Le langage *Java*™

Support de cours DESS et IST3/SETI

Nov 2001 - fév 2002

Thomas LEDUC

Planning des cours (salle 32-42.410) :

- vendredi 30 novembre 2001 de 9h à 12h,
- vendredi 7 décembre 2001 de 9h à 12h,
- vendredi 11 janvier 2002 de 9h à 12h,
- vendredi 18 janvier 2002 de 9h à 12h,
- vendredi 25 janvier 2002 de 9h à 12h,
- mardi 19 février 2002 de 9h à 12h.

Ce cours est disponible aux formats PostScript et PDF sur le site :
`http:/bacchus.lmt.ens-cachan.fr:8080/~leduc/ist/`

L'examen aura lieu le vendredi 22 février 2002 de 14h à 16h en amphi 25B.

Chapitre 1

Introduction au langage

Sommaire

1.1 *Java*™ : simple effet de mode ?

Quelques idées maîtresses :
- "the network is the computer" :
  - changement de perspective : la station est vue comme un périphérique alors que le réseau est le cœur du SI,
  - report de complexité : du poste de "l'usager terminal" vers les serveurs et le réseau. Gestion centralisée et professionnelle,
  - du *fat client* au "thin client" et du *desktop* au *webtop* (navigateur + *JVM*) sur *laptop* ou *desktop*. Le client web vu comme un client universel,
  - accélération : réseau $\times 10$ vs. processeur $\times 2$,
- "write once, run anywhere" : multiplication des "clients" périphériques : électroménager, cellulaires, PDA, NC, du *mini-* au *mainframe*…
- "du stade de concept visionnaire à celui de phénomène de programmation universellement adopté" !

1.2 Bref historique

Année	Événement
1945	V. Bush introduit le concept de l'hypertexte
fév 1982	création de *Sun Microsystems Inc.*, moyenne d'âge 26 ans, 4 employés : Andreas Bechtolsheim, Vinod Khosla, Scott McNealy et Bill Joy
1983	sortie de la Sun-2
1984	Scott McNealy devient président de *Sun Microsystems Inc.*, lancement de NFS, implantation de *Sun* en France
1985	sortie de la Sun-3
1989	lancement de la SPARCstation 1, "Original proposal for a global hypertext project at CERN", par Tim Berners-Lee, `http://www.w3.org/History/1989/proposal.html`
oct 1990	premier client Web développé avec l'environnement de développement du NeXTStep par TBL
déc 1990	P. Naughton, M. Sheridan et J. Gosling lancent le "Green Project" pour préparer la *next wave of computing* et contrer le succès du PC avec un ordinateur simple permettant de piloter l'électroménager de gens "normaux"
aoû 1991	Gosling produit le langage **OAK**, Naughton développe une interface graphique et quelques animations
été 1992	présentation de la mascotte Duke à Scott McNealy dans une démo cartoonesque. La *Green Team* totalise 13 membres
1993	le client Web du NCSA, *Mosaic*, est lancé. La *Green Team* devient société *First Person* et est déclarée économiquement non viable. Elle doit se recentrer sur l'Internet grand-public sous la direction de Bill Joy
1994	le navigateur WebRunner (futur HotJava) est présenté comme prototype. Mise en ligne de *Sun Microsystems Inc.* sur le Web (sun.com)
mar 1995	7 ou 8 copies binaires de *Java*™ v 1.0a, mise en ligne de la 1re version publique sur le Web (1.0a2)
23 mai 1995	J. Gage (*Sun Microsystems Inc.*) et M. Andreessen (*Netscape Communications Corporation*) annoncent à la "SunWorld™ audience" que la technologie *Java*™ va être incorporée au Netscape Navigator™
1996	*JDK* 1.0.1, lancement de l'UltraSPARC, RMI, *JDBC*™, JavaBeans
1997	lancement du Sun Enterprise™ 10000, *JDK* 1.1, . . . , *JDK* 1.1.7
1998	*JDK* 1.2, technologies *Java*™ 2 et *Jini*™, la "Sun Java Software Division" compte 800 employés
1999	acquisition de StarOffice, Forté. . . lancement de Sun Ray 1 client léger. Technologie iPlanet, solution e-commerce en alliance avec Netscape.
2001	*Sun Microsystems Inc.* compte 38 000 employés répartis sur 170 pays et réalisant un CA de 19.2 milliards de $. 2,5 million de programmeurs *Java*™ dans le monde. Les actions de Sun sont cotées sur le "National Market System" sous le symbole SUNW. Technologie "Sun Open Net Environment" (Sun ONE).

1.3 Quelques références documentaires

- les exemples du livre *Le langage Java*™ *- programmer par l'exemple* :
  `http://thomas.leduc.free.fr/`
- *Thinking in Java, 2nd Ed.* présentation du langage en plus de 1000 pages :
  `http://www.EckelObjects.com/`
- les exemples du livre *Le langage Java*™ *: concepts et pratique* de I. Charon :
  `http://www.inf.enst.fr/~charon/coursJava/`
- *Java*™ Standard Edition Platform Documentation :
  `http://java.sun.com/docs/`
- rubrique "Documentation and Training" du coin des développeurs *Java*™, *Sun Microsystems Inc.* :
  `http://developer.java.sun.com/developer/infodocs/`
- *The Java*™ *Tutorial* de *Sun Microsystems Inc.* :
  `http://java.sun.com/docs/books/tutorial/`
- *The Java*™ *Language Specification*, 2e édition, de *Sun Microsystems Inc.* :

`http://java.sun.com/docs/books/jls/`
– *The Java™ Virtual Machine Specification*, 2e édition, de *Sun Microsystems Inc.* :
`http://java.sun.com/docs/books/vmspec/`
– la "*Java*™ software FAQ index" :
`http://java.sun.com/docs/faqindex.html`
– la "*Java*™ Infrequently Answered Questions" :
`http://www.norvig.com/java-iaq.html`

1.4 Quelques références d'outils

– le site officiel : `http://www.javasoft.com/`
  – version 1.4.0 b3 (38 Mo + 30 Mo de doc - versions RC pour 2002) pour Solaris, Windows et Linux : `http://java.sun.com/j2se/1.4/`,
  – version 1.3.1 pour Solaris, Windows et Linux : `http://java.sun.com/j2se/1.3/`,
  – version 1.2.2 pour Solaris, Windows et Linux : `http://java.sun.com/products/jdk/1.2/`,
  – version 1.1.8 pour Solaris et Windows : `http://java.sun.com/products/jdk/1.1/`,
– *Java*™-Linux : `http://www.blackdown.org/`

1.5 Langages concurrents

– Internet C++ (et sa machine virtuelle ICVM, Open Source) : `http://www.xmission.com/~icvm/`
– le C# (Microsoft C-Sharp) : `http://www.csharpindex.com/`

```
1   using System;
2   class Welcome
3   {
4     static void Main() {
5       Console.WriteLine("Welcome to csharpindex.com");
6       }
7     }
```

comparatif `http://www.25hoursaday.com/CsharpVsJava.html`
– C++, Objective-C, Smalltalk, Eiffel, perl,

1.6 Principales caractéristiques du langage

– Java : A simple, object-oriented, network-savy, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded, dynamic language.
– plus d'informations techniques dans "The Java Language Environment White Paper", de J. Gosling et H. McGilton, mai 1996, disponible à l'adresse suivante : `http://java.sun.com/docs/white/langenv/`.
– *Java*™ c'est : un langage, une machine virtuelle, un ensemble de classes standards (réparties dans plusieurs *API*), des outils (*JDB*, *javadoc*)…
– Simple : reprise de la syntaxe du C/C++, sans les pointeurs, ni l'héritage multiple de classes, mais avec un ramasse-miettes,
– Orienté-objet : tout est classe dérivant de *java.lang.Object*, à l'exception de quelques types primitifs,
– Interprété : au niveau d'une *JVM*, après production de *byte code*,
– Robuste : mécanisme d'exceptions, langage fortement typé, ramasse-miettes, pas d'accès direct à la mémoire par pointeur,
– Sécurisé : pas d'accès direct aux ressources physiques, compilateur "exigeant", *JVM Verifier* (vérificateur de *byte code*), *Class Loader* et *Security Manager* (système de fichiers, accès réseau),
– Indépendant du matériel et portable : la *JVM* s'intercale entre le *hardware* et les applications/applets,
– Performant : un peu plus à chaque nouvelle version de *JVM*,
– Distribué : *API* réseau standard conséquente, mécanismes de RMI, JavaIDL (pour intégration et adéquation à CORBA), JNDI (*Java*™ Naming and Directory Interface, pour les services d'annuaires),
– Multithreads : nativement et de façon simple, géré directement par la *JVM*,
– Dynamique : chargement des classes en cours d'exécution, à la demande, pas de phase d'édition de liens,

1.7 Ne pas confondre…

– *Java*™ et *JavaScript*™ :

Java™	*JavaScript*™
Sun Microsystems Inc.	*Netscape Communications Corporation*
précompilé puis interprété	interprété dans le contexte d'un navigateur
autonome	encapsulé dans un document HTML
fortement typé	faiblement typé
assimilable à un exécutable traditionnel	assimilable à un "script batch" traditionnel

– *Java*™ et C++ :
  – *Java*™ reprend la syntaxe du C++ pour faciliter le recyclage de compétences et le transfert technologique,
  – *Java*™ supprime les *struct*, *union*, *enum*, *typedef*, le *préprocesseur*, les primitives à nombre d'arguments variables, l'héritage multiple, les types paramétriques (*templates*), la surcharge d'opérateurs, les pointeurs, le passage d'objets par copie, les variables et fonctions hors classe…

1.8 La programmation orientée objets en bref

– à l'origine : programmation à grande échelle ⇒ compilation séparée, protection des détails d'implémentation, réutilisation du code,
– un constat : il est souvent préférable de structurer un système par rapport à ses données plutôt que par rapport à ses fonctions. Celles-ci semblent en effet plus stables et pérennes que les traitements qu'on leur applique qui sont plus sujets aux évolutions,
– approche objet : structuration du code en terme de classes d'objets (auxquelles on associe des variables d'état et des traitements bien déterminés) et en terme d'instances de classes ou objets les instanciant, interagissant par des échanges de messages,
– une classe = un aspect statique (caractérisé par ses variables d'état, appelées dans ce contexte, attribut ou variables d'instance) + un aspect dynamique (représenté par l'ensemble des opérations que peut réaliser un objet en réponse à une sollicitation, un message… Ces opérations sont appelées méthodes dans le contexte objet, elles sont à l'objet ce que les procédures (ou *subroutines*) sont à la programmation structurée classique.),
– les grands concepts :
  – principe d'abstraction ou encapsulation,
  – principe d'envoi de message,
  – principe d'héritage,
  – polymorphisme et liaison dynamique.

1.8.1 Principe d'abstraction ou encapsulation

Protéger les détails de l'implémentation en évitant qu'une modification bénigne sur un traitement ou une donnée de base ait des répercussions sur tous les programmes appelants. En pratique : on "empaquète" données et traitements au sein d'une même structure, en ne la rendant accessible de l'extérieur que par le biais des opérations (l'interface) laissées visibles à cet usage. Structure logique d'encapsulation : l'objet.

– encapsulation par l'objet : objet = un triplet : un identificateur (appelé aussi *Object Identifier* ou *OID* qui permet de le référencer de façon unique, une variable d'état composite ou non, et un ensemble de lois de comportement. Ces variables d'état et lois de comportement ne sont pas définies directement au niveau de l'objet lui-même mais au niveau de sa classe d'appartenance.
– factorisation par la classe : une classe = un triplet comprenant un mécanisme d'instanciation, des spécifications de propriétés statiques, et un mécanisme de manipulation ou d'accès à ces propriétés.
– niveaux de visibilité : par ordre d'opacité croissante : *public*, *protected*, "*friendly*" (qui est implicite, le mot clef n'apparaît donc pas puisque c'est l'état par défaut) et *private*. La classe peut donc aussi être considérée comme un type abstrait de données composée d'une interface (propriétés statiques et dynamiques "offertes" à l'utilisateur ou à l'environnement extérieur) et d'une implémentation (propriétés statiques et dynamiques masquées pour le monde environnant).

1.8.2 Principe d'envoi de messages

- l'activation d'une propriété dynamique d'un objet (c'est-à-dire l'activation de méthode) s'effectue par le biais d'un envoi de message à l'objet concerné. Cet objet s'appelle aussi objet récepteur. Un message est complètement décrit par la donnée du récepteur, du sélecteur (ou nom de la méthode à activer) et de ses paramètres. On appelle signature d'un message l'ensemble constitué du nom de la méthode à activer et des paramètres du message. On parlera par extension de signature de méthode.
- liaison dynamique : la liaison est le mécanisme qui consiste à sélectionner le code de la méthode à activer lors de la réception d'un message par un objet récepteur. Cette liaison peut être statique, auquel cas elle est réalisée à la compilation du code source, ou dynamique. Dans ce dernier cas, la liaison est réalisée à l'exécution comme c'est le cas en *Java*™.

1.8.3 Principe d'héritage

- la classe = un type abstrait de données que l'on peut spécialiser par le biais de l'héritage de classes mais aussi un moyen d'instancier (de créer) des objets. Ces deux mécanismes de hiérarchisation permettent de construire des graphes d'héritage (encore appelés graphes de spécialisation ou de généralisation) d'une part et des graphes d'instanciation d'autre part.
- L'héritage = un mécanisme de transmission des propriétés statiques et dynamiques d'une classe à l'autre par filiation. C'est aussi ce que l'on appelle de la spécialisation, puisque l'héritage permet de propager des spécifications d'une classe "générale" vers des sous classes "particulières" en offrant la possibilité de les enrichir successivement. En pratique l'héritage consiste à factoriser des propriétés statiques et dynamiques au sein d'une classe mère, de façon à ce qu'un ensemble de classes filles puissent en hériter (et les réutiliser) par simple transmission sans qu'il y ait nécessité de dupliquer le code concerné.
- surcharge de méthode : alors que la redéfinition de méthode consiste à réécrire l'implémentation d'une méthode héritée sans modifier sa signature (seul l'objet récepteur diffère), la surcharge implique une double redéfinition de la signature et du code d'une méthode héritée

1.8.4 Polymorphisme et liaison dynamique

Faculté d'une méthode donnée à pouvoir être appliquée à divers objets instanciant chacun des classes différentes. C'est le cas lorsqu'une sous classe hérite une propriété dynamique publique d'une classe mère sans la surcharger. Ce mécanisme est à associer étroitement avec la liaison dynamique. En effet, la liaison dynamique signifie que lors de l'exécution, le code de la méthode invoquée par envoi de message est recherché par parcours ascendant de la hiérarchie de classe (du graphe d'héritage) de l'objet récepteur. La première méthode trouvée dans ce parcours "ascendant" de graphe, dont la signature correspond à celle du message, est exécutée.

1.9 Les outils

- *JDK* 1.4 (béta) : 135 paquetages, 2757 classes, plus de ? ? 000 méthodes et variables d'instances,
- *JDK* 1.3 : 76 paquetages, 1839 classes, plus de 25 000 méthodes et variables d'instances,
- outils de base du *JDK* : javac (compilateur de source), java (interpréteur de *byte code*), javadoc (générateur de documentation), appletviewer (lecteur de page HTML), jar (*Java*™ archiveur - *tar*), jdb (débogueur), javah (générateur de *header file* dans le cadre de JNI), javap (désassembleur de *byte code*),
- outils RMI : rmic, rmiregistry, rmid, serialver,
- outils sécurité : keytool, jarsigner, policytool,

► **L'indispensable :** un éditeur, un compilateur (générateur de bytecode) et une machine virtuelle.

► **L'optionnel :** l'IDE (Integrated Development Environment) ou le RAD (Rapid Application Development) ⇔ éditeur guidé par la syntaxe, gestionnaire de version/projet, débogueur, générateur d'interface… Le site de référence (120 IDE répertoriés à la date du 21/11/2001) :

`http://www.javaworld.com/javaworld/tools/jw-tools-ide.html`

Quelques noms :
- CodeWarrior for Java de Metrowerks,

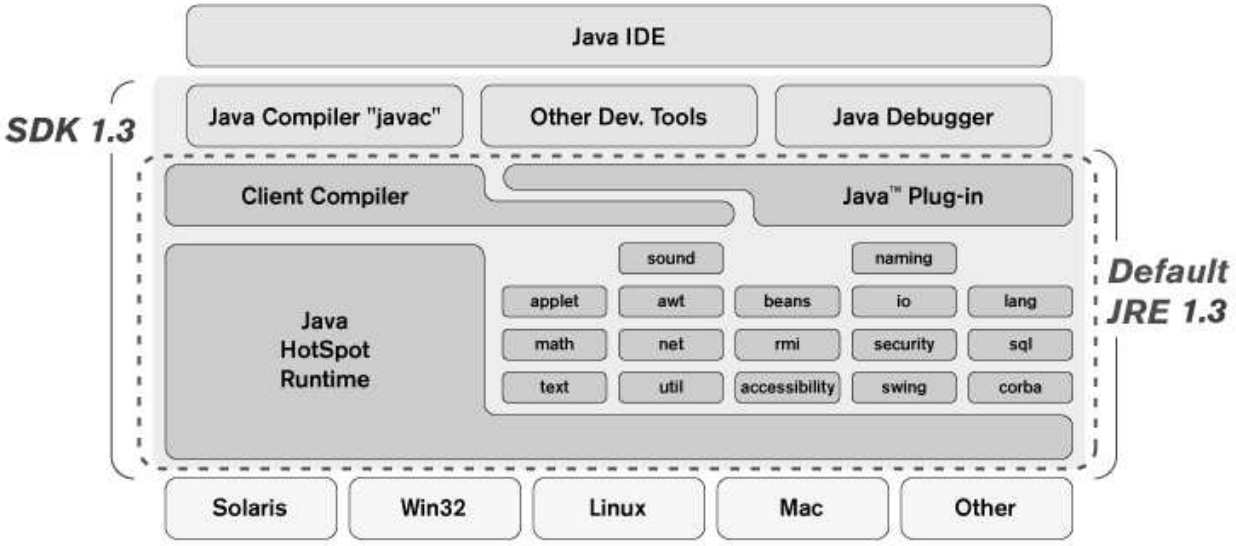


FIG. 1.1 – La plate-forme *Java*™

- Cosmo Code de SGI,
- Forte for *Java*™ de *Sun Microsystems Inc.* (`http://www.sun.com/forte/ffj/`),
- Java Development Environment pour Emacs,
- JBuilder d'Inprise/Borland,
- Oracle JDeveloper d'Oracle Corporation,
- PowerJ de Sybase,
- Visual J++ de Microsoft,
- VisualAge d'IBM,

1.10 Premier programme

```
1  import java.lang.*;
2
3  class helloWorldApp extends java.lang.Object
4  {
5      public static void main(java.lang.String [] arguments)
6      {
7          java.lang.System.out.println("Hello World !");
8      }
9  }
```

Version apurée :

```
1  class helloWorldApp
2  {
3      public static void main(String [] arguments)
4      {
5          System.out.println("Hello World !");
6      }
7  }
```

```
# javac helloWorldApp.java
# java helloWorldApp
# Hello World !
```

- paquetage *java.lang* importé implicitement,
- signature de méthode : nom de la méthode et ensemble des types de ses paramètres (en *Java*™, le type de la valeur de retour n'en fait pas partie),
- programmation statique vs. programmation dynamique,

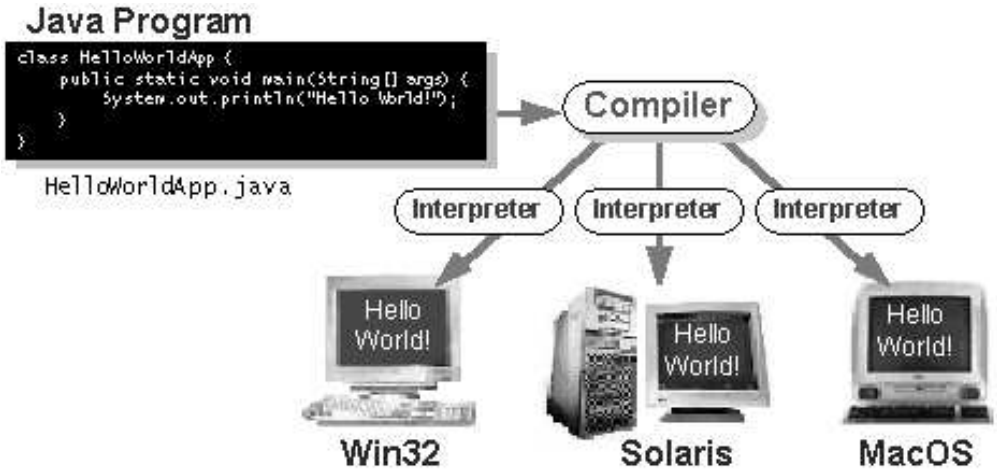


FIG. 1.2 – *Java*™ *byte code* "write once, run anywhere"

1.11 Organisation sommaire du développement en *Java*™

1.11.1 Structuration

En bref :
- une application *Java*™ $\Leftrightarrow\sum$ paquetages
- un paquetage *Java*™ $\Leftrightarrow\sum$ fichiers *.java*
- un fichier *.java* $\Leftrightarrow\sum$ de classes dont une seule est *public*. Le nom du fichier est celui de la classe *public* à laquelle on ajoute l'extension *.java*

```
# cat A.java
public class A {
}
public class B {
}

# javac A.java
A.java:3: class B is public, should be declared in a file named B.java
public class B {
       ^
1 error

# cat A.java
class A {
}
public class B {
}

# javac A.java
A.java:3: class B is public, should be declared in a file named B.java
public class B {
       ^
1 error

# cat A.java
public class A {
}
class B {
```

```
}

# javac A.java
#
```

► **Note :** le paquetage permet de regrouper un ensemble de classes (ou interfaces) très liées au sein d'une "unité de compilation" pour mieux les organiser/gérer.

```
1  package cours1;
2
3  class Class1 {
4      public static void main(String [] arguments)
5      {
6          System.out. println ((new Class1 ()). getClass (). getName());
7      }
8  }
```

```
1  package cours1;
2
3  class Class2 {
4      public static void main(String [] arguments)
5      {
6          System.out. println ((new Class2 ()). getClass (). getName());
7      }
8  }
```

```
# pwd
.../cours1
# javac Class1.java Class2.java
# java -classpath .. cours1.Class2
cours1.Class2
```

1.11.2 Documentation automatique - Utilisation de *javadoc*

Cet outil est fourni par défaut avec le *JDK*. Il vous offre la possibilité de générer automatiquement une documentation structurée et hiérarchisée de votre code source (ou ensemble de codes sources) *Java*™. Un simple appel à cette commande accompagnée du nom d'une classe ou du nom d'un paquetage suffit à générer dans le répertoire de votre choix ou, à défaut, dans le répertoire courant, un certain nombres de fichiers au format HTML ainsi qu'une feuille de style *stylesheet.css*. Cette documentation regroupe des commentaires, un descriptif de chaque classe, des prototypes de méthodes… La navigation au sein de l'ensemble de cette documentation est facilitée par l'existence de liens hypertextes.

Le langage *Java*™ prévoit des "commentaires de documentation" insérés dans le corps même de votre code source et délimités par `/**` en ouverture et `*/` en fermeture. Ces commentaires ne sont pas traités par le compilateur-générateur de bytecode ; ils sont destinés à l'utilitaire *javadoc* de génération de documentation automatique et transformés à cette occasion en HTML.

Il faut commenter toute classe et méthode publique.

Pour plus d'information, consulter :

– `http://java.sun.com/j2se/javadoc/`
– `http://java.sun.com/j2se/javadoc/faq.html`

@author NomAuteur	cette balise insère une entrée relative au nom de l'auteur de la classe courante. Elle ne peut-être utilisée qu'avant une définition de classe. Pour insérer plusieurs noms d'auteurs pour une classe donnée, il suffit d'insérer autant de balise *@author* que voulu.
@deprecated Commentaire	cette balise permet de signaler que la méthode courante est dépréciée et ne doit plus être utilisée. Pour renvoyer à une méthode plus récente, insérez dans votre commentaire la balise : `{@link #maNouvelleMethode()}`.
@exception NomClasse Description	Cette balise a un effet identique à la balise *@throws* décrite plus précisément ci-après.

{*@link* UneRéférence }	Cette balise à un effet relativement comparable à celui de la balise *@see* présentée ci-après. Mais l'hyperlien qu'elle génère s'insère in-situ dans le texte sans introduire de nouvelle section ("See Also") contrairement à *@see*. L'argument `UneRéférence` peut-être aussi précis que possible : `@see paquetage.Classe#methode(liste de types d'arguments) Etiquette`.
@param NomParamètre Description	Cette balise permet de décrire un paramètre d'une méthode. Elle peut-être multi-lignes.
@return Description	Cette balise décrit le type de retour de la méthode courante.
@see UneRéférence	Cette balise génère un hyperlien permettant de se référencer à une autre partie de la documentation. Ce lien peut-être aussi précis que possible : `@see paquetage.Classe#methode(liste de types d'arguments) Etiquette`.
@serial DescriptionChamp *@serialData* NomChamp TypeChamp DescriptionChamp *@serialField* DescriptionDonnées	
@since Commentaire	Cette balise permet de spécifier que la fonctionnalité existe depuis la version indiquée en commentaire.
@throws NomClasse Description	Les balises @throws et @exception sont synonymes. Cette balise permet de spécifier le nom NomClasse de l'exception qui peut-être levée par la méthode courante.
@version NuméroVersion	Cette balise insère une entrée relative au numéro de version du logiciel contenant cette classe. Elle ne peut-être utilisée qu'avant une définition de classe. La convention consiste à utiliser la chaîne `"%I%, %G%"` (SCCS) ou à défaut `"%I%, %U%"` (incrémentation compteur à chaque modification à partir de 1.1 et date de dernière modification).

