aurions alors plus besoin de notre classe `Vitesse1`. Nous pourrions alors compiler et exécuter ce programme, sans difficulté, avec bien évidemment un résultat différent. Cela veut dire que les opérateurs `<<` et `*`, qui sont définis par le compilateur pour le type primitif `double`, doivent être implémentés dans notre classe `Vitesse1`.

Les quatre `friend` de la classe `Vitesse1` correspondent au cas présenté dans le `main()`. Il faut toujours considérer la partie droite et la partie gauche, et nous comprendrons la raison des deux implémentations pour `vit * 1.2` et `0.8 * vit`. Le double est d'un côté ou de l'autre !

S'il y avait d'autres types de variables, il faudrait ajouter d'autres définitions d'opérateurs `friend` ou éventuellement utiliser un transtypage dans le type implémenté. Les deux formes suivantes seront acceptées :

```
vit = vit * 10L;
vit = vit * (double)10L;
```

même pour la deuxième, car le transtypage fonctionne correctement. Par exemple, il ne serait pas nécessaire de définir cette entrée :

```
friend Vitesse1 operator*(Vitesse1& une_vit, long facteur);
```

Le `L` du `10L` indique que nous avons en fait un `long` et non pas un `int` par défaut.

Faut-il éviter les amis (friend) ?

C'est évidemment un non-sens dans la vie de tous les jours, mais pour les professionnels en programmation C++, c'est un vrai débat. Le but de cet ouvrage n'est pas de philosopher pour savoir si nous violons les règles de la programmation orientée objet, mais plutôt de présenter un certain nombre d'alternatives, afin que le lecteur se familiarise avec les techniques du langage et puisse, peu à peu, se faire une opinion personnelle sur l'utilisation correcte ou non de ces formes spécialisées et souvent nécessaires. Par ailleurs, il ne faut pas oublier que ces constructions n'existent pas en Java qui s'en passe très bien. Dans le code qui suit, nous avons réécrit le code précédent en faisant disparaître toutes les instructions utilisant `friend`.

L'astuce majeure est d'utiliser une construction qui n'accède jamais aux attributs privés de la classe :

```
// Vitesse2.cpp
#include <iostream>

using namespace std;

class Vitesse2 {
  private:
    double x, y;

  public:
    Vitesse2(double le_x = 0, double le_y = 0) : x(le_x), y(le_y) {}
    Vitesse2(const Vitesse2 &vit) : x(vit.x), y(vit.y) {}
    Vitesse2& operator=(const Vitesse2& vit) {
      x = vit.x;
      y = vit.y;
    }

    void operator*=(const Vitesse2& une_vit) {
      x *= une_vit.x;
      y *= une_vit.y;
    }

    void operator*=(const double valeur) {
      x *= valeur;
      y *= valeur;
    }

    ostream& toString(ostream& os) const {
      return os << "Direction x:y = " << x << ":" << y;
    }

    void print() {
      cout << "Direction x:y = " << x << ":" << y << endl;
    }
};

Vitesse2 operator*(const Vitesse2& la_vit, const double facteur) {
  Vitesse2 une_vit(la_vit);
  une_vit *= facteur;

  return une_vit;
}

Vitesse2 operator*(const double facteur, const Vitesse2& la_vit) {
  Vitesse2 une_vit(la_vit);
  une_vit *= facteur;

  return une_vit;
}

Vitesse2 operator*(const Vitesse2& la_vit1, const Vitesse2& la_vit2) {
```

```
  Vitesse2 une_vit(la_vit1);
  une_vit *= la_vit2;

  return une_vit;
}

ostream& operator<<(ostream& os, const Vitesse2& vit) {
  return vit.toString(os);
}

int main() {
  Vitesse2 vit(10, 20);
  cout << vit << endl;

  vit = vit * 1.2;
  cout << vit << endl;

  vit = 0.8 * vit;
  cout << vit << endl;

  vit = vit * vit;
  cout << vit << endl;
  return 0;
}
```

Nous n'allons pas épiloguer cette fois-ci sur cette partie de code, mais nous dirons cependant qu'elle nous plaît beaucoup. Le lecteur peut en faire une comparaison directe avec l'exemple précédent. Les débutants auront besoin de plus de temps pour assimiler ces constructions essentielles.

Ce type de code devrait se retrouver très souvent, lorsque les programmeurs décident d'utiliser des opérateurs au lieu de méthodes. L'opérateur `<<` est évidemment essentiel, proche du `toString()` en Java. Cependant, s'il s'agit par exemple d'employer l'opérateur `*` pour la classe `Personne`, nous pourrions nous poser la question de son utilité. Pour une classe `Employee`, nous pourrions nous demander s'il peut augmenter le salaire et l'adapter en fonction de l'inflation. De nombreux programmeurs se sont souvent posé la question de savoir lequel des opérateurs `+` ou `&` (AND) devrait être utilisé pour une classe `string`. Nous pourrions répondre que Java se débrouille très bien sans opérateur avec la méthode `concat()` de sa classe `String` :

```
public String concat(String str);
```

Résumé

Comprendre le comportement de l'opérateur `=` en Java et C++ est essentiel, afin de le surcharger éventuellement en C++. Utiliser ou non les opérateurs en C++, « amicalement » ou non, demande une certaine attention, bien qu'un programmeur Java doive se débrouiller

sans cette ressource C++ essentielle. C'est vraiment un sujet plein de contradictions, mais fondamental dans la compréhension et l'utilisation correcte de ces deux langages.

Exercices

1. Créer en Java une classe `AssigneTest` avec ce code dans la partie `main()` :

```
MonString mstr1 = new MonString("Test1");
MonString mstr2 = new MonString("Test2");
mstr2 = mstr1;

MaClasse mca = new MaClasse(mstr1);
MaClasse mcb = new MaClasse(mstr2);
mcb = mca;
mstr1.setData("Test3");

System.out.println(mca);
System.out.println(mcb);
```

2. Écrire le code nécessaire pour obtenir ce résultat :

```
Test3
Test3
```

   Les deux classes `MonString` et `MaClasse` ne posséderont qu'un seul attribut, respectivement un `String` traditionnel et un objet de `MonString`. L'instruction `mstr1.setData("Test3");` va en fait affecter l'objet identifié avec la variable `mcb`.

3. Reprendre la classe `MaClasse` en C++ afin de traiter le constructeur par défaut, de vérifier si les deux objets sont égaux pour l'opérateur `=` et d'implémenter le code correctement, si le pointeur `pcompteur` est 0.

4. Étendre le code d'`amis2.cpp`, afin qu'il puisse traiter au minimum l'exemple suivant :

```
ea += eb + 1;
eb++;
eb += 10.1 + ea;
cout << eb;
```

   et sortir le résultat :

```
Mon test: 62
```

   si `ea` et `eb` sont initialisés avec des valeurs de 10 et 20.

12
Un héritage attendu

Nous avons appris qu'une classe était en fait un nouveau type, défini par le programmeur, afin de lui permettre de représenter un problème dans le monde de la programmation Java ou C++. Dans ce chapitre, nous allons étendre le concept de classe à celui de classe dérivée.

Les exemples traditionnels que nous trouvons dans la littérature concernent aussi bien des formes graphiques que le personnel d'une compagnie ou les membres d'une société. Dans le cas de formes graphiques, telles que des carrés, des rectangles ou des cercles, nous pourrons créer une classe pour chacune de ces formes à partir d'une classe de base qui possédera des caractéristiques communes, comme un point d'origine ou des méthodes pour dessiner, déplacer, cacher ou encore effacer l'objet. Dans le cas de la classe `Carre`, représentant notre forme carrée, nous dirons qu'elle dérive de la classe de base `Forme`. Pour le personnel d'une compagnie ou les membres d'une société, nous pourrons faire la même analogie avec des directeurs et des employés. Un employé pourra faire des heures supplémentaires, alors qu'un directeur n'aura pas à pointer à son arrivée et à son départ du travail ! Si ce directeur fait partie d'une société en tant que membre honoraire, il n'aura sans doute pas de cotisation à payer. Dans ce dernier cas, il pourra hériter d'une classe de base `Societaire` avec des caractéristiques spécifiques.

L'exemple de java.lang.Integer

Une autre approche, beaucoup plus directe, pour comprendre le concept de classe de base et de classe dérivée, est d'analyser l'une des nombreuses classes de Java. Nous prendrons en fait un exemple avec une seconde dérivation, car toute classe en Java dérive de la classe de base `Object`. En regardant la documentation de la classe `Integer`, nous découvrons ceci :

```
public final class Integer extends Number implements Comparable
```

Nous oublierons la dernière partie, `implements Comparable`, qui nous permettra de comparer des objets de différents types et sur laquelle nous reviendrons plus loin, ainsi que le `final`, qui nous indique que la classe `Integer` ne pourra plus être utilisée comme nouvelle classe de base pour des extensions possibles. La partie essentielle :

```
class Integer extends Number
```

nous indique que la classe `Integer` va hériter des caractéristiques de la classe de base `Number`. En regardant à présent la classe `Number`, nous découvrons ceci :

```
public abstract class Number extends Object implements Serializable
```

Nous faisons de même en oubliant `implements Serializable`, qui permettra de sérialiser des objets au travers d'une interface. Nous devons cependant revenir sur `abstract`. Ce mot-clé nous indique que nous avons affaire à une classe abstraite, `Number`, dont il ne sera pas possible d'instancier un objet directement. Cela peut sembler bien mystérieux, jusqu'au moment où nous découvrons que les classes `BigDecimal`, `BigInteger`, `Byte`, `Double`, `Float`, `Long` et `Short`, en plus d'`Integer`, héritent aussi de la classe abstraite `Number`. Mais que s'est-il donc passé ? C'est en fait très simple. Durant la conception de ces différentes classes, les analystes ont découvert que chacune d'entre elles pouvait hériter de toute une série de fonctionnalités communes. C'est ici que nous rencontrons le terme de réutilisation (*reusability* ou *reuse* en anglais).

Dans notre exemple ci-dessus d'une société, nous pourrions écrire ceci :

```
public class MembreHonoraire extends Societaire
```

qui nous indique que la classe `MembreHonoraire` va simplement étendre la classe `Societaire`. Cependant, durant la conception de nos classes, nous aurions pu découvrir qu'une implémentation telle que :

```
public abstract class Membre
public class MembreHonoraire extends Membre
public class Societaire extends Membre
```

est peut-être plus judicieuse. Les classes `MembreHonoraire` et `Societaire` peuvent être instanciées, ce qui n'est pas le cas de la classe abstraite `Membre`, qui ne va contenir que des méthodes et attributs utilisés par ses sous-classes. Le terme de sous-classe doit être expliqué, car il peut sembler encore moins clair que son équivalent de classe dérivée. Il veut bien dire qu'il hérite de toutes les caractéristiques d'une classe de base et non pas d'une partie, par exemple avec le sens de sous-évaluée. La sous-classe aura en fait plus de caractéristiques que sa superclasse, un autre nom pour la classe de base, qui, elle-même, héritera peut-être d'une autre classe devant se terminer, dans tous les cas en Java, par la classe `Object`. Il n'y a pas ce concept en C++, et la classe `Membre` sera la première classe dans la hiérarchie. En Java, `Membre` héritera automatiquement d'`Object`, et sans déclaration explicite.

La réutilisation

Un programmeur C n'a souvent pas d'autres choix que de copier et de modifier son code. S'il est assez expérimenté, il essaiera d'écrire le maximum de code dans des fonctions génériques et réutilisables.

Le pas suivant est la conception de classes, en Java et en C++, suffisamment génériques pour être réutilisables. Cette notion de classe générique et réutilisable est essentielle lors du développement de nouveaux produits. Il y a plusieurs cas de figure comme ceux-ci :

- Une partie du code peut dépendre de la machine ou du système d'exploitation, et il doit être isolé proprement.
- L'interface graphique d'un jeu d'échecs sur PC peut être utilisée pour un jeu totalement différent.
- Le cœur d'un programme de jeu d'échecs en C++ peut utiliser les `iostreams` pour sauver des parties en cours ou charger des modèles ou des parties de grand maître. Il pourrait même compiler et tourner sur un serveur Unix, Linux ou Windows NT.
- Le déplacement sur l'échiquier peut se faire sur des tableaux avec des méthodes réutilisables pour d'autres types de jeux.

Dans les paragraphes qui suivent, les sujets comme l'héritage, la composition ou les classes abstraites sont essentiels dans ce concept de réutilisation.

Héritage et composition

Le concept d'héritage est fondé sur la création d'une nouvelle classe sur la base d'un type de classe existant. Celui de la composition nous est déjà familier, car il consiste à créer des objets, plus communément appelés des attributs, dans une nouvelle classe. Dans ce code Java :

```
public class SocietaireC {
  private Personne unSocietaire;
  private int cotisation;
}
```

la classe `SocietaireC` possède deux attributs privés, dont la classe `Personne`. Comme nous utiliserons la classe `Societaire` pour un héritage classique, nous avons ajouté la lettre C pour indiquer une composition. Le terme de **sous-objet** est parfois utilisé pour des attributs de classe comme `unSocietaire`, tout comme celui d'**embedded** en anglais, qui apparaît très souvent dans d'autres domaines informatiques. Le terme **embedded,** qui signifie encastré ou caché, s'avère probablement l'un des termes les plus corrects pour nous éclairer dans ce contexte de composition et d'héritage.

Nous remarquons que les variables sont privées, ce qui est une règle essentielle en programmation objet ; nous aurons alors besoin de méthodes particulières pour accéder aux attributs ou aux méthodes de la classe `Personne`. En utilisant la composition, nous ne pouvons pas accéder directement à une méthode de la classe `Personne` au travers d'une instance de la

classe `Societaire`. Cependant, ce serait le cas si nous avions la définition de `Societaire` de cette manière :

```
public class Societaire extends Personne;
```

Avant de revenir sur la syntaxe et des exemples en Java et C++ de classes dérivées, nous allons présenter notre classe `SocietaireC`, qui utilise l'approche de composition. Nous avons un peu raccourci notre classe `Personne`, présentée au chapitre 4, pour des raisons de simplicité dans la présentation. Voici donc la version Java :

```
class PersonneC {
 private String nom;
 private String prenom;
 private int    annee;

 public PersonneC(String lenom, String leprenom, int lannee) {
   nom = lenom;
   prenom = leprenom;
   annee = lannee;
 }

 public void un_test() {
   System.out.println("Nom et prénom: " + nom + " " + prenom);
   System.out.println("Année de naissance: " + annee);
 }
}

public class SocietaireC {
  private PersonneC unSocietaire;
  private int cotisation;

  public SocietaireC(String lenom, String leprenom, int lannee, int lacotis) {
    unSocietaire = new PersonneC(lenom, leprenom, lannee);
    cotisation = lacotis;
  }

  public void un_test() {
    unSocietaire.un_test();
    System.out.println("Cotisation:  " + cotisation);
  }

  public static void main(String[] args) {
    SocietaireC unSoc = new SocietaireC("Haddock", "Capitaine", 1907, 100);
    unSoc.un_test();
  }
}
```

et la version C++ :

```
// SocietaireC.cpp
#include <iostream>
#include <string>
```

```
using namespace std;

class Personne {
  private:
    string nom;
    string prenom;
    int    annee;

  public:
    Personne(const string lenom, const string leprenom, int lannee);
    void un_test();
};

class SocietaireC {
  private:
    Personne *pun_societaire;
    int cotisation;

  public:
    SocietaireC(const string lenom, const string leprenom,int lannee, int lacotis);
    ~SocietaireC();
    void un_test();
};

Personne::Personne(const string lenom, const string leprenom, int lannee)
         :nom(lenom), prenom(leprenom), annee(lannee) {
}

void Personne::un_test() {
 cout << "Nom et prénom: " << nom << " " << prenom << endl;
 cout << "Année de naissance: " << annee << endl;
}

SocietaireC::SocietaireC(const string lenom, const string leprenom, int lannee,
➥int lacotis) {
  pun_societaire = new Personne(lenom, leprenom, lannee);
  cotisation = lacotis;
}

SocietaireC::~SocietaireC() {
  delete pun_societaire;
}

void SocietaireC::un_test() {
  pun_societaire->un_test();
  cout << "Cotisation:  " << cotisation << endl;
}

int main() {
  SocietaireC un_soc("Haddock", "Capitaine", 1907, 100);
```

```
  un_soc.un_test();
  return 0;
}
```

Comme les variables `unSocietaire` en Java et `pun_societaire` en C++ sont des objets de la classe `Personne` et qu'ils sont des attributs de la classe `Societaire`, nous parlons donc de composition. Ce qui est intéressant ici, c'est l'appel de la méthode `un_test()` de la classe `SocietaireC`. Comme nous aimerions aussi appeler la méthode `un_test()` de l'objet de la classe `Personne`, nous devons utiliser la variable d'objet pour appeler cette méthode. Il n'y a rien de mystérieux, mais nous pourrons comparer plus loin ce code avec l'implémentation d'une classe `Societaire` qui hérite de `Personne` et non plus composée d'un objet de la classe `Personne`.

Dans la version C++, `pun_societaire` est un pointeur, et il nécessite donc un destructeur dans sa classe `Societaire` pour effacer les ressources.

Note

Une classe `final` en Java ne peut plus être étendue. Si nous voulions étendre la classe `String` de Java, nous n'aurions qu'une seule solution : utiliser la composition. Nous pourrions alors tout de même étendre cette classe `final`, mais d'une manière beaucoup moins élégante et accessible que si nous avions pu la dériver. Nous aborderons les aspects de performance de classes `final` dans le chapitre 15, traitant des performances.

L'encapsulation des données

L'encapsulation peut être définie comme l'isolement des données. Nous allons voir ici que cet aspect est essentiel pour assurer le maximum de flexibilité et de réutilisation du code.

Nous prendrons ici notre exemple précédent en C++, qui est tout aussi applicable en Java. En définissant la variable privée `*pun_societaire` dans la classe `SocietaireC`, cela signifie qu'il est tout à fait possible de changer le code dans le futur, c'est-à-dire dans les classes `Personne` et `SocietaireC`. L'interface de la méthode `un_test()` de `SocietaireC`, extrêmement simple ici, mais qui pourrait contenir des arguments et une valeur de retour, doit simplement rester identique. Le code de la méthode `un_test()` dans la classe `SocietaireC` pourrait être totalement réécrit, et nous pourrions très bien remplacer `Personne` par une autre classe ou une autre implémentation. Si nous rendions la variable `*pun_societaire` publique de cette manière :

```
class SocietaireC {
  private:
    int cotisation;

  public:
    Personne *pun_societaire;
    SocietaireC(string lenom, string leprenom, int lannee, int lacotis);
```

```
    ~SocietaireC();
    void un_test();
};
```

nous pourrions alors écrire ceci dans le `main()` :

```
int main() {
  SocietaireC un_soc("Haddock", "Capitaine", 1907, 100);
  un_soc.un_test();
  un_soc.pun_societaire->un_test();
  return 0;
}
```

Le résultat sera donc :

```
Nom et prénom: Haddock Capitaine
Année de naissance: 1907
Cotisation:  100
Nom et prénom: Haddock Capitaine
Année de naissance: 1907
```

Nous comprenons bien que cette construction nous ferait perdre tous les avantages décrits ci-dessus. Le fait de protéger et de cacher les données est fondamental en programmation objet et en réutilisation de code. Un code bien écrit, qui doit être modifié pour toute sorte de raisons, nécessitera un minimum ou aucune correction dans les applications qui utiliseront nos API (*Application Programming Interface*).

La syntaxe de l'héritage en Java et C++

Nous avons déjà donné un aperçu ci-dessus de la syntaxe de l'héritage en Java, avec l'exemple de la classe `Integer`. La voici donc à nouveau, pour notre classe `Societaire` :

```
public class Societaire extends Personne {
}
```

En C++ la forme est un peu différente. Nous pourrions dire que les deux-points (`:`) remplacent le `extend` :

```
class Societaire : public Personne {
}
```

Le `public` en Java et en C++ n'a pas la même signification et n'est d'ailleurs pas à la même position. En Java nous connaissons déjà sa signification, comme nous l'avons vu au chapitre 7, lors du traitement des paquets (*package*). Nous rappellerons qu'un fichier `.java` ne peut contenir qu'une seule classe `public`. Cette remarque est judicieuse ici, car, pour simplifier la présentation de nos exemples et exercices, nous avons défini les classes de bases et dérivées dans le même fichier. Il en va de même pour la définition des classes en C++, qui devraient être définies dans des fichiers d'en-tête `.h` pour des raisons de réutilisation.

Enfin, le `public` en C++ indique que tous les membres publics de la classe de base resteront publics dans la classe dérivée. Nous passerons sur ce détail, et il y en a beaucoup en C++, et n'utiliserons par la suite que cette forme avec le `public`.

L'initialisation des constructeurs

Nous allons à présent reprendre notre exemple de classe `Societaire` et montrer deux exemples pratiques d'utilisation. Nous verrons aussi la manière d'initialiser la classe elle-même et sa ou ses classes de base. Notre classe Java `Societaire` se présentera ainsi :

```
class Personne {
 private String nom;
 private String prenom;
 private int    annee;

 public Personne(String lenom, String leprenom, int lannee) {
   nom = lenom;
   prenom = leprenom;
   annee = lannee;
   System.out.println("Constructeur de Personne");
 }

 public void un_test() {
   System.out.println("Nom et prénom: " + nom + " " + prenom);
   System.out.println("Année de naissance: " + annee);
 }
}

public class Societaire extends Personne {
  private int cotisation;

  public Societaire(String lenom, String leprenom, int lannee, int lacotis) {
    super(lenom, leprenom, lannee);
    cotisation = lacotis;
    System.out.println("Constructeur de Societaire");
  }

  public void un_test() {
    super.un_test();
    System.out.println("Cotisation:  " + cotisation);
  }

  public static void main(String[] args) {
    Societaire unSoc = new Societaire("Haddock", "Capitaine", 1907, 100);

    unSoc.un_test();
  }
}
```

alors qu'en C++ nous aurons :

```
// Societaire.cpp
#include <iostream>
#include <string>

using namespace std;

class Personne {
  private:
    string nom;
    string prenom;
    int    annee;

  public:
    Personne(const string lenom, const string leprenom, int lannee);
    ~Personne();
    void un_test();
};

class Societaire : public Personne {
  private:
    int cotisation;

  public:
    Societaire(const string lenom, const string leprenom, int lannee, int lacotis);
    ~Societaire();
    void un_test();
};

Personne::Personne(const string lenom, const string leprenom, int lannee)
         :nom(lenom), prenom(leprenom), annee(lannee) {
  cout << "Constructeur de Personne" << endl;
}

Personne::~Personne() {
  cout << "Destructeur de Personne" << endl;
}

void Personne::un_test() {
 cout << "Nom et prénom: " << nom << " " << prenom << endl;
 cout << "Année de naissance: " << annee << endl;
}

Societaire::Societaire(const string lenom, const string leprenom, int lannee,
➥int lacotis)
           :Personne(lenom, leprenom, lannee) {
  cotisation = lacotis;
  cout << "Constructeur de Societaire" << endl;
}
```

```
Societaire::~Societaire() {
  cout << "Destructeur de Societaire" << endl;
}

void Societaire::un_test() {
  Personne::un_test();
  cout << "Cotisation:  " << cotisation << endl;
}

int main() {
  Societaire un_soc("Haddock", "Capitaine", 1907, 100);

  un_soc.un_test();
  return 0;
}
```

Nous avons introduit des messages dans les différentes parties du code. Ils nous permettent d'analyser le chemin des appels de constructeurs, méthodes et destructeurs (ces derniers en C++ seulement). Dans les deux exemples, nous avons le même résultat :

```
Constructeur de Personne
Constructeur de Societaire
Nom et prénom: Haddock Capitaine
Année de naissance: 1907
Cotisation:  100
```

avec en plus pour le cas C++ et à la fin :

```
Destructeur de Societaire
Destructeur de Personne
```

Les appels des différents constructeurs et destructeurs de la classe de base `Personne` et de sa sous-classe `Societaire` suivent une logique attendue. C'est cependant un aspect important à considérer si des ressources communes étaient utilisées. C'est le cas en particulier pour le constructeur de la sous-classe, qui peut utiliser des ressources allouées par le constructeur de sa classe de base. Cependant, ce genre de problème ne devrait pas apparaître si nous suivons la règle d'allouer le minimum de ressources dans les constructeurs. Ces derniers ne peuvent de toute manière retourner des erreurs qu'en générant des exceptions.

La grande différence entre Java et C++ est la manière d'initialiser la classe de base (`super()` en Java et `:Personne()` en C++) et la façon d'appeler une méthode du même nom de la classe de base (`super.` en Java et `Personne::` en C++). Ce ne sont en fait que des détails de syntaxe, mais il faut se les rappeler. En cas d'erreur, les compilateurs peuvent nous donner des indications intéressantes. Comme illustration, nous prendrons deux exemples. Nous commencerons par le code Java de la classe `Societaire`, où nous mettrons en commentaire la ligne :

```
    // super(lenom, leprenom, lannee);
```

En supprimant cette ligne, nous empêchons une initialisation correcte du constructeur de base. Si nous essayions de compiler ce code, nous obtiendrions ceci :

```
Societaire.java:22: No constructor matching Personne() found in class Personne.
  public Societaire(String lenom, String leprenom, int lannee, int lacotis) {
         ^
1 error
```

Le compilateur va rechercher un constructeur `Personne()` qu'il ne trouve pas. Il y a deux possibilités :

1. Nous corrigeons le code.
2. Nous introduisons un constructeur `Personne()` avec d'éventuelles initialisations pour les variables `nom`, `prenom` et `annee`, et vraisemblablement un constructeur `Societaire()` sans paramètres.

Durant la conception de ce type de classe, ce n'est certainement pas nécessaire d'implémenter cette dernière solution, car une personne sans identité est peu vraisemblable dans une application traditionnelle. Une vue globale du problème est essentielle. Pour une application de base de données où une classe `Personne` serait associée à des coureurs d'un marathon, nous pourrions imaginer une réservation des petits numéros de dossard pour des invités de marque. Il faudrait alors se poser la question de savoir comment positionner et initialiser ces numéros. Nous devrions certainement créer des objets sans identité, où un constructeur par défaut pourrait être nécessaire !

En faisant le même exercice en C++ avec :

```
Societaire::Societaire(string lenom, string leprenom, …, int lacotis) {
  cotisation = lacotis;
  cout << "Constructeur de Societaire" << endl;
}
```

au lieu de :

```
Societaire::Societaire(string lenom, string leprenom, …, int lacotis)
           :Personne(lenom, leprenom, lannee) {
  cotisation = lacotis;
  cout << "Constructeur de Societaire" << endl;
}
```

nous aurions aussi une surprise très similaire à Java :

```
Societaire.cpp:48:
  no matching function for call to `Personne::Personne ()'
Societaire.cpp:34: candidates are:
Personne::Personne(basic_string<char,string_char_traits<char…
Societaire.cpp:19:  Personne::Personne(const Personne &)
```

où la dernière erreur est sans doute plus intéressante et nous indique qu'il manque un constructeur de copie. En fait, nous reviendrions sur la réflexion précédente et finirions par initialiser l'objet de la classe de base correctement.

Combiner héritage et composition

Dans le premier exercice de cette partie qui va suivre, nous combinerons héritage et composition. Pour ce faire, nous devrons aussi connaître en C++ cette construction :

```
class X2 : public X1 {
  Y y;
  X2(int i, int j) : X1(i), y(j) {}
}
```

La classe `X2` hérite de `X1`. Le constructeur de `X2` reçoit deux paramètres, un pour initialiser un attribut de la classe de base `X1`, et un autre pour l'attribut `Y`. Il faut noter le `y` minuscule du `y(j)`. Cette variable est un membre de la classe `X2` et non un appel à un constructeur ou une quelconque méthode de la classe de base `X1`.

Accès public, privé ou protégé

C'est vraiment le moment idéal pour introduire la directive `protected`, car nous connaissons déjà les `public` et `private`. Pour ce faire, avant d'illustrer les différents cas par deux exemples en Java et C++ sur le polymorphisme, nous allons reprendre la définition des directives pour la protection des accès :

- **public** (publique), c'est-à-dire visible pour toutes les classes ;
- **private** (privée), c'est-à-dire visible uniquement par la classe ;
- **protected** (protégée), c'est-à-dire visible par les classes dérivées et packages en Java.

Note

L'auteur profite de l'occasion pour expliquer l'approche choisie ici dans la présentation. Un ouvrage complet présentant dans un seul de ces deux langages à la fois l'héritage et le polymorphisme aurait besoin d'une cinquantaine de pages ! Le lecteur comprendra sans doute cette approche plus directe au travers d'exemples qui apportent plusieurs nouveaux concepts. Le plus important est de couvrir les sujets essentiels afin d'appréhender ces deux langages de la manière la plus rapide.

Le polymorphisme

Une classe qui permet l'interface à diverses autres classes est appelée polymorphe. Ce sont des classes dérivées d'une classe de base qui elle-même contient des définitions de méthodes qui seront en fait codées dans les sous-classes. Si nous possédions une collection de formes graphiques de différents types comme des rectangles ou des cercles, et que nous aimions appliquer une méthode `dessine()` sur chacun de ces objets sans connaître le type de cet objet, nous aurions alors besoin d'un mécanisme particulier.

Dans les deux exemples en Java et C++ qui suivent, nous allons examiner ce concept de polymorphisme et analyser son implémentation. Nous allons rencontrer les deux termes de redéfinition et de surcharge d'une méthode définie dans une classe de base.

Tout le monde connaît bien ces petits hommes bleus appelés les Schtroumpfs et plus particulièrement le Grand Schtroumpf, la Schtroumpfette ou le Schtroumpf Grognon. Nous allons donc schtroumpfer une classe de base nommée `SchtroumpfGeneric1`, de laquelle nous schtroumpferons une sous-classe nommée `SchtroumpfGrognon1`.

Les Schtroumpfs en Java

Nous commencerons par la version Java :

```
class SchtroumpfGeneric1 {
  private   String nom;
  private   int compteur1 = 0;

  protected int compteur2 = 0;

  public SchtroumpfGeneric1(String lenom) {
    nom = lenom;
 }
 public String getNom() {
   return nom;
 }

 public void parle(String texte) {
   System.out.println(getNom() + " dit " + texte);

   compteur1++;
   compteur2++;
 }

 public int getCompteurs() {
   return (100 * compteur1) + compteur2;
 }
}

public class SchtroumpfGrognon1 extends SchtroumpfGeneric1 {
  public SchtroumpfGrognon1(String lenom) {
    super(lenom);
  }

  public void parle(String texte) {
    System.out.println(getNom() + " dit en grognant " + texte);
    // compteur1++;  impossible car private
    compteur2++;
  }

  public static void unStroumpfParle(SchtroumpfGeneric1 unStroumpf, String texte) {
    unStroumpf.parle(texte);
  }

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

```
        SchtroumpfGeneric1 unBleu = new SchtroumpfGrognon1("Petit Grognon");
        unBleu.parle("Bonjour");
        unStroumpfParle(unBleu, "Salut");
        System.out.println("Compteurs: " + unBleu.getCompteurs());
    }
}
```

Si nous l'exécutons, nous « schroumpferons » alors le résultat suivant :

```
Petit Grognon dit en grognant Bonjour
Petit Grognon dit en grognant Salut
Compteurs: 2
```

La raison de définir la variable `unBleu` comme instance de la classe `SchtroumpfGeneric1` est, en fait, l'analyse de la méthode `parle()` sur cet objet. Comme pour des formes, nous pourrions avoir une collection de rectangles ou de cercles que nous aimerions dessiner. Nous aurions alors une collection de formes, que nous pourrions dessiner sans connaître le type de l'objet, c'est-à-dire en utilisant le polymorphisme. Ici, nous n'allons pas essayer de faire parler plusieurs Schtroumpfs, car nous savons que ce serait la pagaille !

La méthode :

```
public static void unStroumpfParle(StroumpfGeneric1 unStroumpf, String texte)
```

pourrait nous sembler étrange. Le `static` est correct, car aucune ressource d'un objet de classe `StroumpfGrognon1` n'est utilisée. Sa vraie place pourrait être dans la classe `StroumpfGeneric1`, mais en fait l'argument `StroumpfGeneric1 unStroumpf` peut se trouver comme paramètre de n'importe quelle méthode ou de n'importe quel constructeur d'une classe qui veut traiter une référence à un `StroumpfGeneric1`.

Le résultat est donc attendu. Nous faisons bien parler notre Schtroumf Grognon deux fois. La méthode `parle()` de la classe de base `StroumpfGeneric1` n'est donc jamais appelée.

Les deux variables `nom` et `compteur1` sont privées (`private`). Il n'est alors pas possible à la classe `SchtroumpfGrognon1` d'accéder directement à ces variables, et il faut donc des méthodes pour y accéder. La variable `compteur2` est protégée (`protected`) et ainsi accessible à toute classe dérivée de `StroumpfGeneric1`. Si nous avions voulu que `compteur2` n'ait une signification que pour `StroumpfGrognon1`, nous l'aurions déplacée dans cette dernière et définie `private` ou éventuellement `protected`, si nous avons l'intention d'étendre encore la hiérarchie de ces hommes bleus. Le résultat `2`, retourné par la méthode `getCompteurs()`, est attendu. La multiplication n'est ici qu'à des fins de présentation, car un Schtroumpf Grognon qui va parler plus de cent fois est du domaine du plausible.

Les Schtroumpfs en C++

Cette partie va nous étonner et nous faire plonger dans les profondeurs des difficultés du langage C++. L'héritage en Java est autrement plus simple et direct.

```
// SchtroumpfGrognon1.cpp
#include <iostream>
#include <string>
```

```
using namespace std;

class SchtroumpfGeneric1 {
  private:
    string nom;
    int compteur1;

  protected:
    int compteur2;

  public:
    SchtroumpfGeneric1(string lenom);
    string getNom();
    void parle(string texte);
    int  getCompteurs();
    static void unStroumpfParle(const SchtroumpfGeneric1 *unStroumpf, string texte);
};

class SchtroumpfGrognon1 : public SchtroumpfGeneric1 {
  public:
    SchtroumpfGrognon1(string lenom);
    void parle(string texte);
};

SchtroumpfGeneric1::SchtroumpfGeneric1(string lenom)
        :nom(lenom), compteur1(0), compteur2(0) {
}
string SchtroumpfGeneric1::getNom() {
  return nom;
}

void SchtroumpfGeneric1::parle(string texte) {
  cout << getNom() << " dit " << texte << endl;
  compteur1++;
  compteur2++;
}

int SchtroumpfGeneric1::getCompteurs() {
  return (100 * compteur1) + compteur2;
}

SchtroumpfGrognon1::SchtroumpfGrognon1(string lenom)
                   :SchtroumpfGeneric1(lenom) {
}

void SchtroumpfGrognon1::parle(string texte) {
  cout << getNom() << " dit en grognant " << texte << endl;
  // compteur1++;  impossible car private
  compteur2++;
}
```

```
void unStroumpfParle(SchtroumpfGeneric1 *unStroumpf, const string texte) {
  unStroumpf->parle(texte);
}

int main() {
  SchtroumpfGrognon1 *un_bleu;
  un_bleu =  new SchtroumpfGrognon1("Petit Grognon");
  un_bleu->parle("Bonjour");
  unStroumpfParle(un_bleu, "Salut");
  cout << "Les compteurs: " << un_bleu->getCompteurs() << endl;
  delete un_bleu;
  return 0;
}
```

Le résultat sera différent :

```
Petit Grognon dit en grognant Bonjour
Petit Grognon dit Salut
Les compteurs: 102
```

Est-ce vraiment ce que nous voulions ? Pas nécessairement ! Si nous avons redéfini la fonction dans la classe dérivée, c'est que nous voulions en fait la surcharger ! Il nous faut donc un mécanisme qui nous permette de forcer le compilateur, afin qu'il exécute la bonne méthode. Il faut noter ici, et c'est aussi applicable en Java, que la redéfinition ou la surcharge s'applique à des méthodes qui ont la même signature et le même type de retour. En ce qui concerne les constructeurs, les destructeurs en C++ et l'opérateur = en C++, il n'y a pas de mécanisme d'héritage comme cela s'applique aux méthodes.

Avant de passer à la solution de ce problème, nous allons examiner le code de plus près pour remarquer la manière d'initialiser les attributs de classes. Si nous faisions comme en Java :

```
 8:        class SchtroumpfGeneric1 {
 9:          private:
10:            string nom;
11:            int compteur1 = 1;
```

nous aurions alors l'erreur de compilation suivante :

```
SchtroumpfGrognon1.cpp:11: ANSI C++ forbids initialization of
                                          member `compteur1'
SchtroumpfGrognon1.cpp:11: making `compteur1' static
SchtroumpfGrognon1.cpp:11: ANSI C++ forbids in-class
        initialization of non-const static member `compteur1'
SchtroumpfGrognon1.cpp:
In method `SchtroumpfGeneric1::SchtroumpfGeneric1(basic_string<char,string_char
➥_traits<char>,__default_alloc_template<false,0> >)':
SchtroumpfGrognon1.cpp:33:
    field `int SchtroumpfGeneric1::compteur1' is static; only
                  point of initialization is its declaration
```

où une affectation n'est possible que pour des variables statiques.

Ceci pour indiquer que l'initialisation des types de base est essentielle, car nous aurions alors tendance à supprimer le « `= 1` » sans penser trop loin. Si nous avions omis `compteur2(0)` dans le constructeur de `SchtroumpfGeneric1`, nous aurions pu recevoir un `5375074` comme résultat. Sur une machine Sparc sous Solaris, nous avons constaté que la variable était initialisée à `0` par le compilateur. Lors d'une réutilisation du code sur une autre machine, il est donc essentiel à la fois d'initialiser toutes les variables et de prévoir les tests correspondants (ici vérifier la méthode `getCompteurs()` avant d'avoir laissé un Schtroumpf, ce qui est loin d'être évident).

Le virtual en C++

Le mot-clé `virtual` attaché à une méthode d'une classe de base va garantir que la bonne version de la méthode va être appelée, c'est-à-dire garantir le polymorphisme qui se fait automatiquement en Java. Il ne servirait à rien d'appliquer la méthode `dessine()` à la classe `Forme`, qui ne saurait dessiner un rectangle si l'objet était un `Rectangle`. La seule chose que pourrait connaître la classe `Forme` serait sans doute la position du rectangle, comme tous autres objets, rien de plus.

Nous allons donc retourner à nos petits hommes bleus avec une fonction ici déclarée virtuelle :

```
// SchtroumpfGrognon2.cpp
#include <iostream>
#include <string>

using namespace std;

class SchtroumpfGeneric2 {
  protected:
    string nom;

  public:
    SchtroumpfGeneric2(string lenom);
    virtual void parle(string texte);
    static void unStroumpfParle(SchtroumpfGeneric2 *unStroumpf, const string texte) {
        unStroumpf->parle(texte);
    };
};

class SchtroumpfGrognon2 : public SchtroumpfGeneric2 {
  public:
    SchtroumpfGrognon2(string lenom);
    void parle(string texte);
};

SchtroumpfGeneric2::SchtroumpfGeneric2(string lenom)
        :nom(lenom) {
```

```
}

void SchtroumpfGeneric2::parle(string texte) {
  cout << nom << " dit " << texte << endl;
}

SchtroumpfGrognon2::SchtroumpfGrognon2(string lenom)
                   :SchtroumpfGeneric2(lenom) {
}
void SchtroumpfGrognon2::parle(string texte) {
  cout << nom << " dit en grognant " << texte << endl;
}

int main() {
  SchtroumpfGrognon2 *un_bleu;
  un_bleu =  new SchtroumpfGrognon2("Gros Grognon");
  un_bleu->parle("Bonsoir");
  SchtroumpfGeneric2::unStroumpfParle(un_bleu, "Bonne nuit");
  delete un_bleu;
  return 0;
}
```

Et le résultat tant espéré :

```
Gros Grognon dit en grognant Bonsoir
Gros Grognon dit en grognant Bonne nuit
```

Il faut noter ici que la méthode `parle()` dans la classe `SchtroumpfGrognon2` pourrait très bien être laissée de côté, même si le mot-clé `virtual` a été utilisé. Il y a certainement un nombre de variétés de Schtroumpfs qui parlent normalement et qui peuvent utiliser la méthode générique sans rechigner ou chanter une chanson.

Il faudra noter l'utilisation simplifiée de la variable `nom` au travers d'un accès `protected` et la forme `SchtroumpfGeneric2::unStroumpfParle()`, car nous avons ici défini cette méthode statique.

Nous pourrions aussi appeler la méthode `parle()` de la classe de base :

```
void SchtroumpfGrognon2::parle(string texte) {
  cout << nom << " dit en grognant " << texte << endl;
  SchtroumpfGeneric2::parle(texte);
}
```

et nous obtiendrions :

```
Gros Grognon dit en grognant Bonsoir
Gros Grognon dit Bonsoir
Gros Grognon dit en grognant Bonne nuit
Gros Grognon dit Bonne nuit
```

Le `Stroumpfgeneric2::parle()` n'a pas trop de sens ici. Il ne va pas parler deux fois même s'il est grognon ou sympa. Nous pourrions toujours trouver un Schtroumpf bégayeur, qui aurait un comportement encore différent ! Cependant, si nous voulions par exemple

imprimer des attributs privés à la classe de base pour des raisons de test, il serait tout à fait plausible d'appeler la méthode de base pour accéder à ces informations. Nous pourrions aussi donner un autre nom ou définir une autre signature à la méthode.

Le passage aux classes abstraites est donc tout à fait naturel. Si nous reprenons notre classe `Forme`, nous pourrons définir dans cette dernière une méthode `dessine()`. Cependant, la classe `Forme` ne saura pas dessiner et devra déléguer la responsabilité totale aux sous-classes comme `Cercle` ou `Rectangle`.

Les classes abstraites en C++

Si nous définissons notre classe `Personne` abstraite, nous pourrions penser que nous avons un problème puisqu'une personne, en principe, est loin d'être un objet abstrait car elle possède au moins un nom, un prénom et un âge. Pour une classe `Forme`, à partir de laquelle nous allons créer d'autres classes comme `Rectangle` ou `Cercle`, c'est un peu différent, un peu plus abstrait. Une forme reste un objet, comme le myope qui aurait de la peine à différencier un cercle d'un rectangle.

La définition d'une classe abstraite en C++ ou Java est utilisée en fait uniquement pour indiquer qu'elle regroupe un certain nombre de fonctionnalités communes. Ces dernières seront utilisées dans des sous-classes qui vont hériter de ces caractéristiques. Si nous déclarons notre classe `Personne` abstraite, nous ne pourrons plus créer d'instances de cette classe, mais seulement de ses dérivées. Cependant, les méthodes resteront accessibles au travers de nouvelles classes comme `Societaire` ou `Employee`.

Voyons à présent comment nous pourrions définir notre classe `Schtroumpf` abstraite.

Fonction purement virtuelle en C++

Nous avons vu précédemment le mot-clé `virtual` utilisé de cette manière :

```
virtual void parle(string texte);
```

Nous allons à présent indiquer une forme en C++ qui au premier abord va nous paraître très étrange :

```
virtual void parle(string texte) = 0;
```

Celle-ci nous indique que la méthode est virtuelle pure et doit être absolument implémentée dans la classe dérivée. Voici donc le code complet de la classe `SchtroumpfGrognon3`, qui utilise la classe abstraite `Schtroumpf` :

```
// SchtroumpfGrognon3.cpp
#include <iostream>
#include <string>

using namespace std;

class Schtroumpf {
```

```
  protected:
    string nom;

  public:
    Schtroumpf(string lenom) : nom(lenom) {};
    virtual void parle(string texte) = 0;
    static void unStroumpfParle(Schtroumpf *unStroumpf, const string texte) {
        unStroumpf->parle(texte);
    };
};

class SchtroumpfGrognon3 : public Schtroumpf {
  public:
    SchtroumpfGrognon3(string lenom) : Schtroumpf(lenom) {} ;
    void parle(string texte);
};

void SchtroumpfGrognon3::parle(string texte) {
  cout << nom << " dit en grognant " << texte << endl;
}

int main() {
  Schtroumpf *un_bleu = new SchtroumpfGrognon3("Un Gros Grognon");
  un_bleu->parle("Bonsoir");
  Schtroumpf::unStroumpfParle(un_bleu, "Bonne journée");
  delete un_bleu;
  return 0; }
```

Il n'y a pas beaucoup de différence avec la forme précédente, si ce n'est que nous avons remanié les deux classes afin d'avoir plus de code en ligne dans la définition. La forme :

```
Schtroumpf *un_bleu = new SchtroumpfGrognon3("Un Gros Grognon");
```

était déjà possible dans la version précédente, et il est important de la mentionner, car nous pourrions par exemple définir une collection de Schtroumpfs du même type avec différents noms. La méthode :

```
void Schtroumpf::parle(string texte) { ..
```

n'a pas besoin d'être définie, car il n'est pas possible de créer directement une instance de la classe `Schtroumpf`. Si nous écrivions ce code :

```
Schtroumpf unpetit = new Schtroumpf("Petit Schtroumpf");
```

nous aurions alors une erreur de compilation :

```
SchtroumpfGrognon3.cpp: In function `int main()':
SchtroumpfGrognon3.cpp:40: cannot declare variable `unpetit' to be
                           of type `Schtroumpf'
SchtroumpfGrognon3.cpp:40: since the following virtual functions are
                           abstract:
SchtroumpfGrognon3.cpp:14: void Schroumpf::parle(basic_string<char,
                         string_char_traits<char>,
```

```
                          __default_alloc_template<false,0> >)
SchtroumpfGrognon3.cpp:40: cannot allocate an object of type `Schtroumpf'
SchtroumpfGrognon3.cpp:40: since type `Schtroumpf' has abstract virtual
                           functions
```

Il nous faut donc définir une sous-classe pour tous les types possibles de Schtroumpf. Si cette solution est vraiment trop lourde, il nous faudrait revenir à une classe `Schtroumpf` générique. Une classe abstraite en C++ est en fait une définition d'interface pour ces classes dérivées.

Destructeur virtuel en C++

Un des aspects essentiels en programmation C++ est la compréhension et l'utilisation correcte des destructeurs virtuels. Si nous écrivons ce morceau de code :

```
// Virtueldestr.cpp
#include <iostream>

using namespace std;

class Mabase {
  private:
    char *preservea;

  public:
    Mabase() {
       preservea = new char[100];
       cout << "preserva alloué" << endl;
    }
    ~Mabase() {
       delete[] preservea;
       cout << "preserva effacé" << endl;
    }
};

class Maderivee : public Mabase {
  private:
    char *preserveb;

  public:
    Maderivee() {
       preserveb = new char[100];
       cout << "preservb alloué" << endl;
    }

    ~Maderivee() {
       delete[] preserveb;
       cout << "preservb effacé" << endl;
    }
};
```

```
int main() {
  Maderivee *pobjet1 = new Maderivee();
  delete pobjet1;
  cout << endl;
  Mabase *pobjet2 = new Maderivee();
  delete pobjet2;
}
```

et que nous regardons le résultat obtenu :

```
preserva alloué
preservb alloué
preservb effacé
preserva effacé
preserva alloué
preservb alloué
preserva effacé
```

nous voyons immédiatement un problème : il manque un `preservb effacé` ! Une première remarque doit être faite pour le constructeur de `Maderivee`. Sa première action est d'appeler le constructeur de `Mabase`, qui va allouer `preservea`, et ceci avant de faire de même pour `preserveb`. Les attributs `preservea` et `preserveb` sont de simples tampons dynamiques de 100 octets pour un usage divers dans les deux classes. Il faut absolument les effacer avec les destructeurs. Nous croyons bien faire avec un destructeur correctement codé pour chaque classe. Le problème viendra lors de ce type de construction :

```
Mabase *pobjet2 = new Maderivee();
```

`pobjet2` est un objet `Mabase` contenant en fait un objet de `Maderivee`. Ensuite, lorsque le destructeur sera appelé au travers du `delete pobjet2`, il ne saura exécuter le code désiré. Pour nous en sortir, il faudra ajouter le mot-clé `virtual` pour le destructeur de la classe de base :

```
virtual ~Mabase() {
```

Nous pouvons modifier le code ci-dessus avec cette ligne de code et vérifier que cette fois le mécanisme virtuel fonctionnera correctement et que toutes les ressources allouées auront été effacées.

Les classes abstraites en Java

Jusqu'à présent, nous avons beaucoup parlé de C++ et peu de Java. La raison en est simple : l'héritage de classes en Java est nettement simplifié. Nous n'avons pas ici de mot-clé `virtual`, avec une extension possible pure virtuelle, nous avons simplement celui d'`abstract`. Une méthode déclarée `abstract` indiquera que la classe ne sait pas comment implémenter le code et doit laisser ce travail à la classe dérivée. C'est le cas de `dessine()`, dans la classe `Forme`. Si l'objet est un cercle, seule la classe `Cercle`, qui possède son rayon comme attribut, saura dessiner cette forme.

Dès qu'une classe Java possède une méthode abstraite, la classe sera abstraite et devra être déclarée `abstract`. Nous pouvons aussi décider de déclarer une classe `abstract` même si elle n'a pas de méthode abstraite. Comme en C++, toute classe abstraite en Java ne peut être instanciée.

Nous avons déjà beaucoup parlé de la classe `Forme` et de ses classes dérivées `Cercle` et `Rectangle`. Voici donc à présent un exemple en Java, où nous commençons avec la classe abstraite `Forme` :

```
public abstract class Forme {
  protected int origineX;
  protected int origineY;

  public Forme(int posx, int posy) {
    origineX = posx;
    origineY = posy;
  }

  public abstract void dessine();
}
```

Elle est abstraite, car la méthode `dessine()` est déclarée `abstract` et ne possède pas de code. Elle correspond à une définition. Nous devrons donc la définir dans une sous-classe de `Forme`. Cela veut dire, aussi, qu'il n'est pas possible de définir une instance de cette classe abstraite. Nous définissons maintenant la classe `Rectangle` comme suit :

```
public class Rectangle extends Forme {
  protected int longueur;
  protected int hauteur;

  public Rectangle(int posx, int posy, int laLongueur, int laHauteur) {
    super(posx, posy);

    longueur = laLongueur;
    hauteur  = laHauteur;
  }

  public void dessine() {
    System.out.print("Notre rectangle à la position " + origineX + "," + origineY);
    System.out.println(" et de dimensions " + longueur + "," + hauteur + " est
    ➥dessiné!");
  }
}
```

Si nous n'avions pas défini la méthode `dessine()`, nous aurions alors reçu cette erreur :

```
Rectangle.java:1: class Rectangle must be declared abstract. It does not define void
➥dessine() from class Forme.
public class Rectangle extends Forme {
             ^
```

Cela peut signifier deux choses :

1. Nous avons oublié le code de `dessine()`. Ce qui est effectivement notre cas.
2. La classe `dessine()` sera définie plus loin dans une autre sous-classe de `Rectangle`, c'est-à-dire dans un troisième niveau de notre hiérarchie.

Comme pour notre classe `Rectangle`, nous pouvons définir une méthode `dessine()` pour la classe `Cercle` :

```
public class Cercle extends Forme {
  protected int rayon;

  public Cercle(int posx, int posy, int leRayon) {
    super(posx, posy);

    rayon = leRayon;
  }

  public void dessine() {
    System.out.print("Notre cercle à la position " + origineX + "," + origineY);
    System.out.println(" et de rayon " + rayon + " est dessiné!");
  }
}
```

Un cercle est vraiment différent, et ce ne serait pas le cas d'un carré, qui peut très bien utiliser la représentation de notre classe `Rectangle`. Nous pourrions très bien créer une nouvelle classe pour définir ce carré ou encore un nouveau constructeur pour la classe `Rectangle` de cette manière :

```
public Rectangle(int posx, int posy, int leCote) {
  super(posx, posy);

  longeur = leCote;
  largeur = leCote;
}
```

Il nous reste à présent à vérifier nos trois nouvelles classes :

```
public class Dessin {
  Forme[] lesObjets;

  public Dessin() {
    lesObjets    = new Forme[4];
    lesObjets[0] = new Rectangle(0,0,1,1);
    lesObjets[1] = new Rectangle(5,1,1,2);
    lesObjets[2] = new Cercle(1,5,1);
    lesObjets[3] = new Cercle(2,2,4);
  }

  public void dessine() {
    for (int i = 0; i < lesObjets.length; i++) {
      lesObjets[i].dessine();
```

```
    }
  }

  static public void main(String args[]) {
    Dessin unDessin = new Dessin();
    unDessin.dessine();
  }
}
```

La classe `Dessin` possède quatre objets, que nous dessinons en utilisant une collection de formes. Il faut toujours prendre des valeurs suffisamment différentes pour contrôler l'utilisation correcte des attributs de classe. Choisir des `int` pour les attributs de position et de grandeur pour les objets que nous aimerions dessiner n'aurait un sens que si nous travaillions en pixels.

Le résultat présenté :

```
Notre rectangle à la position 0,0 et de dimension 1,1 est dessiné!
Notre rectangle à la position 5,1 et de dimension 1,2 est dessiné!
Notre cercle à la position 1,5 et de rayon 1 est dessiné!
Notre cercle à la position 2,2 et de rayon 4 est dessiné!
```

correspond tout à fait à notre attente.

Le transtypage (casting) d'objet

Lorsqu'une méthode existe dans une sous-classe et s'avère spécifique à celle-ci, nous ne pouvons l'appliquer à un objet de sa classe de base sans tester la validité de son existence au moyen d'un transtypage.

Le transtypage en Java

Nous reprendrons l'exemple ci-dessus avec une méthode `dessineTexte()` dans notre classe `Rectangle`, méthode qui n'existe pas dans la classe `Cercle`. Nous trouverons sur le CD-Rom d'accompagnement, dans les classes `Dessin2` et `Rectangle2`, le code qui correspond à ce cas précis. Dans la classe de test `Dessin2`, nous utiliserons à présent des objets `Rectangle2` en lieu et place de `Rectangle`. Dans la classe `Rectangle2`, nous avons ajouté le code suivant :

```
public void dessineTexte() {
  System.out.println("Notre rectangle contient du texte à l'intérieur");
}
```

Dans la classe `Dessin2`, si nous compilions le code :

```
public void dessine() {
  for (int i = 0; i < lesObjets.length; i++) {
    lesObjets[i].dessine();
    lesObjets[i].dessineTexte();
  }
}
```

nous obtiendrions une erreur car la méthode `dessineTexte()` n'existe pas dans la classe `Forme`. Le seul moyen est d'appliquer un transtypage, que nous pourrions faire de cette manière :

```
public void dessine() {
  for (int i = 0; i < lesObjets.length; i++) {
    lesObjets[i].dessine();

    Rectangle2 rec = (Rectangle2)lesObjets[i];
    rec.dessineTexte();
  }
}
```

Si nous exécutons ce code, nous obtiendrons cette erreur :

```
Exception in thread "main" java.lang.ClassCastException: Cercle
        at Dessin2.dessine(Compiled Code)
        at Dessin2.main(Compiled Code)
```

Il est en effet impossible de prendre un rectangle pour un cercle et, en plus, d'essayer de lui appliquer une méthode `dessineTexte()` qui n'est admissible que pour des objets de la classe `Rectangle`. Nous sommes donc contraints d'utiliser l'opérateur `instanceof`, qui va nous permettre d'identifier la classe avant d'y appliquer la méthode `dessineTexte()`.

```
public void dessine() {
  for (int i = 0; i < lesObjets.length; i++) {
    lesObjets[i].dessine();

    if (lesObjets[i] instanceof Rectangle2) {
      Rectangle2 rec = (Rectangle2)lesObjets[i];
      rec.dessineTexte();
    }
  }
}
```

Et le résultat attendu nous sera présenté si nous exécutons la classe de test `Dessin2` :

```
Notre rectangle à la position 0,0 et de dimension 1,1 est dessiné!
Notre rectangle contient du texte à l'intérieur
Notre rectangle à la position 5,1 et de dimension 1,2 est dessiné!
Notre rectangle contient du texte à l'intérieur
Notre cercle à la position 1,5 et de rayon 1 est dessiné!
Notre cercle à la position 2,2 et de rayon 4 est dessiné!
```

Comment éviter le transtypage

C'est effectivement une grande question, après avoir examiné le code précédent. Il y a parfois des cas où cela se présente mal comme lorsque des modifications de dernière minute doivent être apportées au code et, dans ce cas précis, dans une classe dérivée. Il faudrait en fait revenir à la conception des classes de base et de la hiérarchie. Nous aurions dû en fait concevoir une méthode abstraite `dessineTexte()` dans la classe de base

Forme. Nous aurions pu alors activer dessineTexte() sur n'importe quelle sous-classe, qui aurait alors contenu une implémentation sans code de dessineTexte(). Une autre manière de faire aurait été de définir une méthode non abstraite dans Forme et de la surcharger dans la classe Rectangle. Cette dernière possibilité sera donnée en exercice.

Le transtypage en C++

Nous avons vu que lorsque nous avons une collection d'objets de différents types, il est parfois nécessaire d'appliquer un transtypage. Dans l'exemple qui suit, la méthode dirige() n'existe que pour la classe Directeur, alors que la classe de base Employe ne la possède pas. Dans une collection d'instance d'Employe, nous pourrions éliminer les directeurs, mais ce serait sans doute plus convenable de choisir ces derniers pour les faire diriger l'entreprise. Voici donc le code de ces deux classes avec le programme de test associé :

```
// Entreprise.cpp
#include <iostream>
#include <string>

using namespace std;

class Employe {
  protected:
    string nom;

  public:
    Employe(const string le_nom) : nom(le_nom) {};
    virtual string get_nom() { return nom; };
};

class Directeur : public Employe {
  public:
    Directeur(const string le_nom) : Employe(le_nom) { };

    void dirige() {
      cout << "Le directeur " << nom << " dirige !" << endl;
    }
};

int main() {
  Employe *lentreprise[3];
  lentreprise[0] = new Employe("employé1");
  lentreprise[1] = new Employe("employé2");
  lentreprise[2] = new Directeur("directeur1");

  for (int i = 0; i < 3; i++) {
    cout << "Le nom: " << lentreprise[i]->get_nom() << endl;
  }

  for (int i = 0; i < 3; i++) {
```

```
    Directeur *direc = dynamic_cast<Directeur*>(lentreprise[i]);
    if (direc != NULL) {
      direc->dirige();
    }
  }

  for (int i = 0; i < 3; i++) {
    delete lentreprise[i];
  }
}
```

Le nom de l'employé, en considérant qu'un directeur est aussi un employé, est conservé dans la classe de base `Employe`. La méthode virtuelle `get_nom()` est ici juste pour sortir plus tard la liste des employés. Cependant, le mot-clé `virtual` est essentiel pour déclencher ce mécanisme d'identification (RTTI, *Run-Time Type Identification*). Si nous ne déclarons pas au moins une méthode virtuelle, la ligne de code :

```
Directeur *direc = dynamic_cast<Directeur*>(lentreprise[i]);
```

ne compilerait simplement pas.

Nous avons défini une liste de trois employés qui se trouve dans la variable `lentreprise`. Il ne faudra d'ailleurs pas oublier d'effacer ces trois objets, comme nous l'avons fait en fin de programme ou lorsqu'ils ne sont plus utilisés. Nous avons donc une liste d'employés avec deux objets de la classe `Employe` et un de la classe `Directeur`. Lorsque nous traversons la liste, nous n'avons pas de problèmes avec la méthode `get_nom()` car elle est définie dans la classe de base. Il en va tout autrement pour la méthode `dirige()`, qui n'est à disposition que dans la classe Directeur. C'est ici que notre construction `dynamic_cast<Directeur*>` va nous permettre d'obtenir le transtypage si celui-ci est possible. En Java nous avions besoin du `instanceof`, alors qu'ici un simple contrôle sur un pointeur `null` sera suffisant.

Le résultat sera présenté ainsi :

```
Le nom: employé1
Le nom: employé2
Le nom: directeur1
Le directeur directeur1 dirige !
```

Comme ce fut le cas de l'exemple en Java, nous pourrions nous poser toute une série de questions, comme celle de savoir si notre classe `Employe` devrait être abstraite et définir par exemple une sous-classe `Ouvrier`. Dans une entreprise, un directeur peut être absent et un des ouvriers pourrait être nommé remplaçant. Une bonne conception de ses classes et de leurs hiérarchies est essentielle, surtout lorsque nous pourrions éviter de telles constructions comme notre `dynamic_cast<…>`.

L'héritage en Java et en C++ : les différences

Nous allons récapituler, dans ce domaine, les principales différences entre Java et C++ :

1. En Java, toutes les méthodes sont virtuelles par défaut. Elles s'exécutent donc plus lentement.
2. En C++, il faut utiliser `virtual` pour déclarer qu'une méthode est virtuelle, c'est-à-dire pour qu'elle soit définie dans une classe dérivée. En Java, il faut utiliser la directive `abstract` pour la méthode.
3. En déclarant `final` une méthode en Java, celle-ci ne pourra plus être surchargée. Une classe déclarée `final` aura toutes ses méthodes définies `final`. Les méthodes Java `final` peuvent être alors remplacées par du code en ligne et gagner en performance.
4. En C++, si nous voulons que le compilateur remplace le code pour améliorer les performances, il est nécessaire d'utiliser la directive `inline`.

Au chapitre 15, nous reviendrons sur le thème de la performance. Quels sont donc ces facteurs de performance ? Nous serons très surpris !

Résumé

Ce chapitre est également très important car il couvre un certain nombre d'aspects essentiels de la programmation objet en Java et C++. Si la manière Java est nettement simplifiée, le lecteur comprendra aisément que cette partie est l'un des aspects les plus complexes du langage C++.

Exercices

1. Pour montrer les deux approches de composition et d'héritage, écrire deux classes de base `Base1` et `Base2` qui chacune contient un `String` et un `int` comme attributs. Écrire une classe `DoubleBase` qui hérite de `Base1` et qui est composée d'un objet de `Base2` avec en plus un `String` et un `int`. Le constructeur de `DoubleBase` doit recevoir six paramètres afin d'initialiser ces trois couples d'attributs. Tracer les constructeurs avec un message indicatif sur la console.
2. Le président (classe dérivée) d'une société doit faire un discours à l'assemblée, mais comme tous les autres sociétaires (classe de base) il doit aussi faire un rapport. Écrire ces deux classes avec une méthode `parle()` qui va refléter ces deux parties. Les textes des discours et rapports sont passés dans les constructeurs. En C++, nous écrirons un fichier d'en-tête séparé.
3. Pour éviter le transtypage, reprendre la classe `Dessin2` et définir la méthode `dessineTexte()` non abstraite dans la classe de base `Forme`. Inclure toutes les classes dans un fichier commun `Dessin3.java`.

13
Des héritages multiples

Héritage multiple en C++

Nous ne donnerons ici qu'un aperçu rapide et un exemple simple. Nous pensons pouvoir déconseiller, d'une manière générale, l'emploi de l'héritage multiple en C++, car il se révèle beaucoup trop complexe et difficile à maîtriser. Tout au long de cet ouvrage, nous avons demandé au programmeur de procéder à une écriture simple de son code. L'utilisation de l'héritage multiple ne va pas dans ce sens.

Pour la présentation de l'héritage multiple, nous allons adopter une démarche tout à fait particulière. Pour la première et dernière fois dans cet ouvrage, nous allons utiliser une notation abstraite. Jusqu'à présent, nous avions toujours travaillé avec des objets ou des sujets bien réels et concrets, comme des personnes ou des formes à dessiner. Dans la plupart des ouvrages théoriques ou de référence pour les compilateurs, et dans les articles spécialisés, nous rencontrons ce type de notation totalement abstraite, auquel il faut aussi se familiariser et s'habituer : c'est ici ce que nous avons choisi. Nous avons cependant gardé quelques objets de la classe `string`, afin que cela ne devienne pas illisible ni, surtout, impossible à compiler sans la déclaration complète de tous les objets utilisés dans le code !

Soit deux classes `A` et `B` définies comme suit, ainsi que la classe `C` qui hérite de ces deux classes :

```
// multiherit.cpp
#include <iostream>
#include <string>

using namespace std;

class A {
```

```
  private:
    string attributA;

  public:
    A(string unAttribut) {
      attributA = unAttribut;
    }

    string getAttribut() {
      return attributA;
    }

    virtual string getInfo() {
      return "classe A";
    }
};


class B {
  private:
    string attributB;

  public:
    B(string unAttribut) {
      attributB = unAttribut;
    }

    string getAttributB() {
      return attributB;
    }

    virtual string getInfo() {
      return "classe B";
    }
};



class C: public A, public B {
  private:
    string attributC;

  public:
    C(string unAttribut1, string unAttribut2, string unAttribut3);

    string getAttribut();

    virtual string getInfo();
};
```

```
C::C(string unAttribut1, string unAttribut2, string unAttribut3)
  :A(unAttribut1), B(unAttribut2) {
    attributC = unAttribut3;
}

string C::getAttribut() {
  return attributC + " " + A::getAttribut() + " " + getAttributB();
}


string C::getInfo() {
  return "classe C hérite de " + A::getInfo() + " et de " + B::getInfo() +
         " (" + attributC + " " + A::getAttribut() + " " + getAttributB() + ")";
}


int main() {
  C objet1("A1", "B1", "C1");                // objet de la classe C
  cout << objet1.getAttribut() << endl;
  cout << objet1.getInfo() << endl;

  A *objet2 = new C("A2", "B2", "C2");       // objet de la classe A
  cout << objet2->getAttribut() << endl;
  cout << objet2->getInfo() << endl;
  delete objet2;
}
```

Les classes `A` et `B` n'ont rien de particulier au niveau de leurs constructeurs et de leurs attributs. Chaque objet de ces deux classes aura un attribut `string` initialisé par le constructeur, `attributA` pour la classe `A` et `attributB` pour la classe `B`. Nous avons défini volontairement trois méthodes publiques différentes :

```
string getAttribut()       // classe A seulement
string getAttributB()      // classe B seulement
virtual string getInfo()   // classes A et B
```

La classe `C` possède une structure identique aux classes `A` et `B` avec son attribut `attributC`, mais, en plus, hérite des deux classes `A` et `B`. Il faut noter la forme du constructeur de la classe `C` et la manière d'initialiser les classes de base :

```
C::C(string unAttribut1, string unAttribut2, string unAttribut3)
  :A(unAttribut1), B(unAttribut2) {
    attributC = unAttribut3;
}
```

Une construction telle que :

```
C::C(string unAttribut1, string unAttribut2, string unAttribut3)
  :A(unAttribut1), B(unAttribut2), attributC(unAttribut3) {}
```

aurait aussi été acceptée.

La méthode `getAttributB()` de la classe `B` est unique et n'entraîne pas de confusion. Tout objet instancié de la classe `C` pourra donc l'utiliser sans difficulté. Pour `getAttribut()`, qui n'est pas virtuelle et qui existe dans la classe `C` et la classe `A`, c'est beaucoup moins clair. Le résultat présenté, provenant du code inclus dans le `main()` :

```
C1 A1 B1
classe C hérite de classe A et de classe B (C1 A1 B1)
A2
classe C hérite de classe A et de classe B (C2 A2 B2)
```

correspond-t-il à notre attente ? Nous n'avons pas essayé de dérouter l'attention du lecteur avec le `new` pour l'`objet2`, mais nous avons simplement voulu montrer une autre construction. Le résultat `A2` est bien correct. Nous avons un `getAttribut()` sur la classe `A`, alors que `getInfo()` est virtuel et s'applique bien sur la méthode de la classe `C`. Comme nous l'avons vu au chapitre précédent, dans lequel nous avons analysé en détail les aspects de transtypage pour l'héritage simple, nous avons les mêmes difficultés, avec ici encore plus d'ambiguïté.

Nous laisserons donc le lecteur juger par lui-même de l'utilisation possible de l'héritage multiple. Nous conseillerons, dans tous les cas, d'être très prudent lors de la définition du nom des méthodes, comme ici notre `getAttribut()`.

Héritage multiple en Java

Dans le langage C++, une classe peut hériter de plusieurs superclasses, alors qu'en Java ceci n'est pas possible. Une des raisons de cette restriction est la complexité de l'implémentation pour les compilateurs Java. La notion d'interface a été introduite en Java et elle permet de récupérer la plupart des fonctionnalités de l'héritage multiple.

Comme son nom l'indique, l'héritage multiple permet à une classe de refléter le comportement de deux ou de plusieurs parents. Nous nous souviendrons, au chapitre 12 précédent, de la présentation de la classe `java.lang.Integer`, qui était définie ainsi :

```
public final class Integer extends Number implements Comparable
```

À ce moment-là, nous avions laissé la partie `implements Comparable`, dont nous allons maintenant éclaircir le fonctionnement. D'une manière générale, nous pouvons dire que la classe `Integer` hérite à la fois de la classe `Number` et de l'interface `Comparable`. Nous rencontrons ici un nouveau terme, car la forme :

```
public final class Integer extends Number, Comparable
```

n'est pas possible en Java pour deux raisons :

1. L'héritage multiple de plusieurs classes n'existe pas (mais possible en C++).
2. `Comparable` n'est pas une classe, mais une interface.

Cette décision dans la conception de Java est en fait une simplification apportée au langage Java, et à notre avis cela passe très bien. Nous répéterons, encore une fois, qu'il

est tout à fait possible de définir des recommandations pour éviter ou même pour ne jamais utiliser l'héritage multiple en C++, héritage qui rend le code trop complexe. Malheureusement, le langage C++ n'a pas d'interface, bien qu'il possède d'autres atouts très solides, comme les incontournables surcharges d'opérateurs.

Définition d'une interface en Java

Nous pouvons définir une interface comme une classe totalement abstraite et sans code. Le terme ou mot-clé `implements`, dans la définition de la classe qui va utiliser cette interface, nous indique que le code de toutes les méthodes définies dans cette interface doit faire partie de cette classe, comme dans notre exemple ci-dessus avec la classe `Integer` et l'interface `Comparable`.

La classe `Integer` hérite de `Number`. Ainsi, toutes les méthodes publiques disponibles dans cette classe de base sont aussi utilisables, comme `byteValue()` ou `shortValue()`. Cependant, nous héritons aussi, cette fois-ci par une interface de `Comparable`, de la méthode `compareTo(Object o)`, la seule méthode définie ! C'est un drôle d'héritage, dans le sens que `compareTo()` doit être absolument codé. Nous dirons ici que nous avons affaire à un héritage conditionnel ; nous devrons en effet satisfaire les conditions définies par l'interface, c'est-à-dire écrire le code. Cette technique est utilisée pour forcer une implémentation et pour la réutiliser dans d'autres cas.

Si le lecteur a le courage, il peut s'amuser à consulter le code source de la classe `Integer`, qui fait partie du matériel distribué par Sun Microsystems. Le fichier `src.jar` contient le code source de toutes les classes de cette distribution. Il est possible, avec 7-Zip par exemple (logiciel Open Source sous Windows, voir annexe B), d'extraire le fichier `src/java/lang/Integer.java`, dans lequel nous pourrons découvrir ce code indispensable puisque provenant d'une interface :

```
public int compareTo(Object o) {
    return compareTo((Integer)o);
}
```

Ce code nous indique qu'une autre méthode publique de la classe `Integer` sera appelée :

```
public int compareTo(Integer anotherInteger) {
    int thisVal = this.value;
    int anotherVal = anotherInteger.value;
    return (thisVal<anotherVal ? -1 : (thisVal==anotherVal ? 0 : 1));
}
```

pour implémenter le code de comparaison de nos objets. Seul un objet de la classe `Integer` sera accepté, et la documentation, partie intégrante de ce code, nous indique bien l'exception `ClassCastException` qui pourrait être générée si nous essayions de comparer une instance d'`Integer` avec quelque chose d'autre. La double forme `?:` nous permet ici de combiner deux séries imbriquées d'instructions de condition `if` et `else`.

J'ai déjà hérité, que faire avec mon Thread ?

C'est une manière différente de poser le problème. Lorsque nous avons conçu un ensemble de classes dans une hiérarchie, il peut arriver que nous voulions hériter des fonctionnalités d'une autre classe. Pour illustrer ce problème, nous allons profiter de l'occasion pour présenter la classe `Thread` et son équivalent, défini comme interface, `Runnable`. La classe `Thread` permet d'introduire un mécanisme pour que plusieurs tâches puissent s'exécuter en même temps. Chaque opération va donc fonctionner en parallèle et indépendamment des autres. Notre première classe, `MonThread1`, va hériter de cette fameuse classe `Thread`.

```
public class MonThread1 extends Thread {
  public MonThread1(String nom) {
    super(nom);
  }

  public void run() {
    System.out.println("Commence pour " + getName());
    try {
      sleep(1000);
    }
    catch(InterruptedException ie) { }

    System.out.println("Termine pour " + getName());
  }

  public static void main (String[] args) {
      new MonThread1("Processus 1").start();
      System.out.println("Test 1");
      new MonThread1("Processus 2").start();
      System.out.println("Test 2");
  }
}
```

En exécutant ce programme, nous obtiendrons ce résultat :

```
Test 1
Commence pour Processus 1
Test 2
Commence pour Processus 2
Termine pour Processus 1
Termine pour Processus 2
```

Suivant le système d'exploitation et la machine virtuelle dans laquelle le programme est exécuté, il est tout à fait possible d'avoir `Test 2` juste après `Test 1`. Nous l'avons constaté sur une station Sparc de Sun Microsystems sous Solaris.

La méthode `run()` est en fait déjà définie dans la classe de base `Thread` et est ici surchargée, afin d'exécuter le code désiré. Le fait de ne pas avoir associé le résultat de `new MonThread1()` à une variable n'est pas interdit, car ici aucune autre méthode que `start()` ne sera associée plus loin à l'instance créée. La méthode `start()` va lancer un processus parallèle et indépendant du processus courant. Directement à l'exécution de `start()`, le

point d'entrée `run()` de cette même classe va être exécuté pour attendre 1 000 millisecondes avant d'imprimer le message `"Termine"`. Nous pouvons maintenant comprendre pourquoi le `"Commence"` sur le deuxième processus parallèle vient avant la terminaison du premier processus. La partie `main()` sera donc terminée bien avant la terminaison des deux autres processus dans le code de la méthode `run()`. La méthode `sleep()` est statique. Si nous voulons l'utiliser à d'autres occasions, il faudra choisir cette forme :

```
try {
  Thread.sleep(2000);    // pause de 2 secondes
}
catch(InterruptedException ie) { }
```

car ici, notre classe hérite de `Thread`, ce qui explique l'utilisation de `sleep()` sans sa partie gauche.

Une interface au lieu d'un héritage classique

Pour la classe `MonThread2`, nous allons utiliser l'interface `Runnable`, qui est similaire à la classe `Thread` :

```
public class MonThread2 implements Runnable {

  public void depart() {
    Thread t1 =  new Thread(this, "Processus 1");
    t1.start();
    System.out.println("Test 1");

    Thread t2 =  new Thread(this, "Processus 2");
    t2.start();
    System.out.println("Test 2");
  }

  public void run() {
    System.out.println("Commence pour " + Thread.currentThread().getName());
    try {
      Thread.sleep(1000);
    }
    catch(InterruptedException ie) { }

    System.out.println("Termine pour " + Thread.currentThread().getName());
  }

  public static void main (String[] args) {
    MonThread2 mt = new MonThread2();
    mt.depart();
  }
}
```

L'avantage de cette technique est que nous pourrions demander à `MonThread2` d'hériter d'une autre classe dont nous voudrions réutiliser l'implémentation. Si la classe `MonThread2` a

déjà hérité d'une classe et que nous voulions hériter d'une troisième classe, notre choix serait encore une fois une interface. Il est en effet possible d'hériter de plusieurs interfaces en les séparant avec des virgules (voir la classe `Clown1` en fin de chapitre).

Nous retrouvons à nouveau la méthode `run()`. La différence avec la classe précédente est que la méthode `run()` n'est pas surchargée, mais doit être implémentée et codée. Mais, attention, il ne suffit pas de remplacer `extends Thread` par `implements Runnable`. Dans le constructeur de `MonThread1`, le `super()` permettait de définir le nom du `Thread` qui serait récupéré par le `getName()` de cette même classe. Dans cette version, il sera nécessaire d'obtenir le nom du `Thread` au travers de la méthode statique `Thread.currentThread()` sur le processus courant actif. L'origine du problème vient de la nécessité de créer une méthode `départ()` sur un objet de notre classe, méthode qui va instancier deux `Thread`s. Il est en effet impossible de créer un objet pour une interface. Le `new Thread()`, avec deux paramètres, permet de créer un processus parallèle sur celui actif, c'est-à-dire `this`, et de lui donner un nom qui pourra être récupéré avec `getName()`. Enfin, nous constaterons que le résultat est identique au précédent.

Des constantes dans une interface Java

Dans une interface, il est possible de définir des constantes, au contraire des méthodes, dont nous ne pouvons définir que leurs déclarations. Dans cette définition d'interface, que nous retrouverons dans le fichier `MaConstante.java` :

```
public interface MaConstante {
   public int ANNEE = 1970;
}
```

le nom `MaConstante.ANNEE` est une constante. Nous noterons le style Java, qui utilise des majuscules pour ces types de constantes. Pour le vérifier, cette classe :

```
class UneClasse implements MaConstante {
  public void test() {
    System.out.println(ANNEE);
  }
}

public class UnTest {
  public static void main(String[] args) {
    System.out.println(MaConstante.ANNEE);

    UneClasse uc = new UneClasse();
    uc.test();
  }
}
```

nous montre la manière d'accéder à la constante `ANNEE` de l'interface `MaConstante`. La classe `UneClasse` nous permet d'accéder directement à `ANNEE`, car elle implémente l'interface `MaConstante`, qui, il faut le remarquer, ne possède aucune déclaration de méthodes,

mais seulement une constante. Il n'est pas nécessaire de déclarer `ANNEE` comme `final`, car c'est implicitement le cas.

Grouper des constantes dans une interface

D'une manière similaire aux énumérations en C++, que nous avons étudiées au chapitre 2 au travers du mot-clé `enum`, nous pouvons grouper des listes de constantes dans une interface :

```
public interface Jours {
  public int LUNDI = 1, MARDI = 2, MERCREDI = 3,
             JEUDI = 4, VENDREDI = 5, SAMEDI = 6, DIMANCHE = 7;
}
```

Il est alors possible de créer un package avec ces constantes et de les importer lors de leurs utilisations. Le JDK 1.5 a enfin introduit les énumérations (`enum`) dont nous avons parlé au chapitre 6.

Sérialisation et clonage d'objets en Java

Ayant à présent assimilé le concept d'interface en Java, nous pouvons passer rapidement à deux fonctionnalités essentielles du langage Java, la sérialisation et le clonage. Ces deux aspects n'existent pas directement en C++, bien que le constructeur de copie en C++ représente en fait un clonage.

Sérialiser des objets Java

La sérialisation est un mécanisme qui permet, entre autres, d'écrire un objet complet sur le disque. Comme nous travaillons avec des flux, nous pourrions utiliser la sérialisation à d'autres fins de communication que simplement un fichier. C'est un sujet important, mais qui dépasse les limites que nous nous sommes fixées pour cet ouvrage. La description rapide de cette technologie et un exemple concret devraient suffire.

La sérialisation d'objets peut permettre à des instances de classes de persister, même après que le programme est terminé. Ils pourront ensuite être rechargés, sans avoir besoin d'une base de données qui assure cette persistance. Il y a deux aspects importants qui font partie de la technologie Java, ce sont le RMI et les JavaBeans. Le RMI (*Remote Method Invocation*), similaire aux objets CORBA, permet de distribuer des objets sur d'autres machines. La sérialisation permet de transporter le contenu et l'état d'objets sur une autre machine. Les Beans, des composants Java suivant des règles strictes pour leurs définitions et constructions, peuvent récupérer leurs états lorsque l'application est relancée.

Dans l'exercice qui suit, nous allons écrire une classe de test qui va nous enregistrer une instance d'un objet `Personne` dans un fichier et la relire directement. Nous reprendrons notre fameux capitaine Haddock, que nous pourrions stocker, en C++, dans un fichier délimité tel que :

```
"Haddock", "Capitaine", "1907"
```

Ici, nous n'allons ni lire de fichier délimité ni en écrire un pour une exportation possible en C++. Nous comprendrons rapidement que le fichier sérialisé en Java ne pourra pas être lu en C++, car il nous faudrait connaître le format binaire généré par Java.

Dans l'exemple qui suit, nous allons sérialiser un objet de la classe `PersonneSerial`, après avoir défini ces attributs :

```
import java.io.*;

public class PersonneSerial implements Serializable {
 private String nom;
 private String prenom;
 private int annee;

 private static final long serialVersionUID = 6526472292623776149L;

 public PersonneSerial(String lenom, String leprenom, String lannee) {
   nom = lenom;
   prenom = leprenom;
   annee = Integer.parseInt(lannee);
 }

 public void unTest() {
   System.out.print("Nom et prénom: " + nom + " " + prenom);
   System.out.println(" : " + annee);
 }


 public static void main(String[] args) {
   PersonneSerial nom = new PersonneSerial("Haddock", "Capitaine", "1907");
   nom.unTest();

   try {
     ObjectOutputStream out = new ObjectOutputStream
     ➥(new FileOutputStream("haddock.dat"));
     out.writeObject(nom);
     out.close();

     ObjectInputStream in =
             new ObjectInputStream(new FileInputStream("haddock.dat"));
     PersonneSerial nomlu = (PersonneSerial)in.readObject();
     in.close();

     nomlu.unTest();
   }
   catch(Exception e) {
     System.out.println("L'objet de la classe Personne ne peut être sauvé");
     System.out.println(" ou récupéré du fichier haddock.dat");
     return;
   }
 }
}
```

Nous avons affirmé précédemment qu'une interface ne possédait pas de code et que la nouvelle classe devait l'implémenter. Sans entrer dans les détails, nous dirons simplement que tous les attributs de la classe `PersonneSerial` doivent être sérialisables. Pour la variable `annee`, qui est un `int` dont le compilateur connaît la dimension, il n'y aura pas de difficulté pour l'écriture des octets correspondant à la valeur de ce type de variable. Pour les deux `Strings` `nom` et `prenom`, nous pouvons nous imaginer le mécanisme lorsque nous consultons l'API de la classe `String` :

```
public final class String extends Object implements Serializable, Comparable
```

`String` est bien un objet sérialisable. Pour sérialiser notre classe `PersonneSerial`, nous utilisons la classe `ObjectOutputStream` et sa méthode `writeObject()`. L'opération inverse est garantie avec la classe `ObjectInputStream` et sa méthode `readObject()`.

`serialVersionUID` est un code pour identifier la version de la classe. Nous l'avons introduit ici uniquement pour éliminer un message d'alerte lors de la compilation. Pour plus de détails, consultons la documentation de Java afin de générer cette valeur correctement.

Si nous exécutons le programme, nous obtiendrons :

```
Nom et prenom: Haddock Capitaine : 1907
Nom et prenom: Haddock Capitaine : 1907
```

Durant ce transfert, les données ont été correctement sauvegardées et rechargées. À titre d'information, nous pouvons examiner le contenu du fichier binaire `haddock.dat` avec l'outil Linux `od` (voir par exemple l'annexe B, section « Les outils Linux de MSYS ») :

```
od -bc haddock.dat
```

Et le résultat présenté :

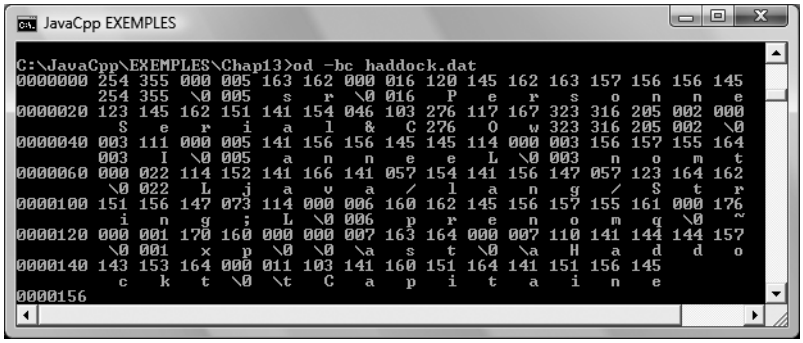

Figure 13-1
Od –bc de haddock.dat

Nous comprenons que nous aurions eu quelques difficultés à lire ce fichier en C++.

Pour terminer, nous devons indiquer qu'il y a des situations où il serait nécessaire de contrôler soi-même la sérialisation. Il y a différentes alternatives, comme celle de redéfinir les méthodes `writeObject()` et `readObject()`.

Le clonage d'objet

Nous revenons à présent rapidement sur ce sujet, qui a déjà été traité au chapitre 11. Ce fut à ce moment-là un passage obligé, car nous avions fait un parallélisme avec l'opérateur = en C++. Nous avions alors rencontré un :

```
public class Clown implements Cloneable { };
```

Nous n'avions aussi aucune méthode d'interface à définir, car la méthode `clone()` de la classe `Object` faisait tout le travail de copie bit à bit. Ce mécanisme est identique à la sérialisation que nous avons vue ci-dessus, dans laquelle tous les objets (`int` et `String`) étaient sérialisables.

Il nous faut donc montrer à présent comment redéfinir la méthode `clone()`. Nous reprenons le code du chapitre 11 et le modifions :

```
public class Clown1 implements Cloneable {
 private String nom;
 private String prenom;
 private int annee;

 public Clown1(String lenom, String leprenom, String lannee) {
   nom = lenom;
   prenom = leprenom;
   annee = Integer.parseInt(lannee);
 }

 public void setAnnee(int lannee) {
   annee = lannee;
 }

 public void unTest() {
   System.out.print("Nom et prénom: " + nom + " " + prenom);
   System.out.println(" : " + annee);
 }

 public Object clone() {
   Clown1 co = null;

   try {
     co = (Clown1)super.clone();
     co.annee = -1;
   }
   catch (CloneNotSupportedException e) {}

   return co;
```

```
  }

  public static void main(String[] args) {
    Clown1 nom1 = new Clown1("Haddock", "Capitaine", "1907");
    Clown1 nom2;
    Clown1 nom3;

    nom2 = nom1;

    nom3 = (Clown1)nom1.clone();

    if (!nom3.equals(nom1)) {
      System.out.println("Object nom1 et nom3 sont différents");
    }

    nom2.setAnnee(1927);

    nom1.unTest();
    nom2.unTest();
    nom3.unTest();

    if (nom2.equals(nom1)) {
      System.out.println("Object nom1 et nom2 sont égaux");
    }
  }
}
```

Il y a ici quelques variantes à noter, en particulier la disparition de la séquence `try` et `catch()`, qui a été déplacée dans la redéfinition de méthode `clone()`. Nous avons aussi modifié la copie des objets en donnant à l'attribut `annee` la valeur de –1 après le clonage.

Nous avons bien le résultat attendu :