1.11.3 Ordre d'appel :

```
* @author       (classes and interfaces only, required)
* @version      (classes and interfaces only, required)
*
* @param        (methods and constructors only)
* @return       (methods only)
* @exception    (@throws is a synonym added in Javadoc 1.2)
* @see
* @since
* @serial       (or @serialField or @serialData)
* @deprecated
```

L'appel à la commande `javadoc -private -author -version -d doc *.java` permet de générer toute la documentation des sources .java du répertoire courant dans le sous répertoire `doc`.

1.11.4 Déploiement avec un *.jar*

Java ARchive

⇔ aggéregation d'arborescences de fichiers de façon à ne former qu'un seul fichier d'archive basé sur le format ZIP,

⇔ utilitaire d'archivage généraliste comparable à la commande *tar*,

⇔ fichier unique simplifiant l'ensemble des transactions multimédias (sons/images/animations/.class) sur le web. Il suffit d'un seul transfert HTTP,

⇔ compression et signature pour authentification,

```
<applet code=Animator.class
      archive="classes.jar, images.jar, sounds.jar"
```

```
      width=460 height=160>
   <param name=foo value="bar">
</applet>
```

Il s'agit ici de fabriquer une archive au format JAR, dans le but de pouvoir l'exécuter par la suite. Considérons l'exemple suivant :

Création des fichiers sources et compilation

```
# mkdir paquetageImpression
# cd paquetageImpression
# <– Génération des fichiers du "paquetageImpression"–>
# javac *.java
```

Listing du fichier `ImpressionGenerique.java` :

```
1  package paquetageImpression;
2
3  public abstract class ImpressionGenerique
4  {
5      public String message = "hello world !";
6      public abstract void afficher();
7
8      public static void main(String[] arguments)
9          throws InstantiationException, IllegalAccessException, ClassNotFoundException
10     {
11         long dateDébut = System.currentTimeMillis(); // mesure du temps
12         ImpressionGenerique tmp = (ImpressionGenerique) Class.forName(arguments[0]).newInstance();
13         tmp.afficher();
14         long dateFin = System.currentTimeMillis(); // mesure du temps
15         System.err.println("Temps d'exécution = " + (dateFin-dateDébut) + " ms");
16     }
17 }
```

Listing du fichier `ImpressionEgal.java` :

```
1  package paquetageImpression;
2
3  public class ImpressionEgal extends ImpressionGenerique
4  {
5      public void afficher()
6      {
7          System.out.println(" = " + message);
8      }
9  }
```

Listing du fichier `ImpressionEtoile.java` :

```
1  package paquetageImpression;
2
3  public class ImpressionEtoile extends ImpressionGenerique
4  {
5      public void afficher()
6      {
7          System.out.println(" * " + message);
8      }
9  }
```

Première méthode de fabrication et exécution d'une archive

```
# cd ..
# jar cf paquetageImpression.jar paquetageImpression
# java -classpath paquetageImpression.jar paquetageImpression.ImpressionGenerique paquetageImpression.ImpressionEtoile
 * hello world !
Temps d'exécution = 3 ms
#
```

Seconde méthode de fabrication et exécution d'une archive (à partir de la version 1.2)

```
# cd paquetageImpression
# echo 'Main-Class : paquetageImpression.ImpressionGenerique' > ./Main-Class
# cd ..
# jar cmf paquetageImpression/Main-Class paquetageImpression.jar paquetageImpression
# java -jar paquetageImpression.jar paquetageImpression.ImpressionEgal
 = hello world !
Temps d'exécution = 4 ms
# du -hs paquetageImpression paquetageImpression.jar
```

```
32k paquetageImpression
4.0k paquetageImpression.jar
```

Ajout de fonctionnalité à partir d'une archive .jar

Etant donné le code source suivant placé dans un répertoire quelconque :

```
1  import paquetageImpression.*;
2
3  public class ImpressionPlus extends ImpressionGenerique
4  {
5      public void affi cher ()
6      {
7          System.out. println ( " + "  + message);
8      }
9  }
```

```
# jar cf paquetageImpression.jar paquetageImpression
# javac -classpath paquetageImpression.jar ImpressionPlus.java
# java -classpath paquetageImpression.jar :. paquetageImpression.ImpressionGenerique ImpressionPlus
 + hello world !
Temps d'exécution = 5 ms
```

Pour plus d'information, consulter : `http://java.sun.com/docs/books/tutorial/jar/`

1.12 Les spécifications du langage et de la *JVM*- syntaxe du langage

Pour plus d'information, consulter :
– `http://java.sun.com/docs/books/jls/`
– `http://java.sun.com/docs/books/vmspec/`

Java™ utilise l'Unicode v 2.1 `http://www.unicode.org` depuis la version 1.1 du langage (la première version utilisée était la 1.1.5, le *JDK* 1.4 est basé sur l'Unicode v 3.0). Un caractère Unicode est représenté par la séquence : `\uXXXX` (où X est un chiffre hexadécimal) en *Java*™.

Attention : la séquence `\u005cu005a` est traduite en `\u005a` et non en `z` (même si `\u005c` et `\u005a` correspondent respectivement aux lettres \ et z).

Découpage en lexèmes de la séquence de caractères suivante : `a--b` donne `a,--,b` (même si cette expression est incorrecte) et non `a,-,-b` (qui est une expression acceptable).

1.12.1 Identificateur

un identificateur ⇔ séquence de caractères illimitée commençant par une "Java letter" et se poursuivant par des caractères Unicode de type "Java letter/digit" ou de code strictement supérieur à `0x00C0` ('À'). Cette séquence ne doit pas correspondre aux littéraux booléens, à *null* ou à l'un des mots-clefs suivants : *abstract*, *default*, *if*, *private*, *throw*, *boolean*, *do*, *implements*, *protected*, *throws*, *break*, *double*, *import*, *public*, *transient*, *byte*, *else*, *instanceof*, *return*, *try*, *case*, *extends*, *int*, *short*, *void*, *catch*, *final*, *interface*, *static*, *volatile*, *char*, *finally*, *long*, *super*, *while*, *class*, *float*, *native*, *switch*, *const*, *for*, *new*, *synchronized*, *continue*, *goto*, *package*, *this*

un "Java letter" est un caractère pour lequel la méthode *Character.isJavaLetter* renvoie *true* et un "Java letter/digit" est un caractère pour letter/digit" *Character.isJavaLetterOrDigit* renvoie *true*. Un "Java letter" est un caractère de l'ensemble A-Z (`\u0041-\u005a`), a-z (`\u0061-\u007a`), et, pour des raisons historiques, le souligné (_, `\u005f`) et le dollar ($, `\u0024`). Un "java digit" est un caractère de l'ensemble 0-9 (`\u0030-\u0039`).

```
identifier ::=
"a..z,$,_" { "a..z,$,_,0..9,unicode character over 00C0" }
```

1.12.2 Littéraux du langage

6 types de littéraux : entier, flottant, booléen, caractère, chaîne de caractères, null.

Littéraux entiers

- décimal ⇔ `0 | [1-9][0-9]*(lL)?`
- octal ⇔ `0 [0-7]*(lL)`
- hexadécimal ⇔ `0(xX) [0-9a-fA-F]*(lL)`

c'est le cas de `0xDadaCafe`, `0777L` ou `2L`.

```
integer_literal
      ::=
      ( ( "1..9" { "0..9" } )
      | { "0..7" }
      | ( "0" "x" "0..9a..f" { "0..9a..f" } ) )
      [ "l" ]
```

Littéraux flottants

- simple précision : `[+-]? [0-9]*[.][0-9]*[eE][+-]?[0-9]*?[fF]`
- double précision : `[+-]? [0-9]*[.][0-9]*[eE][+-]?[0-9]*?[dD]`

```
float_literal
      ::=
      ( decimal_digits "." [ decimal_digits ] [ exponent ] [ float_suffix ] )
      | ( "." decimal_digits [ exponent ] [ float_suffix ] )
      | ( decimal_digits [ exponent ] [ float_suffix ] )

decimal_digits
      ::=
      "0..9" { "0..9" }

exponent
      ::=
      "e" [ "+" | "-" ] decimal_digits

float_suffix
      ::=
      "f" | "d"
```

Littéraux caractères

caractère ou "escape séquence" entre simples quotes

Littéraux chaînes de caractères

caractère ou "escape séquence" entre doubles quotes

1.12.3 Opérateurs et séparateurs

En bref

- un séparateur est un des 9 caractères ASCII (*American Standard Code for Information Interchange* ⇔ code à 7 bits qui est à l'origine de la plupart des encodages actuels. Il correspond aux 128 premiers caractères des tables de codes de la série des normes ISO-8859-X) suivants :
  `( ) { } [ ] ; , .`
- 37 opérateurs parmi :

```
= > < ! ~ ? :
==      <=      >=      !=      &&      ||      ++      --
+ - * / & | ^ % <<      >>      >>>
+=      -=      *=      /=      &=      |=      ^=      %=
<<=     >>=     >>>=
```

Les opérateurs arithmétiques

Opérateur	Type de l'opérande	Signification
$+$	Numérique	Valeur inchangée
$-$	Numérique	Valeur opposée
$++$	Numérique	Post- ou pré-incrémentation
$--$	Numérique	Post- ou pré-décrémentation

► **Les opérateurs arithmétiques unaires**

► **Les opérateurs arithmétiques binaires** *Java*™ introduit ici une nouveauté par rapport aux langages C et C++. En effet, l'opérateur % s'applique aussi, en *Java*™, à des opérandes de type flottant.

Opérateur	Type des opérandes	Signification
$+$	Numérique	Addition
$-$	Numérique	Soustraction
$\times$	Numérique	Multiplication
$/$	Entier	Division entière
	Flottant	Division flottante
%	Entier	Reste dans la division entière
	Flottant	Reste dans la division où l'on tronque le quotient à sa partie entière

► **Attention :** Une division entière par 0 ou le calcul du reste dans la division entière par 0 soulèvent toutes deux une exception de type *java.lang.ArithmeticException*. L'arithmétique entière en *Java*™ ne se comporte en effet pas de façon "non-stop" (elle interrompt donc le cours du programme en générant une erreur) comme l'arithmétique flottante.

► **Remarque :** Il ne faut pas confondre l'opérateur flottant % et la méthode statique *java.lang.Math.IEEEremainder*. En effet, dans le premier cas, le résultat est un flottant de même signe que le dividende[1] de valeur absolue inférieure à celle du diviseur. Ce résultat est obtenu par un traitement similaire à la division euclidienne qui permet d'obtenir le reste dans le cas d'une division entière. Il s'agit du reste dans le cas d'une division où le quotient est tronqué à sa partie entière. Dans le second cas, par contre, le reste correspond à un quotient arrondi à l'entier le plus proche.

Les opérateurs arithmétiques et logiques sur bits

Les opérateurs présentés dans cette section opèrent tous sur des opérandes de type entier. A l'exception de l'opérateur $\sim$ qui est unaire, tous les opérateurs du tableau ci-après sont binaires.

Opérateur	Signification
$\sim$	Passage au complément à 1 (ou négation) bit à bit
&	Conjonction logique (AND) bit à bit
$\vert$	Disjonction logique (OR) bit à bit
^	Disjonction exclusive (XOR) bit à bit
$<<$	Décalage de bits vers la gauche en complétant les bits de droite par des bits nuls
$>>$	Décalage de bits vers la droite en complétant les bits de gauche avec la valeur du bit de poids fort (le signe donc)
$>>>$	Décalage de bits vers la droite en complétant les bits de gauche par des bits nuls

Expression	Type des opérandes	Expression numérique équivalente
$\sim i$	*int* ou *long*	$(-i)-1$
$i << j$	*int*	$i \times 2^{(32+j)\%32}$ si $0 \leq i \leq 2^{(30-(32+j)\%32)}$ ou si $-2^{(31-(32+j)\%32)} \leq i \leq 0$

[1] Qui est l'opérande de gauche, tandis que le diviseur est l'opérande de droite.

$i << j$	*long*	$i \times 2^{(64+j)\%64}$ si $0 \leq i \leq 2^{(60-(64+j)\%64)}$ ou si $-2^{(61-(64+j)\%64)} \leq i \leq 0$
$i >> j$	*int*	$\lfloor i/2^{(32+j)\%32} \rfloor$ si $0 \leq \lvert i \rvert \leq 2^{31}-1$
$i >> j$	*long*	$\lfloor i/2^{(64+j)\%64} \rfloor$ si $0 \leq \lvert i \rvert \leq 2^{63}-1$
$i >>> j$	*int*	$\lfloor i/2^{(32+j)\%32} \rfloor$ si $0 \leq i \leq 2^{31}-1$ $(i >> j+2 <<\sim j)$ si $i \leq 0$
$i >>> j$	*long*	$\lfloor i/2^{(64+j)\%64} \rfloor$ si $0 \leq i \leq 2^{63}-1$ $(i >> j+2L <<\sim j)$ si $i \leq 0$

encadrer un nombre décimal donné entre deux puissances de 2 consécutives.

```
1   class EncadrementEnPuissance2
2   {
3       public static void main ( String [] arguments)
4       {
5           float x = Float . parseFloat (arguments [0]);
6           int seqBits = Float . floatToIntBits (x);
7           int exp = (( seqBits >> 23) & 255) - 127;
8           int signe = seqBits >> 31;
9
10          if ( signe == 0)
11              System.out. println ( "2^" + exp + " <= " + x + " < " + "2^" + (exp+1) );
12          else
13              System.out. println ( "-2^" + (exp+1) + " < " + x + " <= " + "-2^" + exp );
14      }
15  }
```

Les opérateurs logiques sur prédicats

opérandes de types booléens

Opérateur	Arité	Signification
!	Unaire	Négation logique
&	Binaire	Conjonction logique (AND)
|	Binaire	Disjonction logique (OR)
^	Binaire	Disjonction logique exclusive (XOR)
&&	Binaire	Conjonction logique (AND) avec “court-circuit”
||	Binaire	Disjonction logique (OR) avec “court-circuit”

Les opérateurs d’affectation

affectation simple et affectation combinée

expression dont la valeur de retour est la valeur de l’expression située à droite de l’opérateur d’affectation. Tous ces opérateurs d’affectation sont à **effet de bord**, puisqu’ils modifient la valeur de leur opérande de gauche tout en renvoyant une valeur.

Opérateur	Signification
$=$	$i = 3$ permet d’affecter la valeur 3 à la variable i
$+=$ $-=$ $\times=$ $/=$ $\%=$ $<<=$ $>>=$ $>>>=$ $\&=$ $\wedge=$ $\mid=$	$xop = y$ est une façon plus rapide d’écrire $x = xopy$

```
int a, b=5, c= (a=3) * (b*=2);
System.out.println(a + " " + b + " " + c);
```

Cet exemple renvoie 3, 10 et 30 sur la sortie standard lors de l’exécution.

Les opérateurs de comparaison

Opérateur	Type des opérandes	Signification
==	Primitif	Égal
!=	Primitif	Différent
<	Numérique	Inférieur à
>	Numérique	Supérieur à
<=	Numérique	Inférieur ou égal à
>=	Numérique	Supérieur ou égal à

L'opérateur == de test d'égalité est **associatif** de la gauche vers la droite. L'expression $a == b == c$ doit se lire sous la forme $(a == b) == c$ et le résultat renvoyé par un test d'égalité est un booléen.

Le cas particulier de l'opérateur conditionnel d'arité 3

L'opérateur conditionnel retourne une valeur correspondant soit à la valeur de son second, soit à la valeur de son troisième opérande. Les types de ces opérandes doivent être "compatibles" du point de vue de l'affectation puisque la valeur renvoyée par cet opérateur est utilisée directement dans une instruction.

```
condition ? expression-si-condition-vraie : expression-si-condition-fausse;
```

Comme nous l'avons écrit ci-avant, la ligne de code suivante génère immanquablement une erreur de type *Incompatible type for ? :* à la compilation :

```
System.out.println( false ? 17 : "il fait beau" );
```

puisque le compilateur ne force pas la conversion automatique du nombre 17 en un objet de type *java.lang.String*.

L'opérateur de coercition () : conversion explicite et conversion implicite

Typage fort
- conversion à l'identique, conversion vers un type plus étendu (peut provoquer des erreurs : `System.out.println((123456789-(int)(float) 123456789));` renvoie -3 car l'entier 123456789 est traduit sous forme flottante en simple précision par $1.23456792E8$),
- conversion vers un type moins étendu (`((int) 123.7f == 123`, `(float) 1e39 == Infinity`, `(float) 1e-46 == 0.0`, `(byte) 128 == -128`,
- conversion implicite ($\Leftrightarrow$ conversion vers un type plus étendu), conversion explicite : à l'aide de l'opérateur de coercition *(type)*, en utilisant les séparateurs "(" et ")"

La concaténation de chaînes de caractères

```
String s = "le Pont du Gard mesure ";
s += 48.77f;
s += " mètres de hauteur !";
```

L'opérateur *new* d'instanciation d'objet

```
Object unObjet = new Object();
System.out.println( new Object().getClass() );
```

L'opérateur *instanceof* de test d'instanciation

```
1  Integer unEntier = new Integer(13);
2  System.out.println( unEntier instanceof Integer );
3  System.out.println( unEntier instanceof Number );
4  System.out.println( unEntier instanceof Object );
5  System.out.println( unEntier instanceof Comparable );
```

1.12.4 Ordre d'évaluation des opérateurs

les expressions sont lues et évaluées de la gauche vers la droite en *Java*™, dans le cas d'un opérateur binaire, l'opérande de gauche est évalué avant l'opérande de droite.

```
int i=10, j=(i=1) * i;
System.out.println( "j = " + j );  // produit j=1
int k=10; k+=(k=1); // produit 11
```

1.12.5 La précédence des opérateurs et des séparateurs

Considérons l'expression non parenthésée $2+3\times4$. Il y a, a priori, deux façons différentes de l'évaluer : $2+(3\times4)=14$ ou $(2+3)\times4=20$. *Java*™, comme ses prédécesseurs, a décidé que dans l'expression $2+3\times4$, l'opérateur $\times$ l'emporte sur l'opérateur $+$ et que le résultat numérique de l'expression vaut 14. On dit alors que la précédence de l'opérateur $\times$ l'emporte sur la précédence de l'opérateur $+$.

Précédence	Opérateur ou séparateur
Forte	. [ ] ()
⋮	++ −− ! ~ $+_{unaire}$ $-_{unaire}$
⋮	× / %
⋮	+ - $+_{concaténation}$
⋮	<< >> >>>
⋮	< > <= >= *instanceof*
⋮	== !=
⋮	&
⋮	^
⋮	|
⋮	&&
⋮	||
⋮	? :
⋮	= $+=$ $-=$ $\times=$ $/=$ $\%=$ $<<=$ $>>=$ $>>>=$ $\&=$ $\wedge=$ $\mid=$
Faible	,

1.13 En ce qui concerne la suite des événements...

- **cours 2** : instructions du langage ; programmation statique vs programmation dynamique (classes, interfaces, héritage...) ; codage et types de données : des numériques (avec l'arithmétique en précision arbitraire, *java.math*), et chaînes de caractères... aux collections ordonnées et non ordonnées (survol de *java.util*),
- **cours 3** : les entrées-sorties (*java.io*, *java.nio* ?) et les mécanismes de sérialisation et de gestion des exceptions,
- **cours 4** : l'*API* réseau standard (*java.net*), le *multithreading* et le "parallélisme" en général (JOMP implémentation de OpenMP en *Java*™, mpiJava implémentation de MPI en *Java*™...),
- **cours 5** : interfaces graphiques, programmation événementielle et manipulation d'images (*java.awt*, *javax.swing*),
- **cours 6** : l'*API* Reflection (*java.lang.reflect*) et la *Java*™ 2 Platform, Enterprise Edition (J2EE) : JNI, *JDBC*™ (*java.sql*), RMI (*java.rmi*, *rmic*, *rmiregistry* et *rmid*) & CORBA (*org.omg* et *idlj*)... (JNDI ?, EJB ?).

Chapitre 2

Instructions du langage - programmation statique vs programmation dynamique - codage et structures de données

Sommaire

2.1 Les instructions du langage

2.1.1 Les instructions conditionnelles et d'aiguillage

Lorsque l'on imbrique les instructions conditionnelles il importe de se souvenir que chaque clause *else* est liée à l'instruction *if* la plus proche qui ne possède pas déjà elle-même une clause *else*.

▶ **forme conditionnelle réduite** : l'opérateur ternaire

L'instruction conditionnelle *if*

L'instruction d'aiguillage *switch*

L'expression à tester est placée entre parenthèses juste après le *switch*. Elle doit impérativement être de type *char*, *byte*, *short* ou *int* !

L'instruction *default* est facultative. Si elle est omise et qu'aucune étiquette ne vérifie l'expression à tester, l'instruction *switch* n'a aucune incidence sur le programme.

Les étiquettes utilisées pour les clauses *case* sont impérativement des expressions constantes (c'est-à-dire une expression composée uniquement de littéraux ou de constantes déclarées *final static*).

```
1  class AiguillageDeChiffresDecimaux
2  {
3      public static void main ( String [] arguments)
4      {
5          int unEntier = Integer . parseInt (arguments [0]);
6
7          switch( unEntier ) {
8          case 2 :
9              System.out . println ( unEntier + " est un chiffre premier" );
10         case 0 : case 4 : case 6 : case 8 :
11             System.out . println ( unEntier + " est un chiffre pair" );
12             break;
13         case 3 : case 5 : case 7 :
14             System.out . println ( unEntier + " est un chiffre premier" );
15         case 1 : case 9 :
16             System.out . println ( unEntier + " est un chiffre impair" );
17             break;
18         default :
19             System.out . println ( unEntier + " n'est pas un chiffre mais un nombre");
20         }
21     }
22 }
```

2.1.2 Les instructions de contrôle de boucles itératives

La boucle itérative *for*

Le test pour la condition de poursuite est impérativement une expression de type *boolean.*

Il est possible d'utiliser le séparateur "," dans les parties d'initialisation ou d'incrémentation de l'instruction *for* pour modifier deux variables (ou plus) de façon synchrone.

Pour réaliser une boucle infinie, il suffit de placer une tautologie (ou expression de valeur logique toujours égale à *true*) dans la condition de poursuite. Si cette condition de test est omise, elle est remplacée par *true*.

Les expressions : `expression-d-initialisation` et `expression-d-incrémentation` peuvent être vides, auquel cas la partie correspondante de la boucle est ignorée.

En *Java*™, contrairement au langage *C++*, si un compteur est déclaré et initialisé dans l'expression d'initialisation d'une boucle *for*, il n'est connu qu'au sein de l'instruction *for*.

La boucle itérative *while*

il est possible de reproduire quasiment avec une boucle *while* le comportement d'une boucle *for*. En effet, l'instruction suivante :

```
1  for ( expression-d- initialisation ; condition-de-poursuite ;
2  expression-d-incrémentation ) { ... séquence-d-instructions ... }
```

est (presque) équivalente à l'instruction :

```
1  expression-d- initialisation ;
2  while ( condition-de-poursuite) {
3      ... séquence-d-instructions ...
4      expression-d-incrémentation;
5  }
```

Pour être plus précis, il faut remarquer que la réaction au *continue* de ces deux instructions diffère. Ainsi, la portion de code suivante affiche les chiffres 0 et 2 sur deux lignes consécutives :

```
1  for ( int i =0 ; i <3 ; i++) {
2      if ( i==1) continue;
3      System.out . println ( i );
4  }
```

tandis que la portion de code suivante (qui semble être a priori équivalente) commence par afficher le chiffre 0 sur une première ligne, puis boucle à l'infini sur la valeur 1 pour la variable *i*, sans arriver à l'incrémenter et en n'affichant plus rien sur la sortie standard :

```
1  int i=0;
2  while( i <3) {
3      if ( i==1) continue;
4      System.out . println ( i ); i++;
5  }
```

La boucle itérative *do…while*

2.1.3 Les instructions de rupture d'enchaînement séquentiel

La mise en place d'une étiquette

La rupture de séquence *break*

```
1   class TableauA2Entrees
2   {
3       public static void main ( String [] arguments)
4       {
5       global :
6           for ( int i =1 ; ; i++) {
7               for ( int j =1 ; ; j++) {
8                   if ( j>i ) {
9                       System.out. println ();
10                      break;
11                  }
12                  if ( i==6)
13                      break global ;
14                  System.out. print ( "(" + i + "," + j + ")" );
15              }
16          }
17      }
18  }
```

```
(1,1)
(2,1) (2,2)
(3,1) (3,2) (3,3)
(4,1) (4,2) (4,3) (4,4)
(5,1) (5,2) (5,3) (5,4) (5,5)
```

La rupture de séquence *continue*

La rupture de séquence *return*

2.1.4 Les instructions de gestion des exceptions, du *multithreading*

2.2 Programmation statique vs programmation dynamique : mise en œuvre de l'objet en *Java*™

2.2.1 Le modificateur *static*

Un membre statique (déclaré avec le modificateur *static*), aussi appelé membre de classe, est un membre qui est partagé par toutes les instances de cette classe. Il n'en existe qu'un seul exemplaire pour toute la classe et non autant de copies qu'il y a d'objets instances de cette classe. Étudions séparément le cas des variables de classe (et non plus variables d'instances), des blocs d'initialisation et des méthodes statiques.

Variable de classe

Une variable de classe ou variable *static* est donc un attribut de classe qui est partagé par tous les objets qui l'instancient. Ainsi, quand une instance de la classe modifie la valeur ou plus généralement accède en écriture à une variable de classe ou variable *static*, il faut s'assurer au préalable qu'il n'y a pas de tentative d'accès concurrent sur la valeur par une autre instance de la classe.

La classe *java.lang.Math* du langage, qui hérite de la classe *Object*, possède ainsi, en plus de ses diverses méthodes permettant de réaliser quelques calculs mathématiques de base, deux variables[1] de classe : *Math.E* et *Math.PI* qui correspondent respectivement à des valeurs approchées du nombre *e* (base du logarithme népérien) et du nombre π.

Bloc d'initialisation *static*

bloc d'instructions encapsulé dans une définition de classe comme peut l'être une méthode, qui est chargé par exemple d'initialiser des variables de classe. Cette zone de code n'appartient en propre à aucune instance de la classe et peut être assimilée à un "constructeur de classe" (un peu comme il existe des constructeurs pour les instances de classe) dans la mesure où elle est capable de réaliser des initialisations complexes de variables statiques. Ce bloc statique est évalué une fois par chargement de classe, soit lors de la première instanciation de la classe, soit lors du premier appel à l'une de ses méthodes statiques.

[1] Il s'agit en fait de constantes puisqu'elles sont aussi déclarées *fi nal*.

L'exemple ci-après initialise une variable de classe qui est une instance de *java.util.Properties*, à l'aide d'un bloc d'initialisation *static*. L'affichage du contenu de cette variable de classe est réalisé par simple appel à la méthode *list* de la classe *Properties* :

```
1  import java.util.Properties;
2  class MaClasse1
3  {
4      static Properties codesDePays;
5      static
6      {
7          // bloc d'initialisation "static"
8          codesDePays = new Properties();
9          codesDePays.setProperty("DE","Allemagne");
10         codesDePays.setProperty("FR","France");
11         codesDePays.setProperty("IT","Italie");
12     }
13     public static void main(String[] arguments)
14     {
15         // il n'est pas obligatoire d'appeler la propriété codesDePays par son
16         // nom complet (ou qualifié) comme nous le faisons ci-dessous, puisque
17         // celle-ci est appelée dans la classe courante :
18         MaClasse1.codesDePays.list(System.out);
19     }
20 }
```

► **Méthode de classe et programmation structurée** Il est impossible de faire appel à une variable d'instance ou à une méthode d'instance au sein d'une méthode statique. De la même façon, il n'est pas concevable de faire appel à l'auto-référence *this* au sein d'une méthode de classe[2], puisque celle-ci n'opère pas sur une instance spécifique de la classe mais sur la classe elle-même. La classe *java.lang.Math* du langage, qui n'est d'ailleurs pas instanciable, définit ainsi un certain nombre de méthodes statiques telles que *Math.sin()* qui retourne la valeur du sinus de son argument flottant double précision.

Pour reprendre l'exemple des codes de pays que nous avons présenté un peu plus tôt, nous pouvons définir une méthode de classe `afficher` chargée de lister le contenu de la variable d'instance `codesDePays`. Cette méthode de classe peut être invoquée soit, indépendamment de toute instance, en utilisant le nom de classe elle-même, soit, comme une méthode d'instance, à partir d'une instance de la classe :

```
1  import java.util.Properties;
2  class MaClasse2
3  {
4      private static Properties codesDePays;
5      static
6      {
7          // bloc d'initialisation "static"
8          codesDePays = new Properties();
9          codesDePays.setProperty("DE","Allemagne");
10         codesDePays.setProperty("FR","France");
11         codesDePays.setProperty("IT","Italie");
12     }
13     static void affiCher()
14     {
15         MaClasse2.codesDePays.list(System.out);
16     }
17     public static void main(String[] arguments)
18     {
19         MaClasse2.affiCher();                    // appel indépendant d'une instance
20         MaClasse2 monObjet = new MaClasse2(); // instanciation de la classe
21         monObjet.affiCher();                     // appel de la méthode de classe via
22                                                  // une instance particulière
23     }
24 }
```

► **Classe interne *static*** L'introduction de classes internes susceptibles d'être déclarées *static* est l'une des évolutions du langage lors de son passage à la version 1.1. Ces classes internes statiques peuvent être insérées dans le corps des interfaces même si ces dernières n'acceptent pas d'implémentation normalement. En fait, vous pouvez déclarer une classe interne comme étant statique lorsque celle-ci n'a pas besoin de faire appel à l'objet instance de sa classe englobante. Par ailleurs, pour qu'une classe interne puisse avoir des membres déclarés *static*, il importe qu'elle soit elle-même déclarée *static*. C'est ce que réalise l'exemple suivant :

```
1  import java.util.Properties;
2  class MaClasse3
3  {
4      // classe interne statique :
5      static class MaClasseInterne
6      {
7          private static Properties codesDePays;
8          static
9          {
10             codesDePays = new Properties();
11             codesDePays.setProperty("DE","Allemagne");
12             codesDePays.setProperty("FR","France");
13             codesDePays.setProperty("IT","Italie");
14         }
15         static void affiCher()
16         {
17             codesDePays.list(System.out);
18         }
19     }
20     // méthode main de la classe englobante :
21     public static void main(String[] arguments)
22     {
23         // appel à la méthode statique de la classe interne :
24         MaClasse3.MaClasseInterne.affiCher();
```

[2] Ni même à la référence *super*, d'ailleurs !

```
25            // instanciation de la classe interne :
26            MaClasse3.MaClasseInterne monObjet = new MaClasse3.MaClasseInterne();
27            monObjet. affi cher ();
28        }
29    }
```

2.2.2 Instanciation et destruction d'objets

Déclaration sans instanciation

Clonage

```
VoitureDeTourisme laLamborghiniDiabloDuPortier = ( VoitureDeTourisme)
    laLamborghiniDiabloDuJardinier.clone ();
```

► **Note :** Pour pouvoir utiliser la méthode *clone* héritée de la classe *Object*, il importe que la classe `VoitureDeTourisme` "implémente l'interface *java.lang.Cloneable*". Si ce n'est pas le cas, la tentative de clonage provoquera immanquablement la levée d'une exception de type *CloneNotSupportedException*. Par ailleurs, pour pouvoir passer le stade de la compilation, vous devez aussi prévoir de gérer ce type d'exception. C'est exactement ce que fait (mais de façon minimaliste) le code ci-après :

```
1     class VoitureDeTourisme extends VéhiculeAMoteur implements Cloneable {
2       // placer ici les attributs, constructeurs et méthodes de la classe...
3       public static void main ( String [] arguments) throws CloneNotSupportedException
4       {
5         VoitureDeTourisme laLamborghiniDiabloDuJardinier = new VoitureDeTourisme();
6         VoitureDeTourisme laLamborghiniDiabloDuPortier = ( VoitureDeTourisme)
7                 laLamborghiniDiabloDuJardinier.clone ();
8       }
9     }
```

Instanciation à l'aide de la méthode *newInstance*

```
1     class MaClasse
2     {
3       MaClasse()
4       {
5         System.out. println ("passage par le constructeur de la classe MaClasse");
6       }
7       public static void main ( String [] arguments)
8             throws IllegalAccessException , InstantiationException
9       {
10        MaClasse premierObjet = new MaClasse();
11        Class référenceSurMaClasse = premierObjet. getClass ();
12        MaClasse deuxièmeObjet = (MaClasse) référenceSurMaClasse.newInstance ();
13      }
14    }
```

Destruction d'objet

garbage collector

vous pouvez supprimer l'exécution asynchrone du *garbage collector* en lançant la machine virtuelle avec une option telle que *-Xnoclassgc* ou *-noasyncgc*. Il vous reste alors encore la possibilité de provoquer le lancement du *ramasse-miettes* à l'aide de la méthode *System.gc()*.

le *ramasse-miettes* se charge de la ressource mémoire qui n'est plus utilisée, il ne prend pas en charge la fermeture de certaines ressources extérieures telles que les flux d'entrée-sortie ou les connexions réseau que vous avez pu ouvrir. Ces opérations restent en effet à votre charge et vous pouvez vous en acquitter en redéfinissant la méthode *finalize* (de modificateur d'accès *protected*) de la classe *Object* au niveau de la classe concernée. Pour ce faire, procédez de la manière suivante :

```
1     protected void fi nalize () throws Throwable
2     {
3       super. fi nalize ();
4       // ... suite des instructions de la méthode finalize...
5     }
```

Vous remarquerez que la méthode *finalize* définie ci-avant, invoque explicitement *super.finalize* afin que la ou les classes mères dont dérive la classe courante puissent elles-mêmes procéder à leur propre "travail" de libération des ressources avant destruction.

Portée des variables et des objets

la portée d'un objet ou d'une variable de type primitif, défini au sein d'un bloc d'instructions entre accolades ("{" et "}") se limite à la fin de ce même bloc d'instructions au niveau de l'accolade fermante "}". Au delà, l'objet n'est plus référencé et l'espace mémoire correspondant est donc susceptible d'être "ramassé".

► **Note :** Une variable ou un objet déjà défini dans un bloc d'instructions ou une méthode ne peut pas être redéfini à l'intérieur d'un sous-groupe d'instructions inclus dans le premier.

```
1     Object unObjet = new Object();
2     {
3       Object unObjet = new Object();
4       // erreur de compilation
5     }
```

2.2.3 Consultation ou activation de propriétés statiques ou dynamiques

2.2.4 Héritage, surcharge et redéfinition

surcharge de méthode ⇔encore appelée polymorphisme paramétrique ⇔associer, au sein d'une classe, des traitements différents (c'est-à-dire des corps de méthodes différents) à un même nom de méthode en les distinguant uniquement à partir de leurs signatures respectives.

La redéfinition de méthode, contrairement à la surcharge d'une méthode par une classe fille, consiste à remplacer l'implémentation d'une méthode de la classe mère par un code différent au niveau de la classe fille, la signature restant inchangée.

```
1   class NombreComplexe
2   {
3     private double partieRéelle , partieImaginaire ;
4     public double module()
5     {
6       return Math.sqrt ( partieRéelle * partieRéelle + partieImaginaire * partieImaginaire );
7     }
8   }
9
10  class ImaginairePur extends NombreComplexe
11  {
12    private double partieImaginaire ;
13    public double module()
14    {
15      return Math.abs( partieImaginaire );
16    }
17  }
```

► L'auto-référence *this*

L'auto-référence `this` permet de lever l'ambiguïté sur un identificateur quand, pour des questions de lisibilité du code, le paramètre d'un accesseur porte le nom de l'attribut de l'objet qu'il est chargé de modifier.

L'auto-référence `this` permet d'appeler un autre constructeur sur l'objet courant et de réutiliser ainsi d'autres portions de code. Il s'agit ici d'un cas de polymorphisme paramétrique (surcharge du constructeur). C'est exactement ce que nous faisons dans l'exemple suivant :

```
1   class NombreComplexe
2   {
3       double partieRéelle , partieImaginaire ;
4
5       NombreComplexe(double partieRéelle,double partieImaginaire )
6       {
7           this . partieRéelle = partieRéelle ;
8           this . partieImaginaire = partieImaginaire ;
9       }
10      NombreComplexe(NombreComplexe C)
11      {
12          this (C. partieRéelle ,C. partieImaginaire );
13      }
14  }
```

Ordre d'appel des constructeurs dans le cas d'un héritage

Le langage fournit, par défaut, un appel au constructeur vide de la super-classe au niveau du constructeur de la sous-classe (l'instruction *super() ;* est inclue implicitement en première ligne de chaque constructeur de sous-classe). S'il n'existe pas de constructeur vide dans votre super-classe, mais qu'il y existe des constructeurs avec arguments, vous devez appeler explicitement l'un de ces constructeurs avec l'instruction *super(. . . liste d'arguments. . .) ;*

Comme dans le cas de surcharge de constructeur où l'instruction *this() ;* doit se trouver impérativement en première instruction de constructeur, il est obligatoire de placer l'appel au constructeur de la super-classe *super() ;* en première instruction du constructeur de la sous-classe.

L'exemple qui suit précise l'ordre d'appel des constructeurs dans le cas d'un héritage :

```
1   class ClasseMère
2   {
3       protected int attribut = 1;
4       ClasseMère()
5       {
6           attribut += 10;
7           System.out. println ("passage par le constructeur de ClasseMère," + affi cher ());
8       }
9       String affi cher ()
10      {
11          return ("l' attribut vaut :" + attribut );
12      }
13  }
14
15  class ClasseFille extends ClasseMère
16  {
17      ClasseFille ()
18      {
19          super (); // instruction implicite
20          attribut += 100;
21          System.out. println ("passage par le constructeur de ClasseFille ," + affi cher ());
22      }
23
```

```
24          public  static  void main ( String [] arguments)
25          {
26              ClasseFille   monObjet = new ClasseFille ();
27          }
28      }
```

Le résultat obtenu à l'exécution est le suivant :

```
passage par le constructeur de ClasseMère, l'attribut vaut : 11
passage par le constructeur de ClasseFille, l'attribut vaut : 111
```

2.2.5 Valeurs par défaut

Alors que pour toutes les références, la valeur par défaut à l'initialisation est la valeur *null*, dans le cas des variables de type primitif, cette valeur est fonction du type, comme nous pouvons le constater dans le tableau ci-après :

Type	Valeur initiale par défaut
boolean	*false*
char	\u0000
byte	(byte) 0
short	(short) 0
int	0
long	0L
float	+0.0f
double	+0.0d

2.2.6 Les modificateurs de visibilité

Modificateur	Entités concernées	Action
public	Classe, interface, méthode et variable d'instance	Les membres publics sont accessibles partout où la classe l'est elle même. Ils sont hérités par les sous-classes. Si une classe ou une interface est déclarée *public*, alors elle est accessible par tout programme pouvant charger le paquetage où elle se trouve.
protected	Classe interne, méthode et variable d'instance	Les membres d'une classe qui sont déclarés *protected* sont accessibles et hérités par les sous-classes, même si celles-ci sont définies dans un autre *package*. Ils sont aussi accessibles par tout code se trouvant dans le même *package*. Malgré son nom, le niveau de protection *protected* est moins élevé que le niveau par défaut.
protection par défaut	Classe, interface, méthode et variable d'instance	Dans le cas de la protection par défaut, les membres de la classe ne sont accessibles qu'au sein du *package* qui contient la classe. Une sous-classe d'une classe ayant le niveau de protection par défaut n'accède aux membres de la classe mère que si elle se trouve dans le même *package* et pas sinon.

private	Classe interne, méthode et variable d'instance	Il s'agit du niveau de protection le plus "contraignant". Les membres privés ne sont accessibles qu'au niveau de l'instance de la classe considérée par les méthodes de la classe elle-même. Un membre privé n'est pas hérité par une sous-classe.

► **Quelques grands principes concernant la visibilité :** Si vous avez quelques difficultés à attribuer un niveau de visibilité correct à vos classes, interfaces, méthodes et variables d'instances, souvenez-vous de quelques règles de base. Les interfaces, ainsi que leurs méthodes et leurs attributs (qui sont impérativement et implicitement des constantes de type *final static*, même si vous ne le spécifiez pas à proprement parler ce qui n'est pas très recommandé pour la lisibilité de votre code) n'acceptent, en plus du modificateur d'accès par défaut, que le modificateur *public*. Dans les deux cas, que le modificateur soit explicite ou non, l'interface et ses membres sont *public*.

Une variable d'instance de classe ne doit pas, a priori, être déclarée *public* sauf si c'est une constante (qui est donc définie *final*). Si cette variable d'instance n'est pas destinée à être "publiée" et si elle ne représente qu'une propriété d'implémentation de la classe, il faut la rendre *private* ainsi que ses accesseurs. S'il s'agit d'une variable d'instance qui représente une propriété d'interface destinée à être publiée, vous pouvez la déclarer *private* à condition que ses accesseurs soient *protected* ou *public*.

Si vous définissez un certain niveau de protection pour une méthode au sein d'une classe, sachez que vous ne pouvez pas redéfinir cette méthode avec un niveau de protection plus élevé dans une de ses classes filles. Vous avez juste le droit de baisser le niveau de protection de cette méthode lorsque vous la redéfinissez (en effet, une instance de sous-classe peut parfois être manipulée sous forme d'une instance de sa super-classe, ce qui provoquerait des incompatibilités dans le cas d'invocation de méthode "mère" aux droits plus restreints). En ce qui concerne les attributs, par contre, il est aussi possible de les recouvrir au niveau des sous-classes avec une visibilité plus réduite.

2.2.7 Le modificateur *final*

Une méthode dont la définition est précédée du modificateur *final* est une méthode qui ne peut pas être redéfinie dans une sous-classe. L'intérêt de déclarer une méthode *final* réside dans le fait de s'assurer que, pour des questions de sécurité, la fonctionnalité rattachée à cette méthode finale ne peut pas être réécrite par un autre développeur au cours d'un simple héritage de classe. Un autre intérêt est lié à l'optimisation (relative) du code puisque le fait de déclarer une méthode avec le modificateur *final* permet d'inhiber la liaison dynamique de code lors de l'exécution et d'en informer le compilateur qui peut alors essayer d'optimiser la méthode concernée.

Une classe dont la définition est précédée du modificateur *final* est une classe qui ne peut pas être dérivée et dont toutes les méthodes sont implicitement déclarées *final*. En effet, puisque la classe ne peut pas être sous-classée (dérivée), il n'est pas concevable d'envisager que ses méthodes puissent être redéfinies dans une hypothétique sous-classe. Dans l'arborescence dont la classe *Object* est la racine, une classe *final* correspond à une feuille ; c'est exactement le cas de la classe *java.lang.Integer* qui ne peut pas être dérivée.

Constante d'instance et constante de classe

```
1   class ValeurEtReferenceConstante
2   {
3       int VariableDInstance = 128;
4
5       public static void main ( String [] arguments)
6       {
7           final ValeurEtReferenceConstante unObjet = new ValeurEtReferenceConstante ();
8           final int uneValeur; // uneValeur est une constante "blank final"
9
10          // les deux instructions qui suivent sont licites :
11          unObjet. VariableDInstance /= 2; // c'est la référence "unObjet" qui est "final"
12          uneValeur = 17;                  // il s'agit de l'initialisation de "uneValeur"
13
14          System.out. println (uneValeur + " " + unObjet. VariableDInstance );
15
16          // les deux instructions qui suivent provoquent des erreurs à la compilation :
17          unObjet = new ValeurEtReferenceConstante (); // modification de la référence "unObjet"
18          uneValeur--; // modification de la valeur de "uneValeur"
19      }
20  }
```

► **Remarque :** Une variable ou un objet est dit *blank final* lorsqu'il est défini sans être initialisé immédiatement. Quoiqu'il en soit, il est impératif qu'il soit initialisé avant d'être utilisé. Il conserve naturellement son caractère non modifiable à l'issue de cette initialisation "retardée".

Une variable d'instance[3] déclarée à l'aide du modificateur *final* sans être initialisée (une variable d'instance *blank final*, donc) qui est ensuite initialisée au niveau d'un constructeur est une constante d'instance. Il s'agit en effet d'une constante propre à chaque instance de la classe.

Dans le cas contraire, si nous considérons une variable de classe déclarée *final* ou une variable d'instance initialisée à la définition et déclarée *final*, alors il s'agit d'une constante de classe puisque sa valeur ne peut être modifiée par aucune de ses instances. Pourtant, dans le seconde cas, comme la variable n'est pas déclarée *static*, il ne s'agit pas réellement d'une variable (constante) de classe mais bien plutôt d'une variable d'instance qui est constante pour l'ensemble de la classe. L'exemple suivant expose ces différences :

```
1   class ConstantesDeClasseEtInstance
2   {
3       final static int vraieConstanteDeClasse = 13;
4       final int fausseConstanteDeClasse = 128;
5       final int constanteDInstance;
6
7       ConstantesDeClasseEtInstance(int valeur)
8       {
9           constanteDInstance = valeur;
10      }
11      void afficher()
12      {
13          System.out.println("vraieConstanteDeClasse = " +
14                             ConstantesDeClasseEtInstance.vraieConstanteDeClasse);
15          System.out.println("fausseConstanteDeClasse = " +
16                             this.fausseConstanteDeClasse);
17          System.out.println("constanteDInstance = " + this.constanteDInstance);
18      }
19
20      public static void main(String[] arguments)
21          {
22              ConstantesDeClasseEtInstance unObjet = new ConstantesDeClasseEtInstance(17);
23              ConstantesDeClasseEtInstance autreObjet = new ConstantesDeClasseEtInstance(18);
24
25              unObjet.afficher();
26              autreObjet.afficher();
27          }
28  }
```

À l'exécution, nous obtenons le résultat suivant :

```
vraieConstanteDeClasse = 13
fausseConstanteDeClasse = 128
constanteDInstance = 17
vraieConstanteDeClasse = 13
fausseConstanteDeClasse = 128
constanteDInstance = 18
```

final en argument de méthode

L'insertion du modificateur *final* dans la signature d'une méthode au niveau d'un argument (ou de plusieurs), signifie que le compilateur se charge de vérifier qu'aucune tentative de modification (en valeur pour une variable de type primitif ou en référence pour un objet) n'est faite sur l'entité au sein de la méthode. L'exemple ci-après illustre cette notion :

```
1   class FinalEnArgument
2   {
3       int unAttribut;
4
5       static void uneMéthode(final FinalEnArgument unObjet, final int uneValeur)
6       {
7           unObjet.unAttribut++;            // manipulation licite
8           // les deux instructions qui suivent provoquent des erreurs à la compilation :
9           unObjet = new FinalEnArgument(); // modification de référence déclarée final
10          uneValeur++;                     // modification de valeur déclarée final
11      }
12
13      public static void main(String[] arguments)
14          {
15              FinalEnArgument objetBidon = new FinalEnArgument();
16              int valeurBidon = 13;
17              FinalEnArgument.uneMéthode(objetBidon,valeurBidon);
18          }
19  }
```

2.2.8 Classe abstraite et interface

Une classe abstraite n'est pas instanciable

```
1   abstract class Véhicule {
2     // corps de la classe comme défini précédemment...
3   }
4
5   abstract class VéhiculeAMoteur {
6     // corps de la classe comme défini précédemment...
7   }
```

[3] Et non pas une variable de classe déclarée à l'aide du modificateur *static*.

Lorsque l'on utilise une classe abstraite, on peut ne définir que partiellement son implémentation (chaque méthode non implémentée doit elle-même être déclarée *abstract*) en laissant aux sous-classes le soin de la compléter. Cette façon de procéder peut s'avérer utile lorsqu'une propriété dynamique ne peut pas être factorisée au niveau d'une classe généraliste et qu'elle doit impérativement être implémentée au niveau de ses sous-classes.

▶ **Une méthode abstraite** est une méthode précédée du modificateur *abstract*, dont le corps n'est pas défini au niveau de la classe. Définir une méthode abstraite au sein d'une classe, oblige à implémenter le corps de la méthode en question dans chacune de ses sous-classes, à moins que les sous-classes ne soient elles-mêmes déclarées abstraites. La présence d'une méthode abstraite au sein d'une classe, vous oblige à déclarer toute la classe comme étant elle-même abstraite sous peine d'erreur de type *class [...] must be declared abstract* à la compilation.

```
1   abstract class MesureDePerformance
2   {
3     MesureDePerformance()
4     {
5       // à chaque instanciation, on fixe la date initiale du traitement :
6       long dateDébut = System.currentTimeMillis();
7       // on effectue le traitement "abstrait" qui reste à définir en sous-classe :
8       traitementAMesurer();
9       // on affiche le temps mis par ce traitement pour s'exécuter :
10      System.out.println("Temps global de calcul :" +
11                         (System.currentTimeMillis() - dateDébut) + " ms");
12    }
13    // méthode abstraite (parce qu'inconnue a priori) à mesurer :
14    abstract void traitementAMesurer();
15  }
```

```
1   class TraitementQuelconque extends MesureDePerformance
2   {
3     void traitementAMesurer()
4     {
5       int somme = 0;
6       for (short i =0; i<Short.MAX_VALUE; i++)
7           somme += i;
8       System.out.println(somme);
9     }
10    public static void main(String[] arguments)
11    {
12      // le traitement ayant été défini, il suffit d'instancier la
13      // classe TraitementQuelconque pour obtenir son temps d'exécution :
14      TraitementQuelconque monBenchmark = new TraitementQuelconque();
15    }
16  }
```

Interface

En *Java*™, l'interface peut-être vue comme une classe abstraite "pure", dans le sens où elle pousse ce concept dans ses retranchements. Une interface est donc quasiment un cas particulier de classe dont toutes les méthodes sont implicitement abstraites et dont toutes les variables d'instances sont implicitement des constantes *final* et *static*.

```
1   interface VéhiculeADeuxRoues
2   {
3     static final short NOMBRE_DE_ROUES = 2;
4     boolean estTombéATerre();
5     void releverSonDeuxRoues();
6     boolean êtreGaréSurLaBéquille();
7   }
8
9
10  // intérêt :
11    public static void main(String[] arguments)
12    {
13      VéhiculeADeuxRoues[] deuxRouesTombésATerre = new VéhiculeADeuxRoues[4];
14
15      deuxRouesTombésATerre[0] = new Vélomoteur();
16      deuxRouesTombésATerre[1] = new Cyclomoteur();
17      deuxRouesTombésATerre[2] = new Bicyclette();
18      deuxRouesTombésATerre[3] = new Motocyclette();
19
20      for (int i =0; i < deuxRouesTombésATerre.length; i++)
21          deuxRouesTombésATerre[i].releverSonDeuxRoues();
22    }
```

Cette façon de procéder permet de factoriser l'appel aux traitements (qui peuvent être très variés et hétérogènes) en évitant de se soucier de tester l'appartenance d'un objet à une classe ou à une autre. La recherche de la méthode adaptée à l'objet courant se fait dynamiquement, à l'exécution, sans que nous ayons à nous en préoccuper.

▶ **Récapitulatif :** Une interface ⇔classe entièrement abstraite, définie avec le mot clef *interface* du langage. Toutes ses méthodes sont implicitement ou non *abstract* et toutes ses variables d'instances sont des constantes de classes qui sont implicitement ou non *final static*.

Une interface accepte, en plus du modificateur d'accès par défaut, le niveau de protection *public*. Dans le premier cas, l'interface et ses membres sont accessibles de l'ensemble du *package*, dans le second cas, l'interface et ses membres (qui sont alors aussi implicitement *public*) sont accessibles de n'importe où. **Si une classe ne peut hériter, par le mot clef *extends*, des propriétés que d'une seule super-classe, elle peut, a contrario, implémenter (au sens de hériter de), par le mot clef *implements*, plusieurs interfaces**. C'est la raison pour laquelle nous vous invitons à préférer l'interface à la classe abstraite lorsque le choix s'offre à vous (remarque : dans le cas d'une UI un adaptateur — *Adapter* — peut cependant parfois être plus souple pour le développeur qu'un délégué — *Listener*).