```
Object nom1 et nom3 sont différents
Nom et prénom: Haddock Capitaine : 1927
Nom et prénom: Haddock Capitaine : 1927
Nom et prénom: Haddock Capitaine : -1
Object nom1 et nom2 sont égaux
```

Pour finir, il nous faut revenir au clonage et à la sérialisation. En effet, une analyse plus profonde nous montrerait que ces deux domaines devraient être considérés en parallèle, ceci afin d'obtenir une implémentation correcte.

Il faudra toujours se méfier des objets qui possèdent des références communes à d'autres objets.

Résumé

Aussi bien le langage C++ que Java supportent l'héritage multiple, mais sous une forme différente. En C++, cela se fait d'une manière attendue, où nous déclarons plusieurs

héritages lors de la définition de la classe. Comme cette manière de faire se révèle trop complexe, principalement pour les compilateurs, le langage Java utilise le concept d'interface. Cette dernière est en fait similaire à une classe abstraite, dans laquelle le code doit être entièrement implémenté dans la classe qui hérite d'une ou de plusieurs interfaces.

Exercices

1. Essayer de créer une classe `SalaireFrancais` en Java, similaire à la classe `Integer`, c'est-à-dire qui hérite de cette dernière et qui implémente l'interface `Comparable`. La classe `SalaireFrancais` conservera le montant du salaire en euros et le taux de change pour le franc français. La méthode `compareTo(Object o)` de l'interface `Comparable` effectuera la comparaison en franc français.
2. Transformer les classes `Forme`, `Cercle`, `Rectangle` et `Dessin` du chapitre 12 afin d'utiliser une interface. `Forme` devra hériter de la classe `java.awt.Point` et d'une interface `GraphiqueObjet` qui définira deux méthodes `dessine()` et `efface()`. La classe `Dessin` vérifiera les différentes méthodes.

14

Devenir collectionneur

Collectionner des objets n'est certainement pas l'une des préoccupations majeures des programmeurs débutants. Au départ, ces derniers se contentent de tableaux d'entiers ou de chaînes de caractères. Ces tableaux sont souvent fixes et codés statiquement dans le programme. Mais arrive un jour où ces programmeurs amateurs deviennent des professionnels et se rendent compte de la nécessité de sauvegarder ces listes sur un support magnétique. Ensuite, ils vont vouloir effacer ou ajouter des composants. C'est à ce moment-là que les difficultés apparaissent. Lorsqu'une liste d'employés d'une entreprise doit être conservée, nous commençons par créer un tableau d'objets en Java ou C++. Lorsque nous effaçons un employé pour le remplacer éventuellement par un nouveau venu, nous devons être capables de manipuler ces listes, de rechercher et de remplacer un objet après avoir désactivé et recréé les ressources. C'est ici que nos conteneurs, des collections d'objets, vont prendre toute leur signification.

Un employé a peut-être des entrées dans plusieurs tables, qui sont elles-mêmes liées entre elles avec des index ou des clés. Nous entrons ici dans le domaine des bases de données, dans lequel il est tout à fait vraisemblable et raisonnable de considérer et d'utiliser les conteneurs à disposition dans les bibliothèques Java et C++, comme support logiciel pour une interface avec des bases de données traditionnelles.

Note

Ce chapitre est une introduction aux collections et autres algorithmes. Le lecteur devra consulter la documentation des langages Java et C++ sur ce vaste sujet, qui pourrait être couvert dans un ouvrage spécialisé. Le but de ce livre est de donner suffisamment d'exemples pour se familiariser avec ces nombreux algorithmes et classes. Il y a ainsi plus de 80 algorithmes dans le Standard C++ !

Le vector en C++

Le `vector` fait partie de la bibliothèque STL (*Standard Template Library*) et se trouve être le conteneur le plus simple, par lequel nous commencerons notre présentation. Pour illustrer l'utilisation de cette classe, nous allons prendre l'exemple de deux listes, une liste d'entiers (`int`) et une liste de chaînes de caractères (`string`). Nous n'allons pas prendre peur à la vue de la syntaxe des *templates* (modèles), que nous verrons plus loin, au chapitre 19. Nous dirons simplement que la classe `vector` est capable de collectionner différents types d'objets et qu'il n'est pas nécessaire de créer une classe pour chaque type, comme `vectorInt` et `vectorString`. La méthode `replace()` n'est pas simple et il faudra consulter la documentation, par exemple `http://wwwinfo.cern.ch/asd/lhc++/RW/stdlibcr/bas_0007.htm`. Elle permet de remplacer un certain nombre de caractères en fonction de critères définis par ses différents paramètres. La méthode `replace()` du Standard C++ devrait être vérifiée séparément suivant les cas d'utilisation. De plus, cette méthode n'est pas très solide. Si des paramètres sont incorrects ou inconsistants, nous pourrions nous retrouver avec des résultats surprenants. Le dernier paramètre est le plus intéressant : `'0' + i`. `'0'` nous retourne ici la valeur du caractère ASCII du chiffre 0. Si `i` = 2, nous avons bien un 2, c'est-à-dire la valeur binaire ASCII du chiffre 2. Cette construction est possible, car les chiffres de 0 à 9 dans la table des caractères ASCII sont contigüs. Voici donc notre premier exemple :

```
// testvector.cpp
#include <iostream>
#include <string>
#include <vector>

using namespace std;

int main() {
  vector<int> v_entier;          // liste d'int
  vector<string> v_chaine;       // liste de string

  char *un_char = "0";
  string nombre = "nombreX";

  for (int i = 0; i < 5; i++) {         // remplissage
    v_entier.insert(v_entier.end(), i);
    nombre.replace(6, 1, 1, '0' + i);
    v_chaine.insert(v_chaine.begin(), nombre);
  }

  for (int j = 0; j < v_entier.size(); j++) {
    cout << (string)v_chaine[j] << " ";
    cout << v_entier[j] << endl;
  }

  vector<int>::iterator it1;
  vector<string>::iterator it2;
  it2 = v_chaine.end() - 1;
```

```
  for (it1 = v_entier.begin(); it1 != v_entier.end(); it1++) {
    cout << *it2-- << " " << *it1 << endl;
  }

  vector<string>::iterator it_debut = v_chaine.begin();
  vector<string>::iterator it_fin   = v_chaine.end();

  while (it_debut != it_fin) {
    if (*it_debut == "nombre3")  v_chaine.erase(it_debut);
    it_debut++;
  }

  it_debut = v_chaine.begin();
  it_fin   = v_chaine.end() - 1;
  while (it_fin >= it_debut) {
    cout << *it_fin-- << endl;
  }

  cout << "Dimension du vector v_entier: " <<  v_entier.size() << endl;
  cout << "Dimension du vector v_chaine: " <<  v_chaine.size() << endl;
}
```

Avec les déclarations de `v_entier` et de `v_chaine` comme `vector<int>` et `vector<string>`, nous rencontrons cette nouvelle forme, avec les `<>`. Nous aurons donc deux `vector` qui seront définis pour maintenir deux collections distinctes d'entiers et de `string`.

Après leurs déclarations, nos deux `vector` sont vides. Ils seront ensuite remplis avec la méthode `insert()`. Le `vector` `v_entier` contiendra une liste de nombres de 0 à 4. Nous ajoutons chaque fois le nouveau en fin de liste, avec le paramètre de positionnement `v_entier.end()`. Pour le `vector` `v_chaine`, nous faisons l'inverse, avec l'insertion en début de liste. `v_entier.end()` et `v_chaine.begin()` sont en fait des itérateurs, dont nous verrons l'utilité ci-dessous.

La deuxième boucle, `for()`, va nous montrer une liste combinée, qui va apparaître ainsi :

```
nombre4 0
nombre3 1
nombre2 2
nombre1 3
nombre0 4
```

Pour ce faire, nous avons utilisé l'opérateur `[]`, qui est à disposition dans la classe `vector`. Il permet d'accéder à un élément de la collection avec un index. En cas d'erreur d'index, il n'y a pas d'exception générée. Il faut noter que le code du programme n'est correct que si les deux listes possèdent le même nombre d'éléments.

Utiliser un itérateur

La bibliothèque STL met à disposition une classe `iterator` pour notre classe `vector`. Ces types d'itérateurs sont surtout intéressants parce qu'ils nous permettent de travailler

indépendamment du type de conteneur. Ici, c'est un `vector`, mais nous pourrions avoir une autre collection de la bibliothèque STL. Avant la boucle `for()` suivante :

```
for (it1 = v_entier.begin(); it1 != v_entier.end(); it1++) {
  cout << *it2-- << " " << *it1 << endl;
}
```

nous avons les deux déclarations de nos itérateurs :

```
vector<int>::iterator it1;
vector<string>::iterator it2;
```

Ils nous permettront de traverser les deux listes dans deux ordres différents, ce qui permettra de remettre nos listes correctement, puisque la deuxième a été introduite dans l'autre sens :

```
nombre0 0
nombre1 1
nombre2 2
nombre3 3
nombre4 4
```

Il est essentiel de bien utiliser les pointeurs `it1` et `it2` et de vérifier s'ils atteignent les limites permises. Les formes `*it1` et `*it2` nous retournent un pointeur sur les objets, qui peuvent être présentés avec le `cout` traditionnel.

Pour terminer, nous allons effacer l'objet contenant la chaîne de caractères `nombre3` dans la deuxième liste. Ceci se fait avec la méthode `erase()` dans une boucle de test avec un itérateur. Nous constaterons aussi comment les boucles `while()` sont simples à utiliser avec un itérateur. Nous pouvons d'ailleurs les traverser dans un sens comme dans l'autre. Le dernier résultat est attendu :

```
nombre0
nombre1
nombre2
nombre4
Dimension du vector v_entier: 5
Dimension du vector v_chaine: 4
```

Les algorithmes du langage C++

Nous associons souvent, en C++, les algorithmes et les conteneurs. Ce n'est pas tout à fait correct, car ces nombreux algorithmes ne sont pas des méthodes de ces conteneurs, mais sont des fonctions séparées. Ceci peut se faire, car les algorithmes emploient des itérateurs, et il est donc tout à fait possible de les utiliser avec des tableaux C traditionnels. Avant de passer à leur fonctionnement avec des conteneurs, nous allons montrer, dans l'exemple qui suit, un certain nombre de ces algorithmes qui peuvent remplacer avantageusement certaines fonctions C ou constructions traditionnelles :

```
// algoC.cpp
#include <iostream>
```

```
#include <algorithm>

using namespace std;

int main() {
  char pchar1[21] = "<daiKRgAcDlTewQ96Qd>";
  char pchar2[21];
  int  table[7] = {5, 5, 5, 4, 3, 2, 1};

  cout << "Test1: " << pchar1 << endl;

  sort(pchar1 + 1, pchar1 + 19);

  cout << "Test2: " << pchar1 << endl;

  copy(pchar1, pchar1 + 10, pchar2 + 10);
  copy(pchar1 + 10, pchar1 + 20, pchar2);
  pchar2[20] = 0;

  cout << "Test3: " << pchar2 << endl;

  sort(table, table + 7);

  cout << "Test4: ";
  for (int i = 0; i < 7; i++) {
    cout << table[i] << " ";
  }
  cout << endl;

  cout << "Test5: " << count(table, table + 7, 5) << endl;
}
```

Ce code nous donnera le résultat suivant :

```
Test1: <daiKRgAcDlTewQ96Qd>
Test2: <69ADKQQRTacddegilw>
Test3: acddegilw><69ADKQQRT
Test4: 1 2 3 4 5 5 5
Test5: 3
```

Dans la chaîne de caractères `pchar1` (`Test1`), nous commençons par trier, avec la fonction `sort()`, toutes les lettres entre la deuxième et l'avant-dernière position. Nous voyons avec `Test2` que le tri se fait aussi sur les majuscules et autres caractères, tels qu'ils se présentent dans la table ASCII. Les deux paramètres sont des pointeurs.

Pour le `Test3`, nous utilisons l'algorithme `copy()`, qui possède trois paramètres : le début et la fin de la séquence, ainsi que la cible. Ce sont à nouveau des pointeurs.

Nous avons utilisé pour le `Test4` une table avec sept entiers que nous trions. Le `sort()` reçoit bien des pointeurs à des entiers, mais ici l'itération se fera différemment que pour

les caractères, pour lesquels nous avions une donnée par octet. Ces pointeurs sont bien des itérateurs qui connaissent la dimension du type.

Enfin, le `count()` est un de ces nombreux algorithmes à disposition dans le Standard C++ : il va nous retourner le nombre d'objets qui ont la même valeur.

La classe vector en C++ et l'algorithme sort()

Nous passons à présent à un exemple de l'algorithme `sort()` (triage), appliqué à un objet de la classe `vector` :

```
// algovector.cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <ctime>
#include <cstdlib>

using namespace std;

bool monAlgo(int nombre1, int nombre2) {
  int reste1 = nombre1 % 10;
  int reste2 = nombre2 % 10;

  if (reste1 > reste2)  return true;
  if (reste1 == reste2) {
    if (nombre1 < nombre2)  return true;
  }
  return false;
}

int main() {
  vector<int> v_entier;
  srand((unsigned)time(NULL));

  for (int i = 0; i < 16; i++) {
     v_entier.push_back((rand() * 200)/RAND_MAX);
  }

  vector<int>::iterator it_debut = v_entier.begin();
  const vector<int>::iterator it_fin   = v_entier.end();

  sort(v_entier.begin(),  v_entier.end());

  while (it_debut != it_fin) {
    cout << *it_debut++ << " ";
  }
  cout << endl;

  it_debut = v_entier.begin();
```

```
  sort(v_entier.begin(),  v_entier.end(), monAlgo);

  while (it_debut != it_fin) {
    cout << *it_debut++ << " ";
  }
  cout << endl;

  cout << "Dimension du vector v_entier: " <<  v_entier.size() << endl;
}
```

`v_entier` est à nouveau une collection de nombres entiers. Elle contiendra des nombres aléatoires entre 0 et 200. Le remplissage se fait avec la méthode `push_back()`, à laquelle nous reviendrons ci-dessous. Lorsque cette besogne est terminée, nous déclarons nos deux itérateurs, dont un va rester constant tout au long du programme. L'itérateur du début de la collection devra, bien évidemment, être réinitialisé avant chaque nouvelle utilisation. Le premier `sort()` est classique, car il va nous trier nos nombres aléatoires dans l'ordre croissant.

Le deuxième `sort()` est très particulier, car nous lui attribuons une fonction comme paramètre : `monAlgo()`. Cette fonction sera appelée par l'algorithme `sort()` à chaque opération de tri sur deux nombres. Il est donc tout à fait possible de redéfinir totalement l'algorithme de tri pour nos besoins. Nous pourrions même imaginer mélanger encore mieux la liste, si elle ne satisfaisait pas un certain critère de désordre ! Ici, nous trions sur le chiffre des unités en utilisant le reste de la division par 10. Le tri se fait dans l'ordre décroissant, c'est-à-dire les 9 devant. Si le chiffre des unités est pareil, nous trions alors dans l'ordre croissant sur le nombre entier ! Le lecteur se posera la question de savoir si un tel algorithme a une utilité quelconque, et nous lui dirons que nous ne la voyons pas très bien non plus. Il faut parfois trier les employés d'une entreprise suivant des critères mixtes, comme par département, par fonction, par grade, par salaire ou encore par numéro de téléphone ou de sécurité sociale !

Nous présentons enfin un des résultats possibles, puisque la génération de ces nombres se fait d'une manière aléatoire :

```
4 7 8 25 46 59 59 68 74 81 97 137 169 175 179 197
59 59 169 179 8 68 7 97 137 197 46 25 175 4 74 81
Dimension du vector v_entier: 16
```

Faut-il utiliser `push_back()` ou `insert()` sur les `vector` C++ ? C'est effectivement une question des plus intéressantes, en relation avec les performances. Nous verrons, au chapitre 15, un certain nombre d'outils pour mesurer nos programmes, ce qui nous permettrait d'essayer certaines constructions ou alternatives. Avec l'exercice 2 de ce même chapitre 15, nous allons pouvoir mesurer les différences entre les `push_back()` ou autres `insert()`. Nous en présentons ici les conclusions.

La classe `vector` est construite d'une manière linéaire, continue en mémoire et rapide en insertion à la fin. La méthode suivante du premier exemple :

```
v_entier.insert(v_chaine.begin(), i);
```

est catastrophique, surtout lorsque la collection atteint une dimension respectable. Les deux méthodes :

```
v_entier.insert(v_chaine.end(), i);
v_entier.push_back(i);
```

vont insérer les entiers à la fin de la collection, mais `v_chaine.end()` est un itérateur et un appel de méthode en plus. `push_back()` est donc, en principe, la recommandation.

La classe list en C++

Après avoir étudié le classique `vector` en C++, nous avons choisi parmi les nombreux autres conteneurs, comme `array` ou `deque`, la tout aussi classique et puissante `list`.

De la classe `vector`, nous aurions dû retenir au moins trois aspects :

1. Elle possède un opérateur `[ ]` et est contiguë en mémoire.
2. Elle est très rapide en insertion et effacement en fin de liste.
3. Elle est à éviter pour des insertions en début et en milieu de liste.

La classe `list`, en revanche, est composée de liens pour le chaînage interne des objets. Comme pour la classe `vector`, si nous voulions sélectionner trois aspects importants et significatifs de la classe `list`, nous choisirions les suivants :

1. Elle ne possède pas d'opérateur `[ ]`.
2. Elle est rapide pour toutes sortes d'insertions ou d'effacements.
3. Elle possède de nombreuses méthodes spécifiques comme le tri (`sort()`), l'inversion (`reverse()`) ou encore la concaténation de deux listes (`merge()`).

L'exemple suivant nous montre quelques aspects d'utilisation de la classe C++ `list` :

```
// testlist.cpp

#include <iostream>
#include <list>
#include <algorithm>

using namespace std;

int main() {
  int tableau1[] = {1, 2, 4, 8, 16, 32, 64};
  int tableau2[] = {9, 8, 7, 6, 5, 4, 3, 2, 1};

  list<int> liste1(tableau1, tableau1 + 7);
  list<int> liste2(tableau2, tableau2 + 9);

  liste1.merge(liste2);
  liste1.remove(32);
  liste1.sort();
  liste1.unique();
```

```
    liste1.reverse();

    ostream_iterator<int> out(cout, " ");
    copy(liste1.begin(), liste1.end(), out);
    cout << endl;
}
```

Le résultat présenté :

```
64 16 9 8 7 6 5 4 3 2 1
```

est bien une liste inversée (`reverse()`), après avoir été triée (`sort()`). La liste finale est composée de la fusion (`merge()`) de deux listes d'origine, `liste1` et `liste2`, que nous avons construites à partir de deux tableaux d'entiers. En demandant l'affichage de `liste1` après `liste1.merge(liste2)`, on voit que la fusion des deux listes se fait de manière assez hétéroclite. Le `merge()` sert en fait à fusionner des listes triées.

Nous constaterons la forme du constructeur. Ce dernier utilise les pointeurs aux deux tableaux. `tableau1 + 7` est effectivement positionné sur l'élément suivant le dernier dans la liste. L'objet de la collection avec la valeur 32 a été effacé (`remove()`), et nous n'avons gardé que les éléments uniques : 1, 2, 4 et 8 étaient des doublons qui ont été effacés avec la méthode `unique()`. La méthode `unique()` est très particulière, car elle ne va effacer que les éléments identiques et adjacents. Si nous avions appliqué `unique()` avant `sort()`, rien ne se serait passé dans notre cas.

Il faut enfin revenir sur la très jolie forme :

```
ostream_iterator<int> out(cout, " ");
copy(liste1.begin(), liste1.end(), out);
```

qui nous permet d'appliquer un itérateur sur notre traditionnel `cout`. Le deuxième paramètre d'`out()` est intéressant, car il permet d'ajouter un espace de formatage après chaque chiffre. Il y a donc un espace invisible après le dernier chiffre, le 1 !

L'interface List en Java

> **Les collections Java avant le JDK 1.5**
>
> Les exemples donnés ci-dessous sont applicables aux anciennes collections, c'est-à-dire celles existant avant l'introduction des types génériques, lesquels sont abordés à la fin de ce chapitre.
>
> Nous recevrons, lors de la compilation, des messages d'alerte nous indiquant que des conversions devraient être apportées. Tous ces exemples peuvent d'ailleurs être convertis : nous l'avons fait pour l'un d'eux dans le second exercice.

La suite logique à notre présentation des collections est l'interface `List` en Java. Celle-ci définit précisément le comportement de cette collection d'objets ordonnée et séquentielle. Comme ce n'est pas une classe, nous ne pouvons pas l'instancier, mais pouvons utiliser deux de ces implémentations, qui sont par exemple les classes `Vector` et `ArrayList`.

L'exercice suivant va consister à générer vingt nombres aléatoires entre 0 et 4 et à les déposer dans un `Vector`. De cette liste, nous allons extraire dix éléments entre la position 5 incluse et la position 15 exclue et les déplacer dans un `ArrayList`. De cette seconde liste nous ferons un tri et une inversion. Enfin, nous écrirons, toujours sur cette liste, un algorithme de tri qui déplace les 0 en début de liste ! Voici donc le code :

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

public class TestList {

  public static void main (String args[]) {
    List maListe1 = new Vector();
    List maListe2 = new ArrayList();

    for (int i = 0; i < 20; i++) {
      maListe1.add(new Integer((int)(5 * Math.random())));
    }

    for (int i = 0; i < 20; i++) {
       System.out.print(maListe1.get(i) + " ");
    }
    System.out.println("");

    maListe2.addAll(maListe1.subList(5,15));
    maListe1.set(15, new Integer(9));
    maListe1.subList(5,14).clear();

    Iterator it1 = maListe1.iterator();
    while (it1.hasNext()) {
      System.out.print(it1.next() + " ");
    }
    System.out.println("");

    Collections.sort(maListe2);
    Collections.reverse(maListe2);

    Iterator it2 = maListe2.iterator();
    while (it2.hasNext()) {
      System.out.print(it2.next() + " ");
    }
    System.out.println("");

    Collections.sort(maListe2, compareNombre);
    it2 = maListe2.iterator();
    while (it2.hasNext()) {
      System.out.print(it2.next() + " ");
    }
    System.out.println("");
  }
```

```
  static final Comparator compareNombre = new Comparator() {
    public int compare(Object obj1, Object obj2) {
      Integer i1 = (Integer)obj1;
      Integer i2 = (Integer)obj2;
      int num1 = i1.intValue();
      int num2 = i2.intValue();

      if (num1 == 0)  return -1;
      if (num2 == 0)  return 1;
      return 0;
    }
  };
}
```

Ainsi que le résultat :

```
4  1  2  0  2  3  1  3  0  4  3  0  3  3  4  2  1  3  3  0
4  1  2  0  2  4  9  1  3  3  0
4  4  3  3  3  3  3  1  0  0
0  0  4  4  3  3  3  3  3  1
```

Les classes `Vector` et `ArrayList` sont très similaires, et il faudra consulter l'API pour les différences ou écrire des programmes de tests pour en savoir plus sur les performances, et ceci suivant les cas d'utilisation. `Vector` est par exemple synchronisé pour les processus parallèles (*threads*), ce qui n'est pas le cas des `ArrayList`.

Après avoir mentionné que ces collections et ces outils sont définis dans :

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

nous nous pencherons sur :

```
List maListe1 = new Vector();
List maListe2 = new ArrayList();
```

`List` est effectivement une interface, et cette construction est tout à fait possible, car `Vector` et `ArrayList` sont de vraies classes. Cela nous permet de travailler de manière beaucoup plus générique et au besoin d'interchanger les collections, juste en changeant deux ou trois lignes de code.

`5 * Math.random()` sera toujours entre 0 et 4.99999 et nous retournera bien un nombre entier entre 0 et 4. La méthode `add()` ajoute simplement un objet en fin de liste. L'objet ne peut être de type primitif, car il doit hériter de la classe `Object`. Nous utilisons donc naturellement la classe `Integer`.

Le `get(i)` permet un accès direct et séquentiel dans la liste, mais nous pouvons aussi utiliser un itérateur. Il faut d'abord créer une instance de la classe `Iterator` sur la liste désirée. Nous ne pourrons accéder à un élément avec `next()` que s'il existe, en vérifiant le `hasNext()`.

Il est possible de spécifier une liste d'éléments avec la méthode `subList()` avec l'index de départ et de fin. Les méthodes `addAll()` et `clear()` permettent d'ajouter `[5, 15[` ou d'effacer ces parties de listes `[5, 14[` (première valeur incluse, seconde exclue). L'instruction :

```
maListe1.set(15, new Integer(9));
```

est juste là pour montrer que l'objet à l'index 15, qui a déjà été copié dans la deuxième liste, est bien remplacé dans la première.

La classe `Collection` possède un certain nombre de méthodes statiques similaires aux algorithmes en C++ (`sort()` et `reverse()`). Nous avons joué le jeu avec l'écriture de la fonction `compareNombre`, qui est loin d'être évidente à écrire, mais qui est d'une flexibilité absolue. Sa seule fonctionnalité est de prendre les 0 et de les positionner en tête de liste. Les différentes valeurs de retour, 0, –1 et 1, indiquent le résultat de la comparaison de deux objets.

Dans ce type de collections, nous avons continuellement du transtypage :

```
public int compare(Object obj1, Object obj2) {
      Integer i1 = (Integer)obj1;
      Integer i2 = (Integer)obj2;
```

Nous devons connaître le type d'objet, ici `Integer`, que nous avons déposé dans la collection. Si nous insérons un objet d'un mauvais type, nous ne le verrons que lors de l'exécution, avec la génération d'une exception. En fin de chapitre, nous parlerons des types génériques qui sont apparus à partir du JDK 1.5 pour éviter ce genre de problème.

L'interface Set en Java

Au contraire de l'interface `List`, `Set` ne contient pas de doublons. Si un nouvel objet est inséré et qu'il existe déjà, il ne sera pas ajouté à ce type de collection. Dans notre exemple, nous allons utiliser la classe `HashSet`, qui implémente l'interface `Set`. Nous n'entrerons pas dans les détails de la construction de ce type de table (*hash table*), et nous dirons simplement qu'elle garantit une distribution constante des performances suivant que nous ajoutons, effaçons ou recherchons des éléments dans cette collection.

Voici donc le code dont nous allons présenter tout de suite la structure et le fonctionnement :

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

public class TestSet {
  public static void filtre(Collection uneCollect) {
     Integer objet;
     int nombre;

     Iterator it1 = uneCollect.iterator();
     while (it1.hasNext()) {
       objet = (Integer)it1.next();
       nombre = objet.intValue();
```

```
        if ((nombre % 2) == 1) {
          it1.remove();
        }
      }
      System.out.println("");
  }


  public static void viewList(Collection uneCollect) {
     System.out.println("Dimension: " + uneCollect.size());

     Iterator it1 = uneCollect.iterator();
     while (it1.hasNext()) {
       System.out.print(it1.next() + " ");
     }
     System.out.println("");
  }


  public static void main (String args[]) {
    Set maListe1 = new HashSet();

    for (int i = 0; i < 20; i++) {
      maListe1.add(new Integer((int)(20 * Math.random())));
    }

    viewList(maListe1);
    filtre(maListe1);
    viewList(maListe1);
  }
}
```

Le premier travail de cet exemple a été d'écrire une fonction `static` pour nous présenter le résultat : `viewList()`. Cette méthode est totalement générique et pourrait fonctionner pour n'importe quelle collection, car elle utilise simplement un itérateur sur une `Collection`. Cette dernière se trouve être la racine (root) de toutes les collections en Java. Dans `viewList()`, nous avons ajouté l'information sur la taille avec `size()`, pour montrer que notre boucle de vingt répétitions ne va pas allouer vingt nombres. Comme les nombres sont choisis au hasard entre 0 et 19 et que la collection `Set` est triée avec des éléments uniques, nous aurons en moyenne entre douze et quinze objets, bien que la probabilité d'en avoir vingt ne soit pas nulle !

Notre méthode `filtre()` n'est générique que dans ces paramètres, car nous assumons une liste d'objets de type `Integer`. Tous les entiers non pairs sont effacés avec la méthode, à nouveau générique, `remove()`. Nous nous trouverons finalement avec une liste de dix nombres pairs au maximum, entre 0 et 18, comme présenté ici en exécutant la classe `TestSet` :

```
Dimension: 14
19 17 16 15 14 13 12 11 10 9 6 5 4 3
```

```
Dimension: 6
16 14 12 10 6 4
```

Une liste de téléphone en Java avec HashMap

Dans la hiérarchie des classes utilitaires (`java.util.*`), `HashMap` hérite d'une classe qui porte bien son nom, un `Dictionary`. Si nous devions créer une liste de téléphone, ce serait vraisemblablement une approche convenable, c'est-à-dire un dictionnaire de noms de personnes associées chacune à un numéro de téléphone. Dans l'exemple qui suit, nous allons rencontrer deux termes, une clé (*key*) et une valeur, et ceci respectivement pour la personne et son numéro de téléphone.

Nous n'allons pas tergiverser sur les détails de cette classe, mais nous dirons que `HashMap` possède un mécanisme pour obtenir un code (le « hash code ») qui est une valeur numérique calculée à partir des caractères d'un `String`. Cette clé permet de définir les entrées dans la table, qui obtient ainsi des valeurs d'accès raisonnables pour toutes les sortes de fonctions comme l'insertion, l'effacement et la recherche. Le choix de `HashMap` est généralement justifié au détriment d'autres tables comme la `TreeMap`, qui est basée sur un autre algorithme : le Red-Black tree. Pour ceux qui sont intéressés, l'ouvrage de Larry Nyhoof, *C++ An Introduction to Data Structures*, est essentiel (voir annexe G).

Nous passons immédiatement à l'implémentation de notre liste de téléphone, dont nous allons présenter les détails :

```
import java.util.*;

class TestListeTel {
   private Map telListe = new HashMap();

   public void ajoute(String nom, String numTel) {
     if (telListe.put(nom, numTel) != null) {
        telListe.remove(nom);
        telListe.put(nom, numTel);
     }
   }

   public String getNumero(String nom) {
     String resultat = (String)telListe.get(nom);

     if (resultat == null)  return "INCONNU";
     return resultat;
   }

   public String getNom(String numero) {
     String clef;
     String resultat = "";
     int nombre = 0;

     for (Iterator it = (telListe.keySet()).iterator(); it.hasNext(); ) {
```

```
            clef = (String)it.next();

            if ((String)telListe.get(clef) == numero) {
               nombre++;
               resultat += clef + "\n";
            }
         }

         if (nombre == 0)  return "AUCUNE";
         return resultat;
      }

      public String toString() {
         String  resultat = "Liste des numéros de téléphone:\n";
         String  clef;
         int i;

         for (Iterator it = (telListe.keySet()).iterator(); it.hasNext(); ) {
            clef = (String)it.next();
            resultat += clef;

            for (i = clef.length(); i < 30; i++) {
              resultat += " ";
            }

            resultat +=  (String)telListe.get(clef) + "\n";
         }
         return resultat;
      }

      public static void main(String[] args) {
         String tournesol = "Tournesol Tryphon";
         String numBoichat = "098009812";

         TestListeTel ttl = new TestListeTel();
         ttl.ajoute("Haddock Capitaine", "099005512");
         ttl.ajoute(tournesol, "099005519");
         ttl.ajoute("Boichat Jean-Bernard", numBoichat);
         ttl.ajoute("Boichat Nicolas", numBoichat);
         ttl.ajoute("Haddock Capitaine", "099006612");
         System.out.println(ttl);

         System.out.println("Le numéro de " + tournesol + " est " +
                                              ttl.getNumero(tournesol));
         System.out.println("Le numéro " + numBoichat +
                           " est attribué à:\n" + ttl.getNom(numBoichat));
      }
   }
```

La première remarque importante est de constater qu'un numéro ne sera pas nécessairement unique et pourrait être le même pour différents membres de la même famille. Ensuite,

nous choisirons, sans autres commentaires, de remplacer le numéro de téléphone si le nom existe déjà. La méthode `ajoute()` va donc vérifier si le retour de la méthode `put()` de la classe `HashMap` est `null` ou non, afin d'effacer l'entrée précédente si elle existe. Le `Capitaine` a demandé un nouveau numéro, car le `099005512` était trop proche de celui de la boucherie Sanzot !

Si la méthode `put()` est d'une simplicité extrême, il n'en va pas de même pour la recherche des noms et des numéros. Pour la méthode générale `toString()`, nous utiliserons un `iterator`, avec lequel nous allons extraire la clé (c'est-à-dire le nom de la personne), grâce à laquelle, avec `get(clef)`, nous pourrons obtenir la valeur du numéro de téléphone. Nous ajoutons des espaces pour obtenir une présentation convenable :

```
Liste des numéros de téléphone:
Boichat Jean-Bernard    098009812
Tournesol Tryphon    099005519
Boichat Nicolas    098009812
Haddock Capitaine    099006612
```

Nous avons ensuite programmé deux méthodes, `getNumero()` et `getNom()`, pour obtenir le numéro de téléphone d'une personne et, inversement, recevoir la liste de toutes les personnes ayant un numéro déterminé. La méthode `get()` de `HashMap` nous donne directement la valeur, alors que pour la méthode `getNom()` nous traversons toute la table avec un itérateur. Les deux résultats, `INCONNU` et `AUCUNE`, sont évidents. Enfin, nous présenterons tout de même la deuxième partie du résultat :

```
Le numéro de Tournesol Tryphon est 099005519

Le numéro 098009812 est attribué à:
Boichat Jean-Bernard
Boichat Nicolas
```

La même liste de téléphone avec map en C++

Maintenir une liste de téléphone est un exercice traditionnel en programmation. Nous allons donc la reprendre une fois encore, en utilisant cette fois-ci la classe `map` en C++ :

```
// telephone.cpp

#include <iostream>
#include <string>
#include <map>
#include <iterator>

using namespace std;

int main() {
  string tournesol = "Tournesol Tryphon";
  string numBoichat = "098009812";

  map<string, string> list_tel;
```

```
  list_tel[tournesol]              = "099005519";
  list_tel["Boichat Jean-Bernard"] = numBoichat;
  list_tel["Boichat Nicolas"]      = numBoichat;
  list_tel["Haddock Capitaine"]    = "099005512";

  list_tel["Haddock Capitaine"]    = "099006612";

  map<string, string>::iterator it;
  for (it = list_tel.begin(); it != list_tel.end(); it++) {
    cout << (*it).first << " : " << (*it).second << endl;
  }
  cout << endl;

  it = list_tel.find(tournesol);
  if (it == list_tel.end()) {
    cout << "Le numéro de " << tournesol << " est INCONNU" << endl;
  }
  else {
    cout << "Le numéro de " << tournesol << " est "
         << (*it).second << endl;
  }
  cout << endl;

  int nombre = 0;
  string resultat = "Le numéro " + numBoichat + " est attribué à: \n";

  for (it = list_tel.begin(); it != list_tel.end(); it++) {
    if ((*it).second == numBoichat) {
      resultat += (*it).first + "\n";
      nombre++;
    }
  }

  if (nombre == 0) {
    cout << "Il n'y a personne avec le numéro: " << numBoichat << endl;
  }
  else {
    cout << resultat;
  }
}
```

Comme la classe `map` est un modèle (*template*), il nous faut spécifier le type de la clé et celui de sa valeur. Nous pourrions nous poser la question de l'utilisation d'un `int` ou d'un `long` pour le numéro de téléphone ! Cependant, comme celui-ci peut commencer avec un ou plusieurs 0, une représentation avec un `string` est plus simple. L'utilisation des crochets (`[]`), pour ajouter une paire de `string` dans notre dictionnaire, pourrait nous sembler très particulière. Nous nous rappellerons que l'opérateur `[]`, qui est utilisé en général pour accéder à un tableau indexé, peut en fait être redéfini en C++ pour obtenir l'opération désirée. Lors de l'insertion du même nom, nous n'avons pas le même comportement qu'en

Java : nous avons simplement un remplacement de l'entrée sans la nécessité d'un effacement préalable.

Comme dans l'exemple précédent en Java, nous utilisons un `string`, soit directement, soit au travers d'une variable, pour l'ajouter au conteneur. Dans la réalité, cette liste de téléphone pourrait être lue d'un fichier ou d'une base de données. La méthode `find()` nous permet de chercher un numéro dans la liste à partir d'un nom, alors qu'un itérateur doit être utilisé pour sortir la liste complète ou rechercher les noms associés au même numéro. Nous pourrions inverser clé et valeur, mais il faudrait que le numéro de téléphone soit unique. Ce serait certainement applicable dans d'autres situations, bien qu'ici nous puissions aussi retrouver le même nom et prénom !

Écrire ce petit programme avec une classe améliorerait sans aucun doute la présentation, comme c'est le cas dans l'exemple en Java. C'est ce que nous ferons comme exercice.

Il est inutile de présenter le résultat de ce programme, qui est le même que précédemment, à la différence que les numéros de téléphone ne sont pas alignés à la colonne 30.

Les types génériques en Java

Dans ce chapitre, nous avons couvert jusqu'à présent les collections traditionnelles de Java, avant qu'apparaissent, à partir du JDK 1.5, les types génériques. Ces derniers sont assez similaires aux templates en C++ que nous traitons séparément au chapitre 19.

Un premier exemple simple

Nous allons créer une classe `Position` afin de conserver deux points dans un espace à deux dimensions :

```
// Position x,y dans l'espace
public class Position {
  private int x, y;

  public int getX() {
    return x;
  }

  public int getY() {
    return y;
  }

  public Position(int x, int y){
    this.x = x;
    this.y = y;
  }
}
```

C'est une classe traditionnelle avec deux méthodes pour retourner la position. Cette classe se compile sans problème.

Cette première version, compatible avec des JDK antérieurs à la version 1.5, va nous permettre de collectionner un certain nombre de points dans notre espace :

```
import java.util.*;

public class Generics1 {
   public static void main(String[] args) {
      /**
      *Avant le JDK 1.5
      */
      ArrayList liste = new ArrayList();

      liste.add(new Position(5, 80));
      liste.add(new Position(12, 120));
      liste.add(new Position(43, 251));

      Iterator it = liste.iterator();
      while(it.hasNext()){
        Position pos = (Position)it.next();
        System.out.println("X=" + pos.getX() + " Y=" + pos.getY());
      }
   }
}
```

Nous connaissons déjà les détails avec l'itérateur et le problème de transtypage : `(Position)it.next();`.

Si nous compilons ce programme avec le JDK 1.6, sans le paramètre `-Xlint` intégré à Crimson, nous recevrons ceci :

```
Note: Generics1.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

Avec le `-Xlint`, nous en saurons un peu plus :

```
Generics1.java:10: warning: [unchecked] unchecked call to add(E) as a member of the
➥raw type java.util.ArrayList
      liste.add(new Position(5, 80));
                ^
Generics1.java:11: warning: [unchecked] unchecked call to add(E) as a member of the
➥raw type java.util.ArrayList
      liste.add(new Position(12, 120));
                ^
Generics1.java:12: warning: [unchecked] unchecked call to add(E) as a member of the
➥raw type java.util.ArrayList
      liste.add(new Position(43, 251));
                ^
3 warnings
```

Cependant, cela ne nous empêche pas d'exécuter le programme, car il a tout de même été compilé. Voici le résultat attendu de l'extraction de nos trois positions dans l'espace :

```
X=5 Y=80
X=12 Y=120
X=43 Y=251
```

Nous pourrons faire maintenant beaucoup mieux avec le JDK 1.5 qui intègre les types génériques :

```
import java.util.*;

public class Generics2 {
   public static void main(String[] args) {
      /**
      *Depuis le JDK 1.5
      */
      ArrayList<Position> liste = new ArrayList<Position>();

      liste.add(new Position(5, 80));
      liste.add(new Position(12, 120));
      liste.add(new Position(43, 251));

      for (Position pos : liste) {
        System.out.println("X=" + pos.getX() + " Y=" + pos.getY());
      }
   }
}
```

Nous aimerions dire : « Mon dieu, comme c'est simple et propre ! ». Le résultat est évidemment le même que précédemment, mais sans message d'alerte du compilateur.

`ArrayList` est maintenant associé à un type de classe, ici `Position`, et avec la forme :

```
ArrayList<Position> liste = new ArrayList<Position>();
```

Nous allons pouvoir déposer dans `liste` des objets de la classe `Position` sans risquer d'y inclure un objet d'un autre type, ce qui entraînerait alors une erreur lors de l'exécution.

Nous avons aussi la forme :

```
for (Position pos : liste) {
```

Celle-ci est aussi nouvelle dans le JDK 1.5. Nous l'avons déjà découverte au chapitre 4.

Un exemple de la nouvelle forme pour l'`Iterator`, sans transtypage, sera donné dans le second exercice de ce chapitre.

Autoboxing et Fibonacci

Nous connaissons sans doute le nombre de Fibonnaci ou l'avons peut-être rencontré dans le livre de Dan Brown, le *Da Vinci Code*. Nous pouvons aussi consulter l'article sur le site Web de Wikipedia (`http://fr.wikipedia.org/wiki/Nombre_de_Fibonacci`), le lire attentivement en comptant les couples de lapins et nous amuser à programmer cette suite jusqu'à

l'index 21 et avec l'autoboxing, une nouvelle fonctionnalité du JDK 1.5. Nous commencerons par présenter le code :

```
import java.util.*;

public class AutoUnBoxing {
  public static void main(String[] args) {
    ArrayList<Long> fiboListe = new ArrayList<Long>();
    long  fibNum1   = 0;
    long  fibNum2   = 1;

    fiboListe.add(fibNum1); // index 0
    fiboListe.add(fibNum2); // index 1

    long ancienFibo = 0;
    for(int i = 0; i < 20; i++) {
      ancienFibo = fibNum2;
      fibNum2    = fibNum1 + fibNum2;
      fibNum1    = ancienFibo;

      fiboListe.add(fibNum2);
    }

    int index = 0;
    for (long num : fiboListe) {
      System.out.println(index++ + ": " + num);
    }
  }
}
```

Voici son résultat, qui est bien celui présenté sur le site Web de Wikipédia :

```
0: 0
1: 1
2: 1
3: 2
4: 3
5: 5
6: 8
…
20: 6765
21: 10946
```

Nous retrouvons en fait les mêmes mécanismes que les types génériques ou le transtypage corrigé ou adapté depuis le JDK 1.5, mais pour une variable de type primitif. Avant le JDK 1.5, il fallait utiliser un objet de type *wrapper* correspondant, ici un `Long`. L'instruction :

```
fiboListe.add(fibNum2);
```

aurait dû être écrite ainsi dans la version 1.4 du JDK :

```
fiboListe.add(new Long(fibNum2));
```

Le nombre `fibNum2` est une variable de type primitif, mais `new Long(fibNum2)` est un `Object` Java. L'unboxing est l'opération inverse, que nous retrouvons dans la boucle `for()` qui, elle aussi, utilise la nouvelle forme du JDK 1.5.

Résumé

Maîtriser les conteneurs et les algorithmes est essentiel en programmation Java et C++. Nous avons un certain nombre de classes à disposition, qui vont de simples listes à des dictionnaires de paires de clé et de valeur. Le choix devrait se faire en fonction de leur capacité et efficacité, suivant la tâche requise par l'application. Les conteneurs sont en général traversés avec des itérateurs.

Exercices

Ces deux exercices sont très importants pour bien mémoriser et comprendre les notations et les utilisations des classes de collections et des outils associés dans ces deux langages, lesquels sont en relation directe avec les types génériques et les templates.

Nous pourrons ensuite les comparer et les reprendre pour d'autres exemples que nous pourrions inventer nous-mêmes, comme la création et l'accès à de petites bases de données.

1. Reprendre la liste de téléphone en C++ et écrire une classe, comme dans l'exemple en Java, qui implémente les différentes opérations.
2. Reprendre la classe `TestListeTel.java` et la convertir avec les listes génériques en considérant les formats `Map<String, String>` et `HashMap<String, String>`. Nous utiliserons toujours un `Iterator`, mais sans transtypage, avec la forme `Iterator<String>`.

15

Concours de performance

Dans ce chapitre, nous allons analyser les performances de certains programmes et de certaines constructions, en jouant sur les variantes et alternatives offertes par du code sensible à ces aspects. Le but n'est pas nécessairement de rechercher le code le plus performant aux dépens de sa lisibilité ou de sa maintenance. Cette partie est en fait plus importante pour sensibiliser les développeurs, afin de les guider dans certains choix au cours de la conception des programmes et de classes réutilisables.

Comment analyser les performances ?

Dans cette première partie, il nous faut parler des outils qui vont nous permettre de faire des mesures correctes. Suivant la base de temps utilisée, qui sera la seconde ou la milliseconde, il faudra non seulement répéter suffisamment la fonction ou l'opération à mesurer, mais aussi l'exécuter plusieurs fois dans un environnement constant. Les conditions de test doivent être toujours identiques. Cela concerne non seulement la machine, ce qui semble évident, mais aussi d'autres composants qui pourraient donner des résultats erronés, comme la définition de l'écran, le nombre de programmes chargés ou en activité. Nous n'allons pas transférer le dernier JDK de Sun Microsystems pendant l'exécution de ces tests de performance. Les conditions de test doivent être non seulement identiques, mais aussi définies et décrites précisément, comme partie intégrante des résultats et des conclusions que nous pourrions y apporter.

Si le résultat d'un test nous semble erroné, nous aurons tendance à réitérer ce test, puisque nous avons été surpris du résultat. Au contraire, s'il nous paraît correct, nous aurons trop facilement tendance, à tort, à l'accepter, alors qu'il faut en fait, dans tous les cas, répéter le test et repenser à notre environnement : par exemple savoir s'il correspond à la réalité. Des simulations extérieures, telles celles de surcharge, pourraient se révéler nécessaires,

mais s'appliquent généralement aux tests de capacité et aux traitements des erreurs en relation avec les limites du système.

Les outils en Java

Afin de mesurer le temps d'exécution d'un programme ou d'une partie de celui-ci, nous allons rédiger une classe en Java, qui nous permettra d'identifier le temps écoulé avec une précision en millisecondes (ms). Nous utiliserons par la suite cette classe dans nos différents exemples :

```
import java.util.Properties;
import java.io.*;

public class Mesure {
  private long mtime1;
  private long mtime2;
  private long resultat;

  public void start() {
    mtime1 = System.currentTimeMillis();
  }

  public long end() {
    mtime2 = System.currentTimeMillis();
    resultat = mtime2 - mtime1;
    return resultat;
  }

  public String toString() {
    return « Temps écoulé: «  + resultat + «  ms »;
  }

  public static void main(String[] args) {
    Mesure mesure1 = new Mesure();
    System.out.println("Test de la classe Java Mesure");
    Properties props = System.getProperties();
    mesure1.start();
    for (int i = 0; i < 2000000; i++) {
      props.put("myProperty", "true");
    }
    mesure1.end();
    System.out.println(mesure1);
  }
}
```

Les méthodes `start()` et `stop()` de la classe `Mesure` permettent respectivement de lancer et de terminer la mesure. Dans la partie `main()` de notre classe, nous avons inclus un petit programme de test. Sur une machine dotée d'un Pentium III 450 MHz et de 128 Mo de mémoire vive, nous avons obtenu le résultat suivant :

```
Test de la classe Mesure
Temps écoulé: 1540 ms
```

Nous avons ensuite répété l'opération plusieurs fois et obtenu 1 540, 1 490, 1 530 et 1 480 ms. Le temps d'exécution de 1,5 secondes par rapport à la boucle de 2 000 000 est tout à fait raisonnable. Nous pouvons à présent continuer le jeu et mettre en commentaire la méthode :

```
//  props.put("myProperty", "true");
```

Nous obtiendrons alors 0 ms ! Nous pourrions donc nous demander si le compilateur n'est pas trop intelligent pour découvrir qu'il n'y a rien à faire ! Cependant, si nous écrivions maintenant ceci :

```
int j = 0;
for (int i = 0; i < 2000000; i++) {
   j = j + i;
}
```

nous aurions aussi 0 ms ! La boucle de 2 000 000 est donc insignifiante et la conclusion que nous allons proposer devrait être tout à fait correcte ! En cas de doute, il ne faut pas hésiter à apporter des modifications au programme et à recommencer.

L'opération :

```
props.put("myProperty", "true");
```

prend donc 0,765 microseconde pour une moyenne de 1 530 ms en effectuant 2 000 000 fois l'opération. Une autre valeur, beaucoup plus parlante, serait de donner le nombre d'opérations par seconde. Nous en aurions ici 1 307 189. Il serait donc possible d'utiliser cette valeur pour des estimations de calcul pour des applications demandant un grand nombre d'accès ou d'opérations.

Sur un Intel Quad Core Q9450 (début 2008), la valeur ci-dessus de 1 530 ms passe à 125 ms. Il y a sans doute aussi des aspects de performances liés à la version de la machine virtuelle Java.

Les outils en C++

Dans les bibliothèques C et C++ à disposition pour la mesure du temps, il y a aussi une fonction pour retourner l'horloge en millisecondes, mais elle n'est pas toujours à disposition sur toutes les machines et tous les systèmes d'exploitation. Cependant, la fonction C `time()` devrait être disponible sur n'importe quel système. La classe `Mesures` ci-dessous devrait donc fonctionner aussi bien sous DOS que sous Linux. Nous avons ajouté un `s` au nom de la classe. Il nous indiquera que nous travaillons en secondes. La classe suivante, `Mesure`, nous donnera une précision en millisecondes, qui sera plus favorable pour nos exemples et notre patience. Voici donc la première, notre classe `Mesures` :

```
// Mesures.h
#include <ctime>
#include <iostream>
```

```
#include <string>
#include <sstream>

class Mesures {
  private:
    long stime1, stime2, resultat;

  public:
    inline void debut() {
      stime1 = std::time(NULL);
    }

    inline void fin() {
      stime2 = std::time(NULL);
      resultat = stime2 - stime1;
    }

    inline std::string toString() {
      std::ostringstream os;
      os << "Le temps écoulé: " << resultat << " secs" << std::endl;

      return os.str();
    }
};
```

Comme toutes les méthodes de la classe C++ `Mesures` sont `inline`, il n'est pas nécessaire de définir de code supplémentaire dans un fichier `Mesures.cpp`. Nous passons directement à un petit programme de test :

```
// TestMesures.cpp
#include <cmath>
#include "Mesures.h"

using namespace std;

int main()
{
   cout << "Test de la classe C++ Mesures" << endl;

   Mesures mes1;
   mes1.debut();   // signe le début de la mesure

   for (int i = 0; i < 5000000; i++) {
     sin(cos(sinh(cosh(sqrt(i)))));
   }

   mes1.fin();    // signe lae fin de la mesure

   cout << mes1.toString() << endl;
   return 0;
}
```

Et le résultat de ce test farfelu, combinant les fonctions mathématiques du sinus, cosinus, sinus hyperbolique, cosinus hyperbolique et racine carrée, nous donnera ceci :

```
Test de la classe C++ Mesure
Le temps écoulé: 3 secs
```

Un temps de 3 secondes, pour des mesures statistiques convenables, ne serait pas suffisant, car le résultat n'aura une précision qu'à la seconde près. Il faudrait alors multiplier la boucle par 10, voire par 100, et attendre jusqu'à plusieurs minutes pour l'obtention d'une mesure qu'il nous faudra encore et encore répéter.

Par chance, le compilateur que nous avons à disposition nous permet aussi de retourner le temps en millisecondes ; voici le code de notre nouvelle classe `Mesure` (sans `s`) :

```
// Mesures.h
#include <ctime>
#include <iostream>
#include <string>
#include <sstream>

class Mesures {
  private:
    long stime1, stime2, resultat;

  public:
    inline void debut() {
      stime1 = std::time(NULL);
    }

    inline void fin() {
      stime2 = std::time(NULL);
      resultat = stime2 - stime1;
    }

    inline std::string toString() {
      std::ostringstream os;
      os << "Le temps écoulé: " << resultat << " secs" << std::endl;

      return os.str();
    }
};
```

Si nous adaptons le code ci-dessus pour la classe `TestMesures` :

```
// TestMesures.cpp
#include <cmath>
#include "Mesures.h"

using namespace std;

int main()
{
```

```
    cout << "Test de la classe C++ Mesures" << endl;

    Mesures mes1;
    mes1.debut();   // signe le début de la mesure

    for (int i = 0; i < 5000000; i++) {
      sin(cos(sinh(cosh(sqrt(i)))));
    }

    mes1.fin();    // signe lae fin de la mesure

    cout << mes1.toString() << endl;
    return 0;
}
```

nous obtiendrons le résultat en millisecondes cette fois-ci :

```
Test de la classe C++ Mesure
Le temps écoulé: 2688 millisecs
```

Ce résultat est nettement plus dans des normes convenables de calcul statistique.

Gagner en performance : une rude analyse

À l'aide d'un petit exemple pratique, nous allons comprendre qu'une analyse détaillée du code et des résultats de performance est absolument essentielle avant de prendre mot pour mot des arguments sortis tout droit d'ouvrages de référence ou de la bouche des gourous informaticiens. Il est souvent nécessaire de développer un prototype avant de prendre des décisions de conception et de code.

Au chapitre 15, nous avons découvert le mot-clé `final` en Java, pour l'appliquer à des méthodes. Les compilateurs Java vont mettre en œuvre le code nécessaire pour inclure le code en ligne.

Notre classe `DesMath` va nous calculer les fonctions mathématiques du sinus, du cosinus et de la tangente pour différentes valeurs, dans une boucle qui se répétera 2 millions de fois :

```
import java.lang.Math;

public class DesMath {
  private double sin;
  private double cos;
  private double tan;
  private int    compteur;

  public final void calcule(double nombre) {
    sin = Math.sin(nombre);
    cos = Math.cos(nombre);
    tan = Math.tan(nombre);
    compteur++;
  }
```

```
  public int getCompteur() {
    return compteur;
  }


  public static void main(String[] args) {
    DesMath desmath = new DesMath();
    Mesure  mesure  = new Mesure();

    mesure.start();
    for (double i = 0; i <= 2*Math.PI; i += 0.000002) {
      desmath.calcule(i);
    }
    mesure.end();
    System.out.println(mesure);

    System.out.println("Le compteur est " + desmath.getCompteur());
  }
}
```

Nous verrons, au travers de cet exemple, une foule de petits détails qui sont extrêmement importants à considérer lorsqu'il faut analyser un résultat de performance. Une petite erreur peut en effet conduire à une analyse totalement erronée.

En fin de programme, nous aurons le compteur imprimé avec la valeur de 3 141 593 ($1\ 000\ 000 \times \pi$) ! Est-ce vraiment un nombre étrange ? Il est d'abord très grand, et c'est important, surtout si nous tournons le programme sur une machine très rapide. À l'inverse, sur une machine trop lente, nous pourrions attendre plusieurs minutes inutilement. La raison de prendre le `2*Math.PI` est essentielle, car toutes les valeurs possibles du sinus et du cosinus seront testées. Il est fondamental de savoir qu'aussi bien le sinus que le cosinus prendront des temps d'exécution différents suivant la valeur de l'angle. Il faut aussi noter que si la méthode `calcule()` n'était utilisée que pour des angles très petits, comme pour la déviation d'une fusée Ariane au départ, il faudrait certainement revoir notre méthode de calcul de performance.

En exécutant ce code plusieurs fois, nous obtiendrons un résultat régulier autour de 780 ms (Intel Core Quad Q9450 [début 2008]).

À présent, si nous déclarons la méthode `calcule()` comme `final` de cette manière :

```
public final void calcule(double nombre) {
```

le code sera compilé en ligne. Ce qui veut dire que les quatre instructions à l'intérieur de la méthode `calcule` seront en fait dans la boucle `for()`. Nous économisons ainsi 3 141 593 appels de méthode. Comme résultat, nous n'obtiendrons aucune différence. Sommes-nous surpris ? Certainement, car c'est un sujet qui revient relativement souvent dans les ouvrages spécialisés ! Nous pourrions aussi jouer avec une méthode `calcule()` non finale et vide de code, pour montrer qu'un appel de méthode est de toute manière extrêmement rapide et qu'il est inutile de s'attarder sur ce point de détail. Sur une station

Ultra 5 de Sun Microsystems, nous avons constaté une amélioration d'environ 0,1 % pour la méthode `final`.

Que peut apporter une meilleure analyse ?

Si nous reprenons notre code précédent et faisons une analyse plus précise, nous allons découvrir que notre :

```
tan = Math.tan(nombre);
```

pourrait être corrigé en :

```
tan = sin/cos;
```

puisque la tangente est bien la division du sinus par le cosinus. Si nous changeons le code en gardant le `final` et le recompilons, nous obtiendrons 1 150 ms ! C'est presque invraisemblable, puisque globalement nous gagnons un facteur 4 ! À nouveau sur une station Ultra 5, nous n'avons constaté qu'une amélioration de 22 % ! Il y a donc des variantes selon les différentes machines virtuelles, les processeurs et éventuellement les coprocesseurs mathématiques.

Nous pourrions pousser le jeu à écrire :

```
cos = Math.sqrt(1 - (sin*sin));
```

et constater que cette opération nécessite beaucoup trop d'opérations et qu'une partie du bénéfice gagné par notre `tan = sin/cos` serait perdue !

Enfin, il ne faudra pas oublier les coûts pour une telle analyse et surtout pas écrire du code qui pourrait devenir délicat à maintenir car trop complexe. Les différences de performance sur d'autres systèmes sont aussi un point délicat qui pourrait remettre en question certaines de nos conclusions pour des produits livrables sur différentes plates-formes.

Passage par valeur ou par référence en C++

Nous avons examiné, au chapitre 6, les différentes manières de passer des paramètres aux méthodes C++. Nous avions alors recommandé l'utilisation du passage par référence, et, puisque nous avons à présent des outils à disposition, nous allons le constater par les chiffres. Nous avons aussi ajouté le cas du passage par pointeur, afin de vérifier cette autre possibilité. Il est important pour le programmeur de vérifier régulièrement ses préjugés. Pendant des années nous avons pris l'habitude de travailler avec les mêmes règles, qui sont parfois désuètes. Une remise en question n'est jamais interdite. Un processeur qui roule à 1 GHz et les révolutions sur les couches profondes des systèmes d'exploitation affectent non seulement les performances, mais surtout les rapports de performance des fonctions mesurées. De nouvelles instructions processeurs ou de nouveaux bus entre les différentes parties du système affecteront, sans aucun doute, nos idées préconçues.

Nous retrouvons ici notre classe `Personne`, réduite au minimum pour ces tests de performance.

```
// ValRef.cpp
#include <iostream>

#include <string>
#include "Mesure.h"

using namespace std;

class Personne {
  public:
    string method1(const string str) {
      return str.substr(1, 2);
    }

    string method2(const string &str) {
      return str.substr(1, 2);
    }

    string method3(const string *str) {
      return str->substr(1, 2);
    }
};


int main() {
  Personne pers;
  string resultat;
  string teststr("Bonjour Monsieur ! Comment allez-vous ?");

  Mesure mesure;

  mesure.debut();
  for (int i = 0; i < 2000000; i++) {
    resultat = pers.method1(teststr);
  }
  mesure.fin();
  cout << mesure.toString() << endl;

  mesure.debut();
  for (int i = 0; i < 2000000; i++) {
    resultat = pers.method2(teststr);
  }
  mesure.fin();
  cout << mesure.toString() << endl;

  mesure.debut();
  for (int i = 0; i < 2000000; i++) {
    resultat = pers.method3(&teststr);
  }
  mesure.fin();
  cout << mesure.toString() << endl;
}
```

Il est important d'avoir une boucle suffisante pour une bonne précision. Les deux résultats suivants, sur un PC datant de l'an 2000 :

```
Temps écoulé: 5880 millisecs
Temps écoulé: 5170 millisecs
Temps écoulé: 5050 millisecs

Temps écoulé: 5870 millisecs
Temps écoulé: 5100 millisecs
Temps écoulé: 5110 millisecs
```

extraits d'une série de mesures, sont donc satisfaisants. La grandeur de la boucle est suffisante pour garantir une bonne précision. Ce résultat nous donne donc le sentiment que notre recommandation pour une utilisation du passage par référence est tout à fait correcte. Il faut aussi reconnaître que le code, à l'intérieur de ces trois méthodes, ne correspond pas à du code réel d'applications conventionnelles. Nous aurions alors un rapport beaucoup plus faible entre les deux résultats (< 13 %). Cependant, c'est de toute manière un gain (environ 700 ms). Cette manière de faire est aussi plus élégante.

Sur un processeur plus récent (2008), nous obtenons des résultats avoisinant les 1 000 ms, mais la même analyse est applicable.

Nous n'avons pas essayé de comprendre la différence pour la dernière valeur, qui correspond plus ou moins à notre attente. C'est aussi le but des mesures de performance, qui devraient exiger une analyse plus profonde en cas de surprise. C'est loin d'être le cas ici, et cela confirme nos suspicions.

Performance et capacité mémoire

Nous devons tout de même mentionner les problèmes de performance qui pourraient se produire en C++ et en Java par une utilisation exagérée de la mémoire. Il faut aussi mentionner l'utilisation potentielle de mémoire virtuelle (*swap*) qui pourrait se produire et ralentir plus particulièrement des systèmes aux ressources limitées. Le développement d'outils permettant d'obtenir des statistiques d'utilisation de mémoire ou même de processeur est tout à fait réalisable. Nous n'en donnerons pas d'exemple, mais nous devons absolument revenir sur le programme précédent.

Lors de l'exécution de la méthode `method1()`, qui reçoit son paramètre par valeur, nous aurons besoin de mémoire supplémentaire. Dans le cas présent, c'est tout à fait insignifiant. Cependant, il faut se rendre compte que cette méthode pourrait être incluse dans une bibliothèque et utilisée par une multitude de processus parallèles. De plus, le `string` passé à cette méthode pourrait être d'une dimension importante et demander des ressources supplémentaires en mémoire virtuelle.

Les entrées-sorties

Nous allons remarquer, dans cette partie, que les entrées-sorties en programmation sont un des domaines les plus sensibles en ce qui concerne les performances. Si l'application

nécessite une certaine vitesse d'exécution, il est recommandé, durant la conception, de procéder à une phase d'analyse au travers de prototypes. Ces derniers nous donneraient vraisemblablement des indications pour une écriture plus performante de nos programmes d'entrées-sorties.

Un autre aspect peut être essentiel, celui de la machine et de son matériel. Certains systèmes peuvent être construits avec des logiciels, des disques ou des caches plus performants. Un résultat sur une machine X avec une version Y du compilateur ou de la machine virtuelle tournant sur la version Z du système d'exploitation pourrait donner des résultats totalement erronés sur lesquels il serait impossible de tirer des conclusions irréprochables. Les deux programmes qui suivent devraient être exécutés sur la machine et l'environnement où le produit sera finalement installé. Ceci uniquement si la conclusion des résultats de l'analyse de performance entraîne une amélioration significative et sans rendre le code illisible ni difficile à maintenir. Nous pensons qu'il est toujours possible de garder une écriture simple de ces programmes et surtout de documenter les parties les plus délicates et les raisons de certaines constructions au premier abord étranges.

Lecture de fichiers en C++

Notre programme de test de lecture va analyser les différences de performance suivant le nombre d'octets lus par appel de fonction. Les deux limites extrêmes sont respectivement un octet et le fichier entier transféré en mémoire. Ce dernier cas n'a pas de sens, car pour d'énormes fichiers nous toucherions alors à d'autres ressources du système comme la mémoire virtuelle (*swap*). Voici donc le programme qui utilise les `iostream`s du C++ tels que nous les avons étudiés au chapitre 9 :

```
// lecture.cpp
#include <iostream>
#include <sstream>
#include <fstream>
#include <string>

#include "Mesure.h"

using namespace std;

int main(int argc, char **argv) {
  Mesure mes1;
  int dim_bloc;

  if (argc != 3) {
    cerr << "Nombre d'arguments invalide" << endl;
    cerr << "lecture fichier dimension" << endl;
    return -1;
  }

  istringstream entree(argv[2]);
```

```
  entree >> dim_bloc;
  if (dim_bloc <= 0) {
    cout << "La dimension doit être > 0";
    return -1;
  }

  mes1.debut();
  ifstream infile(argv[1], ios::in|ios::binary);
  if (!infile) {
    cout << "Le fichier d'entrée n'existe pas";
    return -1;
  }

  char *tampon = new char[dim_bloc];
  int octets_lus;
  int total_lus = 0;

  for (;;) {                        // lecture par bloc
    infile.read(tampon, dim_bloc);
    octets_lus = infile.gcount();

    for (int i = 0; i < octets_lus; i++) {
      total_lus++;
    }

    if (octets_lus < dim_bloc)  break;   // dernier bloc
  }

  infile.close();
  delete[] tampon;
  mes1.fin();
  cout << mes1.toString() << endl;

  cout << "Nombre d'octets lus: " << total_lus << endl;
}
```

Ce qui est important, avec la classe `Mesure` et ces deux méthodes `start()` et `end()`, c'est d'englober non seulement les blocs de code `for()`, qui vont consommer le maximum de temps, mais aussi l'allocation et la libération des ressources (`new` et `delete`). En cas de doute ou d'analyse plus pointue, il pourrait se révéler nécessaire d'ajouter d'autres instances de la classe `Mesure`. Nous devons aussi choisir une grandeur de fichier suffisante pour obtenir un résultat de l'ordre de la seconde.

Si nous utilisons la méthode `read(tampon, dim_bloc)` avec `dim_bloc` correspondant à un seul octet, nous aurons ce résultat sur un PC datant de l'an 2000 pour un fichier de 4 102 301 octets :

```
Temps écoulé: 1600 millisecs
Nombre d'octets lus: 4102302
Le tableau ci-dessous a été élaboré sur ce même PC et avec une moyenne calculée sur six
➥mesures pour chaque bloc. Lorsqu'une ou deux valeurs étaient en dehors des limites
```

```
➥précédentes, nous les avons parfois éliminées ou alors nous avons à nouveau procédé
➥à de nouveaux tests.
```

Tableau 15-1 Lecture en C++ avec des dimensions de bloc différentes

Bloc	1	2	3	4	10	50	64	128	1024	4096	40960
ms	1600	930	660	550	390	270	220	220	160-220	220 (un 160)	220

En conclusion, nous voyons immédiatement et distinctement une amélioration avec des blocs allant jusqu'à 64. Ensuite, cela devient nettement moins significatif. La figure correcte doit donc certainement se situer entre 512 et 4 096 octets.

Dans la réalité de la programmation, utiliser des tampons de 2, 3 ou 4 octets n'a aucun sens. Soit nous adoptons la lecture octet à octet, soit nous transférons par bloc en mémoire. Comme nous avons un facteur 10 avec des blocs aux alentours de 1 Ko, nous n'hésiterons pas. Cependant, il y a des situations où le logiciel se révèle plus complexe et nécessite la relecture de certains octets en arrière par rapport à la position courante. Il nous faudrait alors travailler avec deux tampons. La solution de relire en arrière avec un repositionnement dans le fichier (`seek`) pourrait être catastrophique pour les performances.

Il faut revenir sur notre 160 ms pour un fichier de 4 102 302 octets. À quoi correspond-il? En une seconde, cela nous donne plus de 25 Mo. C'est un résultat plus qu'honorable sur une machine équipée d'un disque Ultra DMA avec un transfert maximal de 33,3 Mo. Nous rappellerons qu'un CD-Rom 40X possède un taux de transfert de 6 Mo et qu'un disque Ultra2-Wide-SCSI peut aller jusqu'à 80 ou 160 Mo par seconde. Suivant les applications à concevoir, ces chiffres restent des figures essentielles et que ce soit avec un PC datant de 2000 ou de 2008, le résultat de l'analyse est identique.

Influence de l'appel de fonctions successives

Il serait injuste de ne pas revenir sur le programme précédent sans considérer les appels successifs de la méthode `read()`. Cette remarque est aussi applicable en Java. Nous pourrions donc remplacer les deux lignes suivantes :

```
infile.read(tampon, dim_bloc);
octets_lus = infile.gcount();
```

par deux fonctions internes vides de code :

```
read_test(tampon, 1);
octets_lus = gcount_test();
```

dont nous trouverons le code complet sur le CD-Rom, dans le fichier `lecture2.cpp`.

Si nous exécutons `lecture2.exe` en répétant 5 000 000 de fois ces deux instructions, nous obtiendrons un résultat de 31 ms, ce qui est loin d'être insignifiant. Cette valeur de 31 ms sera évidemment beaucoup plus élevée sur un processeur moins récent.

Il faut remarquer que `read_test()` et `gcount_test()` sont vides et devraient contenir le code nécessaire pour transférer les données depuis le système d'exploitation, qui lui-même doit aussi, de temps à autre, accéder au disque ! Il est donc de toute manière plus performant de réduire le nombre d'appels de la méthode `read()` de la classe `ifstream`, même si cette dernière garantit déjà un transfert par bloc (*buffering* ou cache).

Lecture de fichiers en Java

Le programme suivant est conçu sur le même modèle que le programme en C++. Le nom du fichier et la dimension des blocs de lecture sont des arguments passés à l'application.

```
import java.io.*;

public class Lecture {
 public static void main(String[] args) {
  if (args.length != 2) {
    System.err.println("Nombre d'arguments invalide");
    System.err.println("Java Lecture fichier dimension");
    return;
  }

  int dimBloc;

  try {
   dimBloc = Integer.parseInt(args[1]);
  }
  catch(NumberFormatException nfe) {
    System.err.println("Java Lecture fichier dimension");
    System.err.println("dimension doit être un nombre");
    return;
  }

  if (dimBloc < 1) {
    System.err.println("La dimension doit être > 0");
    return;
  }

  byte[] tampon = new byte[dimBloc];

  int octetsLus;
  int totalLus = 0;

  Mesure mesure1 = new Mesure();
  mesure1.start();

  File inFile = new File(args[0]);
  try {
    FileInputStream in = new FileInputStream(inFile);

    for (;;) {
```

```
        octetsLus = in.read(tampon);

        for (int i = 0; i < octetsLus; i++) {
          totalLus++;
        }

        if (octetsLus < dimBloc)  break;   // dernier bloc
      }

      for (int j = 1; j < dimBloc; j++) {
        in.read();
        totalLus++;
      }

      in.close();
    }
    catch (IOException e) {
      System.err.println("Le fichier d'entrée n'existe pas");
      return;
    }

    mesure1.end();
    System.out.println(mesure1);
    System.err.println("Nombre d'octets lus: " + totalLus);
  }
}
```

Pour les premières mesures, nous n'avons pas utilisé le même fichier qu'en C++, car cela prenait trop de temps pour la lecture caractère par caractère.

Avec un bloc d'un seul octet :

```
Temps écoulé: 2520 ms
Nombre d'octets lus: 177008
```

nous obtenons un résultat presque catastrophique ! Pour comparer avec l'exemple et le résultat en C++ ci-dessus, nous devons exécuter la formule suivante, 2 520 / 177 008 × 4 103 302, qui nous donne 58 417 millisecondes. Nous sommes donc trente-six fois plus lents. Le tableau ci-dessous a été élaboré de la même manière que le tableau pour la lecture en C++ et sur un PC datant de l'an 2000.

Tableau 15-2 Lecture en Java avec des dimensions de bloc différentes

Bloc	1	2	3	4	10	50	64	128	1024	4096	40960
ms	2520	1310	840	630	275	65	55	650*	160*	150*	460*

(*) Les mesures avec un fichier de 177 008 octets devenaient trop petites et imprécises. Nous avons donc repris notre fichier de 4 103 302 octets, qui est plus de vingt-trois fois plus grand.

En résumé, le résultat est différent en Java, où nous constatons une progression plus constante. Nous obtenons même une valeur inférieure à la version C++ pour 4 096 octets ! C'est un résultat assez surprenant, mais beaucoup moins que pour 40 960, où un autre phénomène se produit. Nous n'avons pas cherché à en connaître la raison, mais cette dégradation était encore plus significative sur une station Sparc de Sun Microsystems sous Solaris.

Tests globaux de performance

Avec system() en C++

À la livraison du JDK 1.3.1 de Java, nous avons vu apparaître un nouveau compilateur plus performant, selon Sun Microsystems, mais aussi la présence de l'ancienne version, nommée `oldjavac`. Les résultats ne nous ont pas convaincus et nous devons dire, à notre décharge, que nous n'avons pas cherché à en découvrir les vraies raisons. En conséquence, nous ne présenterons pas ces résultats.

Ce qui est important ici, c'est l'utilisation de la fonction C `system()`, qui nous permet d'exécuter une commande système à l'intérieur d'un programme C++. Voici donc un exemple d'utilisation de la fonction `system()` pour mesurer les performances de compilateurs Java :

```
// TestJava.cpp
#include "Mesure.h"

using namespace std;

void unTest(const char *cmd) {
  Mesure mes1;
  mes1.debut();

  system(cmd);

  mes1.fin();
  cout << cmd << ". " << mes1.toString() << endl;
}


int main()
{
   int i;

   cout << "Test de performance de différentes versions de javac" << endl;

   for (i = 0; i < 5; i++) {
     unTest("javac DesMath.java");
   }
```

```
    for (i = 0; i < 5; i++) {
      unTest("oldjavac DesMath.java");  // JDK 1.3.1 seulement
    }

    return 0;
}
```

Le résultat retourné va nous donner un chiffre qui correspond non seulement au temps d'exécution de la compilation, mais aussi au chargement du programme. Au retour de `system()`, la compilation est vraiment terminée, et le fichier `DesMath.class` est bien régénéré et sauvé sur le disque. Il y a donc plusieurs facteurs qui peuvent affecter les performances, comme la vitesse du processeur ou le temps d'accès au disque.

Les résultats de ce type de test pourraient nous donner des indications sur la rapidité des compilateurs dans un environnement déterminé. Les conclusions pourraient être utilisées pour améliorer l'infrastructure et l'efficacité pour un groupe de développeurs dans une entreprise. Souvent, il suffit simplement d'ajouter de la mémoire vive. Nous rappellerons que ce n'est jamais une bonne idée d'éditer de gros fichiers de code sans compilation intermédiaire, car des fautes stupides de syntaxe peuvent nécessiter pas mal de temps à être corrigées.

Avec exec() en Java

Nous allons faire le même travail en Java, c'est-à-dire exécuter une compilation C++ depuis un programme Java et obtenir le temps nécessaire à ce processus. Voici donc le programme en Java, qui est un peu plus complexe que sa version C++ :

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

public class ExecJava {
  public static void main(String args[]) {
    try {
      Process leProcessus;
      String[] cmd = {"g++", "-o", "lecture.exe", "lecture.cpp"};

      System.out.println("Nous allons exécuter: g++ -o lecture.exe lecture.cpp");

      Mesure  mesure  = new Mesure();
      mesure.debut();

      leProcessus = Runtime.getRuntime().exec(cmd);
      try {
        System.out.println("Le résultat: " + leProcessus.waitFor());
      }
      catch(InterruptedException ie) {
        System.err.println("Processus interrompu" + ie);
      }
```

```
      mesure.fin();
      System.out.println(mesure);
    }
    catch(IOException ioe) {
      System.err.println("Processus non exécuté" + ioe);
    }
  }
}
```

Ici, nous présenterons le résultat :

```
Nous allons exécuter: g++ -o lecture.exe lecture.cpp
Le résultat: 0
Temps écoulé: 594 ms
```

La méthode `exec()` nous permet d'exécuter un processus parallèle. Elle est différente de la fonction C `system()`, qui terminera son exécution avant de nous redonner la main. Des fonctions `exec()` en C existent aussi, mais sont fortement dépendantes des systèmes d'exploitation. Ces dernières sont cependant très souvent employées en programmation sous Linux.

La méthode `getRuntime()` est essentielle pour obtenir un objet de la classe `Runtime`. Elle nous permettra d'obtenir toutes les caractéristiques nécessaires pour pouvoir exécuter un processus parallèle avec la commande `exec()`. Cette dernière méthode doit recevoir un tableau de `String` avec la commande et ses paramètres. Elle nous retournera un objet de la classe `Process` sur lequel nous pourrons appliquer la méthode `waitFor()`. Celle-ci nous retournera le contrôle lorsque la compilation sera terminée.

Nous pourrions jouer avec `lecture.cpp` et provoquer une erreur. Dans ce cas, le code retourné serait `1`, mais aucun message d'erreur de compilation n'apparaîtrait. Nous laisserons cette partie plus délicate aux programmeurs désirant approfondir ce sujet. Nous dirons simplement que ce type de traitement est parfaitement possible et qu'il nous fait penser aux erreurs qui apparaissent dans notre console de résultat avec notre éditeur Crimson.

Autres calculs de performance ou de contraintes

Nous aurions pu reprendre une ou plusieurs collections du chapitre 14. C'est une source infinie de combinaisons et d'alternatives. Nous ferons juste un exercice, le dernier, avec la classe `vector` en C++, afin de reprendre les techniques utilisées dans ce chapitre.

Lorsque nous parlons de collections, nous pensons très rapidement à des données persistantes que nous transférerons dans des fichiers ou des bases de données SQL. Ce dernier sujet sera traité au chapitre 20. Si nous devions réserver une place dans un avion, il ne serait pas judicieux de créer une collection en mémoire. Le client peut changer d'avis, prendre son temps ou se faire piquer sa place côté fenêtre par un autre individu de l'autre côté de la planète. Il n'y a aucun aspect de performance de calcul à considérer, mais d'autres contraintes comme l'accès en temps réel. En revanche, s'il faut analyser, convertir ou

mettre à jour globalement une base de données, un transfert dans des collections pour des accès rapides est de toute manière un bon choix.

Lorsque nous transférons un fichier sur Internet, nous devons nous poser la question s'il faut le compresser ou non. Une analyse de performance nous aidera certainement. Celle-ci pourrait se révéler totalement inutile, si le consommateur exige des factures de téléphone minimales ou des accès rapides à d'autres services en parallèle.

Obtenir des évidences, par des tests de capacité et de performance, est une nécessité absolue.

Résumé

Avant de prendre des décisions trop rapides en vue d'améliorer les performances du code, il est donc essentiel de procéder à une analyse détaillée du code. Un certain pragmatisme est nécessaire pour gagner en efficacité et en simplicité. De bons outils permettront d'identifier des améliorations invisibles lors d'une première lecture.

Exercices

1. Réécrire les deux classes Java et C++ `Mesure` avec deux constructeurs. Ils recevront tous les deux une identification de ce qui est mesuré et, pour le deuxième, un paramètre qui correspond au nombre de répétitions. La méthode `toString()` devra retourner, en plus du résultat, l'identification. Si le deuxième constructeur est utilisé, un second résultat sera présenté, non plus en millisecondes, mais en microsecondes et en virgule flottante, après la division par le nombre de répétitions.
2. Reprendre la classe `vector` en C++ du chapitre 14 et insérer un grand nombre d'entiers avec `push_back()`, `insert(vector_instance.end(), i)` et `insert(vector instance.begin(), i)`. Déterminer les performances de chacune de ces méthodes. Pour la meilleure des implémentations, vérifier si l'allocation prédéfinie de la dimension du `vector` apporte ou non une amélioration.

16

Comment tester correctement ?

Le Y2K bug

Le fameux bug de l'an 2000 a-t-il été la plus grande arnaque économique de ces dernières années, comme certains l'affirment ? Dans une certaine mesure, on peut répondre oui. Cependant, si nous examinons les côtés positifs de cet énorme travail de contrôle, il nous faut retenir toutes ces heures de travail où les informaticiens ont planché sur le problème. Ils ont essayé toutes sortes de tests et de combinaisons farfelues, dans le but de trouver une faille ou encore à des fins de prestige personnel ! Ce travail a souvent été élaboré de l'extérieur, par de simples tests de fonctionnalités, comme celle de changer la date sans savoir comment était structuré le programme informatique. D'autres ingénieurs sont allés plus profondément dans le code des applications ou des systèmes d'exploitation. Ils ont écrit des outils de test, par exemple la vérification de certaines fonctions des API (*Application Programming Interface*). En Java et en C++, nous pensons aux classes qui accèdent à la date et à l'heure et qui ont dû être examinées avec une attention redoublée.

Une stratégie de test dès la conception

Le message que nous aimerions faire passer, afin de sensibiliser le programmeur, est de ne pas attendre que l'application soit terminée pour se poser la question de savoir comment la tester. Dès que le programmeur commence à coder, il doit avoir défini une stratégie de développement et une structure pour les tests de ces programmes et de ces modules. Il doit avoir aussi à sa disposition les outils nécessaires. Il y a ainsi parfois des situations où il faudra allouer des ressources dans le but unique d'écrire des outils de test.

Il est aussi essentiel de penser à pouvoir répéter ces tests lorsque le produit sera terminé. Ce dernier aura sans doute été livré sans les modules de test. Il faudra donc définir des moyens pour collectionner les données au moment du problème afin de pouvoir simuler et reproduire les fautes dans un environnement de tests. Chaque interface interne devra pouvoir être vérifiée séparément. Il faudra aussi penser à des tests automatiques, dont nous constaterons les résultats après une nuit de sommeil ou un week-end de ski.

Lors de la définition, dans une entreprise, d'un projet informatique, il est essentiel de mettre au point une stratégie de développement et les différentes phases du projet. Nous devrions y retrouver non seulement la définition et la spécification des tests, mais aussi les résultats sous forme de documents officiels. Il faudra aussi, dans des cas particuliers, investir pour acheter de nouveaux outils ou du matériel spécifique comme des simulateurs.

Avec ou sans débogueur

C'est un sujet controversé. Dans l'annexe E, nous parlerons de NetBeans, un outil de développement qui intègre un débogueur permettant du pas à pas dans un programme Java ou C++. Dans des cas bien spécifiques, il pourra nous aider à identifier notre problème tout en gagnant du temps, c'est tout !

De l'avis de l'auteur, nous pensons que de grosses applications ou projets qui nécessitent absolument l'utilisation d'un débogueur sont simplement mal écrits. C'était autrefois le cas avec du code procédural et des programmeurs qui n'avaient aucune idée de l'écriture en modules, même si ceux-ci n'étaient pas encore orientés objet ou encore élaborés avec des outils de conception (UML).

Les tests de base

Les tests de base (*basic tests* en anglais) sont essentiels en programmation. Dans cet ouvrage, nous nous concentrerons sur ces derniers car ils sont souvent négligés par les informaticiens, au profit des tests d'intégration, de fonctions ou de système, qui viennent en principe beaucoup plus tard dans les phases de développement.

Tester toutes les possibilités d'interface interne d'un programme est du domaine de l'impossible. Nous allons commencer par un petit exercice de réflexion.

Problème : écrire une méthode ou fonction qui extrait un nombre de caractères `n` dans une chaîne depuis la position `pos`. Ces caractères ne peuvent être que des chiffres de 0 à 9.

Test : écrire un programme de test qui contrôle le maximum de conditions possibles.

La première remarque qui nous vient à l'esprit est le fait que nous posions le problème globalement. Nous sommes en fait en train de définir tous les détails de la conception de notre méthode, et c'est loin d'être un mal. Avons-nous oublié un détail ? Bonne question ! C'est l'une des raisons pour lesquelles les entreprises ou sociétés informatiques organisent des inspections de design et de code. Bien que les développeurs soient responsables des

tests de base, nous ne donnons pas en général aux mêmes personnes la tâche de décrire et d'exécuter les tests de fonctions.

La fonction extraction() en C++

Voici donc, en C++, une implémentation possible d'une fonction pour résoudre notre problème :

```
bool extraction(const char *original, char *tampon, int pos, int nombre) {
  if ((original == 0) || (tampon == 0))  return false;
  if ((pos < 0) || (nombre < 0)) return false;

  int longueur;
  int dernier = pos + nombre;

  for (longueur = 0; ; longueur++) {
    if (longueur >= dernier)  break;
    if (original[longueur] == 0) {
      if (longueur < dernier)  return false;
    }
  }

  int i;
  char chiffre;
  for (int i = pos; i < dernier; i++) {
    chiffre = original[i];
    if ((chiffre < '0') || (chiffre > '9'))  return false;
  }

  for (int i = 0; i < nombre; i++) {
    tampon[i] = original[i + pos];
  }

  return true;
}
```

Les morceaux de code source présentés ici font tous partie du fichier `Extraction.cpp` des exemples de ce chapitre. Ce code vraiment particulier est loin d'être évident au premier coup d'œil. Nous devons analyser chaque ligne de code pour en comprendre le sens. Nous aurions pu utiliser une simple fonction C ou la classe `string` du Standard C++ pour faire le travail. Nous avons choisi ici de tester le plus grand nombre de cas d'erreurs possibles et certainement d'avantager aussi la performance. Les deux premiers tests :

```
if ((original == 0) || (tampon == 0))  return false;
if ((pos < 0) || (nombre < 0)) return false;
```

pourraient en fait être supprimés à la livraison du produit, à condition d'avoir exécuté suffisamment de combinaisons de tests. C'est ici que nous pourrions faire la remarque de l'importance d'effacer certaines variables en cas de réutilisation. Dans le code suivant :

```
  char *temporaire = new char[20];
  if (extraction("a56d", temporaire, 1, 2) …
```

```
  ….
  delete[] temporaire;
  temporaire = 0;
  ….
  if (extraction("b67ef", temporaire, 1, 2) …
```

si nous omettions le :

```
temporaire = 0;
```

la mémoire adressée par cette variable pourrait être réutilisée, alors qu'elle pointe sur une position de mémoire libérée et peut-être déjà réallouée par cette application ou une autre !

Enfin, il nous faut faire remarquer que les deux dernières boucles `for()` pourraient être combinées. Mais cela signifierait une autre définition de l'interface, où le tampon de retour pourrait avoir été utilisé, même en cas d'erreur. C'est certainement à éviter, et notre solution est plus convenable.

Le programme de test de la fonction extraction ()

Il nous faut passer à présent à la phase de test. Le programme suivant est avant tout un exemple pour sensibiliser les débutants dans cette phase extrêmement complexe et coûteuse. En effet, il nous a fallu beaucoup plus de temps pour écrire et vérifier le programme de test que pour développer la fonction elle-même ! Cependant, ce programme nous a aussi autorisé à reprendre notre code précédent pour l'améliorer en conséquence.

```
void init_test(char *tampon, int size) {
  for (int i = 0; i < size; i++) {
    tampon[i] = 0;
  }
}

bool doit_etre_vide(const char *tampon, int size, int pos) {
  for (int i = pos; i < size; i++) {
    if (tampon[i] != 0) return false;
  }

  return true;
}

void print(const char *tampon, int size) {
  for (int i = 0; i < size; i++) {
    cout << tampon[i];
  }
  cout << endl;
}
```

Les fonctions `init_test()`, `doit_etre_vide()` et `print()` nous permettent d'écrire un code plus compact. C'est aussi une manière simplifiée de vérifier que notre méthode `extraction()` ne va pas déborder ni produire des résultats inattendus. Il est pratiquement impossible de

tester tous les cas. Le but est d'atteindre une certaine confiance. Si une erreur est découverte, non seulement nous changerons le code, mais nous ajouterons aussi de nouveaux tests.