2.2.9 Héritage multiple d'interfaces, emboîtement et collisions d'identifiants

Collision d'identifiants

Ainsi, **à la compilation de l'interface `Interface4`, l'exemple ci-après s'interrompt sur une erreur de type *The method void méthodeA() inherited from interface Interface2 is incompatible with the method of the same signature inherited from interface Interface3. They must have the same return type*** !

```
1   interface Interface1
2   {
3     void méthode1();
4   }
5
6   interface Interface2 extends Interface1
7   {
8     void méthode2();
9     void méthodeA();
10  }
11
12  interface Interface3 extends Interface1
13  {
14    void méthode3();
15    int méthodeA() throws java.io.IOException;
16  }
17
18  interface Interface4 extends Interface2, Interface3
19  { /* sans méthode propre */ }
20
21  // fichier UneClasse.java
22  class UneClasse implements Interface4
23  {
24    public void méthode1()
25    {
26    }
27    public void méthode2()
28    {
29    }
30    public void méthode3()
31    {
32    }
33    public void méthodeA()
34    {
35    }
36    public int méthodeA() throws java.io.IOException;
37    {
38      return 1;
39    }
40
41    public static void main ( String [] arguments)
42    {
43      UneClasse unObjet = new UneClasse();
44      // dilemme insoluble correctement pour le compilateur Java,
45      // s'agit-il de "int méthodeA()" ou "void méthodeA()" :
46      unObjet.méthodeA();
47    }
48  }
```

Imbrications de classes et d'interfaces : classes et interfaces internes

```
1   class ImbricationDeClassesEtInterfaces
2   {
3     public interface Interface1
4     {
5       public interface Interface11
6       {
7         public interface Interface111
8         {
9         }
10        static class Classe112 implements Interface111
11        {
12        }
13      }
14    }
15    // la classe Classe2 implémente une interface imbriquée dans une interface
16    // qui est elle même imbriquée dans une interface :
17    private class Classe2 implements Interface1.Interface11.Interface111
18    {
19    }
20    // une classe interne peut avoir un niveau de visibilité protected ou private :
21    public class Classe3 implements Interface1
22    {
23      private class Classe31
24      {
25        void uneMéthode()
26        {
27          // on peut définir une classe interne dans le corps d'une méthode.
28          // Dans ce cas, elle n'est accessible qu'au sein de la méthode où elle
29          // est définie.
30          class Classe311
31          {
32          }
33        }
34      }
35    }
36  }
```

Pour récapituler, une classe interne peut être définie comme membre d'une classe, tout comme peut l'être une méthode ou une variable d'instance, mais elle peut aussi être définie à l'intérieur d'une méthode. Sa visibilité est similaire à celle d'une autre propriété de classe et est fonction de son modificateur d'accès (elle peut notamment être déclarée *private*).

Une classe qui est définie à l'intérieur d'une méthode n'est pas accessible de l'extérieur de celle-ci. Une interface peut, de son côté, être imbriquée à l'intérieur d'une interface englobante ou à l'intérieur d'une classe, à condition que celle-ci ne soit pas une classe interne.

► **Classe interne anonyme.** Dans le même registre, il est tout a fait possible d'envisager d'utiliser des classe internes anonymes, c'est-à-dire des classes internes qui n'ont pas d'identifiant de classe ni de constructeur.

Ces classes anonymes peuvent implémenter une interface ou dériver une autre classe, ce qui permet par exemple d'invoquer leurs méthodes. C'est une évolution du langage que nous vous invitons à n'utiliser qu'avec circonspection. . .

2.2.10 Le cas particulier de la méthode *main*

Si la méthode *main* n'existe pas ou si sa signature est incorrecte[4], l'exécution s'interrompt sur un message du type *Exception in thread "main" java.lang.NoSuchMethodError : main*. Sinon, l'application se lance en exécutant cette méthode.

Comme vous pouvez le remarquer, il est impératif que la méthode *main* soit *static*, puisqu'il ne s'agit pas d'une méthode d'instance mais d'une méthode de classe. En ce qui concerne le niveau de visibilité, il est préférable que cette méthode soit publique de façon à ce que l'on puisse l'invoquer de n'importe quelle autre classe en la précédant du nom de sa propre classe, comme ci-après :

```
1   // fichier ClasseA :
2   class ClasseA
3   {
4     public static void main ( String [] lesArguments)
5     {
6       System.out. println ("ClasseA appelle ClasseB");
7       ClasseB.main(null );
8     }
9   }
10
11  // fichier ClasseB :
12  {
13    public static void main ( String [] lesArguments)
14    {
15      System.out. println ("Ici ClasseB , message bien reçu !" );
16    }
17  }
```

Les arguments de la ligne de commande sont rangés dans un tableau de chaînes de caractères qui commence à l'indice 0. Ils sont accessibles par : `lesArguments[0]`, . . . , `lesArguments[lesArguments.length-1]`.

2.2.11 Classe interne et propriétés *private*

Comportement attendu. . .

```
1   public class AttributsPrivate {
2   }
3   class ClasseA {
4     private int i;
5   }
6   class ClasseB {
7     private ClasseA unObj;
8     ClasseB () {
9       unObj.i = 17;
10    }
11  }
```

```
# javac AttributsPrivate.java
AttributsPrivate.java :9 : Variable i in class ClasseA not
accessible from class ClasseB.
    unObj.i = 17 ;
          ^
1 error
```

Comportement inattendu. . .

```
1   public class AttributsPrivate {
2     class ClasseA {
3         private int i;
4     }
5     class ClasseB {
6         private ClasseA unObj;
7         ClasseB () {
8             unObj.i = 17;
9         }
10    }
11  }
```

. . . ne génère aucune erreur à la compilation (et le fait de faire précéder les déclarations de classes internes du modificateur "private" n'y change rien) !

[4] Vous pouvez toutefois modifier son niveau de protection.

2.3 Le passage des arguments : valeur ou référence

Hormis les types primitifs (*boolean*, *char*, *byte*, *short*, *int*, *long*, *float*, *double*), tous les autres types de données manipulés en *Java*™ sont des objets, qu'il s'agisse de tableaux ou non. Les types primitifs sont manipulés par valeur tandis que les autres sont manipulés par référence.

```
1  public class ValeurOuReference
2  {
3      private int attribut ;
4
5      ValeurOuReference(int valeur )
6      {
7          attribut = valeur ;
8      }
9
10     static void uneMéthode (int unEntier , int unTableau [], ValeurOuReference unObjet)
11     {
12         unEntier++;
13         unTableau[0]++;
14         unObjet. attribut ++;
15     }
16
17     public static void main ( String [] arguments)
18     {
19         int monEntier = 3;
20         int monTableau[] = {101};
21         ValeurOuReference monObjet = new ValeurOuReference(227);
22
23         System.out. println ("Avant passage : " + monEntier
24                              + ", " + monTableau[0]
25                              + ", " + monObjet. attribut
26                              );
27         ValeurOuReference.uneMéthode(monEntier,monTableau,monObjet);
28         System.out. println ("Après passage : " + monEntier
29                              + ", " + monTableau[0]
30                              + ", " + monObjet. attribut
31                              );
32     }
33 }
```

```
Avant passage : 3, 101, 227
Après passage : 3, 102, 228
```

2.4 Le codage des données

```
1  #include < stdio .h>
2  int main() {
3    int i = 1;
4    if ( *( char *) &i == 1 )
5      printf (" little endian\n");
6    else printf ("big endian\n");
7    exit (0);
8  }
```

- architectures *big endian* : Sun SPARC/ULTRA, Motorola 68K, HP PA-RISC,
- architectures *little endian* : Intel x86 et Pentium, AMD K*, DEC Alpha RISC,
- architectures *bi-endian* : SGI MIPS, IBM/Motorola PowerPC,
- l'Internet (IP) est *big endian*. . .

et la JVM est aussi *big endian* !

2.4.1 Représentation des types entiers

```
1  public class BornesTypesEntiers
2  {
3      public static void main ( String arguments[])
4      {
5          System.out. println ("Un \"" + Byte.TYPE
6                               + "\" est compris entre " + Byte.MIN_VALUE
7                               + " et " + Byte.MAX_VALUE);
8          System.out. println ("Un \"" + Short .TYPE
9                               + "\" est compris entre " + Short .MIN_VALUE
10                              + " et " + Short .MAX_VALUE);
11         System.out. println ("Un \"" + Integer .TYPE
12                              + "\" est compris entre " + Integer .MIN_VALUE
13                              + " et " + Integer .MAX_VALUE);
14         System.out. println ("Un \"" + Long.TYPE
15                              + "\" est compris entre " + Long.MIN_VALUE
16                              + " et " + Long.MAX_VALUE);
17     }
18 }
```

permet d'apprendre que :

```
Un "byte" est compris entre -128 et 127
Un "short" est compris entre -32768 et 32767
Un "int" est compris entre -2147483648 et 2147483647
Un "long" est compris entre -9223372036854775808 et 9223372036854775807
```

```
1  public class ConvertisseurBase
2  {
3      private String binaire , octal ,hexadécimal;
4
5      ConvertisseurBase ( String valeur )
6      {
7          int valeurEntière = Integer . parseInt ( valeur );
8          binaire = Integer . toBinaryString ( valeurEntière );
9          octal = Integer . toOctalString ( valeurEntière );
10         hexadécimal = Integer .toHexString( valeurEntière );
11     }
12     void affi cher ()
13     {
14         System.out. println ( binaire + " " + octal + " " + hexadécimal);
15     }
16
17     public static void main(String [] arguments){
18         for( int i =0 ; i < arguments. length ; i++) {
19             try {
20                 new ConvertisseurBase(arguments[i ]). affi cher ();
21             }
22             catch ( NumberFormatException uneException) {
23                 System.err . println (arguments[ i ] + " n'est pas une valeur entière ");
24             }
25         }
26     }
27 }
```

```
# java ConvertisseurBase 8 -8
1000 10 8
11111111111111111111111111111000 37777777770 fffffff8
```

Codage avec bit de signe

l'arithmétique associée se complique !

Codage en complément à 1

Pour additionner deux entiers (et dans le cas où le résultat ne dépasse pas la capacité de stockage dans la précision choisie), on procède par addition binaire avec addition de la retenue finale au résultat. Le problème, avec ce type de codage en complément à 1, est lié à la représentation de 0 en base 2.

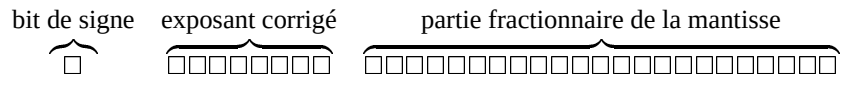
Codage en complément à 2

L'addition de 8 et de -8 donne donc, dans cette représentation : $00001000 + 11111000 = 00000000$ (ici, on n'additionne pas la retenue finale au résultat).

► **Remarque :** On devrait plus précisément parler de complément à 2^n où n désigne le nombre de bits sur lesquels est stocké l'entier (8 dans le cas présent puisque l'on travaille sur le type *byte*). En effet, le codage de -8 dans le cas du complément à 2 sur 8 bits est égal au codage de l'entier $2^8 - 8 = 248$ qui vaut 11111000.

2.4.2 Représentation des types fbttants

un *float* doit respecter le "IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985" sur 4 octets (32 bits) tandis qu'un *double* doit respecter le même standard sur 8 octets (64 bits). Ce standard IEEE ne se contente pas de définir le format d'un nombre fractionnaire signé, puisqu'il inclut aussi la notion de "zéro machine", les infinis signés et une valeur très spéciale appelée *NaN*.



En ce qui concerne la mantisse, le standard IEEE impose qu'elle soit normalisée de la forme binaire suivante : $1, <$ *une suite de* 0 *ou de* 1 $>$. Une mantisse qui suit cette norme est donc toujours comprise entre 1 (possible) et 2 (exclue). Comme le premier bit de la mantisse vaut toujours 1, il est implicite et n'est pas stocké à proprement parler au niveau machine

La conséquence de cette normalisation de la mantisse à une valeur de l'intervalle $[1, 2[$ implique d'adapter la valeur de l'exposant. A propos de l'exposant justement, les 8 bits de stockage offrent $2^8 = 256$ valeurs différentes possibles, ce qui correspond à des nombres allant de 2^{-127} (l'extrêmement petit) à 2^{127} (l'extrêmement grand). Pour ne pas avoir

à gérer un signe au niveau de l'exposant, on convient d'ajouter automatiquement et de façon transparente à l'utilisateur la valeur 127 à l'exposant réel au moment du stockage.

Pour un triplet de codage machine (s,e,m) donné, et dans le cas d'un *float*, la valeur binaire flottante associée est donc :

$$(-1)^s \times (1,m)_2 \times 2^{e-127}$$

Par conséquent, 13.7 est représenté en base 2 par la séquence suivante : 1101.10 1100 1100 1100 1100... Après normalisation, cette séquence devient : $2^3 \times 1.10110\ 1100\ 1100\ 1100\ 1100\ldots$ Par conséquent, si l'on met bout-à-bout le bit de signe, les 8 bits d'exposant (qui valent $3+127=130$ ou encore 10000010 en base 2) et les 23 bits de mantisse normalisée (attention : on ne stocke pas le bit de la partie entière qui vaut toujours 1), on obtient :

$$\overbrace{1}^{\text{bit de signe}}\ \overbrace{10000010}^{\text{exposant corrigé}}\ \overbrace{10110110011001100110011}^{\text{partie fractionnaire de la mantisse}}$$

	Champ signe	Champ exposant	Champ mantisse
float	1 bit	8 bits	23 bits
double	1 bit	11 bits	52 bits

	Champ signe	Champ exposant	Champ mantisse
float	0 : positif 1 : négatif	$\text{exp}_{réel}$=$\text{exp}_{stocké}$-127	$\text{mantisse}_{réelle}$ = 1,$\text{mantisse}_{stockée}$
double	0 : positif 1 : négatif	$\text{exp}_{réel}$=$\text{exp}_{stocké}$-1023	$\text{mantisse}_{réelle}$ = 1,$\text{mantisse}_{stockée}$

Quantité	Champ signe	Champ exposant	Champ mantisse
0.0	0	00000000	0000000 00000000 00000000
-0.0	1	00000000	0000000 00000000 00000000
NaN	0	11111111	**1**000000 00000000 00000000
$+\infty$	0	11111111	0000000 00000000 00000000
$-\infty$	1	11111111	0000000 00000000 00000000

Premier exemple

$$\begin{array}{rccc} f: & \mathbb{R} & \longrightarrow & \mathbb{R} \\ & x & \longmapsto & \frac{1}{x+1-x} \end{array}$$

Vaut *Infinity* ou 1.0 selon la précision et la valeur de x (pb à partir de 2^24 = 16777216 et 2^53 = 9007199254740992).

Second exemple

```
1  class ErreurDeConversion
2  {
3      public static void main ( String [] arguments)
4      {
5          if (.1 f > .1)
6              System.out. println ( "C'est plutôt catastrophique !" );
7      }
8  }
```

Quantité	Champ signe	Champ exposant	Champ mantisse
hline 0.1f	0	01111011	1001100 11001100 11001101
(float) 0.1	0	01111011	1001100 11001100 11001101
(double) 0.1f	0	011 11111011	1001 10011001 10011001 10100000 00000000 00000000 00000000
0.1	0	011 11111011	1001 10011001 10011001 10011001 10011001 10011001 10011010

Pour être plus précis encore, le flottant simple précision 0.1f est arrondi, lorsqu'il est manipulé en machine, à

la valeur 0.100000001490116119375. Le fait de l'étendre en double précision ne change pas sa valeur puisqu'on complète sa mantisse par des 0. Par ailleurs, le flottant double précision 0.1 est arrondi, lorsqu'il est manipulé en machine, à la valeur 0.10000000000000000556. Comme cette dernière valeur est plus faible que la première, il est donc normal, d'un point de vue machine, d'avoir l'inégalité suivante : $0.1\texttt{f} > 0.1$, même si cela est assez dérangeant d'un point de vue pratique.

2.5 Quelques structures de données

2.5.1 Les chaînes de caractères

Toute séquence de caractères entre double quote est une chaînes de caractères ou plus exactement un littéral de type chaîne de caractères, c'est-à-dire une instance de la classe *java.lang.String*.

Immuabilité

Un objet possède le caractère d'immuabilité lorsque ses variables d'instances (ses attributs) ne peuvent être accédées qu'en lecture, c'est-à-dire lorsque celles-ci sont déclarées à l'aide du modificateur d'accès *private* — ce qui les empêche d'être manipulées par une portion de code qui n'est pas directement définie au niveau de la classe — et lorsqu'aucune des méthodes de la classe ne les accède en écriture, à l'exception du constructeur.

```
1  import java.util.*;
2  import java.text.DateFormat;
3
4  public class ClasseDObjetsImmuables
5  {
6      private int attributNonModifiable;
7      private String dateDInstanciation;
8
9      ClasseDObjetsImmuables(int uneValeur)
10     {
11         attributNonModifiable = uneValeur;
12         dateDInstanciation =
13             DateFormat.getDateInstance(DateFormat.LONG,
14                                        Locale.FRANCE).format(new Date());
15     }
16
17     public int getAttributNonModifiable()
18     {
19         return attributNonModifiable;
20     }
21
22     public ClasseDObjetsImmuables doubler()
23     {
24         return new ClasseDObjetsImmuables(getAttributNonModifiable()*2);
25     }
26
27     public void afficher(String commentaire)
28     {
29         System.out.println(commentaire + " l'entier " +
30                            getAttributNonModifiable() +
31                            " a été instancié le " + dateDInstanciation);
32     }
33
34     public static void uneMéthode(ClasseDObjetsImmuables UnObjetImmuable)
35     {
36         ClasseDObjetsImmuables tmp = UnObjetImmuable.doubler();
37         UnObjetImmuable.afficher("Dans uneMéthode :");
38         tmp.afficher("Dans uneMéthode :");
39     }
40
41     public static void main(String[] arguments)
42     {
43         ClasseDObjetsImmuables unObjet = new ClasseDObjetsImmuables(13);
44         unObjet.afficher("Dans le main :");
45         uneMéthode(unObjet);
46         unObjet.afficher("Dans le main :");
47     }
48 }
```

À l'exécution, la classe `ClasseDObjetsImmuables` produit la trace suivante :

```
Dans le main : l'entier 13 a été instancié le 5 septembre 2000
Dans uneMéthode : l'entier 13 a été instancié le 5 septembre 2000
Dans uneMéthode : l'entier 26 a été instancié le 5 septembre 2000
Dans le main : l'entier 13 a été instancié le 5 septembre 2000
```

L'immuabilité, si elle peut paraître élégante de bien des points de vue, parce qu'elle permet de manipuler des objets à partir d'alias en s'assurant de la constance de leurs valeurs, a cependant un inconvénient majeur. Elle génère en effet autant d'instanciations (et de ce fait même autant d'invocations de constructeurs) qu'il y a de modifications

[5] Il aurait aussi pu être arrondi à la valeur machine 0.09999994039535522 45, mais celle-ci est moins proche de la valeur théorique 0.1 que la précédente. Ce n'aurait donc pas été conforme aux directives de la norme IEEE 754 qui préconisent un arrondi au plus proche.

[6] Il aurait aussi pu être arrondi à la valeur machine 0.09999999999999997772, mais celle-ci est moins proche de la valeur théorique 0.1 que la précédente. Ce n'aurait donc pas été, ici encore, conforme aux directives de la norme IEEE 754 qui préconisent un arrondi au plus proche.

d'attributs d'un tel objet ! Le surcoût en temps d'exécution peut alors vite s'avérer non négligeable, d'autant plus que ces instanciations multiples ont une incidence sur l'activation du *ramasse-miettes*.

Pour palier à ce défaut, il est tout à fait possible d'adjoindre une classe "compagne" non immuable à toute classe immuable. Lorsqu'on envisage de modifier massivement un objet immuable, il faut alors lui préférer son clone de la classe "compagne", quitte à revenir à un objet immuable à la fin des modifications. En ce qui concerne la manipulation de chaînes de caractères, on peut tout à fait considérer que la classe *StringBuffer* est la classe "compagne" de la classe immuable *String*.

Classe *java.lang.String*

```
String unVersDeBaudelaire = "Là, tout n'est qu'ordre et beauté,\n";
String unAutreVersDeBaudelaire = new String("Luxe, calme et volupté.\n");
```

Cette seconde manière de procéder est cependant plus coûteuse que la première puisqu'elle produit en fait une double instanciation ! En effet, avant d'être réellement pris en compte par le constructeur de la classe *String*, le littéral chaîne de caractères est converti sous forme d'un objet *String*. Notre propre instanciation se surimpose alors à la première, qui est implicite.

▶ **Concaténation de chaînes de caractères :** l'opérateur +, la méthode *concat* :

```
String versDAragon = "Tous deux adoraient la belle\n".concat("Prisonnière des soldats");
```

Conversion en une chaîne de caractères	Conversion d'une chaîne de caractères
d'un *boolean* : String s=String.valueOf((boolean)true); String s=new Boolean(true).toString();	en un *boolean* : boolean b=Boolean.valueOf("false"). booleanValue();
d'un *byte* : String s=String.valueOf((byte) 17); String s=Byte.toString((byte) 17);	en un *byte* : byte b=Byte.valueOf("17"). byteValue();
d'un *char* : String s=String.valueOf('z'); String s=new Character('z').toString();	en un *char* : char c="z".charAt(0);
d'un tableau de *char* : String s=String.valueOf(new char[] {'a','b','c'});	en un tableau de *char* : char [] tableau="une chaîne". toCharArray();
d'un *short* : String s=String.valueOf((short) 17); String s=Short.toString((short) 17);	en un *short* : short l=Short.valueOf("19"). shortValue();
d'un *int* : String s=String.valueOf(17); String s=Integer.toString(17);	en un *int* : int i=Integer.parseInt("19");
d'un *long* : String s=String.valueOf((long) 17); String s=Long.toString((long) 17);	en un *long* : long l=Long.parseLong("19");
d'un *float* : String s=String.valueOf((float) 17.1); String s=Float.toString((float) 17.1);	en un *float* : float f=Float.valueOf("19.1"). floatValue();
d'un *double* : String s=String.valueOf(17.1); String s=Double.toString(17.1);	en un *double* : double d=Double.valueOf("19.1"). doubleValue();

▶ **Conversion de/vers une chaîne de caractères** D'une manière générale, toute classe hérite de la méthode *toString* définie dans la classe *Object*, et nombreuses sont les classes du paquetage *java.lang* qui redéfinissent cette méthode pour lui donner du sens (c'est notamment le cas des classes "enveloppes" telles que *Integer*, *Double*...). De la même manière, comme vous pouvez le constater à la lecture du tableau qui précède, la classe *String* fournit un ensemble de méthodes *valueOf* (il s'agit ici de polymorphisme paramétrique, puisque c'est à partir de la signature, ou plus exactement du type des arguments, qu'un traitement adapté est appliqué) qui permettent de convertir une entité d'un type primitif en une chaîne de caractères.

▶ **Longueur d'une chaîne : méthode *length***

Méthode	Signification
int indexOf(String uneChaîne)	Cette méthode cherche la première occurrence de la chaîne `uneChaîne` au sein de la chaîne de caractères de l'instance courante. Elle renvoie un entier compris entre 0 et la longueur de la chaîne de caractères de l'instance courante moins un, correspondant à la position du premier caractère de l'occurrence au sein de la chaîne de l'instance courante `this.length()`. S'il n'y a aucune occurrence d'`uneChaîne` dans l'instance courante, le résultat vaut -1.
int indexOf(String uneChaîne,int n)	Cette méthode est équivalente à la précédente, à ceci près qu'elle démarre la recherche de la première occurrence d'`uneChaîne` à partir du n^{e} caractère.
int lastIndexOf(String uneChaîne)	Cette méthode cherche la dernière occurrence de la chaîne `uneChaîne` au sein de la chaîne de caractères de l'instance courante. Elle renvoie un entier compris entre 0 et la longueur de la chaîne de caractères de l'instance courante moins un, correspondant à la position du premier caractère de la dernière occurrence au sein de la chaîne de l'instance courante `this.length()`. Si la chaîne `uneChaîne` n'y a aucune occurrence, le résultat vaut -1.
int lastIndexOf(String uneChaîne,int n)	Cette méthode est équivalente à la précédente, à ceci près qu'elle limite la recherche de la dernière occurrence d'`uneChaîne` à une portion de chaîne de l'instance courante placée avant le n^{e} caractère.

```
1  System.out. println ("Il fait beau".indexOf("a" ,5));
2  // résultat à l'exécution : 10
3  System.out. println ("Il fait beau". lastIndexOf("a" ,5));
4  // résultat à l'exécution : 4
```

Méthode	Signification
char charAt(int n)	Cette méthode retourne le caractère placé à la position *n* dans la chaîne courante. *n* prend une valeur comprise entre 0 (indice du premier caractère de la chaîne courante) et $length() - 1$ (indice du dernier caractère de la chaîne courante).
String substring(int début,int fin)	Cette méthode retourne une portion de chaîne de caractères de la chaîne courante allant du caractère placé à la position `début` jusqu'au caractère placé à la position `fin-1`.
String trim()	Cette méthode retourne une portion de chaîne de caractères de la chaîne courante en supprimant de celle-ci, en début et en fin de chaîne, les caractères de contrôle de l'ASCII tels que la tabulation ($\backslash t$ ou $\backslash u0009$), le retour chariot ($\backslash r$ ou $\backslash u000d$), l'espace ($\sqcup$ ou $\backslash u0020$), la fin de ligne ($\backslash n$ ou $\backslash u000a$), la fin de fichier ($\backslash f$ ou $\backslash u000c$)...

▶ **Extractions** À titre d'exemple, nous vous proposons les deux instructions suivantes :

```
1  System.out. println ("Il fait beau". substring (3,7));
2  // résultat à l'exécution : fait
3  System.out. println ("  \t\n\ tIl fait beau \n\t\n". trim ());
4  // résultat à l'exécution : Il fait beau
```

Méthode	Signification
int compareTo(String uneChaîne)	Cette méthode compare, lexicographiquement (caractères Unicode), la chaîne de caractères `uneChaîne` avec la chaîne de caractères de l'instance courante. Le code de retour est un entier strictement positif si la chaîne courante est strictement supérieure (dans la relation d'ordre lexicographique) à la chaîne `uneChaîne`, strictement négatif si la chaîne courante est strictement inférieure (dans la relation d'ordre lexicographique) à la chaîne `uneChaîne`, et nul dans le cas d'une exacte concordance. Pour plus d'explication, nous vous invitons à vous reporter à la "remarque importante" qui suit immédiatement le tableau.
int compareToIgnoreCase(String uneChaîne)	Cette méthode est équivalente à la précédente en ignorant les différences entre les majuscules et les minuscules.
boolean endsWith(String uneChaîne)	Cette méthode renvoie *true* si la chaîne de l'instance courante *this* se termine par une portion de chaîne de caractères strictement identique à la chaîne `uneChaîne` ou si `uneChaîne` est la chaîne vide, et *false* sinon.
boolean equals(Object unObjet)	Cette méthode renvoie *true* si et seulement si l'objet passé en argument est une instance de la classe *String*, non *null*, strictement identique à la chaîne de caractères de l'instance courante, et *false* sinon.
boolean equalsIgnoreCase(String uneChaîne)	Cette méthode renvoie *true* si la chaîne passée en argument est identique à la chaîne de caractères de l'instance courante, en ignorant les différences entre les majuscules et les minuscules. Elle renvoie *false* sinon.
boolean regionMatches(boolean b,int n,String uneChaîne,int m, int long)	Cette méthode compare une portion de `long` caractères de la chaîne courante (en commençant à l'indice `n`) avec une portion de `long` caractères de la chaîne de caractères `uneChaîne` (en commençant à l'indice `m`). Le résultat retourné vaut *false* si `n` ou `m` est négatif, ou si `n+long` est supérieur à la longueur de la chaîne courante, ou si `m+long` est supérieur à la longueur de la chaîne `uneChaîne`, ou s'il existe un entier `k` de l'intervalle $\{0,\ldots,long-1\}$ pour lequel charAt(n+k) ne renvoie pas le même caractère lorsqu'on l'applique à la chaîne `uneChaîne` et à l'instance courante (en ignorant les différences entre les majuscules et les minuscules si le booléen `b` vaut *true*). Le résultat retourné vaut *true* sinon.
boolean startsWith(String uneChaîne)	Cette méthode renvoie *true* si la chaîne de l'instance courante *this* commence par une portion de chaîne de caractères strictement identique à la chaîne `uneChaîne` ou si `uneChaîne` est la chaîne vide, et *false* sinon.

```
1  System.out. println ("àçï". equalsIgnoreCase("ÀÇÏ"));
2  // résultat à l'exécution : true
3  System.out. println (new String("toto"). equals("toto"));
4  // résultat à l'exécution : true
5  System.out. println (new StringBuffer("toto"). equals("toto"));
6  // résultat à l'exécution : false
7  System.out. println ("il  fait  beau !" .regionMatches(true ,8, "Quel beau temps !" ,5,4));
8  // résultat à l'exécution : true
```

► **Remarque importante :** L'internationalisation n'est pas prise en compte dans la relation d'ordre qui est appliquée par ces méthodes de comparaison. Ainsi, si dans les dictionnaires de langue française le terme "été" est placé avant[7]

[7] En effet, en français, les caractères `e` et `é` sont considérés comme identiques du point de vue de leur ordonnancement dans le dictionnaire. Les accents, comme la cédille, n'ont pas d'influence sur la relation d'ordre.

le mot “eux”, il importe de savoir que l’instruction qui suit produit un 132 à l’exécution car la chaîne de caractères `été` suit la chaîne de caractères `eux` dans la relation d’ordre lexicographique appliquée aux chaînes de caractères Unicodes :

```
System.out. println ("été".compareTo("eux"));
```

En effet, la méthode `chaîne1.compareTo(chaîne2)` parcourt caractère par caractère les deux chaînes `chaîne1` et `chaîne2` tant que, pour une position `p` donnée au sein de ces chaînes inférieure à la longueur de la plus petite d’entre elles, la valeur calculée de la différence `chaîne1.charAt(p)−chaîne2.charAt(p)` est non nulle. Si, pour une position donnée, cette valeur est différente de la valeur 0, alors la méthode se termine et retourne la valeur en question en guise de résultat. Sinon, lorsque cet indice de position `p` atteint la valeur de la longueur de la plus petite des deux chaînes, c’est la valeur résultant de la différence `chaîne1.length()−chaîne2.length()` qui est renvoyée.

Comme le caractère `e` est représenté par la valeur Unicode[8] $\backslash u0065$ (ce qui correspond à la valeur 101 en base 10) et comme le caractère `é` est représenté par la valeur Unicode $\backslash u00e9$ (ce qui correspond à la valeur 233 en base 10), alors la différence entre les représentations numériques de ces deux caractères vaut bien 132 en base 10, ce que vous pouvez vérifier avec l’une des deux instructions qui suit :

```
1  System.out. println ('\u00e9'−'\u0065');
2  System.out. println ('é'−'e');
```

► **Divers**

Méthode	Signification
String replace(char vieuxCaractère,char nouveauCaractère)	Cette méthode remplace, au sein de la chaîne de caractères de l’instance courante, toutes les occurrences du caractère `vieuxCaractère` par le caractère `nouveauCaractère` et retourne la nouvelle chaîne après transformation.
String toLowerCase()	Cette méthode convertit toutes les lettres de la chaîne de caractères de l’instance courante en minuscule et retourne la nouvelle chaîne après transformation.
String toUpperCase()	Cette méthode convertit toutes les lettres de la chaîne de caractères de l’instance courante en minuscule et retourne la nouvelle chaîne après transformation.
String trim()	Cette méthode retourne une portion de chaîne de caractères de la chaîne courante en supprimant de celle-ci, en début et en fin de chaîne, les caractères de contrôle de l’ASCII tels que la tabulation ($\backslash t$ ou $\backslash u0009$), le retour chariot ($\backslash r$ ou $\backslash u000d$), l’espace (␣ ou $\backslash u0020$), la fin de ligne ($\backslash n$ ou $\backslash u000a$), la fin de fichier ($\backslash f$ ou $\backslash u000c$)...

► **Ne pas confondre == et *equals*** Ainsi, l’instruction suivante produit le résultat *false*, car les chaînes de caractères correspondantes ne sont pas stockées au même emplacement :

```
System.out. println (("danger" == "DANGER".toLowerCase()));
```

alors que l’instruction de comparaison de contenus de chaînes de caractères qui suit retourne *true*, ce qui est tout à fait cohérent :

```
System.out. println ("danger".equals("DANGER".toLowerCase()));
```

La classe *java.lang.StringBuffer*

classe “compagne” de la classe immuable *String*. Une instance de la classe *StringBuffer* ⇔ tampon extensible pour les caractères. Contrairement à une instance de la classe *String* qui est immuable, une instance de la classe *StringBuffer* peut voir son contenu modifié sans pour autant être ré-instanciée. On supprime ainsi les instanciations intermédiaires et par là même les allocations de mémoire inutiles.

Ainsi, à des fins d’optimisation, le compilateur transforme implicitement une concaténation de chaînes de caractère en une manipulation sur une instance de *StringBuffer*. L’instruction :

```
String uneChaîne = "Madame "+ "la" + " Marquise";
```

est donc en fait implémentée par le compilateur sous la forme suivante :

```
String uneChaîne = new StringBuffer (). append("Madame ").append("la").
    append(" Marquise"). toString ();
```

[8] En *Java*™, un caractère est stocké sur deux octets et il est représenté, du point de vue de l’Unicode par une séquence de type $\backslash uXXXX$ où chaque X est un chiffre hexadécimal de l’intervalle $\{0,\ldots,9,a,\ldots,f\}$ ou $\{0,\ldots,9,A,\ldots,F\}$.

► **Gestion de la capacité d'un *StringBuffer* :** Le tampon d'une instance de la classe *StringBuffer* a une capacité au moins égale à la longueur de la chaîne de caractères qu'il stocke. S'il est instancié vide, sa capacité initiale est de 16 caractères par défaut, mais il est tout à fait possible de fixer sa capacité à une valeur différente à l'initialisation ou de l'augmenter au cours de son utilisation avec la méthode *setLength*. S'il est instancié avec une chaîne de caractères, sa taille est fixée à la longueur de la chaîne de caractères passée en argument de constructeur, incrémentée de 16 unités. L'exemple qui suit montre les évolutions de la taille d'un *StringBuffer* et la gestion automatique de cette capacité :

```
1   class EvolutionDeCapacite
2   {
3       public static void main ( String [] arguments)
4       {
5           StringBuffer tmp = new StringBuffer ();
6           System.out. println (tmp.capacity ()); // capacité initiale par défaut
7           tmp.append("Ceci est une chaîne de plus de 16 caractères .\n")
8                .append("Comment va évoluer la capacité du StringBuffer courant ?" );
9           System.out. println (tmp.capacity ()); // capacité après concaténation
10          tmp.setLength (500); // augmentation arbitraire de la capacité
11          System.out. println (tmp.capacity ());
12          tmp.setLength (210); // diminution arbitraire de la capacité - reste sans effet
13          System.out. println (tmp.capacity ());
14          // capacité d'un StringBuffer initialisé avec une chaîne de caractères :
15          System.out. println (new StringBuffer("bonjour"). capacity ());
16      }
17  }
```

```
16
102
500
500
23
```

► **Principales méthodes d'accès en écriture à un *StringBuffer*.** Les quelques méthodes que nous présentons dans le tableau ci-après permettent d'accéder en écriture (de modifier) le contenu de la chaîne de caractères qui est stockée sous forme d'une instance de la classe *StringBuffer*.

Méthode	Signification
StringBuffer append(String uneChaîne)	Cette méthode permet de concaténer à la chaîne de l'instance courante de *StringBuffer* le contenu de la chaîne `uneChaîne`.
StringBuffer deleteCharAt(int p)	Cette méthode permet de supprimer le $p+1^e$ caractère de la chaîne de l'instance courante.
StringBuffer insert(int p, String uneChaîne)	Cette méthode permet d'insérer le contenu de la chaîne de caractère `uneChaîne` dans la chaîne de l'instance courante à partir du $p+1^e$ caractère.
StringBuffer replace(int p, int q, String uneChaîne)	Cette méthode permet de remplacer le contenu de la chaîne de l'instance courante placé entre le $p+1^e$ caractère et le q^e caractère par le contenu de la chaîne de caractères `uneChaîne`.
StringBuffer reverse()	Cette méthode permet de renverser la chaîne de caractères de l'instance courante de *StringBuffer*.
void setCharAt(int p, char leCaractère)	Cette méthode permet de remplacer le $p+1^e$ caractère de la chaîne de l'instance courante par le caractère `leCaractère`.

À titre d'exemple, nous vous proposons les instructions suivantes :

```
1   System.out. println (new StringBuffer("C'est " ).append((String ) null ));
2   // résultat à l'exécution : C'est null
3   System.out. println (new StringBuffer("C'est trop !" ). deleteCharAt (7));
4   // résultat à l'exécution : C'est top !
5   System.out. println (new StringBuffer("Quel événement !" ). insert (5," tragique " ));
6   // résultat à l'exécution : Quel tragique événement !
7   System.out. println (new StringBuffer("Il pleut beaucoup !" ). replace (3,8,"ne pleut pas" ));
8   // résultat à l'exécution : Il ne pleut pas beaucoup !
9   System.out. println (new StringBuffer("Relacer"). reverse ());
10  // résultat à l'exécution : recaleR
11  new StringBuffer("une balle" ).setCharAt (5,' i ');
12  // stocke la chaine "une bille" dans le StringBuffer considéré
```

2.5.2 Les collections

Collections indexées

ensembles d'objets ayant un mode d'accès (encore appelé moyen d'indexation ou index[9]) commun.

► **Collections statiques à indexation entière ordonnée : tableaux** En *Java*™, tous les tableaux sont des instances de classes qui héritent directement de la classe mère *java.lang.Object*[10].

```
TypeDesEléments [] nomDuTableau;
TypeDesÉléments autreNomDeTableau[];
TypeDesEléments [] nomDuTableau = new TypeDesEléments(nombreDÉléments);
TypeDesEléments [] nomDuTableau = {uneInstanceDeTypeDesEléments, ...,
  uneInstanceDeTypeDesEléments};
TypeDesEléments [] nomDuTableau = new TypeDesEléments [] {
  uneInstanceDeTypeDesEléments , ..., uneInstanceDeTypeDesEléments};
```

La dimension du tableau est omise à la déclaration. Elle est fixée lors de l'instanciation ou déterminée à partir de l'initialisation quand on fait une déclaration combinée à une initialisation. L'indice des éléments du tableau commence à zéro, `nomDuTableau[0]` permet donc d'accéder au premier d'entre eux et `nomDuTableau[nomDuTableau.length-1]` permet d'accéder au dernier élément du tableau. En effet, comme un tableau est un objet, il possède un attribut public *length* qui correspond à son nombre d'éléments.

► **Remarque :** À l'instanciation d'un tableau multi-dimensionnel, il n'est pas nécessaire de spécifier la taille du tableau dans toutes les dimensions. La syntaxe de l'opérateur *new* impose juste de spécifier la taille de la première dimension qui est la plus importante.

```
1  class TriangleDePascal
2  {
3      int [][] tab;
4
5      TriangleDePascal(int dimension) {
6          tab = new int [ dimension ][];
7          for(int ligne =0 ; ligne<dimension ; ligne++) {
8              // instanciation de la ligne courante :
9              tab[ ligne ] = new int[ ligne +1];
10             // initialisation des éléments de la ligne de la ligne courante :
11             tab[ ligne ][0] = 1;
12             for(int colonne =1 ; colonne<ligne ; colonne++)
13                 tab[ ligne ][ colonne ] = tab[ ligne −1][colonne−1] + tab[ ligne −1][colonne];
14             tab[ ligne ][ ligne ] = 1;
15         }
16     }
17
18     void affi cher () {
19         for(int ligne =0 ; ligne<tab. length ; ligne++) {
20             // affichage de toute la ligne courante :
21             for(int colonne =0 ; colonne<tab[ ligne ]. length ; colonne++)
22                 System.out. print (tab[ ligne ][ colonne ] + " " );
23             System.out. println ();
24         }
25     }
26
27     public static void main ( String [] arguments)
28     {
29         // instanciation d'un Triangle de Pascal sur 7 lignes :
30         TriangleDePascal unPetitTriangle = new TriangleDePascal (7);
31         // affichage du contenu de ce Triangle :
32          unPetitTriangle . affi cher ();
33     }
34 }
```

► **Note :** Les éléments d'un tableau d'objets sont des références à ces objets et non des instances de ceux-ci. Tant qu'il n'y a pas eu d'initialisation des éléments du tableau, leur valeur par défaut est *null*.

Les tableaux sont très pratiques tant que l'on a connaissance de la quantité exacte de données à manipuler et tant que celle-ci ne varie pas au cours de l'exécution. Cette limitation relativement majeure est supprimée si l'on décide d'employer la classe *java.util.Vector*.

► **Collections dynamiques à indexation entière ordonnée : *Vector*** La classe *java.util.Vector* permet de manipuler des tableaux dynamiques d'objets (de type générique *Object*), c'est-à-dire des tableaux dans lesquels on peut insérer ou supprimer des éléments situés à des endroits arbitraires. L'index reste cependant entier. Comme les méthodes de cette classe acceptent des arguments de type *Object*, on peut instancier des objets *Vector* comportant n'importe quel type d'élément. On peut même s'offrir le luxe de manipuler un même *Vector* de différents types d'objets dérivant de *Object* ! Attention toutefois à la manipulation des divers éléments car il vous faudra bien finir par les reconvertir du type *Object* générique dans leur type d'origine[11] !

[9] Un index est une valeur clef qui permet de retrouver rapidement un élément de la structure de données.

[10] Vous pouvez le vérifi er avec l'instruction suivante : `System.out. println (nomDuTableau.getClass(). getSuperclass ());`.

[11] Pour éviter la levée d'exception de type *ClassCastException* due à la tentative de coercition d'un objet dans un type très différent de son type initial, il vous faudra faire un appel permanent à l'opérateur *instanceof* ou à la méthode *isInstance()* de la classe *java.lang.Class* !

```
class CotationEnBourse
{
    String nomDeLaValeur;
    float coursDeLaValeur;

    CotationEnBourse(String valeur, float cours)
    {
        nomDeLaValeur = valeur;
        coursDeLaValeur = cours;
    }
    void affi cher ()
    {
        System.out. println (nomDeLaValeur + " : " + coursDeLaValeur);
    }
}
```

```
import java. util .Enumeration;
import java. util .Vector;

class SiteBoursier
{
    Vector tableauDesValeurs;

    SiteBoursier ()
    {
        tableauDesValeurs = new Vector();
    }

    void introductionDeValeur (CotationEnBourse uneValeur)
    {
        tableauDesValeurs .addElement(uneValeur);
        System.out. println ("=> Vous avez entré la valeur numéro : " + nombreDeValeurs());
    }

    void suppressionDeValeur(CotationEnBourse uneValeur)
    {
        if ( tableauDesValeurs .remove(uneValeur)) {
            System.out. println ("Vous venez de supprimer la valeur :" );
            uneValeur. affi cher ();
            System.out. println ("=> Il vous reste " + nombreDeValeurs() + " valeurs !" );
        }
        else System.err . println ("Cette valeur n' existe pas !" );
    }

    void suppressionDeValeur( int index)
    {
        if ( index < nombreDeValeurs()) {
            CotationEnBourse supprimée = ( CotationEnBourse) tableauDesValeurs .remove(index);
            if ( supprimée != null ) {
                System.out. println ("Vous venez de supprimer la valeur :" );
                supprimée. affi cher ();
                System.out. println ("=> Il vous reste " + nombreDeValeurs() + " valeurs !" );
            }
            else System.err . println ("Cette valeur n' existe pas !" );
        }
        else System.err . println ("Cette valeur n' existe pas !" );
    }

    int nombreDeValeurs()
    {
        return tableauDesValeurs . size ();
    }

    void affi chageDeValeur (CotationEnBourse uneValeur)
    {
        affi chageDeValeur ( tableauDesValeurs . lastIndexOf (uneValeur ));
    }

    void affi chageDeValeur ( int index)
    {
        (( CotationEnBourse) tableauDesValeurs . get ( index )). affi cher ();
    }

    void listingDesValeurs ()
    {
        System.out. println ("Vous avez " + nombreDeValeurs() + " valeurs ." );
        // utilisation d'une Enumeration :
        Enumeration lesValeurs = tableauDesValeurs . elements ();
        while ( lesValeurs .hasMoreElements())
            (( CotationEnBourse) lesValeurs .nextElement ()). affi cher ();
    }

    public static void main ( String [] arguments)
    {
        // commençons par instancier la place boursière que nous allons développer :
        SiteBoursier placeDeParis = new SiteBoursier ();
        // définissons ensuite 4 valeurs :
        CotationEnBourse valeurA = new CotationEnBourse("Société A et Cnie",100f );
        CotationEnBourse valeurB = new CotationEnBourse("valeur B et Cnie",440f );
        CotationEnBourse valeurC = new CotationEnBourse("Société C et Cnie",500f );
        CotationEnBourse valeurD = new CotationEnBourse("Société D et Cnie",120f );
        // introduisons ces valeurs sur la place boursière instanciée précédemment :
        placeDeParis . introductionDeValeur ( valeurA );
        placeDeParis . introductionDeValeur ( valeurB );
        placeDeParis . introductionDeValeur ( valeurC );
        placeDeParis . introductionDeValeur ( valeurD );
        // affichons l'ensemble des valeurs cotées sur notre place boursière :
        placeDeParis . listingDesValeurs ();
        // supprimons la 3ème valeur et la valeur 'valeurB' :
        placeDeParis . suppressionDeValeur (3);
        placeDeParis . suppressionDeValeur ( valeurB );
        // supprimons-les à nouveau même si elles ne sont plus cotées a priori :
        placeDeParis . suppressionDeValeur (3);
        placeDeParis . suppressionDeValeur ( valeurB );
        // affichons l'ensemble des valeurs encore en cotation sur notre place boursière :
        placeDeParis . listingDesValeurs ();
    }
}
```

Le côté un peu particulier de cette classe réside dans l'emploi d'une instance de l'interface *java.util.Enumeration* pour lister l'ensemble des valeurs. Alors qu'une implémentation plus classique de ce listing aurait consisté à faire un :

```
for( int index =0 ; index < nombreDeValeurs () ; index++)
    (( CotationEnBourse) tableauDesValeurs . get ( index )). affi cher ();
```

Nous procédons par :

```
1  Enumeration lesValeurs = tableauDesValeurs.elements();
2  while ( lesValeurs.hasMoreElements())
3      ((CotationEnBourse) lesValeurs.nextElement()). afficher ();
```

Cet exemple que nous venons de présenter nous a permis de créer un vecteur d'objets et de le manipuler dynamiquement. Son principal défaut est relatif à son système d'indexation par valeur entière. L'ordre d'entrée en bourse n'est assurément pas le bon critère de recherche, une indexation à base de nom de valeur pourrait, par exemple, faciliter l'implémentation de notre modèle.

► **Pile : *Stack*** La classe *java.util.Stack* du langage dérive de la classe *java.util.Vector* et implémente la notion de pile[12] pour des objets génériques. La classe pile, telle qu'elle est conçue en *Java*™, reprend toutes les méthodes de la classe mère des *Vector* et l'enrichit de cinq nouvelles fonctionnalités très adaptées à la manipulation des piles. Une pile est donc vue comme un cas particulier de tableau dynamique agrémenté du *push* (action d'empiler un élément en sommet de pile), du *pop* (action de dépiler un élément du sommet de pile), du *peek* (simple consultation, sans dépilement, de l'objet situé au sommet de la pile) ainsi que d'une méthode *empty* pour tester si la pile est vide et d'une méthode *search* qui recherche un élément au sein d'une pile et renvoie sa position.

```
1  import java.util.Stack;
2  import java.util.EmptyStackException;
3  import java.io.BufferedReader;
4  import java.io.InputStreamReader;
5  import java.io.IOException;
6
7  class Parenthesage
8  {
9      String expression;
10     Stack pileDesParenthèses;
11
12     Parenthesage ( String uneExpression)
13     {
14         // initialisons l'expression et instancions la pile des parenthèses :
15         expression = uneExpression;
16         pileDesParenthèses = new Stack();
17     }
18
19     boolean valide ()
20     {
21         try {
22             // parcourons un à un chacun des caractères de l'expression
23             for ( int i=0; i<expression.length (); i++) {
24                 switch ( expression.charAt(i)) {
25                 case '(':
26                     // si le caractère courant est une parenthèse ouvrante, on l'empile :
27                     pileDesParenthèses.push(new Character ('('));
28                     break;
29                 case ')':
30                     // si le caractère courant est une parenthèse fermante, on considère que
31                     // celle-ci referme une parenthèse ouverte précédemment et on la dépile :
32                     pileDesParenthèses.pop();
33                     break;
34                 }
35             }
36         }
37         catch ( EmptyStackException e) {
38             // dépiler une pile vide provoque la levée d'une exception de type
39             // EmptyStackException. Si c'est le cas c'est que dans l'expression, il y a au
40             // moins une parenthèse fermante qui n'a pas été ouverte au préalable :
41             return false;
42         }
43         // si la "pileDesParenthèses" est vide en sortie, cela signifie que le parenthésage
44         // est correct puisque toutes les parenthèses ouvertes sont refermées. Sinon le
45         // parenthésage est incorrect :
46         if ( pileDesParenthèses.empty())
47             return true;
48         else return false;
49     }
50
51     public static void main ( String [] arguments)
52     {
53         try {
54             // définissons le tampon de lecture de l'entrée standard :
55             BufferedReader stdin = new BufferedReader(new InputStreamReader(System.in));
56             // instancions un nouvel objet de type Parenthesage :
57             System.out.print("Entrez votre expression à valider :");
58             Parenthesage laChaîne = new Parenthesage( stdin.readLine());
59             // validons l'expression associée à cet objet du point de vue du parenthésage :
60             if ( laChaîne.valide())
61                 System.out.println("=> expression bien parenthésée !");
62             else System.out.println("=> expression mal parenthésée !");
63         }
64         catch ( IOException e) {
65             System.err.println("Problème à la saisie de l'expression !");
66         }
67     }
68 }
```

► **Collections dynamiques à indexation par chaîne : *Properties*** Collections d'objets sous une forme plus générale de type "clef/valeur" où la clef et la valeur sont des chaînes de caractères et où la clef offre un moyen efficace de retrouver la valeur associée… c'est une structure de données de type "dictionnaire".