```
int main() {
  char tampon[20];

  if (extraction(0, 0, 0, 0) == true) { cout << "Test1a: erreur" << endl; }

  if (extraction("abcd", 0, 0, 0) == true) { cout << "Test2a: erreur" << endl; }

  if (extraction("abcd", tampon, -1, 0) == true) {  cout << "Test3a: erreur" << endl; }

  if (extraction("abcd", tampon, 0, -1) == true) {  cout << "Test4a: erreur" << endl; }


  init_test(tampon, 20);

  if (extraction("abcd", tampon, 0, 0) == false) { cout << "Test5a: erreur" << endl; }

  if (!doit_etre_vide(tampon, 20, 0)) { cout << "Test5b: erreur" << endl; }


  init_test(tampon, 20);

  if (extraction("abcd", tampon, 0, 5) == true) { cout << "Test6a: erreur" << endl; }

  if (!doit_etre_vide(tampon, 20, 0)) { cout << "Test6b: erreur" << endl; }


  init_test(tampon, 20);

  if (extraction("abcd", tampon, 5, 1) == true) { cout << "Test7a: erreur" << endl; }

  if (!doit_etre_vide(tampon, 20, 0)) { cout << "Test7b: erreur" << endl; }


  init_test(tampon, 20);

  if (extraction("abcd", tampon, 2, 4) == true) { cout << "Test8a: erreur" << endl; }

  if (!doit_etre_vide(tampon, 20, 0)) { cout << "Test8b: erreur" << endl; }


 init_test(tampon, 20);

  if (extraction("abcd", tampon, 0, 1) == true) { cout << "Test9a: erreur" << endl; }

  if (!doit_etre_vide(tampon, 20, 0)) { cout << "Test9b: erreur" << endl; }


  init_test(tampon, 20);
```

```
  if (extraction("abcd", tampon, 1, 2) == true) { cout << "Test10a: erreur" << endl; }

  if (!doit_etre_vide(tampon, 20, 0)) { cout << "Test10b: erreur" << endl; }


  init_test(tampon, 20);

  if (extraction("12345", tampon, 0, 1) == false) { cout << "Test11a: erreur"
  ➥<< endl; }

  if (!doit_etre_vide(tampon, 20, 1)) { cout << "Test11b: erreur" << endl; }


  print(tampon,1);
  init_test(tampon, 20);

  if (extraction("12345", tampon, 2, 3) == false) { cout << "Test12a: erreur"
  ➥<< endl; }

  if (!doit_etre_vide(tampon, 20, 3)) { cout << "Test12b: erreur" << endl; }


  print(tampon,3);
  init_test(tampon, 20);

  if (extraction("aaa9876zxx", tampon, 3, 4) == false) { cout << "Test13a: erreur"
  ➥<< endl; }

  if (!doit_etre_vide(tampon, 20, 4)) {  cout << "Test13b: erreur" << endl; }


 print(tampon,4);
}
```

Ce code est parlant de lui-même. Il n'a pas fonctionné à la première écriture, et il a fallu jouer avec les `!`, `true` et `false` correctement. Nous remarquerons le cas d'un nombre de caractères à extraire ayant la valeur 0 : nous pensons convenable d'accepter ce cas, qui pourrait même simplifier l'utilisation de cette interface. Ce point de détail, qui peut avoir été découvert pendant les tests, est essentiel : il doit absolument faire partie de la documentation de la fonction `extraction()`.

En cas de réécriture complète de cette méthode `extraction()`, le code de test est réutilisable, sans changement, à condition que l'interface (l'API) reste identique.

Le programme de test extraction () en Java

La classe `Extraction` en Java et le programme de test seront réalisés en exercice. Nous avons affirmé, au début de cet ouvrage, qu'il pourrait se révéler intéressant d'ajouter une entrée `main()` dans toutes les classes à titre de vérification ou de test. Dans le cas de tests de cette complexité, ce n'est certainement pas une idée géniale d'adopter cette règle et de

livrer ce code avec ces tests. Il serait alors conseillé d'écrire des classes de test séparément. Lorsque nous examinons un tel code :

```
init_test(tampon, 20);

if (extraction("aaa9876zxx", tampon, 3, 4) == false) {
   cout << "Test13a: erreur" << endl;
}

if (!doit_etre_vide(tampon, 20, 4)) {
   cout << "Test13b: erreur" << endl;
}

print(tampon,4);
```

nous avons le sentiment que nous pourrions écrire une fonction pour combiner tous ces contrôles ! C'est ce que nous ferons comme exercice en fin de chapitre.

Suivre à la trace

Nous allons à présent écrire une classe, nommée `Traceur`, qui va nous permettre de tracer l'exécution de programmes.

Définition du problème

Nous aimerions être capables d'ajouter du code de test dans nos applications afin de pouvoir enregistrer des données pendant l'exécution d'un programme. Ces données seront disponibles sur le disque avec la date et l'heure, l'identification de l'enregistrement et des variables qui viennent du programme. Cette classe pourra être utilisée aussi bien pour identifier les cas d'erreurs que pour suivre à la trace les cas normaux. Elle pourrait avoir d'intéressantes applications pour produire des statistiques de performance ou d'utilisation de certaine partie du code que nous aimerions, par exemple, améliorer ou réutiliser dans le futur.

La classe Traceur en C++

Voici à présent une manière de faire, simplifiée au maximum et présentée au travers du fichier d'en-tête pour la définition de la classe `Traceur` :

```
// Traceur.h
#include <fstream.h>
#include <string>

class Traceur
{
  public:
    Traceur(const char *info, const char *nom_fichier);
    ~Traceur();
    void ecrit(const char *texte);
```

```
    void ecrit(const char *texte, int valeur);

  private:
    void commun();       // info fixe

    string tnom;         // le nom du traceur
    ofstream mon_ofstr;  // fstream pour écriture
    int numref;          // numéro de référence
};
```

Le constructeur va donc recevoir l'argument `info` qui sera utilisé pour identifier le traceur ainsi que le nom du fichier dans lequel les données seront écrites. Les deux méthodes `écrit()`, avec des arguments différents, vont nous permettre d'écrire des données depuis le programme. Nous aurions pu aussi ajouter d'autres méthodes comme :

```
void ecrit(const char *texte1, const char *texte2);
void ecrit(const int   valeur);
void ecrit(const char *texte, int valeur1, int valeur2);
```

ou encore avec d'autres types. Le `numref` va être un numéro de séquence que nous pourrions utiliser pour trier ou rechercher des données. Nous allons maintenant découvrir la présence de `<ctime>`, qui nous indique que nous avons bien l'intention d'enregistrer la date et l'heure. Nous passons donc au code proprement dit de la classe `Traceur` :

```
// Traceur.cpp
#include "Traceur.h"
#include <ctime>

using namespace std;

Traceur::Traceur(const char *info, const char *nom_fichier)
{
  tnom = "-" + string(info) + ": ";
  numref = 1;
  mon_ofstr.open(nom_fichier);
}

Traceur::~Traceur()
{
  mon_ofstr.close();
}

void Traceur::ecrit(const char * texte)
{
  commun();
  mon_ofstr << texte << endl;
}

void Traceur::ecrit(const char * texte, int valeur)
{
```

```
  commun();
  mon_ofstr << texte << " " << valeur << endl;
}

void Traceur::commun() {
 time_t maintenant;
 time(&maintenant);
 string lheure(ctime(&maintenant));

 mon_ofstr << string(lheure, 0, 24) << " " << numref++ << tnom;
}
```

C'est vraiment une simplification à l'extrême. Il n'y a pas de retour d'erreurs, et l'ouverture du fichier dans le constructeur n'est pas vraiment judicieuse. Le :

```
string(lheure, 0, 24)
```

est nécessaire pour éliminer le `"\n"` en fin de chaîne, car celui-ci nous est retourné par la fonction C `ctime()`. Le résultat dans le fichier pourra se présenter sous cette forme :

```
Tue Jul 08 11:05:28 2008 1-T2: test2a 2000000
```

`1-T2` représente la valeur `numref` suivie de l'argument donné au constructeur. `Compteur 2000000` vient des deux paramètres de la méthode :

```
ecrit(const char * texte, int valeur)
```

Nous voyons ainsi comment la méthode `commun()` va nous générer la première partie de l'enregistrement. Avant de revenir à un exemple avec la méthode `extraction()`, nous allons donner ici une liste d'améliorations ou d'options que nous pourrions introduire dans ce code :

- traiter les erreurs d'accès au fichier correctement ;
- ajouter les millisecondes à la méthode `commun()` ;
- être capable d'utiliser le même fichier pour plusieurs instances de la classe `Traceur` ;
- pouvoir enclencher ou déclencher l'enregistrement avec une variable d'environnement ;
- introduire un mécanisme pour visualiser (par exemple en Java [AWT ou Swing]) à la fois les enregistrements actifs (besoin de les mémoriser et d'un destructeur) et les données en temps réel.

Tester la classe Traceur en C++

Sur le CD-Rom, nous trouverons le programme de test `TraceurTest.cpp` :

```
// TraceurTest.cpp
#include "Traceur.h"
#include <iostream>
#include <cmath>

using namespace std;
```

```
int main() {
  Traceur t1("T1", "fichier1.txt");
  Traceur t2("T2", "fichier2.txt");

  cout << "Bonjour" << endl;

  t1.ecrit("test1a");

  const long cons = 2000000;

  for (long i = 0; i < cons; i++) {
    sin(cos(i));
  }
  cout << endl;

  t2.ecrit("test2a", cons);
  cout << "Au revoir" << endl;
  t2.ecrit("test2b");
}
```

Pour le compiler, il faudra utiliser le `Makefile` puisque la classe `Traceur.cpp` doit être compilée préalablement. En exécutant `TraceurTest.exe` nous recevrons ceci :

```
Bonjour

Au revoir
```

Il n'y a rien de bien surprenant, car il faudra encore consulter les fichiers de traçage `fichier1.txt` :

```
Tue Jul 08 11:05:28 2008 1-T1: test1a
```

et `fichier2.txt` pour examiner les traces demandées :

```
Tue Jul 08 11:05:28 2008 1-T2: test2a 2000000
Tue Jul 08 11:05:28 2008 2-T2: test2b
```

La classe Traceur en Java

Après avoir codé notre classe `Traceur` en C++, nous sommes arrivés à la conclusion que beaucoup d'améliorations pourraient lui être apportées. Dans la version Java, nous avons choisi de travailler différemment avec les fichiers et d'y ajouter quelques options comme celle de présenter la date et l'heure avec une précision montrant les millisecondes. Voici donc le code Java de la classe `Traceur`, que nous pourrions présenter comme un deuxième prototype avant de pouvoir élaborer une classe Java ou C++ encore plus professionnelle :

```
import java.io.*;
import java.util.Date;
import java.text.DecimalFormat;

public class Traceur {
```

```
static private int sequence = 0;
static private boolean tousActif = false;
static private String nomFichier = "trace";
static private PrintWriter traceout;

private boolean instanceActif = false;
private String  trNom;

public Traceur(String name) {
  trNom = name;

  if (sequence == 0) {
     sequence++;
     try {
       traceout = new PrintWriter(new FileWriter(nomFichier + sequence + ".trc"));
       if (sequence == 1) {
         tousActif = true;
       }
       instanceActif = true;
     }
     catch(IOException ioe) {
       if (sequence == 1) {
         sequence =0;
       }
       return;
     }
     tousActif = true;
  }

  instanceActif = true;
}

private void commun() {
  Date date = new Date();
  long milli = date.getTime()%1000;

  if (tousActif && instanceActif) {
    DecimalFormat df = new DecimalFormat("000");
    traceout.print(date + " " +  df.format(milli) + " " + trNom + " ");
  }
}

public void ecrit(String texte1) {

  if (tousActif && instanceActif) {
    commun();
    traceout.println(texte1);
    traceout.flush();
  }
}
```

```
  public void ecrit(String texte1, long valeur1) {
    if (tousActif && instanceActif) {
      commun();
      traceout.println(texte1 + " " + valeur1);
      traceout.flush();
    }
  }
```

Le traitement des erreurs est à peine plus sophistiqué que la version C++. Il manque encore le code nécessaire pour informer l'utilisateur d'un problème quelconque. Ce point pourrait être plus délicat, car l'utilisation de telle classe pour tracer une application a un sens principalement pour des processus détachés de la console, comme des applications client-serveur.

Les quatre variables statiques nous permettent d'utiliser les mêmes données pour plusieurs instances de la classe. À la création de la première instance de la classe `Traceur`, nous savons qu'il faut ouvrir le fichier en écriture. Cette manière de faire nous permettrait d'introduire de nouvelles méthodes pour fermer le fichier et en ouvrir un nouveau, tout en continuant les enregistrements pour les instances existantes. Le fichier sera donc toujours nommé `trace1.trc`, avec le code existant.

La méthode `commun()` nous enregistre la partie fixe, avec en plus le nom du traceur actif qui est passé dans le constructeur de la classe. Le calcul des millisecondes est intéressant :

```
long milli = date.getTime()%1000;
DecimalFormat df = new DecimalFormat("000");
df.format(milli);
```

Il faut en effet extraire la partie des millisecondes par le reste de la division par 1 000. Le « 000 » du `DecimalFormat` va nous ajouter les 0 nécessaires, si le résultat est plus petit que 100 ou encore plus petit que 10. Nous aurons donc un alignement de la présentation plus précis pour un traitement et une analyse éventuelle.

Notre classe `TestTraceur` représente une manière simple et suffisante pour vérifier notre code.

```
import java.lang.Math;

public class TestTraceur {
  public static void main(String[] args) {
    Traceur tr1 = new Traceur("Premier_traceur");
    Traceur tr2 = new Traceur("Second_traceur");

    tr1.ecrit("Début");
    System.out.println("Début");

    for (long i = 0; i <= 1000000; i++) {
      Math.sin(Math.cos(Math.tan(i)));

      if ((i%200000) == 0) {
        tr2.ecrit("Milieu", i);
```

```
        }
      }

      System.out.println("Fin");
      tr1.ecrit("Fin");
    }
  }
```

Voici maintenant le résultat présent dans le fichier `trace1.trc` :

```
Tue Jul 08 10:56:31 CEST 2008 801 Premier_traceur Début
Tue Jul 08 10:56:31 CEST 2008 832 Second_traceur Milieu 0
Tue Jul 08 10:56:31 CEST 2008 848 Second_traceur Milieu 200000
Tue Jul 08 10:56:31 CEST 2008 848 Second_traceur Milieu 400000
Tue Jul 08 10:56:31 CEST 2008 848 Second_traceur Milieu 600000
Tue Jul 08 10:56:31 CEST 2008 848 Second_traceur Milieu 800000
Tue Jul 08 10:56:31 CEST 2008 864 Second_traceur Milieu 1000000
Tue Jul 08 10:56:31 CEST 2008 864 Premier_traceur Fin
```

Les trois fonctions mathématiques ne sont là que pour ralentir le processus. Le traceur du milieu ne sera activé que toutes les 20 000 fois. Sans le caractère `=` du `<=` de la boucle `for()`, nous n'aurions pas la dernière trace du milieu !

Encore des améliorations pour notre traceur ?

Ce petit exercice pourrait certainement se prolonger jusqu'à la fin de la journée ou de la semaine ! Nous allons nous arrêter ici après avoir mentionné :

- la nécessité d'y ajouter un plus grand nombre de méthodes `ecrit()` ;
- le travail avec plusieurs fichiers simultanément, par exemple pour enregistrer séparément les cas normaux et les cas d'erreurs ;
- l'identification du processus actif, car plusieurs applications ou plusieurs instances de la même application pourraient travailler sur le même fichier (dans une version simplifiée, il suffirait de travailler avec un nom de fichier créé dynamiquement et unique) ;
- l'utilisation d'une variable d'environnement pour tester si le traceur est actif ou non.

Comme ce dernier point sera proposé en exercice, nous allons présenter le code pour tester une variable d'environnement. Comme il n'est pas possible de lire, depuis Java, une variable d'environnement générale DOS ou Linux, nous sommes obligés de passer cette variable au moyen du paramètre `-D` à la machine virtuelle. Pour une classe de test nommée `TraceurETest`, ceci peut se faire de cette manière :

```
java -Dtraceur=actif TraceurETest
```

Nous aurons donc une variable `traceur` qui aura la valeur `actif`. Si nous voulons lire et vérifier cette variable, nous devrons utiliser le code suivant :

```
String traceur_prop = System.getProperty("traceur");
if ((traceur_prop != null) && (traceur_prop.equals("actif"))) {
 ...
}
```

Cette solution n'est cependant pas des plus flexibles. Nous pourrions par exemple lire ce paramètre depuis un fichier ou encore créer un mécanisme de communication sans devoir stopper et relancer le programme.

Résumé

Avant d'écrire les premières lignes de code, le programmeur devra penser à la manière de tester ces programmes. Il faudra bien sûr vérifier chaque module et avoir la possibilité de modifier les paramètres et les conditions. Les tests devront pouvoir être répétés, non seulement durant les phases de développement et d'intégration avec les autres parties du système, mais aussi durant toute la vie du produit. D'autres outils de test devront être pensés pour dépister les erreurs. Ces derniers devront aussi être disponibles en dehors de l'environnement traditionnel de développement.

Exercices

1. Écrire en Java notre fonction C++ `extraction()` comme méthode d'une classe `Extraction` et tester les différents cas d'erreurs. Écrire une classe qui automatise les tests (`Test Extraction`)avec une méthode qui reçoit comme paramètre à la fois les arguments de la fonction `extraction()`, mais aussi le résultat attendu. Lors de l'extraction des caractères, au lieu de vérifier les limites de l'index, utiliser l'exception `StringIndexOutOfBounds Exception`.
2. Écrire en Java une classe de test `TraceurETest` qui possède un attribut de la classe `TraceurE`. `TraceurETest` est instancié avec un constructeur qui reçoit un `String` et un `int` comme arguments. Nous définirons une méthode avec un entier comme paramètre dont une valeur négative serait une erreur. Utiliser le `TraceurE` pour enregistrer toutes les informations. La nouvelle classe `TraceurE` possédera des méthodes particulières pour enregistrer et identifier les erreurs, ainsi que la lecture de la propriété `traceur`. Le traçage ne sera exécuté que si cette propriété (`System.getProperty()`) est marquée comme active (`java -D`).

17

Ces fameux patterns

Qui sont donc ces fameux patterns ?

Les Design Patterns peuvent être considérés comme une approche de design avec un catalogue de classes prédéfinies. L'ouvrage de référence est le fameux *Design Patterns* d'Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides, écrit en 1995. Cette définition de vingt-trois solutions pour des problèmes classiques rencontrés en programmation SmallTalk, C++ et plus tard Java a constitué une étape importante en analyse et programmation orientée objet. John Vlissides a écrit un certain nombre d'articles et de discussions qu'il a inclus dans un autre ouvrage : *Pattern Hatching*. Dans l'introduction de ce dernier ouvrage, nous trouvons une jolie réflexion : « Misconception 9 : The pattern community is a clique of elites », qui se trouve dans le neuvième sujet d'une liste de dix malentendus ou d'idées fausses. Même s'il serait injuste de déclarer les utilisateurs de ces patterns comme faisant partie d'une élite d'ingénieurs informaticiens, l'adoption de ce catalogue de classes ne peut être que bénéfique puisque cette technologie est largement reconnue et utilisée en SmallTalk, en C++ et plus récemment en Java. Un programmeur expérimenté qui découvre cette technologie se rend souvent compte qu'il a en fait déjà utilisé un ou plusieurs de ces patterns sans le savoir !

Les patterns Singleton et Observer

Dans notre ouvrage, nous avons choisi deux patterns bien spécifiques, mais loin d'être inintéressants, le Singleton et l'Observer. Le tableau 17-1 vous en donne les définitions.

Tableau 17-1 Définitions à connaître absolument

Intentions	
Singleton	S'assurer qu'une classe n'a qu'une seule instance et lui donner un point d'accès.
Observer	Définir une dépendance entre des objets et assurer que, quand un objet change d'état, toutes ces dépendances seront averties et mises à jour automatiquement.

Le Singleton ou le constructeur protégé

Comment pouvons-nous interdire la création de plusieurs instances d'une classe ? Il faut d'abord protéger le constructeur et ensuite définir une méthode statique qui va créer notre unique instance. Mais à quoi donc pourrait bien nous servir le pattern Singleton dans la vie de tous les jours ? Nous pourrions donner l'exemple d'une unité hardware utilisée par plusieurs composants, d'un serveur responsable d'imprimer et de distribuer le travail, d'une servlet Java ou encore d'une connexion à une base de données. Nous allons montrer ici deux classes similaires en Java et C++, mais avec une implémentation différente due aux différences du langage, très marquées dans ce cas précis. Nous commencerons par la présentation du pattern Singleton en Java.

Le Singleton en Java

```
public class MonSingleton {
  private static MonSingleton monSingle;
  private static int compteur;
  private static int nbAppel;

  private MonSingleton() {
    compteur = 1;
    nbAppel = 0;
  }

  public static MonSingleton obtientInstance() {
    if (monSingle == null) {
      monSingle = new MonSingleton();
    }

    nbAppel++;
    return monSingle;
  }

  public void augmente() {
    compteur++;
  }

  public String toString() {
    return "Compteur = " + compteur + " (" +  nbAppel + ")";
  }
}
```

Le constructeur privé est la partie essentielle de la classe `MonSingleton`. Il est en effet impossible d'utiliser cette construction :

```
MonSingleton ms = new MonSingleton().
```

car elle sera rejetée par le compilateur. La seule possibilité d'obtenir une instance de cette classe est d'utiliser une méthode statique comme `obtientInstance()`. La classe de test `TestMonSingleton` suivante :

```
public class TestMonSingleton {

  public static void main(String args[]) {
    MonSingleton ms1 = MonSingleton.obtientInstance();
    ms1.augmente();

    MonSingleton ms2 = MonSingleton.obtientInstance();
    ms2.augmente();

    System.out.println(ms1);
  }
}
```

nous retournera le résultat suivant :

```
Compteur = 3 (2)
```

Mais que se passe-t-il précisément ? Nous n'avons pas choisi de créer deux objets de la classe `MonSingleton`, mais avons obtenu une référence à un objet. `ms1` et `ms2` représentant bien le même objet. La deuxième valeur, 2, nous indique le nombre de fois que nous avons utilisé la méthode `obtientInstance()`. Le compteur est de 3, car nous avons commencé à 1 et appelé deux fois la méthode `augmente()`. Il aurait été tout à fait possible de créer l'instance de cette manière :

```
private static MonSingleton monSingle = new MonSingleton();
```

mais nous avons décidé de ne créer l'instance que lors de sa première utilisation, ce qui est tout à fait raisonnable et pratique dans certaines circonstances. Nous allons à présent passer à la version C++.

Le Singleton en C++

```
#include <iostream>
#include <sstream>
#include <string>

using namespace std;

class MonSingleton {
  private:
    static MonSingleton monSingleton;
    static int nbAppel;
```

```
    int compteur;

    MonSingleton(int le_compteur) : compteur(le_compteur) {
      nbAppel = 0;
    }

    MonSingleton(MonSingleton &);
    void operator=(MonSingleton &);

  public:
    static MonSingleton& obtientInstance() {
      nbAppel++;
      return monSingleton;
    }

    void augmente() {
      compteur++;
    }

    string toString() {
      ostringstream sortie;
      sortie << "Compteur = ";
      sortie << compteur << " (" << nbAppel << ")" << ends;

      return sortie.str();
    }
};


MonSingleton MonSingleton::monSingleton(1);

int MonSingleton::nbAppel(1);


int main() {
  MonSingleton& ms1 = MonSingleton::obtientInstance();
  ms1.augmente();

  MonSingleton& ms2 = MonSingleton::obtientInstance();
  ms2.augmente();

  cout << ms1.toString() << endl;
}
```

Le résultat est identique au programme Java ci-dessus, mais avec quel travail ! C'est vraiment ici que nous voyons à la fois les faiblesses et la puissance de chacun de ces deux langages. Il est inutile de philosopher afin de savoir pour lequel nous prêcherions. Ce qui est intéressant ici, c'est de découvrir, en quelques lignes de code, un concentré d'un certain nombre de différences entre ces deux langages. Nous commencerons par les deux déclarations :

```
MonSingleton(MonSingleton &);
void operator=(MonSingleton &);
```

Ces deux lignes de code ainsi que le constructeur normal :

```
MonSingleton(int le_compteur)
```

sont tous les trois déclarés privés. Nous empêchons alors toutes les formes possibles de création d'objets ou de copie de la classe `MonSingleton`. Puisque nous avons défini au moins un constructeur, il n'y aura pas de constructeur par défaut. Nous avons aussi les deux :

```
MonSingleton MonSingleton::monSingleton(1);
int MonSingleton::nbAppel(1);
```

qui nous initialisent les parties statiques de l'objet. Vive le C++ !

Dans le cas d'un Singleton responsable d'imprimer des documents, nous pourrions nous imaginer qu'il contrôle une série d'imprimantes, distribuant le travail, mais aussi désactivant les appareils hors service ou les mettant en veille pour la nuit. Le Singleton devrait alors gérer les ressources d'une manière centralisée.

Le pattern Observer

Java MVC : l'interface Observer et la classe Observable

Avant de passer d'une manière tout à fait naturelle à l'architecture MVC (*Model View Controller*), nous allons redonner la définition d'intention du pattern Observer : « Définir une dépendance de un à plusieurs objets et, lorsqu'un objet change d'état, assurer que toutes ces dépendances seront averties et mises à jour automatiquement. »

Prenons l'exemple d'un compteur. Il fait partie d'un modèle (le M de MVC), qui correspond en général à un ensemble de données. Si le modèle change, c'est-à-dire ici la valeur du compteur, nous aimerions que toutes les vues (les *views* ou les représentations, le V de MVC) soient averties du changement et apparaissent avec la nouvelle valeur. Ce compteur peut aussi faire partie d'un ensemble de valeurs qui sont utilisées pour représenter un graphique. Ce graphique peut très bien exister sous différentes formes et en même temps sur l'écran ou encore à des endroits différents sur le réseau.

Il nous reste encore le C de MVC, le contrôleur. Il représente par exemple le clic sur un objet graphique pour augmenter ou diminuer la valeur de notre compteur. Cela pourrait être aussi l'entrée complète d'une nouvelle valeur dans un champ à l'écran.

L'architecture MVC est essentielle et fait partie de la décomposition traditionnelle d'une application en programmation Smalltalk. Il ne faut pas oublier ici que certains affirment que Java hérite plus de Smalltalk que de C++. C'est certainement juste au niveau des concepts orientés objet, mais absolument pas pour la syntaxe du langage.

L'architecture MVC est aussi tout à fait présente en Java avec l'interface `Observer` et la classe `Observable`. Nous noterons ici que les développeurs Java de chez Sun Microsystems

ont pris quelques libertés, certainement avec raison, en intégrant et en rassemblant le View et le Controller pour des raisons de simplification et d'efficacité.

Le pattern Observer en Java

Nous allons maintenant présenter notre exemple de compteur ; nous le souhaitons observable par d'autres objets, afin que ces derniers soient avertis lorsqu'il change de valeur. Nous commencerons par la classe `CompteurObservable`, qui, comme son nom l'indique, hérite de la classe Java `Observable` :

```
import java.util.Observable;

public class CompteurObservable extends Observable {
  private int compteur;

  public CompteurObservable(int initiale) {
    compteur = initiale;
  }

  public int getValue() {
    return compteur;
  }

  public void augmente() {
    compteur++;
    setChanged();
    notifyObservers();
  }
}
```

Le constructeur devra recevoir une valeur initiale du compteur. La méthode `augmente()` n'a rien de particulier, si ce n'est que deux méthodes de sa classe de base seront appelées : `setChanged()` et `notifyObservers()`. Ces deux méthodes sont nécessaires pour que les objets dépendant de `CompteurObservable` puissent être avertis d'un changement d'état. Si nous regardons de plus près la classe `Observable`, nous allons y découvrir la méthode `addObserver(Observer o)`. Il y a donc un mécanisme pour ajouter des observateurs aux objets de la classe `Observable`.

Nous allons créer à présent un objet graphique fictif, que nous allons appeler `Compteur Graphique` :

```
import java.util.*;

public class CompteurGraphique implements Observer {
  CompteurObservable compteur;
  int forme;

  public CompteurGraphique(CompteurObservable leCompteur, int uneForme) {
    forme = uneForme;
    compteur = leCompteur;
```

```
    compteur.addObserver(this);
  }

  public void update(Observable ob, Object arg) {
    System.out.print("Montre le compteur " + compteur.getValue());
    System.out.println(" dans la forme " + forme);
  }
}
```

L'attribut `forme` n'est ici que pour identifier quel objet est actif. Cependant, nous pourrions l'utiliser pour définir différentes formes graphiques à la présentation de notre compteur. Une autre classe aurait aussi été une alternative. L'important ici est que la classe reçoit une référence à une instance de `CompteurObservable`. Après l'avoir conservée dans la variable compteur, nous appelons la méthode `addObserver()` avant d'ajouter un observateur de plus pour cet objet.

La méthode `update()` de l'interface `Observer` doit être codée et présente. S'il se passe un changement d'état de l'objet compteur, nous sortirons un message à titre de test. Nous ne reviendrons pas sur les paramètres de la méthode `update()`, qui ne sont pas utilisés ici.

Pour finir, nous écrirons une classe de test :

```
public class TestCompteur {

  public static void main(String[] args) {
    CompteurObservable compteur = new CompteurObservable(50);

    CompteurGraphique dessin1 = new CompteurGraphique(compteur, 1);
    CompteurGraphique dessin2 = new CompteurGraphique(compteur, 2);

    compteur.augmente();
  }
}
```

dont le résultat est attendu :

```
Montre le compteur 51 dans la forme 2
Montre le compteur 51 dans la forme 1
```

En effet, les deux objets (Views) `dessin1` et `dessin2` sont notifiés, si nous augmentons le compteur (Model).

Résumé

Ce chapitre peut être considéré comme une ouverture d'esprit à diverses technologies orientées objet. Nous espérons que les lecteurs feront d'autres découvertes dans ces catalogues de classes que sont les Design Patterns.

Exercices

1. Réécrire la classe `MonSingleton` en C++ avec une allocation dynamique de `monSingleton` (opérateur `new`) et en utilisant des pointeurs pour les objets. Nous ne nous poserons pas la question de l'utilité d'un destructeur dans ce cas précis.
2. Écrire en Java une classe `Singleton` nommée `MonIP` qui va conserver l'adresse IP de la machine. Cette adresse est dynamique et peut changer. Écrire deux classes `Application1` et `Application2` qui devront être notifiées lorsque le numéro d'IP change. Nous utiliserons pour cela le pattern `Observer`. Les constructeurs d'`Application1` et d'`Application2` ne reçoivent pas de paramètres. Créer deux instances de ces dernières, changer l'adresse IP au travers d'une méthode de la classe `MonIP` et vérifier que le mécanisme fonctionne correctement.

18

Un livre sur Java sans l'AWT !

Nous trouverons dans ce chapitre toutes les motivations qui nous ont entraînés à présenter tout de même ces composants d'interface graphique. Comme ceux-ci sont fortement dépendants de la machine, de la carte graphique ou du système d'exploitation, nous comprendrons que de tels outils et bibliothèques ne soient pas disponibles pour des applications C++. Les programmeurs Java et C++ désirant développer des applications GUI (*Graphical User Interface*) doivent en principe se tourner vers des outils tels que Visual C++ (Microsoft) ou NetBeans (voir annexe E). Microsoft met aussi à disposition des éditions Express de Visual Studio pour le C++ et le C#, qui permettent de développer des applications graphiques dans ces deux langages mais pour Windows uniquement.

L'AWT (*Abstract Window Toolkit*) de Java permet de développer non seulement des applications graphiques traditionnelles, mais aussi des applets qui sont exécutées à l'intérieur de pages HTML et ceci dans notre navigateur Internet favori.

Au chapitre 21, nous montrerons aussi un exemple d'application Java qui utilise les composants Swing. Ces derniers représentent une extension améliorée et plus récente des composants AWT. Nous pouvons faire ici un parallélisme avec les classes MFC (*Microsoft Foundation Classes*), qui présentent une analogie avec Swing, de la programmation GUI C++ sous Windows avec une approche plus orientée objet. Nous aurions pu laisser de côté les composants AWT et faire la même démarche avec les composants Swing. Cependant, ces derniers ne sont pas des plus rapides sur des machines anciennes avec peu de ressources CPU ou mémoire.

Pour ceux qui s'intéressent à l'histoire du développement de Java, une analyse de l'évolution de l'AWT depuis son origine est particulièrement intéressante. Certains aspects de l'implémentation ont été totalement revus et corrigés dans les dernières versions du langage Java. Nous ne donnerons pas plus de détails, sinon que nous ne parlerons ici que de la version 1.2 ou supérieure. Pour les programmeurs utilisant de l'ancien code Java,

nous conseillerons de compiler les programmes avec l'option `-deprecation`, afin d'identifier les méthodes et techniques dépréciées. Nous encouragerons aussi l'utilisation systématique des classes internes (*inner class*).

Pour les utilisateurs de Crimson, il faudra se référer à l'annexe C pour le problème relié à la « Capture output ».

Apprendre à programmer avec des applications graphiques

Si nous examinons cette application graphique dont nous allons très rapidement étudier le code en Java :

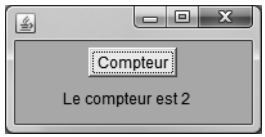

Figure 18-1
Notre premier compteur

nous savons qu'elle pourrait très bien être développée par un grand nombre de langages de programmation comme Visual Basic de Microsoft, Delphi (le Pascal de Borland), ou encore Visual C++ ou C# de Microsoft.

Dans cette application, le seul vrai problème concerne la tasse de café. En effet, si nous voulions l'écrire en Visual Basic, ce serait certainement la seule difficulté que nous pourrions rencontrer : dessiner cette tasse et à cet endroit. Pour le reste, créer un bouton et une étiquette en Visual Basic, est à la portée d'un débutant : chaque fois que nous cliquons sur le bouton, nous augmentons un compteur et présentons un message rafraîchi avec la nouvelle valeur.

L'approche visuelle de ces outils est essentielle, mais le code généré automatiquement n'apportera rien aux débutants qui désirent étudier et comprendre les bases du langage. Le seul avantage certain de ces programmes de développement est leurs outils de débogage et leur documentation intégrée.

Ici, pour pouvoir comprendre et étendre les exemples de ce chapitre, il faudra consulter la documentation des composants AWT qui est livrée avec le JDK de Sun Microsystems (voir annexe B) et disponible dans le format HTML.

Le code de notre première application AWT

Voici donc à présent le code en Java de l'application que nous venons de présenter, et que nous avons créée à la main avec un éditeur traditionnel. Les applications générées par des outils comme NetBeans (voir annexe E), qui nous permettent de placer nos composants

sur une feuille vide, ne généreront pas un code aussi simple et dénué de toutes fioritures. Ils utiliseront sans doute leurs propres classes, principalement dans le but de positionner correctement les différents composants dans la fenêtre. Dans le code qui suit :

```
import java.awt.*;
import java.awt.event.*;

public class JuniorAnonymFrame extends Frame {
  private Button bouton;
  private Label  etiquette;
  private int    compteur;

  public JuniorAnonymFrame() {
    super("Notre compteur");
    compteur = 0;
    setLayout(new FlowLayout());

    bouton = new Button("Compteur");
    bouton.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
         compteur++;
         String texte = new String("Le compteur est " + compteur);
         etiquette.setText(texte);
      }
    } );

    etiquette = new Label("Le compteur est 0 ");
    add(bouton);
    add(etiquette);
  }


  public static void main(String args[]) {
    JuniorAnonymFrame f = new JuniorAnonymFrame();

    f.setSize(200,100);
    f.setVisible(true);
    f.addWindowListener(new WindowAdapter() {
       public void windowClosing(WindowEvent e) {
          System.exit(0);
       }
    } );
  }
}
```

l'entrée `main()` de notre programme va nous créer une instance de notre classe `JuniorAnonymFrame` qui hérite de `Frame`, la classe essentielle pour nous offrir notre première application GUI. Les méthodes `setSize()` et `setVisible()` sont traditionnelles. Les valeurs de 200 et 100 ont été positionnées après quelques tentatives, qui dépendent de la présentation choisie (`setLayout(new FlowLayout())`) et des composants insérés avec la méthode `add()`. Il y a

donc deux éléments, un bouton et une étiquette. L'étiquette n'est jamais active, mais sera modifiée avec `setText()` chaque fois que le compteur est augmenté par un clic sur le bouton. Nous avons nommé ce dernier `Compteur`.

Il y a deux éléments actifs, le bouton et la fenêtre Windows. Cette dernière demande des explications. La réponse est évidente lorsque nous comprenons qu'il faut traiter la sortie de l'application Windows. Celle-ci peut être exécutée avec la petite croix en haut à droite du cadre de la fenêtre. Nous allons comprendre rapidement comment traiter ces événements.

Classes anonymes et classes internes

Les concepts de classes anonymes et de classes internes sont essentiels en programmation Java. Nous allons très rapidement en comprendre le mécanisme en analysant le code extrait de notre précédente application.

La méthode `addWindowListener()`, appliquée sur l'objet `f` de la classe `JuniorAnonymFrame`, hérite de `Frame`, où cette méthode est effectivement définie ; elle doit recevoir une instance d'une interface `WindowAdapter`. Cela peut sembler bien compliqué pour un programmeur « junior », auquel nous dirions plutôt : nous allons coller à la fenêtre un outil qui va traiter notre clic de souris, afin d'appeler la méthode statique `System.exit()` pour quitter le programme.

```
f.addWindowListener(new WindowAdapter() {
   public void windowClosing(WindowEvent e) {
      System.exit(0);
   }
} );
```

En fait, en regardant ce code, nous nous rendons compte que l'instance de l'interface `WindowAdapter`, dont nous devons produire le code, puisque c'est une interface, c'est-à-dire une définition et non une classe, se trouve directement entre les deux parenthèses de la méthode `addWindowListener()`. C'est ce que nous appelons une classe anonyme.

En ce qui concerne le bouton, nous suivons pour la méthode `addActionListener()` la même démarche :

```
bouton.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
     compteur++;
     String texte = new String("Le compteur est " + compteur);
     etiquette.setText(texte);
  }
} );
```

Nous remarquons que le code de la classe anonyme peut accéder aux attributs `compteur` et `etiquette`. À chaque clic, nous augmenterons donc le compteur et le présenterons comme nouveau texte de l'étiquette avec la méthode `setText()`.

L'étape suivante est la transformation de ces classes anonymes en classes internes, dont nous allons présenter tout aussi directement le code :

```
import java.awt.*;
import java.awt.event.*;

public class JuniorInternFrame extends Frame {
  private Button bouton;
  private Label  etiquette;
  private int    compteur;

  public JuniorInternFrame() {
    compteur = 0;
    setLayout(new FlowLayout());

    bouton = new Button("Compteur");
    bouton.addActionListener(new EvtCompte());

    etiquette = new Label("Le compteur est 0  ");

    add(bouton);
    add(etiquette);
    setBackground(Color.lightGray);
  }

  class EvtCompte implements ActionListener {
     public void actionPerformed(ActionEvent e)
{
        compteur++;
        String texte = new String("Le compteur est " + compteur);
        etiquette.setText(texte);
     }
  }

  static class EvtQuit extends WindowAdapter {
    public void windowClosing(WindowEvent e) {
      System.exit(0);
    }
  }

  public static void main(String args[]) {
    JuniorInternFrame f = new JuniorInternFrame();

    f.setSize(200,100);
    f.setVisible(true);
    f.addWindowListener(new EvtQuit());
  }
}
```

Nous voyons à présent l'apparition de ces fameuses classes internes qui font la joie des programmeurs Java. Si nous examinons les fichiers produits par la compilation de `JuniorInternFrame.java`, nous serons peut-être surpris :

```
JuniorInternFrame.class
JuniorInternFrame$EvtCompte.class
JuniorInternFrame$EvtQuit.class
```

Mais nous avions déjà ceci dans la version `JuniorAnonymFrame.java` avec des classes anonymes :

```
JuniorAnonymFrame.class
JuniorAnonymFrame$1.class
JuniorAnonymFrame$2.class
```

Après le `$` qui suit le nom de la classe, `JuniorInternFrame`, nous voyons apparaître le nom de la classe interne, c'est-à-dire `EvtCompte` et `EvtQuit`. Dans le cas de classes anonymes, sans nom, le compilateur Java va simplement allouer un numéro. Ici, nous avons 1 et 2 pour la classe `JuniorAnonymFrame`. Une analyse des fichiers `.class` nous donne ainsi une idée sur la structure interne des classes compilées.

S'adapter aux événements traditionnels de l'API

Dans le cas de la fenêtre, nous utilisons la méthode `addWindowListener()`, et pour le bouton, nous avons `addActionListener()`. Par événements traditionnels, nous entendons les actions simples, telles qu'un clic sur un bouton. Ces « écouteurs » sont nécessaires pour permettre à notre code d'obtenir le contrôle lorsqu'un événement se produit. Dans le cas de la fenêtre, si nous oublions de traiter l'événement qui veut fermer l'application, nous ne pourrons simplement jamais quitter le programme. Pour tous les composants que nous utilisons, nous devons consulter la documentation de l'API, qui va nous indiquer quelle méthode est disponible et quelle interface doit être utilisée. Dès ce moment-là, le compilateur nous force à implémenter le code nécessaire et à choisir l'opération à exécuter. Si nous voulions oublier `addWindowListener()` dans le cadre principal de l'application, il nous faudrait alors introduire un deuxième bouton pour quitter avec notre `System.exit()` ou bien décider de sortir, après une série de clics sur notre bouton unique. Il est essentiel de constater que si l'utilisateur ne bouge pas, il ne va absolument rien se passer ! Si nous voulions alors quitter le programme de toute manière, il faudrait introduire un *timer*, c'est-à-dire une horloge interne !

Il nous faut revenir sur la partie essentielle du code, qui est certainement plus lisible que la classe `JuniorAnonymFrame`, puisque le code du traitement des événements se fait en dehors du constructeur :

```
public JuniorInternFrame() {
  compteur = 0;
  setLayout(new FlowLayout());

  bouton = new Button("Compteur");
  bouton.addActionListener(new EvtCompte());

  etiquette = new Label("Le compteur est 0  ");

  add(bouton);
```

```
    add(etiquette);
    setBackground(Color.lightGray);
  }
```

La première méthode intéressante est la mise en place du gestionnaire de mise en forme, `setLayout()`. L'AWT possède de nombreux gestionnaires pour placer les éléments sur le cadre de l'application. Nous ne donnerons pas plus de détails, sinon qu'il est aussi possible d'inclure plusieurs panneaux sur le cadre principal, ceux-ci pouvant avoir chacun un gestionnaire différent. Les deux classes `Button` et `Label` reçoivent un texte comme paramètre de leurs constructeurs respectifs. Comme elles héritent toutes les deux de la classe `java.awt.Component`, elles peuvent être ajoutées au cadre principal avec la méthode `add()`. Cette dernière est définie dans la classe `java.awt.Container`, classe de base de `java.awt.Window`, elle-même classe de base de `java.awt.Frame`.

Le positionnement des éléments dans le cadre de la fenêtre dépendra du gestionnaire de mise en forme choisi, de la longueur des textes passés aux constructeurs, ainsi que de la dimension de la fenêtre choisie avec `setSize()` dans `main()` pour l'objet `f` de notre classe de `JuniorInternFrame`. Enfin, il est possible de changer la couleur de fond, le gris clair, avec `setBackground()` et la couleur `Color.lightGray`, qui est statique.

C'est sans doute le bon moment pour indiquer que `JuniorInternFrame` hérite de `Frame`, qui hérite de `Window`, qui hérite de `Container`, qui hérite finalement de `Component` et d'`Object`. Lorsqu'il s'agit de retrouver une méthode dans cette hiérarchie, une véritable forêt, une chatte n'y retrouverait plus ces petits ! Les méthodes `setSize()` et `setBackground()` sont définies dans la classe `Component`. Notre bouton est aussi dérivé de `Component` et l'instruction :

```
bouton.setBackground(Color.red);
```

aurait attribué un rouge « pétant » à l'arrière-plan de notre bouton, ce qui n'aurait pas été des plus présentables.

Et si on s'adaptait à d'autres types d'événements ?

Dans la version qui suit, nous allons effectivement nous adapter à d'autres événements ! Il est possible d'ajouter une écoute à un autre type d'événement, afin de recevoir, par exemple, une notification si nous nous déplaçons sur un composant. Nous pourrions ajouter d'autres actions, par exemple au moment où nous pressons ou lâchons le bouton de la souris. Dans l'exemple qui suit, nous allons contrôler le moment où nous entrons et sortons de la zone du bouton `Compteur` :

```
import java.awt.*;
import java.awt.event.*;

public class JuniorFrame extends Frame {
  private Button bouton;
  private Label  etiquette;
  private int    compteur;

  public JuniorFrame() {
```

```
        compteur = 0;
        setLayout(new BorderLayout());

        bouton = new Button("Compteur");
        bouton.addActionListener(new EvtCompte());
        bouton.addMouseListener(new EvtMouse());

        etiquette = new Label("Le compteur est 0");
        etiquette.setAlignment(Label.CENTER);

        add(bouton, BorderLayout.CENTER);
        add(etiquette, BorderLayout.SOUTH);
        setBackground(Color.lightGray);
    }

    private void nouveauTexte(String texte) {
        etiquette.setText(texte);
    }

    class EvtCompte implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            compteur++;
            nouveauTexte("Le compteur est " + compteur);
        }
    }

    class EvtMouse implements MouseListener {
        public void mouseEntered(MouseEvent e) {
            nouveauTexte("Clique sur le bouton pour augmenter le compteur");
        }

        public void mouseExited(MouseEvent e) {
            nouveauTexte("Le compteur est " + compteur + " ");
        }

        public void mouseClicked(MouseEvent e) { }
        public void mousePressed(MouseEvent e) { }
        public void mouseReleased(MouseEvent e) { }
    }

    public static void main(String args[]) {
      JuniorFrame f = new JuniorFrame();

      f.setSize(300, 100);
      f.setVisible(true);
      f.addWindowListener(new WindowAdapter() {
         public void windowClosing(WindowEvent e) {
            System.exit(0);
         }
      } );
    }
}
```

Notre nouvelle application va apparaître comme sur la figure 18-2.

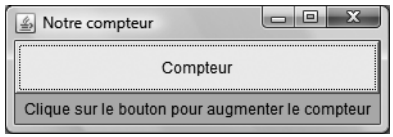

Figure 18-2
L'application compteur

Deux ou trois modifications sont apparues dans notre application. Le titre d'abord, `Notre compteur`, qui vient de l'appel de la méthode `super()` dans le constructeur de notre nouvelle classe `JuniorFrame`. Le texte `Clique sur le bouton pour augmenter le compteur` va apparaître et disparaître lorsque nous entrons dans la zone, trop grande d'ailleurs, de notre bouton `Compteur`. Ceci peut se faire grâce à l'ajout d'un autre gestionnaire d'événements, `addMouseListener()`, grâce auquel nous allons pouvoir recevoir une notification à l'entrée et à la sortie de cette zone. Nous avons alors ajouté une nouvelle classe interne, `EvtMouse`, qui va définir les cinq implémentations de méthodes requises pour l'interface `MouseListener`. Seules deux sont intéressantes pour nos besoins, `mouseEntered()` et `mouseExited()`, dans lesquelles nous avons utilisé une nouvelle méthode publique `nouveauTexte()` qui va nous accomplir notre code répétitif, consistant à changer le texte dans l'étiquette.

Nous avons choisi un autre gestionnaire de mise en forme, le `BorderLayout`, qui nous donne un bouton beaucoup trop large. Nous ne reviendrons pas trop sur les détails de :

```
add(bouton, BorderLayout.CENTER);
add(etiquette, BorderLayout.SOUTH);
```

qui nous positionne le bouton au centre et au sud du cadre ainsi que de :

```
etiquette.setAlignment(Label.CENTER);
```

qui nous centre l'étiquette.

Applets ou applications

Le code précédent nous a permis de nous familiariser avec l'écriture d'une application GUI, l'utilisation de l'AWT et du traitement des événements. Mais qu'en est-il des applets ?

Une `applet` est un programme Java qui sera exécuté à l'intérieur d'un document HTML, c'est-à-dire à partir de Netscape ou d'Internet Explorer. Nous allons mieux le comprendre en examinant ce document HTML, qui contient une référence à une `applet` pour une classe nommée `JuniorApplet`, que nous allons examiner plus loin. Voici donc le code source du fichier :

```
<HTML>
<TITLE>Mon compteur</TITRE>
<BODY>
```

```
<h1>Mon applet Junior</h1>

<hr>
<applet code=JuniorApplet.class width=450 height=100></applet>
<hr>

</BODY>
</HTML>
```

La balise `<applet>` indique que le navigateur devra charger le code Java compilé à partir de sa machine virtuelle intégrée. Il y a aussi les deux paramètres `width` et `height` qui correspondent à la largeur et à la hauteur de l'applet dans le document HTML. Il ne faut pas oublier que d'autres balises HTML peuvent être présentes et que les valeurs que nous avons choisies, `450` et `100`, nécessitent quelquefois des ajustements. Il ne faut pas oublier non plus qu'une fenêtre du navigateur peut être redimensionnée et peut parfois donner quelques surprises, surtout si celle-ci devient trop petite.

À présent, nous allons examiner une déclinaison de notre classe `JuniorFrame`, que nous allons transformer en applet. La caractéristique intéressante de cette classe `JuniorApplet` est qu'elle est construite de telle manière qu'elle peut être à la fois utilisée comme applet et comme application :

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class JuniorApplet extends Applet {
  private Button bouton;
  private Label  etiquette;
  private int    compteur;

  public void init() {
     compteur = 0;
     setLayout(new FlowLayout());
     setFont(new Font("Courier New", Font.BOLD, 18));

     bouton = new Button("Compteur");
     bouton.addActionListener(new EvtCompte());
     bouton.addMouseListener(new EvtMouse());

     etiquette =
            new Label("Clique sur le bouton pour augmenter le compteur");
     etiquette.setAlignment(Label.CENTER);

     add(bouton);
     add(etiquette);
     setBackground(Color.lightGray);
  }
```

```
  private void nouveauTexte(String texte) {
     etiquette.setText(texte);
  }

  class EvtCompte implements ActionListener {
     public void actionPerformed(ActionEvent e) {
        compteur++;
        nouveauTexte("Le compteur est " + compteur);
     }
  }

  class EvtMouse implements MouseListener {
     public void mouseEntered(MouseEvent e) {
        nouveauTexte("Clique sur le bouton pour augmenter le compteur");
     }

     public void mouseExited(MouseEvent e) {
        nouveauTexte("Le compteur est " + compteur + " ");
     }

     public void mouseClicked(MouseEvent e) { }
     public void mousePressed(MouseEvent e) { }
     public void mouseReleased(MouseEvent e) { }
  }

  public static void main(String args[]) {
    JuniorApplet applet = new JuniorApplet();

    Frame f = new Frame("JuniorApplet");
    f.addWindowListener(new WindowAdapter() {
       public void windowClosing(WindowEvent e) {
          System.exit(0);
       }
    } );

    f.setSize(500, 120);
    f.add(applet);
    applet.init();
    applet.start();
    f.setVisible(true);
  }
}
```

Cette nouvelle classe, `JuniorApplet`, hérite de la classe `Applet` définie dans :

```
import java.applet.*;
```

et n'a pas les mêmes caractéristiques ni constructeurs que la classe `Frame`. Par exemple, il n'est pas possible d'initialiser des paramètres dans la classe de base avec `super()`.

Avant de passer aux autres détails, il nous faut présenter `appletviewer`, cet outil distribué avec le JDK de Sun Microsystems. Si nous exécutons la commande dans le répertoire `C:\JavaCpp\EXEMPLES\Chap18` et dans une fenêtre DOS :

```
appletviewer junior.html
```

nous obtiendrons ceci :

Figure 18-3
L'applet compteur

`appletviewer` est un outil très pratique pour tester nos applets en lieu et place d'un navigateur Web traditionnel. Au contraire de Netscape ou d'Internet Explorer, il ne montre pas le code HTML, comme ici le titre `H1`. Il possède cependant un certain nombre d'outils dans son menu, pour, par exemple, arrêter ou relancer l'applet.

C'est aussi le bon moment pour faire remarquer que notre applet est située sur notre machine, c'est-à-dire sur notre disque. Dans la réalité, c'est différent. Les pages HTML ainsi que les applets viennent de l'extérieur, du site Web où est enregistré tout le matériel nécessaire, dont font partie par exemple les images ou les fonds d'écran. Comme l'applet vient de l'extérieur et sera exécutée sur le PC hôte, nous pourrions nous imaginer qu'elle pourrait écrire des fichiers ou effacer volontairement des ressources qui ne lui appartiennent pas.

Nous pouvons faire l'exercice suivant après avoir ajouté :

```
import java.io.*;
```

en début de fichier. Ensuite, nous introduisons le code suivant :

```
try {
   PrintWriter out = new PrintWriter(new FileWriter("test.txt"));
   out.println("Salut");
   out.close();
}
catch(IOException ioe) {
   return;
}
```

après la ligne :

```
nouveauTexte("Le compteur est " + compteur);
```

Cela veut dire que chaque fois que nous activons le bouton, le fichier `test.txt` sera écrit sur le disque. À nouveau depuis le répertoire de travail `C:\JavaCpp\EXEMPLES\Chap18` et dans une fenêtre DOS , nous pouvons le vérifier avec :

```
java JuniorApplet
```

qui nous générera effectivement un fichier.

Cependant, si nous exécutons :

```
appletviewer junior.html
```

et si nous actionnons le bouton, nous aurons une surprise :

```
Exception occurred during event dispatching:
java.security.AccessControlException: access denied (java.io.FilePermission
➥fichier.txt write)
        at java.security.AccessControlContext.checkPermission(Comp..)
        at java.security.AccessController.checkPermission(Compiled Code)
        at java.lang.SecurityManager.checkPermission(Compiled Code)
        at java.lang.SecurityManager.checkWrite(Compiled Code)
        at java.io.FileOutputStream.<init>(Compiled Code)
        at java.io.FileWriter.<init>(Compiled Code)
        at JuniorApplet$EvtCompte.actionPerformed(Compiled Code)
        at java.awt.Button.processActionEvent(Compiled Code)
        …
```

Il y a donc un mécanisme de protection. Sans ce dernier, les applets n'auraient jamais pu survivre dans la tempête du développement technologique de ces dernières années. En d'autres mots, si un programmeur veut utiliser Java pour construire des outils système, il ne pourra pas utiliser d'applet.

init() et start() pour une applet

Il faut évidemment revenir sur ce code essentiel. Dans notre première classe `JuniorFrame`, nous avions placé tout le code d'initialisation dans le constructeur. Pour `JuniorApplet`, c'est différent, et ce code fait partie de la méthode `init()`. Cette manière de faire est nécessaire, car la méthode `init()` sera appelée la première fois, et seulement une fois, lorsque l'applet sera démarrée. La classe `Applet` possède une méthode `start()` qui peut être redéfinie en cas de nécessité. Chaque fois que le navigateur revient à une page contenant une applet, la méthode `start()` est activée, ce qui n'est pas le cas de la méthode `init()`. En passant, nous noterons qu'il existe aussi une méthode `stop()` qui pourrait être utilisée pour arrêter une activité, par exemple une animation.

Le reste du code de la classe `JuniorApplet`, pour le traitement de la souris et des événements, est similaire à notre classe `JuniorFrame` précédente.

Enfin, il nous faut revenir au code de la partie `main()`. C'est là que nous trouvons la fameuse astuce qui consiste à inclure un objet de la classe `Applet` dans un `Frame`. L'instruction :

```
f.add(applet);
```

va déposer l'applet à l'intérieur de l'instance `f` de `Frame`, qui est capable de fonctionner comme application. Les deux instructions suivantes :

```
applet.init();
applet.start():
```

correspondent à la discussion précédente pour démarrer l'applet. La classe `JuniorApplet` pourra donc fonctionner à la fois comme applet et comme application.

Un mot sur les servlets

Les servlets fonctionnent plus ou moins comme des applets, mais sont des applications Java qui tournent sur un serveur HTTP supportant cette technologie. Les servlets sont équivalentes aux scripts CGI, très souvent écrites en Perl, en PHP4, mais aussi en C et en C++. Elles sont capables, par exemple, de traiter des feuilles de requêtes (`FORM`) en HTML et de générer du code HTML dynamique.

Pour se familiariser avec cette technologie, on peut recommander de télécharger le kit de Sun Microsystems :

```
http://java.sun.com/products/jsp/
http://java.sun.com/products/servlet/
```

qui permettra de développer de petites applications servlets et de les exécuter sur son PC sans la nécessité de se connecter à Internet.

Dans l'annexe E, nous présenterons NetBeans avec lequel nous pourrions développer des servlets et les tester avec un serveur Web Apache Tomcat intégré.

get(), set() et les JavaBeans

En consultant par exemple la documentation de `java.awt.Component`, nous constaterons les nombreuses méthodes qui sont marquées *deprecated*, c'est-à-dire qui ne devraient plus être utilisées. Nous avons entre autres la méthode `size()`, qui nous retourne la dimension du composant AWT :

```
public Dimension size();
```

Eh bien, celle-ci devrait être remplacée par la méthode `getSize()`, qui retournera une `Dimension`.

```
public Dimension getSize();
```

En fait, il doit exister une variable privée nommée `size`, à laquelle nous mettons une majuscule pour la première lettre et ajoutons un `get()` ou un `set(…)` ; c'est la manière Beans.

```
public void setSize(int width, int height)
```

C'est un peu l'histoire des Beans… rien à voir avec les petites histoires drôles ou les manières de Mr Bean !

Qu'est-ce qu'un Bean ?

Une des origines des Beans se trouve certainement dans une inspiration venue des composants de deuxième génération, qui sont apparus dans des outils de programmation comme Delphi de Borland. L'idée est de promouvoir une technique qui permet de créer un bloc de code avec un certain standard, afin qu'un outil puisse découvrir par lui-même les propriétés et les événements pour ce composant. Nous n'irons pas plus loin sur le sujet, qui dépasse encore une fois les buts fixés pour cet ouvrage. Nous mentionnerons aussi l'existence de la classe `Beans`, qui fait partie du paquet `java.beans`, où nous trouvons par exemple une interface essentielle comme `BeanInfo`.

Beans et C++

Il n'y a pas de Beans en C++, mais rien ne nous empêcherait de définir le même standard pour accéder aux variables privées d'une classe, comme dans cet exemple :

```
class UnBean
{
  public:
    int getAttribut() {
      return attribut;
    }

    void setAttribut(int valeur) {
      attribut = valeur;
    }

  private:
    int attribut;
};
```

Résumé

Cet aperçu de l'AWT, les composants graphiques de Java, était nécessaire. Il nous a permis de nous familiariser avec les classes internes et anonymes en Java. La diversion sur les jeunes programmeurs était aussi une manière de présenter l'AWT comme un départ possible en programmation. Nous savons à présent différencier une applet d'une application et même les combiner.

Exercice

Une application AWT un peu plus évoluée.

> **Note**
> Le lecteur ne possédera pas toutes les bases pour faire ce travail. Il devra consulter la documentation de l'AWT et de ses nombreuses classes, interfaces et méthodes de composants.

Reprendre la classe `JuniorApplet` et lui apporter les modifications suivantes :

– Ajouter un bouton `Mise à zéro` pour remettre le compteur à 0.

– Ajouter un bouton `Quitte`, mais seulement pour une application (`Frame`). Ce bouton n'apparaîtra pas pour une applet.

– Les deux ou trois boutons seront inclus dans un panneau (`Panel`) avec une présentation `FlowLayout`. Le panneau et l'étiquette utiliseront une présentation `BorderLayout` à l'intérieur de l'applet, et ceci avec une orientation nord-sud. Ces deux dernières sont des paramètres de la méthode `add()`.

– Nous définirons trois `ActionListeners` pour nos boutons, mais le `MouseListeners` sera commun. Pour identifier le bouton concerné, il faudra utiliser la méthode `getSource()` sur l'événement. Il sera alors nécessaire de définir les trois boutons comme des attributs de la classe et non comme des variables de la méthode `init()`.

– Nous nous amuserons avec différents types de caractères et polices, comme le gras ou l'italique. La méthode `setFont()` sur chaque composant devra être utilisée.

19

Un livre sur C++ sans templates !

Les modèles ou patrons en C++

Ce chapitre est spécifique au C++. Il serait inconvenant de laisser de côté un aspect aussi essentiel que les modèles en C++. Dans une certaine mesure, ils peuvent être comparés aux types génériques de Java, apparus à partir du JDK 1.5, et que nous avons déjà traités au chapitre 14.

Le terme de modèle va de pair avec le concept de programmation générique. Il est en effet possible de programmer ces classes et ces algorithmes pour plusieurs variétés de types ou d'objets. Cette technique permet avant tout d'éliminer des centaines, voire des milliers de lignes de code qui vont pouvoir s'appliquer sur différents types d'objets. Maîtriser et comprendre ce concept et ce style d'écriture est sans doute un bagage supplémentaire intéressant pour les programmeurs de classes et d'interfaces en Java.

Nous ne pourrions parler des modèles sans mentionner la bibliothèque STL (*Standard Template Library*). Apparue ces dernières années, elle sera toujours plus utilisée. Un programmeur C++ expérimenté qui débute en Java pourra avoir quelques difficultés lorsqu'il devra, par exemple, convertir en Java du code contenant des classes de la bibliothèque STL, ainsi que des algorithmes génériques, afin d'y retrouver rapidement l'équivalence et la même efficacité.

Nous ne reviendrons pas sur les nombreux algorithmes du Standard C++. Nous en avons déjà utilisé quelques-uns au chapitre 14 avec les collections. Algorithmes et modèles sont en fait indissociables.

Un modèle de fonction

Les programmeurs débutants sont souvent effrayés par la syntaxe des modèles. Il est parfois nécessaire de s'y reprendre à plusieurs fois avant de compiler avec succès son premier programme. La syntaxe étant relativement lourde, le programmeur aura souvent la tentation de bâcler son code, sans vraiment faire l'effort d'en comprendre la forme. Après quelques tentatives et exercices, il se rendra finalement compte, certainement avec étonnement, que ce n'était pas la mer à boire.

Comme premier exemple, nous allons définir une fonction générique qui nous retourne soit le résultat de la somme de deux objets s'ils sont différents, soit, s'ils sont identiques, un seul de ces deux objets. Si nous pensons à des nombres, tels que 10 et 11 ou 20 et 20, nous n'en voyons pas vraiment l'utilité ! En revanche, dans une suite de mots tels que `la maison` ou `le le`, nous en verrions plus facilement l'application dans un traitement de texte dans lequel on pourrait demander confirmation avant d'effacer des mots répétés.

Avant de passer à l'écriture de la fonction générique, qui devrait fonctionner sur toutes sortes d'objets, nous devons nous poser un certain nombre de questions. Cette méthode va utiliser les opérateurs `==` et `+`. Il faut donc que ces opérateurs existent, ceci pour les objets dont nous aimerions créer une instance pour cette fonction générique. Nous allons voir à présent ces détails, ainsi que la syntaxe des modèles dans notre premier exemple :

```
// unefonction.cpp
#include <iostream>

using namespace std;

template<class T> T somme(const T& obj1, const T& obj2) {
  if (obj1 == obj2)  return obj1;
  return obj1 + obj2;
}

int main()
{
  cout << somme<int>(10, 11) << endl;
  cout << somme<int>(20, 20) << endl;
  cout << somme<double>(2.2, 7.8) << endl;
  cout << somme<string>("Hello", "Bonjour") << endl;
  return 0;
}
```

Le préfixe `template<class T>` indique le début d'une déclaration pour un modèle de type `T`. Ce dernier peut être de type primitif comme `char`, `int` ou `double` ou encore le nom d'une classe standard, comme `string` ou autres classes propriétaires. Le reste de la définition est pareil à la déclaration d'une méthode normale. Si nous voulions avoir une fonction `somme()` uniquement valable pour des `string`, il suffirait d'effacer le `template<class T>` et de remplacer tous les `T` par des `string` et nous obtiendrions ceci :

```
string somme(const string& obj1, const string& obj2) {
  if (obj1 == obj2)  return obj1;
  return obj1 + obj2;
}
```

Nous voyons donc que pour pouvoir construire une fonction générique il nous faut absolument définir tous les opérateurs et les méthodes utilisés à l'intérieur du modèle. La fonction `somme()` peut être utilisée, à condition que les opérateurs + et == soient implémentés. Si nous sommes attentifs et définissons correctement nos propres classes, nous pouvons donc aussi les rendre utilisables par ces fonctions génériques. Nous allons à présent créer une nouvelle classe, `maClasse`, pour exécuter notre fonction `somme()` sur deux instances de cette même classe :

```
// uneclasse.cpp
#include <iostream>

using namespace std;

class UneClasse {
   private:
     int compteur;

   public:
      UneClasse(int valeur = 0):compteur(valeur) {};

      UneClasse(const UneClasse &objet1):compteur(objet1.compteur) {};

      UneClasse& operator=(const UneClasse& objet1) {
         compteur = objet1.compteur;
      }

      friend int operator== (const UneClasse &objet1, const UneClasse &objet2) {
         if (objet1.compteur == objet2.compteur)  return 1;
         return 0;
      }

      UneClasse operator+ (const UneClasse &original) const {
         int nouveau = compteur + original.compteur;

         UneClasse nouvelle(nouveau);
         return nouvelle;
      }

      friend ostream& operator<<(ostream& stream, const UneClasse &original) {
         return stream << original.compteur;
      }
};

template<class T> T somme(const T& obj1, const T& obj2) {
  if (obj1 == obj2)  return obj1;
```

```
  return obj1 + obj2;
}


int main() {
  UneClasse mca(1);
  UneClasse mcb(2);

  cout << somme<UneClasse>(mca, mcb) << endl;
  cout << somme<UneClasse>(mca, mca) << endl;
  cout << somme<UneClasse>(mcb, mcb) << endl;
}
```

et son résultat :

```
3
1
2
```

L'implémentation de la classe `UneClasse` est presque complète. Nous y avons ajouté un constructeur par défaut, un constructeur de copie, ainsi que l'opérateur d'affectation. L'opérateur `==` considère l'égalité des deux objets si l'attribut compteur est identique. Nous aurions même pu vérifier si les deux objets étaient identiques, ce qui est le cas de nos deuxième et troisième exemples, mais nous constatons que ce n'est pas nécessaire. Il est évident qu'aussi bien la définition de la classe que celle du modèle devraient être dans un ou deux fichiers d'en-tête.

Un modèle de classe

Nous allons à présent passer à un modèle de classe et constater qu'il n'y a rien de nouveau et que la syntaxe est la même. Voici donc notre classe `Point` :

```
// unmodele.cpp
#include <iostream>

template <class T> class Point
{
   T posx;
   T posy;

public:
   Point(T x, T y) {
     posx = x;
     posy = y;
   }

   void test();
};

template <class T>
```

```
void Point<T>::test() {
  std::cout << "Position: " << posx << ":" << posy << std::endl;
};


using namespace std;

int main()
{
  Point<int> pint(1,2);
  pint.test();

  Point<double> pdouble(1.1,2.22);
  pdouble.test();
  return 0;
}
```

qui pourrait représenter aussi bien un pixel dans une image qu'une position en centimètres avec plusieurs décimales dans un espace à deux dimensions. Comme pour le modèle de fonction, nous retrouvons notre `template <class T>` et nous avons déjà l'impression d'être familiers avec les modèles, dont tout programmeur C++ avancé devrait absolument considérer l'utilisation. Le résultat du programme :

```
Position: 1:2
Position: 1.1:2.22
```

nous confirme que le code se compile et fonctionne comme prévu.

Nous retrouvons les formes `std::cout` et `std::endl` du chapitre 6, puisque le code, avant le `using namespace std;`, peut être considéré comme un fichier d'en-tête.

Résumé

Les modèles (*templates*) sont l'un des aspects essentiels de la programmation C++. Ils sont très présents dans l'implémentation de la bibliothèque Standard (STL, *Standard Template Library*) et doivent être absolument considérés durant l'apprentissage du langage C++.

Exercice

Créer un modèle de fonction nommé `soustraction()` qui va soustraire deux entiers (`int`) ou deux objets de notre classe `UneClasse`, que nous allons modifier pour supporter l'opérateur `-`. La nouvelle classe `UneClasse1` sera aussi corrigée, afin que son attribut `compteur` ne soit jamais négatif. Dans ce cas-là, sa valeur sera 0, comme d'ailleurs le résultat de la soustraction si elle se révèle négative.

20

Impossible sans SQL !

Pour cette partie, nous supposerons que le lecteur possède une licence d'Access de Microsoft et quelques notions de cet outil de base de données. Dans la première partie, nous allons revenir sur notre fichier délimité, celui que nous avons déjà rencontré au chapitre 9. Nous allons créer, en C++, des données que nous importerons dans Access. Nous n'utiliserons que Java pour accéder à notre base de données, avec des instructions SQL, car nous n'avons pas de bibliothèque en C++. Ces dernières existent, mais elles sont en général livrées avec des produits commerciaux comme Oracle, IBM DB3, Microsoft SQL Server ou Sybase. Une alternative serait d'utiliser Linux et MySQL, par exemple. Nous y reviendrons en fin de chapitre.

Nous en profiterons pour présenter le langage SQL et la technologie ODBC de Microsoft.

Création d'un fichier délimité en C++

Le programme qui suit va nous créer un fichier délimité `Personnes.txt`. La première ligne du fichier va contenir la définition des champs de notre base de données, c'est-à-dire le nom, le prénom et l'année de naissance. Les enregistrements viendront ensuite sur chaque ligne. Le dernier champ, l'année de naissance, n'est pas mis entre guillemets. Nous verrons comment Microsoft Access convertira ce champ, comme un entier, dans la base de données. Nous utiliserons ici les termes de champs ou de colonnes, qui sont en fait équivalents.

```
// write_db.cpp
#include <iostream>
#include <sstream>
#include <fstream>
#include <string>
```

```
using namespace std;

class Personne {
  private:
    string nom;     // nom de la personne
    string prenom;  // prénom de la personne
    int    annee;   // année de naissance de la personne

  public:
    Personne::Personne(string leNom, string lePrenom, int lAnnee)
             :nom(leNom), prenom(lePrenom), annee(lAnnee) {}

    // char *get_delim() const {
    string get_delim() const {
      ostringstream enregistrement;
      enregistrement << "\"" << nom << "\";\"" << prenom << "\";" << annee << ends;

      return enregistrement.str();
    }
};

class WritePersonDB {
  private:
    ofstream outfile;

  public:
    bool write_debut(const char *fichier) {
      outfile.open(fichier, ios::out);

      if (!outfile.is_open()) {
        return false;
      }

      outfile << "\"Nom\";\"Prenom\";\"Annee\"";
      return true;
    }

    void write_record(const Personne &personne) {
      outfile << endl;
      outfile << personne.get_delim();
    }

    void write_fin() {
      outfile.close();
    }
};

int main() {
  WritePersonDB wpersonne_db;

  Personne personnes[2] = {
```

```
    Personne("Haddock", "Capitaine", 1907),
    Personne("Kaddock", "Kaptain",   1897)
  };

  if (!wpersonne_db.write_debut("personnes.txt")) {
    cout << "Ne peut pas écrire dans le fichier personnes.txt" << endl;
    return -1;
  }

  wpersonne_db.write_record(personnes[0]);
  wpersonne_db.write_record(personnes[1]);

  wpersonne_db.write_fin();

  cout << "Fichier personnes.txt généré" << endl;
}
```

C'est un très joli programme, dans lequel nous utilisons toute une série de techniques présentées dans cet ouvrage.

Il y a d'abord le `ostringstream`, que nous avons analysé en détail au chapitre 9 et qui nous permet d'assembler des `string`. Nous pourrions utiliser l'opérateur `+`, mais nous aurions une difficulté avec `annee`, qui est un entier.

Si nous ôtions le `const` du `get_delim()`, le programme ne compilerait pas. Cet aspect a été considéré au chapitre 6, lors du traitement du suffixe `const` pour les méthodes.

Le reste ne présente pas de difficultés particulières, à condition de maîtriser les entrées et sorties que nous avons étudiées au chapitre 9. En effet, l'écriture d'un fichier XML est presque identique. Nous avons créé deux classes pour faire ce travail, mais il y a certainement d'autres approches tout aussi élégantes et structurées. `WritePersonDB` pourrait certainement hériter d'une classe de base indépendante du format de la table.

Il faut cependant retenir un détail. Nous n'écrivons jamais de nouvelle ligne `endl` à la fin de l'enregistrement, mais seulement au début. C'est important, car le dernier enregistrement contiendrait une ligne vide et donnerait des difficultés à l'importation dans Microsoft Access.

Création d'une base de données sous Microsoft Access

Avant de pouvoir enregistrer des données dans Microsoft Access ou y accéder, il nous faut d'abord créer une base de données. Nous lancerons donc Microsoft Access 2000 pour créer un fichier `bd1.mdb` dans le répertoire `Chap20`, dans lequel nous avons notre code C++ et Java correspondant. Nous n'allons pas créer de table, mais notre fichier `Personnes.txt` :

```
"Nom";"Prenom";"Annee"
"Haddock";"Capitaine";1907
"Kaddock";"Kaptain";1897
```

qui vient d'être généré par notre programme C++, sera importé dans une table `Personnes` de Microsoft Access. Pour ce faire, il suffit de lancer Access, de sélectionner Fichier > Données externes > Importer et de spécifier le fichier texte `Personnes.txt`. L'assistant d'importation demande un certain nombre de paramètres qui n'ont rien de mystérieux. Il ne faudra pas oublier d'indiquer l'option Première ligne contient le nom des champs, et nous laisserons Microsoft Access ajouter une clé primaire. La table se nommera aussi `Personnes`. Si nous examinons le contenu de la table ou sa définition, nous sommes absolument satisfaits :

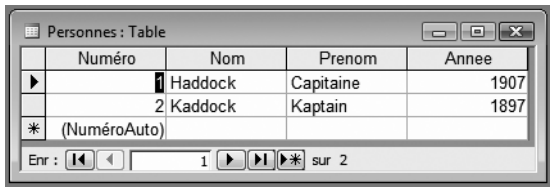

Figure 20-1
Contenu de la table Personnes

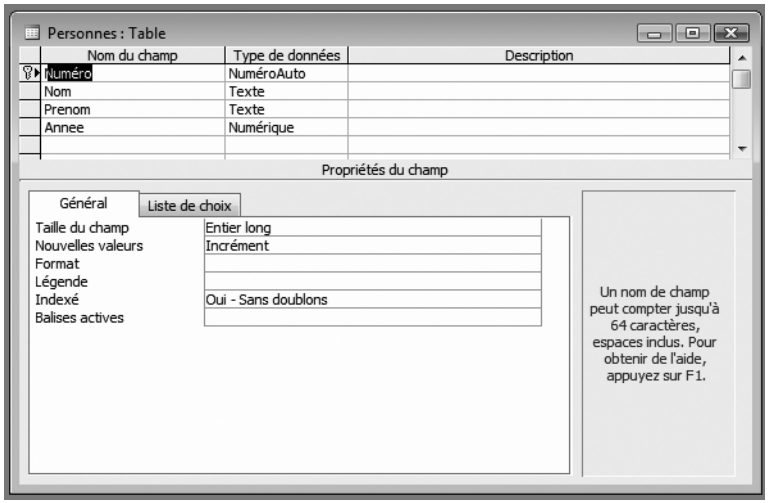

Figure 20-2
Définition de la table Personnes

La première ligne de notre fichier nous a bien défini les champs de notre base de données, et la colonne `Annee` a bien été considérée comme numérique. Nous pouvons à présent fermer notre base de données ainsi qu'Access.

Activation d'ODBC - XP

L'étape suivante est l'activation d'ODBC (*Open DataBase Connectivity*). ODBC va nous permettre de communiquer entre Java et notre base de données Access disponible dans le fichier `bd1.mdb`.

Sous Windows XP, nous devons installer l'accès ODBC à notre fichier Access en allant dans le menu Démarrer puis, Panneau de configuration>Outils d'administration>Sources de données ODBC>Sources de données utilisateur.

En assumant que l'utilisateur n'a jamais utilisé l'ODBC, nous remplacerons MS Access Database par ODBC-Java-test :

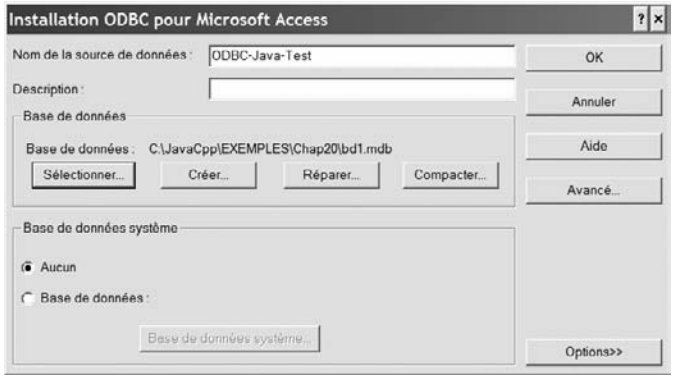

Figure 20-3

Windows XP – Installation ODBC pour bd1.mdb

Avec de cliquer sur le bouton Sélectionner, nous devrons naviguer sur notre disque pour trouver et définir notre référence à `C:\JavaCpp\EXEMPLES\Chap20\bd1.mdb`.

Après avoir cliqué sur le bouton OK, nous devrions retrouver notre ODBC-Java-test dans la liste qui peut varier d'un système à l'autre en fonction des outils installés.

Figure 20-4

Windows XP – Access driver

Depuis l'onglet Sources de données utilisateur, nous pourrions aussi cliquer sur le bouton Ajouter pour obtenir la fenêtre de la figure 20-5, spécifier le driver Access et ajouter notre ODBC-Java-test. Cette séquence est nécessaire dans le cas de plusieurs entrées ODBC.

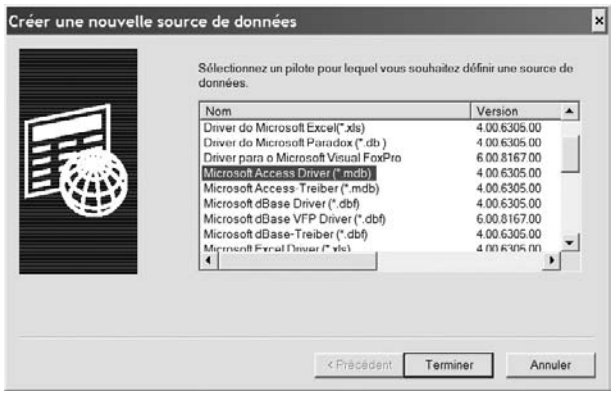

Figure 20-5

Windows XP – Nouvelle entrée Access

Activation d'ODBC – Vista

Sous Windows Vista, l'installation est presque équivalente ; il faut juste ne pas se perdre dans ce système d'exploitation remodelé ! L'accès se fait depuis le Panneau de configuration (Affichage classique)>Outils d'administration>Source de données ODBC.

Notre base de données `bd1.mdb` est une base de données Access.

Figure 20-6

Windows Vista – Nouvelle entrée ODBC pour Access

Le driver est encore Microsoft Access.

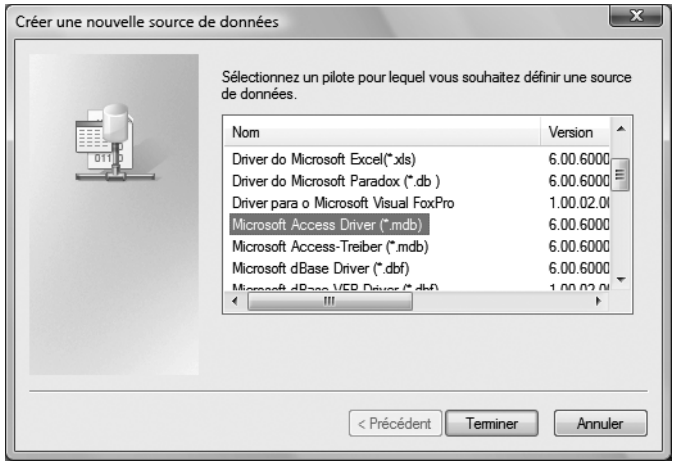

Figure 20-7
Windows Vista – Driver Access

Nous devrons choisir notre source de base de données et son nom d'accès.

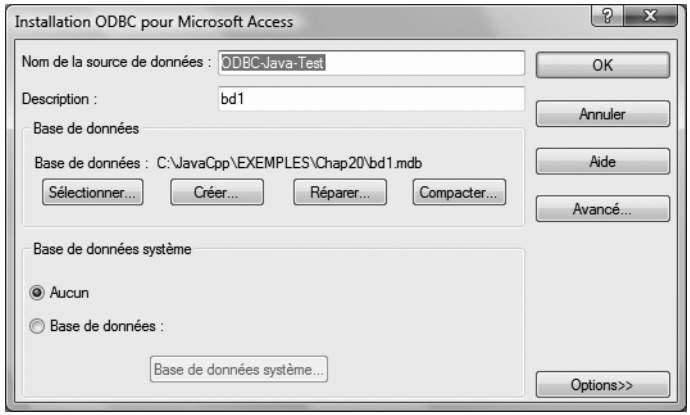

Figure 20-8
Windows Vista – Source de données ODBC-Java-Test

Dès cet instant, le driver ODBC sera actif sur cette base de données `bd1.mdb`.

Cette entrée restera active, même au prochain démarrage de Windows XP ou Vista. Pour effacer cette ressource du système, il faudra supprimer l'entrée en utilisant le bouton correspondant. Le nom de la source utilisée est essentiel : ODBC-Java-Test. C'est grâce à lui que l'interface Java JDBC décrite ci-après fera le lien avec `jdbc:odbc:ODBC-Java-Test`.

Pour les personnes qui utilisent d'anciennes versions de Windows ou d'Access, ou encore des versions anglaises, les termes sont parfois différents. Pour les onglets, DSN est parfois remplacé par Sources de données et Connexion multiple par Groupement de connexions. Il faut alors essayer quelques combinaisons, comme celle d'utiliser les Sources de données utilisateur au lieu des Sources de données système.

Accès ODBC à partir de Java

Un nouveau venu en programmation Java sera évidemment surpris. Nous connaissons déjà les qualités de ce langage dans le domaine d'Internet, mais il est tout aussi puissant en programmation pour l'accès aux bases de données. La première instruction du programme qui va suivre :

```
import java.sql.*;
```

devrait nous conduire à consulter la documentation du JDK de Sun Microsystems. Nous y découvrirons le terme de JDBC, *Java Database Connectivity,* qui correspond au logiciel qui va nous permettre d'accéder à notre base de données `bd1.mdb` au travers d'ODBC. Nous allons présenter le code et passer ensuite aux détails :

```
// ODBC 32 bits doit être actif
// ODBC-Java-Test associé à Chap20\bd1.mdb

import java.sql.*;

public class PersonnesODBC {
 public static void main(String args[]) {
   String url = "jdbc:odbc:ODBC-Java-Test";

   Connection con;
   Statement stmt;
   String requete = "select Nom, Prenom, Annee FROM Personnes";

   try {
     Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
   }
   catch(java.lang.ClassNotFoundException e) {
     System.err.print("ClassNotFoundException: ");
     System.err.println(e.getMessage());
     return;
   }

   try {
     System.out.println("Connexion avec ODBC-Java-Test");
     con = DriverManager.getConnection(url, "", "");
     stmt = con.createStatement();

     System.out.println("Exécute la requête");
```

```
      ResultSet rs = stmt.executeQuery(requete);

      System.out.println("Les entrées dans notre base de données: ");
      while (rs.next()) {
        String nom = rs.getString("Nom");
        String prenom = rs.getString("Prenom");
        int annee = rs.getInt("Annee");

        System.out.println(nom + " " + prenom + " est né en " + annee);
      }

      stmt.close();
      con.close();

    }
    catch(SQLException ex) {
      System.err.println("SQLException: " + ex.getMessage());
    }
  }
}
```

Si nous faisons une analyse détaillée de ce code, une des premières surprises viendra de nos deux instances de `Connection` et de `Statement`. Ce sont des interfaces, et cela signifie que nous devons implémenter le code. Mais comment donc cela est-il possible pour ce grand nombre de méthodes définies dans ces deux interfaces ? Pour s'en rendre compte, il suffit de consulter la documentation de l'API telle que nous l'avons décrite dans l'annexe B. La réponse est relativement simple :

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

Un appel à la fonction `forName()` va nous charger le driver nécessaire, ici notre `sun.jdbc.odbc.JdbcOdbcDriver`. Nous pourrions aussi communiquer à l'extérieur, avec un serveur SQL de Microsoft, d'Oracle, de Sybase ou MySQL. Dans ces cas-là, il faudrait utiliser le driver approprié, auquel nous ajouterions l'adresse IP du serveur, ainsi que le numéro du port.

C'est certainement ici que nous pourrions faire la remarque que, même si la connectivité peut être garantie avec différents serveurs, il n'est pas certain que toutes les commandes SQL fonctionnent. Il y a en effet des différences. Une personne familière avec les commandes SQL sous Access découvrira tout de suite des différences en ouvrant un ouvrage décrivant l'implémentation de MySQL. Des adaptations pourraient se révéler nécessaires.

La variable `url`, que nous avons appelée ainsi pour des raisons évidentes, est en fait le nom du serveur SQL. Ici, ce nom est un peu particulier, car il correspond à notre entrée définie dans notre administrateur ODBC et se trouve être le premier paramètre de la première instruction importante de notre classe :

```
con = DriverManager.getConnection(url, "", "");
```

La méthode `getConnection()` possède plusieurs implémentations. Nous avons choisi celle avec le nom et le mot de passe de l'utilisateur. Ils sont vides, car nous ne les avons pas définis avec l'administrateur ODBC de Windows. L'instruction :

```
stmt = con.createStatement();
```

est nécessaire pour obtenir un objet de la classe `Statement` avant de pouvoir exécuter des instructions SQL.

Requête SQL

Le langage SQL permet d'exécuter des requêtes dans une base de données. Dans le programme précédent, nous n'avons qu'une seule requête :

```
SELECT Nom, Prenom, Annee FROM Personnes
```

`SELECT` est une commande SQL qui nous permet de sélectionner les champs `Nom`, `Prenom` et `Annee` depuis (`from`) la table `Personnes`. Nous pouvons en fait nous amuser avec ce langage, en utilisant le mode SQL sous Microsoft Access, lorsque nous choisissons une `Requête Selection`. La commande :

```
SELECT Nom,Prenom,Annee FROM Personnes;
```

peut alors être entrée, et nous pouvons nous placer dans Access en mode `Feuilles de données` pour examiner le résultat. Il est possible de ne spécifier qu'une partie des champs :

```
SELECT Nom,Prenom FROM Personnes;
```

La suppression d'`Annee` dans la commande `SELECT` nécessiterait évidemment des modifications de la classe `PersonnesODBC`. Une condition peut aussi être spécifiée :

```
SELECT * FROM Personnes WHERE Annee = 1907;
```

Et seuls les enregistrements avec l'année égale à 1907 seront présentés. Le caractère `*` indique que tous les champs seront exposés. Avant de passer à la dernière partie de programme Java précédent, nous allons montrer son résultat :

```
Exécute la requête
Les entrées dans notre base de données:
Haddock Capitaine est né en 1907
Kaddock Kaptain est né en 1897
```

Pour extraire des données de notre table `Personnes`, il nous faut exécuter une requête que nous avons définie dans la variable `requete` :

```
String requete = "SELECT Nom, Prenom, Annee FROM Personnes";
```

À cet effet, la méthode `executeQuery()` sur l'objet `stmt` de la classe `Statement` sera utilisée. En cas d'erreur quelconque de la requête, qui pourrait être un format incorrect ou une table effacée, la méthode `executeQuery()` générera une exception.

Pour terminer, nous lisons les résultats de la requête dans une boucle sur chaque enregistrement, avec la méthode `next()` sur l'objet `rs`, qui est un `ResultSet` retourné après l'exécution

de notre requête sur les trois colonnes de la table. La méthode `getString()` reçoit le nom des champs tels qu'ils ont été définis dans la commande SQL. Nous avons ici trois possibilités, et le résultat nous montre le contenu de la table avec une ligne de texte formaté par enregistrement.

Nous pourrions utiliser l'AWT ou le Swing de Java pour présenter nos résultats d'une manière plus propre. Ceci nous permettrait d'écrire de vraies applications de base de données, comme nous pouvons le faire avec les outils de Microsoft Access.