L'instruction qui suit permet de récupérer (par le biais de la méthode statique *getProperties* de la classe *java.lang.System*) les variables d'environnement et de les afficher sur la sortie standard (à l'aide de la méthode *list* de la classe *java.util.Properties*) :

[12] La pile est une structure de données de type LIFO (*Last In First Out*), ce qui signifie que le dernier élément qui y est introduit est aussi le premier à en sortir.

```
System. getProperties (). list (System.out );
```

```
1  import java . util . Properties ;
2  import java . util .Enumeration;
3  import java . io . IOException;
4  import java . io . File ;
5  import java . io . FileInputStream ;
6  import java . io . FileOutputStream;
7  import java . io . BufferedReader;
8  import java . io . InputStreamReader;
9
10 class CodesDePays
11 {
12     Properties lesCodes;
13
14     CodesDePays()
15     {
16         lesCodes = new Properties ();
17     }
18
19     void chargerLesCodesDePays(String nomFichier)
20     {
21         try {
22             FileInputStream entrée = new FileInputStream(new File(nomFichier));
23             // chargement de la table des propriétés depuis un fichier du disque :
24             lesCodes.load( entrée );
25             System.out. println ("Chargement de la table des codes terminé." );
26         }
27         catch ( IOException e ) {
28             System.err . println ("Pas de chargement de la table des codes ..." );
29         }
30     }
31
32     void insererCodes ()
33     {
34         String codePays,nomPays;
35         boolean continuer = true;
36         // définissons le tampon de lecture de l'entrée standard :
37         BufferedReader stdin = new BufferedReader(new InputStreamReader(System.in));
38         do {
39             try {
40                 System.out. print ("Entrez le code pays ( ou retour chariot pour terminer ) : " );
41                 codePays = stdin .readLine ();
42                 System.out. print ("Entrez le nom du pays (ou retour chariot pour terminer ) : " );
43                 nomPays = stdin .readLine ();
44                 // si l'une des deux chaînes est vide, on s'arrête :
45                 if (! codePays.equalsIgnoreCase("") && ! nomPays.equalsIgnoreCase(""))
46                     lesCodes. setProperty (codePays,nomPays);
47                 else continuer = false ;
48             }
49             catch ( IOException e ) {
50                 System.err . println ("Problème à la lecture des données en entrée !" );
51             }
52         } while ( continuer );
53     }
54
55     void affi cher ()
56     {
57         System.out. println ("Contenu de la table des codes/noms de pays :" );
58         // utilisation d'une énumération :
59         Enumeration énumérationDesCodes = lesCodes.propertyNames();
60         while ( énumérationDesCodes.hasMoreElements()) {
61             String clef = ( String ) énumérationDesCodes.nextElement();
62             System.out. println ( clef + " " + lesCodes. getProperty ( clef ));
63         }
64         // nous aurions pu simplement appeler : lesCodes.list(System.out);
65     }
66
67     void enregistrerLesCodesDePays( String nomFichier)
68     {
69         try {
70             FileOutputStream sortie = new FileOutputStream(new File(nomFichier));
71             // enregistrement de la table des propriétés dans un fichier sur le disque :
72             lesCodes. store ( sortie ,"Codes de pays à la date du ..." );
73             System.out. println ("Enregistrement de la table des codes terminé." );
74         }
75         catch ( IOException e ) {
76             System.err . println ("Problème à l' enregistrement de la table des codes ..." );
77         }
78     }
79
80     public static void main ( String [] arguments)
81     {
82         // création de la table de codes pays :
83         CodesDePays tab = new CodesDePays();
84         // chargement du fichier des codes pays (s'il existe) :
85         tab .chargerLesCodesDePays("codesPays.txt" );
86         // insertion de nouveaux codes de pays :
87         tab . insererCodes ();
88         tab . affi cher ();
89         // enrichissement du fichier des codes pays :
90         tab .enregistrerLesCodesDePays("codesPays. txt" );
91     }
92 }
```

La classe *java.util.Properties* bien que très utile dans certains cas particuliers de traitements de données de type clefs/valeurs est cependant trop contraignante quant aux types mêmes des clefs et des valeurs. Celles-ci doivent en effet impérativement être des chaînes de caractères (*String*) ! Pour étendre cette notion, nous allons donc maintenant nous intéresser à une collection dynamique à indexation de type quelconque.

► **Collections dynamiques à indexation par objet : *Hashtable* ou tableaux associatifs** La classe *java.util.Hashtable* propose un certain nombre de méthodes pour enrichir une table de hachage (*put* et *putall*), lui retirer une de ou toutes ses associations clefs/valeurs (*remove* ou *clear*), tester si une clef ou une valeur s'y trouve (*contains*, *containsKey* ou *containsValue*) ou si la table est vide (*isEmpty*)... Pour lister l'ensemble de ses clefs et/ou de ses valeurs, il faut au préalable implémenter une *Enumeration* des clefs (à l'aide de la méthode *keys*) et la parcourir ensuite. L'exemple

ci-après instancie une table de hachage, l'enrichit de trois associations clefs/valeurs de types bien différents à l'aide de la méthode *put* puis en liste le contenu par le biais d'une *Enumeration* :

Intérêt d'une telle structure de données à l'aide d'un exemple simple, où l'on n'a pas connaissance a priori de la quantité et de la valeur des données à traiter. Comment dénombrer les occurrences de chaque mot d'un texte ?

```
1  import java.util.Enumeration;
2  import java.util.Hashtable;
3  import java.io.StreamTokenizer;
4  import java.io.File;
5  import java.io.FileReader;
6
7  public class DecompteDesMotsDUnTexte
8  {
9      StreamTokenizer fluxEntrant;
10     Hashtable tableDesMots;
11
12     DecompteDesMotsDUnTexte (String nomDeFichier) throws java.io.FileNotFoundException
13     {
14         tableDesMots = new Hashtable();
15         // initialisation du flux entrant :
16         fluxEntrant = new StreamTokenizer(new FileReader(new File(nomDeFichier)));
17         fluxEntrant.ordinaryChar('.');
18     }
19
20     void traitement() throws java.io.IOException
21     {
22         Integer tmp, un = new Integer(1);
23         // parcours du flux en entrée et traitement de chaque mot par initialisation
24         // ou incrémentation de son compteur d'occurrences :
25         while (fluxEntrant.nextToken() != StreamTokenizer.TT_EOF) {
26             if (fluxEntrant.ttype == StreamTokenizer.TT_WORD) {
27                 Object déjaEnregistré = tableDesMots.get(fluxEntrant.sval);
28                 // test pour vérifier si l'objet (le mot) existe déjà en tant que
29                 // clef dans la table de hachage :
30                 if (déjaEnregistré == null)
31                     // le compteur du mot est initialisé à 1 :
32                     tmp = un;
33                 else
34                     // le compteur du mot est incrémenté d'une unité :
35                     tmp = new Integer(Integer.parseInt(déjaEnregistré.toString()) + 1);
36                 // le couple mot/compteur est ré-introduit dans la table :
37                 tableDesMots.put(fluxEntrant.sval, tmp);
38             }
39         }
40     }
41
42     void affichage()
43     {
44         // énumération des éléments de la liste par le biais d'une Enumération :
45         Enumeration laListe = tableDesMots.keys();
46         while (laListe.hasMoreElements()) {
47             String clef = laListe.nextElement().toString();
48             System.out.println(clef + "(" + tableDesMots.get(clef) + " occurrences)");
49         }
50     }
51
52     public static void main(String[] arguments) throws java.io.IOException
53     {
54         // on passe en argument de la ligne de commande le nom du fichier à traiter :
55         DecompteDesMotsDUnTexte toto = new DecompteDesMotsDUnTexte(arguments[0]);
56         toto.traitement();
57         toto.affichage();
58     }
59 }
```

En conclusion, nous pouvons donc constater que la table de hachage peut être vue comme l'ultime moyen de gérer des associations clefs/valeurs composites et complexes avant d'utiliser un gestionnaire de bases de données.

Collections non indexées : ensembles

Un ensemble est une collection non indexée ni ordonnée qui ne contient qu'un seul exemplaire de chacun de ses éléments. Dire qu'un ensemble comporte des objets uniques signifie que la méthode *equals* de la classe *java.lang.Object* ne doit jamais renvoyer *true* pour deux objets du même ensemble.

```
1  import java.util.Iterator;
2  import java.util.Set;
3  import java.util.TreeSet;
4
5  class CribleEratosthene
6  {
7      Set leCrible;
8
9      CribleEratosthene(int n)
10     {
11         leCrible = new TreeSet();
12         for(int i=2; i<n; i++)
13             leCrible.add(new Integer(i));
14     }
15
16     void afficher()
17     {
18         Iterator i = leCrible.iterator();
19         while (i.hasNext())
20             System.out.println(i.next());
21     }
22
23     void éliminerLesMultiplesDe(int n)
24     {
25         Set tmp = new TreeSet();
26         Integer variant;
27
28         Iterator i = leCrible.iterator();
29         while (i.hasNext()) {
30             variant = (Integer) i.next();
31             if ((variant.intValue() % n == 0) && (variant.intValue() > n))
32                 tmp.add(variant);
```

```
33              }
34              leCrible .removeAll(tmp);
35          }
36
37          void éliminerLesNonPremiers()
38          {
39              for ( int i =0 ; i < leCrible . size () ; i ++) {
40                  Object [] tmp = leCrible . toArray ();
41                  éliminerLesMultiplesDe ((( Integer ) tmp[i ]). intValue ());
42              }
43          }
44
45          public static void main ( String [] arguments)
46          {
47              CribleEratosthene ce = new CribleEratosthene ( Integer . parseInt (arguments [0]));
48              ce. éliminerLesNonPremiers();
49              ce. affi cher ();
50          }
51      }
```

```
1       class CribleEratostheneRapide
2       {
3           public static void main ( String [] arguments)
4           {
5               int i , j , max = Integer . parseInt (arguments [0]);
6               int racineDeMax = (int )( Math. sqrt (max)+1);
7               boolean[] nombresPremiers = new boolean[max+1];
8
9               // Initialisation du tableau
10              for ( i =2 ; i <= max ; i++)
11                  nombresPremiers[i ] = true ;
12
13              // Test pour connaître les nombres premiers
14              for ( i =2 ; i <= racineDeMax ; i++)
15                  if ( nombresPremiers[i ] == true)
16                      for ( j =2 ; i*j <= max ; j++)
17                          nombresPremiers[i*j ] = false ;
18
19              // Affichage des résultats
20              for ( i =2 ; i <= max ; i++)
21                  if ( nombresPremiers[i ] == true)
22                      System.out . println ( i );
23          }
24      }
```

2.5.3 Arithmétique en précision arbitraire

Ne pas confondre le paquetage mathématique *java.math* (livré en standard avec le *JDK* depuis la version 1.1 du langage) avec la classe *Math* du paquetage *java.lang*.

Nous vous proposons d'admettre l'expression mathématique suivante du nombre π sous forme d'une somme infinie de termes et de vous reporter au site `http://www.mathsoft.com/asolve/plouffe/plouffe.html` pour plus d'information :

$$\pi = \sum_{i=0}^{\infty} \left(\frac{4}{8\times i+1} - \frac{2}{8\times i+4} - \frac{1}{8\times i+5} - \frac{1}{8\times i+6} \right) \times \frac{1}{16^i}$$

Les limites de la double précision

Étant donnée l'égalité de MM. Bailey, Borwein et Plouffe (celle-ci est plus couramment appelée formule BBP), notre premier réflexe pour calculer une valeur approchée de π, est de l'implémenter avec des flottants en double précision (des *double*, donc). C'est exactement ce que nous réalisons en quelques lignes de *Java*™ dans la méthode *main* de la classe `PiEnDouble` qui suit. Comme vous pouvez le constater, celle-ci n'est absolument pas optimisée dans la mesure où elle invoque, par exemple, la méthode *pow* de la classe *java.lang.Math* à chaque pas de calcul pour déterminer les puissances de 16 successives.

```
1       class PiEnDouble
2       {
3           public static void main ( String [] arguments)
4           {
5               double pi=0;
6               for ( int i =0 ; i<Integer . parseInt (arguments [0]) ; i ++) {
7                   pi += (4./(8* i +1) - 2./(8* i +4) - 1./(8* i +5) - 1./(8* i +6))/ Math.pow(16,i);
8                   System.out . println ("Valeur approchée de Pi à l' itération numéro " +
9                                         (i +1) + ": " + pi );
10              }
11          }
12      }
```

À l'exécution, nous obtenons la trace suivante qui confirme bien la rapidité de convergence de cette "formule miracle" puisqu'elle permet d'obtenir les 15 premières décimales de π en 11 itérations seulement :

```
# java PiEnDouble 12
Valeur approchée de Pi à l'itération numéro 1 : 3.1333333333333333
Valeur approchée de Pi à l'itération numéro 2 : 3.1414224664224664
Valeur approchée de Pi à l'itération numéro 3 : 3.1415873903465816
Valeur approchée de Pi à l'itération numéro 4 : 3.1415924575674357
Valeur approchée de Pi à l'itération numéro 5 : 3.1415926454603365
Valeur approchée de Pi à l'itération numéro 6 : 3.141592653228088
Valeur approchée de Pi à l'itération numéro 7 : 3.141592653572881
```

```
Valeur approchée de Pi à l'itération numéro 8 : 3.141592653588973
Valeur approchée de Pi à l'itération numéro 9 : 3.1415926535897523
Valeur approchée de Pi à l'itération numéro 10 : 3.1415926535897913
Valeur approchée de Pi à l'itération numéro 11 : 3.141592653589793
Valeur approchée de Pi à l'itération numéro 12 : 3.141592653589793
```

Mais, comme vous pouvez le constater avec cette méthode, nous ne pouvons pas obtenir (immédiatement du moins) la valeur de π avec plus de 15 décimales. Pour ce faire, il va être nécessaire de travailler en arithmétique flottante en précision arbitraire. C'est aussi ce que l'on appelle une arithmétique BIGNUM[13] qui nécessite d'écrire des bibliothèques de classes dédiées aux calculs et à la représentation des nombres avec une précision quelconque, arbitrairement choisie aussi grande que nécessaire.

Arithmétique BIGNUM et paquetage *java.math*

java.math $\Leftrightarrow$ deux classes, *BigInteger* et *BigDecimal*, dédiées respectivement aux calculs entier et flottant en précision arbitraire. Ces deux classes, qui sont immuables, dérivent directement de *java.lang.Number*. La classe *BigInteger* propose une arithmétique entière en précision arbitraire calquée sur le modèle de l'arithmétique entière standard avec les types primitifs. Elle offre, à ce titre, un certain nombre de méthodes telles que *add*, *divide*, *multiply*, *negate*... qui permettent respectivement d'additionner, diviser ou multiplier deux instances de *BigInteger* ou de calculer l'opposé d'un *BigInteger*.

La deuxième classe du paquetage, *BigDecimal*, propose un éventail de méthodes de calculs arithmétiques élémentaires (*add*, *divide*, *multiply*, *negate*...), de comparaison, de conversions... Cette classe offre en outre au développeur la possibilité de choisir l'arrondi parmi 8 modes d'arrondis différents[14] lors des opérations *divide* et *setScale* (réajustement du nombre de décimales). À titre d'exemple, nous vous proposons le code ci-après qui consiste à évaluer $1/3$ avec deux décimales selon deux modes d'arrondis différents :

```
1  import java.math.*;
2
3  class BigNum
4  {
5      public static void main ( String [] arguments)
6      {
7          BigDecimal un = new BigDecimal("1.00");
8          BigDecimal trois = new BigDecimal("3");
9          System.out.println (un.divide( trois ,BigDecimal.ROUND_DOWN));
10         System.out.println (un.divide( trois ,BigDecimal.ROUND_UP));
11     }
12 }
```

À l'exécution, il produit la trace qui suit :

```
0.33
0.34
```

► **Calcul des 2000 premières décimales de π avec la classe *BigDecimal*** en 3 minutes sur un serveur *SGI - Origin 2000* à 64 processeurs *Mips R10 000*, sous système d'exploitation *Irix 6.5.8*

```
1  import java.math.*;
2
3  final class PiEnBigDecimal
4  {
5      static final int précision = 4000;
6      static final int arrondi = BigDecimal.ROUND_DOWN;
7      static final BigDecimal moinsDeux = new BigDecimal(-2).setScale(précision);
8      static final BigDecimal moinsUn = new BigDecimal(-1).setScale(précision);
9      static final BigDecimal zéro = new BigDecimal(0).setScale( précision );
10     static final BigDecimal un = new BigDecimal(1).setScale( précision );
11     static final BigDecimal quatre = new BigDecimal(4).setScale( précision );
12     static final BigDecimal cinq = new BigDecimal(5).setScale( précision );
13     static final BigDecimal six = new BigDecimal(6).setScale( précision );
14     static final BigDecimal huit = new BigDecimal(8).setScale( précision );
15     static final BigDecimal seize = new BigDecimal(16).setScale( précision );
16     private static BigDecimal inversePuissance16=un, huitFoisI =zéro;
17
18     public static BigDecimal calculDUneHexadécimale(int i)
19     {
20         BigDecimal résultat = quatre.divide( huitFoisI .add(un), arrondi ).
21             add(moinsDeux.divide( huitFoisI .add(quatre ), arrondi )).
22             add(moinsUn.divide( huitFoisI .add(cinq ), arrondi )).
23             add(moinsUn.divide( huitFoisI .add(six ), arrondi )).
24             multiply( inversePuissance16 );
25         inversePuissance16 = inversePuissance16 . divide( seize , arrondi );
```

[13] Ce terme est une contraction de *"BIG NUMber"* qui signifie "grand nombre".

[14] Les 8 modes d'arrondis sont les suivants : *ROUND_CEILING*, *ROUND_DOWN*, *ROUND_FLOOR*, *ROUND_HALF_DOWN*, *ROUND_HALF_EVEN*, *ROUND_HALF_UP*, *ROUND_UNNECESSARY* (aucun arrondi), *ROUND_UP*. Nous vous invitons à consulter la documentation de la classe pour plus de précision.

```
26              huitFoisI = huitFoisI.add(huit);
27              return résultat;
28          }
29
30          public static void afficher(BigDecimal pi)
31          {
32              StringBuffer chaîne = new StringBuffer(pi.toString());
33              chaîne.insert(2,'\n');
34              for(int i=13; i<chaîne.length(); i+=11)
35                  chaîne.insert(i, ((i-57)%55 != 0) ? ' ' : '\n');
36              System.out.println(chaîne.substring(0,2202));
37          }
38
39          public static void main(String[] arguments)
40          {
41              BigDecimal pi=zéro;
42              for(int i=0; i<Integer.parseInt(arguments[0]); i++)
43                  pi = pi.add(calculDUneHexadécimale(i));
44              afficher(pi);
45          }
46      }
```

Chapitre 3

Les entrées-sorties et les mécanismes de sérialisation et de gestion des exceptions

Sommaire

3.1 *Java*™ et les exceptions

La gestion des erreurs par exceptions, telle qu'elle est faite en *Java*™, permet de distinguer et d'isoler la partie concernant le traitement des erreurs au sein d'un programme. Ce moyen permet de faire du contrôle d'erreur de façon élégante en s'affranchissant des fameux champs additionnels, appelés codes d'erreur, propres à certains autres langages de programmation plus "traditionnels".

```
1   try {
2       ... bloc d' instructions critiques ...
3   }
4   catch (UneClasseDException uneExceptionInterceptée) {
5       ... séquence-d-instructions-correspondant-à-un-type-d-erreur ...
6   }
7   catch (UneAutreClasseDException uneAutreExceptionInterceptée) {
8       ... séquence-d-instructions-correspondant-à-un-autre-type-d-erreur ...
9   }
10   [...]
11  finally {
12      ... séquence-d-instructions-à-exécuter-dans-tous-les-cas ...
13    // qu'une exception soit levée (et éventuellement interceptée) ou non.
14  }
```

▶ **Relayer une exception avec *throws*** La détection d'erreur est une opération et le traitement de l'erreur en est une autre. "Externaliser" le traitement de l'erreur et le confier à un gestionnaire d'exception dédié par *throws*.

```
1   [ modificateurDeVisibilité ] [ typeRenvoyé] maMéthode([listeDArguments])
2           throws UneClasseDException, UneAutreClasseDException...
3   {
4       ... séquence-d-instructions-de-la-méthode ...
5   }
```

propagation (on parle aussi de remontée) d'exception de méthode appelée à méthode appelante se fait tant que la pile d'exécution n'est pas vide. Si aucune méthode de la pile ne capture l'exception, celle-ci provoque la fin de l'exécution et la génération d'un message d'erreur.

► **Lever une exception avec *throw*** on lève une exception avec l'instruction *throw* du langage.

```
throw new UneClasseDException(UneListeDArguments);
```

3.1.1 Classification des exceptions en *Java*™

Une exception est un événement qui survient au cours de l'exécution d'un programme et qui interrompt le cours de son déroulement "normal". D'un point de vue plus technique, c'est un objet d'une classe qui hérite de la classe *java.lang.Throwable*. Cet objet est instancié lors d'un incident (on dit que l'exception correspondante est levée), le traitement courant est suspendu et l'exception remonte la pile d'exécution de méthode appelée à méthode appelante tant que la pile n'est pas vide. Si aucune méthode de la pile n'intercepte l'exception, celle-ci provoque la fin de l'exécution et la génération d'un message d'erreur.

Il existe en fait deux types d'exceptions : les premières sont dites "non contrôlées"[1]. Elles dérivent des classes *java.lang.RuntimeException* (incidents tels qu'une division entière par zéro, une activation de méthode sur une référence à *null*, un accès à un élément de tableau dont l'indice est hors des bornes…) ou *java.lang.Error* (erreurs graves de la machine virtuelle). Les secondes sont dites "contrôlées"[2] dans la mesure où le compilateur vérifie qu'elles sont bien interceptées ou relayées (c'est-à-dire spécifiées dans les signatures de méthodes) au cours de la phase de compilation. D'un point de vue pratique, les exceptions non contrôlées qui dérivent de *RuntimeException* ou *Error* peuvent être interceptées mais ne doivent pas être levées volontairement par un développeur d'application avec la clause *throw*, contrairement aux exceptions contrôlées. Nous vous invitons par ailleurs à ne pas étendre les classes *RuntimeException* ou *Error* car, en le faisant, vous provoquez la confusion en laissant croire que l'événement survenu est provoqué, non par l'applicatif, mais par la machine virtuelle et car vous privez un potentiel relecteur ou utilisateur de votre classe du bénéfice des clauses *throws* en ce qui concerne la documentation implicite de votre code.

Exemple d'exception non contrôlée dérivant de *Error* :

Elles ne sont ni ne doivent jamais être interceptées. À titre d'information, nous provoquons ci-après une récursion infinie, et donc un débordement de pile :

```
1  class RecursiviteInfinie
2  {
3      void méthodeRécursive()
4      {
5          méthodeRécursive();
6      }
7      public static void main (String [] arguments)
8      {
9          new RecursiviteInfinie (). méthodeRécursive();
10     }
11 }
```

```
Exception in thread "main" java.lang.StackOverfbwError
    at RecursiviteInfinie.méthodeRécursive(RecursiviteInfinie.java : 5)
    at RecursiviteInfinie.méthodeRécursive(RecursiviteInfinie.java : 5)
    ...
```

Il est très fortement déconseillé (instabilité du comportement de l'applicatif/*JVM*) d'intercepter l'exception *StackOverflowError* en encapsulant l'activation de la méthode `méthodeRécursive`, au sein du *main*, dans un *try…catch*, de la manière suivante :

```
1  try {
2      new RecursiviteInfinie (). méthodeRécursive();
3  }
4  catch (StackOverfbwError uneErreur) {}
```

Exemple d'exception non contrôlée dérivant de *RuntimeException* :

Quatre cas précis d'exceptions non contrôlées dérivant de *RuntimeException*. Dans la mesure où les spécifications du langage ne l'imposent pas, nous aurions cependant très bien pu nous passer des quatre *try…catch* dans l'exemple de la classe `ExceptionNonControleeDuRuntime` qui suit, et le compilateur n'aurait rien signalé d'anormal (ce qui ne serait pas le cas avec tout autre type d'exception) :

```
1  class ExceptionNonControleeDuRuntime
2  {
3      static void main (String [] arguments)
4      {
5          // commençons par lever et intercepter une exception de type
6          // IndexOutOfBoundsException :
```

[1] *Unchecked*.
[2] *Checked*.

```
7             char[] tableau = {'b','o','n','j','o','u','r'};
8             try {
9                 tableau [1980] = 'z';
10            }
11            catch (IndexOutOfBoundsException uneException) {
12                System.err.println ("exception de type :" + uneException);
13            }
14            // levons et interceptons ensuite une exception de type
15            // ArithmeticException :
16            int un=1,zéro=0;
17            try {
18                System.out.println (un/zéro);
19            }
20            catch (ArithmeticException uneException) {
21                System.err.println ("exception de type :" + uneException);
22            }
23            // levons et interceptons ensuite une exception de type
24            // NullPointerException :
25            Object unObjet = null;
26            try {
27                System.out.println (unObjet.toString ());
28            }
29            catch (NullPointerException uneException) {
30                System.err.println ("exception de type :" + uneException);
31            }
32            // levons et interceptons ensuite une exception de type
33            // NumberFormatException :
34            try {
35                int unEntier = Integer.parseInt ("12.3");
36            }
37            catch (NumberFormatException uneException) {
38                System.err.println ("exception de type :" + uneException);
39            }
40        }
41    }
```

```
exception de type : java.lang.ArrayIndexOutOfBoundsException
exception de type : java.lang.ArithmeticException : / by zero
exception de type : java.lang.NullPointerException
exception de type : java.lang.NumberFormatException : 12.3
```

Exemple d'exception contrôlée :

Dans la classe `ExceptionControlee`, nous présentons les deux types de traitements d'exceptions contrôlées (intercepter ou relayer) dans le cas précis d'opérations d'entrée-sortie, mais nous aurions très bien pu les mettre en œuvre avec d'autres types de traitements et d'autres exceptions contrôlées.