Création d'une nouvelle table depuis Java

Comme nous avons déjà créé une base de données et qu'elle est actuellement accessible, rien ne nous interdit de créer de nouvelles tables, avant d'y introduire des données. Nous profiterons de cette occasion pour présenter les instructions SQL pour la création de table, ainsi que la modification d'enregistrement existant. Nous allons remarquer, très rapidement, que le code qui va suivre ressemble comme un frère à notre première classe, `PersonneODBC` :

```
// ODBC 32 bits doit être actif
// ODBC-Java-Test associé à Chap20\bd1.mdb

import java.sql.*;

public class CreationODBC {
 public static void main(String args[]) {
   String url = "jdbc:odbc:ODBC-Java-Test";

   Connection con;
   Statement stmt;

   try {
     Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
   }
   catch(java.lang.ClassNotFoundException e) {
     System.err.print("ClassNotFoundException: ");
     System.err.println(e.getMessage());
     return;
   }

   try {
     System.out.println("Connexion avec ODBC-Java-Test");
     con = DriverManager.getConnection(url, "", "");
     stmt = con.createStatement();

     System.out.println("Création de la table Telephone");
     stmt.executeUpdate("CREATE TABLE Telephone
                         (Nom VARCHAR, Prenom VARCHAR, NumTel VARCHAR)");

     System.out.println("Insertion d'objets dans la table Telephone");
     stmt.executeUpdate("INSERT INTO Telephone (Nom, Prenom, NumTel)
```

```
                        VALUES ('Boichat', 'Jean-Bernard', '04698761272')");
      stmt.executeUpdate("INSERT INTO Telephone (Nom, Prenom)
                        VALUES ('Boichat', 'Nicolas')");

      System.out.println("Modifie un enregistrement dans la table Telephone");
      stmt.executeUpdate("UPDATE Telephone SET NumTel = '04698761272'
                        WHERE Prenom = 'Nicolas'");

      System.out.println("Exécute la requête");
      ResultSet rs = stmt.executeQuery("SELECT Nom, Prenom, NumTel FROM Telephone");

      System.out.println("Les entrées dans notre table Telephone: ");
      while (rs.next()) {
        String nom     = rs.getString(1);
        String prenom  = rs.getString(2);
        String numTel  = rs.getString(3);
        System.out.println("Enregistrement: " + nom + " " + prenom + " : " + numTel);
      }

      stmt.close();
      con.close();
    }
    catch(SQLException ex) {
      System.err.println("SQLException: " + ex.getMessage());
    }
  }
}
```

Dans le code précédent, nous découvrons les instructions SQL suivantes :

```
CREATE TABLE Telephone (Nom VARCHAR, Prenom VARCHAR, NumTel VARCHAR);
INSERT INTO Telephone (Nom, Prenom, NumTel)
       VALUES ('Boichat', 'Jean-Bernard', '04698761272');
INSERT INTO Telephone (Nom, Prenom)
       VALUES ('Boichat', 'Nicolas');
UPDATE Telephone SET NumTel = '04698761272'
                WHERE Prenom = 'Nicolas';
SELECT Nom, Prenom, NumTel FROM Telephone;
```

Notre nouvelle table se nommera `Telephone` et contiendra trois colonnes pour le nom, le prénom et le numéro. Ces trois champs seront des `VARCHAR`, c'est-à-dire des chaînes de caractères traditionnelles. L'instruction `CREATE` en SQL fait le travail, et nous constaterons la manière de définir le nom des champs et leur type.

L'instruction `CREATE TABLE` va donc créer la table `Telephone` dans la base de données `db1.mdb`, puisque cette dernière est connectée à ODBC-Java-Test, comme nous l'avons défini au travers de l'administrateur ODBC. Pour définir le nom des champs, il suffit de spécifier, entre parenthèses, la liste de ceux-ci suivie par le type désiré. Ces paires sont séparées par des virgules.

L'instruction `INSERT INTO` nous permet d'introduire des enregistrements dans la base de données. Après le nom de la table, nous devons indiquer, entre parenthèses, le nom des champs affectés. Avec `VALUES()`, nous attribuons des valeurs à chacun des noms indiqués précédemment. Si nous avions un champ avec des entiers, les guillemets ne seraient pas nécessaires pour des chiffres. Il est possible de définir moins de colonnes que la définition de la table. Mais c'est un risque, et il faudrait définir une valeur de défaut lors de la création de la table. Ici, ce n'est pas très important, mais en cas de nécessité il faudrait consulter la documentation de la commande `CREATE`.

La commande `UPDATE` est capable de modifier des enregistrements existants sur des colonnes et des critères particuliers. Le paramètre `SET` définit quel champ sera mis à jour avec sa valeur, et le paramètre `WHERE`, la condition. Ici, tous les enregistrements qui ont le prénom `Nicolas` seront définis avec le même numéro de téléphone.

Nous connaissons déjà la commande `SELECT`, qui nous permet de reprendre dans le code Java le résultat de la requête. Le `getString()` est utilisé ici différemment. Nous utilisons non pas le nom de la colonne, mais son index, qui commence à 1 et qui dépend du nombre de champs définis dans les paramètres de la commande `SELECT`, c'est-à-dire 3 dans notre cas.

Pour les quatre premières instructions SQL, nous devons utiliser la méthode `executeUpdate()`, alors que le select doit utiliser `executeQuery()`.

MySQL et Linux : recommandations

Pour ceux qui désirent approfondir leurs connaissances en Java, en C++ et en SQL, nous recommandons :

- d'acquérir l'ouvrage *MySQL & mSQL* de Randy Jay Yarger, George Reese & Tim King (voir annexe G) ;
- d'installer une version récente de Linux avec une distribution de MySQL ;
- d'installer LAMP (Linux, Apache, MySQL et PHP), `http://fr.wikipedia.org/wiki/LAMP`.

Pour ces deux derniers points, nous pourrions suggérer Ubuntu (voir annexe F). Pour plus d'informations, nous vous recommandons les sites suivants :

- `http://doc.ubuntu-fr.org/mysql`
- `http://www.aide-ubuntu.com/Installer-Apache2-Mysql-5-et-Php-5`

En ce qui concerne MySQL, nous trouverons l'information nécessaire sur les sites :

- `http://www.mysql.com/`
- `http://www.mysql.com/download.html`

Le langage PHP (*Hypertext Preprocessor*) est devenu incontournable pour le développement de pages dynamiques sur Internet. Nous consulterons le site de Wikipédia pour davantage d'informations sur PHP (`http://fr.wikipedia.org/wiki/PHP:_Hypertext_Preprocessor`).

NetBeans (voir annexe E) peut aussi être utilisé pour développer des applications PHP avec des accès MySQL : `http://www.netbeans.org/`.

Beaucoup de serveurs Web sont aujourd'hui accessibles au public et équipés pour développer des applications PHP et MySQL.

Autres choix d'interfaces

Des bibliothèques C++ et une multitude d'exemples et d'interfaces pour d'autres langages, comme Perl, PHP ou Python, sont à disposition. Une version shareware existe pour Windows, mais n'a pas été considérée dans cet ouvrage, car le travail et la préparation pour l'installation de la bibliothèque C++ auraient dépassé les objectifs fixés pour cet ouvrage. Mais cette dernière possibilité reste tout de même une alternative.

Résumé

Nous espérons que ce chapitre aura intéressé un grand nombre de lecteurs. Le traitement et l'accès des bases de données sont des sujets essentiels en informatique. La manière de procéder en Java est élégante et devrait amener certains programmeurs à utiliser ou à élargir leurs connaissances, comme celle de remplir des collections à partir d'une base de données ou d'écrire des programmes interactifs pour le Web (CGI et servlets).

Exercices

1. Modifier la classe `CreationODBC` afin que tous les enregistrements soient effacés si la table existe déjà. Pour ceci nous utiliserons l'instruction SQL `DELETE * FROM Telephone`. Partager les différentes fonctions dans des méthodes qui retourneront une erreur en cas de difficulté.

2. Créer deux tables `Table1` et `Table2`. Les deux tables auront la même structure, avec deux colonnes, contenant un index (`Champ1`) et un texte (`Champ2`). Remplir ces deux tables avec des données et créer une requête, qui extrait les deux champs de texte pour un index déterminé qui existe dans les deux tables. Pour faire cet exercice, nous devrons créer nos tables en SQL avec, par exemple :

   ```
   CREATE TABLE Table1 (Champ1 INTEGER, Champ2 VARCHAR)
   ```

   La recherche sur deux tables pourra se faire ainsi, et sur une seule ligne :

   ```
   SELECT Table1.Champ1, Table1.Champ2, Table2.Champ2
   FROM Table1, Table2
   WHERE Table1.Champ1 = Table2.Champ1
   ```

21

Java et C++ main dans la main : le JNI

Pourquoi et comment utiliser JNI ?

Le JNI (*Java Native Interface*) permet à Java d'utiliser du code écrit dans d'autres langages et avec d'autres compilateurs. Nous choisirons ici notre compilateur C++ et écrirons quelques petites fonctions pour nous donner une idée des possibilités et surtout de la manière de générer le code, qui est loin d'être triviale. C'est pour cette raison qu'une approche avec des exemples simples devrait nous faciliter la tâche.

Il existe de nombreux cas où le JNI s'avère très utile, et nous en citerons quelques-uns :

- réutiliser du code C et C++ existant sans devoir réécrire tout le code en Java ;
- utiliser des API existantes en C et C++ ;
- accéder à des ressources hardware dans lesquelles il ne serait pas possible, ou difficile, d'écrire le pilote en Java ;
- utiliser de l'AWT de Java pour l'interface utilisateur pour du code écrit dans d'autres langages.

Note

Pour les programmeurs qui utilisent les anciennes versions 1.2 ou 1.3 du JDK de Java et qui voudraient compiler les exemples de ce chapitre avec le compilateur C++ de GNU, il faudra modifier l'un des fichiers d'en-tête du JDK distribué par Sun Microsystems. Dans le fichier :

```
jdk1.2\include\win32\jni_md.h
```

ou :

```
jdk1.3.1\include\win32\jni_md.h
```

la ligne :

```
Typedef __int64 jlong;
```

devra être remplacée par :

```
#ifdef __GNUC__
typedef long jlong;
#else
typedef __int64 jlong;
#endif
```

Des salutations d'une bibliothèque C++

Nous avons choisi un premier petit exercice qui consiste à recevoir des salutations d'une bibliothèque écrite en C++ à partir d'un programme Java. Nous allons examiner les détails après avoir présenté ce code :

```
class SalutCPP {
  public native void salutations();

  static {
    System.out.println("Java: chargement de la bibliothèque salut.dll");
    System.loadLibrary("salut");
  }
}

class MonPremierJNI {
  public static void main(String[] args) {
    System.out.println("Java: création d'une instance de SalutCPP");
    SalutCPP monSalut = new SalutCPP();
    System.out.println("Java: appel de la méthode native salutations()");
    monSalut.salutations();
    System.out.println("Java: fin de programme");
  }
}
```

Il y a deux nouveautés dans ce code. La première consiste en la déclaration de la méthode native, qui se fait de cette manière :

```
public native void salutations();
```

Le mot-clé `native` indique que la méthode `salutations()` sera disponible dans une bibliothèque extérieure écrite dans un autre langage. Cette dernière détermine le nom de la méthode

ainsi que d'éventuels paramètres ou des valeurs de retour dont nous verrons les détails plus loin. La deuxième nouvelle instruction :

```
System.loadLibrary("salut");
```

va permettre à la machine virtuelle de Java de charger la bibliothèque `salut`, qui sera ici, sous Windows, une bibliothèque dynamique nommée `salut.dll` qui devra se trouver dans le chemin d'accès (`PATH`) ou dans le répertoire de nos deux classes `MonPremierJNI.class` et `SalutCPP.class`. Le code ci-dessus va compiler sans aucune difficulté, mais ne pourra pas être exécuté immédiatement, puisque le code de la bibliothèque n'a pas encore été préparé ni assemblé.

javah pour le code C++

Après la création de notre classe `SalutCPP.class`, qui contient la définition de la méthode `salutations()`, nous devons à présent exécuter `javah` sur cette classe. Cet outil va nous générer le fichier d'en-tête C++, afin que nous puissions ensuite coder le code correspondant en C++ et créer notre bibliothèque `salut.dll`. En exécutant :

```
javah SalutCPP
```

le code suivant est alors généré dans le fichier `SalutCPP.h` :

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class SalutCPP */

#ifndef _Included_SalutCPP
#define _Included_SalutCPP
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     SalutCPP
 * Method:    salutations
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_SalutCPP_salutations(JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

Il s'agit à présent de créer le code dans le fichier `SalutCPP.cpp`, qui doit inclure au minimum le code de la fonction `Java_SalutCPP_salutations()`. Nous allons le faire de la manière la plus simple qui soit :

```
#include "SalutCPP.h"

#include <iostream>
```

```
using namespace std;

JNIEXPORT void JNICALL Java_SalutCPP_salutations(JNIEnv *, jobject) {
  cout << "C++: salutations depuis la bibliothèque salut.dll" << endl;
}
```

Nous noterons que les définitions de `JNIEXPORT`, de `JNICALL` et de `JNIEnv` sont données dans le fichier d'en-tête `jni.h`, qui est lui-même défini dans notre fichier précédent `SalutCPP.h`.

Création de notre salut.dll

Il reste enfin à compiler ce code C++ et à générer notre bibliothèque Windows dynamique `salut.dll`. Ceci peut se faire avec un `Makefile`, qui contiendra toutes les parties nécessaires à la génération de notre code :

```
JDK = "C:/Program Files/Java/jdk1.6.0_06"
all:   java cpp salut.dll

java:   MonPremierJNI.class SalutCPP.h

cpp:    SalutCPP.o


MonPremierJNI.class:   MonPremierJNI.java
                       javac MonPremierJNI.java

SalutCPP.h:            MonPremierJNI.class
                       javah SalutCPP

SalutCPP.o:            MonPremierJNI.class SalutCPP.cpp
                       g++ -I$(JDK)/include -I$(JDK)/include/win32 -c SalutCPP.cpp

salut.dll:             SalutCPP.o
                       dllwrap --driver-name=c++ --output-def salut.def
                       --add-stdcall-alias -o salut.dll -s SalutCPP.o
```

Si le programmeur décide d'utiliser une autre version du JDK de Java, il devra modifier la première ligne du `Makefile`.

Lorsque nous compilons `SalutCPP.cpp`, il est nécessaire d'inclure, avec le paramètre de compilation `–I`, les chemins d'accès de `jni.h` (ici, `…/jdk1.6.0_06/include`), mais aussi de `jni_md.h`, qui est inclus dans `jni.h` et qui se trouve dans le sous-répertoire `win32` du JDK. Pour la génération de la bibliothèque avec `dllwrap`, nous ne reviendrons pas sur les détails, qui ne sont pas vraiment nécessaires pour comprendre ou étendre plus tard cet exemple.

Enfin, il nous faut revenir à notre programme Java `MonPremierJNI`, que nous pouvons à présent exécuter :

```
java MonPremierJNI
```

Nous obtiendrons le résultat attendu :

```
Java: création d'une instance de SalutCPP
Java: chargement de la bibliothèque salut.dll
Java: appel de la méthode native salutations()
C++: salutations depuis la bibliothèque salut.dll
Java: fin de programme
```

Nous conseillerons ici aux lecteurs de consulter l'annexe C, « Installation et utilisation de Crimson », et son exercice 2.

JNI, passage de paramètres et Swing

L'exemple précédent ne pouvait être plus simple. Aucun paramètre n'était passé à la fonction, et aucune valeur de retour n'était utilisée. Afin d'illustrer ces deux derniers aspects, nous allons profiter de l'occasion pour présenter Swing, avec cette petite application qui apparaîtra comme sur la figure 21-1.

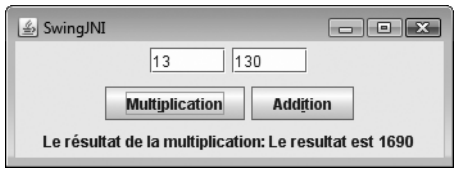

Figure 21-1
Notre premier calcul

Cette application Java est écrite non avec les composants AWT que nous avons déjà présentés, mais avec le dernier kit de développement pour les interfaces graphiques, qui fait partie de Java depuis la version 1.2, Swing. Ce dernier fournit des composants d'interface Java plus performants et portables que l'AWT, mais avec le défaut d'être beaucoup plus lent que son prédécesseur. Ce problème tend à s'atténuer sur des machines récentes qui ont à la fois plus de mémoire et de vitesse de processeur.

La fenêtre présentée possède deux champs d'entrée pour des nombres. Les deux boutons nous permettent de choisir entre une multiplication ou une addition de ces nombres. Le résultat sera présenté sous les deux boutons avec le message approprié.

Bien que ces deux opérations ne présentent aucune difficulté, nous allons nous rendre la tâche plus compliquée, en codant l'opération arithmétique dans une bibliothèque dll écrite en C++. Pour ce faire, nous avons choisi d'écrire une classe Java nommée `OperationCPP`, qui va nous permettre d'exécuter une opération définie par le paramètre du constructeur. Cette opération pourra être répétée au travers de la méthode :

```
execute(int num1, int num2)
```

Voici donc notre classe, qui va définir et effectuer l'interface JNI :

```
public class OperationCPP {
  private String lOperation;

  public native String operation(String loper, int num1, int num2);

  static {
    System.out.println("Java: chargement de la bibliothèque operation.dll");
    System.loadLibrary("operation");
  }

  public OperationCPP(String lOper) {
    lOperation = lOper;
  }

  public String execute(int num1, int num2) {
    return operation(lOperation, num1, num2);
  }
}
```

Lorsqu'une instance de la classe `OperationCPP` a été obtenue, il ne sera plus possible de changer d'opération puisqu'elle a été déterminée par le paramètre du constructeur. C'est un choix que nous avons fait, et uniquement pour illustrer notre exemple JNI. Comme dans l'exemple précédent, une bibliothèque, nommée ici `operation.dll`, sera chargée à la première instantation de la classe `OperationCPP`. La méthode native `operation()` devra encore être codée. Avant d'écrire notre application Swing, il est judicieux et préférable d'écrire une classe de test pour vérifier notre classe `OperationCPP` :

```
public class MultiplicationJNI {
  public static void main(String[] args) {
    OperationCPP maMult = new OperationCPP("Multiplication");
    System.out.println("Résultat: " + maMult.execute(12, 13));
  }
}
```

Après avoir compilé la classe `MultiplicationJNI` et avant de pouvoir l'exécuter, nous devrons écrire le code C++ correspondant à notre interface JNI. De la même manière que dans notre premier exemple, nous devrons générer le fichier d'en-tête C++ avec la commande :

```
javah OperationCPP
```

Cette dernière va nous créer notre fichier `OperationCPP.h` :

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class OperationCPP */

#ifndef _Included_OperationCPP
#define _Included_OperationCPP
#ifdef __cplusplus
```

```
extern "C" {
#endif
/*
 * Class:     OperationCPP
 * Method:    operation
 * Signature: (Ljava/lang/String;II)Ljava/lang/String;
 */
JNIEXPORT jstring JNICALL Java_OperationCPP_operation(JNIEnv *, jobject,
                                                      jstring, jint, jint);

#ifdef __cplusplus
}
#endif
#endif
```

Nous devons rappeler qu'à l'origine notre définition était la suivante :

```
public native String operation(String loper, int num1, int num2);
```

et nous voyons à présent apparaître de nouveaux types de variables qui sont `jstring` et `jint`, dont nous pouvons trouver la définition (`typedef`) dans les fichiers d'en-tête `jni.h` et `jni_md.h` ; nous pouvons consulter ces derniers dans la distribution du JDK de Sun Microsystems dans les répertoires `../jdk1.6.0_06/include` et `../jdk1.6.0_06/include/win32`.

`jint`, sous Windows, sera interprété comme un `long`. Afin d'enrichir notre exemple, nous avons défini une classe C++ nommée `Operation`, qui conservera l'opération arithmétique de la même manière que nous l'avons définie en Java. La définition de la classe sera codée dans le fichier `Operation.h` :

```
#include <string>

class Operation {
  private:
    string loperation;     // nom de l'opération

  public:
    Operation(const char *loper);
    int execute(int num1, int num2);
};
```

et l'implémentation dans le fichier `OperationCPP.cpp`, qui contiendra aussi l'interface JNI nécessaire à notre application :

```
// OperationCPP.cpp
#include "OperationCPP.h"
#include "Operation.h"

#include <iostream>
#include <sstream>

using namespace std;
```

```
Operation::Operation(const char *loper) {
   loperation = loper;
}

int Operation::execute(int num1, int num2) {
   if (loperation == "Multiplication") {
     return num1 * num2;
   }

   if (loperation == "Addition") {
     return num1 + num2;
   }
   return 0;
}


JNIEXPORT jstring JNICALL Java_OperationCPP_operation(JNIEnv *env, jobject obj,
➥jstring loperation, jint jnum1, jint jnum2) {
   const char *loper = env->GetStringUTFChars(loperation, 0);

   Operation mon_oper(loper);

   env->ReleaseStringUTFChars(loperation, loper);

   int resultat = mon_oper.execute(jnum1, jnum2);

   ostringstream un_stream;
   un_stream << "Le résultat est " << resultat << endl;

   return env->NewStringUTF(un_stream.str().c_str());
}
```

La méthode de classe `Operation::execute()` est des plus traditionnelles. L'attribut `loperation` de la classe `Operation` détermine la fonction à exécuter, c'est-à-dire une addition ou une multiplication.

La fonction JNI `Java_OperationCPP_operation()`, qui est responsable de l'interface avec Java, est d'un autre domaine de complexité. Nous avons choisi volontairement un des paramètres comme un `String` ainsi que le retour de la fonction. Ce type d'interface pourrait être accepté comme standard. Un `String` peut en effet être composé d'un type quelconque dont une conversion est possible en Java ou encore de plusieurs morceaux de types différents avec un séparateur choisi préalablement.

La fonction `GetStringUTFChars()` :

```
const char *loper = env->GetStringUTFChars(loperation, 0);
```

retourne un pointeur à une chaîne de caractères (UTF-8 : codé sur 8 bits) à partir du `String` Java qui devrait être `Addition` ou `Multiplication`. `loper` restera valide jusqu'à l'appel de `ReleaseStringUTFChars()`. Comme le `resultat` est un entier, nous utilisons un `ostringstream` qui va nous convertir la valeur de retour en une chaîne de caractères.

Enfin, la fonction :

```
env->NewStringUTF(un_stream.str().c_str());
```

construit un nouvel objet `java.lang.String` composé d'une chaîne de caractères Unicode. De `un_stream`, nous retirons un `String` avec `str()` et ensuite une chaîne de caractères avec `c_str()` demandée par `NewStringUTF()`. Ces formes C et C++ sont plus que décourageantes.

Pour terminer, nous dirons que les fonctions `GetStringUTFChars()`, `Release-StringUTFChars()` et `NewStringUTF()` utilisent la variable `env` (de type `JNIEnv`), qui est un pointeur à une structure possédant elle-même une liste de pointeurs à toutes les fonctions JNI.

Il faut noter que toutes ces fonctions sont documentées dans la distribution du JDK de Sun Microsystems et qu'un bon programmeur C ou C++ ne devrait pas avoir de grandes difficultés à utiliser d'autres fonctions ou à adapter ce code avec d'autres compilateurs.

La dernière phase est la compilation de notre fichier `OperationCPP.cpp` et la création de la bibliothèque `operation.dll`.

```
g++ -O2 -I$(JDK)/include -I$(JDK)/include/win32 -c OperationCPP.cpp
dllwrap --driver-name=c++ --output-def operation.def
     --add-stdcall-alias -o operation.dll -s OperationCPP.o
```

Avant de passer à notre interface graphique en Java (Swing), il est conseillé de vérifier notre code en exécutant :

```
java MultiplicationJNI
```

Le programme nous retournera :

```
Java: chargement de la bibliothèque operation.dll
Résultat: 156
```

Notre interface Swing

Nous ne présenterons Swing que très brièvement, car ce n'est pas le but de cet ouvrage. Un bon programmeur Java qui possède, en outre, quelques connaissances de l'AWT, que nous avons aussi rapidement présenté, ne devrait pas avoir de difficultés à comprendre ce code. Il est cependant suffisamment intéressant pour donner l'envie de l'adapter ou d'écrire d'autres applications plus substantielles.

Si nous examinons à nouveau notre application Swing :

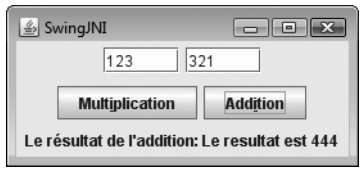

Figure 21-2

Notre second calcul

et le code qui suit :

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class SwingJNI {
  public void creationDesComposants(JFrame frame) {
    final JLabel label = new JLabel("Le résultat de la multiplication ou de
    ➥l'addition");

    final ChampNombre entree1 = new ChampNombre(10, 5);
    final ChampNombre entree2 = new ChampNombre(20, 5);

    JButton bouton1 = new JButton("Multiplication");

    bouton1.setMnemonic(KeyEvent.VK_I);
    bouton1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        OperationCPP oper = new OperationCPP("Multiplication");
        int num1 = entree1.getValue();
        int num2 = entree2.getValue();

        label.setText("Le résultat de la multiplication: " + oper.execute(num1,
        ➥num2));
      }
    });
    label.setLabelFor(bouton1);

    JButton bouton2 = new JButton("Addition");

    bouton2.setMnemonic(KeyEvent.VK_I);
    bouton2.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        OperationCPP oper = new OperationCPP("Addition");
        int num1 = entree1.getValue();
        int num2 = entree2.getValue();

          label.setText("Le résultat de l'addition: " + oper.execute(num1, num2));

      }
    });
    label.setLabelFor(bouton2);

    JPanel panneau1 = new JPanel();
    panneau1.setLayout(new FlowLayout());
    panneau1.add(entree1);
    panneau1.add(entree2);
    JPanel panneau2 = new JPanel();
    panneau2.add(bouton1);
    panneau2.add(bouton2);
```

```
    JPanel panneau3 = new JPanel();
    panneau3.add(label);

    frame.getContentPane().add(panneau1, BorderLayout.NORTH);
    frame.getContentPane().add(panneau2, BorderLayout.CENTER);
    frame.getContentPane().add(panneau3, BorderLayout.SOUTH);
    return;
  }

  public static void main(String[] args) {
    try {
      UIManager.setLookAndFeel(
      UIManager.getCrossPlatformLookAndFeelClassName());
    }
    catch (Exception e) { }

    JFrame frame = new JFrame("SwingJNI");
    SwingJNI app = new SwingJNI();

    app.creationDesComposants(frame);

    frame.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    });

    frame.pack();
    frame.setVisible(true);
  }
}
```

nous remarquerons d'abord les trois `JPanel`, qui sont `panneau1`, `panneau2` et `panneau3`. Dans le premier, nous découvrons nos champs d'entrées, avec nos deux `123` et `321`. Le deuxième, `JPanel`, contient nos boutons de la sélection de l'opération arithmétique, et le dernier, notre `panneau3`, l'étiquette `JLabel`, qui nous indiquera les résultats. Ces trois panneaux sont ajoutés à la fenêtre de base (`JFrame`) et nous permettent d'obtenir une présentation (*layout*) décente. Il y a d'autres manières de faire, mais cela dépasse de loin les buts que nous nous sommes fixés dans le cadre de cet ouvrage.

Les deux méthodes `actionPerformed()` sont les points d'entrée de nos deux opérations arithmétiques. Nous y retrouvons le paramètre du constructeur de la classe `operationCPP` et la méthode `execute()`.

La partie la plus intéressante est liée à la classe `ChampNombre`, que nous avons écrite afin de filtrer les caractères qui sont entrés :

```
import javax.swing.*;
import javax.swing.text.*;

import java.awt.Toolkit;
```

```
import java.text.NumberFormat;
import java.text.ParseException;
import java.util.Locale;

public class ChampNombre extends JTextField {
  private Toolkit toolkit;
  private NumberFormat integerFormatter;

  public ChampNombre(int value, int columns) {
    super(columns);
    toolkit = Toolkit.getDefaultToolkit();
    integerFormatter = NumberFormat.getNumberInstance(Locale.FRANCE);
    integerFormatter.setParseIntegerOnly(true);
    setValue(value);
  }

  public int getValue() {
    int retVal = 0;
    try {
      retVal = integerFormatter.parse(getText()).intValue();
    }
    catch (ParseException e) {
      toolkit.beep();
    }
    return retVal;
  }

  public void setValue(int value) {
    setText(integerFormatter.format(value));
  }

  protected Document createDefaultModel() {
    return new DocumentNombre();
  }

  protected class DocumentNombre extends PlainDocument {
    public void insertString(int pos, String str, AttributeSet as)
                                          throws BadLocationException {
      char[] source   = str.toCharArray();
      char[] resultat = new char[source.length];
      int j = 0;

      for (int i = 0; i < resultat.length; i++) {
        if (Character.isDigit(source[i])) {
          resultat[j++] = source[i];
        }
        else {
          toolkit.beep();  // si une lettre est entrée
        }
      }
```

```
        super.insertString(pos, new String(resultat, 0, j), as);
      }
    }
  }
```

La classe `ChampNombre` étend la classe `JTextField` de Swing, et nous l'utilisons en lieu et place du traditionnel `JTextField`. Cette nouvelle classe nous permettra de ne voir apparaître que des nombres dans nos deux champs d'entrées. Les méthodes `setValue()` et `getValue()` nous permettent de déposer ou de recevoir nos deux nombres. La surcharge de la méthode `createDefaultModel()` de la classe `JTextField` est essentielle. À cet effet nous avons créé une autre classe, `DocumentNombre`, qui ne fait rien d'autre que filtrer chaque caractère individuellement, en ignorant tout ce qui n'est pas un chiffre. La classe `Document` sert de modèle pour les composants Swing.

L'utilisation de la classe `Toolkit` et de la méthode `beep()` pourrait apparaître dans d'autres applications. Une alarme sonore est toujours intéressante.

Pour terminer, et avant d'exécuter notre application :

```
java SwingJNI
```

il faudra compiler nos deux classes :

```
javac ChampNombre.java
javac SwingJNI.java
```

Nous rappellerons que les différentes phases de la compilation, qui sont plus complexes dans ce cas-ci, peuvent être regroupées avantageusement dans un `Makefile`. Ce dernier se trouve sur le CD-Rom accompagnant cet ouvrage.

Une analyse plus profonde des résultats nous montrerait que, si les valeurs des entrées dépassent une certaine limite, le résultat peut devenir négatif ou nul. Un traitement correct de ces cas d'erreurs devrait être considéré.

Nous n'avons pas jugé nécessaire d'inventer un exercice. Cependant, nous répéterons que la clé du JNI et son point de départ se situent dans le `Makefile`. C'est à cet endroit qu'apparaissent clairement les différentes phases de la création d'une telle interface.

Résumé

Ce qu'il faut retenir de la technologie JNI est la possibilité de réutiliser du code existant faisant partie du système lui-même, ou encore d'applications qui ont nécessité des années de développement. Pour cet ouvrage, il s'agit également d'un lien possible entre les deux langages Java et C++.

22

Quelques applications usuelles

Le but de ce chapitre est de titiller l'esprit des programmeurs afin de leur donner des sujets de programmes d'application en Java et C++. Certains de ces sujets pourraient d'ailleurs convenir à des projets de fin d'études (voir en fin d'annexe G, « Rechercher des sujets de travaux pratiques »). Tous ces sujets ont déjà été partiellement traités dans cet ouvrage, mais nous avons préféré les inclure ici en vue de conserver l'identité de chaque chapitre.

Coupons et collons

Après avoir étudié les différents aspects des entrées et sorties en C++ et en Java, il nous a semblé intéressant de montrer un exemple complet d'application. Une bonne maîtrise des entrées et des sorties est un avantage certain lorsqu'il s'agit d'écrire de petits outils système.

Combien de fois nous sommes-nous trouvés en face du problème ? Mon fichier est trop grand, il ne tient pas sur une disquette. Ou alors il est tellement énorme que je ne peux le joindre à un e-mail ! Ou encore je ne parviens pas à transférer le JDK 1.6 chez un ami, car soit la ligne n'est pas assez bonne, soit le transfert par FTP s'interrompt régulièrement.

La solution est simple : nous coupons en morceaux et nous reconstruisons plus tard. C++ et Java sont deux langages tout à fait appropriés pour ce genre d'exercice. Cependant, il nous faut tout de même construire un outil un peu plus intelligent pour le traitement des erreurs (disquette fautive, disquette confondue avec une autre, etc.).

Commençons donc par un peu de design :

- Par défaut, le fichier sera coupé pour remplir une disquette haute densité (1 457 664 octets).
- Si un paramètre est spécifié, il sera considéré comme un nombre de mégaoctets correspondant à la dimension des morceaux.
- Le résultat sera composé de fichiers au format 8.3 de DOS, et un fichier de contrôle `.acr` permettra de reconstruire le fichier d'origine avec son nom de départ.
- Un mécanisme simple de contrôle doit permettre d'identifier les erreurs. Le fichier `.acr` contiendra l'information nécessaire.
- En cas d'erreur, le fichier fautif, c'est-à-dire le morceau, doit être identifiable.

Afin de vérifier le contenu du fichier, nous utilisons l'opérateur ^ (XOR).

Nous allons à présent aborder la partie du code qui va traiter notre problème et qui est relativement substantielle. Cette fois-ci, nous ferons une analyse globale pour les deux versions Java et C++. Nous commencerons par la partie « Coupe », qui consiste à couper notre fichier en morceaux, et nous terminerons par la reconstruction avec « Colle ». Le listing du code Java est présenté en premier, car il nous donne un meilleur aperçu de la structure avec ses séries forcées de `try` et `catch()`.

Nous coupons en Java

```
import java.io.*;

public class Coupe {
  private String  mon_fichier;
  private int     dimension;
  private String  merreur;
  private String  facr;
  static public String  fin = "\r\n";

  public Coupe(String le_fichier, int la_dim) {
    mon_fichier = le_fichier;
    dimension = la_dim;

    int index_point = mon_fichier.indexOf(".");

    if (index_point > 8) index_point = 8;
    else {
      if (index_point < 0) {
        index_point = mon_fichier.length();
        if (index_point > 8) index_point = 8;
      }
    }

    if (index_point == 0)  facr = "xxx";
    else  facr = mon_fichier.substring(0, index_point);
```

```
}

public boolean coupe() {
  FileOutputStream out    = null;
  FileWriter outacr = null;
  byte[] tampon = new byte[512];
  int octets_lus = 512;   // nombre d'octets lus
  int octets_trans = 0;   // nombre d'octets transférés
  int total = 0;          // nombre total d'octets transférés
  int num_morceau  = 1;
  String le_morceau = null;
  int cs1 = 0;
  int cs2 = 0;

  try {
    FileInputStream in = new FileInputStream(new File(mon_fichier));

    try {
      outacr = new FileWriter(new File(facr + ".acr"));
      outacr.write(mon_fichier + fin);
    }
    catch (IOException e) {
      merreur = "Fichier " + facr + " ne peut être écrit";
      return false;
    }

    while ((octets_lus = in.read(tampon)) > 0) {
      total += octets_lus;

      if (octets_trans == 0) {
        le_morceau = facr + "." + num_morceau;
        try {
          cs1 = num_morceau;  // toujours différent si fichier identique
          cs2 = 0;
          out = new FileOutputStream(new File(le_morceau));
          outacr.write(le_morceau + fin);
        }
        catch (IOException e) {
          merreur = "Fichier " + le_morceau + " ne peut être écrit";
          return false;
        }
      }

      for (int j = 0; j < octets_lus; j++) {
        cs1 = cs1 ^ (int)tampon[j++];

        if (j != octets_lus) {
          cs2 = cs2 ^ (int)tampon[j];
        }
      }
```

```
        try {
          out.write(tampon, 0, octets_lus);
          octets_trans += octets_lus;
          if (octets_trans == dimension) {
            out.close();
            outacr.write(((256*cs1) +  cs2) + fin);
            octets_trans = 0;
            num_morceau++;
          }
        }
        catch (FileNotFoundException e) {
          merreur = "Fichier " + le_morceau + " ne peut être écrit";
          return false;
        }
      }

      if (octets_trans > 0) {
        outacr.write(((256*cs1) +  cs2) + fin);
        out.close();
      }

      outacr.write("EnD" + fin);
      outacr.write(total + fin);
      outacr.close();
      in.close();
    }
    catch (IOException e) {
      merreur = "Fichier " + mon_fichier + " n'existe pas: " + e;
      return false;
    }
    return true;
  }

  public String erreur_message() {
    return merreur;
  }

  public static void main(String[] args) {
    int args_len = args.length;

    if ((args_len < 1) || (args_len > 2)) {
      System.err.println("Nombre de paramètres invalide");
      System.err.println("java Coupe fichier  (pour disquettes)");
      System.err.println("java Coupe dim fichier  (en fichiers de dim >= 1024)");
      System.err.println("java Coupe Kn fichier  (en fichiers de n Kilo octets)");
      System.err.println("java Coupe Mn fichier  (en fichiers de n Mega octets)");
      return;
    }

    int fi = 0;
    int adim = 1457664;   // dimension d'une disquette
```

```
    if (args_len == 2) {  // dimension donnée
      fi = 1;
      char first_char = args[0].charAt(0);
      if ((first_char == 'K') || (first_char == 'M')) {
        adim = 1024 * Integer.parseInt(args[0].substring(1, args[0].length()));
        if (first_char == 'M')  adim *= 1024;
      }
      else {
        adim = Integer.parseInt(args[0]);
        if ((adim % 512) != 0) {
          System.err.println("La dimension doit être un multiple de 512");
          return;
        }
      }
    }

    Coupe ma_coupe = new Coupe(args[fi], adim);
    if (ma_coupe.coupe()) {
      System.out.println("Coupe terminée correctement");
    }
    else {
      System.err.println("Coupe erreur: " + ma_coupe.erreur_message());
    }
  }
```

Nous coupons en C++

```
// coupe.cpp
#include <string>
#include <iostream>
#include <sstream>
#include <fstream>

using namespace std;

class Coupe {
 private:
   string    mon_fichier;
   string    facr;
   int       dimension;
   string    merreur;

 public:
   Coupe(const string le_fichier, const int la_dim);
   bool    coupe();
   string  erreur_message() {
     return merreur;
   }
};
```

```
Coupe::Coupe(const string le_fichier, const int la_dim) {
 mon_fichier = le_fichier;
 dimension = la_dim;

 int index_point = mon_fichier.find(".");

 if (index_point > 8) index_point = 8;
 else {
   if (index_point < 0) {
     index_point = mon_fichier.length();
     if (index_point > 8) index_point = 8;
   }
 }

 if (index_point == 0)  facr = "xxx";
 else  facr = mon_fichier.substr(0, index_point);
}

bool Coupe::coupe() {
 ifstream  in;
 ofstream  out;
 ofstream  outacr;
 string fileName;

 char tampon[512];
 int octets_lus = 512;   // nombre d'octets lus
 int octets_trans = 0;   // nombre d'octets transférés
 int total = 0;          // nombre total d'octets transférés
 int num_morceau  = 1;
 int cs1 = 0;
 int cs2 = 0;

 in.open(mon_fichier.c_str(), ios::in|ios::binary);
 if (in.fail() != 0) {
   merreur = "Fichier " + mon_fichier + " ne peut être ouvert";
   return false;
 }

 ostringstream oss;
 oss << facr << ".acr";
 fileName = oss.str();

 outacr.open(fileName.c_str(), ios::out);
 if (outacr.fail() != 0) {
   merreur = "Fichier " + fileName + " ne peut être écrit";
   return false;
 }

 outacr << mon_fichier << endl;

 for (;;) {
```

```
    in.read(tampon, 512);
    if ((octets_lus = in.gcount()) <= 0)  break;

    total += octets_lus;

    if (octets_trans == 0) {
      ostringstream strout;
      strout << facr << "." << num_morceau;
      fileName = strout.str();

      out.open(fileName.c_str(), ios::out|ios::binary);
      if (out.fail() != 0) {
        merreur = "Fichier " + fileName + " ne peut être ouvert";
        return false;
      }

      cs1 = num_morceau;  // toujours différent si fichier identique
      cs2 = 0;
      outacr << fileName << endl;
    }

    for (int j = 0; j < octets_lus; j++) {
      cs1 = cs1 ^ (int)tampon[j++];

      if (j != octets_lus) {
        cs2 = cs2 ^ (int)tampon[j];
      }
    }

    out.write(tampon, octets_lus);
    octets_trans += octets_lus;
    if (octets_trans == dimension) {
      out.close();
      outacr << ((256*cs1) +  cs2) << endl;
      octets_trans = 0;
      num_morceau++;
    }
  }

  if (octets_trans > 0) {
    outacr << ((256*cs1) +  cs2)  << endl;
    out.close();
  }

  outacr << "EnD" << endl;
  outacr << total << endl;
  outacr.close();
  in.close();
  return true;
}
```

```
int main(int argc, char *argv[]) {
 string file_name;    // nom du fichier à couper
 int fi = 1;          // position dans argv
 int cdim = 1457664;  // dimension d'une disquette

 if ((argc < 2) || (argc > 3)) {
   cerr << "Nombre invalide de paramètres" << endl;
   cerr << "coupe fichier      (pour disquettes)" << endl;
   cerr << "coupe dim fichier  (en fichiers de dim >= 1024 octets)" << endl;
   cerr << "coupe Kn fichier   (en fichiers de n Kilo-octets)" << endl;
   cerr << "coupe Mn fichier   (en fichiers de n mégaoctets)" << endl;

   file_name = "prgm.exe";
   cdim = 102400;  // K100
   // return -1;
 }
 else {
   if (argc == 3) { // dimension données
     fi = 2;
     char *pdimension;
     pdimension = &argv[1][0];

     if ((argv[1][0] == 'K') || (argv[1][0] == 'M')) {
       pdimension++;
       istringstream(pdimension) >> cdim;
       cdim *= 1024;
       if (argv[1][0] == 'M') {
           cdim *= 1024;
       }
     }
     else {
       istringstream(pdimension) >> cdim;
     }
   }

   if (cdim < 1024) {
     cerr << "La dimension des fichiers doit être >= 1024" << endl;
     return -1;
   }

   file_name = (char *)argv[fi];
 }

 Coupe ma_coupe(file_name, cdim);

 if (ma_coupe.coupe() == false) {
   cerr << "Erreur: " << ma_coupe.erreur_message() << endl;
   return -1;
 }
```

```
 cout << "Coupe terminée correctement" << endl;
 return 0;
}
```

La partie du `main()` est réduite à un minimum. Si nous avons un ou plusieurs paramètres, nous noterons l'utilisation de la variable `fi` pour l'accès à la position correcte du nom du fichier dans le tableau des arguments (`args` et `argv`). Nous le comprendrons mieux après avoir examiné la commande, ses paramètres et son résultat :

```
coupe.exe K100 prgm.exe
Coupe terminée correctement
```

Nous couperons donc le fichier `prgm.exe` en morceaux de 100 Ko.

La vérification des cas d'erreurs aurait pu être étendue. Accepter un `K` ou un `M` minuscule serait un choix aussi acceptable. Un paramètre avec une autre lettre pourrait donner un message d'erreur plus significatif.

Le lecteur devrait avoir été immédiatement surpris par l'instruction :

```
if ((adim % 512) != 0)
```

Cette restriction du programme est essentielle pour une construction simple du programme. Durant l'analyse, nous avons constaté que la valeur de 1 457 664, c'est-à-dire la capacité maximale d'une disquette haute densité, était en fait un multiple de 512. Des coupes en kilo-octets ou mégaoctets sont aussi des multiples de 512. Comme nous devons générer plusieurs morceaux à partir d'un fichier qui peut être énorme, il serait impensable de copier caractère par caractère. Ce serait beaucoup trop lent, et 512 est sans doute une limite inférieure, comme nous l'avons vu au chapitre 15 sur les performances. Lors de la lecture par blocs de 512 octets et de la copie dans un morceau, il y a un moment où nous devrons fermer le morceau pour en ouvrir un autre. Comme les morceaux sont des multiples de 512, le processus de lecture va coïncider avec l'écriture. Il n'y aura pas la nécessité de copier les caractères restants depuis le tampon de lecture et de garder un index de position. De cette manière la conception et la structure du programme s'en trouvent simplifiées.

Le constructeur de la classe `Coupe` possède un minimum de fonctionnalités, et c'est une bonne habitude. Le string `facr` est construit pour limiter le nom des morceaux dans un format DOS 8.3. Le `xxx` est un cas particulier où le nom du fichier commence par un point, ce qui peut être le cas sous Linux. Le code du `main()` et du constructeur est tout à fait dans la ligne de la définition d'une classe qui pourrait être reprise sans retouche avec une interface utilisateur graphique du style AWT ou Swing en Java. Tous les paramètres qui sont préparés, décodés et contrôlés dans le `main()` se feraient alors dans le code associé aux méthodes des classes AWT ou Swing. La méthode `erreur_message()` va aussi dans cette direction, car aucun message ne sera directement sorti par la méthode principale `coupe()`. En cas d'erreur, le programme `main()` décidera de sortir le message d'erreur sur la console, comme procéderait un programme UI, qui ferait apparaître ce message textuellement dans une fenêtre.

Un exemple de fichier .acr

Si nous examinons le fichier `prgm.acr` que nous avons généré par la commande :

```
coupe.exe K100 prgm.exe
```

ci-dessus, nous y découvrirons ceci :

```
prgm.exe
prgm.1
29947
prgm.2
5759
prgm.3
32392
prgm.4
32374
EnD
329253
```

Dans ce fichier, nous retrouvons le nom du fichier d'origine et les différents morceaux. Ces derniers sont entrelacés avec des nombres, dont nous comprendrons rapidement la signification. Dans le cas présent, tous les fichiers `.n` auront une dimension de 100 Ko, comme nous l'avons défini dans la commande. Le dernier chiffre, après le code `EnD`, correspond à la variable `total`, qui nous donne en fait la dimension du fichier d'origine.

Les variables `cs1` et `cs2` correspondent à notre mécanisme pour identifier la validité des fichiers et les erreurs possibles. C'est cette valeur sur deux octets que nous retrouvons dans notre fichier `prgm.acr`, avec le nom du fichier coupé. Un fichier pourrait avoir été mal écrit sur disquette ou coupé pendant un transfert FTP. Nous utilisons l'opérateur `^` (XOR) sur chaque caractère pair et impair.

Il y a un certain nombre de petits trucs, comme le `cs1` initialisé avec le numéro du morceau, afin de diminuer les risques, minimes, d'obtenir une même valeur. Enfin, si la dimension du fichier est impaire, il ne faudra pas calculer le dernier `cs2`. Il ne faudra pas s'étonner de retrouver des valeurs négatives dans un fichier `.acr`, c'est tout à fait correct.

Recollons les morceaux

La reconstruction du fichier à partir de morceaux définis dans un fichier `.acr` est un peu plus simple. Il suffit en fait de lire les fichiers, les uns après les autres, bloc par bloc, et de les insérer dans un nouveau fichier après un certain nombre de vérifications.

Nous collons en Java

```
import java.io.*;

public class Colle {
  private String fichier_acr;
  private String merreur;
```

```
public Colle(String le_fichier) {
  fichier_acr = le_fichier;
}

public boolean colle() {
  String mon_fichier;
  String un_fichier;
  int dim_total = 0;      // dimension du fichier construit
  byte[] tampon = new byte[512];
  int octets_lus   = 0;   // nombre d'octets lus
  int octets_trans = 0;   // nombre d'octets transférés
  int num_morceau  = 0;
  int cs  = 0;            // cs du fichier acr
  int cs1 = 0;
  int cs2 = 0;

  try {
    BufferedReader inacr = new BufferedReader(new FileReader(fichier_acr));

    try {
      mon_fichier = inacr.readLine();   // le fichier final
    }

    catch(IOException ioe) {
      merreur = "Erreur de lecture du fichier .acr";
      return false;
    }

    if (mon_fichier.length() > 50) {
      merreur = "Fichier trop long (>50) ou erreur de lecture du fichier .acr";
      return false;
    }

    try {
      FileOutputStream out =new FileOutputStream(new File(mon_fichier));

      try {
        for (;;) {
          un_fichier = inacr.readLine();
          if (un_fichier == null) {
             merreur = "Fichier .acr incorrect";
             return false;
          }

          if (un_fichier.equals("EnD")) {
            out.close();
            dim_total = Integer.parseInt(inacr.readLine());

            if (dim_total == octets_trans)  return true;
            else {
```

```
            merreur = "Nombre invalide de caractères transférés";
            return false;
          }
        }
        cs = Integer.parseInt(inacr.readLine());
        num_morceau++;

        cs1 = num_morceau;
        cs2 = 0;

        FileInputStream in = new FileInputStream(un_fichier);
        try {
          for (;;) {
            octets_lus = in.read(tampon);
            if (octets_lus <= 0)  break;

            for (int j = 0; j < octets_lus; j++) {
              cs1 = cs1 ^ (int)tampon[j++];

              if (j != octets_lus) {
                 cs2 = cs2 ^ (int)tampon[j];
              }
            }

            out.write(tampon, 0, octets_lus);
            octets_trans += octets_lus;
          }

          in.close();
        }
        catch(IOException ioe) {
           merreur = "Erreur de lecture pour " + un_fichier;
           return false;
        }

        if (cs != ((256*cs1) +  cs2)) {
           merreur = "Acr invalide pour le fichier " + un_fichier;
           return false;
        }
      }
    }
    catch(IOException ioe) {
      merreur = "Fichier .acr invalide";
      return false;
    }
  }
  catch (FileNotFoundException e) {
      merreur = "Le fichier " + mon_fichier + " ne peut être écrit";
      return false;
  }
```

```
    }
    catch (IOException e) {
      merreur = "Le fichier " + fichier_acr + " n'existe pas";
      return false;
    }
  }

  public String erreur_message() {
    return merreur;
  }

  public static void main(String[] args) {
    int args_len = args.length;

    if (args.length != 1) {
      System.err.println("Nombre de paramètres invalides");
      System.err.println("java Colle fichier_acr");
      return;
    }

    String le_fichier_arc = args[0];

    if (le_fichier_arc.lastIndexOf(".acr") != (le_fichier_arc.length() - 4)) {
      System.err.println("Ce n'est pas un fichier .acr");
      return;
    }

    Colle ma_colle = new Colle(le_fichier_arc);
    if (ma_colle.colle()) {
      System.out.println("Colle terminée correctement");
    }
    else {
      System.err.println("Colle erreur: " + ma_colle.erreur_message());
    }
  }
}
```

Nous collons en C++

```
// colle.cpp
#include <string>
#include <iostream>
#include <fstream>

using namespace std;

class Colle {
 private:
   string    fichier_acr;
   string    merreur;
```

```
 public:
   Colle(const string le_fichier);
   bool    colle();
   string  erreur_message() {
     return merreur;
   }
};


Colle::Colle(const string le_fichier) {
 fichier_acr = le_fichier;
}


bool Colle::colle() {
 string    mon_fichier;
 string    un_fichier;
 // ifstream  in;
 ifstream  inacr;
 ofstream  out;
 int dim_total = 0;      // dimension du fichier construit
 char tampon[512];
 int octets_lus   = 0;   // nombre d'octets lus
 int octets_trans = 0;   // nombre d'octets transférés
 int num_morceau  = 0;
 int cs  = 0;             // cs du fichier acr
 int cs1 = 0;
 int cs2 = 0;

 inacr.open(fichier_acr.c_str(), ios::in);
 if (inacr.fail() != 0) {
   merreur = "Erreur de lecture du fichier .acr";
   return false;
 }

 inacr >> mon_fichier;  // le fichier final

 if (mon_fichier.length() > 50) {
   merreur = "Fichier trop long (>50) ou erreur de lecture du fichier .acr";
   return false;
 }

 out.open(mon_fichier.c_str(), ios::out|ios::binary);
 if (out.fail() != 0) {
   merreur = "Le fichier " + string(mon_fichier)+ " ne peut être écrit";
   return false;
 }

 for (;;) {
   inacr >> un_fichier;
```

```
if (un_fichier.length() < 1) {
  merreur = "Fichier .acr incorrect";
  return false;
}

if (un_fichier.compare("EnD") == 0) {
  out.close();
  inacr >> dim_total;

  if (dim_total == octets_trans)  return true;
  else {
    merreur = "Nombre invalide de caractères transférés";
    return false;
  }
}

inacr >> cs;
num_morceau++;

cs1 = num_morceau;
cs2 = 0;

cout << "TEST: [" << un_fichier.c_str() << "]" << endl;

ifstream  in;
in.open(un_fichier.c_str(), ios::in|ios::binary);
if (in.fail() != 0) {
  merreur = "fichier " + string(un_fichier) + " ne peut être ouvert";
  return false;
}

for (;;) {
  in.read(tampon, 512);
  octets_lus = in.gcount();
  if (octets_lus <= 0)  break;

  for (int j = 0; j < octets_lus; j++) {
    cs1 = cs1 ^ (int)tampon[j++];

    if (j != octets_lus) {
      cs2 = cs2 ^ (int)tampon[j];
    }
  }

  out.write(tampon, octets_lus);
  octets_trans += octets_lus;
}

in.close();

if (cs != ((256*cs1) +  cs2)) {
```

```
      merreur = "Acr invalide pour le fichier " + string(un_fichier);
      return false;
    }
  }
}


int main(int argc, char *argv[]) {
 string le_fichier_arc;

 if (argc != 2) {
   cerr << "Nombre de paramètres invalides" << endl;
   cerr << "Colle fichier_acr" << endl;

   le_fichier_arc = "prgm.acr";
   // return -1;
 }
 else {
   le_fichier_arc = argv[1];
 }
 if (le_fichier_arc.rfind(".acr") != (le_fichier_arc.length() - 4)) {
   cerr << "Ce n'est pas un fichier .acr" << endl;
   return -1;
 }

 Colle ma_colle(le_fichier_arc);

 if (ma_colle.colle()) {
   cout << "Colle terminée correctement" << endl;
 }
 else {
   cerr << "Colle erreur: " << ma_colle.erreur_message() << endl;
 }
}
```

Pour recoller nos morceaux, il faudra compiler le code ci-dessus et exécuter par exemple la commande :

```
colle.exe prgm.acr
Colle terminée correctement
```

Il n'y a pas besoin ici de connaître la dimension des morceaux, car une simple lecture suffira. Nous refaisons le même calcul avec les `cs1` et `cs2`. S'il y a une erreur, nous devrions le remarquer immédiatement. Dans le pire des cas, il y aura encore une vérification avec la dimension du fichier et la variable interne `dim_total`. La méthode `read()` en Java est particulière. Elle va bien lire 512 octets, car elle va déterminer en fait la longueur de la variable `byte[] tampon`.

Est-il possible d'avoir une erreur dans la reconstruction ? Nous ne le pensons pas. Après avoir écrit ces quatre programmes, il suffit de vérifier quelques cas simples, comme de raccourcir un fichier `.n` ou de le remplacer par un autre. Il serait aussi possible de tester

les deux phases avec un fichier réel, par exemple un exécutable, et d'utiliser des outils DOS ou Linux pour comparer les fichiers générés ou de l'exécuter après reconstruction. Il semble aussi évident de combiner l'utilisation des différentes versions, comme celle de « couper » en Java et de « coller » en C++.

Nous pourrions à présent utiliser les techniques décrites au chapitre 15 et mesurer les performances respectives de ces deux versions en Java et C++.

Un message sur notre téléphone mobile

> **Note**
>
> Le site Web et ses différents liens utilisés dans l'exemple qui suit n'existent plus aujourd'hui. Nous aurions pu l'adapter à un autre site qui aurait vraisemblablement subi le même sort très rapidement. De nos jours, ces types de sites Web interactifs sont très volatiles ou subissent de continuelles adaptations.

Nous avons cependant gardé cet exercice qui nous montre un certain nombre de techniques que l'on rencontre fréquemment avec le protocole HTTP.

Cet exercice n'a pas vraiment de sens en C++. En effet, le support pour les bibliothèques Internet ou le protocole HTTP est insuffisant, complexe et de plus différent pour chaque système d'exploitation. Le cœur de cet exercice est loin d'être élémentaire, mais il permettrait d'y ajouter des extensions et des utilitaires pour devenir finalement une application solide. L'exemple choisi, envoyer un message sur un téléphone mobile, est absolument dans la ligne des nouvelles technologies d'aujourd'hui. C'est en fait une interaction avec un formulaire Web que nous pourrions réutiliser pour une multitude d'autres applications Internet comme le commerce électronique.

Tout le monde connaît ou devrait connaître le Web. Pourtant, l'écriture de pages HTML est sans aucun doute moins maîtrisée, et encore moins le protocole HTTP. Le but dans cette partie n'est pas de présenter le protocole HTTP dans ses détails, mais assez cependant pour comprendre le code. Si le lecteur décide de s'étendre sur le sujet, il devra sans doute installer un serveur Web sur sa machine (par exemple Apache, que nous pouvons télécharger à partir du site `http://www.apache.org`), ainsi que Perl, un langage pour créer ses scripts sur le serveur.

Lorsque nous accédons à une page Web avec notre navigateur, nous spécifions une adresse URL de la forme : `http://www.eyrolles.com`.

Nous avons parfois des adresses FTP, comme `ftp://ftp.microsoft.com`, qui nous permettent d'accéder à des sites FTP anonymes afin de télécharger par exemple de la documentation, des graticiels ou de nouveaux drivers.

Ici, `ftp://` ou `http://` spécifient le protocole : FTP pour *File Transfer Protocol* et HTTP pour *HyperText Transfer Protocol*.

Nous devons à présent passer rapidement aux formulaires qui nous sont présentés dans des pages Web afin d'entrer des données. Si nous retournons à la page principale des éditions

Eyrolles, un champ d'entrée pour une recherche rapide nous est proposé. Ce morceau de code HTML se présente ainsi :

```
<form name="form_recherche"
    method="GET" action="/Leol/Ouvrages/resultat_recherche_ouvrages.php3">
<tr>
   <td><input type="text" name="words" size="11" maxlength="30"  ></td>
   <td><input type=Image name=ok src="/images/ok.gif"  border=0 alt=OK></td>
</tr>
</form>
```

Le code HTML `form` est ici pour nous indiquer que nous avons affaire à un formulaire. À l'intérieur de cette « forme », nous avons une série de données intéressante :

- `GET` – Nous indique quelle méthode du protocole est utilisée.
- `Action` – Quel script est exécuté sur le serveur des éditions Eyrolles.
- `Name` – Le nom de la variable, ici `words`, où sera transféré le texte que nous désirons rechercher avec un maximum de 30 caractères.
- L'extension `.php3` nous indique que nous avons affaire à un script écrit en PHP, version 3, un langage récent pour les scripts CGI (*Common Gateway Interface*) des serveurs HTTP. Si nous avions l'extension `.pl`, cela indiquerait que le langage Perl est utilisé. Ce dernier est incontournable pour les développeurs d'applications Internet.

Dans l'exemple que l'auteur a choisi, un site en langue allemande en Suisse, il est possible d'introduire un numéro de téléphone mobile ainsi que le message à envoyer :

```
<form action="../include/sms_mail.asp" method="POST">
  <input type="hidden" name="TargetGroup" value="4">
  <input type="text" size="7" maxlength="7" name="Nummer">
  <input type="text" size="40" maxlength="160" name="Message">
  <input type="submit" name="B1" value="Abschicken">
</form>
```

Ici, nous travaillons avec une autre méthode, le `POST`. L'extension `.asp` nous indique que le serveur HTTP utilise la technologie Active Server Page de Microsoft. Il y a aussi un champ caché (*hidden*), nommé `TargetGroup`, avec lequel il faudra absolument passer avec la valeur 4 au travers du protocole.

Il nous faut à présent répondre à la question que tout le monde doit se poser : mais pourquoi donc faisons-nous cela, alors qu'il est tout à fait possible d'entrer ces données dans une page Web de notre explorateur ? La réponse est simple ! Il y a une foule de conditions où il n'est pas possible d'utiliser le Web. Prenons un exemple : nous devons partir du travail, et notre compilation va durer encore une bonne heure ; ou bien nous attendons un e-mail. Suivant le résultat, demain matin, nous déciderons de dormir un peu plus longtemps ou d'aller faire un footing. Le programme suivant peut être intégré dans un script et retourner sur notre téléphone mobile, par exemple, un « OK » ou un « NOT OK — il y a le feu dans la maison ». Le programme est une application Java et non pas une applet. Cette dernière serait uniquement exécutable à l'intérieur d'un navigateur.

Nous passons à présent au code complet pour le traitement de cette application :

```
import java.net.*;
import java.io.*;
import java.util.Properties;

public class EnvoieSMS {
  private boolean ph = false;
  private boolean pp = false;
  private boolean anum = false;
  private boolean ames = false;
  private String  proxyHost;
  private String  proxyPort;
  private String  numero;
  private String  message;

  public EnvoieSMS(String[] les_param) throws Exception {
    for (int i = 0; i < les_param.length; i++) {
      if (les_param[i].equals("-pH")) {  // proxy host
        i++;
        if (i >= les_param.length) break;
        proxyHost = les_param[i];
        ph = true;
      }

      if (les_param[i].equals("-pP")) {  // proxy port
        i++;
        if (i >= les_param.length) break;
        proxyPort = les_param[i];
        pp = true;
      }

      if (les_param[i].equals("-N")) {   // numéro téléphone mobile
        i++;
        if (i >= les_param.length) break;
        numero = les_param[i];
        anum = true;
      }

      if (les_param[i].equals("-M")) {   // message
        i++;
        if (i >= les_param.length) break;
        message = les_param[i];
        ames = true;
      }
    }

    if (ph & pp) {
      Properties props = System.getProperties();
      props.put("proxySet", "true");
```

```
      props.put("proxyHost", proxyHost);
      props.put("proxyPort", proxyPort);
    }
  }

  public String envoie() throws Exception {
    URL url;
    URLConnection urlConn;
    DataOutputStream printout;
    DataInputStream input;

    if (anum & ames) {

      String urlstr = "http://www.ericsson.ch/include/sms_mail.asp";
      url = new URL(urlstr);

      urlConn = url.openConnection();
      urlConn.setDoInput(true);
      urlConn.setDoOutput(true);
      urlConn.setUseCaches(false);
      urlConn.setRequestProperty("Content-Type", "application/
      ➥x-www-form-urlencoded");
      urlConn.setRequestProperty("Referer", "http://www.ericsson.ch/
      ➥4/sms_online.asp");

      // Envoie l'information
      printout = new DataOutputStream (urlConn.getOutputStream ());

      String contenu = "Number=" + numero + "&Message=" +
                  java.net.URLEncoder.encode(message, "UTF-8") + "&TargetGroup=4";
      printout.writeBytes(contenu);
      printout.flush();
      printout.close();

      // Obtient le résultat
      BufferedReader in =
      new BufferedReader(new InputStreamReader(urlConn.getInputStream()));

      boolean envoi_ok = false;
      String str;
      while (null != ((str = in.readLine())))
      {
        if (str.indexOf("Ihre Message wurde verschickt") != -1) envoi_ok = true;
      }
      in.close ();

      if (envoi_ok) {
        return "Message envoyé si numéro correct";
      }

      return "Message pas envoyé";
```

```
    }

    return "Aucun message présent";
  }

  public static void main(String[] args) throws Exception {
    EnvoieSMS mon_sms = new EnvoieSMS(args);
    System.out.println(mon_sms.envoie());
  }
}
```

La classe `EnvoieSMS` n'a été conçue que pour entrer tous les paramètres sur la console. La ligne de commande :

```
java EnvoieSMS -N1234567 -MDe_xxxx:_Je_suis_malade_aujourd_hui
```

enverra le message `De_xxxx:_Je_suis_malade_aujourd_hui` sur le numéro de téléphone mobile 1234567 situé en Suisse (le code d'accès 079 étant ajouté par le serveur).

Les paramètres `-pPnom`_du_proxy et `-pNnuméro_du_port` seraient des paramètres nécessaires si l'utilisateur se trouvait derrière un pare-feu, c'est-à-dire un intranet. Ce sont les mêmes paramètres que nous retrouvons dans les préférences ou propriétés de notre navigateur Internet.

Le constructeur reçoit tous les arguments du programme, vérifie les paramètres et, éventuellement, ajoute les propriétés du proxy avec la classe `Properties` si nous nous trouvons derrière un pare-feu.

La méthode `envoie()` contrôle préalablement si le message et le numéro existent. Nous aurions pu définir d'autres méthodes si nous avions voulu, par exemple, écrire une petite application Java avec Swing, dans laquelle nous aurions pu ajouter une liste de numéros où le message pourrait être envoyé.

Nous passons à présent à la partie la plus complexe du code. Les adresses URL sont directement dans le code ainsi que les paramètres nécessaires au protocole HTTP. Le plus déroutant est le fait que la connexion doit se faire sur l'adresse URL du serveur HTTP qui va devoir traiter la requête, et ceci avec le `Referer` situé sur la page d'origine. Le `Referer` se trouve être la page source où figure le formulaire HTML présenté à l'utilisateur d'un explorateur Web. En fait, nous simulons un accès, comme si nous venions d'un explorateur. Le `String contenu` contient les trois paramètres nécessaires à la requête qui sera contrôlée et traitée par le serveur Web.

Après l'envoi de la requête, nous lisons la réponse et analysons le résultat afin d'identifier si la transaction a été effectuée jusqu'à ce point. Le message `Ihre Message wurde verschickt` (« Votre message fut envoyé ») indique que le message a été vraisemblablement envoyé, car il semble que le serveur nous retourne une erreur seulement si le numéro n'a pas sept chiffres, alors qu'un numéro invalide semble accepté.

Nous répéterons que ce code est valable uniquement pour ce cas précis et devrait être adapté pour un autre formulaire HTML.

La description des différentes méthodes appliquées à l'objet `urlConn` de la classe `URLConnection` se trouve dans la documentation Java sur le CD-Rom (API). Avec la méthode `setRequestProperty()`, il est possible d'initialiser une propriété qui sera utilisée dans le protocole HTTP. Dans notre exemple nous avons les deux paramètres `Content-Type` et `Referer`, qui sont essentiels. Le dernier sera contrôlé par le script CGI du serveur, et la requête sera rejetée si nous essayons d'y accéder d'une autre page Web, qui est notre formulaire d'entrée. Enfin, la méthode `java.net.URLEncoder.encode()` doit être utilisée pour tout texte transmis. Si nous faisons :

```
System.out.println(java.net.URLEncoder.encode("Salut André", "UTF-8"));
```

nous obtiendrons :

```
Salut+Andr%E9
```

qui correspond au message codé requis par le protocole HTTP.

Programmons le jeu d'Othello

Les règles du jeu

De la même manière qu'aux échecs, le jeu d'Othello est un jeu à deux joueurs qui se joue sur un plateau unicolore de 64 cases, huit sur huit. Les deux joueurs disposent en tout de 64 pions bicolores, noirs d'un côté et blancs de l'autre. Par commodité, chaque joueur a devant lui 32 pions mais ils ne lui appartiennent pas et il doit en donner à son adversaire si celui-ci n'en a plus. Chacun joue un pion à la fois, jusqu'à ce que l'échiquier soit rempli ou qu'aucun des deux joueurs ne puisse continuer.

Au départ, quatre pions sont placés au centre, deux blancs et deux noirs, de manière à ce qu'il y ait un blanc et un noir sur chaque ligne horizontale. Si le joueur blanc commence, il doit poser un pion blanc sur une case vide adjacente à un pion noir. Cependant, cette condition n'est pas suffisante : en posant son pion, il doit impérativement encadrer un ou plusieurs pions adverses entre le pion qu'il pose et un pion blanc, déjà placé sur le plateau. Il retourne alors de sa couleur le ou les pions qu'il vient d'encadrer.

Le gagnant est celui qui obtient le plus grand nombre de pions de sa couleur. Il est facile de comprendre que les quatre bords et surtout les quatre coins sont des lieux stratégiques. Ces derniers notamment jouent un rôle essentiel en fin de partie, quand l'adversaire peut aussi se trouver bloqué et doit laisser passer son tour.

C'est un excellent jeu à programmer mais très complexe pour ce qui est de l'analyse et de la conception.

La conception

Nous avons vu au chapitre 5, lorsque nous avons traité les tableaux multidimensionnels, comment initialiser notre jeu :

```
int othello[10][10]
```

avec les bords de chaque côté afin de permettre un contrôle plus simple des limites du terrain. Si nous avions travaillé avec un tableau de huit sur huit, nous aurions dû continuellement tester des `< 0` ou `> 7`. Notre approche est certainement raisonnable.

La deuxième partie du puzzle se trouve dans l'exercice 1 du chapitre 5. Ce nouveau tableau, ici en C+ :

```
const int Othello2::positions[8][2] = {
      {0, 1}  , {1, 1}   , {1, 0}  , {1, -1} ,
      {0, -1} , {-1, -1} , {-1, 0} , {-1, 1}
    };
```

nous définit les huit directions possibles à vérifier. À présent nous allons programmer un morceau de code qui va utiliser ce tableau pour vérifier qu'une position est jouable.

Le jeu d'Othello en C++

```
// othello2.cpp
#include <iostream>

using namespace std;

class Othello2 {
  private:
    static const int dim = 10;         // 8 ¥ 8 plus les bords
    static const int case_vide  =  0;
    static const int case_bord  = -1;

    int othello[dim][dim];      // le jeu
    static const int positions[8][2];

  public:
    static const int pion_blanc =  1;
    static const int pion_noir  =  2;

    Othello2();
    void dessine();
    bool jouable(const int pion, const int posx, const int posy);
};

const int Othello2::positions[8][2] = {
      {0, 1}  , {1, 1}   , {1, 0}  , {1, -1} ,
      {0, -1} , {-1, -1} , {-1, 0} , {-1, 1}
    };

Othello2::Othello2()
{
 int i = 0;  // position horizontale
 int j = 0;  // position verticale
```

```
 for (i = 0; i < dim; i++) {  // les bords
   othello[i][0]     = case_bord;
   othello[i][dim-1] = case_bord;
   othello[0][i]     = case_bord;
   othello[dim-1][i] = case_bord;
 }

  for (j = 1; j < dim-1; j++) {  // intérieur vide
    for (i = 1; i < dim-1; i++) {
      othello[i][j] = case_vide;
    }
 }

 othello[4][5] = pion_blanc;
 othello[5][4] = pion_blanc;
 othello[6][4] = pion_blanc;
 othello[4][4] = pion_noir;
 othello[5][5] = pion_noir;
}

void Othello2::dessine()
{
 int i = 1;  // position horizontale
 int j = 1;  // position verticale

 cout << " 12345678" << endl;

 for (j = 1; j < dim -1; j++) {
   cout << j;
   for (i = 1; i < dim - 1; i++) {
     switch(othello[i][j]) {
       case pion_noir:
         cout << "N";
         break;
       case pion_blanc:
         cout << "B";
         break;
       default:
         cout << " ";
         break;
     }
   }
   cout << endl;
 }
}

bool Othello2::jouable(const int pion, const int posx, const int posy) {
 if (othello[posx][posy] != case_vide)  return false;

 int pion_inverse = pion_blanc;
 if (pion == pion_blanc) pion_inverse = pion_noir;
```

```
 int mult = 2;
 int piont;
 for (int dir = 0; dir < 8; dir++) {
    if (othello[posx + positions[dir][0]][posy + positions[dir][1]] == pion_inverse) {
     for (;;) {
       piont = othello[posx + (mult * positions[dir][0])][posy
                               + (mult * positions[dir][1])];
       if (piont == pion)  return true;
       if (piont != pion_inverse) break;
       mult++;
     }
   }
 }

 return false;
}


int main()
{
 Othello2 lejeu;
 int x, y;
 for (;;) {
   lejeu.dessine();

   cout << "Noir joue en x: ";
   cin  >> x;
   if (x == 0) break;
   cout << "y: ";
   cin  >> y;

   bool resultat = lejeu.jouable(Othello2::pion_noir, x, y);
   cout << "Jouable en " << x << ":" << y << " = " << resultat << endl;
 }
}
```

Le jeu d'Othello en Java

```
import java.io.*;

public class Othello2 {
  static final int dim = 10;

  private int[][] othello = new int[dim][dim];
  private final int[][] positions = {
      {0, 1}  , {1, 1}   , {1, 0}  , {1, -1} ,
      {0, -1} , {-1, -1} , {-1, 0} , {-1, 1}
    };
```

```
private final int case_vide  =  0;
private final int case_bord  = -1;
public  final int pion_blanc =  1;
public  final int pion_noir  =  2;


public Othello2() {
  int i = 0;  // position horizontale
  int j = 0;  // position verticale

  for (i = 0; i < dim; i++) {  // les bords
    othello[i][0]     = case_vide;
    othello[i][dim-1] = case_vide;
    othello[0][i]     = case_vide;
    othello[dim-1][i] = case_vide;
  }

  for (j = 1; j < dim-1; j++) {  // intérieur vide
    for (i = 1; i < dim-1; i++) {
      othello[i][j] = case_vide;
    }
  }

  othello[4][5] = pion_blanc;
  othello[5][4] = pion_blanc;
  othello[6][4] = pion_blanc;
  othello[4][4] = pion_noir;
  othello[5][5] = pion_noir;
}

public void dessine() {
  int i = 1;  // position horizontale
  int j = 1;  // position verticale

  System.out.println(" 12345678");

  for (j = 1; j < dim -1; j++) {
    System.out.print(j);
    for (i = 1; i < dim -1; i++) {
      switch(othello[i][j]) {
        case pion_noir:
          System.out.print("N");
          break;
        case pion_blanc:
          System.out.print("B");
          break;
        default:
          System.out.print(" ");
          break;
      }
    }
```

```
    System.out.println();
  }
}


public boolean jouable(int pion, int posx, int posy) {
 if (othello[posx][posy] != case_vide)  return false;

 int pion_inverse = pion_blanc;
 if (pion == pion_blanc) pion_inverse = pion_noir;

 int mult = 2;
 int piont;
 for (int dir = 0; dir < 8; dir++) {
   if (othello[posx + positions[dir][0]][posy + positions[dir][1]] == pion_inverse)

     for (;;) {
       piont = othello[posx +
         (mult * positions[dir][0])][posy + (mult * positions[dir][1])];
       if (piont == pion)  return true;
       if (piont != pion_inverse) break;
       mult++;
     }
   }
 }

  return false;
}

public static void main(String[] args) {
  Othello2 monjeu = new Othello2();

  int x = 0;
  int y = 0;
  boolean resultat = true;

  for (;;) {
    monjeu.dessine();

    System.out.print("Noir joue en x: ");
    try {
      BufferedReader stdin = new BufferedReader(new InputStreamReader(System.in));

      x = Integer.parseInt(stdin.readLine());
      if (x == 0) break;
      System.out.print("y: ");
      y = Integer.parseInt(stdin.readLine());

      resultat = monjeu.jouable(monjeu.pion_noir, x, y);
    }
    catch (IOException ioe) {
```

```
        System.err.println(ioe);
      }
      System.out.println("Jouable en "+ x + ":" + y + " = " + resultat);;
    }
  }
}
```

Par rapport à la première version `othello1.cpp` et `Othello1.java` du chapitre 5, nous voyons déjà quelques améliorations avec la définition de constantes. En C++, nous aurions pu définir des `typedef`, mais nous les avons volontairement laissés de côté. Java ne possède pas d'énumérations (voir chapitre 2).

De plus, nous avons ajouté un pion blanc supplémentaire afin de contrôler notre code avec des combinaisons un peu plus étendues. La boucle dans le `main()` ne se terminera que lorsque nous donnerons un 0 comme entrée. Cela nous permet de contrôler toute une série de positions possibles. La position 7 (x) 4 (y) est bien jouable pour les noirs. Exécutons à présent le programme :

```
 12345678
1
2
3
4   NBB
5   BN
6
7
8
Noir joue en x: 7
y: 4
dir: 0
dir: 1
dir: 2
dir: 3
dir: 4
dir: 5
dir: 6
Jouable en 7:4 = 1
Jouable en 7:4 = true     // en Java
```

Il est très important de passer le temps nécessaire à tester une fonction telle que `jouable()`, car dès qu'elle est vérifiée, elle ne sera en principe plus touchée. Nous verrons ci-après que sa logique devra en fait être réutilisée lorsqu'il faudra retourner les adversaires sur l'échiquier et non pas sortir de la boucle lorsque nous avons trouvé une position jouable dans une des directions.

Le premier test est évident : il faut que la position soit vide. Ensuite, nous avons la boucle autour des positions dans les huit directions où le pion doit être un adversaire :

```
for (int dir = 0; dir < 8; dir++) {
  if (othello[posx + positions[dir][0]][posy + positions[dir][1]] == pion_inverse) {
```

Si nous découvrons qu'un adversaire est un voisin, nous devons nous diriger dans la direction de la diagonale avec :

```
piont = othello[posx + (mult * positions[dir][0])]
                [posy + (mult * positions[dir][1])];
```

Le facteur `mult` (2, 3, 4, ...) s'applique sur le tableau des directions et va nous permettre d'identifier si nous avons un ou plusieurs adversaires coincés entre deux de nos pions.

L'instruction :

```
if (piont != pion_inverse) break;
```

contrôle en fait deux cas possibles, une case vide ou un bord. Dans ces deux cas, il faut continuer avec la prochaine direction, car la position pour cette direction n'est pas jouable.

Lors du développement de ce type de programme, il y a plusieurs méthodes pour résoudre un problème. Nous pouvons par exemple ajouter des `cout` ou `System.out.println` à l'intérieur des boucles et sortir les valeurs des index, de `mult` ou encore des valeurs et des positions courantes. Une autre alternative pourrait être le traceur que nous avons conçu au chapitre 16 ! Avec un débogueur, par exemple NetBeans (voir annexe E), nous pourrions bien évidemment accélérer le processus.

Nous avons vu aussi, au chapitre 9, comment recharger et sauvegarder sur le disque une partie momentanément interrompue. Celle-ci pourrait être en fait réutilisée à des fins de test lors de l'implémentation de la logique pour déterminer les meilleures positions à jouer ou de nouvelles stratégies. Mais, avant de terminer cette partie, nous allons présenter rapidement les fonctions qu'il nous faudrait sans doute considérer pour aboutir à la construction du programme complet. Nous n'allons pas nous occuper de la partie graphique, qui peut être conçue totalement séparément, sans nous occuper de la partie intelligente du jeu. Voici donc une première liste, afin de préparer un programmeur potentiel pour son travail :

- Sortir à l'écran à des fins de test la partie courante. La méthode `dessine()` ci-dessus peut être adaptée.
- Permettre au joueur d'entrer la position à jouer en mode DOS.
- Lorsque la position a été vérifiée comme jouable, retourner tous les adversaires en satisfaisant les conditions.
- Tester dans l'échiquier entier toutes les positions jouables pour identifier si un joueur doit passer son tour et si la partie est terminée.
- Introduire un système qui limite les risques de donner des positions à l'adversaire dans les bords ou les coins.
- Permettre d'introduire de nouvelles stratégies et de faire jouer le programme contre lui-même.

Suggestions d'autres applications

Nous allons donner ici deux sujets d'applications que le lecteur pourrait développer lui-même. Ces applications sont intéressantes dans le sens où elles nécessitent un certain nombre de techniques et de fonctions de bibliothèques que nous avons utilisées dans cet ouvrage.

Archiver des fichiers

Le point de départ serait le chapitre 9, dans lequel nous avons appris comment accéder aux répertoires du disque, aux noms des fichiers et à leurs propriétés. Les deux langages se prêtent très bien à cette application, malgré les quelques difficultés en C++, dont nous aimerions dire, comme toujours, qu'elles proviennent de la portabilité de ces fonctions d'accès aux ressources du système.

Nous pourrions imaginer une base de données comportant différentes versions du même document avec leurs dates de création (ou bien la source de nos exercices et exemples, en Java et en C++). En Java, nous pourrions même compresser ces versions, puisque nous avons une bibliothèque à disposition (jar).

Télécharger un site Web entier

Ce travail pourrait certainement constituer un projet de fin d'études en informatique. Le langage serait évidemment Java avec son interface Internet naturelle. Télécharger une page Web n'est pas un problème, mais analyser son contenu avec ses codes HTML est beaucoup moins évident. Une page Web installée sur un disque local contiendra vraisemblablement des liens extérieurs. Nous pourrions alors télécharger tous les documents disponibles sur le même répertoire et référencés dans ces pages.

Il y a souvent des images dans ces pages Web, qui sont généralement disposées dans des sous-répertoires, pour des raisons structurelles. Après téléchargement de ces fichiers ou documents supplémentaires, les références URL (`HREF`) pourraient être modifiées, afin que les liens correspondent à des accès relatifs sur le disque. Des références à de gros documents, comme des fichiers de type `.exe`, `.zip` ou encore `.pdf`, pourraient être conservées, afin d'éviter de gros transferts inutiles. Une référence relative dans une page Web pourrait être transformée en une référence absolue sur un site `http://`.

Résumé

Écrire de temps à autre des applications d'une certaine consistance va permettre aux programmeurs de rafraîchir et d'étendre leurs connaissances. Dans ce contexte, le langage Java, avec sa très vaste API, ses nombreux produits associés livrés par Sun Microsystems, sa simplicité d'utilisation pour Internet et sa portabilité, est sans aucun doute plus simple et motivant pour les débutants. Le langage C++, beaucoup plus complexe, apportera sans doute beaucoup plus en ce qui concerne les techniques de programmation.

Incontestablement, les programmeurs professionnels devraient maîtriser ces deux langages, qui font et vont faire partie encore plus dans le futur des langages essentiels du monde informatique.

23

L'étape suivante : le langage C# de Microsoft

Que vient donc faire le C# dans cet ouvrage ?

Lorsque l'auteur a entrepris l'étude du langage C# (communément désigné *C Sharp*) et de sa programmation, il a été surpris d'y découvrir de nombreuses similitudes avec les langages C++ et Java. Comme C# est très proche de ces deux langages, il a pensé qu'il serait intéressant non seulement de le présenter brièvement, mais aussi de fournir avec ce livre tous les outils et documents qui permettraient aux lecteurs intéressés, avec un bon bagage Java et C++ (ce qui devrait être le cas en cette fin d'ouvrage), de programmer rapidement dans ce nouveau langage.

En effet, selon nous, un programmeur avec une expérience C++ ou Java n'aura aucune difficulté pour programmer avec ce nouveau langage.

Le langage C# va certainement prendre de plus en plus d'importance avec l'avènement du Framework (plate-forme) .NET de Microsoft sur les systèmes d'exploitation Windows et aussi prochainement sur d'autres systèmes comme Unix ou Linux.

En outre, ces fonctionnalités ajoutées au langage C# vont sans doute donner l'opportunité aux programmeurs C++ et Java de comprendre encore mieux ces deux langages et d'améliorer leur code et leur design. Il ne faut pas oublier que le langage C# est le plus récent, donc en principe le plus solide au niveau de la conception. En revanche, si les programmeurs ont utilisé des méthodes et des techniques particulières qui ont fait leurs preuves dans les langages C++ et Java mais qui ne s'appliquent pas directement au langage C#, il n'est pas interdit de les adapter, au lieu de simplement les oublier. Nous citerons deux exemples comme la documentation avec Javadoc et les modules de test avec une entrée `main()` pour chaque classe publique.

Dans ce chapitre, nous donnerons un certain nombre d'exemples simples mais représentatifs et soulignerons quelques différences majeures avec les langages Java et C++.

Un peu d'histoire

L'historique des langages C++, Java et C# est essentielle pour comprendre les différences entre ces langages et de leurs évolutions respectives. Le tableau 23-1 donnera au lecteur un aperçu des dates importantes de l'histoire de ces trois langages.

Tableau 23-1 Quelques dates historiques pour nos trois langages

C C++	1971	Les premiers pas du langage C. Le langage C++ est tellement lié au langage C et à son histoire, que nous devons aussi indiquer deux dates majeures de référence.
	1978	Apparition du langage C, le standard défini par K&R (Kernighan et Ritchie).
	1983	Apparition du langage C++, une extension du langage C développée par Bjarne Stroustrup.
	1998	Apparition du standard C++ d'ANSI et ISO.
	2000	Octobre : apparition d'Internet C++, une tentative parmi d'autres.
	2003	ISO/IEC 14882 : la norme ISO du C++.
Java	1991	Premiers pas du langage Java développé par Sun Microsystems.
	1995	Apparition de Java 1.
	1998	Apparition de Java 2 – la version 1.2.
	2000	Nouvelle version de Java 2 – la version 1.3.
	2002	Nouvelle version de Java 2 – la version 1.4.
	2003	Apparition de Java 5 – la version 1.5.
	2006	Apparition de Java 6 – la version 1.6.
C#	2000	Juin : apparition de la première version qui inclut le Framework .NET.
	2001	Décembre : apparition d'ECMA (le standard).
	2005	Version 3.0 de la spécification du C#.
	2007	Version 3.5 du Framework .NET utilisé pour les langages Visual Basic, Visual C++ et C#.

Depuis 2001, plusieurs versions de Java et de C# sont apparues. Peu de gros changements ont été nécessaires, car les spécifications des langages étaient déjà mûres et solides. En 2008, nous sommes à la version 1.6 pour Java et 3.5 pour le Framework .NET de Microsoft. Pour le C++, la plupart des compilateurs ont été mis à jour afin de prendre en compte certaines dépréciations dans la bibliothèque Standard C++, traitées dans cet ouvrage.

Donner une date précise pour la sortie de chacune des versions ne fait pas vraiment de sens, car des versions bêta sont disponibles avant les sorties officielles et ensuite des mises à jour et corrections apparaissent régulièrement. Pour le programmeur, il suffit d'indiquer la version exacte, par exemple 1.6.6 pour le Java utilisé dans cet ouvrage.

C++, Java et C# : les différences majeures

Quelles sont les différences majeures entre ces trois langages orientés objet ? Est-il possible de les comparer sans équivoque ou préjugé ? Évidemment non ! Mais nous allons tout de même analyser ce qui représente, selon notre point de vue, les principales différences :

- Le langage C++ est un langage orienté objet, difficile et seulement utilisable par des « professionnels ». Son code binaire est performant et directement exécuté par le processeur de la machine. Il faut recompiler le code, qui en outre n'est pas nécessairement portable, si on veut l'exécuter sur un autre type de machine. De nombreuses bibliothèques de classes C++ existent, mais sont souvent dépendantes du système d'exploitation.
- Le langage Java hérite du C++ mais est beaucoup plus facile à assimiler et à programmer. Il permet d'éviter de grosses fautes de programmation difficiles à corriger. Le code compilé peut s'exécuter sur tout système d'exploitation où une machine virtuelle, qui interprète ce code, a été installée. Il est évidemment moins performant que le C++ (mais est-ce un réel problème ?) et possède une bibliothèque de classes très riche disponible sur de nombreuses plates-formes (Windows, Linux ou d'autres Unix).
- Le langage C# de Microsoft est le seul de ces trois langages à être développé par un fabricant de systèmes d'exploitation. Ce langage qui hérite du C++ (affirmation de Microsoft) a cependant une terrible ressemblance avec Java. Il ne serait certainement pas ce qu'il est aujourd'hui si Java n'avait pas existé.

Ainsi, les langages Java ou C# sont certainement ceux que nous recommanderions comme premier langage de programmation. Si un choix doit se faire entre le C++ et Java, ou entre C++ et C#, nous conseillons vivement d'opter pour les seconds nommés.

Hello world en C#

Compiler et exécuter des programmes peut se faire de la même manière que les exemples Java et C++ de cet ouvrage. Nous avons choisi pour le langage C# une intégration dans l'éditeur Crimson. L'installation du SDK de Framework de .NET qui permet de compiler des fichiers source C#, la configuration de Crimson et l'accès à la documentation des outils et des bibliothèques de ce même Framework sont décrites dans l'annexe C. Avant de pouvoir compiler et exécuter les exemples ci-après, il faut installer correctement ces outils.

Mais revenons à notre classe Java du chapitre 1 :

```
public class Hello {
   public static void main(String[] args) {
      System.out.println("Hello world en Java: " + new java.util.Date());
   }
}
```

C'est une version condensée, tout à fait correcte, puisque l'`import` n'est en fait pas nécessaire et que le `new` nous retournera un objet sur lequel une méthode `toString()` est disponible.

Le résultat devrait être similaire à celui-ci :

```
Hello world en Java: Sat Mar 01 16:35:38 CET 2003
```

Rappelons les raccourcis clavier que nous avons définis dans Crimson (voir annexes C et E) :

Ctrl+F1	Compilation de fichier Java. Par exemple, `javac Hello.java`. Résultat : fichier `Hello.class`.
Ctrl+F2	Exécution de classe Java. Par exemple, `java Hello` Le programme `java` cherchera implicitement `Hello.class`.
Ctrl+F8	Compilation C#. Par exemple, `csc HelloCs.cs`. Résultat : exécutable `HelloCs.exe`.
Ctrl+F6	Exécution de fichier exécutable `.exe`. Par exemple, `HelloCs.exe`. `HelloCs.exe` ne pourra pas être exécuté sur un PC qui n'a pas l'environnement adéquat (similaire au JRE de Java) : voir la section « Installation du SDK 3.5 » de l'annexe D.

Passons donc directement à la version C# (fichier `HelloCs.cs`) avant d'en analyser les différences :

```
class HelloCs {
  public static void Main() {
    System.Console.WriteLine("Hello world en C#: " + System.DateTime.Now);
  }
}
```

Nous avons rajouté un `Cs` dans les noms de classes et de fichiers pour les différencier du C++. En effet, si nous compilons un fichier `Hello.cpp` en C++ ou un `Hello.cs` en C#, nous pourrons nous retrouver avec le même `Hello.exe`.

Si nous exécutons ce code compilé, nous obtiendrons par exemple :

```
Hello world en C#: 01.03.2003 16:36:01
```

Au premier coup d'œil, nous ne voyons que quelques différences mineures. Avec Crimson (menu Windows/Tile Horizontally), nous pouvons positionner nos deux fichiers de cette manière pour les comparer, voir figure 23-1.

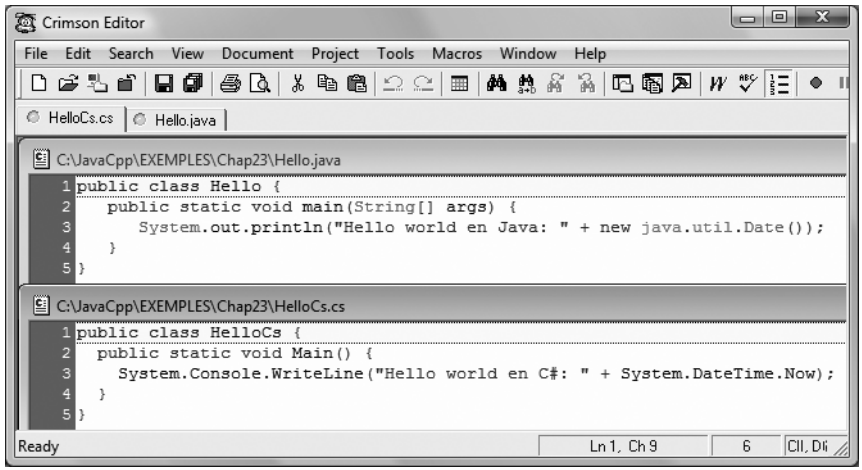

Figure 23-1

Nos deux classes Hello.java et HelloCs

Voici une première liste de différences et des explications supplémentaires absolument indispensables :

- En C#, la classe `HelloCs` n'a pas besoin d'être `public` bien que cela soit possible.
- Il n'est pas nécessaire que la classe `HelloCs` soit dans un fichier `HelloCs.cs` comme en Java. Nous pouvons en fait nommer cette classe `Hello` et garder le nom du fichier `HelloCs.cs`. Ce dernier est en réalité le nom par défaut passé au compilateur qui générera l'exécutable `HelloCs.exe`.

Il est possible de donner un nom spécifique au fichier exécutable final avec l'option `out` :

```
C:/Windows/Microsoft.NET/Framework/v3.5/csc.exe /out:Hello.exe HelloCs.cs
```

De cette manière, nous pouvons effectivement définir une classe `Hello`, la déposer dans un fichier source `HelloCs.cs` et avoir un exécutable nommé `Hello.exe`. Cependant, nous recommandons cette manière de faire, forcée en Java, mais combien judicieuse, lorsqu'il s'agit de s'y retrouver dans son code et ses classes. Voici encore d'autres différences et commentaires :

- En Java, il est interdit de définir plusieurs classes `public` dans la même source. En C#, cela est possible, et, à nouveau, nous recommandons la manière de faire Java.
- Le `Main()` de C# commence par une majuscule, comme d'ailleurs toutes les méthodes (membres) d'une classe dans le langage C#. Lorsqu'on examine rapidement du code source, c'est en général à cette particularité que nous voyons si ce code est écrit en Java ou en C#.

- Le `String[] args` en Java, la signature du `main()`, n'est pas obligatoire en C#.
- Le `System` est différent. En Java, c'est une classe, et en C#, un espace de noms. Nous le comprendrons mieux dans l'exemple qui va suivre.
- Dans les deux exemples, il y a quatre classes, `System` et `Date` en Java et `Console` et `DateTime` en C#, et les méthodes sont évidemment différentes, par exemple `println()` avec un `p` minuscule et `WriteLine()` en C# avec un `W` majuscule. Les conventions sont donc différentes bien que nous retrouvions la même convention qu'en Java pour la majuscule `L` de `Line`.

Écrivons à présent une variante de la classe `HelloCs`, c'est-à-dire `HelloCs1` :

```
using System;

public class HelloCs1 {
  public static void Main() {
    Console.Out.WriteLine("Hello1 world en C#: " + DateTime.Now);
  }
}
```

Nous sommes beaucoup plus proches de l'exemple Java. Le `using System` permet au compilateur de définir l'espace de noms où le compilateur C# pourra retrouver les classes `Console` et `DateTime`. En Java, les classes sont retrouvées implicitement dans le cas du package `java.Util`, nous aurions effectivement pu écrire en début de code un :

```
import java.Util.*;
```

qui est presque équivalent à un espace de noms en C#.

Dans l'exemple `HelloCs`, nous avions `Console.WriteLine()`, c'est-à-dire la méthode `WriteLine()` de la classe `Console`. Comme en Java, c'est une méthode statique qui n'a pas besoin d'un objet instancié. En examinant la documentation, nous découvrirons que `WriteLine()` s'applique à la sortie standard (le `out` de Java, le `stdout` de C++). Le `Out` de C# (`Error` pour l'équivalent `err` en Java) est une propriété de la classe `Console` qui va nous retourner le flux de sortie au travers de la classe `TextWriter`. Cette dernière possède également une méthode `WriteLine()`, comme la classe `Console`.

Le terme « propriété » en C# est presque équivalent au terme « attribut » pour les classes en Java. La documentation du Framework du SDK de .NET va évidemment nous aider à comprendre tous les détails, ici les propriétés et méthodes, toutes statiques, de la classe `Console` dans l'espace de noms `System` (voir figure 23-2).

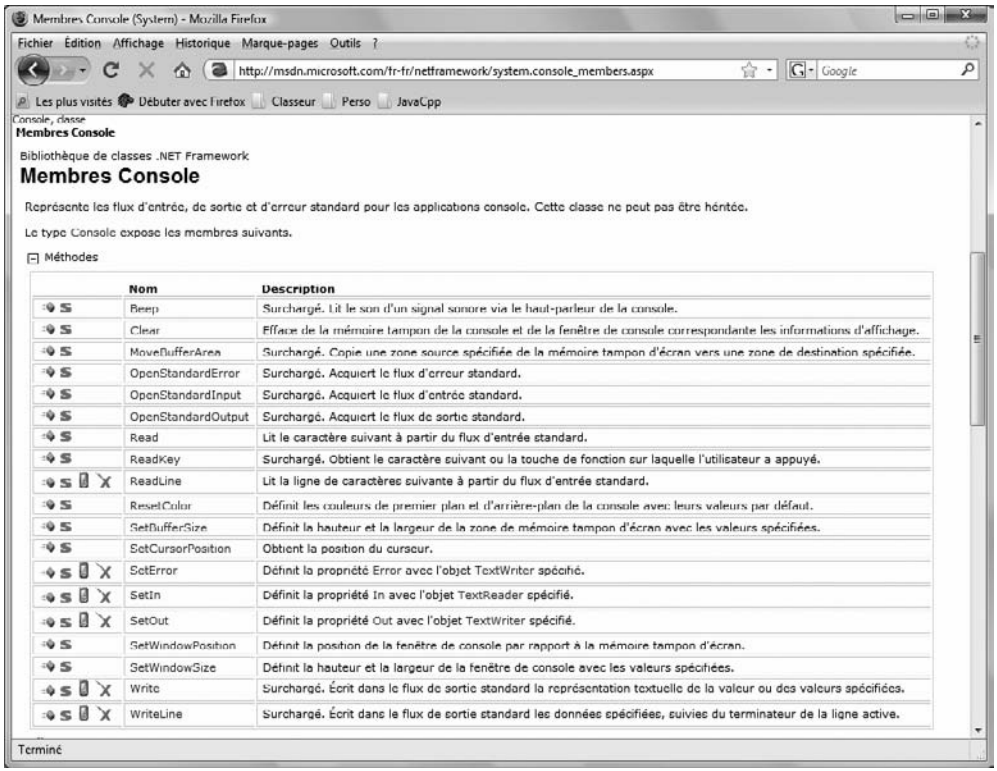

Figure 23-2

Les méthodes de la classe Console

Le lien Web à cette page se trouve sur : `http://msdn.microsoft.com/fr-fr/netframework/system.console_members.aspx`

Il nous faut à présent passer à la classe `DateTime` et sa propriété `Now` qui n'est pas ici une méthode, car elle ne possède pas de `()`. Nous pouvons alors remarquer que la propriété `Now` n'a pas une bonne précision et est au mieux de 10 millisecondes.

- `DateTime` est une structure en C# alors que `Date` est une classe Java. Dans la documentation, il ne faut pas rechercher `DateTime` dans le tableau des classes de l'espace de noms `System`, mais plus bas, dans le tableau des structures. Nous reviendrons très rapidement sur les structures, essentielles en C#, plus loin dans ce chapitre.
- Il n'y a qu'un seul `Main()` possible en C#, comme en C++ d'ailleurs avec `main()`.

Si nous compilons ensemble les deux classes ci-dessus de cette manière :

```
C:/Windows/Microsoft.NET/Framework/v3.5/csc.exe /out:Test.exe HelloCs.cs HelloCs1.cs
```

nous aurons plusieurs problèmes dont le premier viendra des deux entrées `Main()`. Le système d'exploitation doit savoir laquelle des deux entrées est valable. Ce problème

n'existe pas en Java, puisque nous fonctionnons sur la base d'une classe unique et publique avec la restriction des noms de fichiers que nous connaissons.

Les Makefile avec C#

L'exemple qui suit nous offre un exemple de `Makefile` adapté pour compiler des fichiers C++, Java et C#, ici trois fichiers source `hello.cpp`, `Hello.java` et `HelloCs.cs` :

```
JDK = "C:/Program Files/Java/jdk1.6.0_06/"
NET = "C:/Windows/Microsoft.NET/Framework/v3.5"

all:     cpp java csharp

cpp:     hello.exe
java:    Hello.class
csharp:  HelloCs.exe

hello.exe:   hello.o
             g++ -o hello.exe hello.o

hello.o:     hello.cpp
             g++ -c hello.cpp

Hello.class: Hello.java
             $(JDK)/bin/javac Hello.java

HelloCs.exe: HelloCs.cs
             $(NET)/csc.exe HelloCs.cs
```

Rappelons qu'il faut absolument utiliser des tabulateurs et non des espaces (excepté pour les deux premières instructions). Nous avons choisi ici des paramètres pour sélectionner les chemins d'accès des compilateurs pour les langages Java et C##. Dans cet exemple, nous utilisons les versions 1.6 de Java et 3.5 de .NET.

Si nous compilons des sources C# pour créer des bibliothèques `dll`, il faut évidemment ajouter les paramètres appropriés au compilateur `csc.exe`.

Espace de noms

Dans cet ouvrage, nous n'avons pas développé de grosses applications où nous aurions certainement dû construire des bibliothèques `dll` en C++ ou des paquets archivés dans des fichiers `jar` en Java.

Nous avons donc pris l'habitude d'écrire simplement nos quelques fichiers dans le même répertoire, tout à fait suffisant pour notre apprentissage.

Mais que se passerait-il maintenant si nous voulions compiler les deux fichiers suivants de cette manière ?

```
csc /out:TestBonjour.exe TestBonjour.cs Bonjour.cs
```

Notre exécutable `TestBonjour.exe` est composé de deux parties compilées simultanément.

Nous imaginons que le module `TestBonjour` possède l'entrée `Main()` et appelle la classe `Bonjour` pour nous retourner des salutations. Nous allons simplement montrer une manière de faire avec deux fichiers `TestBonjour.cs` et `Bonjour.cs` présents dans le même répertoire :

```
// TestBonjour.cs
using System;

public class TestBonjour {
  public static void Main() {
    Console.Out.WriteLine(MonEspace.Bonjour.UnBonjour());
  }
}
```

La nouveauté ici est `MonEspace`, un espace de noms. Il nous indique que la classe `Bonjour` se trouve dans cet espace et doit se définir de cette manière :

```
// Bonjour.cs
namespace MonEspace {

 public class Bonjour {
   public static string UnBonjour() {
     return "Bonjour";
   }
 }
}
```

Nous ne pouvons plus compiler ces fichiers indépendamment dans Crimson et il nous faut définir une autre procédure avec un fichier `make` que nous allons construire comme ceci :

```
#TestBonjour.mk

NET = C:/WINDOWS/Microsoft.NET/Framework/v1.0.3705

all: TestBonjour.exe

TestBonjour.exe:  Bonjour.cs TestBonjour.cs
                  $(NET)/csc.exe /out:TestBonjour.exe Bonjour.cs TestBonjour.cs
```

Nous allons donc définir à présent un fichier `.mk` pour chaque groupe de fichiers C# à compiler ensemble. Nous avons choisi une extension `.mk` pour notre fichier `make` qui se nomme `TestBonjour.mk` et non `Makefile`. Cette procédure est possible, car le `make -f` du menu de Crimson (Ctrl+F4) accepte en fait n'importe quel nom de fichier.

Pour compiler notre programme, nous devons éditer notre code, c'est-à-dire les fichiers `Bonjour.cs` et `TestBonjour.cs`, et ensuite le sauver avant de passer à la fenêtre du fichier `Test-Bonjour.mk` pour exécuter un `make` avec Ctrl+F4.

En cas d'erreur dans les fichiers `Bonjour.cs` ou `TestBonjour.cs`, il faudra nous positionner manuellement dans la fenêtre et la ligne du fichier correspondant, où se trouve la faute.

Pour exécuter le programme depuis Crimson (Ctrl+F6), il n'est pas nécessaire de se déplacer dans la fenêtre du fichier `TestBonjour.cs`, car nous avons choisi justement un nom de fichier `.mk` qui est le même que le fichier `.exe` exécutable généré.

Cela peut sembler un peu compliqué, bien qu'il soit toujours possible, pour de petits programmes pour lesquels nous n'aimerions pas réutiliser nos classes, d'adopter une méthode simple qui consiste, comme en C++, à tout déposer dans le même fichier source :

```
// TestBonjour1.cs
using System;

public class TestBonjour {
  public static void Main() {
    Console.Out.WriteLine(Bonjour.UnAutreBonjour());
  }

 public class Bonjour {
   public static string UnAutreBonjour() {
     return "Un autre bonjour";
   }
 }
}
```

Les structures en C#

Le pas suivant est un plongeon brutal dans les profondeurs du langage C#. Ceux qui maîtrisent bien le langage Java devraient trouver l'approche de l'auteur assez subtile et intéressante. Les structures sont d'ailleurs une des clés de la compréhension de ce langage.

Mais reprenons tout d'abord un cas de figure traditionnel que nous avons déjà traité avec les langages C++ et Java.

Nous voulons concaténer une chaîne de caractères et un nombre pour construire une nouvelle chaîne de caractères correspondant à une identification. C'est un problème classique en programmation. D'un côté nous avons un nom de famille et de l'autre son année de naissance. Nous désirons obtenir un code d'identification, c'est-à-dire :

```
code = nom + annee;
```

Nous allons tout de suite présenter deux solutions possibles en Java et C# dont nous remarquerons très rapidement les similitudes. Tout d'abord en Java :

```
public class Concat {
 public static void main(String[] args) {
   String nom   = "Dupond";
   int    annee = 1948;

   String code  = nom + annee;
```

```
    System.out.println(code + " de longueur" + code.length());
  }
}
```

Nous savons déjà que le `int` en Java est de type primitif codé sur 4 octets (32 bits) dont le premier bit indique s'il est négatif. Nous savons aussi que le compilateur possède le support nécessaire pour l'opérateur + sur les variables de type primitif, sinon il utilisera la méthode `toString()` pour des objets (comme pour la classe `Date` dans notre premier exemple `Hello.java`).

Comme en Java, le nom des variables commence par une minuscule. Nous allons alors découvrir que nous pouvons pratiquement copier et coller le code précédent et reprendre certaines constructions et méthodes de l'exemple `HelloCs.cs` pour pouvoir finaliser la version C# :

```
using System;

public class ConcatCs {
 public static void Main() {
   string nom   = "Dupond";
   int    annee = 1948;

   String code  = nom + annee;

   Console.Out.WriteLine(code + " de longueur " + code.Length);
 }
}
```

Le résultat est alors exactement le même pour les versions Java et C# :

```
Dupond1948 de longueur 10
```

Mais revenons au code C#. Ici, nous pouvons utiliser `string` ou `String`, cela reste la même chose. Si nous ôtions l'instruction `using`, nous aurions un problème, car les classes `String` et `Console` sont dans l'espace de noms `System`. Voici une possibilité de code correct :

```
public class ConcatCs1 {
 public static void Main() {
   string nom   = "Dupond";
   int    annee = 1948;

   System.String code  = nom + annee;

   System.Console.Out.WriteLine(code + " de longueur " + code.Length);
 }
}
```

Le `string` est en fait un synonyme reconnu par le compilateur pour la classe `System.String`, un objet de type référence équivalent à une instance de classe `String` en Java. En Java, nous avions `length()`, une méthode de la classe `String`. En C#, c'est une propriété avec un `L` majuscule : `Length`.

Mais qu'en est-il du type `int` ? C'est ici que nous allons enfin comprendre le titre de cette section : « Les structures en C# ». Ce n'est pas un type primitif comme en Java mais un type valeur (les objets, les instances de classes, sont de type référence). Bien que les types primitif ou valeur soient pratiquement équivalents, le `int` de C# est en réalité un synonyme, comme pour `string`. Le code C# suivant est en fait équivalent :

```
public class ConcatCs2 {
 public static void Main() {
   System.String nom   = "Dupond";
   System.Int32  annee = 1948;

   System.String code  = nom + annee;

   System.Console.Out.WriteLine(code + " de longueur " + code.Length);
 }
}
```

En consultant la documentation, nous découvrirons que le `System.Int32` est une structure et que sa représentation est comme en Java un nombre entier signé de 32 bits.

Cette structure, en plus d'une location mémoire pour stocker les 32 bits, possède aussi des méthodes, par exemple `ToString()`, avec un `T` majuscule.

Pour se familiariser avec les structures C#, nous allons simplement écrire un petit exemple, très peu orienté objet, mais qui montre quelques aspects nouveaux et très différents de ce langage :

```
public struct  MaStructure {
 public string Nom;
 public int    Annee;

 public int Length {
  get {
   return ToString().Length;
  }
 }

 public override string ToString() {
  return Nom + Annee;
 }

 // Surcharge de l'opérateur -
 public static System.String operator - (System.String str, MaStructure ms) {
    return str + ms.ToString().ToUpper();
 }
}

public class TestMaStructure {
 public static void Main() {
   MaStructure code;
   code.Nom   = "Dupond";
```

```
    code.Annee = 1948;

    System.Console.Out.WriteLine("Mon code: " + code + " " + code.Length);
    System.Console.Out.WriteLine("Mon code: " - code);
  }
}
```

L'`override` est nécessaire, car la méthode `ToString()` existe déjà et nous devons forcer une redéfinition en C#. Si nous n'avions pas surchargé (`override`) cette méthode, le `ToString()` nous aurait retourné le nom de la structure, c'est-à-dire `MaStructure`.

Le résultat présenté est alors :

```
Mon code: Dupond1948 10
Mon code: DUPOND1948
```

Ce n'est pas un très bon exemple en faveur des programmeurs Java, car il nous montre entre autres que :

- Avec ce type de structure, nous sommes plus proches du langage C.
- Le `get` (obtenir), ou `set` (mettre) qui n'est pas utilisé ici, définit la façon de construire une propriété. Cette manière de faire n'existe ni en C++ ni en Java et est plus un héritage de Visual Basic de Microsoft. Nous trouvons ce même type de procédé en Delphi (Borland) bien que cela nous fasse penser à des Beans en Java. Un attribut de classe en Java ou C++ ne peut être une fonction, qui va, comme ici, nous calculer la longueur de notre code.
- Enfin, nous retrouvons une surcharge d'opérateur qui n'existe qu'en C++. Nous avons utilisé l'opérateur -, mais cela n'a rien à voir avec une soustraction, puisque nous mettons en majuscules notre code d'identification.

Mais retombons sur nos pieds, avec deux derniers exemples, pour montrer qu'il est tout à fait possible d'écrire des classes et programmes en C# qui ressemblent terriblement à Java. On pourrait d'ailleurs se demander comment des programmeurs qui viennent d'une culture Visual Basic, sans n'avoir jamais véritablement programmé objet, pourraient « utiliser » ce langage !

La classe Personne du chapitre 4

Voici comment se présenterait notre classe `Personne` du chapitre 4, directement « traduite » en C# :

```
using System;

public class PersonneCs {
 private string nom;
 private string prenom;
 private int    annee;
```

```
  public PersonneCs(string lenom, string leprenom, string lannee) {
    nom    = lenom;
    prenom = leprenom;
    annee  = Int32.Parse(lannee);
  }


  public PersonneCs(string lenom, string leprenom, int lannee) {
    nom    = lenom;
    prenom = leprenom;
    annee  = lannee;
  }


  public void Un_test() {
    Console.Out.WriteLine("Nom et prénom: " + nom + " " + prenom);
    Console.Out.WriteLine("Année de naissance: " + annee);
  }


  public static void Main() {
    PersonneCs nom1 = new PersonneCs("Haddock", "Capitaine", "1907");
    PersonneCs nom2 = new PersonneCs("Kaddock", "Kaptain",   1897);

    nom1.Un_test();
    nom2.Un_test();
  }
}
```

Et le résultat, pour être convaincu :

```
Nom et prénom: Haddock Capitaine
Année de naissance: 1907
Nom et prénom: Kaddock Kaptain
Année de naissance: 1897
```

La seule vraie différence est la méthode statique `Parse()` de la structure `Int32`.

Créer et instancier des classes en Java (C++) ou C# est donc pratiquement équivalent. Évidemment, si nous analysons plus en profondeur le langage C#, nous y découvrirons d'autres aspects et d'autres fonctionnalités comme les destructeurs ou les pointeurs, deux concepts qui n'existent qu'en C++. En revanche, nous pourrons retrouver les concepts Java d'héritage et d'interface.

Nous pensons cependant que nous avons atteint notre but dans ce « petit » chapitre en montrant quelques aspects essentiels de ce nouveau langage.

Couper et coller en C#

Pour terminer ce chapitre, nous avons simplement repris et adapté notre fameux copier/coller du chapitre 22 en C++ et Java, pour montrer ici une version C#.

Pour des programmes de cette dimension, il peut s'avérer nécessaire d'ajouter le `/debug` pour émettre des informations de débogage. Avec cette option de compilation, nous aurons alors la ligne précise où le problème se situe. Ceci se produira par exemple lors de la génération d'une exception.

Le champ `Argument` dans la fenêtre `Tools/Configure User tools/Compilation C#` devra être dans ce cas : `/debug $(FileName)`. Nous verrons alors la présence après recompilation de nouveaux fichiers portant l'extension `.pdb`.

Présentons tout d'abord le code avant de donner quelques détails techniques :

```
using System;
using System.IO;

class CoupeCs {
 private string  monFichier;
 private int     dimension;
 private string  merreur;
 private string  facr;

 static public string  fin = "\r\n";

 public CoupeCs(string leFichier, int laDim) {
   monFichier = leFichier;
   dimension  = laDim;

   int indexPoint = monFichier.IndexOf(".");

   if (indexPoint > 8) {
    indexPoint = 8;
   }
   else {
     if (indexPoint < 0) {
       indexPoint = monFichier.Length;
       if (indexPoint > 8) indexPoint = 8;
     }
   }

   if (indexPoint == 0) {
     facr = "xxx";
   }
   else {
     facr = monFichier.Substring(0, indexPoint);
   }
 }
```

```
public Boolean Coupe() {
 FileStream fsOut    = null;
 StreamWriter sWrAcr = null;
 byte[] tampon = new byte[512];
 int octetsLus = 512;   // nombre d'octets lus
 int octetsTrans = 0;   // nombre d'octets transférés
 int total = 0;         // nombre total d'octets transférés
 int numMorceau  = 1;
 string leMorceau = null;
 int cs1 = 0;
 int cs2 = 0;

 try {
   FileStream fsIn = new FileStream(monFichier, FileMode.Open, FileAccess.Read);

   try {
     sWrAcr = new StreamWriter(facr + ".acr");
     sWrAcr.Write(monFichier + fin);
   }
   catch (IOException) {
     merreur = "Fichier " + facr + " ne peut être écrit";
     return false;
   }

   while ((octetsLus = fsIn.Read(tampon, 0, 512)) > 0) {
     total += octetsLus;

     if (octetsTrans == 0) {
       leMorceau = facr + "." + numMorceau;
       try {
         cs1 = numMorceau;  // toujours différent si fichier identique
         cs2 = 0;
         fsOut = new FileStream(leMorceau, FileMode.Create, FileAccess.Write);
         sWrAcr.Write(leMorceau + fin);
       }
       catch (IOException) {
         merreur = "Fichier " + leMorceau + " ne peut être écrit";
         return false;
       }
     }

     for (int j = 0; j < octetsLus; j++) {
       cs1 = cs1 ^ (int)tampon[j++];

       if (j != octetsLus) {
         cs2 = cs2 ^ (int)tampon[j];
       }
     }

     try {
```

```
        fsOut.Write(tampon, 0, octetsLus);
        octetsTrans += octetsLus;
        if (octetsTrans == dimension) {
          fsOut.Close();
          sWrAcr.Write(((256*cs1) +  cs2) + fin);
          octetsTrans = 0;
          numMorceau++;
        }
      }
      catch (FileNotFoundException) {
        merreur = "Fichier " + leMorceau + " ne peut être écrit";
        return false;
      }
    }

    if (octetsTrans > 0) {
      sWrAcr.Write(((256*cs1) +  cs2) + fin);
      fsOut.Close();
    }

    sWrAcr.Write("EnD" + fin);
    sWrAcr.Write(total + fin);
    sWrAcr.Close();
    fsIn.Close();
  }
  catch (IOException) {
    merreur = "Fichier " + monFichier + " n'existe pas: ";
    return false;
  }

  return true;
}

public string ErreurMessage() {
  return merreur;
}

public override string ToString() {
  return facr + ".acr";
}

public static void Main(string[] args) {
  int argsLen = args.Length;

  if ((argsLen < 1) || (argsLen > 2)) {
    Console.Error.WriteLine("Nombre de paramètres invalides");
    Console.Error.WriteLine("CoupeCs fichier      (pour disquettes)");
    Console.Error.WriteLine("CoupeCs dim fichier  (en fichiers de dim >= 1024)");
    Console.Error.WriteLine("CoupeCs Kn fichier   (en fichiers de n Kilo-octets)");
    Console.Error.WriteLine("CoupeCs Mn fichier   (en fichiers de n mégaoctets)");
    return;
```

```
    }

    int fi = 0;
    int adim = 1457664;   // dimension d'une disquette
    if (argsLen == 2) {   // dimension donnée
      fi = 1;
      char firstChar = args[0][0];

      try {
        if ((firstChar == 'K') || (firstChar == 'M')) {
          adim = 1024 * Int16.Parse(args[0].Substring(1));
          if (firstChar == 'M')  adim *= 1024;
        }
        else {
          adim = Int16.Parse(args[0]);
          if ((adim % 512) != 0) {
            Console.Error.WriteLine("La dimension doit être un multiple de 512");
            return;
          }
        }
      }
      catch(FormatException) {
        Console.Error.WriteLine("La dimension est incorrecte: K(nombre), M(nombre) ou
        ➥nombre");
        return;
      }
    }

    CoupeCs maCoupe = new CoupeCs(args[fi], adim);
    if (maCoupe.Coupe()) {
      Console.Out.WriteLine("Coupe terminée correctement: " + maCoupe);
    }
    else {
      Console.Error.WriteLine("Coupe erreur: " + maCoupe.ErreurMessage());
    }
  }
}
```

```
using System;
using System.IO;

public class ColleCs {
 private string fichierAcr;
 private string merreur;

 public ColleCs(string leFichier) {
   fichierAcr = leFichier;
 }
```

```
public bool Colle() {
 string monFichier;
 string unFichier;
 int dimTotal = 0;        // dimension du fichier construit
 byte[] tampon = new byte[512];
 int octetsLus   = 0;     // nombre d'octets lus
 int octetsTrans = 0;     // nombre d'octets transférés
 int numMorceau  = 0;
 int cs  = 0;             // cs du fichier .acr
 int cs1 = 0;
 int cs2 = 0;

 try {
   StreamReader sRdAcr = new StreamReader(fichierAcr);

   try {
     monFichier = sRdAcr.ReadLine();   // le fichier final
   }

   catch(IOException) {
     merreur = "Erreur de lecture du fichier .acr";
     return false;
   }

   if (monFichier.Length > 50) {
     merreur = "Fichier trop long (>50) ou erreur de lecture du fichier .acr";
     return false;
   }

   try {
     FileStream fsOut = new FileStream(monFichier, FileMode.Create,
     ➥FileAccess.Write);
     try {
       for (;;) {
         unFichier = sRdAcr.ReadLine();
         if (unFichier == null) {
           merreur = "Fichier .acr incorrect";
           return false;
         }

         if (unFichier.Equals("EnD")) {
           fsOut.Close();
           dimTotal = Int32.Parse(sRdAcr.ReadLine());

           if (dimTotal == octetsTrans)  return true;
           else {
             merreur = "Nombre invalide de caractères transférés";
             return false;
           }
         }
```

```
          cs = Int32.Parse(sRdAcr.ReadLine());
          numMorceau++;

          cs1 = numMorceau;
          cs2 = 0;

          FileStream fsIn = new FileStream(unFichier, FileMode.Open,
          ➥FileAccess.Read);
          try {
            for (;;) {
              octetsLus = fsIn.Read(tampon, 0, 512);
              if (octetsLus <= 0)  break;

              for (int j = 0; j < octetsLus; j++) {
                cs1 = cs1 ^ (int)tampon[j++];

                if (j != octetsLus) {
                  cs2 = cs2 ^ (int)tampon[j];
                }
              }

              fsOut.Write(tampon, 0, octetsLus);
              octetsTrans += octetsLus;
            }

            fsIn.Close();
          }
          catch(IOException) {
            merreur = "Erreur de lecture pour " + unFichier;
            return false;
          }

          if (cs != ((256*cs1) +  cs2)) {
            merreur = "Acr invalide pour le fichier " + unFichier;
            return false;
          }
        }
      }
      catch(IOException) {
        merreur = "Fichier .acr invalide";
        return false;
      }
    }
    catch (FileNotFoundException) {
      merreur = "Le fichier " + monFichier + " ne peut être écrit";
      return false;
    }

  }
  catch (IOException) {
    merreur = "Le fichier " + fichierAcr + " n'existe pas";
```

```
    return false;
  }
 }

 public string ErreurMessage() {
   return merreur;
 }


 public static void Main(string[] args) {
   if (args.Length != 1) {
     Console.Error.WriteLine("Nombre de paramètres invalides");
     Console.Error.WriteLine("ColleCs fichierAcr");
     return;
   }

   string leFichierArc = args[0];

   if (leFichierArc.LastIndexOf(".acr") != (leFichierArc.Length - 4)) {
     Console.Error.WriteLine("Ce n'est pas un fichier .acr");
     return;
   }

   ColleCs maColle = new ColleCs(leFichierArc);
   if (maColle.Colle()) {
     Console.Out.WriteLine("Colle terminée correctement");
   }
   else {
     Console.Error.WriteLine("Colle erreur: " + maColle.ErreurMessage());
   }
 }
}
```

Il y a quelques petites différences de syntaxe, en plus de celles que nous avons déjà mentionnées. La séquence `try catch` est sans doute la plus significative.

Pour les fonctions d'entrée et de sortie, il faut se référer aux classes de l'espace de noms `System.IO`. Ces classes sont très similaires à celles que nous avons déjà utilisées dans le chapitre 9. Il suffit en fait de naviguer dans la documentation pour trouver son bonheur :

`http://msdn.microsoft.com/fr-fr/netframework/system.io.aspx`

Résumé

Nous espérons que ce chapitre consacré au langage C# a donné au lecteur un premier aperçu de ces possibilités et montré quelques différences majeures avec ces frères C++ et Java. Cette introduction, très brève, devrait permettre à un bon programmeur Java (ou C++) de développer rapidement de petites applications en C#.

Annexes

A

Contenu du CD-Rom

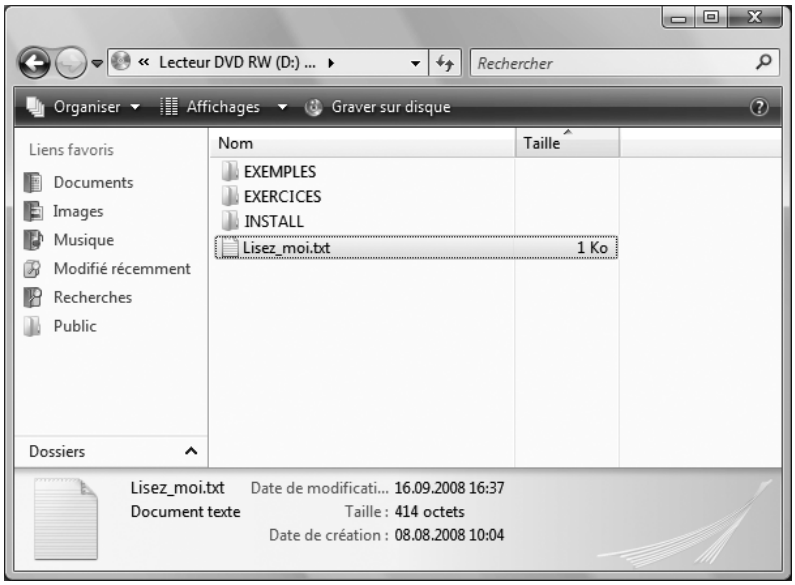

Figure A-1
Contenu du CD-Rom

Le CD-Rom fourni avec cet ouvrage contient les différents outils qui seront installés dans les annexes B à E.

Pour suivre les exemples et exercices de ce livre, il convient d'installer tous les outils présentés dans les annexes B (outils Java et C++) et C (éditeur Crimson). L'éditeur Crimson nous permettra d'éditer nos sources de code et de les compiler dans Java ou C++.

Les lecteurs des éditions précédentes de ce livre commençaient souvent par tout installer, sans que cela soit vraiment nécessaire.

Les exemples, et éventuellement les exercices, doivent également être copiés sur votre ordinateur, à un emplacement déterminé, comme pour les outils. Si d'autres répertoires ou disques que ceux mentionnés dans ce livre sont choisis, il faudra suivre les instructions et adapter les chemins d'accès. Ceci est décrit et répété dans les annexes respectives.

Le chapitre 23, dédié au langage C#, nécessite l'installation du .NET (`dotnetfx35.exe`), laquelle peut être effectuée séparément ou ultérieurement (voir annexe D). Nous pouvons utiliser l'éditeur Crimson pour éditer les sources C# (voir annexe C).

L'installation de NetBeans (voir annexe E) peut aussi être réalisée indépendamment ou plus tard. À noter cependant que MinGW devra obligatoirement être installé pour la partie C++ (voir annexe B).

Tous les détails, les caractéristiques et les vérifications d'installation sont expliqués dans les différentes annexes.

B

Installation des outils de développement pour Java et C++

Avant d'installer les éditeurs qui nous permettront de créer, de modifier et de corriger nos programmes, il nous faut installer les outils de développement, c'est-à-dire les programmes qui vont nous permettre de compiler du code source écrit en Java ou en C++. Nous nous intéresserons d'abord aux outils destinés au langage Java et examinerons ensuite ceux du langage C++. Les outils installés dans cette annexe à partir du CD-Rom sont en principe pris en charge par toutes les versions du système d'exploitation de Microsoft, de Windows 98 à Windows Vista. Cependant, nous ne donnerons ici que des exemples avec XP et Vista, en alternance.

Quelques lecteurs des précédentes éditions de cet ouvrage avaient rencontré des difficultés lors de l'installation des logiciels. Dans cette annexe et les suivantes, nous avons fait un effort particulier pour eux. Les « professionnels » pourront lire plus rapidement les passages consacrés aux différentes phases d'installation et s'intéresser à de nombreux détails techniques qui les aideront non seulement à comprendre certaines particularités de ces outils, mais aussi à les installer ou à les utiliser sur des systèmes configurés différemment.

Cependant, si un lecteur rencontre une difficulté quelconque, l'auteur l'encourage tout d'abord à consulter le site Web consacré à cet ouvrage (voir l'avant-propos). Ensuite, si nécessaire, il pourra contacter l'auteur et lui expliquer le problème afin que celui-ci puisse apporter les modifications et explications supplémentaires dans les prochaines éditions de ce livre, et ainsi aider tous les lecteurs qui seraient également confrontés à des complications particulières.

L'espace disque total requis pour l'installation de tous les outils et de la documentation est d'environ 600 Mo. Il faudra donc s'assurer que l'espace est suffisant sur le disque `C:`.

Si un autre disque ou d'autres partitions sont utilisés, il faudra alors installer les composants sur l'un ou l'autre, voire les deux. Dans ce cas, il faudra adapter les configurations, ce qui est en général mentionné lors de l'installation de chaque composant.

Attention !

Il peut s'avérer nécessaire, en cas de difficultés d'installation, de copier tous les fichiers du CD-Rom dans un répertoire de travail temporaire (par exemple, `C:\TEMP`) avant de procéder à l'installation depuis cet emplacement.

Installation de 7-Zip

Tout programmeur qui télécharge des logiciels sur Internet et les installe ensuite sur son PC aura besoin, tôt ou tard, d'un outil de décompression.Pour cet ouvrage, nous avons choisi le logiciel Open Source 7-Zip, qui est une alternative à WinZip. Pour le télécharger, rendons-nous sur le site Web suivant : `http://www.7-zip.org/`.

Si certains lecteurs ont déjà installé WinZip et qu'ils le maîtrisent bien, ils n'auront pas besoin d'exécuter le programme `7z457.exe` situé dans le répertoire `INSTALL` du CD-Rom. Nous avons choisi ici une installation sous Windows Vista mais la procédure est identique pour Windows XP. En exécutant le fichier `7z457.exe`, nous obtenons la fenêtre présentée à la figure B-1 :

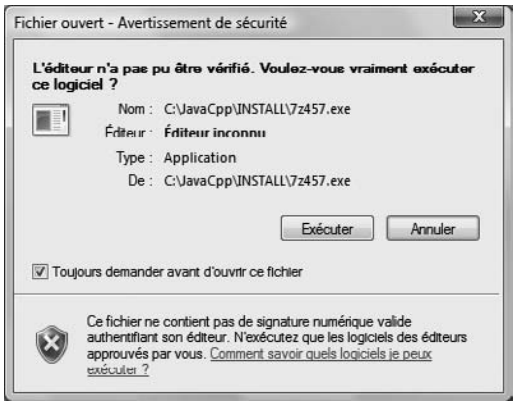

Figure B-1

Début de l'installation de 7-Zip

Cliquons sur le bouton Exécuter afin d'accepter le logiciel, puis sur Autoriser.

Le programme d'installation est capable d'identifier où est installé Windows (ici, sur le disque `C:`). Certains programmeurs décident souvent de créer un répertoire `Outils` pour des programmes de ce type ou encore de l'installer à la racine, par exemple, `C:\7Zip`. Si le lecteur ne sait pas vraiment quoi faire, il peut opter pour la procédure proposée par défaut, qui est tout à fait satisfaisante (figure B-2).

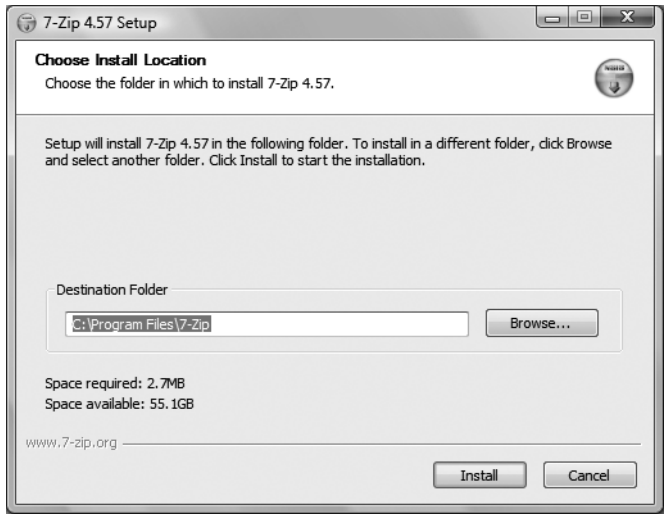

Figure B-2
Le répertoire d'installation de 7-Zip

Cliquons simplement sur le bouton Install pour lancer l'installation.
À la fin de la procédure, cliquons sur le bouton Finish (figure B-3).

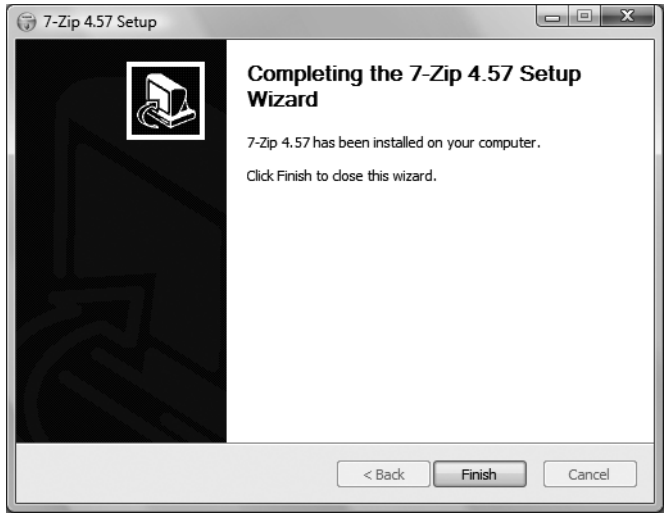

Figure B-3
Fin de l'installation de 7-Zip

7-Zip, tout comme WinZip, permet au lecteur d'explorer des fichiers compressés qui sont le plus souvent composés d'un catalogue de plusieurs fichiers et parfois même d'un ensemble plus complet comprenant des répertoires et des sous-répertoires. Des outils sont disponibles non seulement pour extraire des fichiers individuellement, mais aussi pour les examiner avant de les décompresser. Le fichier `jdk-6-doc.zip` du CD-Rom d'accompagnement contient toute la documentation de Java pour les programmeurs. C'est un seul et unique fichier de 53 Mo qui contient plus de 12 000 fichiers faisant au total 259 Mo ! La compression de fichier texte est impressionnante. À la section « Installation de la documentation » de cette annexe, nous verrons comment configurer l'éditeur pour afficher des fichiers texte internes.

Grâce à 7-Zip, nous pouvons évidemment compresser un ou plusieurs fichiers, par exemple avant d'envoyer de nombreux documents ou fichiers de n'importe quel format à des amis, que ce soit sur un média ou par e-mail.

Installation des exemples et des exercices

Les exemples de ce livre vont nous permettre de vérifier rapidement l'installation des divers composants. C'est pourquoi nous allons commencer par eux.

`X:` est la lettre associée au CD-Rom (`D:` ou autres, cela dépend du PC et des disques, partitions ou autres lecteurs installés). Nous allons commencer par copier les dossiers `X:\EXEMPLES` et `X:\EXERCICES` du CD-Rom sur le PC dans le répertoire `C:\JavaCpp`. Après cette opération, nous devrions obtenir les répertoires présentés à la figure B-4 :

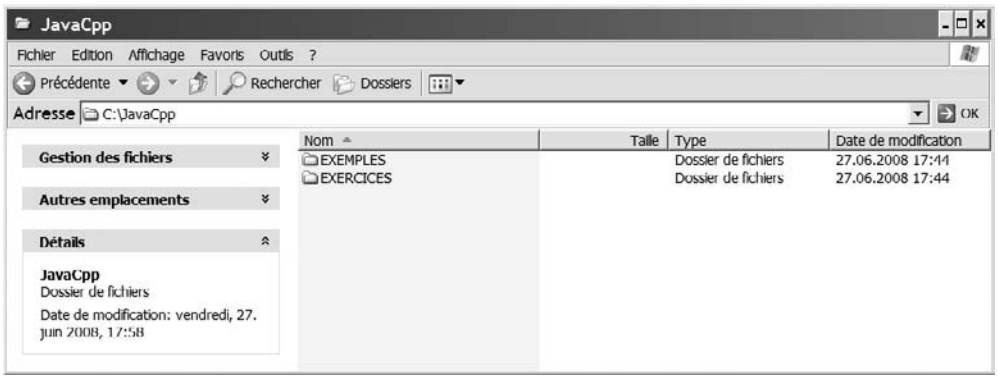

Figure B-4

Répertoires des exemples et des exercices

Pour les exercices, nous conseillons au lecteur d'essayer de les résoudre et de les programmer avant de consulter leurs solutions.

Installation du JDK de Sun Microsystems

Avant de passer à l'installation d'un JDK, il faudra identifier si une version est déjà installée sur le PC. Dans ce cas, une nouvelle installation ne sera pas nécessaire : il suffira de déterminer l'emplacement du JDK et si celui-ci se trouve déjà dans la variable d'environnement `PATH`. Il s'agit de mises au point traditionnelles et nous pouvons déjà indiquer que l'installation de plusieurs JDK ne causerait aucun problème.

Qu'est-ce que le JDK ?

JDK signifie *Java Development Kit*. C'est un kit de développement mis à disposition par Sun Microsystems pour développer des programmes en Java et les exécuter avec une machine virtuelle (voir chapitre 1).

Désinstallation des anciennes versions

Tout d'abord, il faut s'assurer que d'anciens kits JDK ne sont pas déjà installés sur le système. En effet, il est conseillé de ne garder que la dernière version du JDK 1.6 (officiellement nommé JDK 6), fournie sur le CD-Rom accompagnant cet ouvrage, à moins que d'autres outils ou programmes utilisent des versions plus anciennes. Il arrive d'ailleurs souvent que plusieurs versions soient installées sur un PC, en particulier pour la documentation. C'est inutile et cela prend vraiment beaucoup de place. En revanche, si des applications sont générées par d'anciennes versions du JDK et pour d'anciennes machines virtuelles JRE (voir chapitre 1), il faudra les garder.

Avant d'effacer un répertoire existant, par exemple `C:\jdk.1.4.2` (emplacement non standard), il est recommandé de vérifier qu'il n'existe pas un moyen de désinstaller ce JDK avec les outils de Windows (Panneau de configuration > Ajouter ou supprimer des programmes). Comme le JDK peut être utilisé sans être réinstallé, il suffit parfois d'effacer simplement le répertoire et son contenu. Il faudra évidemment aussi effacer les anciennes références dans les variables d'environnement comme `PATH`, `CLASSPATH` ou encore `JAVA_HOME` (Panneau de configuration > Système > Avancé > Variables d'environnement). Le `PATH` est souvent très long et il faudra l'éditer pour l'examiner plus précisément.

Téléchargement à partir du site Web de Sun Microsystems

Le JDK peut aussi être téléchargé depuis le site de Sun Microsystems :

```
http://java.sun.com/javase/downloads/index.jsp
```

Comme toujours, ce lien pourrait disparaître et être remplacé par un autre. Dans ce cas, il faudra consulter le site :

```
http://java.sun.com/
```

cliquer sur le lien Downloads et rechercher le JDK.

Nous trouverons aussi sur ce site des versions destinées à d'autres systèmes d'exploitation comme Linux. La version Windows incluse sur le CD-Rom d'accompagnement est : `jdk-6u6-windows-i586-p.exe`. Elle correspond à la version 1.6.6 du JDK.

Le code Java de cet ouvrage a été vérifié pour cette version. Si une version inférieure était utilisée, des adaptations pourraient s'avérer nécessaires. Pour les versions supérieures, Sun Microsystems garantit la compatibilité ou donnera des indications pour des méthodes ou classes éventuellement dépréciées.

Nous trouverons évidemment les versions JRE (voir chapitre 1) sur ce même site.

Installation à partir du CD-Rom

Avant de procéder à l'installation de la version 1.6 du JDK, le lecteur doit s'assurer qu'il possède au moins 250 Mo d'espace sur l'un de ses disques. De plus, s'il désire installer la documentation, ce qui est une nécessité pour les programmeurs Java, il faut y ajouter 290 Mo !

Nous allons à présent décrire la procédure d'installation du JDK sous Windows XP. À noter qu'elle serait identique sous Windows Vista (il faudra juste accepter, au démarrage, la demande d'autorisation).

Pour commencer, il suffit de cliquer sur le fichier `X:\INSTALL\jdk-6u6-windows-i586-p.exe` du CD-Rom d'installation fourni avec ce livre (`X:` désignant le lecteur CD-Rom). La fenêtre de la figure B-5 devrait alors apparaître :

Figure B-5

Démarrage de l'installation du JDK 1.6

Après avoir cliqué sur le bouton Next >, la fenêtre de la figure B-6 apparaîtra :

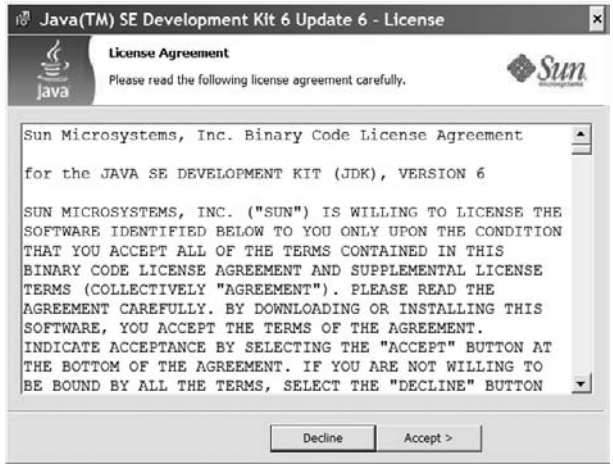

Figure B-6

Accepter la licence du JDK 1.6

Nous accepterons évidemment l'accord de licence (bouton Accept >) afin d'accéder à la fenêtre suivante (figure B-7) :

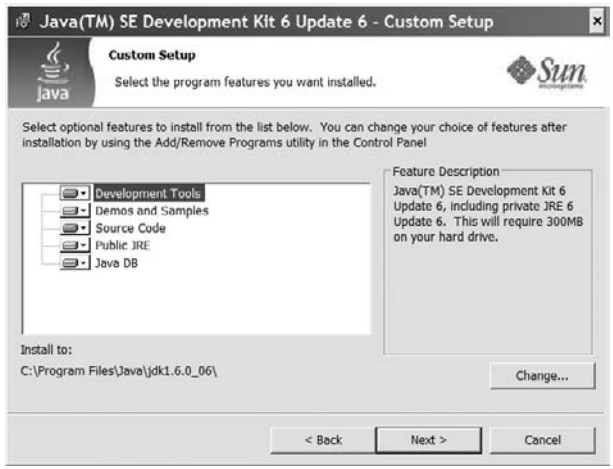

Figure B-7

Configuration de l'installation du JDK 1.6

Nous ne modifierons aucune option et poursuivrons l'installation en conservant également le répertoire d'installation proposé par défaut si nous avons suffisamment d'espace disque sur cette partition. Après avoir cliqué sur le bouton Next >, la fenêtre de la figure B-8 s'affichera à l'écran :

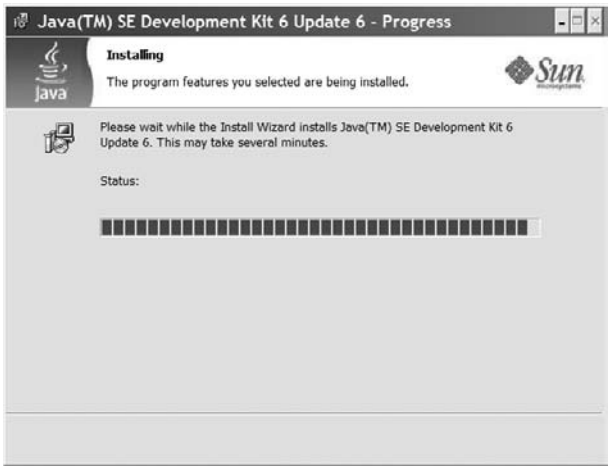

Figure B-8

Progression de l'installation du JDK 1.6

Nous attendrons que l'installation se termine avec quelques annonces publicitaires de Sun Microsystems (figure B-9) :

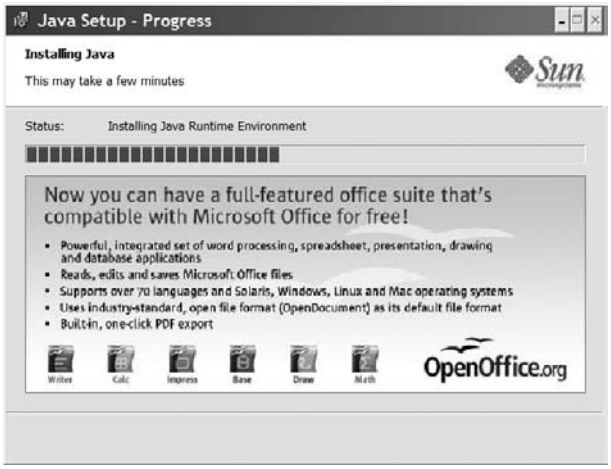

Figure B-9

État d'avancement de l'installation du JDK 1.6

À la fin de l'installation, la fenêtre de la figure B-10 apparaîtra :

Figure B-10
Fin de l'installation

Il suffira de cliquer sur le bouton Finish pour terminer la procédure d'installation. Le lecteur pourra éventuellement enregistrer le logiciel auprès de Sun Microsytems mais cela n'est pas indispensable.

Nous reviendrons par la suite sur le `PATH`, c'est-à-dire le chemin d'accès du compilateur `javac.exe`.

Installation de MinGW (g++) et MSYS

Le compilateur C++ que nous avons choisi pour compiler les différents exemples et exercices de cet ouvrage est intégré au paquet MinGW (*Minimalist GNU for Windows*).

Nous vous invitons à consulter les sites Web suivants pour plus d'informations sur MinGW :

```
http://www.mingw.org/
http://en.wikipedia.org/wiki/MinGW
http://fr.wikipedia.org/wiki/GNU
```

Le MSYS contient notamment l'outil `make` utilisé tout au long de l'ouvrage (voir la description du `Makefile` au chapitre 1).

MingGW et MSYS sont en fait des outils Linux utilisables dans un environnement Windows. Dans les précédentes éditions de ce livre, nous utilisions Cygwin (`http://www.cygwin.com/`) mais les versions actuelles de MinGW et de MSYS suffisent et fonctionnent parfaitement avec NetBeans également (voir annexe E).

Installation simplifiée de MinGW et MSYS

Nous allons à présent copier les outils MinGW et MSYS directement depuis le CD-Rom d'accompagnement. Cette opération va nous simplifier la tâche et nous évitera une installation fastidieuse et compliquée à partir de différents sites Web voir la section « MinGW et MSYS sur Internet » ci-après). L'installation simplifiée consiste à copier les répertoires `MinGW` et `msys` du dossier `INSTALL` du CD-Rom sur le disque `C:`.

Si un autre disque est choisi (par exemple, `D:`), il faudra modifier toutes les références à `MinGW` et `msys` utilisant le disque `C:`, dans l'installation et dans l'ouvrage (voir les annexes C et E, pour l'éditeur Crimson et NetBeans).

Si MinGW et MSYS sont déjà installés, il faudra les effacer, les déplacer ou les réinstaller ailleurs.

Avant de pouvoir vérifier l'installation, il nous faut adapter le `PATH` pour ajouter les références aux outils MSYS et MinGW.

Nous allons décrire ici l'installation sous Windows XP, qui est similaire à celle sous Windows Vista. Pour commencer, il convient de sélectionner le menu Démarrer > Panneau de configuration > Système > Avancé (figure B-11).

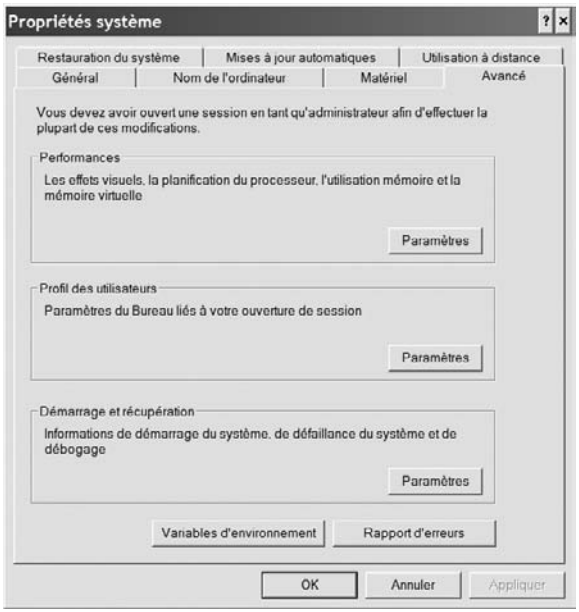

Figure B-11

Propriétés système – Windows XP

Sous Windows Vista, le chemin est Démarrer > Panneau de configuration > Système > Paramètres système avancés (avec demande d'autorisation) :

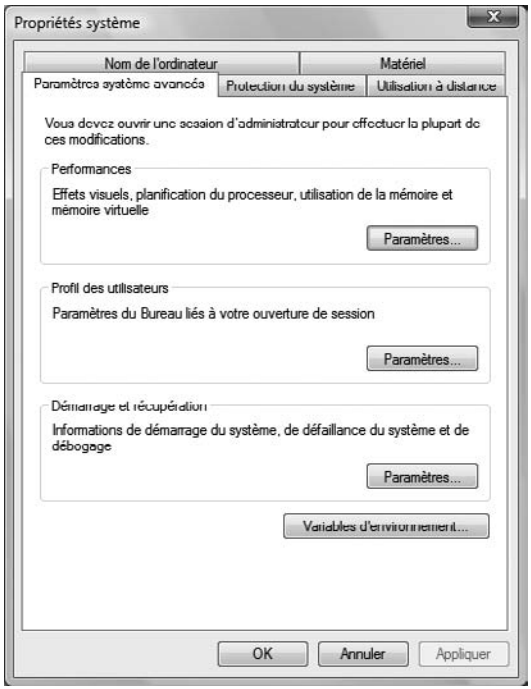

Figure B-12

Propriétés système – Windows Vista

Nous devons ensuite cliquer sur le bouton Variables d'environnement puis, dans la rubrique Variable système, sélectionner Path et cliquer sur le bouton Modifier (la procédure sous Windows Vista est similaire) : il faudra ajouter devant le contenu actuel de la variable `Path` les deux valeurs existantes, sans oublier les points-virgules (;) (figure B-13) :

```
C:\msys\1.0\bin;C:\MinGW\bin;
C:\Program Files\Java\jdk1.6.0_06\bin;
```

Figure B-13

Windows Path pour les outils C++ et Java

Il faudra également s'assurer qu'il n'y a pas d'espaces en trop ou que la variable `Path` n'est pas trop longue, ce qui peut arriver sur des systèmes assez anciens, où certains outils possèdent un chemin d'accès compliqué.

Vérifications finales et dernières mises au point

Les outils de développement sont à présent installés. Il s'agit maintenant de procéder à une vérification à la fois globale et pointue avec les exemples du chapitre 1 et l'interface JNI du chapitre 21.

Nous terminerons par l'installation de la documentation supplémentaire disponible sur le CD-Rom puis par la préparation d'un environnement de développement convivial.

Vérification de l'installation des outils

Nous allons à présent vérifier que les outils pour les compilateurs C++ et Java, ainsi que l'outil `make`, sont correctement installés. Cette partie est essentielle avant de passer à l'installation de l'éditeur Crimson (voir annexe C) ou de NetBeans (voir annexe E).

La méthode la plus simple consiste à saisir manuellement les commandes. Ouvrons pour cela une fenêtre DOS : allons dans le menu Démarrer > Exécuter, puis tapons `cmd` dans le champ Ouvrir (figure B-14) :

Figure B-14
Passage en mode commande (DOS)

Ensuite, nous entrerons :

```
C:
```

suivi de :

```
cd C:\JavaCpp\EXEMPLES\Chap01
```

afin de se retrouver dans le répertoire des exemples du premier chapitre (les exemples et exercices ont été installés sur le PC en début de cette annexe à partir du CD-Rom).

Figure B-15
DOS – Positionnement dans le répertoire de travail

Nous pouvons aussi créer un raccourci sur le Bureau vers la commande `cmd.exe` (figure B-16) :

Figure B-16
Raccourci vers la commande cmd.exe

Il faudra ensuite effectuer un clic droit sur ce raccourci et modifier l'entrée Démarrer dans par `C:\JavaCpp\EXEMPLES`. Le raccourci devra également être renommé en `JavaCpp EXEMPLES` (figure B-17) :

Figure B-17
Raccourci vers les exemples

De cette manière, nous pourrons très facilement ouvrir une fenêtre DOS pour ensuite entrer `cd Chap01` ou un autre répertoire choisi.

Que faut-il vérifier ?

- que le compilateur Java (`javac.exe`) fonctionne et que les fichiers binaires sont exécutables ;
- que le compilateur C++ (`g++` de MinGW) fonctionne et que les fichiers binaires sont exécutables ;
- que le `make.exe` (`Makefile`) fonctionne correctement.

Dans cette même fenêtre DOS, située dans le répertoire `C:\JavaCpp\EXEMPLES\Chap01`, nous allons exécuter les commandes suivantes :

```
java -version
javac -version
g++ -v
make -v
```

Elles nous indiqueront non seulement que les chemins d'accès sont corrects, mais également que les outils sont installés au bon endroit (figure B-18). Dans le cas contraire, il faudra vérifier celui ou ceux qui nous retournent un résultat inattendu. Les exécutables `java` et `javac` font partie de l'installation du JDK, et le `g++` et le `make` des utilitaires MinGW et MSYS. Une erreur classique est une faute de frappe dans la variable `Path` (figure B-14).

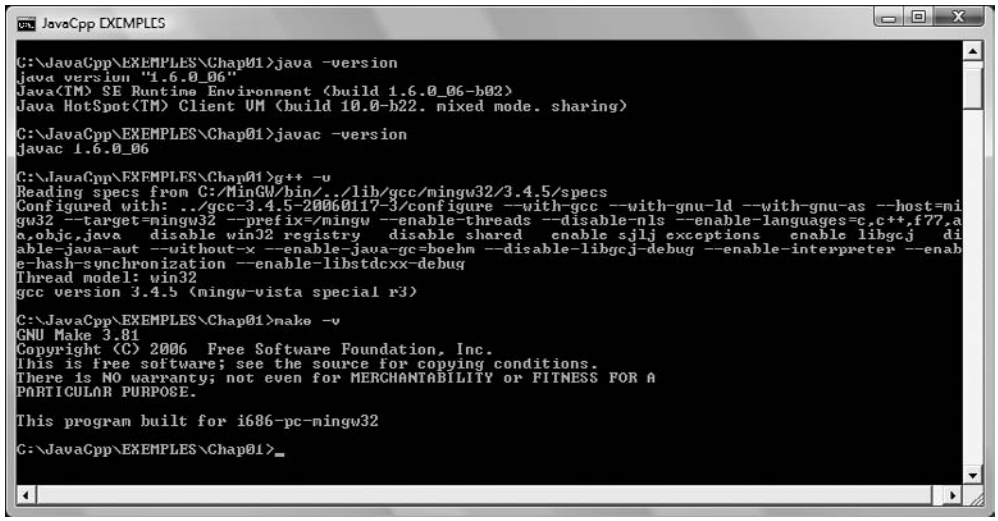

Figure B-18

Vérification des outils et des versions

Enfin, pour terminer la vérification, un simple `make` dans le répertoire `C:\JavaCpp\EXEMPLES\Chap01` fera l'affaire, ainsi que l'exécution des deux binaires Java et C++ générés :

```
make
java Hello
Hello.exe
```

Si nous avons déjà compilé du code ou que nous avons des doutes, nous pouvons toujours, avant de lancer ces trois commandes, exécuter le fichier `efface.bat`. Il permet d'effacer les fichiers objets et binaires afin de forcer les recompilations (`javac` et `g++`) au travers du `make`. Le script `efface.bat` n'a pas de `PAUSE` à la fin et rend l'opération invisible, si celle-ci est exécutée depuis l'explorateur Windows par un double-clic sur le fichier de script. Ces fichiers `efface.bat`, également présents dans chacun des répertoires des exercices, exécutent en fait un `make clean` qui appelle le point d'entrée `clean` (nettoyer) du `Makefile` (voir les explications détaillées au chapitre 1). Si le lecteur a des difficultés, il lui suffit de consulter ces fichiers pour en comprendre le mécanisme.

Si tout se passe correctement, nous pouvons passer à l'installation de l'éditeur Crimson (voir annexe C). Le lecteur devrait comprendre ici, que si un autre éditeur de texte était utilisé pour modifier ou créer des fichiers source Java, C++ et des `Makefile` associés, il pourrait toujours utiliser une fenêtre DOS pour compiler et exécuter les programmes du livre ou ses propres programmes.

Le fameux chapitre 21 avec JNI

Pour vérifier que l'installation des outils s'est correctement déroulée, nous pouvons procéder à un test encore plus complet : exécuter le `Makefile` du chapitre 21. Le `make` sera alors

vérifié, mais aussi les compilateurs `javac` et `g++` ainsi que le `dllwrap` de la distribution MinGW qui va nous générer une bibliothèque `dll` pour Windows.

Pour ce faire, nous nous rendrons dans le répertoire du chapitre 21 (figure B-19) :

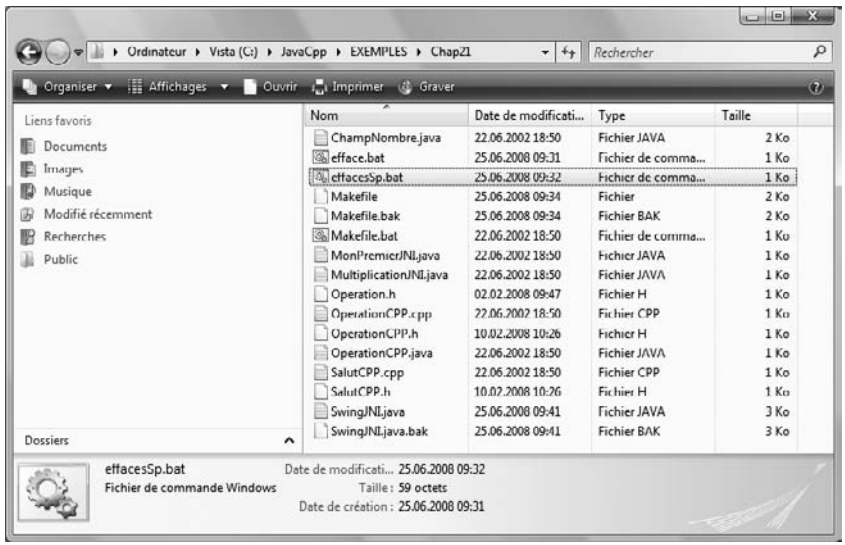

Figure B-19

Répertoire du chapitre 21 sous Windows Vista

Nous exécuterons tout d'abord le fichier `efface.bat` (pour assurer une nouvelle compilation) puis le fichier `Makefile.bat` (figure B-20) :

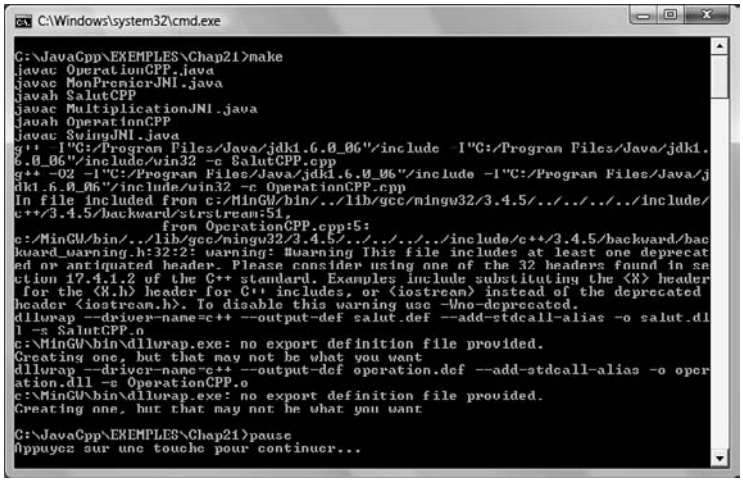

Figure B-20

Makefile du chapitre 21

Les messages d'information sont corrects. Si des erreurs apparaissent, il faudra chercher la source du problème, par exemple la référence au JDK (première ligne du `Makefile`).

Le chapitre 21 reste la référence de l'auteur si une autre version de JDK ou de MinGW est utilisée. Un simple `make` dans le répertoire du chapitre 21 suffit (`Makefile.bat`) ! Cela va compiler les sources C++ et Java et vérifier tous les composants et les compatibilités de versions.

Pour terminer la vérification, nous pourrons double-cliquer sur le fichier `test.bat` dans le répertoire `C:\JavaCpp\EXEMPLES\Chap21` (figure B-21) :

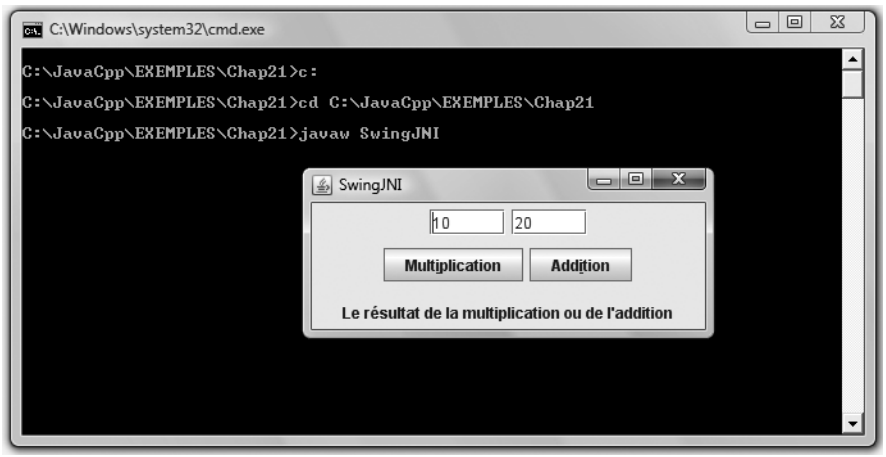

Figure B-21

Test du chapitre 21

Ceci nous indiquera que tous les outils Java et C++ semblent être correctement installés.

Le fichier source src.jar

Si nous avons installé le paquet `Java Sources`, nous pouvons consulter le fichier `src.zip` compressé (répertoire `C:\Program Files\Java\jdk1.6.0_06`)avec un outil comme 7-Zip. La figure B-22 montre comment ouvrir un fichier `.zip` avec 7-Zip en effectuant un clic droit sur le fichier et en sélectionnant 7-Zip > Open archive.

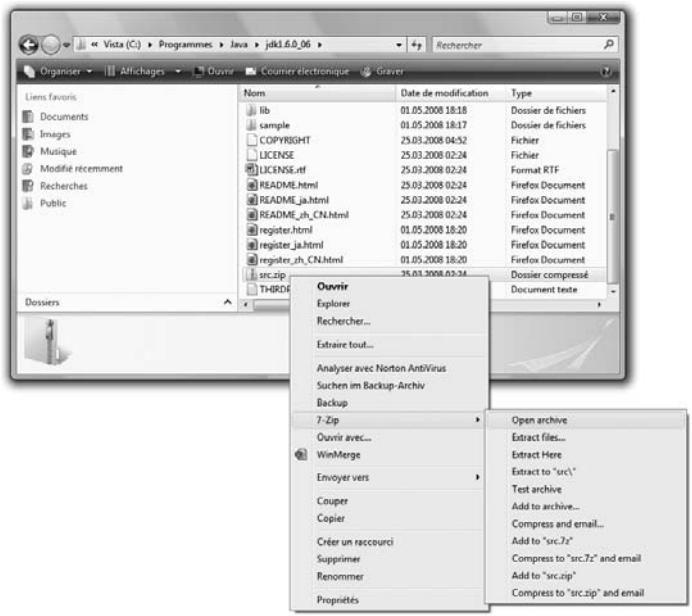

Figure B-22
Ouvrir un fichier .zip avec 7-Zip

Si, par exemple, nous recherchons la classe `String`, nous la trouverons dans le sous-répertoire `src\java\lang` de l'archive en y naviguant avec 7-Zip (figure B-23) :

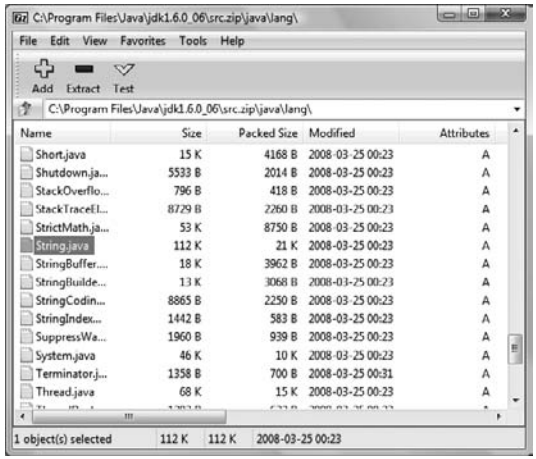

Figure B-23
Code source de la classe Java String

Avant de visualiser la classe `String.java`, il faut au préalable configurer l'éditeur employé par 7-Zip (menu Tools > Options > Editor) (figure B-24) :

Figure B-24
Éditeur pour 7-Zip

Nous choisirons l'éditeur Wordpad pour Windows Vista et indiquerons donc le fichier `wordpad.exe`, situé dans le répertoire `C:\Program Files\Windows NT\Accessories\`, dans le champ Editor.

Sans cette configuration, nous utiliserions par défaut NotePad (Bloc-notes), qui ne peut traiter correctement des fichiers texte créés sous Linux. C'est pourquoi il faut utiliser `Edit` (qui appelle l'éditeur par défaut) et non `Open` dans 7-Zip.

Dans l'exemple qui suit, nous avons recherché, dans le fichier `Strings.java`, la méthode `equals()` qui nous permet de savoir si deux objets de la classe `String` sont identiques :

```
/**
 * Compares this string to the specified object.
 * The result is <code>true</code> if and only if the argument is not
 * <code>null</code> and is a <code>String</code> object that represents
 * the same sequence of characters as this object.
 *
 * @param   anObject   the object to compare this <code>String</code>
 *                     against.
 * @return  <code>true</code> if the <code>String </code>are equal;
 *          <code>false</code> otherwise.
 * @see     java.lang.String#compareTo(java.lang.String)
 * @see     java.lang.String#equalsIgnoreCase(java.lang.String)
 */
public boolean equals(Object anObject) {
  if (this == anObject) {
```

```
    return true;
  }
  if (anObject instanceof String) {
    String anotherString = (String)anObject;
    int n = count;
    if (n == anotherString.count) {
      char v1[] = value;
      char v2[] = anotherString.value;
      int i = offset;
      int j = anotherString.offset;
      while (n-- != 0) {
        if (v1[i++] != v2[j++]) {
          return false;
        }
      }
  return true;
    }
  }
  return false;
}
```

Nous y trouvons non seulement la documentation (dans les commentaires Javadoc avant le code source), mais également que :

- si ces deux objets sont les mêmes, il n'est pas nécessaire de vérifier le contenu ;
- si l'objet n'est pas un `String`, il est inutile de vérifier si chaque caractère est identique.

Cela donnera aux programmeurs d'autres exemples de code ou encore la confirmation que la méthode utilisée fait bien le travail qu'elle est censée faire.

Installation de la documentation

De la même manière que pour l'ouverture du fichier `src.zip` ci-dessus, nous allons ouvrir le fichier `jdk-6-doc.zip` situé dans le dossier `INSTALL` du CD-Rom (clic droit sur le fichier puis 7-Zip > Open archive) afin d'installer la documentation JDK. À noter que la procédure est identique sous Windows XP et Vista.

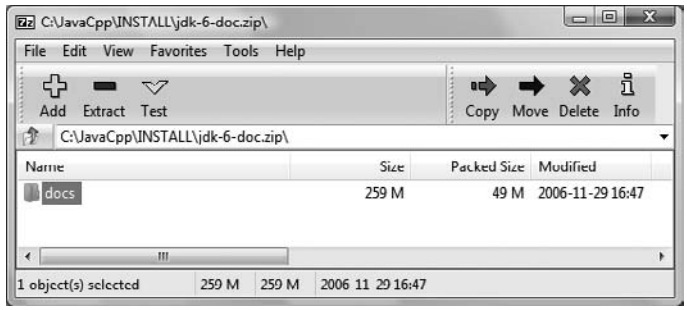

Figure B-25

Parcours de la documentation du JDK avec 7-Zip

Avant de continuer et de procéder à l'installation de la documentation de la version 1.6 du JDK, le lecteur devra s'assurer qu'il possède au moins 259 Mo d'espace libre sur son disque.

Pour obtenir la fenêtre représentée à la figure B-26, nous allons sélectionner Extract files… au lieu de Open archive dans le menu contextuel.

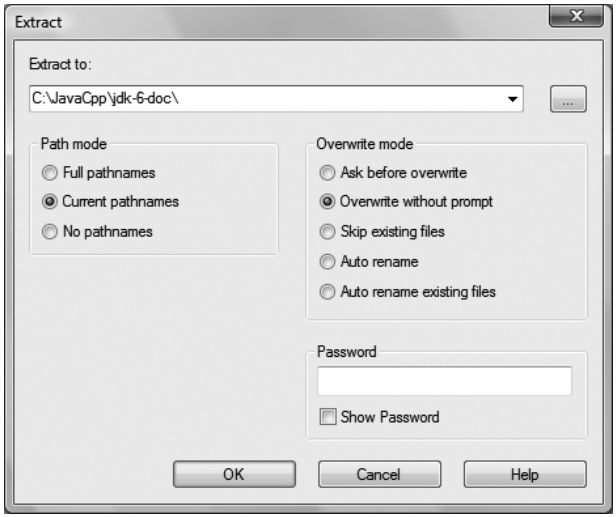

Figure B-26

Emplacement de la documentation du JDK

Nous avons choisi le répertoire `C:\JavaCpp\jdk-6-doc` car il sera plus facile de le déplacer ultérieurement sur un autre disque ou partition en cas de problème d'espace disque. En le copiant dans le répertoire d'installation du JDK, nous risquons de l'oublier (beaucoup d'espace disque) lors de l'installation de nouvelles versions, ou de le perdre lors d'une désinstallation ou réinstallation.

Raccourci ou favori

Ajouter un raccourci sur le Bureau ou créer un favori dans l'explorateur Windows pour cette documentation peut se révéler très pratique. Le lecteur pourra ainsi naviguer facilement dans l'explorateur pour y rechercher ces classes ou autres méthodes.

Le raccourci suivant sera sans aucun doute le préféré des programmeurs Java : `C:/JavaCpp/jdk-6-doc/docs/api/index.html`.

En effet, il représente la page d'accès pour naviguer dans toutes les classes distribuées avec le JDK 1.6 (figure B-27) :

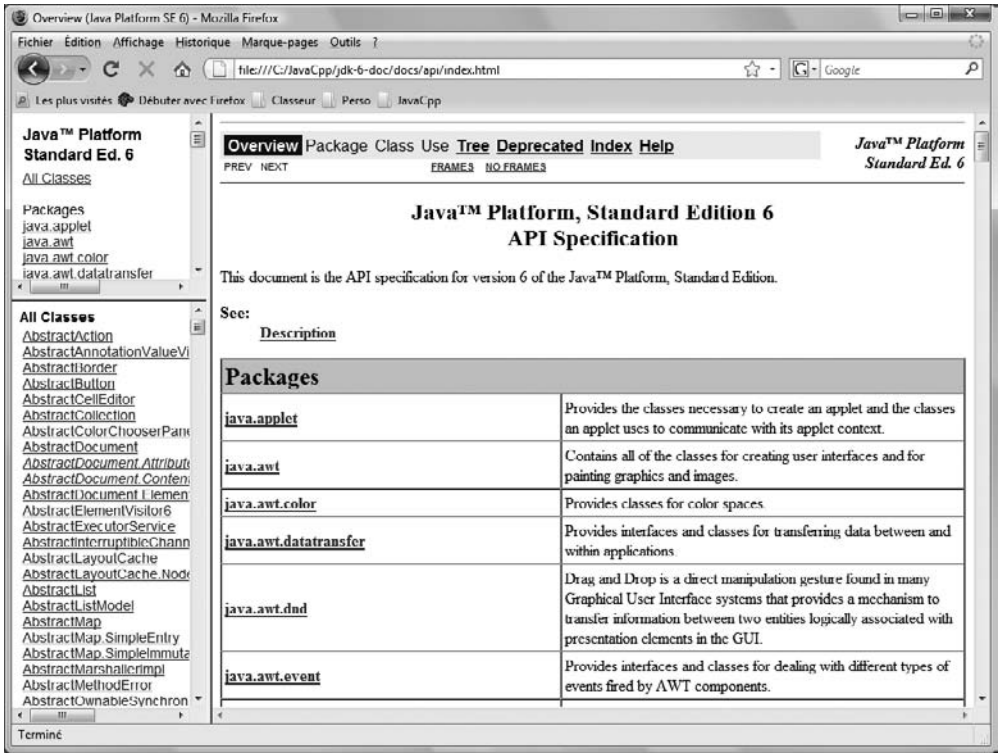

Figure B-27

Index de la documentation du JDK 1.6

Nous utilisons ici le navigateur Firefox, mais Internet Explorer (IE) convient tout aussi bien.

Les deux cadres situés à gauche de la figure B-27 nous permettent de naviguer dans les packages, par exemple, l'AWT. Si nous désirons examiner uniquement les classes de l'AWT, nous sélectionnerons alors le lien du package `java.awt`. Si nous cherchons la classe `String`, plusieurs méthodes sont possibles :

- En utilisant la barre de navigation verticale du cadre situé en bas à gauche : la liste des classes étant triée par ordre alphabétique, il nous suffira de descendre dans la liste jusqu'à l'entrée `String`.
- Nous pouvons aussi choisir `Index`, dans le menu du haut du cadre principal, et sélectionner la lettre `S`. Une longue liste de variables, de méthodes ou de constructeurs nous

est alors présentée. La barre verticale de défilement, dans le cadre de gauche « All Classes », est plus difficile à manipuler car il y a beaucoup de sélections pour le `S` avant d'atteindre la classe `String`. Cette manière de faire avec l'`Index` est plus pratique pour atteindre directement un constructeur ou rechercher une méthode dont on ne se rappelle que le nom (par exemple, `parseInt()`).

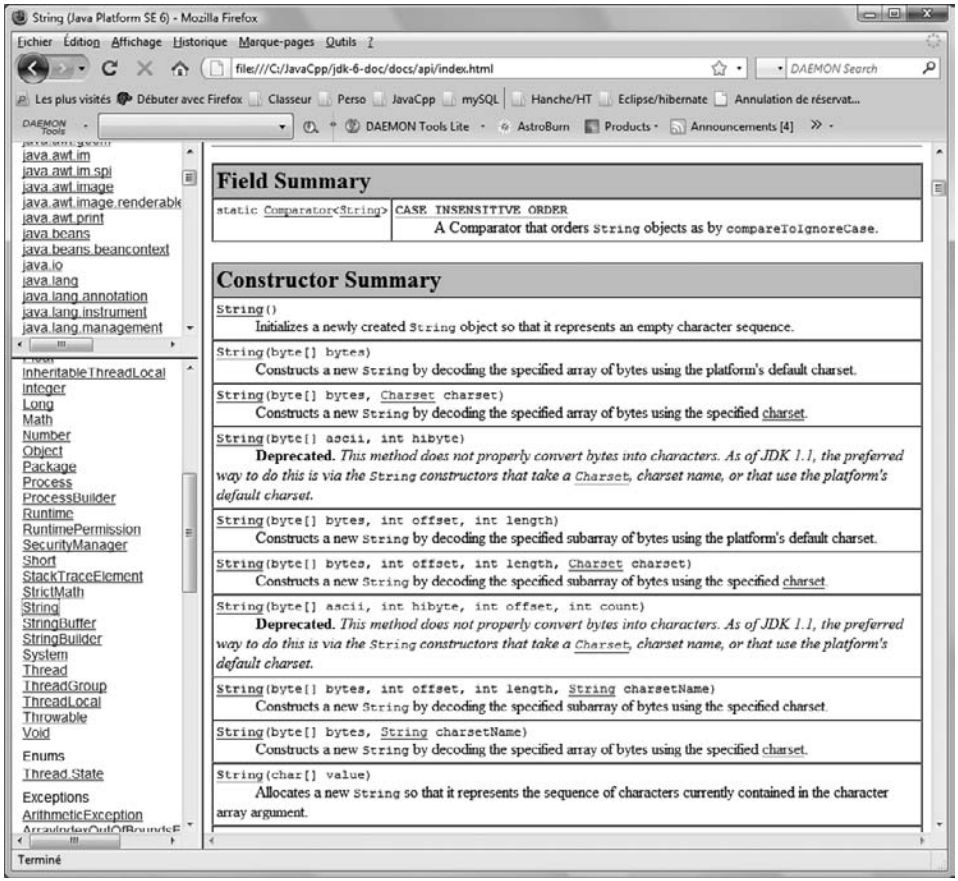

Figure B-28

Documentation de la classe String

- Nous savons peut-être déjà que `String` est dans le package `java.lang`. Si nous sélectionnons ce dernier dans le cadre situé en haut à gauche (vers la fin du premier tiers), nous remarquons que la liste du cadre situé en bas à gauche est considérablement réduite et nous atteindrons très rapidement la classe `String` (figure B-28). Pour afficher à nouveau la liste complète, il suffira de cliquer sur le lien `All Classes` du cadre situé en haut à gauche.

Évidemment, lorsque nous nous trouvons dans une classe comme `String`, nous pouvons continuer notre navigation à l'infini, à la recherche de nouveautés.

Nous y trouverons, par exemple, une référence à la fameuse classe `StringBuffer` que nous avons rencontrée au chapitre 5, car notre `String` est immuable. Un simple clic sur ce lien fera notre bonheur !