Le premier d'entre eux consiste à intercepter et traiter directement une exception de type *FileNotFoundException* en insérant l'instruction critique qui consiste à instancier un *FileReader* à partir d'un nom de fichier `nomDeFichier`, au sein d'un *try...catch*. Le second type de traitement consiste à déclarer que le constructeur `ExceptionControlee` et la méthode `setNomDeFichier` sont susceptibles de lever chacun une exception de type *IOException* (c'est le rôle du *throws IOException* placé au niveau de la signature du constructeur et de la méthode). Cette exception n'est pas interceptée immédiatement dans le corps des méthodes concernées mais transmise à la méthode appelante (il s'agit de la méthode *main* en l'occurrence) qui décide à son tour de l'intercepter ou de la relayer. Dans ce cas précis, la méthode *main* ne l'intercepte même pas et, en cas d'incident, l'exception provoque alors une interruption de l'exécution.

```
1   import java.io.*;
2
3   class ExceptionControlee
4   {
5       private BufferedReader stdin = null;
6       private Reader données = null;
7       private String nomDeFichier;
8
9       ExceptionControlee () throws IOException
10      {
11          stdin = new BufferedReader(new InputStreamReader(System.in));
12      }
13
14      public void setNomDeFichier() throws IOException
15      {
16          if (stdin != null)
17              nomDeFichier = stdin.readLine ();
18      }
19
20      public static void main (String [] arguments) throws IOException
21      {
22          ExceptionControlee monObjet = new ExceptionControlee();
23
24          while (monObjet.données == null) {
25              System.out.print ("Entrez un nom de fichier :");
26              monObjet.setNomDeFichier();
27              try {
28                  monObjet.données = new FileReader(monObjet.nomDeFichier);
29              }
30              catch (FileNotFoundException uneException) {
31                  System.out.println ("=> fichier inconnu.");
32              }
33          }
34          System.out.println ("Merci, à bientôt !");
35      }
36  }
```

```
# java ExceptionControlee
Entrez un nom de fichier : ExceptionControlee
=> fichier inconnu.
Entrez un nom de fichier : ExceptionControlee.java
Merci, à bientôt !
```

3.1.2 Enrichir la hiérarchie des classes d'exceptions

Pour enrichir la hiérarchie des classes d'exceptions, il suffit de créer une nouvelle classe dérivée de la classe *java.lang.Exception* :

```
1  class MonException extends Exception
2  {
3      MonException()
4      {
5      }
6      MonException(String uneChaîne)
7      {
8          super(uneChaîne);
9      }
10 }
```

Toute nouvelle exception créée de cette manière hérite ainsi des 2 constructeurs et des méthodes *fillInStackTrace*, *getLocalizedMessage*, *getMessage*, *printStackTrace* et *toString* de la classe *Throwable*.

Une fois cette nouvelle classe d'exception implémentée, il est possible de l'instancier (à l'aide de l'opérateur *new*) et de la lever (à l'aide de l'opérateur *throw*) comme nous le faisons dans le corps de la méthode `méthode11` de la classe `LeverMonException` ci-après :

```
1  class LeverMonException
2  {
3      LeverMonException() throws MonException
4      {
5          méthode1();
6      }
7
8      void méthode1() throws MonException
9      {
10         méthode11();
11     }
12
13     void méthode11() throws MonException
14     {
15         throw new MonException("bref descriptif ");
16     }
17
18     public static void main ( String [] arguments)
19     {
20         try{
21             new LeverMonException();
22         }
23         catch ( MonException exceptionInterceptée ) {
24             exceptionInterceptée . printStackTrace ();
25         }
26     }
27 }
```

```
# java LeverMonException
MonException : bref descriptif
    at LeverMonException.méthode11(LeverMonException.java :15, Compiled Code)
    at LeverMonException.méthode1(LeverMonException.java :10, Compiled Code)
    at LeverMonException.<init>(LeverMonException.java :5, Compiled Code)
    at LeverMonException.main(LeverMonException.java :21, Compiled Code)
```

3.2 *Java*™ et les IO

Une application peut avoir besoin de communiquer avec des périphériques tels que les terminaux, les disques, le réseau... elle le fait par le biais des flots de données (*streams*). Un *stream*, ou flot de données, est un canal de communication logique à sens unique doté d'un "écrivain qui l'alimente en écriture" à l'une de ses extrémités et d'un "lecteur qui le décharge en lecture" à l'autre extrémité.

première tendance avec la version 1.0 du langage et une seconde avec la version 1.1 ! Ces deux tendances cohabitent maintenant au sein du paquetage *java.io* et cette cohabitation est appelée à durer puisque la deuxième vague a enrichi la première de quelques fonctionnalités supplémentaires. La première permet de manipuler des flots de données de type octet (*byte*, sur 8 bits, ce qui correspond à l'ensemble restreint aux caractères ISO-Latin-1) et la seconde permet de manipuler des flots de données de type caractère Unicode (*char*, sur 16 bits). Dans le premier cas, tout ce qui a trait aux entrées (opérations de lecture dans un flot de données) dérive de la classe *java.io.InputStream* et

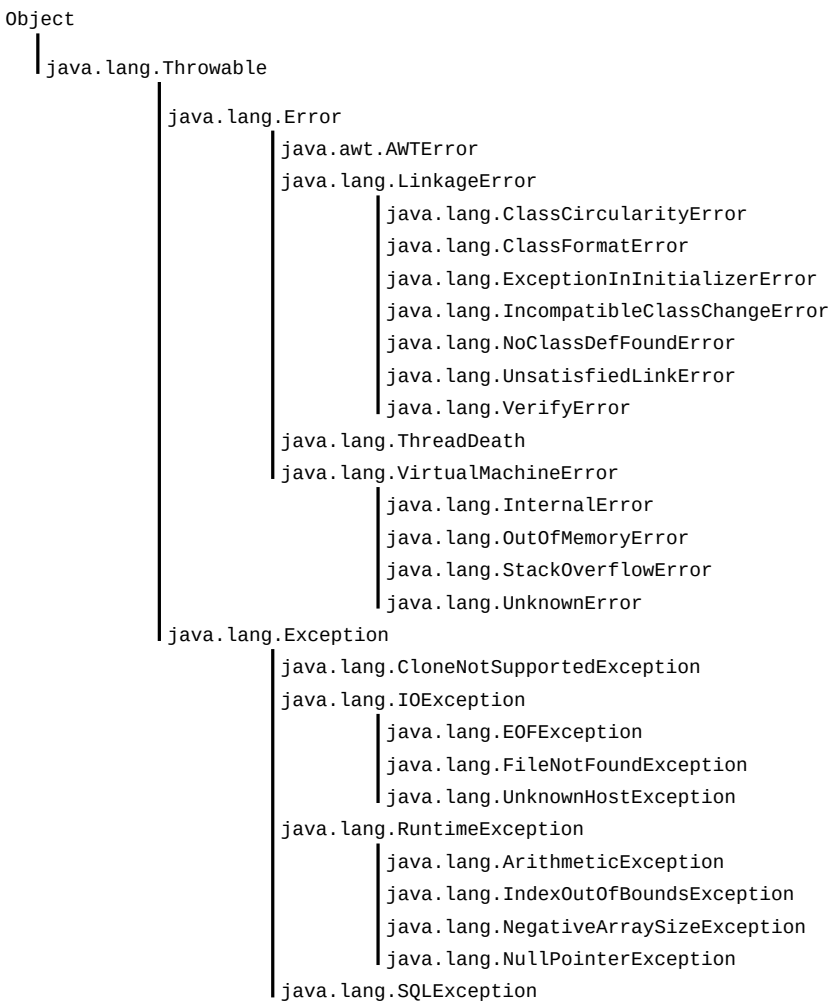


FIG. 3.1 – Version simplifiée de la hiérarchie des classes d'exceptions et d'erreurs.

tout ce qui a trait aux sorties (opérations d'écriture dans un flot de données) dérive de la classe *java.io.OutputStream*. Dans le second cas, la lecture dérive de la classe *java.io.Reader* et l'écriture dérive de la classe *java.io.Writer*. Les classes *java.io.InputStreamReader* et *java.io.OutputStreamWriter* sont des passerelles qui permettent respectivement de convertir un *InputStream* en *Reader* et un *OutputStream* en *Writer*.

3.3 flux standards (flux liés à la console et au clavier)

sont définis comme variables de classe (champs *static*) de *java.lang.System*. Alors que le flux d'entrée standard est un flux de type générique *InputStream*, les flux de sortie standard et de sortie d'erreurs sont de type *PrintStream* (une sous-classe de *FilterOutputStream* et *OutputStream*).

```
1    static InputStream System.in; // l'entrée standard
2    static PrintStream System.out; // la sortie standard
3    static PrintStream System.err; // la sortie d'erreurs standard
```

La classe *PrintStream*, qui étend la classe *FilterOutputStream*, permet d'écrire de façon "formatée" des caractères sur le flux de sortie, le tampon d'écriture est automatiquement vidé (c'est-à-dire que les caractères correspondants sont alors écrits réellement dans le flux) à chaque retour chariot ou à chaque appel à la méthode *println*. La classe suivante permet de lire une série d'entiers ou de flottants sur l'entrée standard (le clavier donc, lorsque celle-ci n'est pas redirigée) et de déterminer la nature de ce qui a été frappé au clavier :

```
1  import java.io.*;
2
3  class LireEntreeStandard
4  {
5      public static void main (String [] arguments) throws IOException
6      {
7          /*  l'entrée standard "System.in" est vue sous forme d'un
8           * BufferedReader. Si nous avions souhaité écrire ce qui suit à la
9           * mode Java 1.0, nous aurions utilisé un DataInputStream comme ici :
```

```
10              * DataInputStream entrée = new DataInputStream(System.in);
11              * Attention, la méthode "readLine" de la classe DataInputStream est
12              * appelée à disparaître dans les versions futures du langage, elle
13              * est en effet déclarée "deprecated"! */
14             BufferedReader entrée = new BufferedReader(new InputStreamReader(System.in));
15             /* la  sortie standard "System.out" est vue sous  forme d'un
16              *  PrintWriter. La valeur booléenne "true" en deuxième argument de
17              *  constructeur permet d'activer le mécanisme automatique de "flush"
18              *   (ou auto-vidage) sur le tampon d'écriture à chaque appel à la
19              *  méthode "println" ou à chaque tentative d'écriture d'un "\n" : */
20             PrintWriter  sortie = new PrintWriter(System.out, true);
21             /* la  sortie d'erreur "System.err" est vue sous  forme d'un
22              *  PrintWriter. La valeur booléenne "true" en deuxième argument de
23              * constructeur permet d'activer le mécanisme automatique de "flush"
24              *  (ou auto-vidage) sur le tampon d'écriture à chaque appel à la
25              * méthode "println" ou à chaque tentative d'écriture d'un "\n" : */
26             PrintWriter  erreur = new PrintWriter(System.err, true);
27
28             String  ligneCourante;
29             long unEntier;
30             double unFlottant;
31
32             sortie.println("Entrez un entier ou un flottant au clavier :");
33             while ((ligneCourante = entrée.readLine()) != null) {
34                 try {
35                     // tentative de conversion en entier :
36                     unEntier = Long.parseLong(ligneCourante);
37                     sortie.println("=> " + ligneCourante + " est un entier");
38                 }
39                 catch (NumberFormatException e) {
40                     try {
41                         // tentative de conversion en flottant :
42                         unFlottant = Double.parseDouble(ligneCourante);
43                         sortie.println("=> " + ligneCourante + " est un flottant");
44                     }
45                     catch (NumberFormatException ee) {
46                         erreur.println("=> ce n'est ni un entier ni un flottant !");
47                     }
48                 }
49             }
50             entrée.close();
51         }
52     }
```

Nous aurions tout aussi bien pu, pour nous en faciliter l'écriture, définir les flux `sortie` et `erreur` comme des alias respectifs de *System.out* et *System.err* (c'est-à-dire comme des flux de type *PrintStream*). À la place de :

```
1       PrintWriter  sortie = new PrintWriter(System.out, true);
2       // ...
3       PrintWriter  sortie = new PrintWriter(System.out, true);
```

nous aurions alors simplement écrit :

```
1       PrintStream  sortie = System.out;
2       // ...
3       PrintStream  erreur = System.err;
```

▶ **Redirection des flux standards :** La redirection des flux standards se fait à l'aide des trois méthodes statiques suivantes de la classe *java.lang.System* :

```
1       static void System.setIn(InputStream entrée) // redirection de l'entrée standard
2       static void System.setOut(PrintStream sortie) // redirection de la sortie standard
3       static void System.setErr(PrintStream erreur) // redirection de la sortie d'erreurs standard
```

3.3.1 Lecture et écriture de flux de caractères sur disque

Le système de fichiers vu par la classe *File*

Hormis dans certains cas d'applications ou applets circulant sur le réseau et aux droits restreints par le *SecurityManager*, une application *Java*™ peut accéder au système de fichiers de la machine hôte avec les droits de l'utilisateur qui l'a lancée. La classe *File* du paquetage *java.io* fournit un certain nombre de primitives de manipulation du système de fichiers courant.

```
1   import java.io.*;
2
3   class ManipulationClasseFile
4   {
5       static void tests(File f) throws IOException
6       {
7           String nomFichier = f.getCanonicalPath();
8
9           if (f.exists()) {
10              System.out.println(nomFichier + " existe bien");
11              if (f.canRead())
12                  System.out.println("Il est accessible en lecture");
13              if (f.canWrite())
14                  System.out.println("Il est accessible en écriture");
15              if (f.isFile()) {
16                  System.out.print("C'est un fichier");
17                  System.out.println(" de " + f.length() + " octets");
18              }
19              else if (f.isDirectory()) {
20                  System.out.println("C'est un répertoire qui contient");
21                  String [] contenuRépertoire = f.list();
22                  for (int i=0; i<contenuRépertoire.length ; i++)
23                      System.out.println("\t" + contenuRépertoire[i]);
24              }
25          }
26      }
27
28      public static void main(String [] arguments) throws IOException
29      {
30          for (int i=0; i<arguments.length ; i++)
```

```
31                  tests (new File(arguments[i ]));
32          }
33      }
```

Lire et écrire séquentiellement dans un fichier : les classes *FileReader* et *FileWriter*

Java™ ne se contente pas, dans son *API* standard, de nous livrer des classes de manipulation des flots standards et du système de fichiers. Il offre aussi, en natif, des classes qui étendent les flux de base en les liant à des fichiers. Ces classes, qui permettent un accès séquentiel en lecture ou écriture, s'appellent respectivement *FileInputStream* (version 1.0 du langage) et *FileReader* (version 1.1 du langage) pour la lecture, *FileOutputStream* (version 1.0 du langage) et *FileWriter* (version 1.1 du langage) pour l'écriture. Nous proposons ci-après un exemple d'utilisation des classes *FileReader* et *FileWriter*[3]. Il s'agit d'abord d'écrire 5 chaînes de 4 caractères (des couleurs) dans un fichier appelé `exemple.txt`. La méthode `lectureDeCaractères` se charge ensuite de relire le contenu de ce fichier et d'en afficher le contenu :

```
1   import java.io.*;
2
3   class LireEcrireDansUnFichier
4   {
5       // la constante DIM correspond à la taille (constante) des chaînes de
6       // caractères qui correspondent aux diverses couleurs "vert", "bleu",
7       // "gris", "brun" et "noir" que l'on souhaite stocker dans un fichier sur
8       // disque :
9       final static short DIM = 4;
10
11      static void écritureDeCaractères ( String nomFichier) throws IOException
12      {
13          // le tableau des couleurs :
14          String [] couleurs = {"vert","bleu","gris","brun","noir"};
15          // le flux en écriture sur disque :
16          Writer sortie = new FileWriter(new File(nomFichier));
17
18          for( int i =0 ; i<couleurs.length ; i++)
19              sortie.write ( couleurs[i ] );
20          sortie.close ();
21      }
22
23      static void lectureDeCaractères ( String nomFichier) throws IOException
24      {
25          // instanciation d'un tableau de "DIM" caractères :
26          char[] tableau = new char[DIM];
27          // le flux en lecture sur disque :
28          Reader entrée = new FileReader(new File(nomFichier));
29
30          while ( entrée.read( tableau ) != -1) {
31              // affichage des caractères lus sur la sortie standard :
32              System.out.println ( tableau );
33          }
34          entrée.close ();
35      }
36
37      public static void main ( String [] arguments) throws IOException
38      {
39          écritureDeCaractères ("exemple.txt");
40          lectureDeCaractères ("exemple.txt");
41      }
42  }
```

► **Attention :** N'oubliez pas que l'emploi des classes d'écriture sur un flux lié à un fichier, *FileOutputStream* et *FileWriter*, présuppose que vous ayez d'abord testé l'existence ou non du fichier sur lequel vous écrivez. En effet, toute nouvelle instanciation d'un tel objet écrase le contenu du fichier concerné.

► De l'importance des tampons d'entrée-sortie :

les classes *BufferedReader* et *BufferedWriter* Les classes *BufferedReader* et *BufferedInputStream* permettent de lire un flux avec un tampon de lecture, ce qui permet d'accélérer les entrées-sorties en réduisant le nombre d'appels aux primitives *read* (plusieurs octets ou caractères Unicodes sont en effet lus en une seule passe). Leurs homologues pour les opérations d'écriture sur flux sont les classes *BufferedWriter* et *BufferedOutputStream*. La classe `ComparaisonDesPerformances` que nous proposons à présent permet de comparer les temps respectifs mis pour lire le contenu d'un fichier avec ou sans *buffer* de lecture.

```
1   import java.io.*;
2
3   class ComparaisonDesPerformances
4   {
5       public static void main ( String [] arguments) throws FileNotFoundException,IOException
6       {
7           int c;
8           long nbCaractères1=0,nbCaractères2=0,nbCaractères3=0,nbCaractères4=0;
9           long date0,date1,date2,date3,date4;
10          String nomFichier = arguments[0];
11          date0 = System.currentTimeMillis (); // saisie de la date courante
12
13          // lecture non bufferisée par Reader (v 1.1)
14          Reader fluxEnEntrée1 = new FileReader(new File(nomFichier));
15          while ((c = fluxEnEntrée1.read ()) != -1) nbCaractères1++;
16          fluxEnEntrée1.close ();
17          date1 = System.currentTimeMillis ();
18
19          // lecture bufferisée par BufferedReader (v 1.1)
```

[3]Nous vous invitons à les préférer, à l'usage, à leurs homologues *FileInputStream* et *FileOutputStream* de la version 1.0 du langage.

```
20              BufferedReader fluxEnEntrée2 = new
21                  BufferedReader(new FileReader(new File(nomFichier )));
22              while (( c = fluxEnEntrée2.read ()) != -1) nbCaractères2++;
23              // String ligne; while ((ligne = fluxEnEntrée2.readLine()) != null) {}
24              fluxEnEntrée2. close ();
25              date2 = System. currentTimeMillis (); // saisie de la date courante
26
27              // lecture non bufferisée par InputStream (v 1.0)
28              InputStream fluxEnEntrée3 = new FileInputStream(new File(nomFichier));
29              while (( c = fluxEnEntrée3.read ()) != -1) nbCaractères3++;
30              fluxEnEntrée3. close ();
31              date3 = System. currentTimeMillis (); // saisie de la date courante
32
33              // lecture bufferisée par InputStream (v 1.0)
34              InputStream fluxEnEntrée4 = new
35                  BufferedInputStream(new FileInputStream(new File(nomFichier )));
36              while (( c = fluxEnEntrée4.read ()) != -1) nbCaractères4++;
37              fluxEnEntrée4. close ();
38              date4 = System. currentTimeMillis (); // saisie de la date courante
39
40              // affichage du verdict :
41              if (( nbCaractères1 == nbCaractères2) && (nbCaractères1 == nbCaractères3) &&
42                  ( nbCaractères1 == nbCaractères4 )) {
43                  System.out. println ("Pour lire " + nbCaractères1 + " caractères , il faut ..." );
44                  System.err . println ("par Reader non bufferisé :" + ( date1-date0) + " ms");
45                  System.err . println ("par Reader bufferisé :" + ( date2-date1) + " ms");
46                  System.err . println ("par InputStream non bufferisé :" + ( date3-date2) + " ms");
47                  System.err . println ("par InputStream bufferisé :" + ( date4-date3) + " ms");
48              }
49          }
50      }
```

```
Pour lire 1678373 caractères, il faut...
par Reader non bufferisé : 276732 ms
par Reader bufferisé : 1016 ms
par InputStream non bufferisé : 13982 ms
par InputStream bufferisé : 849 ms
```

```
1   import java . io .*;
2
3   class Grep
4   {
5       // cette classe reproduit, de façon très simplifiée, le comportement de la
6       // commande grep (avec l'option -n) du monde Unix.
7       public static void main ( String [] arguments) throws IOException
8       {
9           // le premier argument de la ligne de commande correspond à la
10          // séquence de caractères à rechercher dans le fichier dont le nom est
11          // passé en deuxième argument de la ligne de commande :
12          String ligne , chaîne = arguments[0], nomFichier = arguments[1];
13
14          Reader entrée = new FileReader(new File(nomFichier));
15          LineNumberReader fluxEnEntrée = new LineNumberReader(entrée);
16          while (( ligne = fluxEnEntrée . readLine ()) != null ) {
17              if ( ligne . indexOf( chaîne ) != -1)
18                  System.out. println ( fluxEnEntrée . getLineNumber() + ":" + ligne );
19          }
20          fluxEnEntrée . close ();
21          entrée . close ();
22      }
23  }
```

▶ **Note :** Comme le constructeur de la classe *LineNumberReader* accepte un flux d'entrée de type générique *Reader*, nous aurions très bien pu écrire cette classe de manière plus générique avec un flux d'entrée quelconque, c'est-à-dire que ce flux aurait pu ne pas être un *FileReader* mais un *InputStreamReader* par exemple... Ainsi, pour filtrer l'entrée standard, nous aurions remplacé dans le code précédent l'instruction :

```
Reader entrée = new FileReader(new File(nomFichier));
```

par :

```
Reader entrée = new InputStreamReader(System.in);
```

3.3.2 Lecture et écriture de flux binaires sur disque : les classes *DataInputStream* et *DataOutputStream*

DataInputStream et *DataOutputStream* sont des classes qui fournissent des filtres permettant de lire ou d'écrire sur des flux, des représentations binaires de chaînes de caractères ou des représentations binaires des types de base codés sur plus d'un octet, comme, par exemple *readLong*, *writeDouble*... Les données, lorsqu'elles sont sauvées sur un disque, sont stockées dans un format binaire *big Endian*[4] ; elles ne sont donc pas éditables avec un éditeur ASCII traditionnel. À titre illustratif, nous vous proposons de compiler et d'exécuter l'exemple ci-après et de chercher à éditer `exemple.dat` avec un éditeur qui ne soit pas un éditeur de fichiers binaires :

[4] Pour vous en assurer, nous vous invitons, par exemple, à éditer le fichier de résultats avec l'utilitaire `od` muni de l'option `-t fD`, sur une plate-forme Unix ayant une architecture *big Endian*.

```
1  import java.io.*;
2
3  class FluxBinaires
4  {
5      static void écritureBinaire (String nomFichier) throws IOException
6      {
7          double[] valeur = {3.14159,6.55957,2.71828};
8          // ouverture du flux de données en écriture :
9          DataOutputStream sortie = new
10             DataOutputStream(new FileOutputStream(nomFichier));
11         for(int i=0; i<valeur.length ; i++)
12             sortie.writeDouble(valeur[i]);
13         sortie.close();
14     }
15
16     static void lectureBinaire (String nomFichier) throws IOException
17     {
18         // ouverture du flux de données en lecture :
19         DataInputStream entrée = new
20             DataInputStream(new FileInputStream(nomFichier));
21         try {
22             while (true) {
23                 System.out.println(entrée.readDouble());
24             }
25         }
26         catch (EOFException e) {
27             // l'exception instance de la classe EOFException a été
28             // levée suite à une tentative de lecture sur un fin de
29             // fichier.
30         }
31         entrée.close();
32     }
33
34     public static void main (String [] arguments) throws IOException
35     {
36         écritureBinaire ("exemple.dat");
37         lectureBinaire ("exemple.dat");
38     }
39 }
```

► **Note :** Pour optimiser les opérations d'entrée-sortie, nous aurions intérêt à manipuler des flux avec tampons d'entrée-sortie. Pour ce faire, il faut déclarer les flux *entrée* et *sortie* comme suit :

```
1  DataOutputStream sortie = new DataOutputStream(new
2      BufferedOutputStream(new FileOutputStream(nomFichier)));
3  // ...
4  DataInputStream entrée = new DataInputStream(new
5      BufferedInputStream(new FileInputStream(nomFichier)));
```

Les mécanismes de sérialisation et désérialisation que nous allons présenter à présent constituent d'excellents moyens pour gérer la persistance de données composites sous forme binaire.

3.3.3 Sérialisation et désérialisation : les classes *ObjectInputStream* et *ObjectOutputStream*

La sérialisation est un procédé apparu depuis la version 1.1 du langage, qui permet de stocker dans un fichier ou de transporter sur le réseau un objet ayant une structure complexe. Elle peut être vue comme un "pliage des paramètres d'une structure de données" dans un but de transport ou de stockage. Pour "sérialiser", il faut instancier la classe *ObjectOutputStream*. Cette opération consiste à ouvrir un flux en écriture, flux qui possède la capacité de convertir en une suite d'octets l'objet sur lequel s'applique la méthode d'écriture.

- tous les types primitifs du langage et la plupart des objets instances de classes de l'*API* standard (comme *String*, *Vector*, *Properties*, *Hashtable*...) sont sérialisables et désérialisables par simple appel aux méthodes *writeObject* et *readObject* ;
- tout objet dont les variables d'instances sont sérialisables peut être lui même sérialisé, à condition que la classe qu'il instancie implémente l'interface *java.io.Serializable*. Cette interface ne définit aucune méthode et constitue juste un marqueur à l'attention de la machine virtuelle ;
- la sérialisation (comme la désérialisation) concerne uniquement la sauvegarde (et la restauration) des données relatives aux objets eux-mêmes. Les variables de classes (déclarées *static*) ne sont pas concernées par le mécanisme standard de sérialisation mis en œuvre par l'implémentation de la classe *Serializable*. Par ailleurs, si une variable d'instance est déclarée à l'aide du modificateur *transient*, alors elle est considérée comme "éphémère" ou "transitoire" et, à ce titre, elle n'est pas concernée par la gestion persistante que constituent les mécanismes de sérialisation et désérialisation ;
- pour mettre en place des mécanismes de sérialisation plus spécifiques, vous pouvez surcharger les méthodes privées *writeObject* et *readObject* (veillez à le faire de façon "symétrique"). Vous pouvez même forcer vos classes à implémenter l'interface *Externalizable*, ce qui vous oblige à écrire la totalité du code de lecture et d'écriture sur flux.