Nous n'oublierons pas non plus la référence Web suivante :

`http://java.sun.com/javase/6/docs/`

qui nous fournira une description détaillée des différents sujets à disposition dans la documentation (figure B-29) :

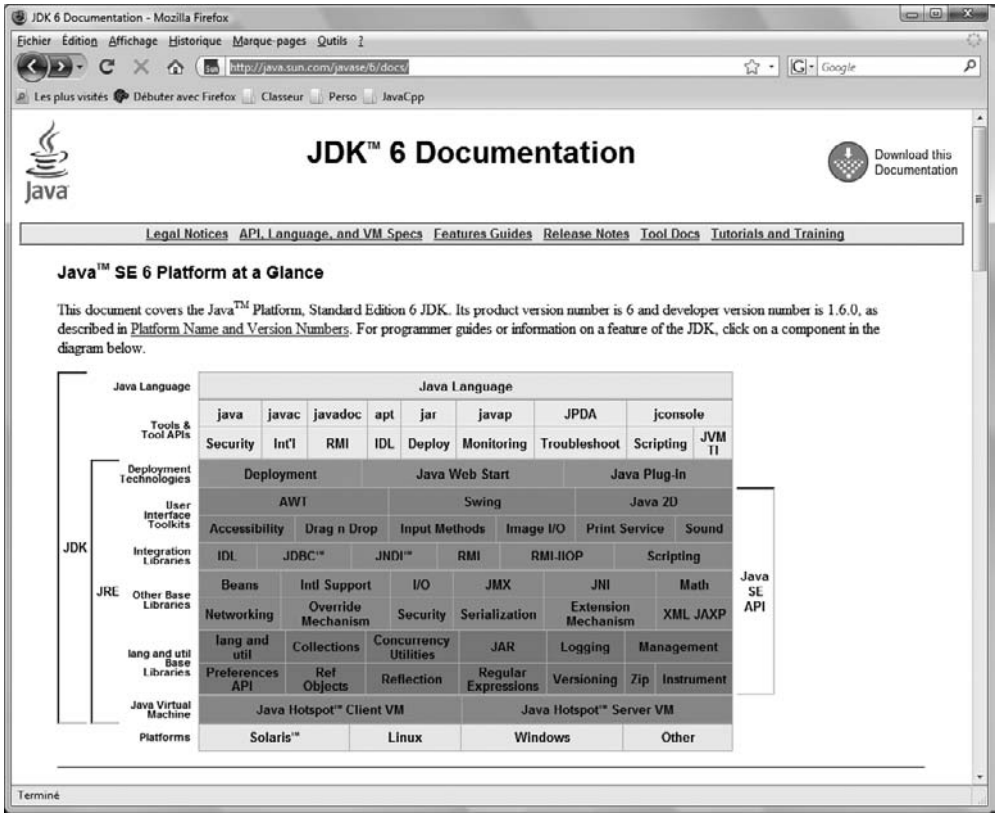

Figure B-29

Les outils et la documentation du JDK

Seuls certains de ces documents, comme la documentation API du JDK nécessaire pour cet ouvrage, ont été inclus sur le CD-Rom. En cas de nécessité, nous trouverons sur ce site les descriptions complètes des outils comme `java`, `javac`, `javadoc` et bien d'autres (figure B-30) :

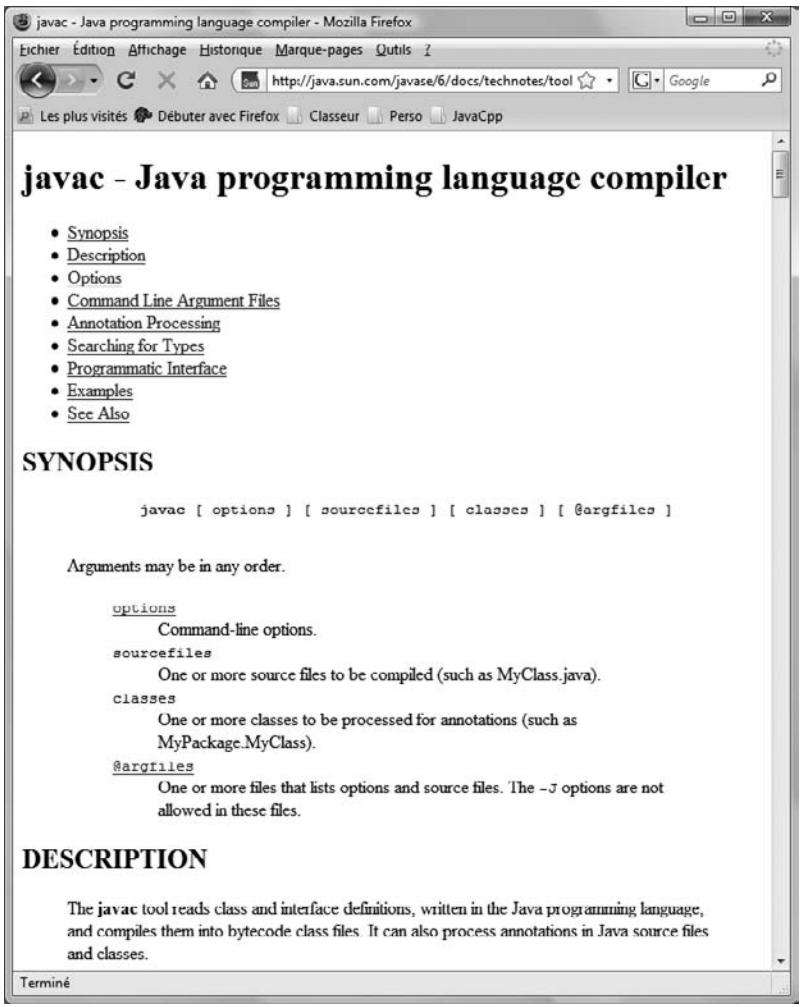

Figure B-30

La documentation de javac.exe

Ces pages nous permettront de découvrir toute la documentation disponible sur les technologies et outils distribués avec le JDK 1.6. Nous y trouverons non seulement la description de nombreux paramètres (comme `-classpath`, `-Xlint` ou encore `-deprecation`), mais aussi les variantes possibles sur d'autres systèmes d'exploitation.

MinGW et MSYS sur Internet

Ces outils et paquets de logiciels ont été préparés et préinstallés par l'auteur afin d'en faciliter la configuration. Cette section figure ici uniquement à titre d'information.

Si le lecteur désire exécuter cette procédure ou tester de nouvelles versions, nous lui donnerons ici quelques références.

Pour commencer, nous nous rendrons sur le site `http://www.mingw.org/` (figure B-31).

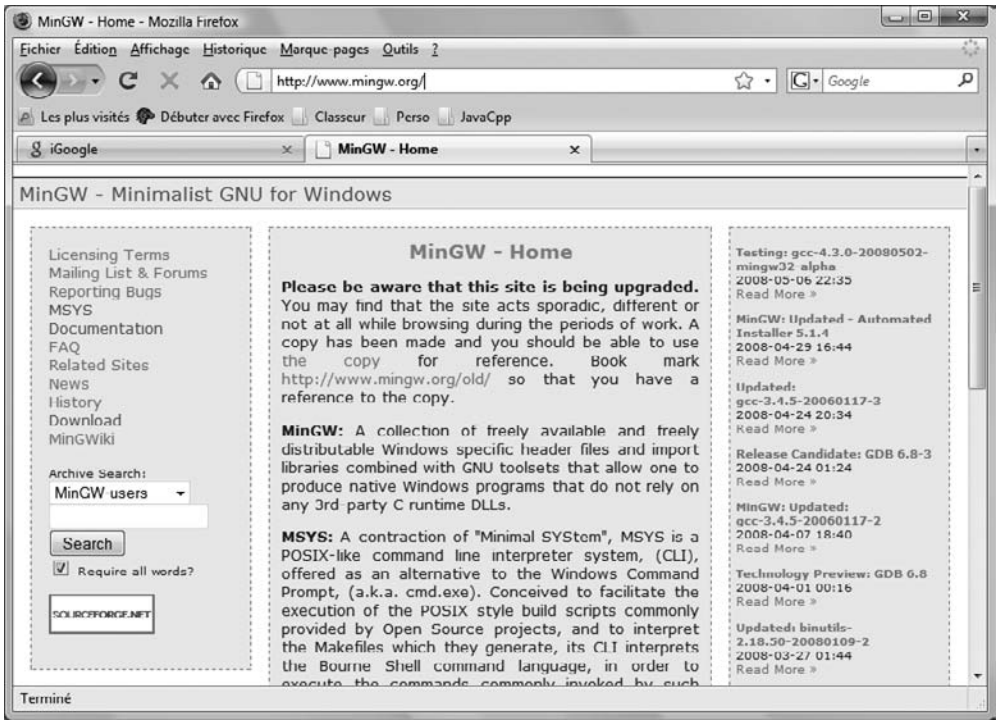

Figure B-31

Le site officiel de MinGW

Nous y découvrons la définition complète de MinGW que nous ne traduirons pas ! En cliquant sur le lien Download, nous pourrons télécharger la dernière version de MinGW. Il faudrait alors vérifier que l'environnement avec l'éditeur Crimson ou avec NetBeans fonctionne correctement. Il est recommandé de consulter plus particulièrement les liens FAQ, MSYS ou Documentation.

Il est également possible de télécharger MinGW à partir du site Sourceforge.net (`http://sourceforge.net/projects/mingw/`), puis cliquons sur le lien Download de cette page. Il faudra ensuite télécharger les différents paquets de distribution. Ces derniers seront pour la plupart des outils de téléchargement qui vont continuer la procédure automatiquement. En fin de compte, il s'agira simplement de les installer dans des répertoires comme `C:\MinGW` et `C:\msys`. Le `make` et le `gdb` (GNU débogueur) sont sans doute les plus délicats : ils sont préinstallés ici et fonctionnent aussi dans NetBeans (voir annexe E).

Nous pouvons connaître la version de `g++` et `make` avec les commandes :

```
g++ -v
make -v
```

Pour `g++`, la version 3.4.5 de MinGW au minimum devrait être utilisée pour les exemples et exercices en C++ de cet ouvrage. Pour `make`, comme nous utilisons des fonctions simples et standards, c'est moins important, mais la version 3.81 utilisée dans cet ouvrage nous donnerait une référence en cas de difficultés (voir la section suivante). Sous Linux (voir annexe F), ces versions sont en général supérieures et plus solides.

Problèmes potentiels avec le make et MSYS

Cette partie n'est utile que pour les lecteurs qui installent eux-mêmes MinGW ou MSYS. Ces informations sont évidemment disponibles sur Internet, mais de manière dispersée. En cas de difficulté, le lecteur peut s'adresser à l'auteur ou essayer les diverses solutions décrites ci-après.

Voici un exemple de problème rencontré : lors des tests effectués sous Windows Vista et après quelques semaines de travail, le `make` dans Crimson s'est mis à ne plus fonctionner pour une raison inconnue et sans qu'il soit possible de corriger le problème simplement. Le message suivant revenait à chaque fois :

```
AllocationBase 0x0, BaseAddress 0x715B0000, RegionSize 0x330000, State 0x10000
C:\msys\1.0\bin\make.exe: *** Couldn't reserve space for cygwin's heap, Win32 error 0
```

Pour résoudre ce problème, nous avons renommé le fichier `make.exe` du répertoire `C:\msys\1.0\bin` en `make1.exe`. Dans le répertoire `C:\MinGW\bin`, nous avons ensuite copié le fichier `mingw32-make.exe` en `make.exe` (copié et non renommé). Suite à ces opérations, l'exécution de la commande `make` sous DOS appelait bien le `make` de MinGW et non celui de MSYS. Dans Crimson (voir annexe C) le chemin d'accès complet du `make` est défini, donc utilisé correctement.

Lors d'un `make clean` à partir de Crimson, ce problème s'est à nouveau présenté, mais plus tard. Nous avons pu le corriger en remplaçant le fichier `msys-1.0.dll` par une version plus récente dans le répertoire `C:\msys\1.0\bin`. Nous avons conservé celui d'origine et l'avons renommé en `msys-1.0_old.dll`.

Cette configuration, avec ces corrections, correspond à l'installation à partir du CD-Rom. Ces problèmes pourraient se manifester à nouveau si le lecteur faisait lui-même une installation complète de MinGw et/ou de MSYS à partir d'Internet.

Les outils Linux de MSYS

Pour ceux qui sont intéressés par les outils Linux pour PC de MSYS, nous allons donner ici un bref aperçu des possibilités qu'ils offrent. Ils peuvent s'avérer très utiles pour les programmeurs qui doivent, par exemple, rechercher de l'information ou écrire de petits programmes. Ils sont de plus bien souvent plus efficaces que les outils DOS.

Nous allons également donner quelques exemples simples et pratiques, dans un environnement Windows Vista, afin d'aiguiser l'esprit du lecteur. Il pourra ainsi y découvrir de nouvelles ressources de travail et de compétences.

Dans l'annexe F, « Apprendre Java et C++ avec Linux », nous avons fourni d'autres références, que nous conseillons de consulter en parallèle, pour la version Ubuntu de Linux.

La commande msys.bat

La commande `msys.bat` du répertoire `C:\msys\1.0`, ouvre une fenêtre qui ressemble étrangement à une fenêtre DOS ou à une console Linux (figure B-32) :

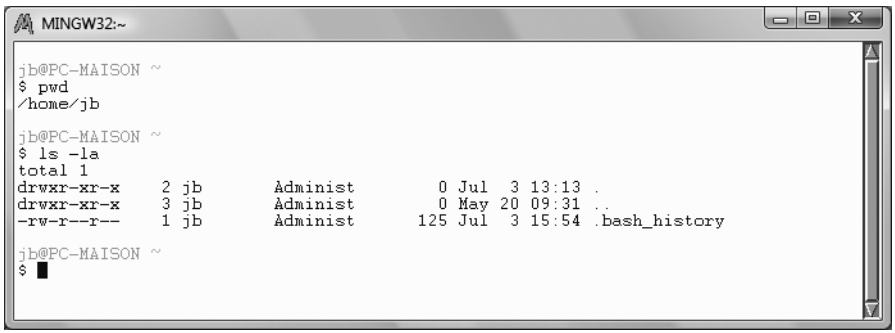

Figure B-32

bash Mingw console

Nous sommes presque dans un environnement Linux, c'est une console bash shell (figure B-33) :

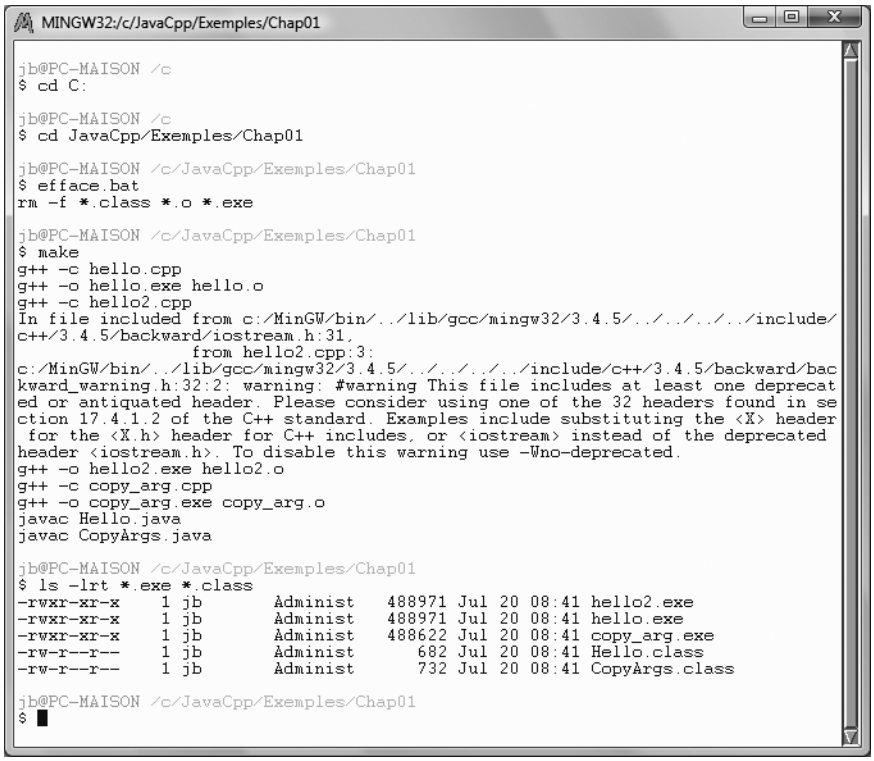

Figure B-33

bash – Quelques exemples

Avec la commande `cd`, nous pouvons nous positionner sur le bon disque, la bonne partition et le répertoire souhaité. Nous voyons que la commande `efface.bat` fonctionne car elle n'utilise que la commande `make`. Il n'y a pas de commandes DOS internes telles que `del` ou `pause` qui ne seront pas reconnues. La commande `make` donne évidemment le même résultat que sous DOS. La commande `ls` est décrite ci-après et affiche les fichiers créés et triés par date.

Pour revenir au `make`, si nous avions un doute sur son installation, nous pourrions entrer la commande :

```
which make
```

qui nous retournerait le résultat suivant :

```
/c/MinGW/bin/make
```

La commande `which` indique le chemin d'accès du `make`. La commande `set` nous retournera le `PATH` des commandes et nous verrons qu'il correspond à Windows. Dans cette console Linux, nous pourrions entrer la commande `notepad` (figure B-34).

L'application Windows Notepad serait alors exécutée et présentée à l'écran, mais la console bash resterait bloquée et en attente de la terminaison du Notepad. Avec la forme `notepad &`, la console ne serait pas bloquée et indiquerait le numéro du processus `notepad` lancé en arrière-plan, ici le 3764. Nous sommes vraiment dans le monde Linux où nous pourrions « tuer à la main » le processus `notepad` (avec la commande `kill -9`).

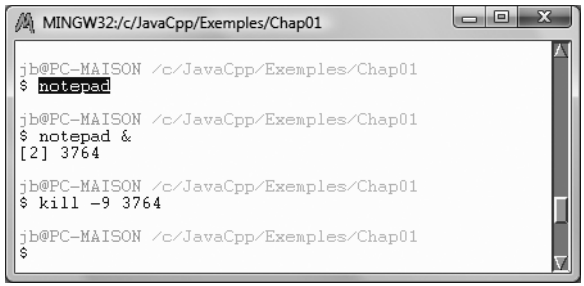

Figure B-34

Commandes notepad et kill

Sur cette capture, nous avons surligné le premier `notepad` avec la souris afin de pouvoir le copier à l'extérieur dans Windows (Ctrl+V) ou dans cette fenêtre en cliquant sur la molette de la souris.

Il y a beaucoup de références sur le Web pour les *man pages* de Linux qui nous donnent la description complète de commandes comme `kill` ou celles décrites dans les section suivantes (`cd`, `ls` ou encore `pwd`).

```
http://fr.wikipedia.org/wiki/Man_(commande_Unix)
http://jp.barralis.com/linux-man/man1/ls.1.php
http://jp.barralis.com/linux-man/lire.php?man=man1
```

Dans cette dernière, il faudra sélectionner les commandes Linux disponibles le répertoire `C:\msys\1.0\bin` : une partie seulement des nombreuses commandes Linux fait partie de la distribution MSYS sous Windows.

La commande cd

La commande `cd` est similaire à celle du DOS. Elle nous permet de nous déplacer directement dans un répertoire connu, comme les exemples de ce livre et particulièrement ceux du chapitre 1.

```
cd C:/JavaCpp/EXEMPLES/Chap01
```

Nous remarquerons que nous utilisons ici le caractère / (Linux) au lieu du caractère \ (Windows). Les commandes `cd ..` et `cd ../Chap02`, nous permettront, respectivement, de remonter d'un niveau ou de passer directement au deuxième chapitre.

Avec `cd ../Chap` suivi du caractère de tabulation (Tab), nous obtenons une liste de toutes les possibilités de dossiers dont le nom commence par Chap, se trouvant à ce niveau (Ctrl+L nous permet d'effacer le contenu de la console). Il suffira d'entrer 22, par exemple, pour sélectionner le chapitre 22 (figure B-35) :

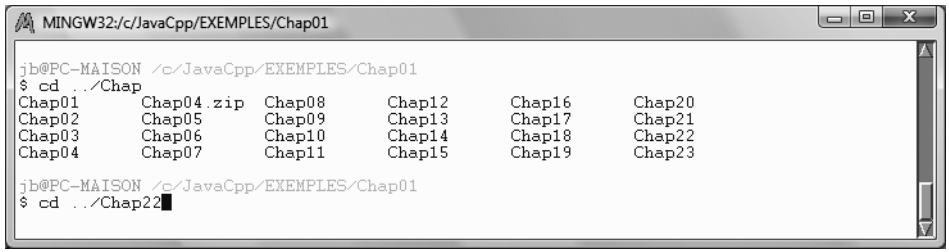

Figure B-35

Recherche avec Tab sous bash

Les commandes ls et pwd

La commande `ls` est très similaire à la commande `dir` du DOS. Nous commencerons par sa forme la plus simple (nous sommes dans le répertoire `C:\JavaCpp\EXEMPLES`, figure B-36) :

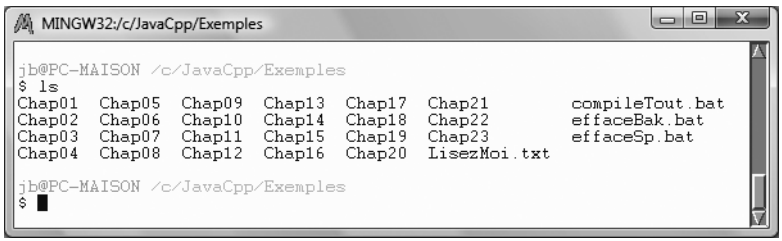

Figure B-36

La commande ls de Linux

La commande `ls` affiche uniquement les noms de fichiers et de répertoires, sans aucune autre indication. Nous emploierons donc la forme la plus utilisée et la détaillée, `ls -lrt` (figure B-37) :

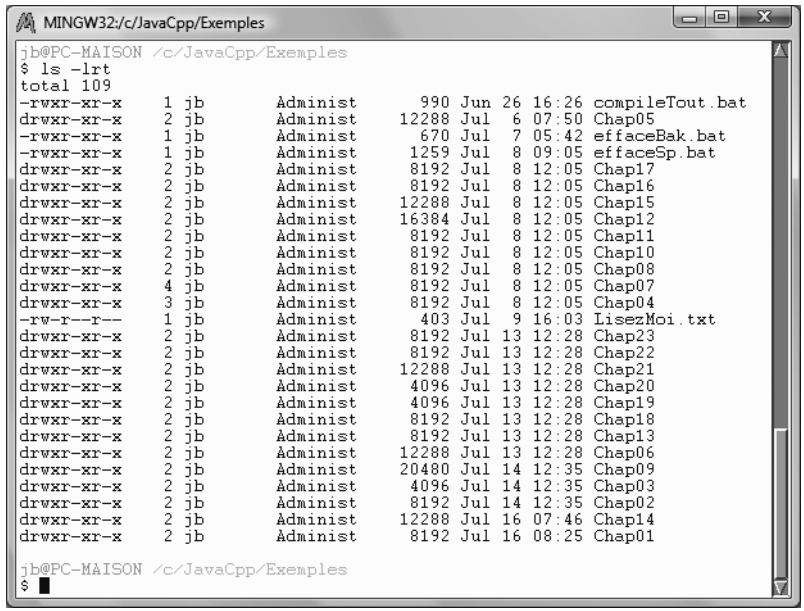

Figure B-37

La commande ls étendue

Le `d` initial indique que nous avons affaire à des répertoires, sinon nous aurions un simple tiret (`-`), comme pour le fichier `LisezMoi.txt` dans la figure B-37. Les autres lettres correspondent aux permissions des fichiers (`r` pour *read*, `w` pour *write* et `x` pour *executable*), distribués en trois groupes (utilisateur, groupe et tous). Dans cet exemple, seul l'administrateur, ou l'utilisateur avec ces mêmes droits, peut modifier ou effacer ces fichiers et répertoires.

L'option `-r` nous permet de trier par date en commençant par le plus ancien : ici le `Chap01` a été modifié récemment, il apparaît donc en fin de liste.

Nous ne donnerons pas plus de détails : nous sommes dans le monde Linux et un bon ouvrage sur le sujet pourrait aider le lecteur (pour d'autres références, voir aussi l'annexe F « Apprendre Java et C++ avec Linux »).

Copie dans un fichier

Nous pouvons obtenir le résultat d'une commande dans un fichier et non plus à l'écran :

```
ls -l */* >tous_les_fichiers.txt
```

La séquence `*/*` indique que nous allons extraire la liste d'informations de tous les fichiers au deuxième niveau du répertoire. Le résultat sera ensuite transféré dans le fichier texte `tous_les_fichiers.txt`.

Emploi du pipe

Il est possible de chaîner des commandes Linux :

```
ls -lt */* | egrep ".java" | egrep In
```

Le paramètre `t` de `-lt` permettra de trier les fichiers de manière à ce que ceux qui ont été modifiés en dernier apparaissent en début de liste.

Le résultat sera passé à la commande `egrep`, qui va filtrer les fichiers afin de ne retenir que ceux contenant `.java`. Les guillemets (`""`) du `egrep` peuvent s'avérer nécessaires pour les noms de fichiers comportant des espaces ou des caractères spéciaux.

Enfin, un deuxième filtre `egrep` va sortir seulement les fichiers contenant `In`. En restant dans le répertoire de travail `C:\JavaCpp\EXEMPLES`), le résultat sera (figure B-38).

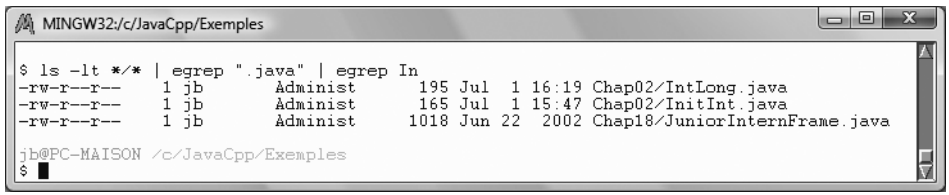

Figure B-38

Le pipe de Linux – Plusieurs commandes chaînées

On peut inverser le filtre d'`egrep` avec `-v` pour recevoir toutes les lignes qui ne possèdent pas `Chap01` :

```
ls | egrep -v Chap01
Chap02
Chap03
Chap04
```

Awk, l'un des outils essentiels de Linux

La commande `awk` est un autre filtre encore plus puissant qu'`egrep`. Nous allons présenter quelques exemples de son utilisation, en commençant par :

```
ls -lt */*.java | awk '{ print $9 " " $5 }'
```

Une commande `awk` pouvant s'avérer complexe, nous avons la possibilité de l'éditer avec Crimson dans notre répertoire de travail (nous sommes toujours dans `C:\JavaCpp\EXEMPLES`).

Dans le cas présent, nous avons créé un fichier `monAwk.sh` (`.sh` est l'extension habituelle des scripts shell sous Linux) contenant cette commande. Nous pouvons maintenant l'exécuter (figure B-39) :

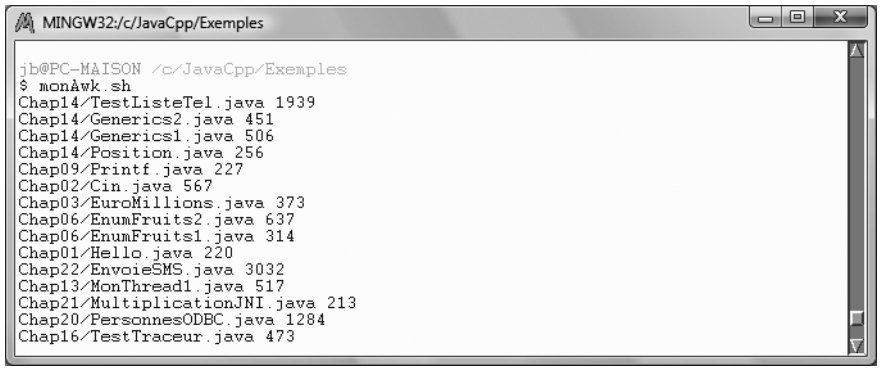

Figure B-39

La fameuse commande awk de Linux

Nous n'entrerons pas trop dans les détails de cette commande, qui est en fait un véritable langage de programmation. Nous indiquerons simplement que ce script va extraire les colonnes 9 et 5 du résultat de la commande `ls -lt` et insérer un espace entre les deux. Le fichier `TestListeTel.java` apparaît en début de liste, car les fichiers sont triés par date de modification, et le `1939` (la colonne 5), correspond à la taille en octets du fichier.

Un script de sauvegarde

Nous pouvons écrire un fichier script de sauvegarde pour tous les fichiers Java de la manière suivante :

```
ls */* | egrep ".java" | awk '{ print "cp " $1 " " $1 ".bk1"}' > backup.sh
```

Ce script peut se révéler très utile car `backup.sh` contient ceci :

```
cp Chap01/CopyArgs.java Chap01/CopyArgs.java.bk1
cp Chap01/Hello.java Chap01/Hello.java.bk1
cp Chap02/Cin.java Chap02/Cin.java.bk1
cp Chap02/Feux.java Chap02/Feux.java.bk1
cp Chap02/InitInt.java Chap02/InitInt.java.bk1
cp Chap02/IntLong.java Chap02/IntLong.java.bk1
cp Chap02/MathTest.java Chap02/MathTest.java.bk1
… etc.
```

Nous pouvons aussi compresser des fichiers avec :

```
tar cvfj sources.tar.bzip2 `ls */* | egrep ".java"`
```

L'option `cvfj` de la commande `tar` de Linux, va nous créer une archive d'un ensemble de fichiers compressée avec le codage `bzip2`. Le `ls */* | egrep ".java"` étant inclus entre les ``, il sera exécuté en premier avant d'être passé à la commande `tar`. La liste produite avec `ls` contiendra tous les fichiers `.java` contenus dans le répertoire du second niveau.

7-zip, que nous avons installé en début d'annexe, peut également lire des fichiers `.bzip2` et `.tar`. Pour ouvrir, par exemple, le fichier `sources.tar.bzip2` avec 7-Zip, il suffit d'effectuer un clic droit dessus depuis l'explorateur Windows et de sélectionner Open archive (figure B-40) :

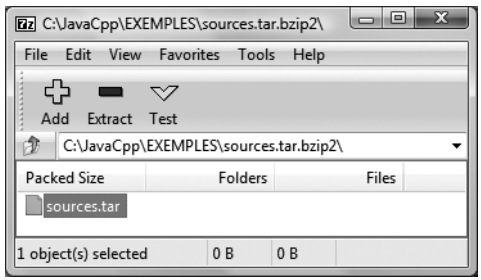

Figure B-40

Le fichier sources.tar.bzip2 contient le fichier sources.tar

Il suffit ensuite de cliquer sur `sources.tar` (aussi reconnu par 7-Zip) pour ouvrir le contenu de l'archive `.tar` de Linux (figure B-41) :

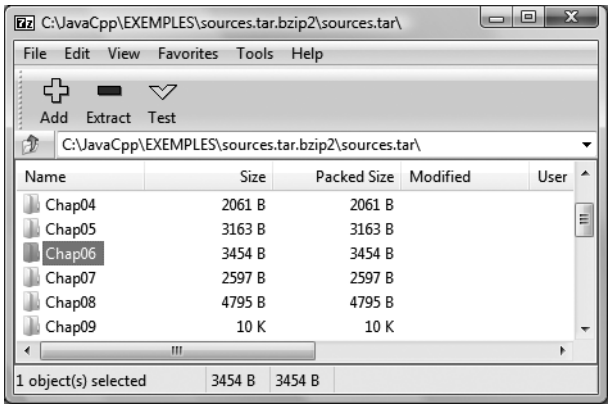

Figure B-41

Contenu de l'archive sources.tar

7-Zip nous permet de naviguer dans cette archive et de retrouver tous les fichiers Java que nous avons sauvegardés, ici pour le chapitre 6 (figure B-42) :

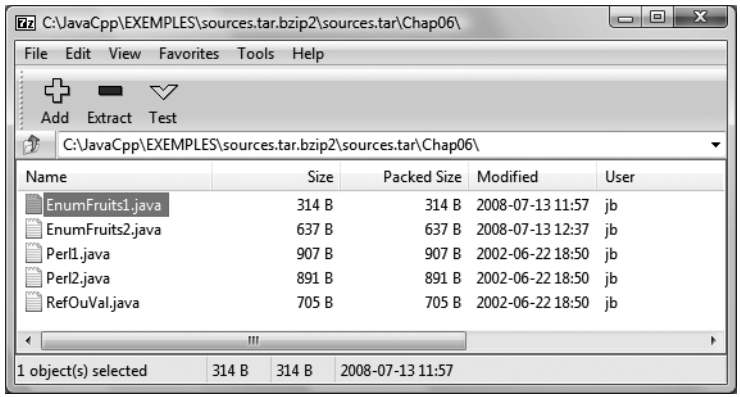

Figure B-42

Fichiers .java du chapitre 6

Dans le nom du fichier, avec la forme ` `, nous pourrions aussi intégrer la date et l'heure du fichier afin d'effectuer une sauvegarde datée :

```
date | awk `{ print $2 $3 "-" $4 }` | awk `{ print substr($1, 1, 8)  substr($1,10,2)} `
```

Cependant, cette commande ne fonctionne pas toujours à cause du format de la date à certaines heures ! À vous de le découvrir et de le corriger !

C

Installation et utilisation de Crimson

L'éditeur Crimson est un éditeur simple, convivial et particulièrement approprié pour une utilisation dans le cadre de ce livre.

Les exemples et exercices en Java, C++ ou C#, basés sur du code source peu volumineux et des compilations au travers de `Makefile` (voir chapitre 1 et annexe B), sont tout à fait adaptés à l'éditeur Crimson.

Cette annexe décrit les configurations nécessaires aux différents outils de compilation et d'exécution. La quantité d'informations mises à disposition ici devrait permettre de résoudre ou de pallier les problèmes ou difficultés qui pourraient survenir suivant la version de Windows et sa configuration. L'installation de Crimson ne devrait pas prendre plus de 5 minutes et elle peut se résumer en 3 points :

1. Installer Crimson à partir du CD-Rom.
2. Installer les menus (section « Configuration préparée »).
3. Vérifier les autres aspects (associations et démarrage) en fonction de la version de Windows utilisée afin de définir comment le lecteur utilisera finalement l'éditeur Crimson.

Crimson est un logiciel Open Source gratuit pouvant être installé sur toutes les versions du système d'exploitation de Microsoft, de Windows 98 à Windows Vista. La présentation dans cette annexe a été faite sous Windows Vista, mais l'installation de Crimson sous Windows XP est identique.

Site Web de Crimson

Les dernières versions officielles (3.70) et bêta (3.72) de l'éditeur Crimson datent, respectivement, de septembre 2004 et mai 2008.

Nous donnerons ici deux URL de sites Web que nous jugeons intéressants :

```
http://www.crimsoneditor.com
http://www.emeraldeditor.com
```

La première correspond au site officiel de Crimson, la seconde permet d'accéder à des versions bêta (3.72 à ce jour). Une version Emerald semble voir le jour et serait une nouvelle version Open Source de Crimson.

La version 3.72 bêta 241 fonctionne mais n'a pas été considérée ici : sous Windows Vista, il n'a pas été possible de déplacer avec la souris un fichier de l'explorateur vers l'éditeur.

Installation

Réinstallation de Crimson

Si nous souhaitons réinstaller Crimson ou installer une nouvelle version dont les fichiers de configuration resteront compatibles avec l'ancienne, nous devrons être très attentifs. Il nous faut donc vérifier que Crimson n'est pas déjà installé avant de passer à l'installation ; ce sera le cas pour les lecteurs des éditions précédentes de cet ouvrage.

Il nous est arrivé d'essayer cette démarche sous Windows Vista et nous avons été obligés d'effacer des entrées dans les registres de Windows et de lancer une installation dans un répertoire moins sensible que `C:\Program Files` (avec demande d'autorisation). C'est pour cette raison que nous avons utilisé le répertoire d'installation sous `C:\JavaCpp`.

Sous Windows XP, si nous avons déjà installé ou mal installé/désinstallé Crimson, nous pourrions recevoir le message de la figure C-1 :

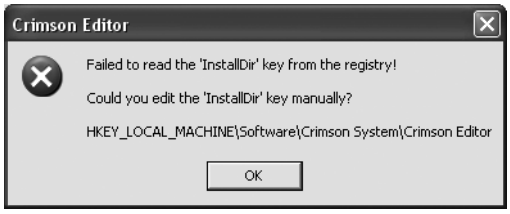

Figure C-1

Windows XP – Message d'erreur s'affichant car Crimson est déjà installé.

Ceci indiquerait que Crimson utilisera un autre chemin d'accès pour les menus inclus dans le répertoire `INSTALL\CrimsonPref` du CD-Rom. Dans ce cas, il faudrait refaire une installation complète ou encore entrer les menus manuellement.

Configuration préparée

Dans le répertoire `INSTALL\CrimsonPref` situé à la racine du CD-Rom, nous avons inclus les fichiers de configuration que le lecteur pourra copier directement dans le répertoire d'installation proposé, soit `C:\JavaCpp\Crimson Editor`.

Cette copie doit absolument se faire après l'installation de Crimson. Elle a été préparée par l'auteur et devrait pouvoir fonctionner sans qu'il soit nécessaire d'installer tous les menus décrits dans la suite de cette annexe (un long travail manuel, rébarbatif et source d'erreurs).

Après l'installation de Crimson, la copie des fichiers depuis le répertoire `INSTALL\CrimsonPref` du CD-Rom et après avoir relancé l'éditeur, nous devrions retrouver tous les menus tels que présentés sur la figure C-13.

Installation à partir du CD-Rom

Nous allons décrire ici l'installation de la version 3.70 de Crimson avec l'installateur de Windows. Comme indiqué précédemment, il n'est pas conseillé d'installer la version 3.72 bêta, sauf si celle-ci a été améliorée depuis nos tests et résout le problème mentionné de déplacement de fichiers avec la souris.

Le programme d'installation de la version 3.70 se trouve dans le répertoire `INSTALL` du CD-Rom d'accompagnement.

> **Attention !**
> Il peut s'avérer nécessaire, en cas de difficultés d'installation, de copier les fichiers du CD-Rom dans un répertoire de travail temporaire (par exemple, `C:\TEMP`) avant de procéder à l'installation.

Il suffit de double-cliquer sur l'exécutable `crimson_editor_3.70.exe` pour démarrer l'installation.

Sous Windows Vista, nous aurons encore la demande traditionnelle d'autorisation (figure C-2) :

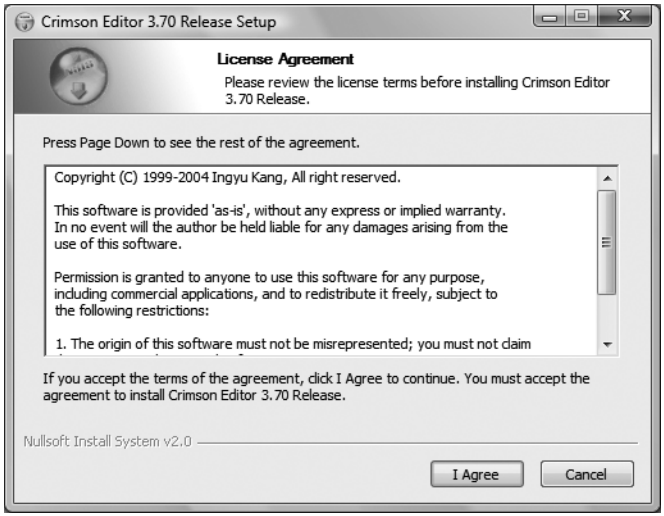

Figure C-2

Démarrage de l'installation

Nous accepterons les termes de la licence et cliquerons sur le bouton I Agree, ce qui aura pour effet d'ouvrir la fenêtre présentée à la figure C-3.

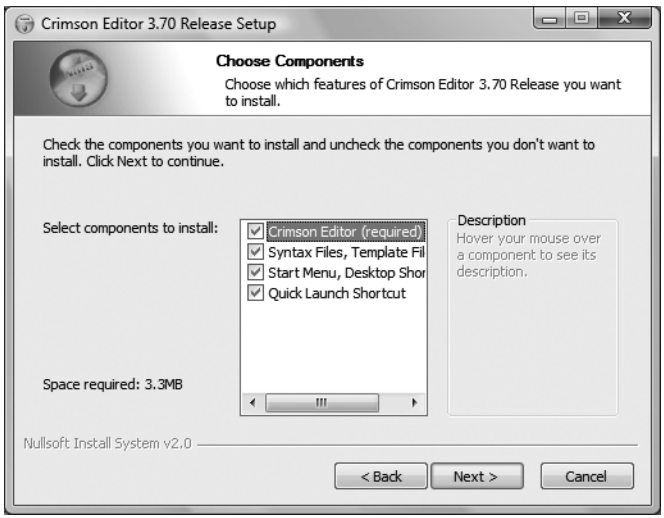

Figure C-3

Sélection des composants à installer

Nous choisirons tous les composants et continuerons l'installation en cliquant sur le bouton Next. La fenêtre de la figure C-4 apparaîtra alors, nous demandant de sélectionner un répertoire d'installation :

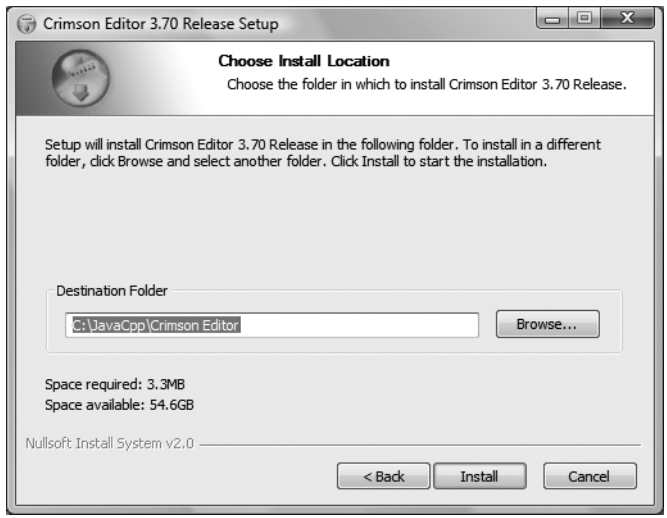

Figure C-4
Répertoire d'installation

Le répertoire `C:\JavaCpp\Crimson Editor` est le répertoire recommandé, particulièrement sous Windows Vista, si nous désirons reprendre la configuration des outils préparée par l'auteur (voir la section précédente « Configuration préparée »). Pour lancer l'installation, il suffira ensuite de cliquer sur le bouton Install.

Association des fichiers à Crimson dans l'explorateur

Pour une utilisation aisée de Crimson, il faudrait associer les fichiers `.java`, `.cpp` et `.h` à l'éditeur Crimson. Malheureusement, cette procédure peut s'avérer plus délicate sous Windows Vista que sous Windows XP. Les prochaines versions de Crimson pourraient corriger ce problème (voir la section « Demande d'autorisation sous Windows Vista » ci-après).

Sous Windows XP (ou éventuellement Windows Vista), nous pouvons associer les fichiers `.java`, `.cpp` et `.c` via le menu Tools > Preferences > File > Association de l'éditeur Crimson (voir figure C-10). Dans le champ Associated, il faudra retrouver `C:\JavaCpp\Crimson Editor\cedt.exe "%1"`.

Avec l'explorateur de Windows, nous montrons ici un exemple sous Windows XP (version anglaise) avec le fichier `Personne.h` situé dans le répertoire `C:\JavaCpp\EXEMPLES\Personne.h` (figure C-5) :

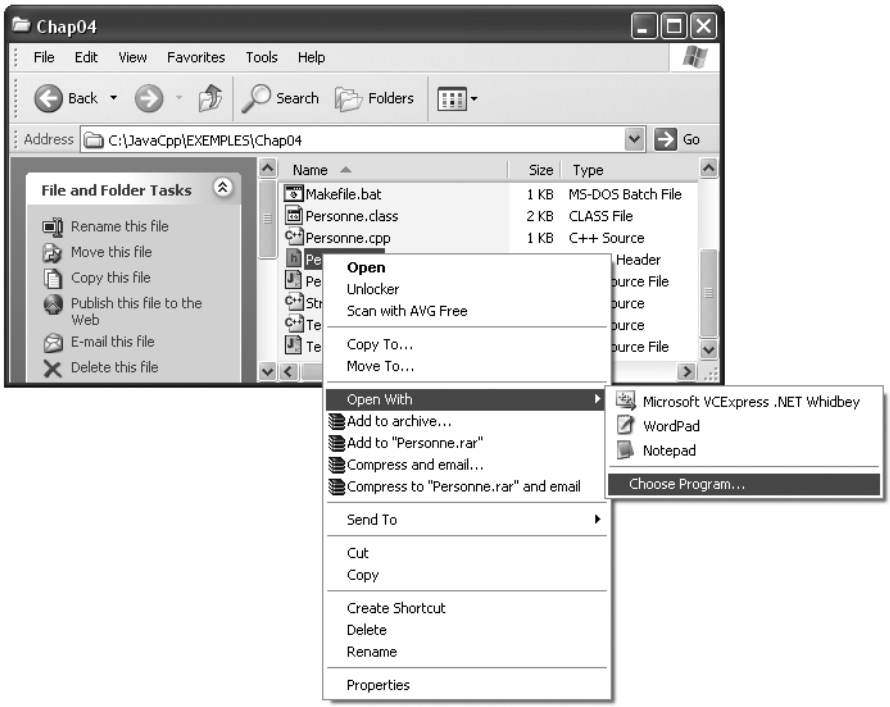

Figure C-5

Association avec les fichiers source

L'éditeur Crimson correspond au programme `cedt.exe` dans le répertoire d'installation de Crimson (`C:\JavaCpp\Crimson Editor`). L'entrée Choose Program… (choisir le programme) du menu contextuel nous permettra de rendre l'association effective. Normalement, Crimson se trouvera dans la liste présentée, sinon nous devrons le rechercher en cliquant sur le bouton Browse. Il ne faut pas oublier de cocher l'option Always use the selected... : c'est grâce à elle que Crimson est automatiquement lancé lors d'un double-clic sur un fichier `.h`, `.cpp` ou `.java` dans l'explorateur de Windows.

La procédure décrite ci-dessus est à répéter pour les extensions `.cpp` et `.java`.

Lorsqu'un programme est associé de cette manière, nous pourrons ensuite double-cliquer sur le fichier ou utiliser le bouton droit de la souris et sélectionner Ouvrir avec dans le menu contextuel. Pour les fichiers `Makefile`, dans la mesure où ils ne comportent pas d'extension, il faudra les ouvrir via le menu File > Open ou les déplacer depuis l'explorateur de Windows avec la souris.

Installation d'un raccourci

De la même manière que nous avons ajouté un raccourci sur le Bureau pour notre répertoire de travail `JavaCpp` (voir annexe B, section « Vérification de l'installation des outils »), il est aussi conseillé d'y installer un raccourci vers l'éditeur Crimson. Pour cela, il suffit d'effectuer un clic droit sur le Bureau et de sélectionner Nouveau > Raccourci dans le menu contextuel.

La figure C-6 présente les propriétés de ce raccourci :

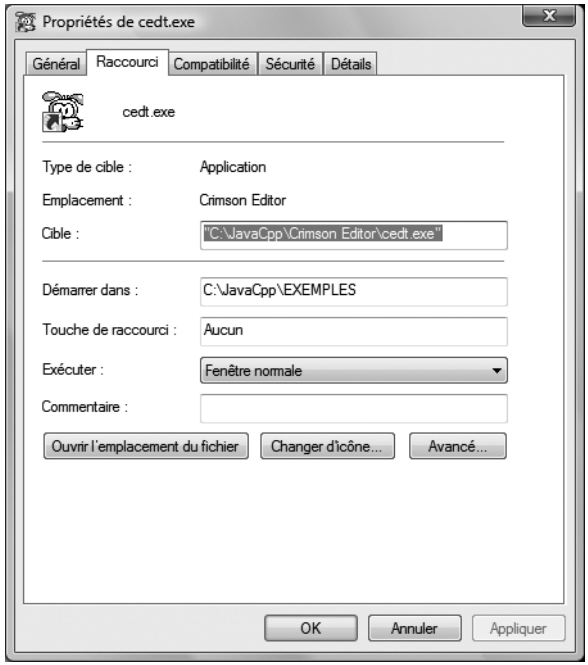

Figure C-6
Propriétés du raccourci vers l'éditeur Crimson

Le répertoire de démarrage n'est pas très important, car il sera modifié et conservé à chaque redémarrage du logiciel. Si, par exemple, nous travaillons dans le chapitre 13, un File > Open (ouvrir un fichier) nous y conduira à nouveau.

Premier démarrage de Crimson

Lors d'une installation ou réinstallation, en particulier sous Windows XP, la fenêtre de la figure C-7 peut apparaître : nous l'ignorerons.

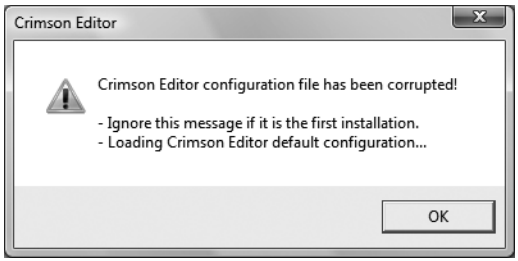

Figure C-7

Message à ignorer

Sous Windows Vista seulement, et au démarrage, nous pourrions voir apparaître l'alerte de la figure C-8.

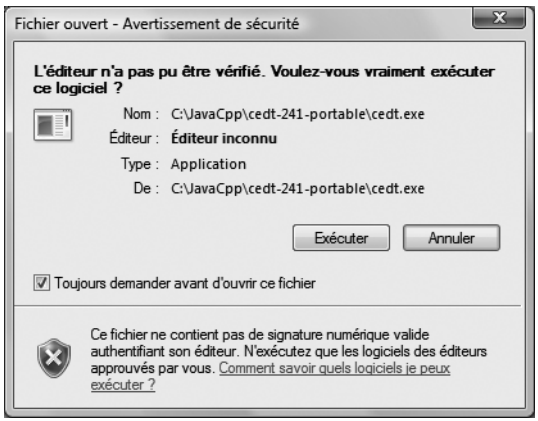

Figure C-8

Association avec les fichiers source

Il faudra décocher la case Toujours demander avant d'ouvrir ce fichier, afin de ne plus avoir ce message. L'exemple de la section suivante concerne la version 3.72 bêta 241 (non recommandée, pour la raison évoquée plus haut).

Demande d'autorisation sous Windows Vista

Suivant les configurations de Windows Vista et les comptes utilisateur, il peut arriver que l'éditeur Crimson (`cedt.exe`) ne démarre pas sans demande d'autorisation et ceci à chaque démarrage.

Dans ce cas il y a trois possibilités connues et vérifiées par l'auteur :

- Nous acceptons ce « phénomène rébarbatif » et nous autorisons Crimson à chaque démarrage. Nous pouvons aussi démarrer Crimson et le laisser ouvert en permanence.

- Dans le répertoire `C:\JavaCpp\Crimson Editor`, nous effectuons un clic droit sur le fichier `cedt.exe` et sélectionnons Propriétés. Nous cliquons ensuite sur l'onglet Compatibilité et désactivons les options Exécuter ce programme en mode de compatibilité pour : et Exécuter les paramètres en tant qu'administrateur. Nous procédons de la même manière pour l'option Afficher les paramètres pour tous les utilisateurs (cette dernière demande une autorisation).
- Une autre alternative consiste à désactiver le contrôle des comptes d'utilisateurs, ce qui n'est pas nécessairement une bonne idée pour la sécurité du PC. Pour cela, il faudrait sélectionner le menu Démarrer > Panneau de configuration > Comptes d'utilisateurs > Activer ou désactiver le contrôle de compte d'utilisateur et décocher l'option Utiliser le contrôle d'utilisateurs pour nous aider à protéger notre ordinateur. Le redémarrage de Windows Vista serait ensuite nécessaire.

Glisser les fichiers depuis l'explorateur

C'est une procédure que nous recommandons lors du travail dans un ou plusieurs chapitres. En effet, nous pouvons très facilement sélectionner un ou plusieurs fichiers `.cpp`, `.h`, `.java`, `.txt`, `.cs` ou encore un `Makefile` avec le bouton gauche de la souris, et les glisser dans Crimson (figure C-9). C'est la manière la plus conviviale pour ouvrir rapidement un fichier.

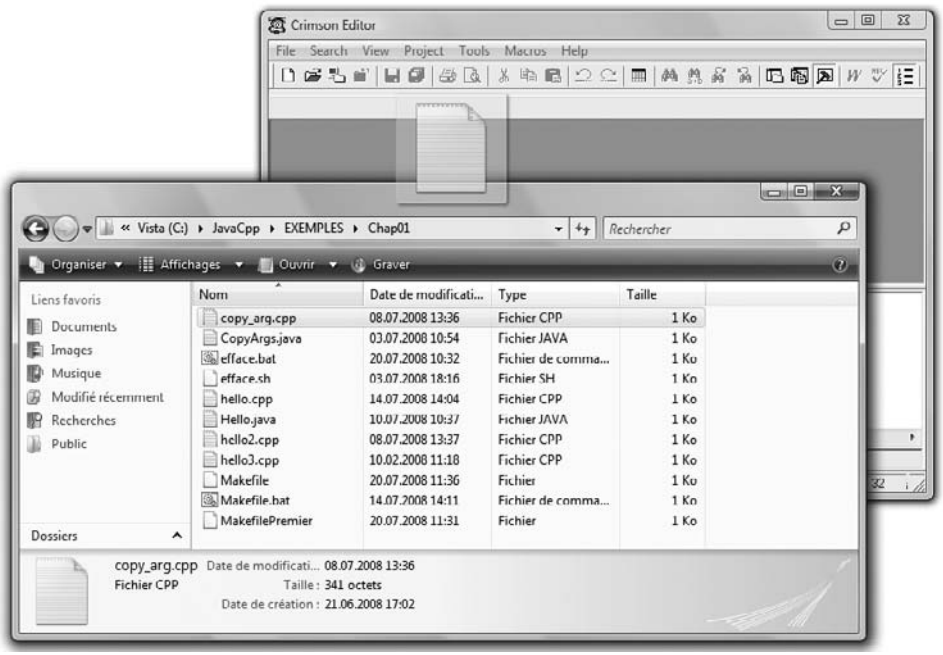

Figure C-9

Ouvrir des fichiers dans Crimson depuis l'explorateur de Windows

Il est aussi possible de les déplacer sur l'icône de l'application dans la barre des tâches de Windows si Crimson est déjà ouvert. Il convient de faire bien attention et de ne pas se tromper de fenêtre ou de choisir le Bureau : dans ce cas, les fichiers seraient alors déplacés au lieu de s'ouvrir dans Crimson.

En cas de difficultés, il suffira d'utiliser le « navigateur » de Crimson (menu File > Open) qui fonctionne parfaitement, même pour charger plusieurs fichiers d'un même répertoire.

Configuration de Crimson

La configuration de Crimson s'effectue via le menu Tools > Preferences de l'éditeur (figure C-10) :

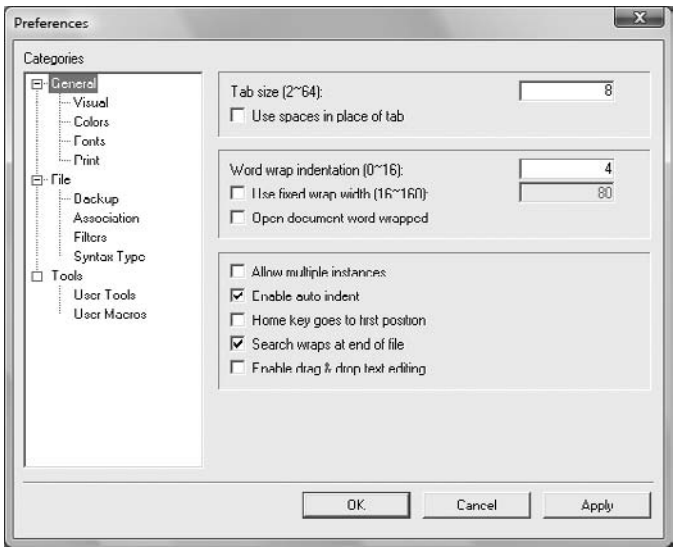

Figure C-10

Configuration de Crimson

Il n'est pas conseillé d'utiliser de tabulateurs (voir le chapitre 3, section « Recommandations pour la forme »). Nous conseillons de conserver la valeur 8 pour le champ Tab size (2~64) et de ne pas cocher l'option Use spaces in place of tab : seuls les `Makefile` ont besoin de tabulateurs.

Dans la rubrique Visual (figure C-11), il ne faudra pas oublier d'activer l'option Show line numbers afin de faire apparaître les numéros de lignes utiles en cas d'erreurs de compilation Java, C++ ou C#.

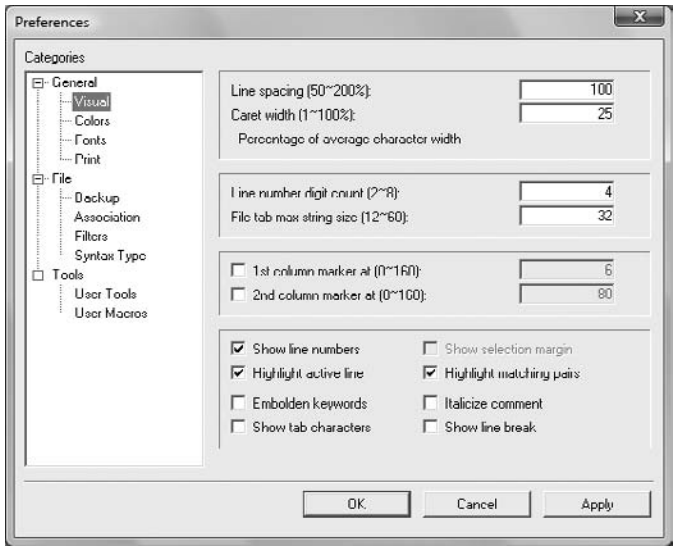

Figure C-11

Activation des numéros de lignes dans Crimson

Dans la rubrique Backup (figure C-12), nous activerons l'option Make backup file before saving a file afin de créer un fichier de sauvegarde lors de l'enregistrement des fichiers. Nous spécifierons l'extension `.bak` pour cette sauvegarde dans le champ Backup extension.

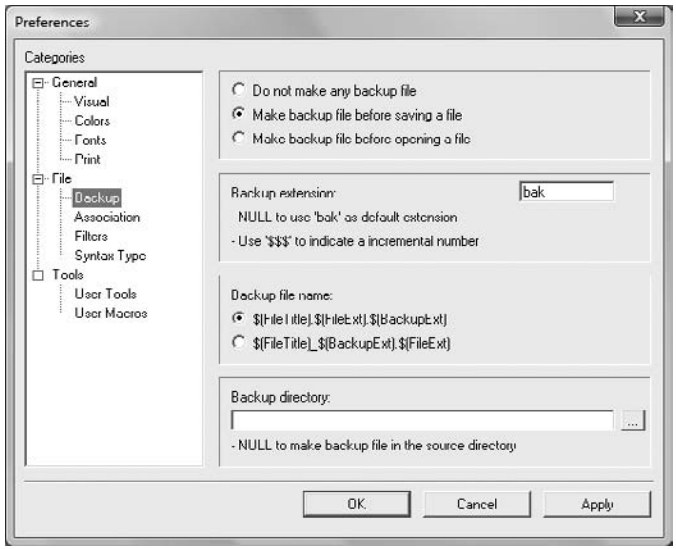

Figure C-12

Configuration des fichiers de sauvegarde dans Crimson

Ce type de sauvegarde n'est pas très évolué, mais pourrait suffire en cas de petite catastrophe. Il ne faudra pas oublier d'effectuer une copie régulière de son travail dans un autre répertoire, un autre disque voire un support externe. Le fichier `effaceBak.bat` situé dans le répertoire `C:\JavaCpp\EXEMPLES` permet de procéder à un nettoyage complet des fichiers `.bak` dans tous les répertoires des exemples.

Configuration des menus

Nous allons à présent mettre en place la liste des outils qu'il faut absolument définir dans la rubrique Tools > User Tools des préférences. Ils nous permettront de compiler et d'exécuter nos programmes Java et C++.

Ces outils, après leur configuration, seront disponibles via le menu Tools de l'éditeur Crimson (figure C-13) :

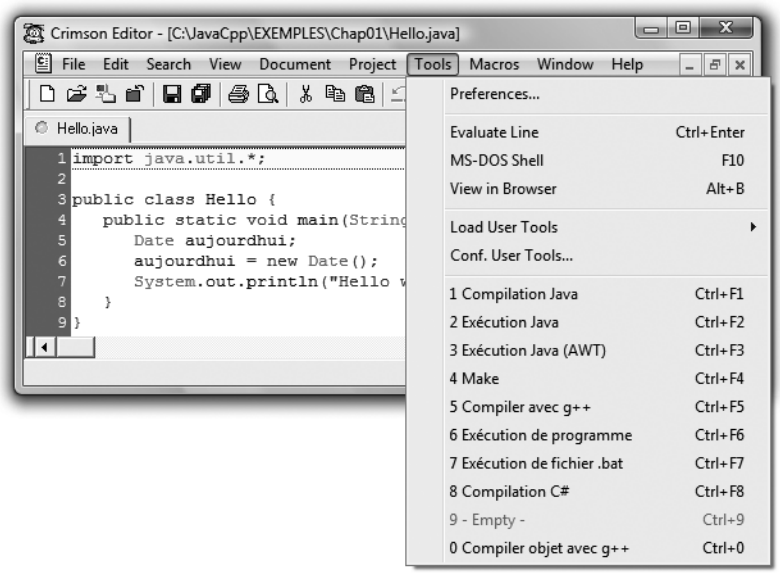

Figure C-13

Liste des outils disponibles dans Crimson

Nous devrions pouvoir nous passer de cette configuration manuelle et nous simplifier le travail en suivant la configuration préparée par l'auteur (voir section « Configuration préparée » au début de cette annexe). Si ce n'est pas possible, il faudra saisir manuellement chacune de ces entrées. Il conviendra, de toute façon, de lire attentivement chaque rubrique pour en comprendre tous les mécanismes. À la section « Jouer avec Crimson » du chapitre 1, il est conseillé de reprendre les aspects essentiels des procédures d'édition, de compilation et d'exécution, après avoir pris connaissance de ces différents menus.

Pour chaque menu, les champs suivants seront disponibles (figure C-14) :

- Menu Text : nom de la commande ;
- Command : emplacement sur le disque de la commande à exécuter ;
- Argument : les arguments de la commande ;
- Initial Dir : le répertoire initial de travail ;
- Hot key : le raccourci clavier.

Nous pourrons également activer/désactiver un certain nombre d'options, par exemple, Capture output.

Menu 1 : Compilation Java

La figure C-14 présente la configuration du menu Compilation Java (Ctrl+F1), menu 1 de la figure C-13 :

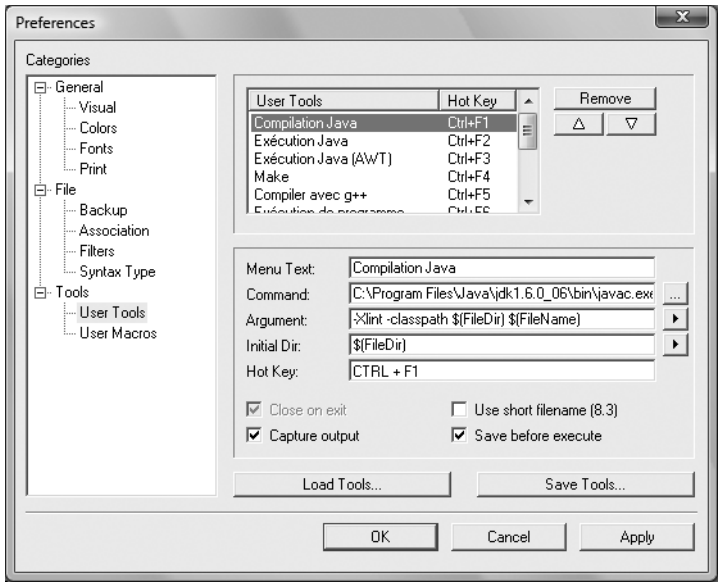

Figure C-14

Configuration du menu Compilation Java dans Crimson

Voici le récapitulatif des paramètres à spécifier pour ce menu :

- Menu Text : Compilation Java ;
- Command : `C:\Program Files\Java\jdk1.6.0_06\bin\javac.exe` ;
- Argument : `-Xlint -classpath $(FileDir) $(FileName)` ;
- Initial Dir : `$(FileDir)` ;

- Hot Key : CTRL+F1 ;
- Option(s) à activer : Capture output et Save before execute.

En guise d'exemple, nous allons ouvrir le fichier `Hello.java` depuis le répertoire `C:\JavaCpp\EXEMPLES\Chap01`.

La combinaison de touches Ctrl+F1, ou le menu 1 associé Compilation Java, va nous permettre de compiler notre code source Java avec le `java.exe` installé dans l'annexe B. Le résultat est présenté dans la fenêtre inférieure de la figure C-15 et nous voyons ici qu'il n'y a pas d'erreurs. Le champ Initial Dir, ici spécifié à `$(FileDir)`, est important, car il nous positionne dans le répertoire correct. Le `classpath` sur le répertoire est correct, mais en fait pas nécessaire.

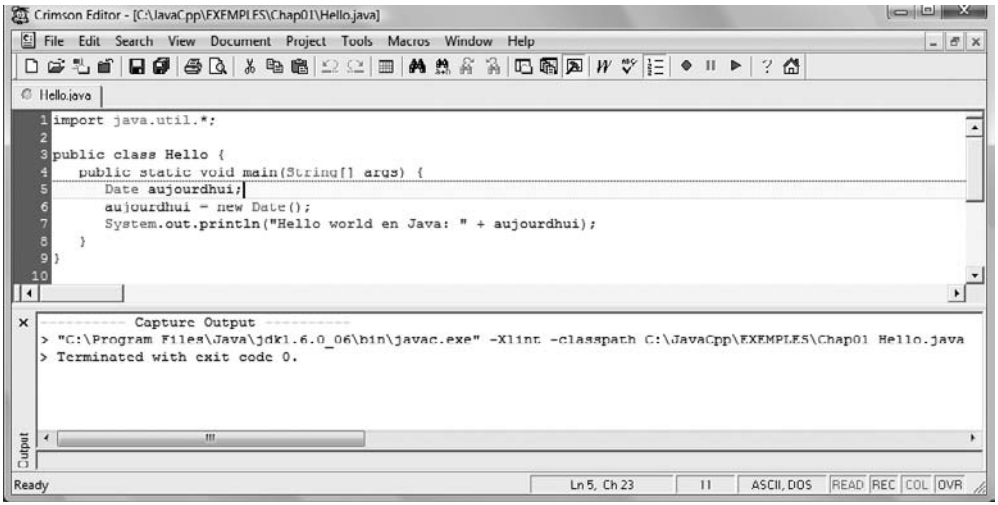

Figure C-15

Compilation du fichier Hello.java dans Crimson

Au passage, ajoutons une remarque à propos de l'argument `-Xlint`. Ce paramètre du compilateur Java va nous aider à analyser et corriger des constructions du langage qui ne suivent pas à 100 % la spécification. Ce paramètre ne se trouve dans aucun des `Makefile` de cet ouvrage, c'est pourquoi le `make` (menu 4) pourra nous indiquer une note comme :

```
Note: Recompile with -Xlint:deprecation for details
```

Le fichier `.class` sera quand même généré et donc exécutable. Si nous effectuons à nouveau un `make`, et si aucun des autres programmes ne comporte d'erreurs, nous recevrons tout de même un :

```
make: Nothing to be done for `all'.
```

car il a déjà été compilé (mais avec de petites erreurs acceptables).

Pour revoir ce message d'alerte, il faudra effacer tous les objets avec `efface.bat` et procéder à un nouveau `make`. En utilisant le menu 1 (`javac.exe -Xlint`), nous obtiendrons davantage d'informations, comme les méthodes dépréciées d'anciennes versions de Java et des suggestions sur leur conversion.

Lors de la configuration des menus dans Crimson, nous pouvons cliquer sur les icônes situées à droite des différents paramètres afin d'accéder à un menu contextuel. Nous y trouverons, par exemple, l'entrée File Directory, qui spécifiera automatiquement un `$(FileDir)` dans le champ correspondant (figure C-16) :

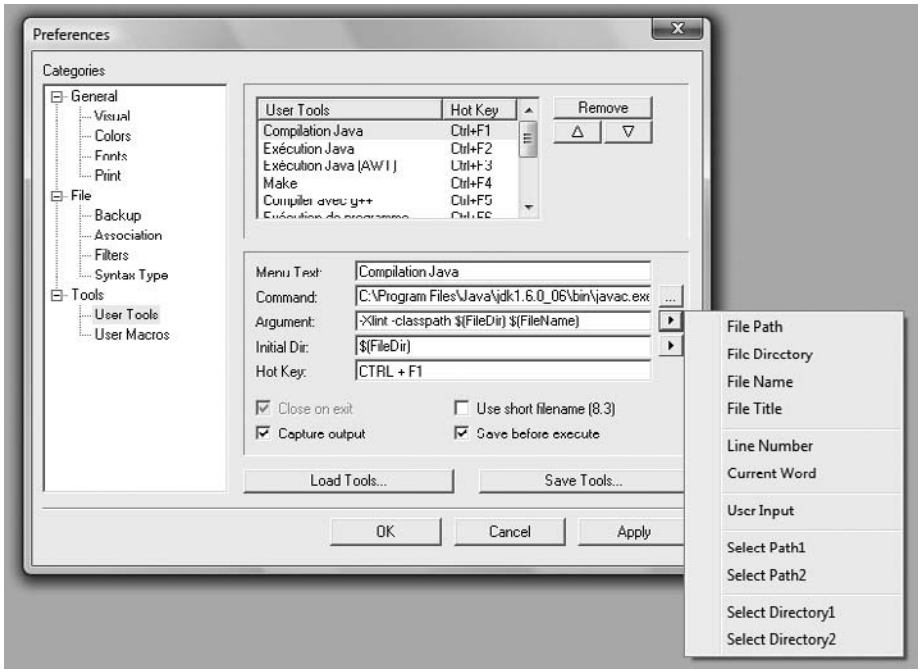

Figure C-16

Menu contextuel des paramètres de configuration des menus

Les options Capture output et Save before execute doivent être activées. Sans la première, nous n'aurions pas de résultat de compilation dans la fenêtre inférieure de Crimson. Sans la seconde, le fichier, que nous avons peut-être modifié et corrigé en cas d'erreur, ne serait pas sauvegardé.

Menu 2 : Exécution Java

Voici les paramètres de configuration à spécifier pour ce deuxième menu :

- Menu Text : Exécution Java ;
- Command : `C:\Program Files\Java\jdk1.6.0_06\bin\java.exe` ;

- Argument : `-classpath $(FileDir) $(FileTitle)` ;
- Initial Dir : `$(FileDir)` ;
- Hot Key : CTRL+F2 ;
- Option à activer : Capture output.

La figure C-17 présente le résultat de l'exécution de la classe `Hello` du chapitre 1 avec ces paramètres :

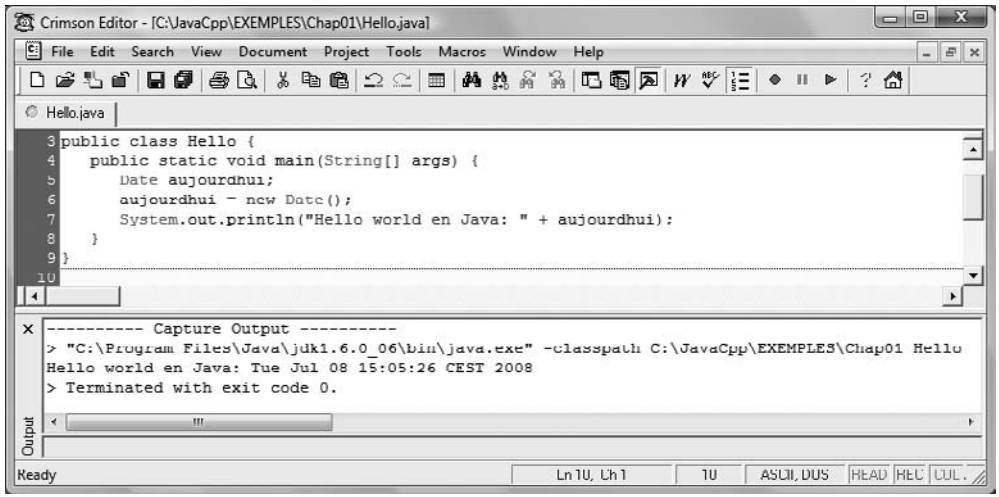

Figure C-17

Exécution de la classe Hello

L'argument `$(FileTitle)` est intéressant : nous n'exécutons pas `java Hello.class` mais `java Hello`. Nous n'avons pas besoin de l'extension `.class` du fichier compilé.

Nous pouvons redimensionner la fenêtre principale de Crimson à souhait ainsi que ses différents panneaux à l'aide de la souris.

Menu 3 : Exécution Java (AWT)

Voici les paramètres à spécifier pour ce troisième menu :

- Menu Text : Exécution Java (AWT) ;
- Command : `C:\Program Files\Java\jdk1.6.0_06\bin\javaw.exe` ;
- Argument : `-classpath $(FileDir) $(FileTitle)` ;
- Initial Dir : `$(FileDir)` ;
- Hot Key : CTRL+F3 ;
- Option à activer : Capture output.

Le menu est similaire au menu précédent, mais avec `javaw` pour exécuter des classes AWT ou Swing (voir chapitre 18).

Menu 4 : Make

Voici les paramètres à spécifier pour le menu Make :

- Menu Text : Make ;
- Command : `C:\MinGW\bin\make.exe` ;
- Argument : `-f $(FileName) all` ;
- Initial Dir : `$(FileDir)` ;
- Hot Key : CTRL+F4 ;
- Options à activer : Capture output et Save before execute.

La figure C-18 présente le résultat obtenu pour le `Makefile` du chapitre 1 (à condition que tous les objets n'aient pas encore été compilés – `efface.bat` peut être utilisé) :

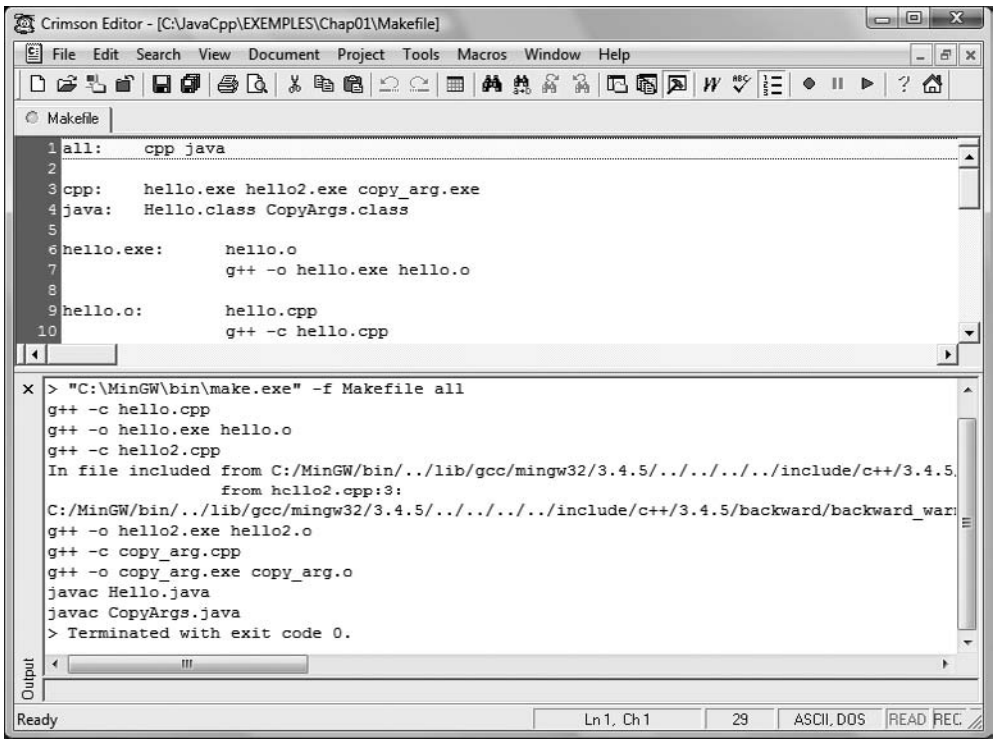

Figure C-18

Exécution make d'un Makefile

L'argument `-f $(FileName)` est important : il nous permet d'exécuter des fichiers `make` qui ne s'appellent pas nécessairement `Makefile`. Les erreurs qui apparaissent ici sont expliquées dans le chapitre 1.

Dans la section « Problèmes potentiels avec le make » de l'annexe B, nous avons mentionné les difficultés susceptibles d'être rencontrées avec le `make`, qu'il faut modifier selon la machine et la version de Windows. Le chemin d'accès indiqué ici semble fonctionner dans tous les cas. Lors d'une installation séparée de MSYS et MinGW, il pourrait s'avérer nécessaire de corriger la référence du `make` en `C:\msys\1.0\bin\make.exe`.

Menu 5 : Compiler avec g++

Les paramètres à spécifier pour ce cinquième menu sont :

- Menu Text : Compiler avec g++ ;
- Command : `C:\MinGW\bin\g++.exe` ;
- Argument : `-o $(FileTitle) $(FileName)` ;
- Initial Dir : `$(FileDir)` ;
- Hot Key : CTRL+F5 ;
- Options à activer : Capture output et Save before execute.

La figure C-19 présente le résultat obtenu pour un exemple du chapitre 1 :

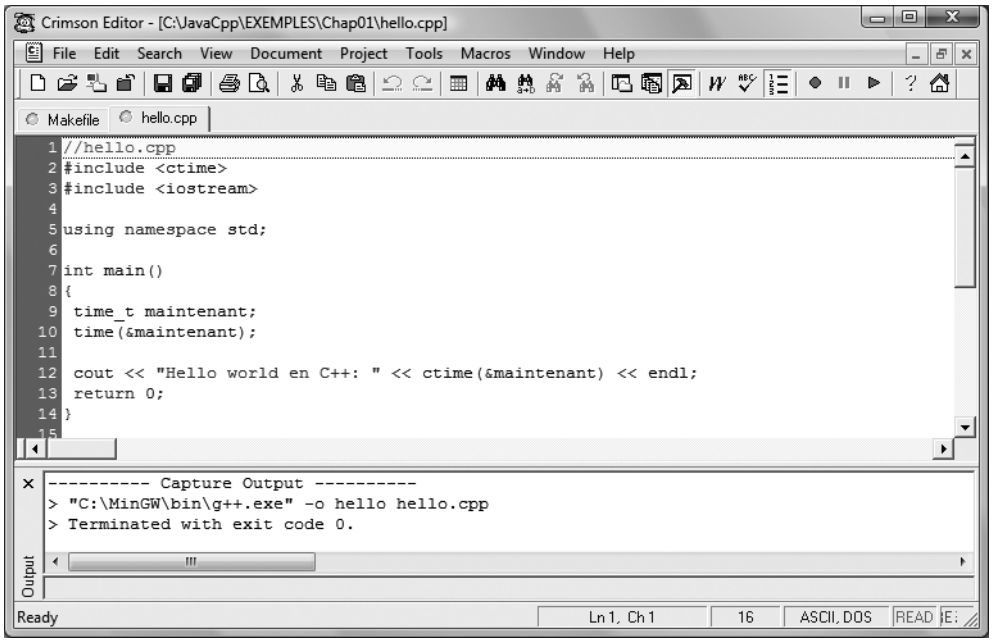

Figure C-19

Compilation d'un fichier C++

À présent, nous comprenons mieux les différences entre `$(FileTitle)` et `$(FileName)`. L'option Save before execute est évidemment nécessaire.

Menu 6 : Exécution de programme

Voici la liste des paramètres à spécifier pour ce sixième menu :

- Menu Text : Exécution de programme ;
- Command : `C:\Windows\system32\cmd.exe` ;
- Argument : `/C $(FileTitle)` ;
- Initial Dir : `$(FileDir)` ;
- Hot Key : CTRL+F6 ;
- Option(s) à activer : Capture output.

La suite de l'exemple ci-dessus est naturelle (figure C-20) :

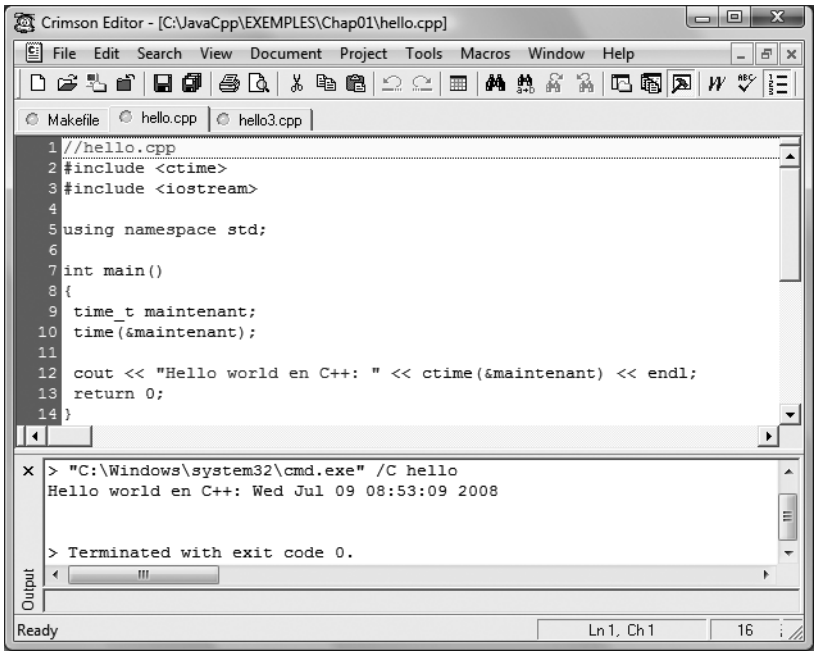

Figure C-20

Exécution d'un fichier .exe

Nous rappelons qu'il faudra avant de cliquer sur ce menu, avoir au premier plan la fenêtre de la source concernée. Si nous désirons exécuter `hello3.exe`, (compilé à partir de `hello3.cpp`) il faudra cliquer sur l'onglet le plus à droite.

En exécutant `hello` (`$(FileTitle)`), nous avons bien un appel à `hello.exe` (reconnu par le DOS/Windows). C'est pour cette raison que ce menu fonctionne aussi pour des programmes compilés à partir de code source C#.

Nous ne sauvegardons pas le fichier source en cas d'exécution car ce n'est pas nécessaire ici, contrairement à la compilation.

Menu 7 : Exécution de fichier .bat

Les paramètres à spécifier pour ce menu sont :

- Menu Text : Exécution de fichier .bat ;
- Command : `C:\Windows\system32\cmd.exe` ;
- Argument : `/C $( FileName)` ;
- Initial Dir : `/C $( FileDir)` ;
- Hot Key : CTRL + F7 ;
- Option(s) à activer : Capture output.

Cette configuration est identique à celle du menu 6, à la seule différence que nous avons `$(FileName)` au lieu de `$(FileTitle)`. Ce menu sera pratique pour exécuter, par exemple, les fichiers `efface.bat` présents dans chaque répertoire des exemples et des exercices. L'option Save before execute a été désactivée car un fichier `.bat` est rarement modifié.

Lorsqu'un fichier est modifié, la puce verte dans l'onglet qui contient son nom devient rouge. Il faudra alors le sauvegarder avant de cliquer sur ce menu, afin que les modifications soient prises en compte (menu File > Save).

Si un fichier `.bat` possède une `pause` ou que le programme lui-même possède une interaction avec l'utilisateur, il est possible de voir ceci à l'écran lors de la prochaine exécution :

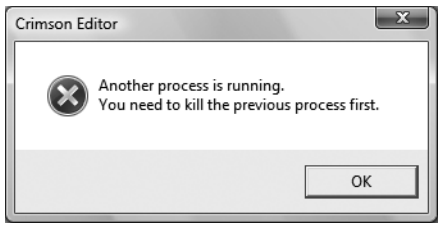

Figure C-21

Application qui ne s'est pas terminée correctement.

Il faudra alors se positionner dans la fenêtre Output de Crimson et appuyer sur la touche Entrée de notre clavier. Crimson, dans ces cas-là, pourrait empêcher une nouvelle exécution de programmes. Cependant, dans tous les cas, nous pourrions toujours quitter Crimson définitivement.

Menu 8 : Compilation C#

Les paramètres de configuration de ce menu sont :

- Menu Text : Compilation C# ;
- Command : `C:\Windows\Microsoft.NET\Framework\v3.5\csc.exe` ;
- Argument : `$(FileName)` ;
- Initial Dir : `$(FileDir)` ;
- Hot Key : CTRL+F8 ;
- Options à activer : Capture output et Save before execute.

Ce menu sera choisi pour la compilation des programmes C# du chapitre 23. Nous compilerons des sources avec une extension `.cs` pour produire des binaires `.exe`. Ces derniers pourront être exécutés avec le menu 6 en restant dans la fenêtre de la source `.cs`.

Menu 0 : Compiler objet avec g++

Voici la liste des paramètres à spécifier pour ce dernier menu :

- Menu Text : Compiler objet avec g++ ;
- Command : `C:\MinGW\bin\g++.exe` ;
- Argument : `-c $(FileName)` ;
- Initial Dir : `$(FileDir)` ;
- Hot Key : CTRL+0 ;
- Options à activer : Capture output et Save before execute.

Nous désirons ici ne compiler que le fichier `.cpp` en fichier `.o`, sans générer l'exécutable final `.exe`. Ce n'est pas vraiment nécessaire, mais pratique. Si plusieurs objets composent un binaire, nous devrons utiliser le `Makefile`.

Fermer toutes les fenêtres

Lorsque plusieurs fichiers sont ouverts simultanément, nous pouvons aussi, si nous possédons un écran large, les arranger comme présenté sur la figure C-22. Pour cela, il suffit de sélectionner le menu Window > Tile Vertical de Crimson. Il est également possible de les présenter horizontalement et en cascade.

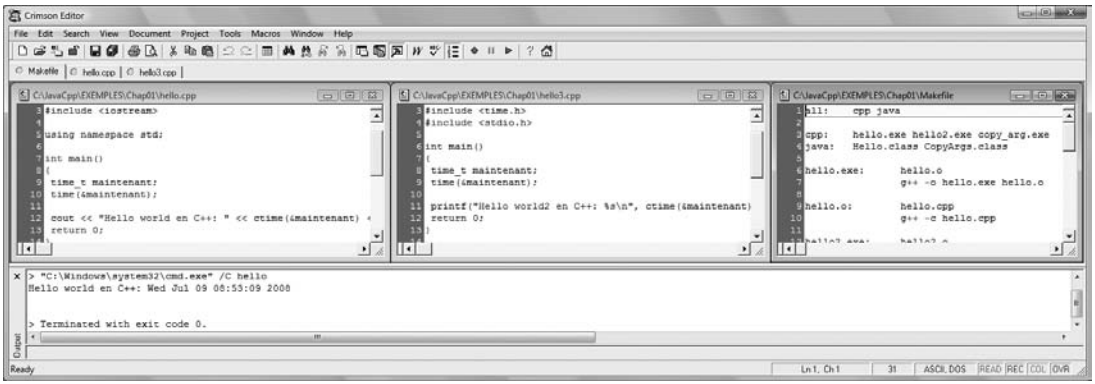

Figure C-22

Présentation de trois fichiers en mode Mosaïque verticale

Nous pourrons naviguer rapidement d'un fichier à l'autre et ainsi utiliser facilement l'un des outils présentés ci-dessus.

Si nous fermons Crimson, les paramètres seront conservés lors du prochain démarrage et les fichiers s'ouvriront selon la présentation choisie.

Si nous naviguons entre plusieurs chapitres et avons ouvert trop de fichiers, nous pouvons les fermer individuellement par le menu File > Close, ou bien tous à la fois avec le menu File > Close All.

Remarques générales

Ces outils sont très pratiques mais l'utilisateur devra cependant réfléchir un peu s'il essaie, par exemple, de compiler une source `.cpp` (C++) avec le compilateur `javac`. En effet, les messages d'erreurs retournés par les compilateurs seront, dans ce cas, tout à fait déroutants.

Compiler très souvent et commencer par le squelette du programme, sans trop de code, est toujours un conseil à donner. Le simple oubli d'un guillemet (`"`) ou d'une accolade (`}`) peut conduire le compilateur à indiquer une erreur plusieurs dizaines de lignes plus bas, surtout en C++. C'est parfois un casse-tête dont la seule solution est de commenter de gros morceaux de code avec des barres obliques (`//`) ou des barres obliques accompagnées d'astérisques (`/* */`), ce qui équivaut à revenir au squelette des blocs.

Des outils comme NetBeans (voir annexe E) pour Java permettent une compilation préalable lors de la frappe du code !

Exercices

Les solutions à ces deux exercices sont laissées aux lecteurs. Nous n'avons pas utilisé le menu 9 dans cette annexe à cet effet.

1. Introduire la commande `appletviewer` (voir chapitre 18) sur un document `.html` dans le menu Tools Crimson.
2. Introduire une fonction `make` qui exécute un `Makefile` depuis un fichier `.cpp` ou `.java`. Cette fonction peut s'avérer utile pour le chapitre 21 (JNI) où il y a trop de dépendances : un simple `make` facilitera le travail.

D

Installation du SDK du Framework de .NET

Dans cette annexe, nous allons découvrir comment installer le SDK du Framework (plate-forme) de .NET de Microsoft.

Pour installer ce SDK, il faut disposer de Windows NT4, 2000, 2003 ou XP (sous Vista, il est déjà installé d'office), et d'un espace disque suffisant : l'installation de la version 3.5 de .NET nécessite environ 35 Mo. De plus, il faut toujours réserver suffisamment de place sur ce disque pour permettre des mises à jour, de nouvelles applications ou de nouvelles versions de Microsoft Office, par exemple, en général également installées sur cette partition.

Cette partie est nécessaire pour exécuter les exemples et les exercices du chapitre 23, « L'étape suivante : le langage C# de Microsoft ».

Installation du SDK 3.5

L'installation ne sera sans doute pas nécessaire sous Windows Vista ou sous tout autre système d'exploitation dans lequel le SDK du Framework est déjà installé.

Le fichier d'installation se trouve dans le répertoire `INSTALL` du CD-Rom et se nomme `dotnetfx35.exe`.

Si nous le réinstallons sous Windows Vista, nous verrons apparaître la fenêtre représentée à la figure D-1 :

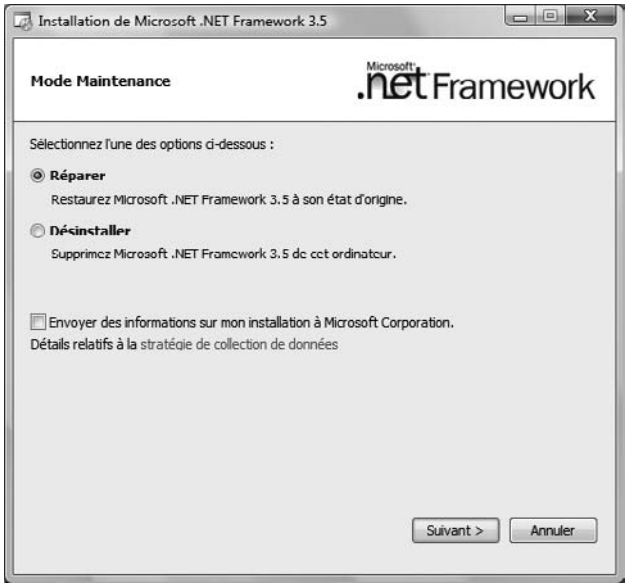

Figure D-1

Réinstaller le .NET sous Windows Vista

Dans ce cas, nous pouvons très bien annuler la procédure.

En nous rendant dans le répertoire `C:\WINDOWS\Microsoft.NET\Framework\`, nous pourrions constater qu'une ou plusieurs autres versions sont déjà installées, par exemple la version v2.0, et nous pourrions décider de la (ou les) conserver. Notre code C# est compatible avec la première version du .NET. Il faut simplement s'assurer que le fichier `csc.exe` est bien présent dans le répertoire choisi, par exemple `C:\Windows\Microsoft.NET\Framework\v3.5`.

Suivant la version retenue, il faudra adapter le chemin d'accès dans les `Makefile` du chapitre 23, par exemple :

```
NET = "C:/Windows/Microsoft.NET/Framework/v3.5"
```

Pour lancer l'installation sous Windows XP, il convient de cliquer sur le programme `dotnetfx35.exe` situé dans le dossier `INSTALL` du CD-Rom (figures D-2 et D-3) :

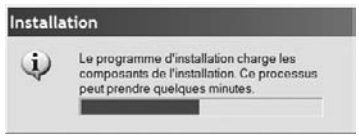

Figure D-2

.NET- Préparation de l'installation

Figure D-3

.NET - Bienvenue dans l'installation

L'installation nécessite une connexion Internet (nous ne sommes cependant pas certains que ce soit nécessaire pour toutes les configurations) :

Figure D-4

.NET – Progression de l'installation

À la fin du téléchargement et de l'installation, il suffira de cliquer sur le bouton Quitter lorsque la fenêtre de la figure D-5 apparaîtra :

Figure D-5
.NET – Installation terminée

Téléchargement de .NET depuis Internet

Le SDK inclus sur le CD-Rom de ce livre est par ailleurs disponible sur le site Web de Microsoft à l'adresse suivante :

```
http://msdn.microsoft.com/fr-fr/netframework/aa569263.aspx
```

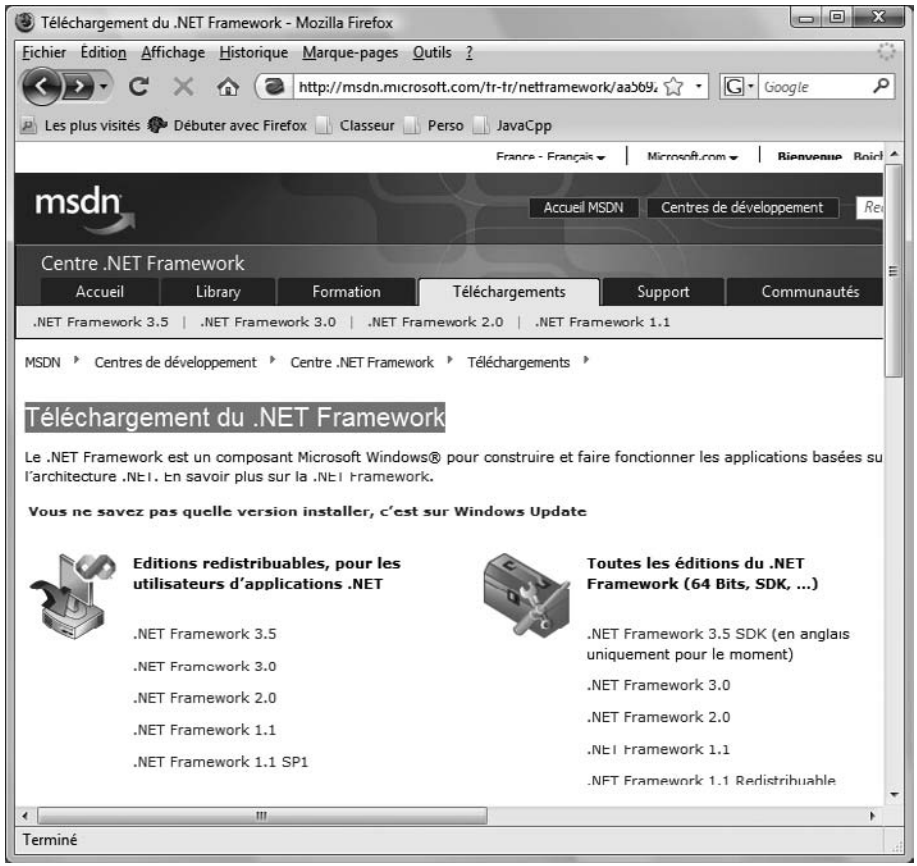

Figure D-6

Le site Web du SDK de Microsoft .NET en France

Pas de nouveau PATH

Dans l'édition précédente de cet ouvrage, le `PATH` des commandes Windows avait été étendu durant l'installation pour permettre l'accès à la commande `csc.exe`.

Ici, ce n'est pas le cas. Tous les accès se feront dans Crimson et dans une fenêtre DOS au moyen de la référence complète. Nous allons voir un exemple ci-après.

Vérification de l'installation du .NET Framework SDK

Si nous avons suivi les procédures d'installation de l'annexe B, le fichier `HelloCs.cs` du chapitre 23 devrait avoir été installé depuis le CD-Rom dans le répertoire de travail `C:\JavaCpp\EXEMPLES\Chap23`.

Dans une fenêtre de commande DOS et dans ce répertoire, nous allons exécuter la commande suivante :

```
csc.exe HelloCs.cs
```

depuis le chemin d'accès du .NET `C:\Windows\Microsoft.NET\Framework\v3.5`, et la commande :

```
HelloCs.exe
```

Ceci nous permet de vérifier la compilation du code C# et l'exécution du fichier binaire produit par la compilation (figure D-7) :

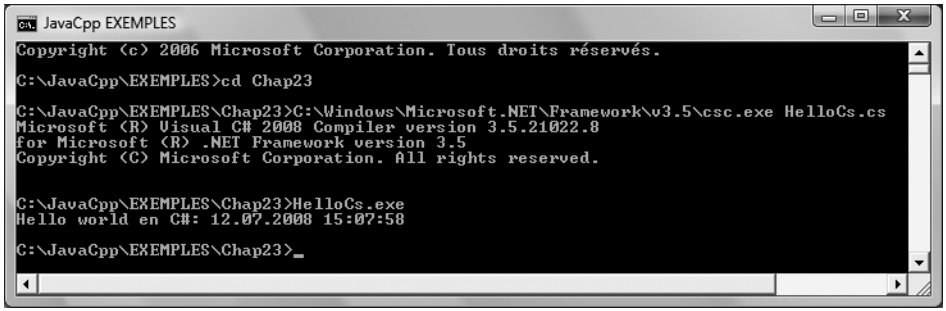

Figure D-7

C# - Vérification de l'installation

Documentation du SDK de .NET

Nous allons à présent partir à la découverte de .NET en consultant le centre de développement .NET Framework sur le site Web `http://msdn.microsoft.com/fr-fr/netframework/default.aspx` (figure D-8).

Il nous donne un aperçu de .NET et de la structure de cette nouvelle plate-forme (Framework) développée par Microsoft Corporation ces dernières années.

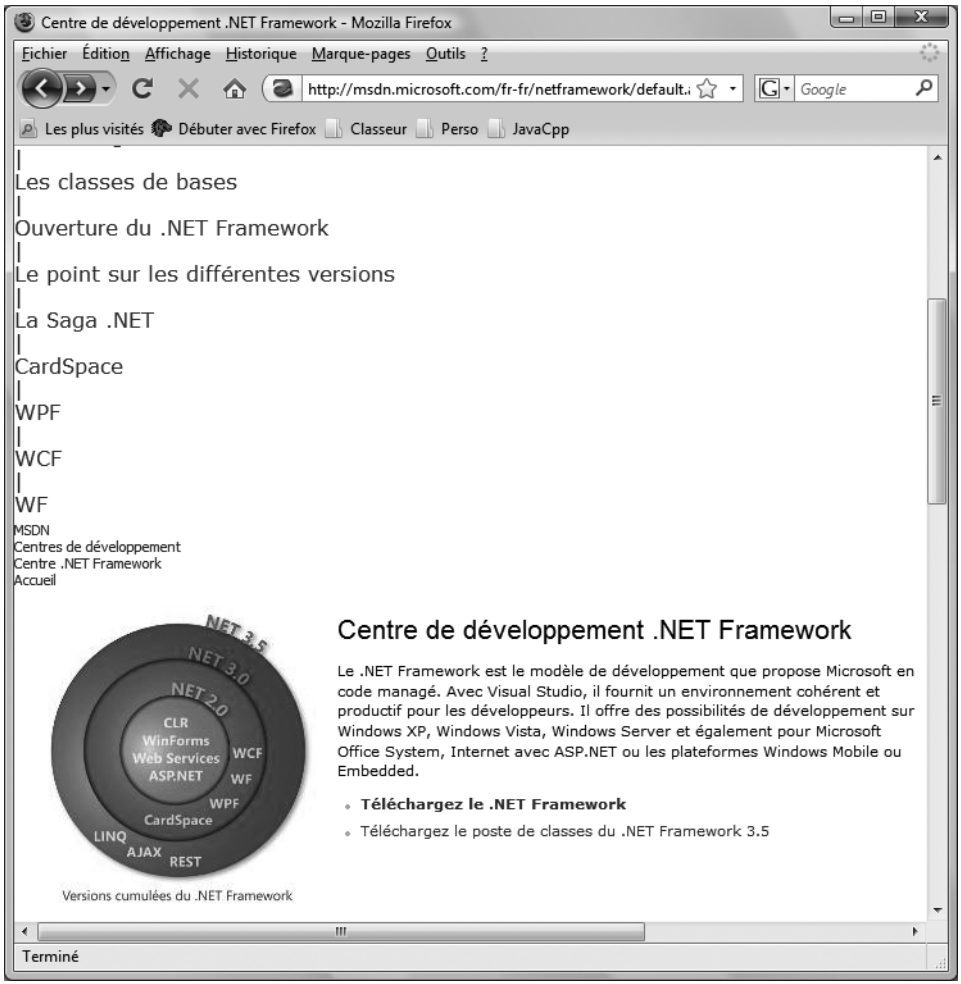

Figure D-8

Centre de développement .NET Framework

Un des points de départ de la documentation est sans doute le lien intitulé Les classes de base sur la figure D-8.

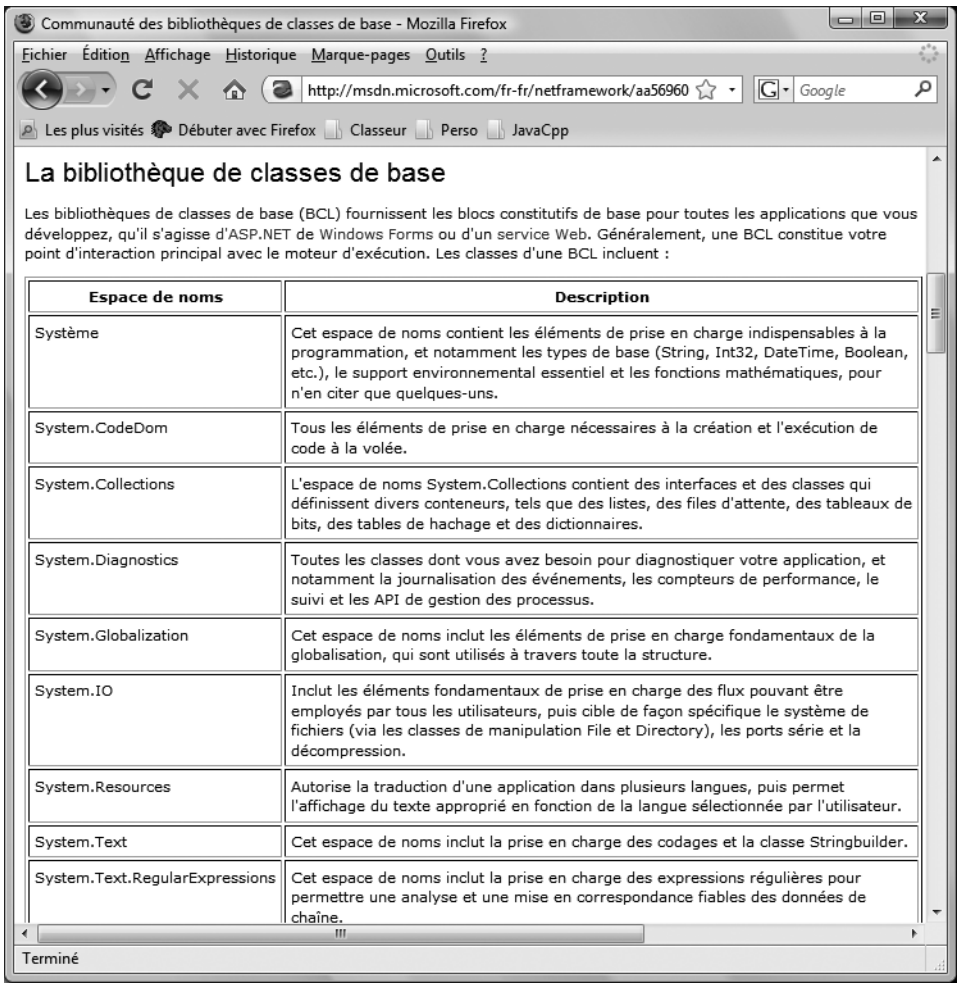

Figure D-9

La bibliothèque des classes de base

La navigation se fait au travers de la hiérarchie des classes, ici pour la classe `Console` (figure D-10) :

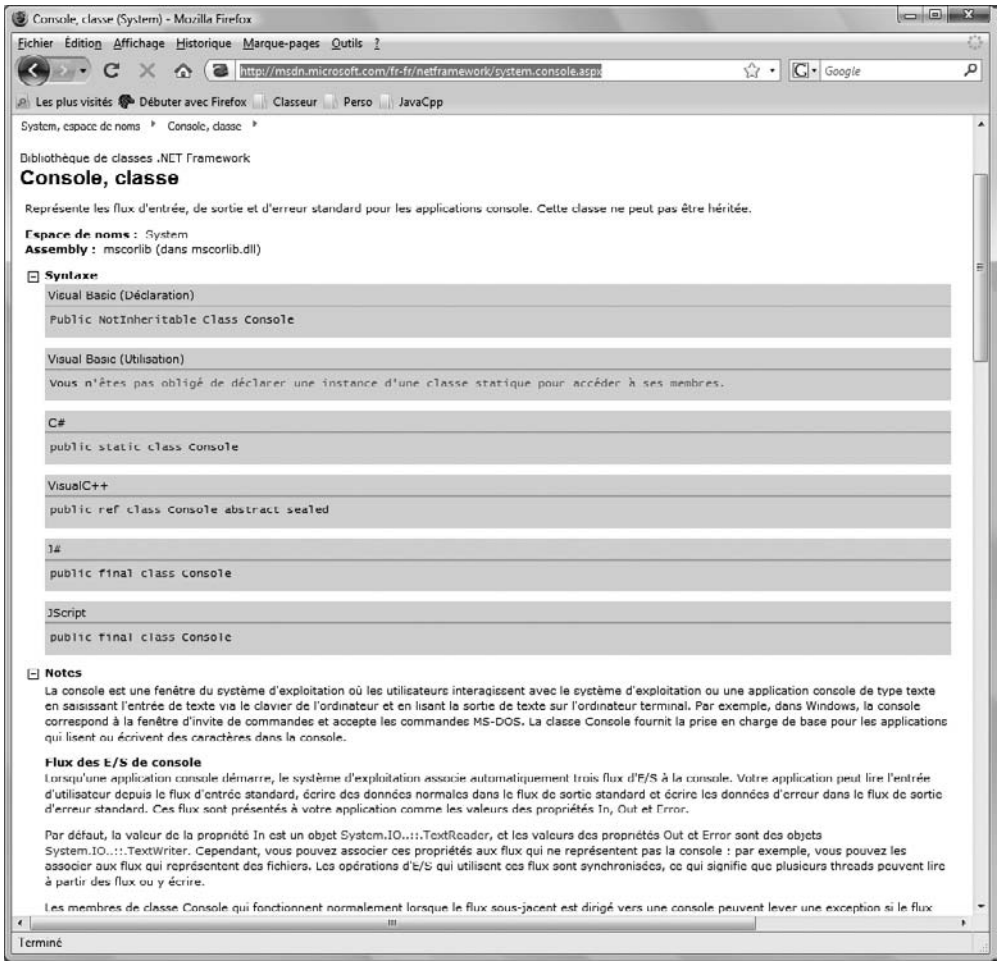

Figure D-10

La classe Console dans le détail

Nous voyons évidemment les références pour les autres langages comme le Visual Basic ou le C++ de Microsoft, qui supportent aussi le Framework .NET.

En continuant notre promenade, nous pourrions retrouver les méthodes (les membres) de la classe `Console` dont nous avons parlé au chapitre 23 :

```
http://msdn.microsoft.com/fr-fr/netframework/system.console_members.aspx
```

Le compilateur du langage C#

Retrouver la référence du compilateur `csc.exe` n'est pas évident :

```
http://msdn.microsoft.com/fr-fr/library/2fdbz5xd.aspx
```

Nous y retrouverons toute une série de rubriques nous permettant de comprendre et d'utiliser les options du compilateur (figure D-11) :

Figure D-11

Options du compilateur C#

Visual C# - L'édition Express

Sur le site Web `http://www.microsoft.com/express/vcsharp/`, Microsoft met à disposition une édition gratuite d'un outil de développement C# (figure D-12) :

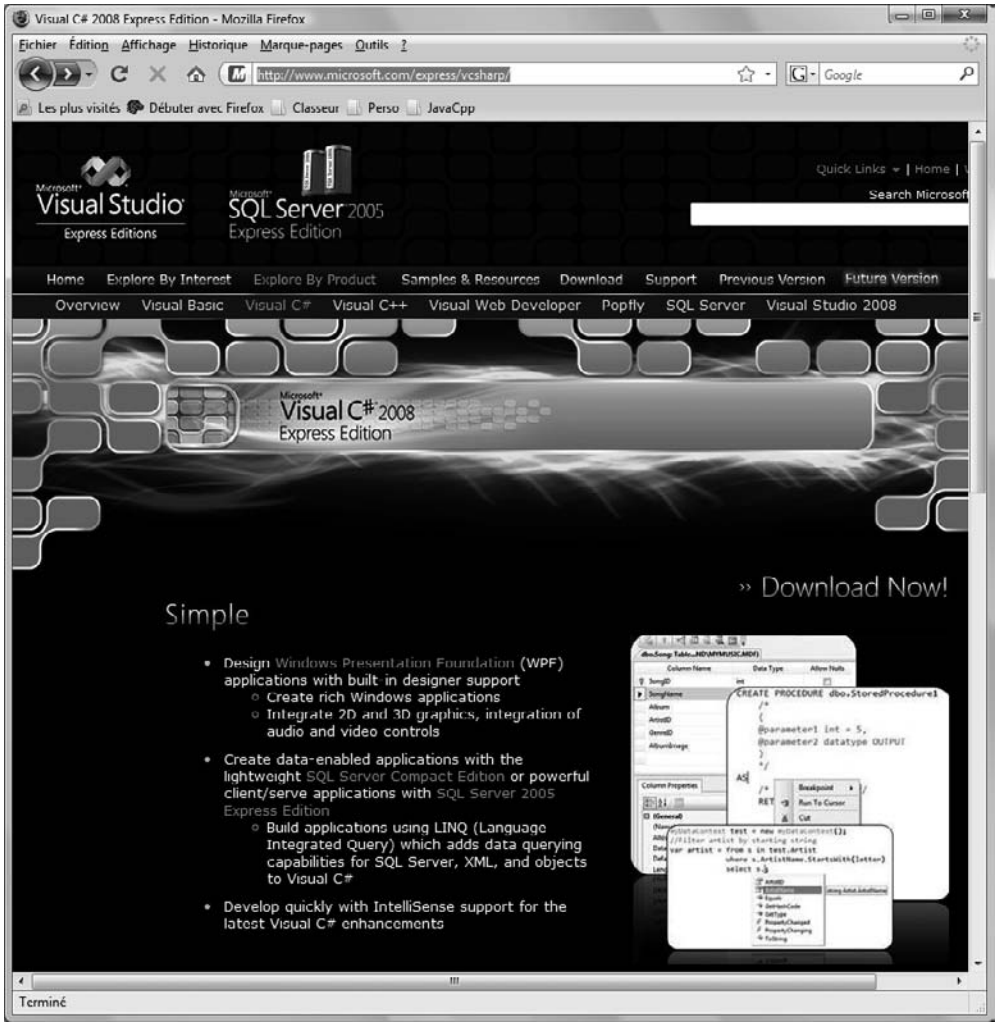

Figure D-12

Visual C# 2008 - Express Edition

Il faudra la télécharger et ensuite demander une clé d'activation auprès de Microsoft qui nous sera envoyée par e-mail.

Les bases de C# données dans cet ouvrage et les ressemblances avec Java, devraient permettre à un bon programmeur de débuter rapidement avec la programmation C# et l'environnement .NET.

De la même manière que pour NetBeans (voir annexe E), il sera possible d'écrire rapidement de vraies applications graphiques Windows. La bibliothèque .NET et ses outils permettront aussi de programmer en C# des fonctions spécifiques à Windows, comme des bibliothèques DLL ou des services.

E

Apprendre Java et C++ avec NetBeans

Généralités

Est-ce le titre d'un nouvel ouvrage ? Pourquoi pas !

NetBeans fait son apparition dans cette édition car un module, permettant d'éditer et de compiler du code C++ en plus de Java, a été récemment intégré dans son environnement. Par ailleurs, l'environnement MinGW, utilisé dans cet ouvrage, est directement intégré et reconnu.

NetBeans (`http://www.netbeans.org/`), tout comme Eclipse (`http://www.eclipse.org/`), est un outil de développement Open Source très puissant et particulièrement bien construit et adapté pour créer des applications ou encore des produits complexes.

Pour plus d'informations sur NetBeans en français, nous recommandons le le site Web suivant :

`http://www.netbeans.org/index_fr.html`

L'installation de NetBeans 6.1 a été vérifiée sous Windows XP et Vista. Les développeurs de NetBeans étant très actifs, de nouvelles versions apparaissent régulièrement. Cependant, avant de passer à une version plus récente (voire bêta), l'auteur conseille au lecteur de se familiariser tout d'abord avec cette version 6.1. Les différences entre versions peuvent être significatives et ne plus correspondre à la description et aux configurations décrites dans cette annexe.

NetBeans est relativement gourmand en ressources. Il peut fonctionner sur un processeur Intel traditionnel avec 512 Mo de mémoire, mais cela reste frustrant pour le développeur.

Pour un investissement minimal, il est recommandé de doubler la mémoire. Utiliser NetBeans sur un processeur Quad Core d'Intel avec 3 Go de mémoire est un vrai plaisir. Un écran de très grande définition (typiquement 1 600 × 1 200 pixels) permettra également de bien visualiser le code source des gros projets.

Linux

Pour un ordinateur aux performances limitées (mémoire et processeur), nous conseillons au lecteur d'utiliser NetBeans dans un environnement Linux, par exemple, Ubuntu (voir annexe F, « Apprendre Java et C++ avec Linux »).

L'installation et l'utilisation de NetBeans sous Linux et sous Windows sont équivalentes. Les différences de configuration sont indiquées dans l'annexe F.

Téléchargement de nouvelles versions

Le site Web de NetBeans (`http://www.netbeans.org/`) permettra aux lecteurs de vérifier si de nouvelles versions existent et de les télécharger le cas échéant afin de les installer (complètement ou uniquement certains composants, comme Java). Les versions Linux sont aussi disponibles sur ce site (voir annexe F).

Documentations et didacticiels

Ce même site Web fera découvrir aux lecteurs un nombre impressionnant de documents, d'exemples, de didacticiels ou encore de vidéos de présentation.

Installation à partir du CD-Rom

L'espace disque requis est au minimum de 140 Mo. Si d'autres composants sont installés, nous pouvons atteindre plus de 350 Mo (voir les options et paquets supplémentaires ci-dessous).

Nous pouvons évidemment installer NetBeans 6.1 séparément, mais il nous faudra au moins une machine virtuelle Java avant de commencer l'installation. Pour utiliser la partie C/C++ de NetBeans, les outils MinGW et MSYS doivent être préalablement installés. Il est donc préférable de procéder tout d'abord aux installations décrites dans l'annexe B (installation des outils incluant Java) et aux vérifications nécessaires.

NetBeans est un outil de développement permettant d'éditer du code et de le compiler. Des débogueurs y sont intégrés (Java et C++), lesquels permettent de faire du pas à pas dans le code source pendant l'exécution afin de s'assurer du bon déroulement du programme ou encore d'examiner le contenu des variables. Nous en donnerons d'ailleurs quelques exemples dans les sections « Naviguer et déboguer » et « Déboguer un projet C++ avec NetBeans » de cette annexe.

La version NetBeans choisie pour cet ouvrage est la version complète 6.1. Elle intègre non seulement le développement de programmes Java, mais aussi celui de programmes C++. Le lecteur pourra de plus s'intéresser par la suite à d'autres aspects et d'autres outils, comme les diagrammes UML ou l'écriture de programmes pour téléphones portables ou le Web (servlets).

La procédure d'installation détaillée ici a été réalisée sous Windows XP mais elle est identique sous Windows Vista. Pour lancer l'installation de NetBeans, nous double-cliquerons sur le programme `netbeans-6.1-ml-windows.exe` situé dans le répertoire `INSTALL` du CD-Rom.

Figure E-1
NetBeans 6.1 – Démarrage de l'installation

Nous cliquerons ensuite sur le bouton Next > afin de poursuivre l'installation (figure E-2) :

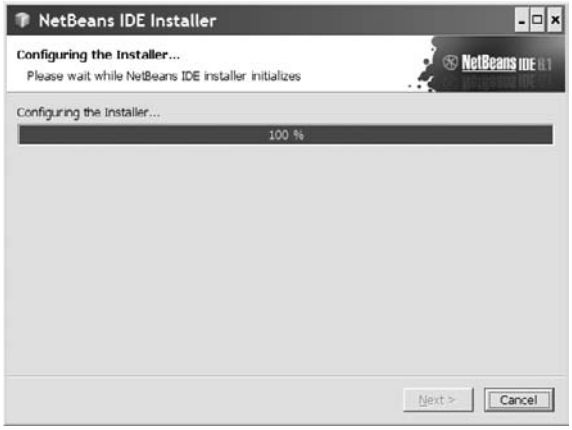

Figure E-2
Installation et initialisation de NetBeans 6.1

La fenêtre de la figure E-3 apparaîtra alors :

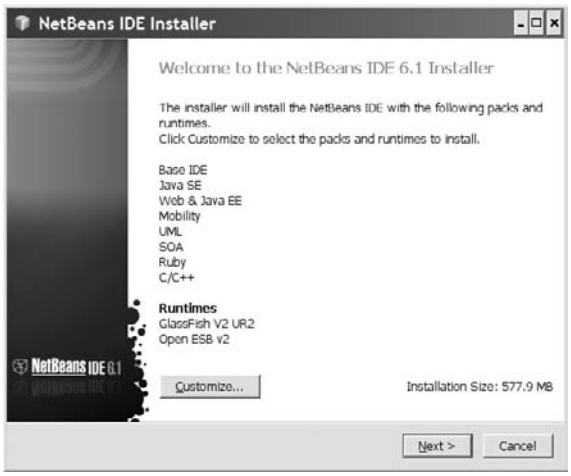

Figure E-3

Présentation des composants optionnels

Comme l'installation de tous les paquets demande plus de 577 Mo, le lecteur pourra décider de n'en garder que quelques-uns, suivant l'espace disque dont il dispose. Pour sélectionner les composants à installer, il faudra cliquer sur le bouton Customize…, ce qui aura pour effet d'ouvrir la fenêtre de la figure E-4.

Figure E-4

Sélection des composants à installer

Ici, nous avons conservé les composants Web & Java EE ainsi que UML. Ils ne sont pas vraiment nécessaires pour cet ouvrage mais nous y reviendrons par la suite. Avec ces

deux composants, l'espace disque requis est d'environ 300 Mo (il serait de 140 Mo seulement si Web & Java EE et UML n'étaient pas sélectionnés). Nous cliquerons ensuite sur OK et la fenêtre de la figure E-5 apparaîtra alors, récapitulant les composants à installer et l'espace disque nécessaire. À ce stade, nous pourrons à nouveau cliquer sur le bouton Customize… pour ajouter ou éliminer un composant.

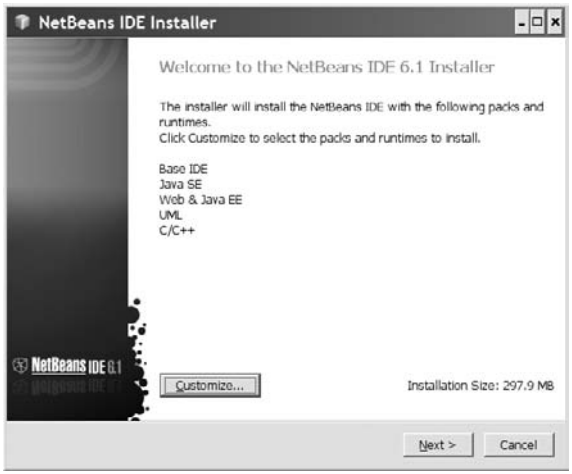

Figure E-5

Liste des composants à installer

Nous cliquerons ensuite sur Next > pour lancer la dernière partie de l'installation. La fenêtre de la figure E-6 s'affichera alors à l'écran :

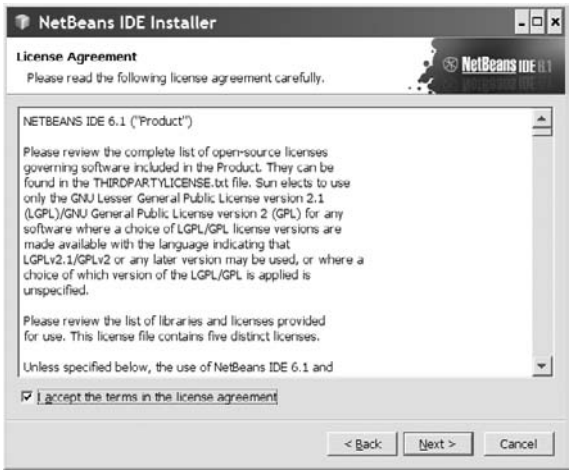

Figure E-6

Termes du contrat de licence de NetBeans 6.1

Nous accepterons les termes du contrat et nous cliquerons sur Next >. Dans la fenêtre suivante (figure E-7), nous choisirons les répertoires d'installation de NetBeans :

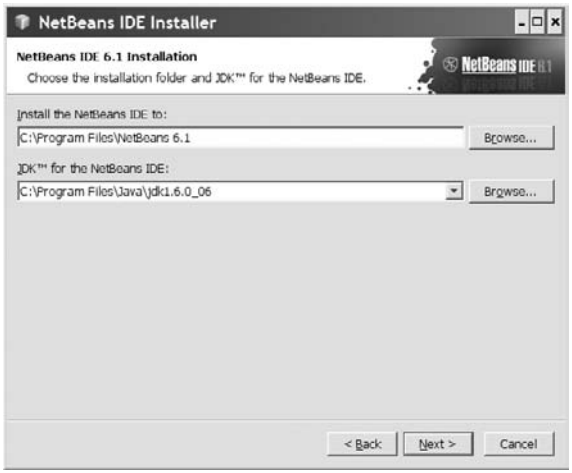

Figure E-7
Choix des répertoires d'installation de NetBeans

Nous conserverons les deux répertoires spécifiés par défaut. Nous constaterons au passage que le second, JDK, a été identifié automatiquement (comme installé dans l'annexe B). Si ce n'était pas le cas, il faudrait le définir manuellement au moyen du bouton Browse… Si l'espace disponible sur le disque `C:` est insuffisant, le lecteur pourra choisir d'installer NetBeans et ses composants sur une autre partition. Une fois les chemins d'accès aux répertoires d'installation renseignés, nous cliquerons alors sur le bouton Next > pour continuer. La fenêtre suivante (figure E-8) indiquera alors le répertoire d'installation choisi et l'espace disque nécessaire.

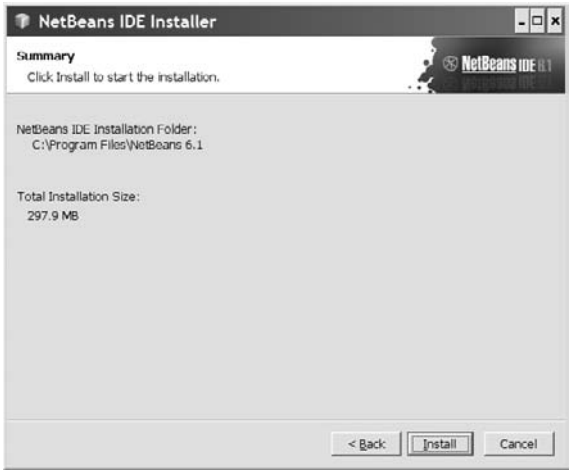

Figure E-8
Installation finale de NetBeans

Si tout est correct, nous pourrons cliquer sur le bouton Install.

Plusieurs messages apparaîtront ensuite dans une nouvelle fenêtre, indiquant l'état d'avancement de l'installation (figure E-9) :

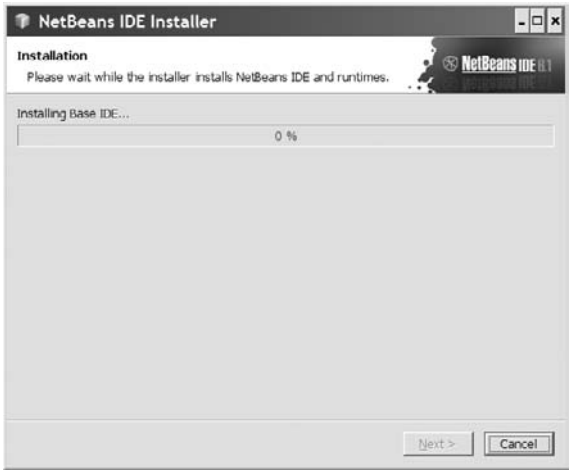

Figure E-9
Progression de l'installation

La durée de l'installation dépendra de la capacité de la machine (mémoire, CPU et disque dur). Avant la fin de l'installation, la fenêtre suivante s'affichera (figure E-10) :

Figure E-10
Fin de l'installation de NetBeans 6.1

Nous activerons l'option Register the NetBeans IDE after finishing the installer, si nous désirons nous enregistrer gratuitement afin de recevoir les informations de mise à jour pour ce produit. Pour terminer l'installation, nous cliquerons enfin sur Finish.

L'icône représentée à la figure E-11 devrait être alors apparaître sur le Bureau (dans le cas contraire, nous pourrons facilement créer un raccourci vers NetBeans). Pour démarrer NetBeans, il suffira alors de double-cliquer dessus.

Figure E-11

NetBeans – Icône de démarrage

Au premier lancement de NetBeans, nous pourrions rencontrer le message de la figure E-12 si nous avions déjà installé NetBeans ou dans le cas d'une réinstallation ou d'une mise à jour :

Figure E-12

Reprise d'anciennes configurations

Ici, nous conseillerons de cliquer sur le bouton Non afin d'avoir un nouvel environnement vierge. Si nous cliquons sur Oui, tous nos anciens projets seront importés et il faudra sans doute revoir chacun d'entre eux si l'ancienne version de NetBeans est trop différente par rapport à la nouvelle.

La fenêtre de démarrage de NetBeans s'affichera alors (figure E-13) :

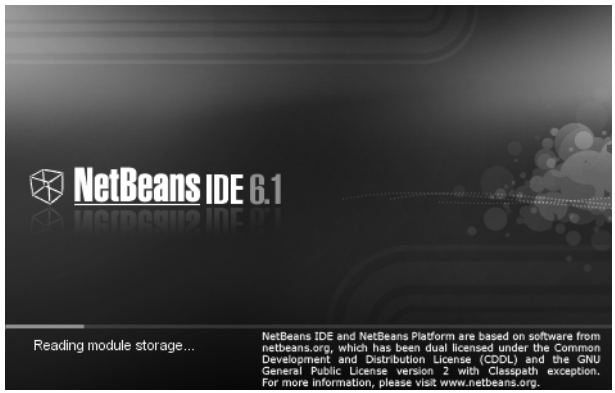

Figure E-13

Démarrage de NetBeans 6.1

Le premier démarrage sera un peu plus lent que les suivants car NetBeans configurera en arrière-plan l'environnement. L'interface de travail par défaut apparaîtra ensuite (figure E-14) :

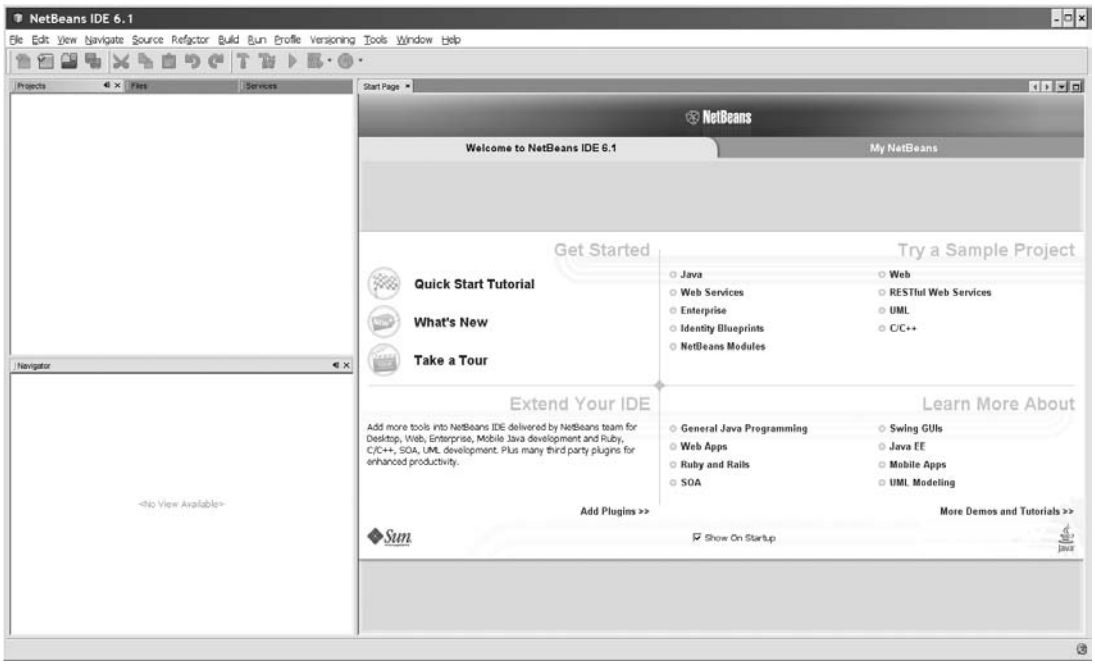

Figure E-14

Interface de travail par défaut de NetBeans

La rapidité d'apparition de cette fenêtre dépendra des ressources de l'ordinateur. À noter qu'un écran de grande définition pourrait aider considérablement à la bonne visualisation des programmes et au travail quotidien.

Une petite icône située en bas à droite de l'interface de travail nous indique que des mises à jour sont disponibles (figure E-15).

Figure E-15

Icône indiquant que des mises à jour sont disponibles.

Nous pouvons alors choisir de les installer en cliquant dessus (il y aura sans doute plus de 18 mises à jour, figure E-16) ou de les ignorer (voir le commentaire situé après la figure E-17) :

Figure E-16

Mise à jour des paquets

Une fois les mises à jour téléchargées et installées, la fenêtre de la figure E-17 apparaîtra, suite à l'installation du plug-in UML :

Figure E-17

Contrat de licence pour le paquet UML

Nous cliquerons ici sur le bouton Cancel afin de nous assurer que la version utilisée dans cette annexe correspond bien à celle décrite dans ce livre. Dès que l'utilisateur sera familier avec l'outil, il est recommandé de procéder à cette mise à jour (l'équipe de développement de NetBeans en propose très régulièrement, il est conseillé de s'enregistrer pour recevoir des e-mails d'information). Si nous acceptons la mise à jour (bouton Update), le lecteur pourrait s'attendre à de petites variations sur les fonctions décrites dans la suite de cette annexe.

Dans la fenêtre principale de NetBeans (onglet Start Page), nous désactiverons ensuite l'option Show On Startup. De cette manière, l'écran de bienvenue ne s'affichera plus lors des prochaines ouvertures de NetBeans. Pour que cette modification soit bien prise en compte, nous fermerons NetBeans (menu File > Exit) et le relancerons. Il peut s'avérer intéressant de visiter les liens proposés sur cet écran de bienvenue avant de l'effacer. Nous y trouverons en effet différentes informations sur le produit.

Au prochain démarrage, nous obtenons l'interface de travail présentée à la figure E-18 :

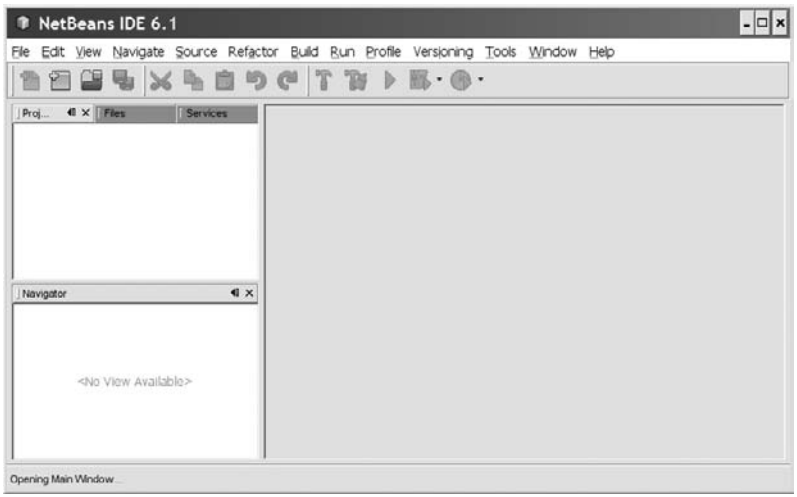

Figure E-18

Interface de travail de NetBeans, vierge au démarrage

L'interface de travail se présentera toujours selon l'aspect, la dimension et la position choisis avant de quitter NetBeans. Suivant la dimension de l'écran, la taille du code source (affiché dans la fenêtre de droite) et les informations indiquées dans les fenêtres Projets, Files ou Navigator, le lecteur pourra redimensionner à souhait chaque fenêtre horizontalement ou verticalement. Pour cela, il faudra placer le curseur de la souris entre les fenêtres, cliquer sur le bouton gauche et déplacer la barre verticale qui apparaîtra alors.

Configuration pour le C++ et le make

Le menu Tools > Options > C/C++ permet de contrôler la configuration de NetBeans et de corriger éventuellement le chemin d'accès du `make` (figure E-19).

Dans l'annexe B, section « Problèmes potentiels avec le make », nous avons mentionné les difficultés susceptibles d'être liées à cet outil, qu'il nous faudrait alors éventuellement remplacer par une autre version, suivant la machine et la version de Windows. Le chemin d'accès indiqué ici semble fonctionner dans tous les cas.

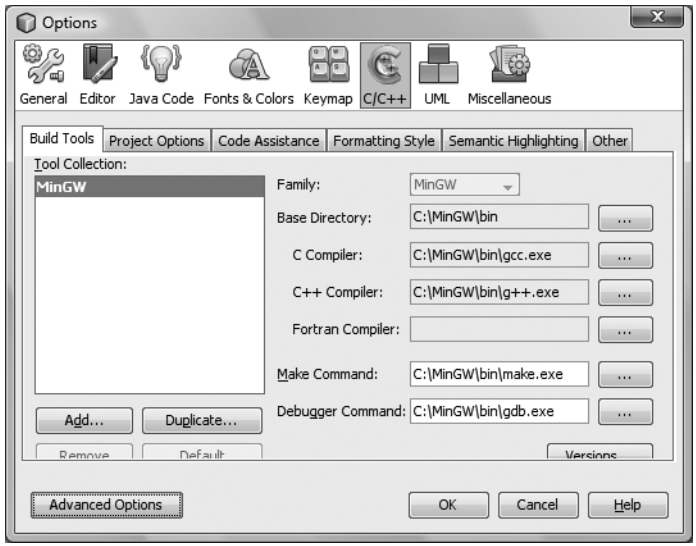

Figure E-19

NetBeans – Configuration du C++ et du make

Présentation de NetBeans

Nous allons ici présenter l'utilisation de NetBeans pour Java et C++ sous Windows Vista (c'est identique sous Windows XP).

Cet ouvrage a été construit avec des petits modules plutôt que des grosses applications ou projets. Il n'a donc pas été envisagé, dans cette édition, d'intégrer tous les chapitres ou tous les exemples dans des projets NetBeans.

En revanche, un programmeur qui acquerra de l'expérience avec Java ou C++ et qui voudra développer une application substantielle y trouvera son bonheur. Il pourra créer un nouveau projet en y intégrant son code ou bien consulter des exemples ou des exercices de cet ouvrage. Pour des applications graphiques Java, le cœur du programme sera sans doute développé avec NetBeans et ensuite intégré avec les classes ou les bibliothèques existantes.

NetBeans et Java

Java et la classe Personne

Pour débuter simplement, nous prendrons l'exemple de la classe `Personne` du chapitre 4.

Nous avons tout d'abord créé un nouveau répertoire `C:\JavaCpp\NetBeans\Chap04a` et copié les fichiers `Personne.java` et `TestPersonne.java` (figure E-20) :

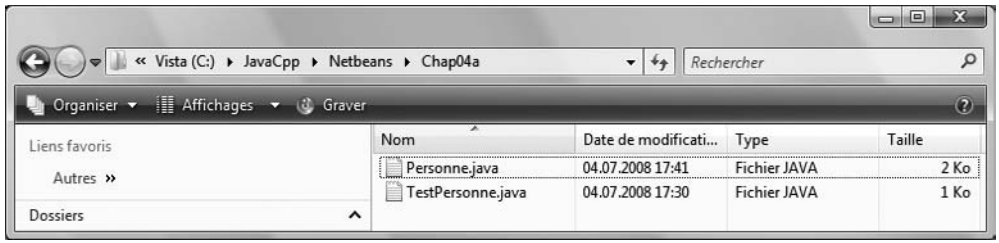

Figure E-20

Préparation d'un projet NetBeans

Lorsque nous éditerons par la suite ces fichiers depuis NetBeans, ils seront modifiés à cet endroit.

Le fichier `Personne.java`, dans ce cas précis, contient du texte avec des accents utilisant malheureusement l'encodage ANSI. Nous conseillons aux lecteurs d'employer l'encodage UTF-8 dans NetBeans. Pour vérifier quel type d'encodage est utilisé, il faut effectuer un clic droit sur le nom du projet, ici `JavaProject4a`, et sélectionner Properties (figure E-21) :

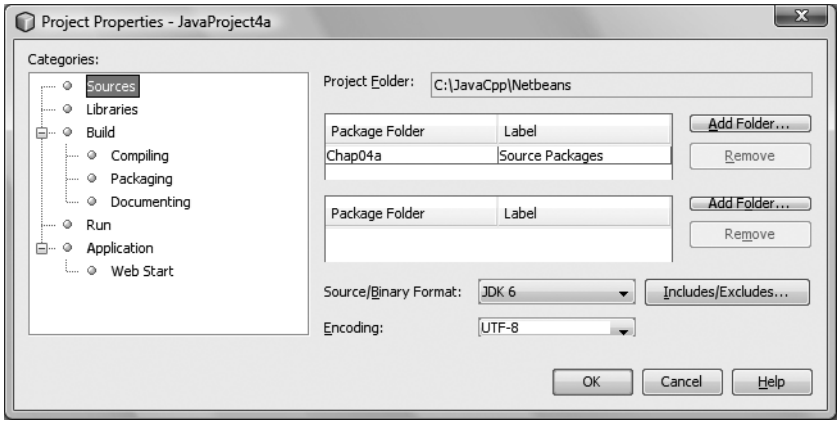

Figure E-21

Configuration de l'encodage UTF-8 dans NetBeans

L'encodage UTF-8 (champ Encoding) est préférable si nous désirons par la suite inclure des textes comportant des caractères particuliers comme c'est le cas pour l'alphabet cyrillique ou arabe. Ces textes peuvent aussi faire partie de la documentation ou de l'exécution du code.

Nous avons deux choix possibles : convertir les fichiers ANSI qui contiennent des accents en UTF-8 ou les conserver tels quels et procéder ensuite à des corrections manuelles dans l'éditeur de NetBeans (voir ci-dessous). Pour une conversion préalable, nous pouvons ouvrir le fichier `Personne.java` avec le Bloc-notes et sélectionner le menu Enregistrer sous en spécifiant le même nom de fichier mais l'encodage UTF-8 (figure E-22) :

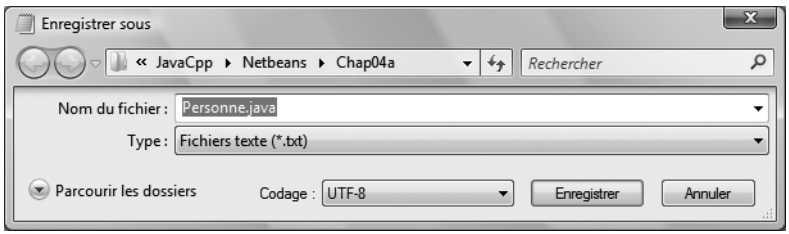

Figure E-22

Sauvegarde du fichier Personne.java avec l'encodage UTF-8

Comme Windows possède son propre type de fichier, il faudra encore apporter une petite correction dans la source et depuis NetBeans. Nous y reviendrons.

Nouveau projet avec source existante

Voici le cas qui nous intéresse ici. En effet, nous aimerions créer un nouveau projet avec du code existant depuis notre répertoire `C:\JavaCpp\NetBeans\Chap04a`. Pour cela, nous sélectionnons le menu File > New project de NetBeans ou saisissons le raccourci clavier Ctrl+Shift+N (figure E-23).

Nous choisirons Java Project with Existing Sources, afin d'utiliser le code que nous venons de préparer.

Nous nommerons le projet `JavaProject4a` en souvenir du chapitre 4, le `a` étant ajouté car nous pensons créer d'autres projets pour ce chapitre par la suite. Le répertoire du projet est important et il est conseillé de n'en spécifier qu'un seul, soit `C:\JavaCpp\NetBeans`, dans lequel tous nos projets seront stockés (figure E-24).

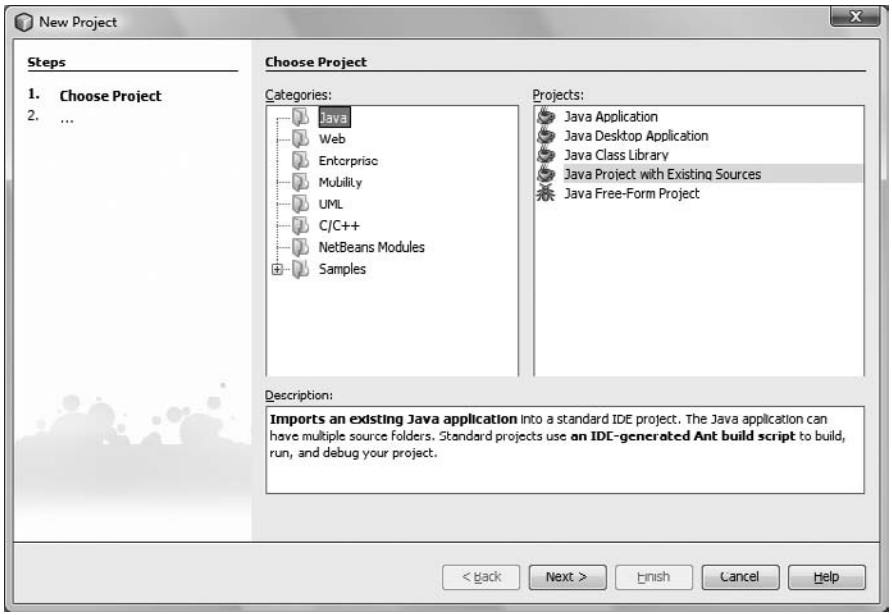

Figure E-23

Création d'un nouveau projet avec source existante dans NetBeans

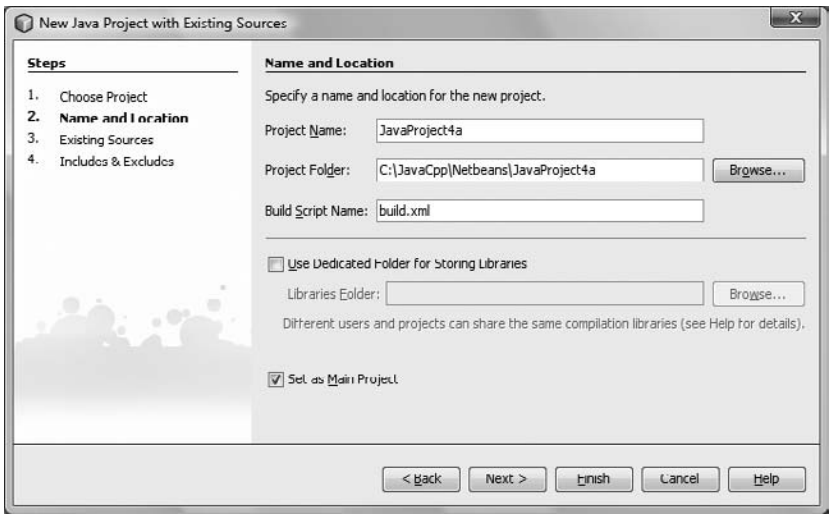

Figure E-24

Spécification du nom et du répertoire du projet NetBeans

Ensuite, nous cliquerons sur le bouton Next > et indiquerons la référence à nos deux fichiers source dans le répertoire créé précédemment (figure E-25) :

Figure E-25

Spécification des fichiers source du projet

Nous cliquerons à nouveau sur le bouton Next > et indiquerons ensuite quels fichiers source nous désirons employer (figure E-26) :

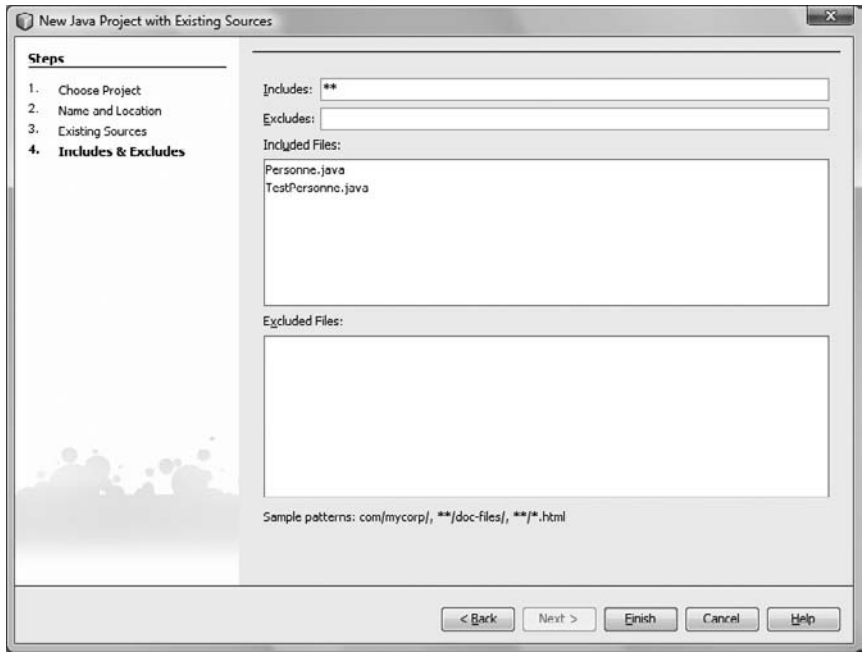

Figure E-26

Récupération de nos deux fichiers Java

Nous cliquerons enfin sur Finish pour terminer.

Pour obtenir cette l'arborescence de la figure E-27, il nous faudra cliquer sur le signe + situé devant <default package>.

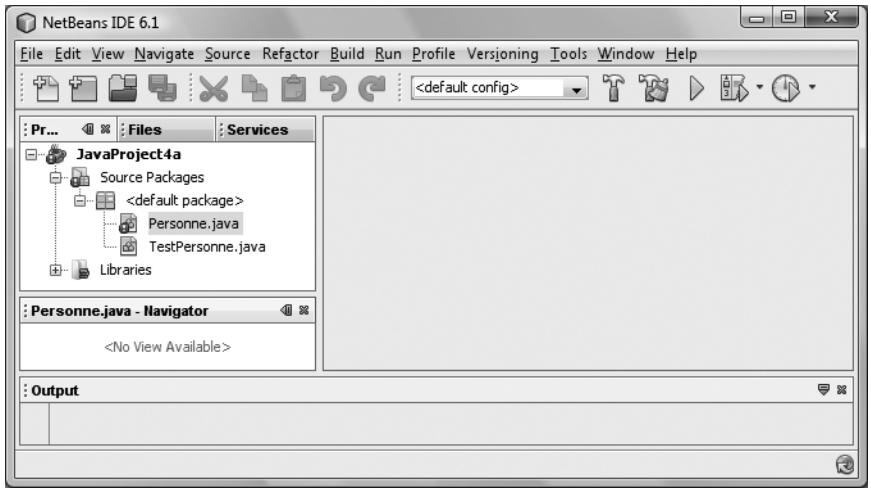

Figure E-27

Projet NetBeans presque prêt

Nous constatons qu'un point d'exclamation rouge est présent devant le fichier `Personne.java`. Ceci indique que le fichier comporte un caractère non reconnu, généré par Windows lors de la sauvegarde sous le Bloc-notes en mode UTF-8. Nous l'effacerons tout simplement.

Nous aurions pu également corriger les caractères accentués non reconnus sans passer par la conversion Bloc-notes. Dans un fichier au codage ANSI, un texte comme « Prénom » apparaîtra sous la forme « Pr□nom » dans NetBeans et il suffira alors de corriger ce caractère non reconnu.

D'ailleurs, nous conseillerons d'utiliser NetBeans pour réaliser la documentation Javadoc et plus particulièrement si celle-ci est en français (figure E-28) (voir le chapitre 4 et la section « Javadoc » de cette annexe).

En cliquant sur la touche F11 du clavier, qui correspond au raccourci du menu Build Main Project, nous allons pouvoir sauvegarder nos éventuelles adaptations de code ou de documentation et construire nos classes Java compilées (figure E-29).

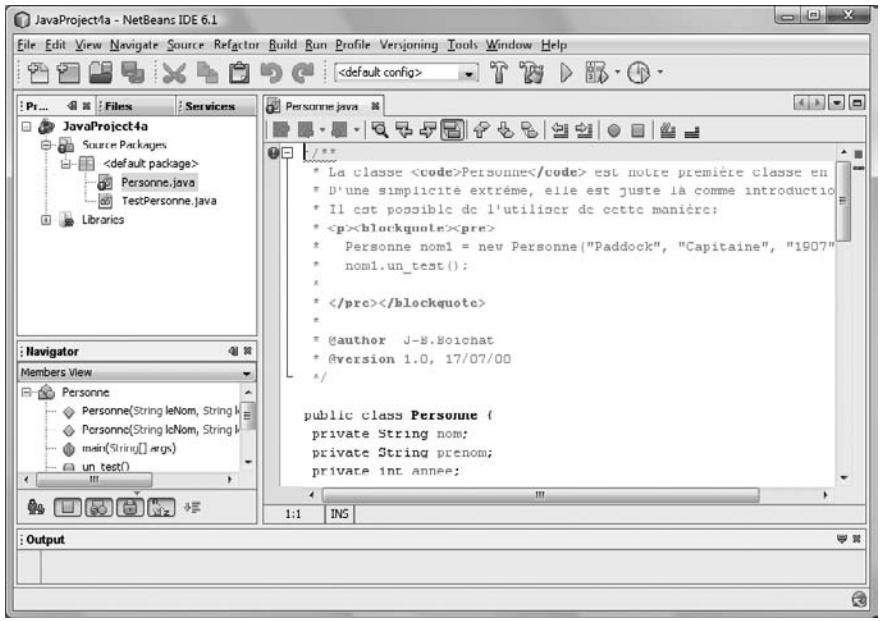

Figure E-28

Les dernières adaptations UTF-8

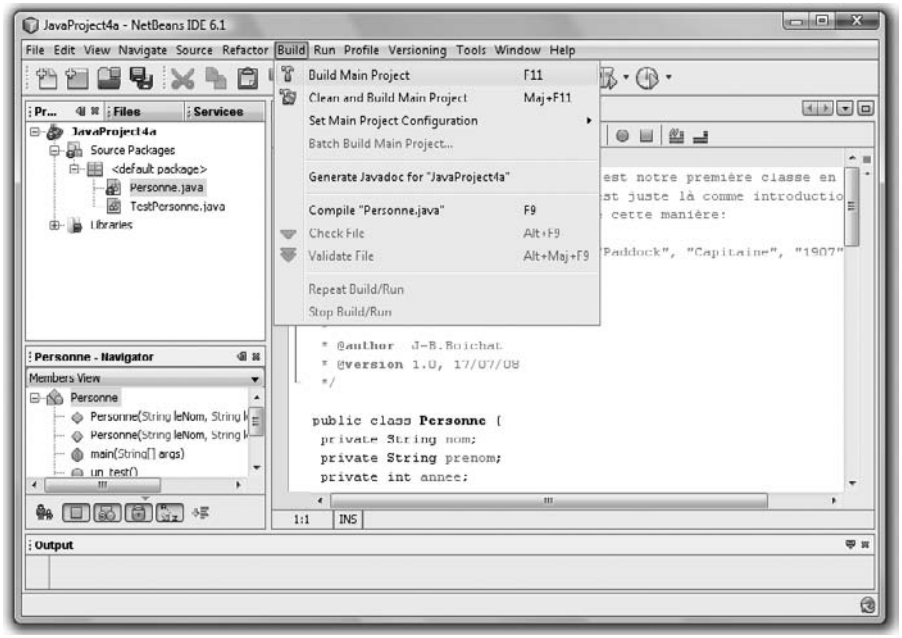

Figure E-29

Le fichier est prêt pour notre première compilation Java.

Après la compilation du projet, si nous naviguons dans l'explorateur de Windows (figure E-30), nous pourrons constater qu'il a été enregistré, ainsi que les fichiers Java, dans le répertoire `JavaProject4a`. Les fichiers compilés sont dans le répertoire `C:\JavaCpp\NetBeans\JavaProject4a\build\classes`.

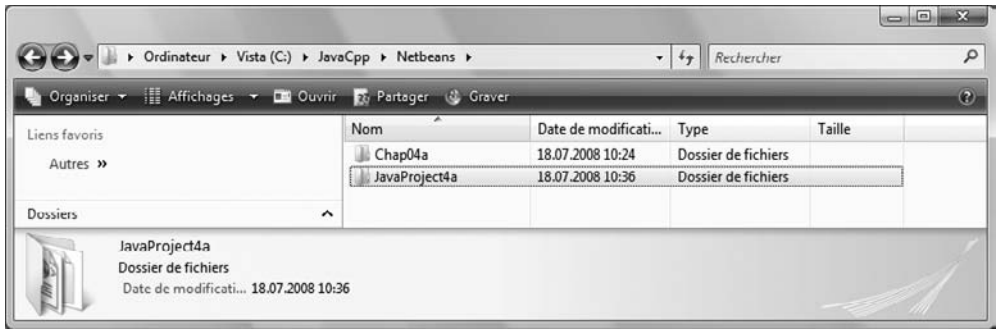

Figure E-30

Nos répertoires de travail sous NetBeans

Le menu Run de NetBeans va nous permettre d'exécuter le projet. Comme nous avons deux classes avec des points d'entrée (`main()`), NetBeans va nous demander depuis quelle entrée du projet nous allons exécuter le programme (figure E-31) :

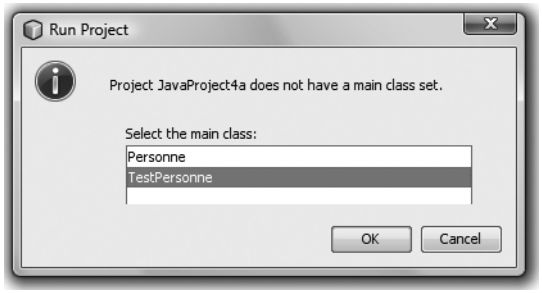

Figure E-31

Sélection de la classe TestPersonne

Nous sélectionnerons `TestPersonne`, notre classe de test :

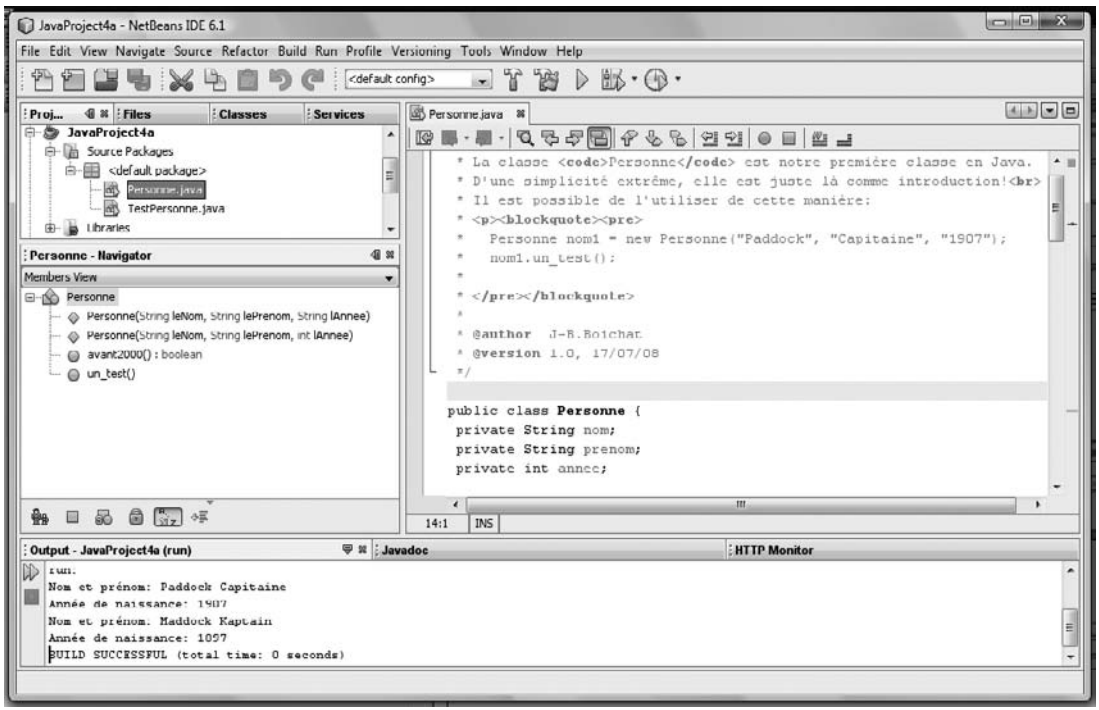

Figure E-32

Première exécution depuis NetBeans

Le résultat de l'exécution de la classe `TestPersonne.java` s'affiche dans la fenêtre Output de NetBeans. Dans le cas présenté, `TestPersonne.java` n'est pas ouvert avec l'éditeur. Ce n'est pas nécessaire et, de plus, tout le code principal de notre projet se trouve dans la classe `Personne`.

Comme il s'agit de la première compilation, les éléments de la fenêtre Navigator sont désormais accessibles et nous allons comprendre rapidement son utilité.

Pour exécuter à nouveau le projet (toujours avec `TestPersonne` comme point d'entrée), nous pouvons sélectionner le menu Run (touche F6) ou cliquer sur l'icône en forme de triangle vert dans la barre d'outils de NetBeans (figure E-32).

Si nous désirons changer le point d'entrée du `main()` pour le projet, il faudra effectuer un clic droit sur le nom du projet dans la fenêtre des projets, en haut à gauche de l'interface de travail, et sélectionner Properties. Nous cliquerons ensuite sur Run, puis sur le bouton Browse… situé à côté du champ MainClass afin de choisir le nouveau point d'entrée (figure E-33).

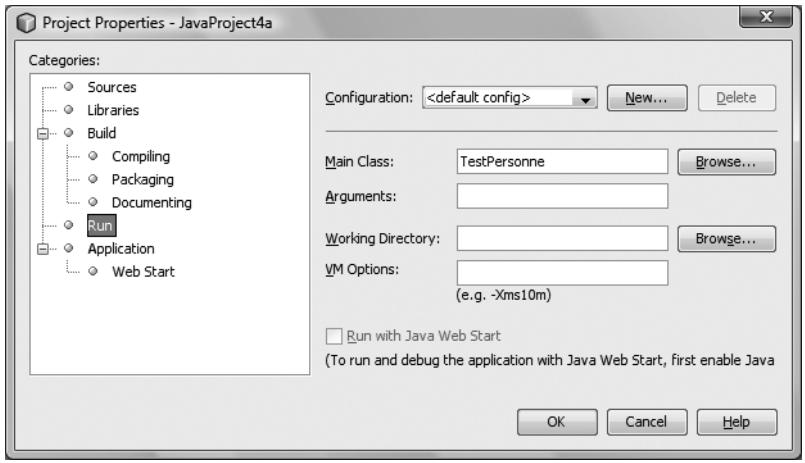

Figure E-33

Modification du main() par défaut

Distribuer nos applications

Une fois le développement de cette application terminé, il pourrait s'avérer intéressant de l'exécuter en dehors de l'environnement de NetBeans et de pouvoir la distribuer à l'extérieur. Nous en avons parlé rapidement à la fin du chapitre 4, section « Nos classes Java dans un paquet .jar » ; NetBeans construit ainsi un paquet .jar et le dépose dans le répertoire de distribution :

```
C:\JavaCpp\NetBeans\JavaProject4a\dist
```

Le fichier est ici nommé `JavaProject4a.jar` et nos amis ou clients pourraient l'exécuter avec :

```
java JavaProject4a.jar
```

à condition qu'une machine virtuelle Java version 1.6 soit installée sur leur PC (une version JRE suffit, voir chapitre 1).

Nous pourrons ensuite ouvrir les fichiers `.jar` avec 7-Zip et consulter leur contenu ainsi que le fichier `MANIFEST.MF` dont nous avons parlé au chapitre 4. Il contient, par exemple, la définition du point d'entrée, ici `Main-Class:TestPersonne`.

Naviguer et déboguer

Nous allons passer maintenant en revue quelques fonctionnalités proposées par NetBeans. La première, sans doute la plus utilisée, consiste à double-cliquer sur le nom du fichier dans la fenêtre Projets, ici `TestPersonne.java`. Le fichier s'ouvre alors pour l'édition et le

fichier précédent, `Personne.java`, reste accessible (il suffit de cliquer sur l'onglet du même nom, figure E-34) :

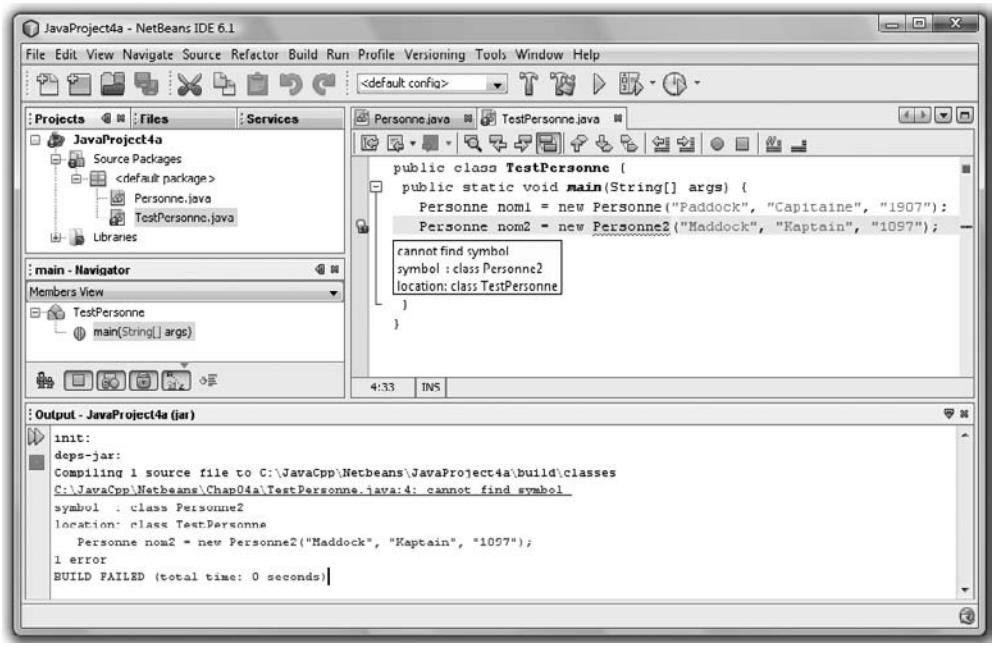

Figure E-34
Naviguer et déboguer

La fenêtre Navigator n'est pas intéressante ici car il n'y a qu'une seule entrée, le `main()`, dans la classe `TestPersonne.java`. Cependant, en fonction du fichier ouvert dans NetBeans, cette fenêtre peut contenir un large choix de méthodes ou d'attributs de classe, comme c'est le cas à la figure E-32. Si nous cliquions sur `un_test()`, la fenêtre de droite, permettant l'édition de `Personne.java`, se positionnerait sur le code de cette méthode de classe. Les différents pictogrammes nous informent des différents types ou attributs (par exemple, `public`, `private` ou autres).

Sur l'exemple de la figure E-34, nous avons ajouté le chiffre `2` après `Personne`, sur la deuxième ligne de code. Une petite ampoule apparaît alors immédiatement devant la ligne : elle nous indique un problème, même sans compiler. En se positionnant sur l'ampoule, ou en compilant le fichier avec le menu Build (le résultat s'affiche alors dans la fenêtre Output), nous verrons rapidement que la classe `Personne2` n'existe pas et que le chiffre `2` est de trop. Nous pouvons également cliquer sur la ligne de l'infobulle `cannot find symbol` : NetBeans se positionnera alors correctement sur la ligne 4 du fichier `TestPersonne.java`. Ceci s'avérera particulièrement pratique lorsque nous aurons beaucoup d'erreurs et dans plusieurs fichiers. Si le fichier comportant les erreurs n'est pas encore ouvert, il le sera automatiquement et le curseur sera positionné sur la ligne posant problème.

Nous supprimons donc le `2` de `Personne` afin de corriger l'erreur et cliquons sur la quatrième ligne de code (`Personne nom2 …`). Nous constatons alors qu'un carré rouge est apparu devant la ligne (figure E-35) :

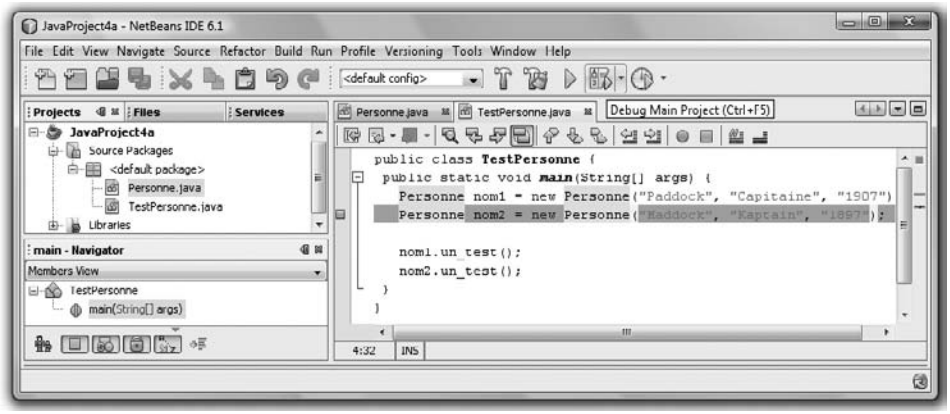

Figure E-35

Déboguer : se positionner au bon endroit

Ce carré rouge indique un point d'arrêt du débogueur. Le programme peut maintenant être lancé avec l'icône Debug (Ctrl+F5), également accessible depuis le menu Run ou en cliquant sur le bouton à droite du triangle vert de la barre d'outils.

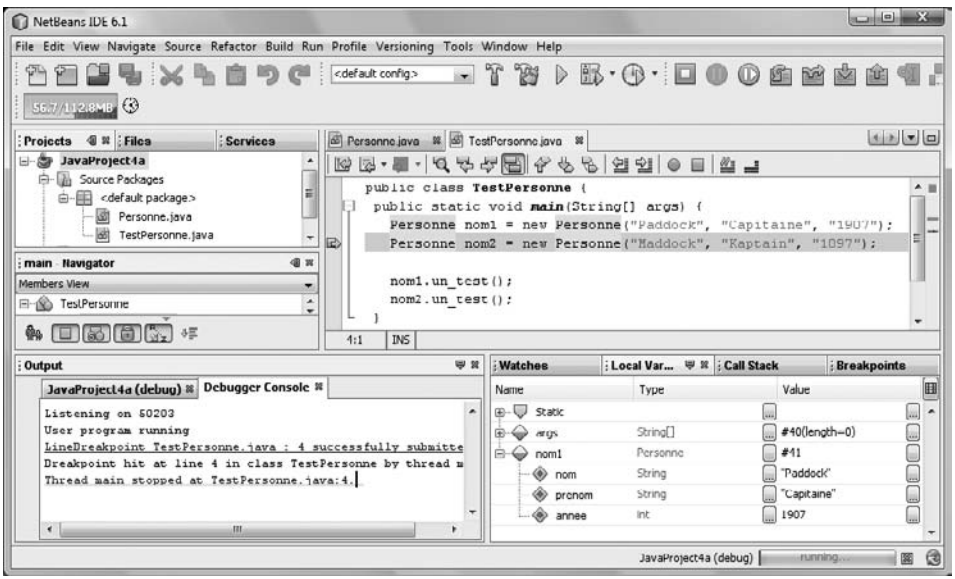

Figure E-36

Nous faisons du pas à pas.

Le programme s'est arrêté sur la ligne spécifiée, mais elle n'a pas encore été exécutée. Dans la fenêtre située en bas à droite de l'interface de NetBeans (figure E-36), nous cliquons sur l'onglet Local Variables afin d'examiner les variables locales déjà initialisées, par exemple l'objet `nom1` de la classe `Personne`. Différentes options sont maintenant possibles, comme le Step Into (F7), symbolisé par une flèche verticale vers le bas (accessible via la quatrième icône en partant de la droite dans la barre d'outils, ou par le menu Run).

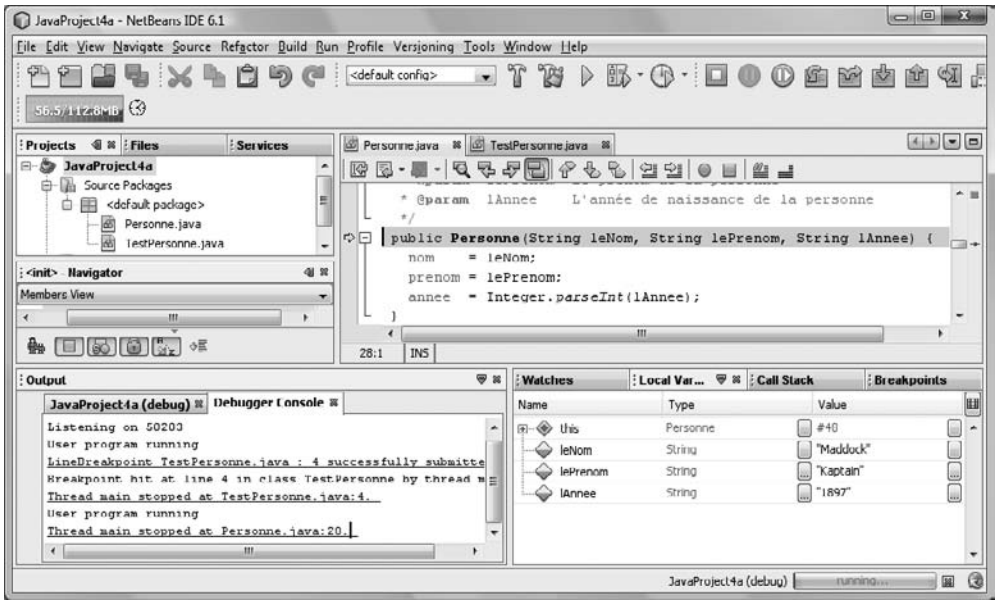

Figure E-37

Pas à pas à l'intérieur d'une classe

L'option Step Into nous a permis d'« entrer dans » le `new Personne("Maddock", "Kaptain", "1897");`, c'est-à-dire le constructeur de la classe `Personne` pour ce nouvel objet `nom2`. La variable `leNom` est bien `"Maddock"` : nous sommes à l'entrée du constructeur.

Nous laissons le lecteur faire lui-même d'autres découvertes : il peut ainsi continuer le programme (F5) et le terminer, pendant une pause de débogueur ou à n'importe quel instant si d'autres points d'arrêt n'ont pas été introduits entre-temps. D'autres aspects sont présentés ci-après pour le C++ et suivent les mêmes concepts (par exemple, Build and Clean Main Project du menu Build).

Javadoc

Pour notre classe `Personne` du chapitre 4, nous avons édité la documentation Java avec Crimson. Dans ce même chapitre, nous avons donné quelques indications pour coder correctement les mots-clés (par exemple, `@param` ou `@return`). Nous allons à présent montrer

toute la puissance d'un outil comme NetBeans pour nous aider à faire ce travail correctement et proprement.

La première chose à faire est d'activer la fonction Javadoc via le menu Tools > Options > Java Code > Hints de NetBeans (figure E-38) :

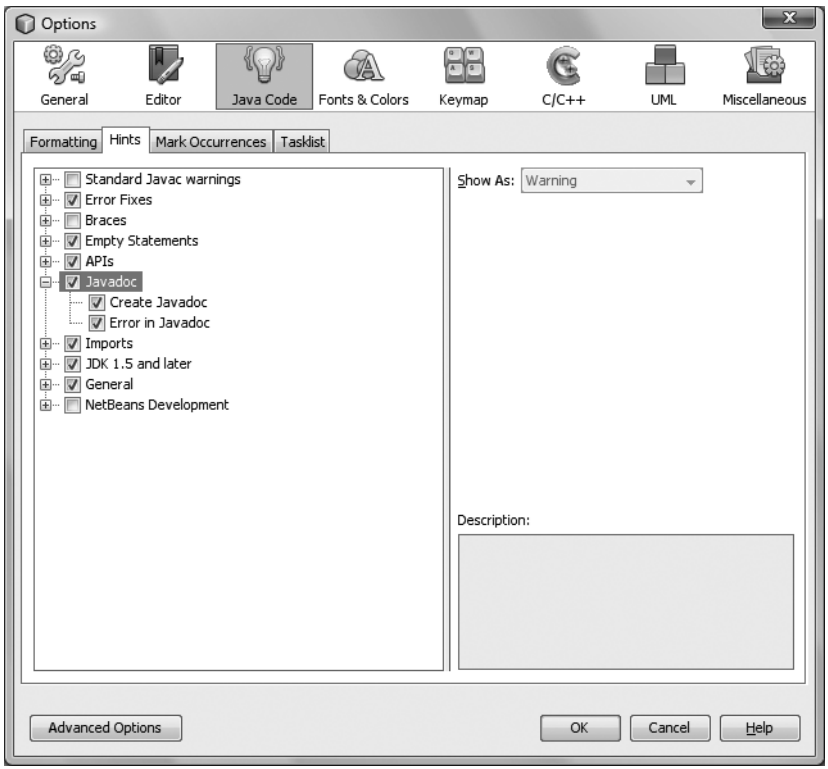

Figure E-38

Activation de la fonction Javadoc

En guise d'exemple, nous allons ajouter une nouvelle méthode appelée `avant2000()` qui va nous retourner un `boolean` (`true` ou `false`, vrai ou faux), nous indiquant si la personne est née avant l'an 2000.

```
public boolean avant2000() {
     if (annee < 2000) {
         return true;
     }
     return false;
 }
```

Lors de l'édition de cette méthode, nous constatons que NetBeans nous aide durant la saisie du texte. Par exemple, si nous définissons la méthode sans son contenu, ce que nous faisons généralement pour nous assurer que les accolades (`{ }`) sont bien présentes, NetBeans va nous indiquer que le retour manque (`missing return statement`) si nous nous positionnons sur le point d'exclamation rouge dans la marge. Nous aurons d'autres indications si le `f` du `if` ou bien une parenthèse est omis. Ceci s'avère très utile, car nous n'avons pas besoin d'attendre la compilation pour corriger la faute.

Lorsque ce code est terminé, nous cliquons sur `avant2000()(`la première ligne de la méthode). Une petite ampoule apparaît alors devant la ligne, nous indiquant que la documentation manque. Il faudra alors cliquer sur la première ampoule puis sur la seconde qui apparaîtra ensuite (figure E-39) :

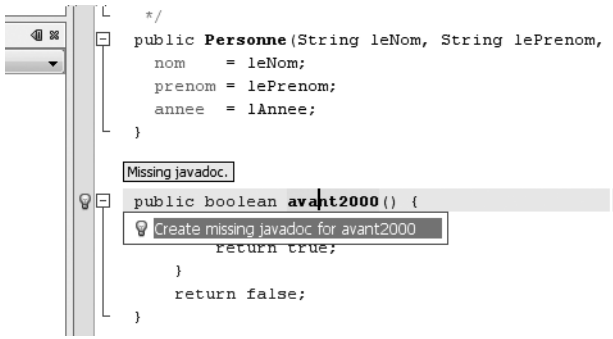

Figure E-39
Générer du Javadoc manquant

afin que le bloc de texte prêt pour l'entrée de la documentation soit généré automatiquement :

```
/**
 *
 * @return
 */
```

Il suffira alors de saisir correctement le texte désiré.

Grâce au menu Window > Other > Javadoc View, nous pouvons afficher la documentation formatée de notre méthode `avant2000()` dans la fenêtre Output (figure E-40) :

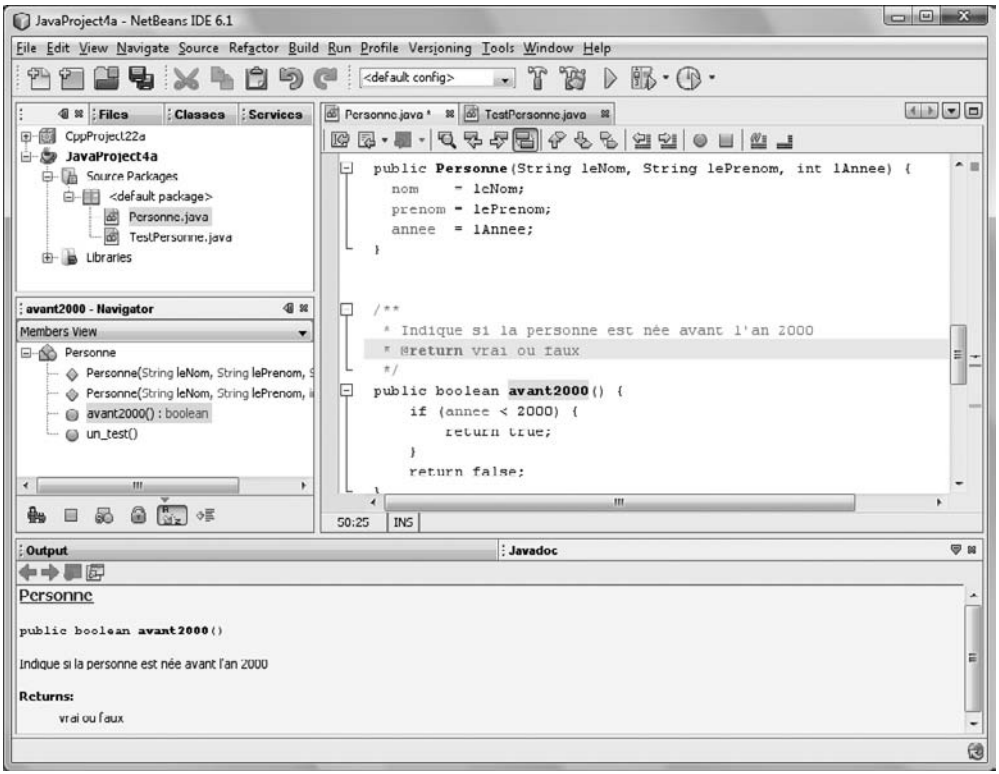

Figure E-40

Saisir et visualiser le Javadoc

En effectuant un clic droit sur le projet `JavaProject4a` dans la fenêtre en haut à gauche, nous pourrons sélectionner l'entrée Generate Javadoc. Si nous cliquons dessus, nous obtiendrons la fenêtre représentée à la figure E-41.

La documentation pour nos méthodes `public`, en particulier la nouvelle méthode `avant2000()`, est ainsi générée et disponible dans le répertoire `C:/JavaCpp/NetBeans/JavaProject4a/dist/javadoc`.

Cette manière de faire est évidemment plus conviviale et efficace que celle décrite au chapitre 4, plus traditionnelle.

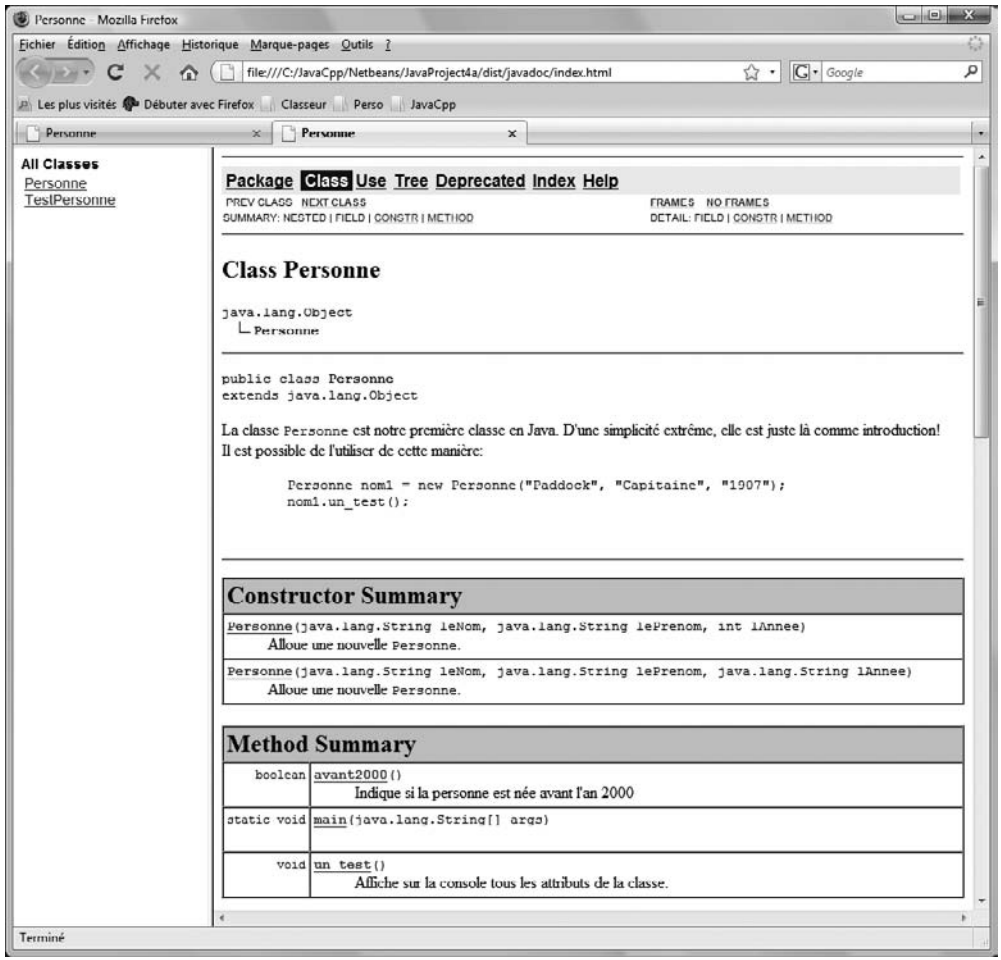

Figure E-41

Javadoc final pour la classe Personne

UML – Diagramme de classe

Nous allons maintenant présenter rapidement un exercice de *Reverse Engineering* : générer des diagrammes UML (*Unified Modeling Language*) et en particulier des diagrammes de classe à partir d'un code existant. Notre classe `Personne` représente un bon exemple pour ce genre d'exercice et sa mise en pratique sera très facile grâce à NetBeans.

Nous commencerons par effectuer un clic droit sur le projet `JavaProject4a` dans la fenêtre en haut à gauche, afin d'accéder à la fonctionnalité Reverse Engineer… (figure E-42).

Nous conserverons les paramètres par défaut et cliquerons sur le bouton OK.

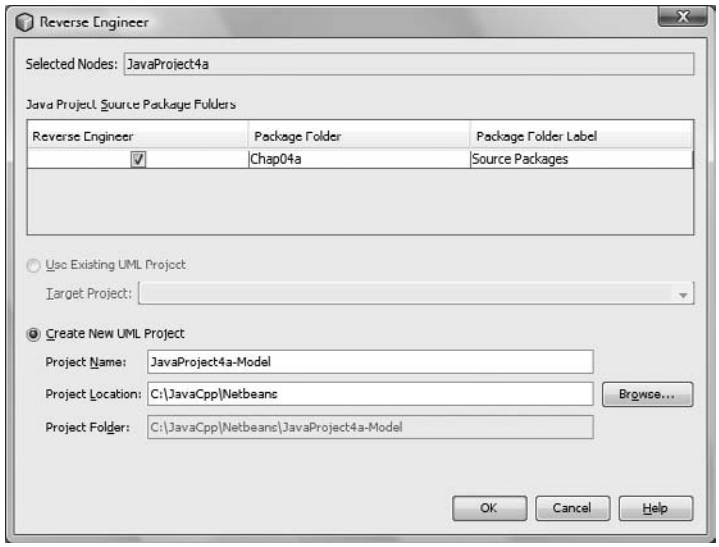

Figure E-42

NetBeans prêt pour du Reverse Engineering

Nous ouvrirons ensuite le projet `JavaProject4a-Model` (créé lors du Reverse Engineering), le `Model` (le répertoire des diagrammes UML), et effectuerons un clic droit sur la classe `Personne` afin de sélectionner Create Diagram from Selected Elements :

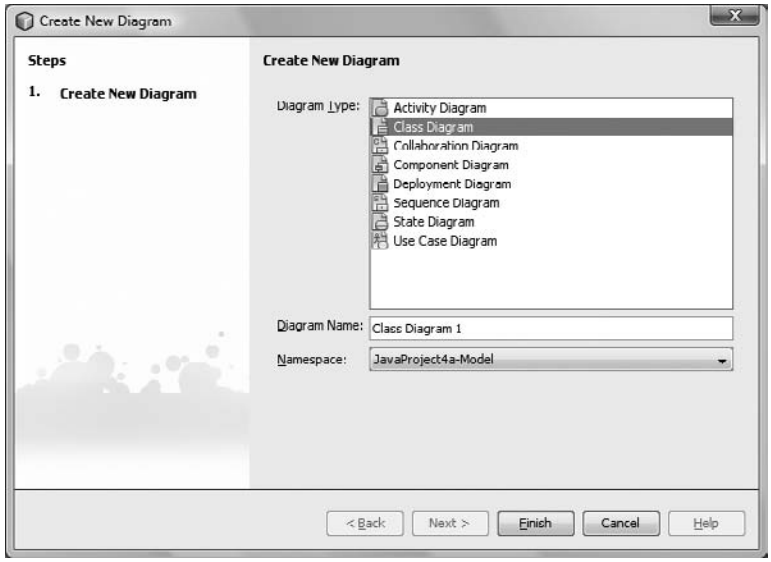

Figure E-43

Diagramme de classe à créer

Nous sélectionnerons Class Diagram et cliquerons sur le bouton Finish. La fenêtre de la figure E-44 s'affichera alors :

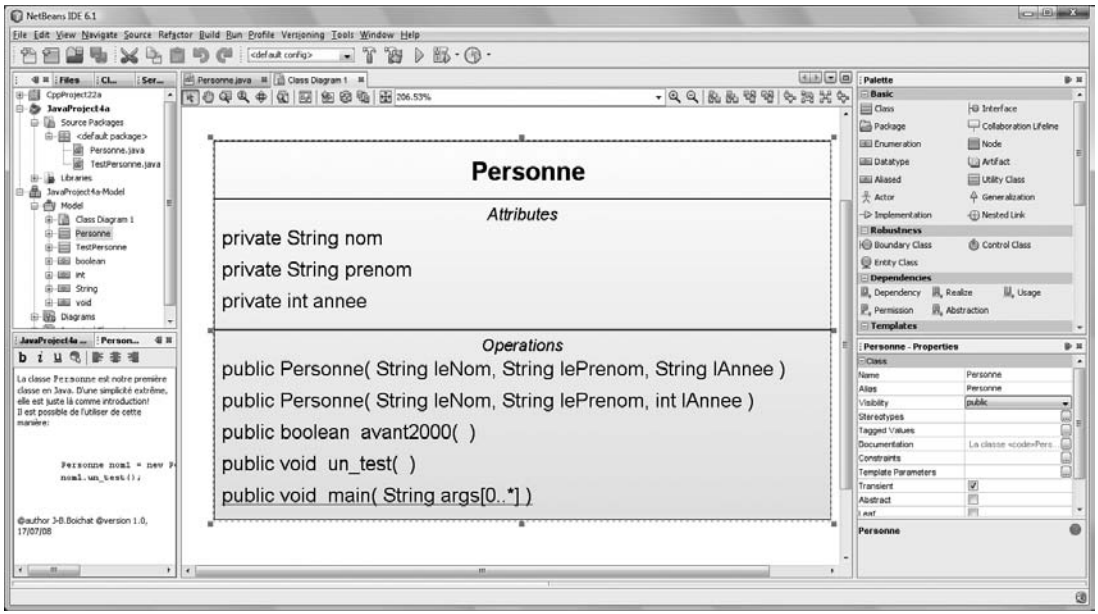

Figure E-44

Diagramme pour la classe Java Personne

Nous laisserons le lecteur affamé de technologie découvrir le monde UML avec cette référence sur le Web :

`http://fr.wikipedia.org/wiki/Unified_Modeling_Language`

L'étape suivante serait évidemment de définir ses propres classes ou d'étendre la classe `Personne` à partir de ces diagrammes UML, et de laisser NetBeans générer le code Java.

NetBeans et C++

Le jeu d'Othello dans NetBeans

Pour le C++, nous allons procéder de la même manière que pour Java mais avec le fichier `othello2.cpp` du chapitre 22. Comme pour l'exercice avec Java, nous allons copier ce fichier dans le répertoire `C:\JavaCpp\NetBeans\Chap22a`.

NetBeans va nous demander un `Makefile` à importer, contrairement à l'exemple détaillé pour Java. Nous allons donc copier ce `Makefile` dans `C:\JavaCpp\NetBeans\Chap22a` à partir des exemples du chapitre 22 et ne garder que la partie concernant le C++ et notre fichier `othello2.cpp` comme ceci :

```
all:  cpp

cpp:  othello2.exe

othello2.exe:   othello2.o
                g++ -g -o othello2.exe othello2.o

othello2.o:     othello2.cpp
                g++ -g -c othello2.cpp
clean:
                rm -rf *.o *.exe
```

Les entrées `all` et `clean` sont requises par un `Makefile` standard, mais absolument par NetBeans.

Par rapport à la version d'origine, nous avons ajouté `-g` après chaque `g++`. Lors de la compilation, ce paramètre va nous permettre un débogage pas à pas avec le débogueur `gdb` que nous avons intégré à notre distribution MinGW. Pour une installation personnalisée depuis Internet (voir annexe B), il faudra sans doute rechercher et installer cet outil séparément.

Nous pouvons très bien charger ce « projet » dans Crimson et vérifier qu'il fonctionne (figure E-45) :

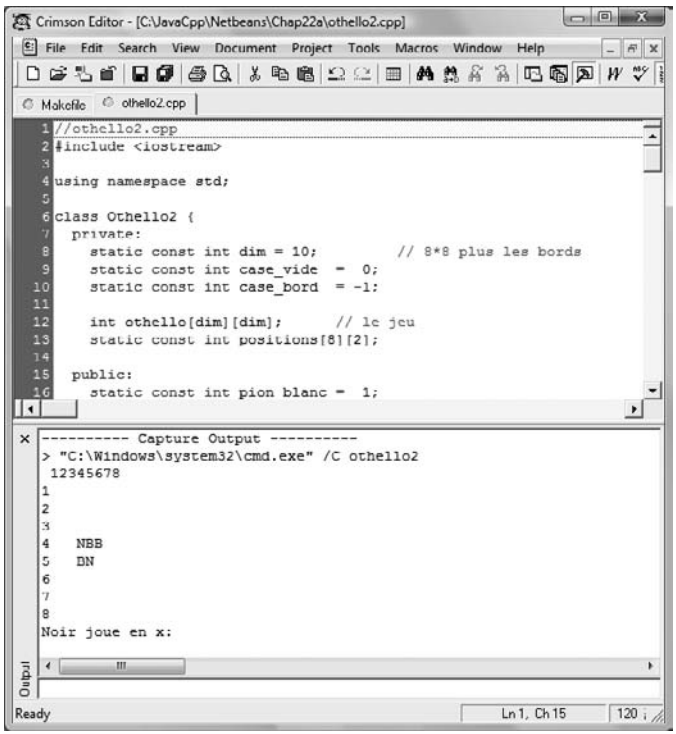

Figure E-45

Vérification avec Crimson

Nous avons laissé volontairement le programme `othello2.exe` actif dans Crimson afin de montrer un cas particulier. Si le fichier `othello2.exe` est actif et chargé, il est possible que nous l'oubliions. Il faudrait alors ouvrir le Gestionnaire des tâches de Windows (Ctrl +Alt+Suppr) et le stopper manuellement en cliquant sur le bouton Arrêter le processus (figure E-46) :

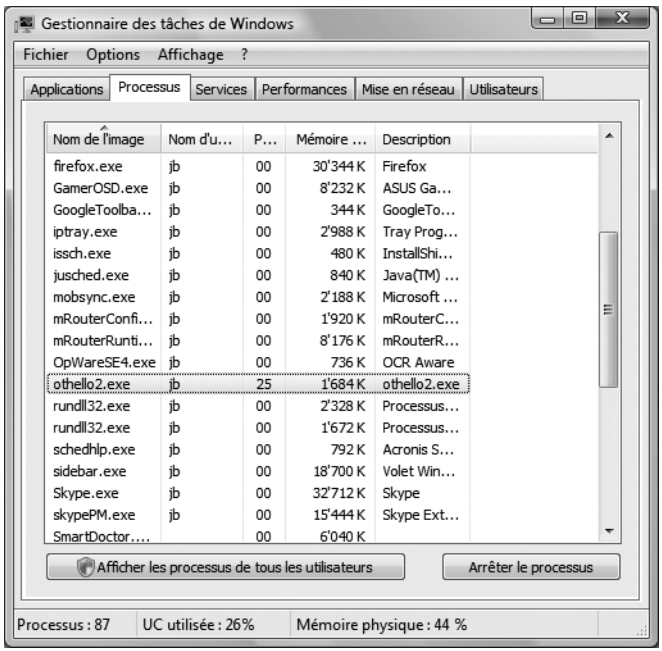

Figure E-46

Arrêt d'un processus dans le Gestionnaire des tâches de Windows

Il arrive parfois qu'un thread (processus parallèle) reste actif en tâche de fond, notamment lors de développement de programmes C++. Cette procédure doit donc être connue.

Création du projet C++ dans NetBeans

Nous allons à présent créer un nouveau projet dans NetBeans via le menu File > New Project (figure E-47). Il est également possible d'utiliser le raccourci clavier Ctrl+Shift+N.

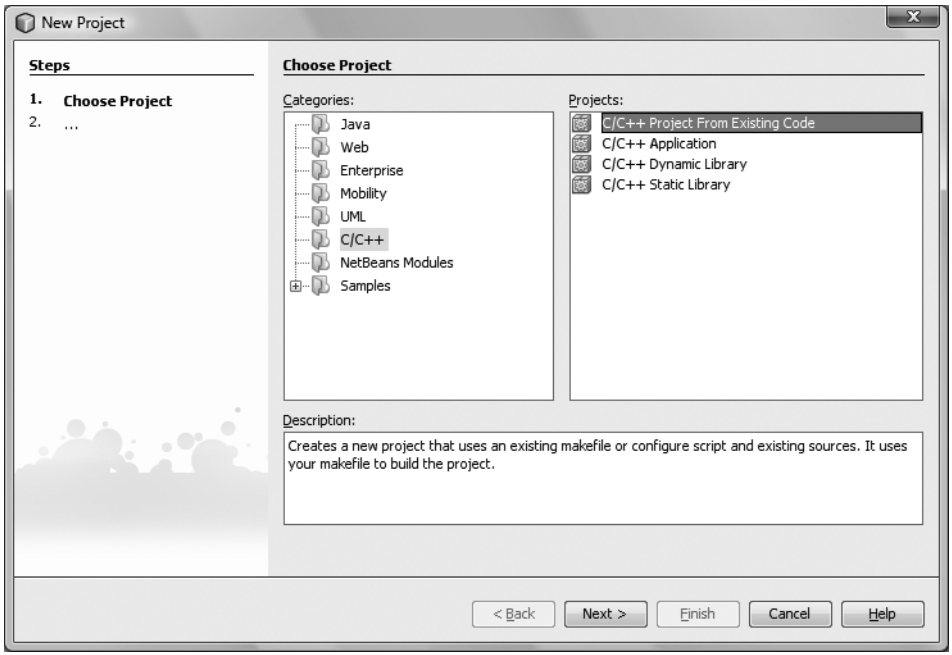

Figure E-47

Project C++ avec du code existant

Le projet C++ de NetBeans nécessite un `Makefile` que nous venons de préparer (figure E-48).

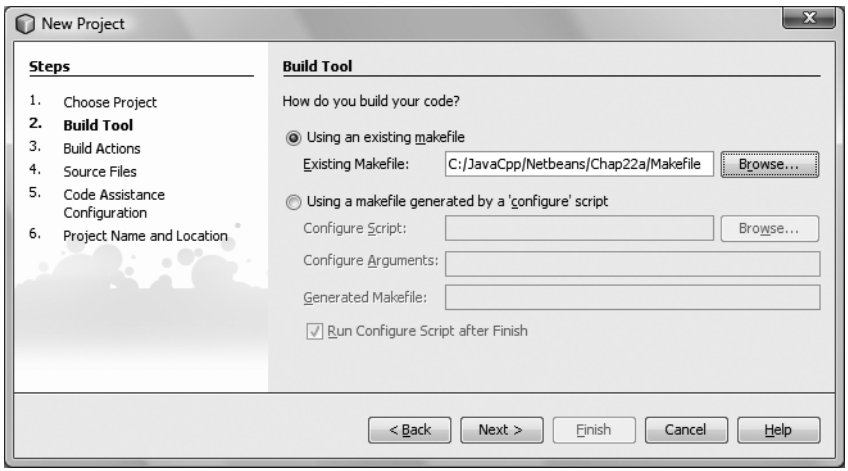

Figure E-48

Sélection du Makefile existant

Nous cliquerons ensuite sur le bouton Next > afin d'obtenir la fenêtre de la figure E-49.

Nous conserverons le répertoire de travail spécifié par défaut : c'est ici que les adaptations effectuées dans le fichier `othello2.cpp` seront effectives. Pour continuer, nous cliquerons sur le bouton Next >.

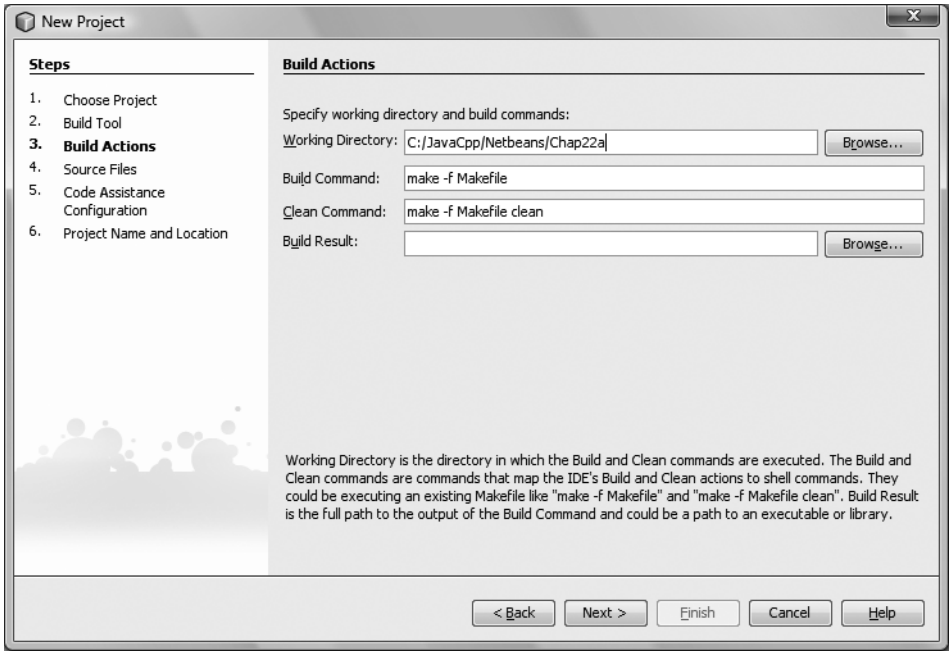

Figure E-49
Le répertoire des sources

La fenêtre de la figure E-50 apparaîtra alors. Nous n'ajouterons pas d'autres répertoires et continuerons en cliquant sur Next >.

Dans la fenêtre suivante (figure E-51), nous sélectionnerons la configuration manuelle et cliquerons à nouveau sur le bouton Next >.

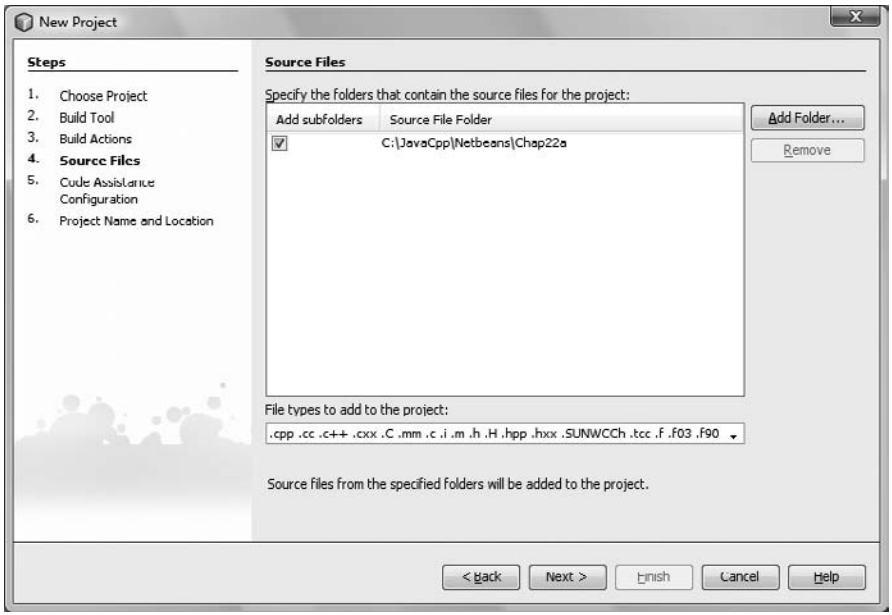

Figure E-50

Pas d'autres sources

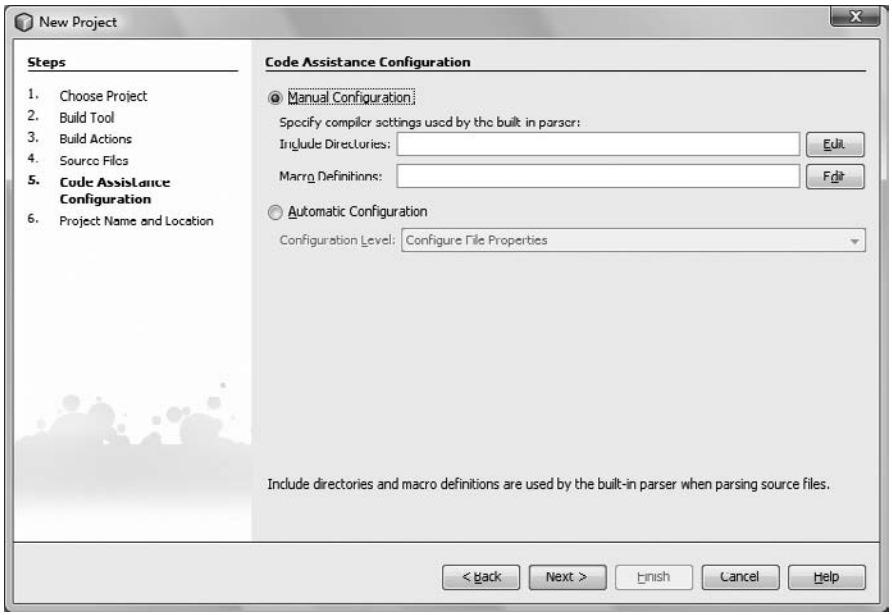

Figure E-51

Configuration manuelle de notre projet C++

Dans la fenêtre suivante (figure E-52), par analogie avec notre projet Java, nous nommerons le projet `CppProject22a` (le répertoire racine de travail reste le même) :

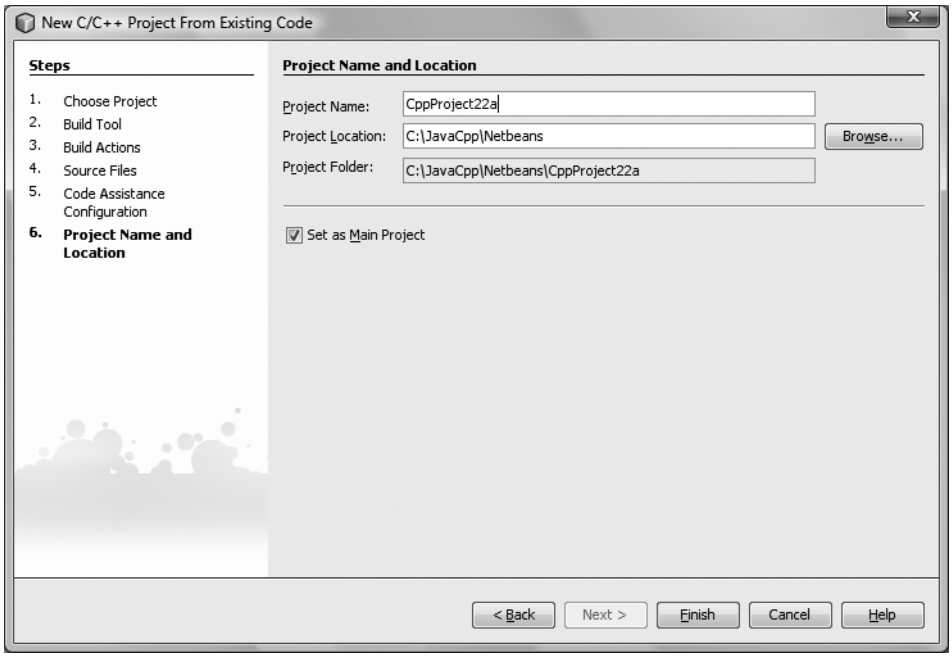

Figure E-52

Nom du projet C++

Nous vérifierons que l'option Set as Main Project est bien activée et nous cliquerons enfin sur le bouton Finish pour terminer la procédure.

Le projet `CppProjet22a` est maintenant marqué comme projet principal (il apparaît en gras dans la liste des projets) voir figure E-53. Si nous avions gardé notre projet Java tel quel, nous aurions encore les deux fichiers Java ouverts dans leurs onglets (comme sur la figure E-53).

Si nous voulons revenir sur l'ancien projet `JavaProject4a`, il suffit d'effectuer un clic droit sur le nom du projet et de sélectionner Set as Main Project dans le menu contextuel.

Si nous sélectionnons Build (Construction) dans le menu contextuel (figure E-54), seul le projet principal sera compilé.

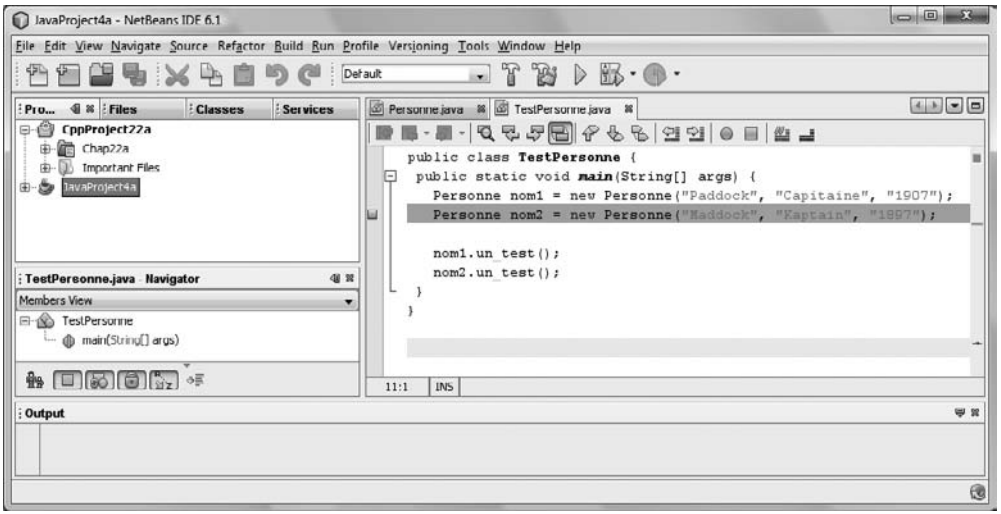

Figure E-53

Du nettoyage en vue

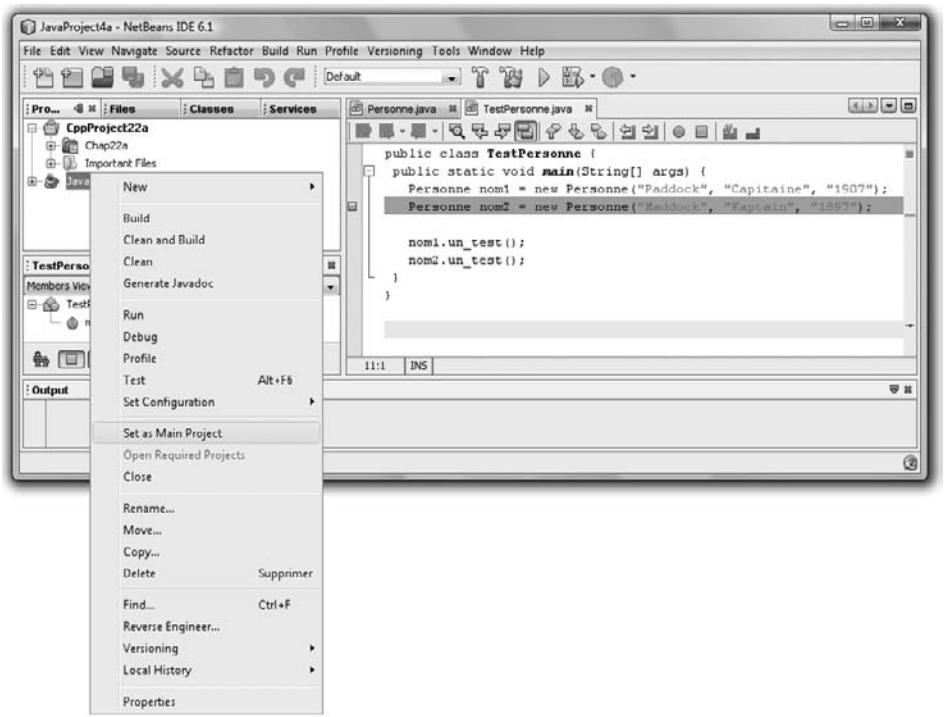

Figure E-54

Définition du projet actif

Nous conservons `CppProjet22a` comme projet principal et fermons les fichiers Java en cliquant sur la croix située à droite de leurs onglets respectifs dans la fenêtre principale de NetBeans. Nous déroulons ensuite `Chap22a` dans la fenêtre Projets et cliquons sur `othello2.cpp` pour le faire apparaître dans la fenêtre d'édition (figure E-55) :

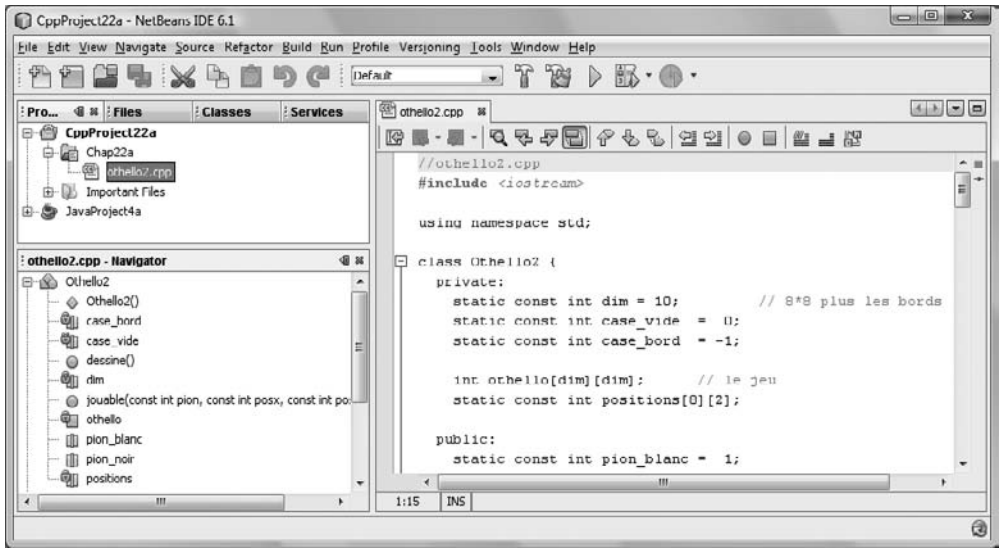

Figure E-55

Notre source C++ dans NetBeans

Dans la fenêtre Navigator, nous retrouvons les mêmes fonctionnalités que pour Java : les méthodes et autres attributs sont présentés. En double-cliquant sur `dim`, par exemple, la ligne de code où la variable statique est définie sera activée dans la fenêtre d'édition.

Nous allons maintenant ouvrir le fichier `Makefile` pour le visualiser dans l'éditeur. Pour cela, nous cliquerons sur le signe + situé devant `Important Files` dans la fenêtre Projets (figure E-56).

Le menu Build de NetBeans contient deux entrées majeures : Build Main Project (F11) et Build and Clean Main Project (Maj+F11). Nous retrouvons le concept du `make` : la première entrée du menu recompile uniquement ce qui a changé ; la seconde appelle le `make clean`, pour effacer tous les objets et ainsi s'assurer qu'une compilation complète sera exécutée.

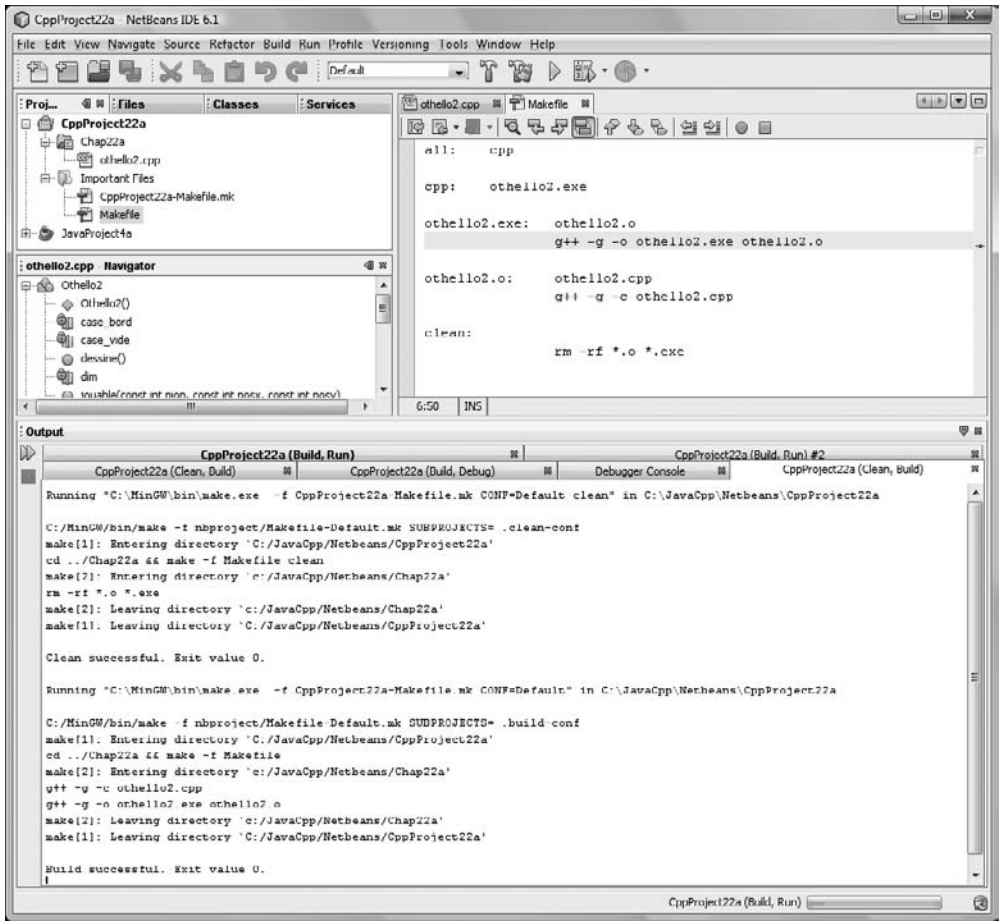

Figure E-56

Premier make de notre projet C++

Dans la figure E-56, nous venons d'exécuter un Build and Clean Main Project. Ce qui est important dans cet exemple qui précède, c'est l'`Exit value 0` que l'on voit dans la fenêtre Output : la compilation a été effectuée correctement.

De la même manière que pour la compilation Java, si nous avions une erreur lors du Build, nous pourrions l'identifier facilement et naviguer en cliquant sur le lien dans la fenêtre Output (figure E-57) :

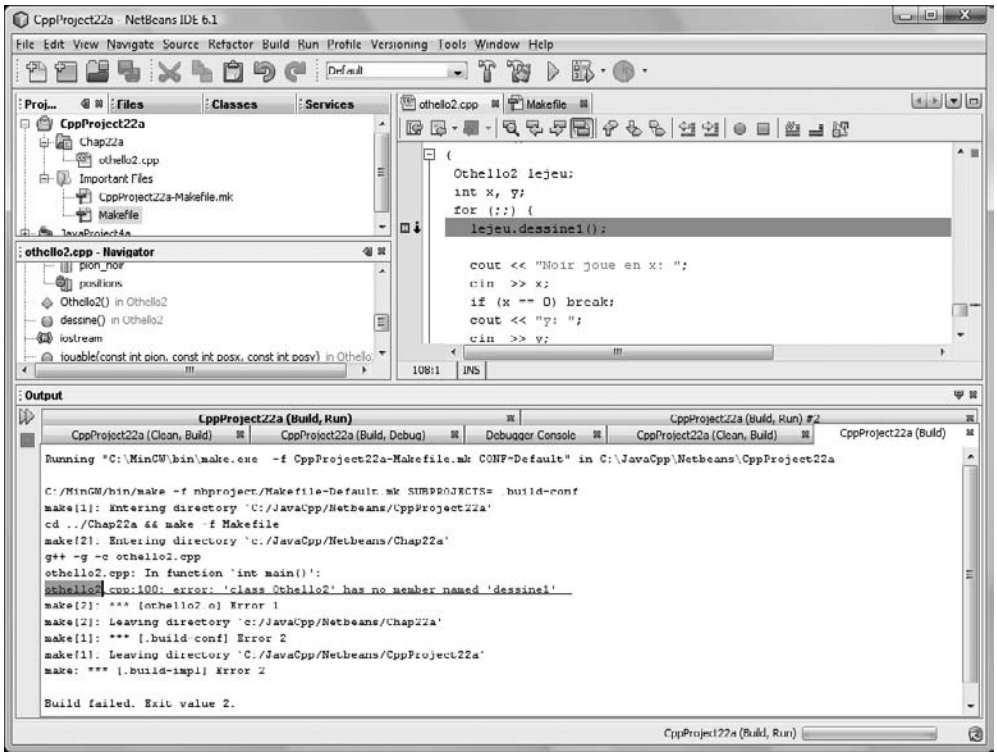

Figure E-57

Correction avec l'éditeur et après compilation

Ici, la méthode `dessine1()` n'existe pas et doit être corrigée en `dessine()`.

Déboguer un projet C++ avec NetBeans

Si nous avons étudié ce programme, et si nous l'exécutons depuis NetBeans (menu Run) ou depuis le navigateur de Windows (l'exécutable `othello2.exe` se trouve dans le répertoire `C:\JavaCpp\NetBeans\Chap22a`), nous savons déjà que le point faible est le contrôle de la validité des saisies utilisateur (figure E-58).

Figure E-58

Exécution du fichier othello2.exe

Si nous saisissons la lettre `a` au lieu d'un chiffre compris entre 1 et 8, nous rencontrerons un sérieux problème, car notre code ne traite pas tous les cas d'erreurs. Nous n'avons pas besoin de débogueur pour cela, il manque définitivement beaucoup de code pour améliorer cette partie et gérer les mauvaises saisies.

Cependant, pour jouer un peu, nous pouvons tout de même effectuer un pas à pas avec le débogueur (MinGW gdb) après avoir indiqué un point d'arrêt (figure E-59) :

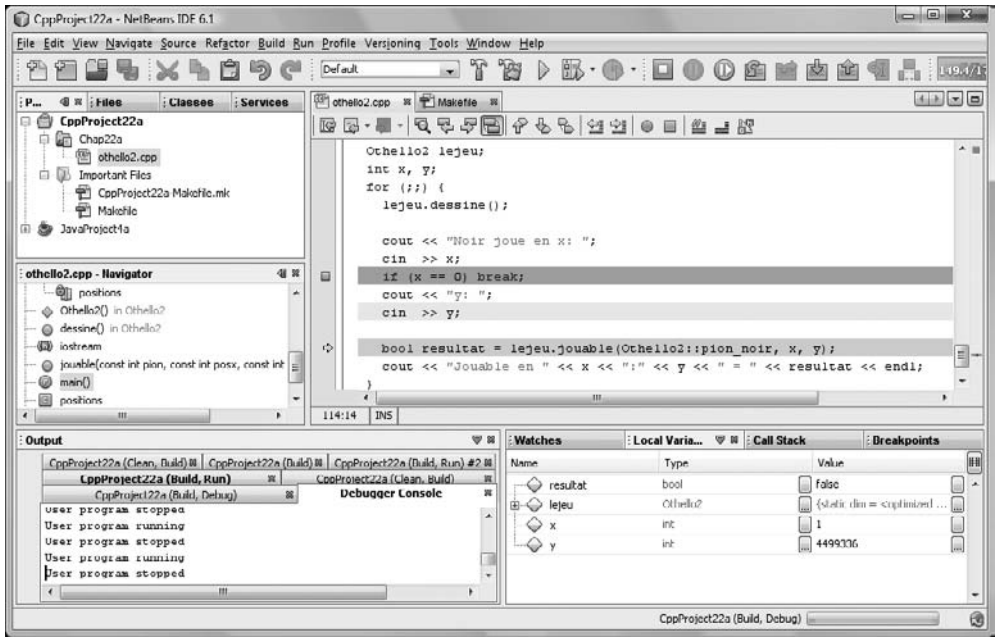

Figure E-59

Utilisation du débogueur en C++

Ici, nous avons atteint l'instruction où les positions `x` et `y` ont déjà été calculées avec notre entrée, une lettre `a` après le `1` ! Le `x` est correct, mais pas le `y` qui est étrange (onglet Local Variables) et la méthode `jouable()` rencontrera une sérieuse difficulté lors de l'exécution de sa première ligne de code.

Nous souhaitons au lecteur beaucoup de succès en jouant un peu plus avec Othello et NetBeans et en effectuant les améliorations nécessaires pour le programme. Écrire un bon programme de jeu pour Othello est un excellent exercice !

Conclusion

NetBeans est un outil formidable et extrêmement convivial et productif. Nous n'avons pas utilisé et décrit toutes ses fonctionnalités et le lecteur devrait y trouver beaucoup de plaisir et très rapidement. Les nombreux didacticiels sur Internet montrent sa popularité actuelle.

Programmer en Java ou en C++ pour des applications complexes nécessite un outil de cette classe. Comme il est Open Source et continuellement mis à jour, n'ayons aucune hésitation. Pour des débutants en Java qui désirent rapidement développer des applications graphiques, NetBeans est sans doute actuellement le meilleur choix.

F

Apprendre Java et C++ avec Linux

Dans cette annexe sur Linux, nous n'allons évidemment pas décrire l'installation d'un tel système sur un PC, ni son utilisation quotidienne, lesquelles ne diffèrent pas beaucoup de Windows XP ou Vista. Il y a suffisamment d'ouvrages sur le sujet, mais nous donnerons tout au long de l'annexe un certain nombre de références sur Internet.

Le but de cette annexe est de donner au lecteur quelques pistes pour installer les outils de compilation pour les langages de programmation Java et C++. Nous donnerons aussi des indications sur les utilitaires d'édition et NetBeans.

En fin d'annexe B, nous avons déjà décrit certaines commandes Linux dans le cadre de leur utilisation avec MSYS et MinGW sous Windows.

L'annexe E sur NetBeans est directement utilisable sous Linux.

La version de Linux que nous avons utilisée est Ubuntu 8.04 (kernel 2.6.24-16-generic). Elle est installée sur un PC en dual boot avec Vista. Comme promis, voici les premières références sur le Web comme pistes de départ. Nous y trouverons aussi bien des généralités, les sites de téléchargement, que des instructions et informations plus détaillées sur son installations et ses nombreux outils :

```
http://fr.wikipedia.org/wiki/Ubuntu_8.04_LTS
http://www.ubuntu-fr.org/
http://www.ubuntu.fr/
http://www.ubuntu-fr.org/telechargement
http://doc.ubuntu-fr.org/installation/vista_ubuntu
```

Démarrage de Linux

Au démarrage, Linux se présentera tel que sur la figure F-1. Il a été configuré pour notre ouvrage avec les outils Java, C++ et NetBeans.

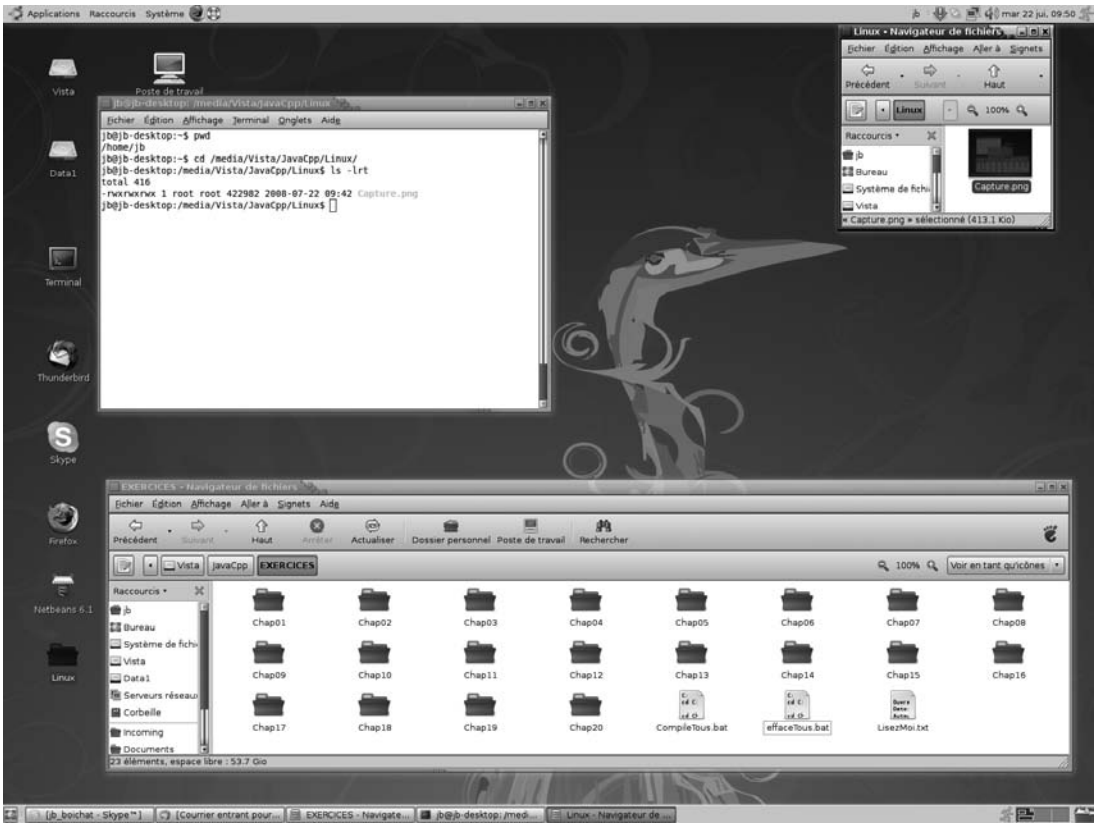

Figure F-1

Ubuntu pour Java, C++ et NetBeans

L'interface utilisateur est GNOME :

`http://www.gnomefr.org/`

Nous voyons que les fichiers Vista sont accessibles. Thunderbird, le logiciel de messagerie électronique, est d'ailleurs configuré ici pour utiliser les mêmes répertoires de configuration et de travail que Vista, c'est-à-dire depuis la partition NTFS. Après quelques heures de travail, nous serons très vite à l'aise avec cet environnement de travail, par ailleurs extrêmement rapide.

La fenêtre située en haut à gauche est une fenêtre Gnome Terminal, lancée au moyen de l'icône Terminal du Bureau. Nous l'utiliserons tout au long de nos explications et exemples qui vont suivre. Nous y trouvons ici les commandes Linux suivantes :

```
pwd
cd /media/Vista/JavaCpp/Linux
ls -lrt
```

- `pwd` – Le répertoire de travail de l'utilisateur `jb` ; il s'agit de `/home/jb`. Nous pouvons y créer des fichiers ou des répertoires (`mkdir`).
- `cd` – Cette commande nous permet de naviguer dans les différents répertoires, ici le répertoire `/media/Vista/JavaCpp/Linux`. Linux, au démarrage, reconnaît les différentes partitions. Le point de montage `/media/Vista` est l'équivalent de `C:` sous Windows XP ou Vista. Nous savons déjà que le caractère `\` de Windows est l'équivalent de `/` de Linux. En fait, nous avons créé un répertoire Linux sous `C:\JavaCpp\Linux`, dans lequel nous avons déposé (comme nous pouvons le constater avec le résultat de la commande `ls -lrt`) l'image du livre pour cette partie Linux, afin de pouvoir la réutiliser depuis Windows pour préparer cet ouvrage. Linux peut accéder, en lecture et écriture, au disque NTFS de Windows (avec les anciennes versions de Linux, nous ne pouvions que lire un disque Windows). En revanche, Windows XP ou Vista ne peuvent pas accéder par défaut au disque Linux.

La fenêtre située en haut à droite nous montre, dans un navigateur de fichiers Linux, le fichier de la première image de cette annexe, soit une capture d'écran du bureau d'Ubuntu. L'autre navigateur, dans la grande fenêtre inférieure, est aussi situé sur la partition de Windows XP où nous avons déposé nos exercices : `/media/Vista/JavaCpp/EXERCICES`, équivalent de `C:\JavaCpp\EXERCICES` sous Windows.

Pour plus d'informations sur les commandes Linux, nous vous conseillons de consulter la section « Les outils Linux de MSYS » de l'annexe B ou encore cette référence :

`http://doc.ubuntu-fr.org/tutoriel/console_ligne_de_commande`

Installation des outils

Trois composants sont nécessaires, lesquels ne sont généralement pas disponibles après une installation standard de Linux :

- les outils pour la compilation de programmes C++ ;
- les outils pour la compilation de programmes Java ainsi que sa machine virtuelle (JRE) ;
- NetBeans.

Nous ne décrirons pas ici l'installation de ces trois paquets : en effet, chaque distribution de Linux a ses propres outils d'installation et manières de faire.

NetBeans est évidemment optionnel. Son installation et son utilisation peuvent se faire plus tard, si désiré.

Pour la partie C++, nous avons choisi l'installation standard de paquets depuis Ubuntu. Dans le cas de Java et NetBeans, nous avons téléchargé les binaires `jdk-6u6-linux-i586.bin` et `netbeans-6.1-linux.sh` depuis leurs sites respectifs :

```
http://java.sun.com/
http://www.netbeans.org/index.html
```

et les avons installés dans le répertoire courant de l'utilisateur `/home/jb`.

Pour Java, la version minimale requise est la version 6 update 6, et pour NetBeans, elle doit être 6.1 ou supérieure (correspondant aux divers exemples et options de l'annexe E).

Nous avons également copié depuis le disque Windows (aussi possible depuis le CD-Rom) tous les chapitres des exemples dans un nouveau répertoire `/home/jb/JavaCpp/EXEMPLES`. Nous pourrons alors les modifier et les utiliser à souhait.

Depuis le répertoire racine de l'utilisateur `jb`, soit `/home/jb`, nous pouvons, via un terminal, consulter le contenu des trois répertoires pour un contrôle rapide :

```
ls JavaCpp jdk1.6.0_06 netbeans-6.1
```

Nous y découvrirons nos trois répertoires copiés ou installés lors de l'exécution des fichiers d'installation de Java et de NetBeans :

```
JavaCpp:
EXEMPLES

jdk1.6.0_06:
…
netbeans-6.1:
…
```

Pour le C++, nous avons procédé à l'installation traditionnelle avec Ubuntu :

```
sudo apt-get install build-essential
```

La commande `sudo` va nous donner accès aux droits d'administrateur. Il faudra alors connaître et entrer son mot de passe.

La commande `apt-get install` va nous permettre d'installer le paquet `build-essential` qui contient les compilateurs C, C++, les bibliothèques et autres utilitaires comme le `make` de nos `Makefile` bien connus (voir chapitre 1).

Les informations que nous avons données, avec les références, devraient suffire pour un nouveau départ avec Linux.

Vérification de l'installation

Les quatre commandes que nous avons déjà utilisées sous Windows nous sont familières :

```
java -version
javac -version
g++ -v
make -v
```

Elles vont nous indiquer que la machine virtuelle de Java, les compilateurs Java et C++ ainsi que le `make` sont bien installés. Voici un résultat possible :

```
jb@jb-desktop:~$ java -version
java version "1.6.0_06"
Java(TM) SE Runtime Environment (build 1.6.0_06-b02)
Java HotSpot(TM) Server VM (build 10.0-b22, mixed mode)

jb@jb-desktop:~$ javac -version
javac 1.6.0_06

jb@jb-desktop:~$ make -v
GNU Make 3.81
Copyright (C) 2006  Free Software Foundation, Inc.
Ceci est un logiciel libre ; voir le source pour les conditions de copie.
Il n'y a PAS de garantie ; tant pour une utilisation COMMERCIALE que pour
RÉPONDRE A UN BESOIN PARTICULIER.
Ce logiciel est construit pour i486-pc-linux-gnu

jb@jb-desktop:~$ g++ -v
Utilisation des specs internes.
Cible : i486-linux-gnu
Configuré avec: ../src/configure -v --enable-languages=c,c++,fortran,objc,obj-c++,
➥treelang --prefix=/usr --enable-shared --with-system-zlib --libexecdir=/usr/lib --
➥without-included-gettext --enable-threads=posix --enable-nls --with-gxx-include-
➥dir=/usr/include/c++/4.2 --program-suffix=-4.2 --enable-clocale=gnu --enable-
➥libstdcxx-debug --enable-objc-gc --enable-mpfr --enable-targets=all --enable-
➥checking=release --build=i486-linux-gnu --host=i486-linux-gnu --target=i486-
➥linux-gnu
Modèle de thread: posix
version gcc 4.2.3 (Ubuntu 4.2.3-2ubuntu7)
```

Les exemples du chapitre 1

Nous pouvons à présent passer aux exemples du chapitre 1 que nous avons précédemment installés. En ouvrant ou utilisant à nouveau une console Terminal (utilisons le raccourci Ctrl+L pour effacer le contenu) :

```
cd JavaCpp/EXEMPLES/Chap01
ls
```

nous obtiendrons la liste de nos fichiers :

```
copy_arg.cpp  CopyArgs.java  efface.bat  hello2.cpp  hello3.cpp  hello.cpp  Hello.java
➥Makefile  Makefile.bat  MakefilePremier
```

Nous pouvons maintenant lancer une compilation de tous nos exemples Java et C++, depuis ce répertoire :

```
make
```

Le résultat du `make` est le même que sous Windows :

```
g++ -c hello.cpp
g++ -o hello.exe hello.o
g++ -Wall -c hello2.cpp
Dans le fichier inclus à partir de /usr/include/c++/4.2/backward/iostream.h:31,
          à partir de hello2.cpp:3:
/usr/include/c++/4.2/backward/backward_warning.h:32:2: attention : #warning This file
➥includes at least one deprecated or antiquated header. Please consider using one of
➥the 32 headers found in section 17.4.1.2 of the C++ standard. Examples include
➥substituting the <X> header for the <X.h> header for C++ includes, or <iostream>
➥instead of the deprecated header <iostream.h>. To disable this warning use -Wno-
➥deprecated.
g++ -o hello2.exe hello2.o
g++ -c copy_arg.cpp
g++ -o copy_arg.exe copy_arg.o
javac Hello.java
javac CopyArgs.java
```

Avec notre `ls -lt` (`dir` de Windows), nous pouvons lister les fichiers source et les nouveaux fichiers générés :

```
-rwxrwxrwx 1 jb jb  373 2008-07-22 10:06 MakefilePremier
-rwxrwxrwx 1 jb jb   13 2008-07-22 10:06 Makefile.bat
-rwxrwxrwx 1 jb jb  542 2008-07-22 10:06 Makefile
-rwxrwxrwx 1 jb jb  220 2008-07-22 10:06 Hello.java
-rwxrwxrwx 1 jb jb  216 2008-07-22 10:06 hello.cpp
-rwxrwxrwx 1 jb jb  192 2008-07-22 10:06 hello3.cpp
-rwxrwxrwx 1 jb jb  223 2008-07-22 10:06 hello2.cpp
-rwxrwxrwx 1 jb jb   12 2008-07-22 10:06 efface.bat
-rwxrwxrwx 1 jb jb  301 2008-07-22 10:06 CopyArgs.java
-rwxrwxrwx 1 jb jb  341 2008-07-22 10:06 copy_arg.cpp
-rw-r--r-- 1 jb jb 2876 2008-07-22 17:08 hello.o
-rwxr-xr-x 1 jb jb 9063 2008-07-22 17:08 hello.exe
-rw-r--r-- 1 jb jb 2880 2008-07-22 17:08 hello2.o
-rw-r--r-- 1 jb jb  682 2008-07-22 17:08 Hello.class
-rwxr-xr-x 1 jb jb 9064 2008-07-22 17:08 hello2.exe
-rw-r--r-- 1 jb jb 3184 2008-07-22 17:08 copy_arg.o
-rwxr-xr-x 1 jb jb 9184 2008-07-22 17:08 copy_arg.exe
-rw-r--r-- 1 jb jb  732 2008-07-22 17:08 CopyArgs.class
```

Comme sous Windows, nous pouvons maintenant exécuter notre classe `Hello.class` compilée :

```
jb@jb-desktop:~/JavaCpp/EXEMPLES/Chap01$ java Hello
Hello world en Java: Tue Jul 22 17:09:22 CEST 2008
```

Le nom de notre exécutable est `hello.exe` (nous allons y revenir), lequel doit être précédé des caractères `./` (chemin d'accès restreint sous Linux) :

```
./hello.exe
Hello world en C++: Tue Jul 22 17:10:21 2008
```

gedit comme éditeur Linux

Nous choisirons le bien connu gedit comme éditeur pour Linux. Toujours dans une fenêtre console et avec la commande :

```
gedit Hello.java hello.cpp Makefile &
```

nous allons charger ces trois fichiers dans `gedit`. Le caractère `&` lance gedit en arrière-plan. En l'absence de ce caractère, la commande resterait en attente dans la console et il faudrait quitter gedit ou exécuter un Ctrl+C dans cette console.

Les trois fichiers sont accessibles via des onglets.

Nous commencerons par enlever les extensions `.exe` pour les binaires dans le `Makefile`. C'est une convention DOS/Windows qui n'est pas nécessaire sous Linux. Ainsi, le fichier `hello.exe`, par exemple, devient `hello` (figure F-2) :

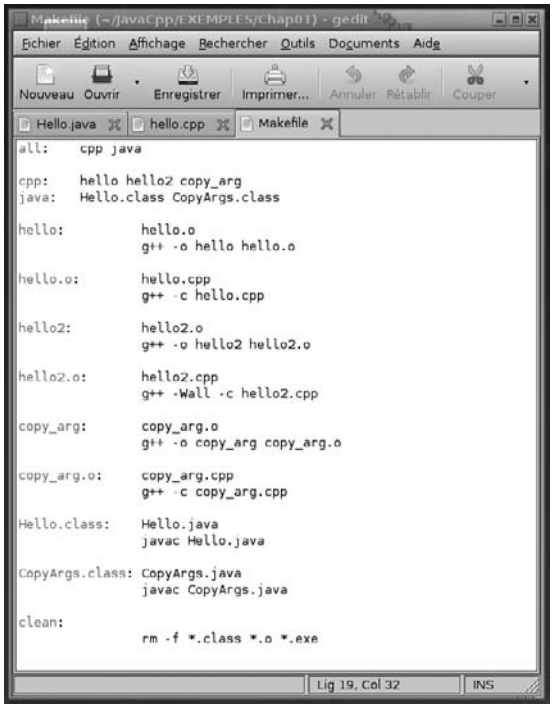

Figure F-2

gedit avec le Makefile du chapitre 1

Comme second exercice, nous allons ajouter dans le fichier `hello.cpp` un `Linux` devant notre `Hello world en C++`, avant de compiler le fichier à nouveau (figure F-3) :

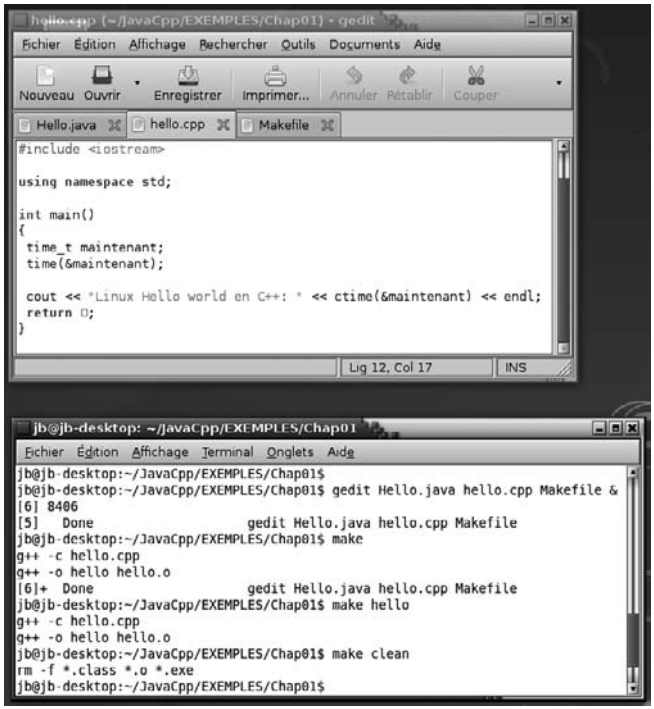

Figure F-3

Environnement de travail sous Linux pour Java et C++

Nous pouvons garder ces deux fenêtres ouvertes et travailler de cette manière. C'est un peu moins souple qu'avec Crimson sous Windows, ou encore NetBeans (voir annexe E), mais c'est tout à fait possible pour nos exemples et exercices.

L'entrée du `clean` (nettoyage) dans le `make` devra aussi être corrigée en remplaçant le `*.exe` par tous les binaires générés par le `g++`.

NetBeans sous Linux

L'interface et l'utilisation sont identiques à celles de Windows :

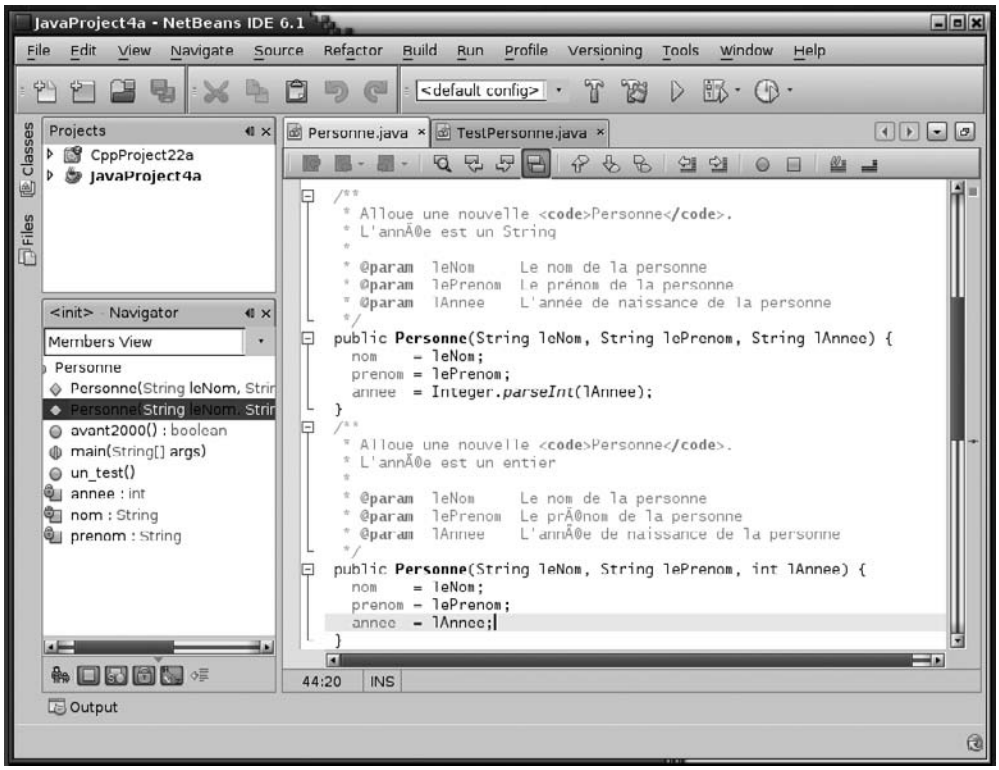

Figure F-4

NetBeans sous Linux avec nos deux projets de l'annexe E

Nous devrons recréer les deux projets comme nous l'avons fait sous Windows dans l'annexe E. Il n'est pas conseillé de travailler sur un même projet NetBeans à la fois sous Windows et sous Linux car c'est trop compliqué. L'auteur n'a, de plus, jamais essayé d'exporter un projet NetBeans de Linux à Windows ou vice versa. En revanche, nous pouvons toujours recopier un ensemble de fichiers source Java ou C++ dans le répertoire des sources de NetBeans, et ceci, lorsque le projet a été créé et préparé.

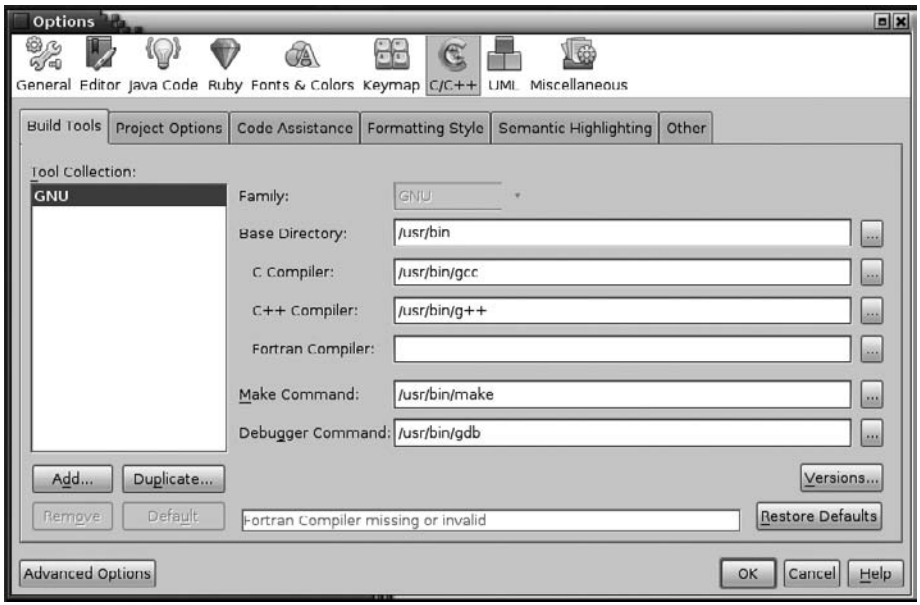

Figure F-5

Environnement Linux pour le C++

Sous Linux, l'auteur a installé l'environnement Java et NetBeans en premier afin de développer uniquement des applications Java. C'est seulement plus tard que les outils de développement C++ ont été installés (voir la section précédente « Installation des outils » avec : `sudo apt-get install build-essential`). Nous avons alors procédé à la configuration manuelle en spécifiant les différents compilateurs et outils (le `make` et le débogueur). Selon les distributions de Linux et les configurations, certaines de ces entrées peuvent déjà exister.

L'annexe E nous a présenté quelques pistes pour débuter rapidement sous l'environnement NetBeans avec Java et C++, ainsi que la méthode à suivre pour réutiliser le code des exemples et des exercices de cet ouvrage dans des projets NetBeans.

G

Dans la littérature et sur le Web

Dans la littérature

Les ouvrages qui couvrent les langages de programmation comme Java et C++ sont innombrables. Juger leurs qualités ou leurs défauts serait de toute manière subjectif. Nous pourrions bien évidemment conseiller de tous les acheter, ce qui ne semble la meilleure solution ni pour les nouveaux venus dans ce domaine, ni pour le porte-monnaie. Un ouvrage va plaire à une personne, alors que son voisin va le détester ! Finalement, la meilleure méthode est sans doute d'examiner soi-même les ouvrages et de faire alors son choix.

Pour visualiser l'ensemble des ouvrages traitant de ces langages sur le site Web des éditions Eyrolles (`http://www.eyrolles.com/`), il suffit d'entrer dans le champ de recherche l'un des mots-clés suivants : `Java`, `C++` ou `C#`.

Avant d'acheter, veillons à toujours consulter le début de l'ouvrage : c'est peut-être une édition périmée. Cette remarque est surtout valable en Java où le langage a beaucoup évolué ces dernières années. Il faudra alors s'assurer que la version 1.5 est traitée. La date de l'ouvrage est aussi une référence. Il faudra évidemment vérifier si les références ISBN sont correctes.

Tableau G-1 C++, Java et C# dans la littérature

Titre de l'ouvrage	Caractéristiques
Le langage C++	Bjarne Stroustrup Campus Press ISBN-13 : 978-2744070037 Tout programmeur C++ devrait posséder cet ouvrage essentiel et traduit en français. La référence !
C++ Primer, 4th Edition	Stanley B. Lippman, Josée Lajoie et Barbara E. Moo Addison-Wesley Professional ISBN-13 : 978-0201721485 Très bon ouvrage qui pour nous représente un must ; l'un de nos préférés.
Thinking in C++	Bruce Eckel Prentice Hall ISBN-13 : 978-0139798092 La deuxième édition, nettement plus à jour, est en deux volumes et disponible électroniquement sur le site : `http://www.mindview.net/Books/TICPP/ThinkingInCPP2e.html`.
C++: An Introduction to Data Structures	Larry Nyhoff Prentice Hall ISBN-13 : 978-0023887253 Une excellente présentation. Les conteneurs du STL y sont présentés. Destiné aux programmeurs avancés.
Au cœur de JAVA, 8e édition (volumes 1 et 2)	Cay Horstmann et Gary Cornell Campus Press ISBN-13 : 978-2744073120 Ouvrage essentiel pour les débutants et traduit en français pour le volume 1.
Thinking in Java, 4e édition	Bruce Eckel Prentice Hall PTR ISBN-13 : 978-0131872486 Vraisemblablement le meilleur ouvrage sur Java pour les programmeurs C++. Référence du site Web : `http://www.mindviewinc.com/Index.php`.
Java in a Nutshell 5th Edition	David Flanagan O'Reilly Media ISBN-13 : 978-0596007737 Plutôt un manuel de référence. A été l'un des premiers ouvrages sur le marché à ce sujet.
Visual C++ 6	Ivor Horton Eyrolles ISBN-13 : 978-2212090437 Une autre approche du C++ avec une très bonne première partie indépendante de l'outil de Microsoft.
Effective C++, 3rd Edition	Scott Meyers Addison-Wesley Professional ISBN-13 : 978-0321334879 Cinquante sujets pour améliorer son code et son design en C++.

Tableau G-1 C++, Java et C# dans la littérature *(suite)*

Titre de l'ouvrage	Caractéristiques
Client/Server Programming With Java and Corba, 2nd Edition	Robert Orfali et Dan Harkey John Wiley & Sons ISBN-13 : 978-0471245780 Ouvrage essentiel pour Corba, mais aussi pour JDBC.
Design Patterns : Elements of Reusable Object-Oriented Software	Erich Gamma, Richard Helm et Ralph Johnson Addison-Wesley Professional ISBN-13 : 978-0201633610 La référence en matière de Design Patterns.
Sams Teach Yourself Java 2 in 21 Days	Laura Lemay et Rogers Cadenhead Sams ISBN-13 : 978-0672329432
MySQL & mSQL	Randy Jay Yarger, George Reese et Tim King O'Reilly ISBN-13 : 978-1565924345 SQL, C-API, C++, Java JDBC, Perl, PHP et Python.
Perl Black Book	Steven Holzer CoriolisOpen Press ISBN-13 : 978-1576104651 Un très bon ouvrage pour programmeurs Perl débutants ou avancés. Contient beaucoup d'exemples et de solutions. Plusieurs chapitres sont consacrés à Internet et aux scripts CGI.
Professional C# 2005 with .NET 3.0	C. Nagel, Bill Evjen, Jay Glynn, Karli Watson et Morgan Skinner Wrox Press ISBN-13 : 978-0470124727 Un ouvrage solide avec beaucoup d'informations sur l'architecture .NET de Windows.
C# 3.0 in a Nutshell	Joseph Albahari et Ben Albahari O'Reilly Media ISBN-13 : 978-0596527570 Parmi six autres ouvrages du même éditeur, avec références aux classes de la bibliothèque Framework de .NET.

Sur le Web

Donner une liste complète, précise et surtout correcte de sites Web qui peuvent nous aider dans notre travail est simplement impossible. Les sites changent souvent ou sont parfois réorganisés. Occasionnellement, nous avons de la chance et nous tombons sur un site exceptionnel. C'est le cas du site de Roedy Green en Java, que nous présenterons plus loin. Il nous offre des centaines de liens URL.

Le projet GNU

Une partie du logiciel disponible sur le CD-Rom est distribuée en tant que logiciel GNU. Pour en comprendre le concept, il est intéressant de visiter le site suivant :

```
http://www.gnu.org/home.fr.html
```

à partir de :

```
http://www.gnu.org
```

C'est est une page Web en français qui présente le projet GNU et l'organisme nommé Free Software Foundation. Ce site contient également des articles en anglais sur la philosophie, l'histoire et la licence GNU.

Le site Web suivant nous présente une série de licences qui font partie du projet GNU :

```
http://www.gnu.org/philosophy/license-list.html
```

Les compilateurs GNU pour C et C++

Les outils et compilateurs GNU que nous avons choisis font partie de MinGW, qui peut être téléchargé depuis le site :

```
http://www.mingw.org/
```

Plus de détails et de références sont donnés dans l'annexe B.

Les newsgroups

C'est une source importante d'informations et de contacts. Les newsgroups existaient avant le Web. C'est un autre point de départ :

```
http://groups.google.com/groups?group=comp&hl=fr
```

GNU EMACS

EMACS était présenté dans les anciennes éditions de ce livre. C'est toujours une alternative, principalement sous Linux. Les sites suivants sont de bons points de départ :

```
http://www.gnu.org/software/emacs/windows/ntemacs.html
http://www.gnu.org/software/emacs/
```

Il ne faut pas oublier le JDE :

```
http://tecfa.unige.ch/guides/emacs/jde/jde.htm
```

C'est un module très pratique qui fonctionne aussi bien sous Windows que sous Linux.

C++

Voici une des excellentes références pour la documentation de la bibliothèque Standard C++ :

```
http://cs.stmarys.ca/~porter/csc/ref/cpp_standlib.html
```

Java

Pourquoi ne pas commencer par le fantastique Java Glossary de Roedy Green :

`http://mindprod.com/jgloss/jgloss.html`

Il pourrait être la source de toutes les recherches sur le Web dans le domaine Java.

Sans oublier évidemment l'incontournable :

`http://java.sun.com/`

Sur le site :

`http://developer.java.sun.com/developer/`

pour certains sujets, il faudra s'enregistrer. Pour les experts, c'est un must. Nous y trouverons des sites consacrés à des domaines particuliers et essentiels comme les « Web services » :

`http://java.sun.com/webservices/`

Il y a beaucoup d'autres sites Java, mais :

`http://www.alphaworks.ibm.com/`

est un site à visiter absolument pour suivre le développement des nouvelles technologies, en particulier pour Java. Beaucoup de code et d'exemples sont disponibles. Des outils comme Eclipse (`http://www.eclipse.org/`), alternative à NetBeans, y sont présentés.

C#

Comme toute première référence, nous indiquons ce site Web consacré à C# :

`http://www.hitmill.com/programming/dotNET/csharp.html`

Il possède de nombreux liens indirects. Mentionnons aussi un site français :

`http://www.csharpfr.com/`

Perl et Python

Rappelons-nous la fonction Perl `splice()` du chapitre 6. Python est un langage similaire à Perl et tout aussi intéressant. Sur ces deux sites Web :

```
http://www.perl.org/
http://www.python.org/
```

nous pouvons trouver non seulement la description de ces deux langages, mais aussi le binaire téléchargeable pour différentes plates-formes. Les outils ou modules associés sont riches et utilisables pour un grand nombre d'applications.

Le Web lui-même

`http://www.w3.org/`

Ce site est celui du World Wide Web Consortium. Il nous offre par exemple la définition du protocole HTTP ou des langages HTML ou XML, des feuilles de style CSS, mais aussi

des liens sur d'autres forums comme le WAP (*Wireless Application Protocol*). Lorsque de nouvelles technologies Internet prendront forme, elles apparaîtront évidemment sur ce site.

Autres recherches d'informations

Il existe un grand nombre de moteurs de recherche. Ils représentent un point de départ essentiel pour retrouver des produits ou des articles relatifs à des domaines particuliers. Le programmeur Java, C ou C++ rencontre souvent un problème très spécifique, et pourra évidemment se tourner en premier lieu vers :

```
http://www.google.com
```

Il suffit souvent de donner deux ou trois mots-clés pour faire notre bonheur :

- Java FileInputStream readline ;
- C# Java comparing.

Si nous voulons sur ce dernier sujet des informations en français, nous pouvons alors spécifier : C# Java comparaison.

Nous y découvrirons un certain nombre d'articles sur les différences entre Java et C#, mais aussi C++.

Rechercher des sujets de travaux pratiques

Par travaux pratiques, nous entendons par exemple un travail de fin de cycle pour un étudiant en informatique. Cependant, cela concerne aussi les débutants, les enseignants ou encore plus simplement les programmeurs professionnels qui tiennent à se recycler et à garder un contact continu avec les nouvelles technologies. Le but étant de rechercher des idées de sujets de travaux informatiques, consulter cette ressource fantastique qu'est le Web peut se révéler très intéressant. De plus, un exercice substantiel dans un domaine particulier est souvent plus motivant et offre une opportunité pour écrire une grande quantité de code. Cela permet d'assimiler beaucoup plus rapidement un langage de programmation ou d'améliorer ses connaissances.

Cet ouvrage contient déjà un aperçu sur une foule de sujets qui pourraient être élargis et étudiés d'une manière beaucoup plus approfondie. Le dernier chapitre de cet ouvrage en est déjà un aperçu. Nous pouvons également consulter les sites Java déjà présentés ci-dessous :

```
http://developers.sun.com/
http://developer.java.sun.com/developer/
http://developers.sun.com/downloads/
```

Nous y trouverons un grand nombre d'API ou autres interfaces, comme :

- XML Parsing ;
- Java Speech ;

- Javamail ;
- Ajax ;
- Database (MySQL, Apache Derby).

Ce sont assurément de très bons sujets pour des travaux pratiques en informatique.

Avec NetBeans (voir annexe E), nous pourrons aussi développer directement des applications avec ces technologies. De nombreux modules de développement y sont intégrés.

Voici encore quelques exemples :

```
http://www.netbeans.org/kb/trails/mobility.html
https://blueprints.dev.java.net/ajax-faq-fr.html
http://www.java-tips.org/java-tutorials/tutorials/introduction-to-java-servlets-
with-netbeans.html
```

Index

Jean-Bernard Boichat

Ingénieur de développement, Jean-Bernard Boichat est spécialiste en C++, Java, systèmes embarqués, méthodologie et technologies Internet. Il travaille actuellement dans le domaine des systèmes d'information pour les passagers des transports publics.

Faire d'une pierre deux coups

Java et C++ sont deux langages à la syntaxe très proche. Grâce à l'apprentissage de leurs différences, défauts et qualités intrinsèques, vous serez mieux préparé pour concevoir un code plus propre, fondé sur une vision élargie de ces deux langages, de leurs possibilités et de leurs limites.

Comment est structuré ce livre ?

L'ouvrage présente et compare les concepts communs aux langages Java et C++ (déclarations de variables, tableaux...), mais expose également les particularités de chacun. Les chapitres sont organisés selon un niveau de difficulté croissant, avec exercices corrigés à la clé.

À qui s'adresse ce livre ?

- Aux débutants en programmation qui souhaitent apprendre les deux langages les plus demandés dans le monde professionnel
- Aux développeurs maîtrisant l'un des langages et souhaitant s'initier à l'autre
- Aux étudiants en informatique (IUT, 2e cycle, écoles d'ingénieurs)

Au sommaire