► **Gestion d'un fond bibliothécaire par arbre binaire de recherche** la sérialisation et la désérialisation offrent des mécanismes de gestion persistante des données complexes (l'arbre a été chargé depuis le disque et réinstancié "à la volée") :

```
1  import java.io.*;
2
3  class BibliothequeEnArbreBinaire implements Serializable
```

```
{
    String auteurs, titre ;
    // les deux pointeurs vers les livres "fils" dans l'arbre binaire
    // de recherche :
    BibliothequeEnArbreBinaire filsGauche, filsDroit ;

    // les constructeurs :
    BibliothequeEnArbreBinaire()
    {
        auteurs = null;
        titre = null;
        filsGauche = null;
        filsDroit = null;
    }

    BibliothequeEnArbreBinaire( String auteurs, String titre )
    {
        this(); // appel au constructeur par défaut
        this.auteurs = auteurs;
        this.titre = titre ;
    }

    void insérer ( String auteurs, String titre )
    {
        if ( this.auteurs == null ) {
            // l'arbre est vide, on l'initialise avec le premier titre :
            this.auteurs = auteurs;
            this.titre = titre ;
        }
        else if ( this.auteurs.compareTo(auteurs) >= 0) {
            /* comme l'auteur à insérer est placé avant l'auteur courant dans
             * la relation d'ordre lexicographique appliquée aux noms
             * d'auteurs, on descend dans le sous-arbre gauche : */
            if ( filsGauche == null )
                // si l'on est sur une feuille on procède à l'insertion :
                filsGauche = new BibliothequeEnArbreBinaire( auteurs, titre );
            else
                filsGauche.insérer ( auteurs, titre );
        }
        else {
            /* dans ce cas, l'auteur à insérer est placé après l'auteur
             * courant dans la relation d'ordre lexicographique appliquée aux
             * noms d'auteurs, on descend donc dans le sous-arbre droit : */
            if ( filsDroit == null )
                // si l'on est sur une feuille on procède à l'insertion :
                filsDroit = new BibliothequeEnArbreBinaire( auteurs, titre );
            else
                filsDroit.insérer ( auteurs, titre );
        }
    }

    void afficher ()
    {
        /* pour afficher le contenu de l'arbre binaire de recherche en
         * respectant l'ordre lexicographique, on le parcourt de manière
         * récursive en commençant par descendre dans la portion gauche de
         * l'arbre, puis en continuant par le contenu du noeud lui-même, avant
         * de terminer par la portion droite de l'arbre : */
        if ( filsGauche != null ) filsGauche.afficher ();
        System.out.println ("AUTEUR(S) : " + auteurs + " -- TITRE : " + titre );
        if ( filsDroit != null ) filsDroit.afficher ();
    }

    void stocker( String nomDeFichier) throws IOException
    {
        // cette méthode met en oeuvre le mécanisme de sérialisation :
        ObjectOutputStream sortie =
            new ObjectOutputStream(new FileOutputStream(nomDeFichier));
        sortie.writeObject( this );
        sortie.flush(); // on force l'écriture effective sur disque
        sortie.close();
    }

    static BibliothequeEnArbreBinaire charger( String nomDeFichier)
        throws IOException,ClassNotFoundException
    {
        // cette méthode met en oeuvre le mécanisme de désérialisation :
        BibliothequeEnArbreBinaire tmp;
        ObjectInputStream entrée =
            new ObjectInputStream(new FileInputStream(nomDeFichier));
        tmp = ( BibliothequeEnArbreBinaire ) entrée.readObject();
        entrée.close();
        return tmp;
    }

    void enrichirLaBibliothèque () throws IOException
    {
        // définissons le tampon de lecture de l'entrée standard :
        BufferedReader stdin = new BufferedReader(new InputStreamReader(System.in));
        boolean continuer = true;
        String auteursTmp,titreTmp;

        do {
            System.out.print ("Nom(s) d'auteur(s) ( ou retour chariot pour terminer ) : " );
            auteursTmp = stdin.readLine();
            System.out.print (" Titre ( ou retour chariot pour terminer ) : " );
            titreTmp = stdin.readLine();
            // tant que les chaînes de caractères correspondant à auteursTmp
            // et titreTmp ne sont pas vides, on continue :
            if (! auteursTmp.equalsIgnoreCase("") && ! titreTmp.equalsIgnoreCase(""))
                insérer (auteursTmp,titreTmp);
            else continuer = false ;
        } while ( continuer );
    }

    public static void main ( String [] arguments)
        throws IOException,ClassNotFoundException
    {
        BibliothequeEnArbreBinaire maBNF;
        // le premier et unique argument de la ligne de commande correspond au
        // nom du fichier de la bibliothèque à enrichir :
        String nomDeFichier = arguments[0];
        File leFichier = new File(nomDeFichier);

        if ( leFichier.exists())
```

```
119             // le fichier existe, on procède à la désérialisation :
120             maBNF = charger(nomDeFichier);
121         else
122             maBNF = new BibliothequeEnArbreBinaire();
123
124         // on enrichit la bibliothèque :
125         maBNF.enrichirLaBibliothèque();
126
127         // on stocke l'état de la bibliothèque après affichage :
128         maBNF.afficher();
129         maBNF.stocker(nomDeFichier);
130     }
131 }
```

► **Stockage/déstockage de données composites dans le cas d'une UI** Autre exemple de sérialisation mise en œuvre dans le cas du stockage/déstockage de données composites en environnement graphique :

```
1   import java.awt.*;
2   import java.awt.geom.*;
3   import java.awt.event.*;
4
5   import javax.swing.*;
6   import javax.swing.event.*;
7
8   import java.util.*;
9   import java.io.*;                                                   // pour la sérialisation
10
11  public class GrapheReactif implements Serializable                 // pour la sérialisation
12  {
13      private Vector tableauTâches = null;
14      private Vector tableauEntités = null;
15      // une donnée "transient" n'est pas concernée par la (dé-)sérialisation...
16      transient private JFrame fenêtre = new JFrame("Tâches et liens");// pour la sérialisation
17
18      void charger()
19      {
20          try {
21              ObjectInputStream entrée =
22                  new ObjectInputStream(new FileInputStream("etat.ser"));
23              GrapheReactif tmp = (GrapheReactif) entrée.readObject();
24              entrée.close();
25
26              tableauTâches = tmp.tableauTâches;
27              tableauEntités = tmp.tableauEntités;
28
29              System.out.println("Désérialisation réussie !");
30          }
31          catch (IOException uneException) {
32              tableauTâches = new Vector();
33              tableauEntités = new Vector();
34              // initialisation par défaut :
35              Tache t1 = new Tache("1",new Point(50,50)), t2 = new Tache("2",new Point(250,100));
36              tableauTâches.addElement(t1);
37              tableauTâches.addElement(t2);
38              tableauEntités.addElement(t1);
39              tableauEntités.addElement(t2);
40              tableauEntités.addElement(new Lien("1-2",t1,t2));
41
42              System.err.println(uneException);
43          }
44          catch (ClassNotFoundException uneException) {
45              System.err.println(uneException);
46          }
47      }
48
49      void décharger()
50      {
51          try {
52              ObjectOutputStream sortie =
53                  new ObjectOutputStream(new FileOutputStream("etat.ser"));
54              sortie.writeObject(this);
55              sortie.flush();
56              sortie.close();
57              System.out.println("Sérialisation réussie !");
58          }
59          catch (IOException uneException) {
60              System.err.println(uneException);
61          }
62      }
63
64      GrapheReactif()
65      {
66          // processus de désérialisation : chargement de la configuration...
67          charger();
68
69          fenêtre.addWindowListener(new WindowAdapter() {
70                  public void windowClosing(WindowEvent événement)
71                  {
72                      // processus de sérialisation : enregistrement de la configuration...
73                      décharger();
74                      fenêtre.setVisible(false);
75                      fenêtre.dispose();
76                      System.exit(0);
77                  }
78              });
79
80          fenêtre.setContentPane(new ArdoiseGraphique(tableauEntités,tableauTâches));
81          fenêtre.pack();
82          fenêtre.setVisible(true);
83      }
84
85      public static void main(String[] arguments)
86      {
87          new GrapheReactif();
88      }
89  }
90
91  /* ****************************************** */
92  interface EntiteTracable extends Serializable              // pour la sérialisation
93  {
94      final static int RAYON = 10;
95      public void tracerEntité(Graphics2D g2);
```

```
96  }
97  /* ********************************************** */
98  class Lien implements EntiteTracable
99  {
100     private Tache a = null , b = null;
101     private String étiquette = null;
102
103     Lien( String étiquette , Tache a , Tache b)
104     {
105         this . étiquette = étiquette ;
106         this .a = a;
107         this .b = b;
108     }
109     public void tracerEntité (Graphics2D g2)
110     {
111         g2.drawLine(a.getX () , a.getY () , b.getX () , b.getY ());
112         g2.drawString( étiquette ,
113                         (a.getX () + b.getX ())/2 , ( a.getY () + b.getY ())/2);
114     }
115 }
116 /* ********************************************** */
117 class Tache implements EntiteTracable
118 {
119     private String étiquette = null;
120     private Point coordonnées = null;
121     // Note : java.io.NotSerializableException: java.awt.geom.Ellipse2D$Double
122     transient private Ellipse2D.Double cadre = new Ellipse2D.Double();
123
124     Tache( String étiquette , Point coordonnées)
125     {
126         this . étiquette = étiquette ;
127         this .coordonnées = coordonnées;
128     }
129
130     public void tracerEntité (Graphics2D g2)
131     {
132         setCadre ();
133         g2. fill (cadre );
134         g2.drawString( étiquette , coordonnées.x + RAYON , coordonnées.y + RAYON);
135     }
136     public void setCadre ()
137     {
138         if ( cadre == null ) // pour la sérialisation
139             cadre = new Ellipse2D.Double();
140         cadre .setFrame(coordonnées.x - RAYON/2 , coordonnées.y - RAYON/2 , RAYON , RAYON);
141     }
142     public void setCoordonnées(Point coordonnées)
143     {
144         this .coordonnées = coordonnées;
145         setCadre ();
146     }
147     public String getEtiquette ()
148     {
149         return étiquette ;
150     }
151     public int getX()
152     {
153         return coordonnées.x;
154     }
155     public int getY()
156     {
157         return coordonnées.y;
158     }
159     public Ellipse2D getCadre()
160     {
161         return cadre;
162     }
163 }
164 /* ********************************************** */
165 class AdaptateurDEvenements extends MouseInputAdapter
166 {
167     private Vector tableauTâches = null;
168     private ArdoiseGraphique ardoise = null;
169     private Tache tâcheDeRéférence = null;
170
171     AdaptateurDEvenements(Vector tableauTâches, ArdoiseGraphique ardoise )
172     {
173         this . tableauTâches = tableauTâches;
174         this . ardoise = ardoise ;
175     }
176     public void mousePressed(MouseEvent événement)
177     {
178         Enumeration lesTâches = tableauTâches . elements ();
179
180         while ( lesTâches . hasMoreElements()) {
181             Tache t = ( Tache ) lesTâches . nextElement ();
182             if ( t . getCadre (). contains ( événement. getPoint ()))
183                 tâcheDeRéférence = t ;
184         }
185     }
186
187     public void mouseReleased(MouseEvent événement)
188     {
189         if ( tâcheDeRéférence != null ) {
190             tâcheDeRéférence.setCoordonnées(événement.getPoint ());
191             ardoise . repaint ();
192         }
193         tâcheDeRéférence = null ;
194     }
195
196     public void mouseDragged(MouseEvent événement) {
197         if ( tâcheDeRéférence != null ) {
198             tâcheDeRéférence.setCoordonnées(événement.getPoint ());
199             ardoise . repaint ();
200         }
201     }
202 }
203 /* ********************************************** */
204 class ArdoiseGraphique extends JPanel
205 {
206     private Vector tableauEntités = null;
207     private Vector tableauTâches = null;
208
209     ArdoiseGraphique(Vector tableauEntités , Vector tableauTâches)
210     {
```

```
211             this . tableauEntités  =  tableauEntités ;
212             this . tableauTâches  =  tableauTâches ;
213              setPreferredSize  (new Dimension(400,200));
214             setBackground(Color . white );
215             setFont (new Font("Monospaced",Font.BOLD,16));
216
217             AdaptateurDEvenements gestionSouris = new AdaptateurDEvenements(tableauTâches,this );
218             addMouseListener( gestionSouris );
219             addMouseMotionListener(gestionSouris );
220              repaint  ();                                    // invocation de paintComponent
221         }
222
223         public void paintComponent(Graphics g)
224         {
225             super.paintComponent(g );                       // commençons par effacer "l'ardoise"...
226             Graphics2D g2 = (Graphics2D) g;
227             Enumeration  lesEntités  =  tableauEntités . elements ();
228             while  ( lesEntités . hasMoreElements())
229                  (( EntiteTracable )  lesEntités . nextElement ()).  tracerEntité (g2);
230         }
231     }
```

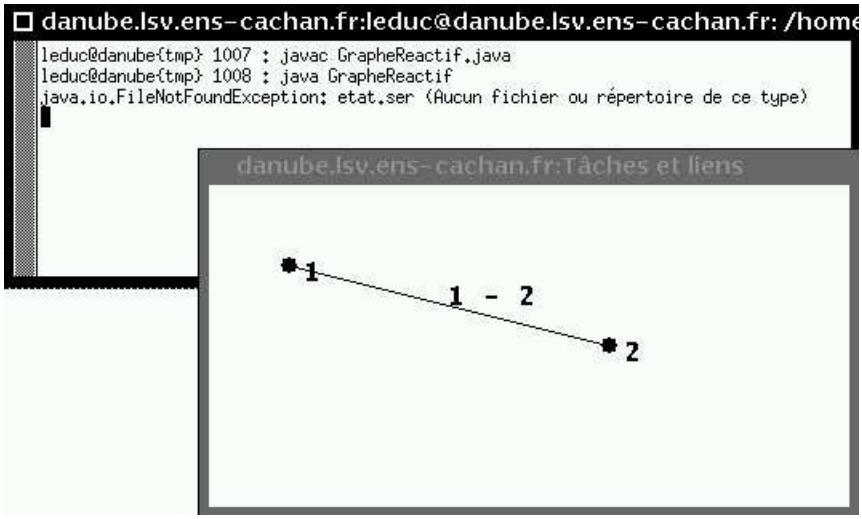


FIG. 3.2 – Première étape :

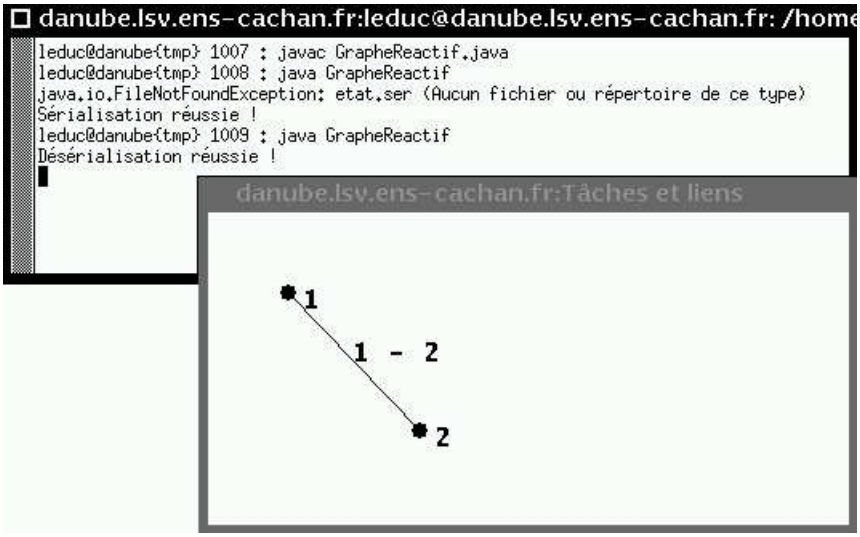


FIG. 3.3 – Deuxième étape :

3.3.4 Les fichiers à accès direct : la classe *RandomAccessFile*

La classe *RandomAccessFile* du paquetage *java.io*, qui existe depuis la version 1.0 du langage, permet de lire et écrire des données à n'importe quel endroit dans un fichier. C'est ce qui la différencie des classes *InputStream* (ou

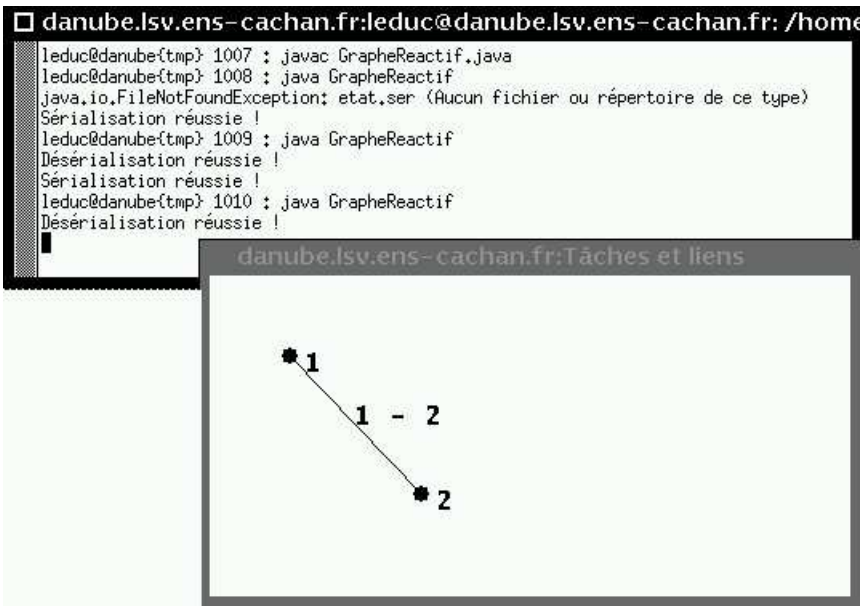


FIG. 3.4 – Troisième étape :

Reader) et *OutputStream* (ou *Writer*) qui permettent, elles, d'accéder soit en lecture, soit en écriture, mais toujours séquentiellement, aux données d'un fichier. En effet, elle offre, en plus des fonctionnalités de lecture et d'écriture classiques (c'est-à-dire celles des classes *DataOutputStream* et *DataInputStream*), un mécanisme de pointeur de fichier et des méthodes comme *seek* et *getFilePointer* qui permettent de se déplacer au sein du fichier et de localiser la position du pointeur dans le fichier.

```
1   import java.io.*;
2
3   class AccesDirect
4   {
5       final static int MAX = 100;
6       static void écritureDirecte (String nomFichier) throws IOException
7       {
8           // accès en écriture (mode "rw") au fichier de nom "nomFichier" :
9           RandomAccessFile monFichier = new RandomAccessFile(nomFichier,"rw");
10          for (int i =1 ; i<MAX ; i++) {
11              monFichier. writeInt (i );
12              if ((i & 1) == 0)
13                  monFichier.writeBoolean(true );  // i est un entier pair
14              else
15                  monFichier.writeBoolean( false ); // i est un entier impair
16          }
17          monFichier.close ();
18      }
19
20      static void lectureDirecte (String nomFichier) throws IOException
21      {
22          // accès en lecture (mode "r") au fichier de nom "nomFichier" :
23          RandomAccessFile monFichier = new RandomAccessFile(nomFichier,"r");
24          // accédons au 17ème enregistrement du fichier comportant 100
25          // couples entier/parité associée pour déterminer la parité de
26          // l'entier 17 :
27          monFichier.seek ( 16 * (4+1) );
28          System.out. println (monFichier. readInt () + " " + monFichier.readBoolean ());
29          // pour obtenir la parité de l'entier 78, à partir du début du
30          // fichier, nous devons nous déplacer de 77 * (4+1) octets :
31          monFichier.seek ( 77 * (4+1) );
32          System.out. println (monFichier. readInt () + " " + monFichier.readBoolean ());
33          monFichier.close ();
34      }
35
36      public static void main (String [] arguments) throws IOException
37      {
38          écritureDirecte ("exemple.dat");
39          lectureDirecte ("exemple.dat");
40      }
41  }
```

3.3.5 Un analyseur lexical : la classe *StreamTokenizer*

```
1   import java.io.*;
2
3   public class Wc
4   {
5       static long nombreL = 0;
6       static long nombreW = 0;
7       static long nombreC = 0;
8       static boolean compterL = true;
9       static boolean compterW = true;
10      static boolean compterC = true;
11
12      public static void usage ()
```

```
13      {
14          final String USAGE =
15              "Usage : java Wc [-lwc] (nom de fichier(s) | -)\n"
16              + "\t-l : affiche le nombre total de lignes \n"
17              + "\t-w : affiche le nombre total de mots\n"
18              + "\t-c : affiche le nombre total de carateres \n"
19              + "\t(Sans nom de fichier ou si '-' n'est pas specifie ,\n"
20              + "\tc'est l'entree standard qui est lue par defaut !)\n";
21          System.err.println(USAGE);
22          System.exit(1);
23      }
24
25      protected static void wcAffichage(long L,long W,long C,String nomDeFichier)
26      {
27          System.out.println((compterL ? "\t" + L : "")
28                             + (compterW ? "\t" + W : "")
29                             + (compterC ? "\t" + C : "")
30                             + ((nomDeFichier != null) ? " " + nomDeFichier : " total")
31                             );
32      }
33
34      protected static void wcAuxiliaire(Reader fr, String nomDeFichier) throws IOException
35      {
36          long nombreAuxiliaireL = 0;
37          long nombreAuxiliaireW = 0;
38          long nombreAuxiliaireC = 0;
39
40          // calcul du nombre de caractères dans le flot d'entrée
41          if (nomDeFichier == null) {
42              // lecture de l'entrée standard
43              nombreAuxiliaireC = System.in.available();
44          }
45          else {
46              // lecture du fichier de données (si possible)
47              File f = new File(nomDeFichier);
48              if (f.canRead())
49                  nombreAuxiliaireC = f.length();
50          }
51          // calcul du nombre de lignes et de mots dans le flot d'entrée
52          // par le biais d'un analyseur lexical :
53          StreamTokenizer analyseurSimple = new StreamTokenizer(fr);
54          // remise à zéro des tables d'analyse lexicale :
55          // tous les caractères sont identiquement traités
56          analyseurSimple.resetSyntax();
57          // tous les caractères de 0 à 255 sont traités comme des parties de mots
58          analyseurSimple.wordChars(0,255);
59          // 5 types de délimiteurs d'unités lexicales
60          analyseurSimple.whitespaceChars('\b','\b');
61          analyseurSimple.whitespaceChars('\r','\r');
62          analyseurSimple.whitespaceChars('\t','\t');
63          analyseurSimple.whitespaceChars('\n','\n');
64          analyseurSimple.whitespaceChars(' ',' ');
65          // il n'y a pas de conversion automatique en minuscule
66          // analyseurSimple.lowerCaseMode(false);
67          while (analyseurSimple.nextToken() != StreamTokenizer.TT_EOF)
68              nombreAuxiliaireW++;
69          nombreAuxiliaireL = analyseurSimple.lineno() - 1;
70
71          // incrémentation des compteurs généraux :
72          nombreL += nombreAuxiliaireL;
73          nombreW += nombreAuxiliaireW;
74          nombreC += nombreAuxiliaireC;
75          // affichage des compteurs propres au fichier courant :
76          wcAffichage(nombreAuxiliaireL,nombreAuxiliaireW,nombreAuxiliaireC,nomDeFichier);
77      }
78
79      protected static void wcParFichier(final String nomDeFichier) throws IOException
80      {
81          FileReader fr = null;
82          try {
83              // le flux d'entrée (qu'il soit de type flux sur un
84              // fichier ou flux sur l'entrée standard) est lu avec un
85              // tampon d'entrée dans un soucis d'efficacité :
86              if ("-".equals(nomDeFichier))
87                  wcAuxiliaire(new BufferedReader(new InputStreamReader(System.in)), null);
88              else {
89                  fr = new FileReader(nomDeFichier);
90                  wcAuxiliaire(new BufferedReader(fr), nomDeFichier);
91                  if (fr != null) fr.close();
92              }
93          }
94          catch (FileNotFoundException uneException) {
95              System.err.println("Problème : " + uneException);
96          }
97          catch (IOException uneException) {
98              System.err.println("Problème : " + uneException);
99              if (fr != null) fr.close();
100         }
101     }
102
103     public static void analyseOptions(String lesOptions)
104     {
105         compterL = false;
106         compterW = false;
107         compterC = false;
108         if (lesOptions.indexOf('l') > -1) compterL=true;
109         if (lesOptions.indexOf('w') > -1) compterW=true;
110         if (lesOptions.indexOf('c') > -1) compterC=true;
111     }
112
113     public static void main(String[] arguments) throws IOException
114     {
115         int numéroDuPremierFichier = 0;
116
117         if (arguments.length == 0)
118             usage();
119         else {
120             if ((arguments[0].length() > 1) && arguments[0].startsWith("-")) {
121                 analyseOptions(arguments[0]);
122                 numéroDuPremierFichier = 1;
123             }
124             for (int i=numéroDuPremierFichier; i<arguments.length; i++)
125                 wcParFichier(arguments[i]);
126             if (arguments.length > 1+numéroDuPremierFichier)
127                 wcAffichage(nombreL,nombreW,nombreC,null);
```

```
128             }
129         }
130     }
```

3.3.6 Chiffrement/déchiffrement lors des opérations d'IO

procéder par dérivation des classes abstraites *java.io.FilterReader* et *java.io.FilterWriter* et redéfinition des méthodes respectives *read* et *write*.

```
1   import java.io.*;
2
3   public class ChiffrerDechiffrer
4   {
5       public static void main(String [] arguments) throws IOException
6       {
7           new Chiffrer(arguments[0], arguments[0]+"~").encoderTexte();
8           new Dechiffrer(arguments[0]+"~", arguments[0]+"~~").décoderTexte();
9       }
10  }
11
12  class Chiffrer
13  {
14      Reader src;
15      Chiffrement dest;
16
17      Chiffrer(String src, String dest) throws IOException
18      {
19          this.src = new BufferedReader(new FileReader(src));
20          this.dest = new Chiffrement(new FileWriter(dest));
21      }
22
23      void encoderTexte() throws IOException
24      {
25          int tmp;
26
27          while ((tmp = src.read()) != -1) {
28              dest.write(tmp);
29          }
30          src.close();
31          dest.close();
32      }
33  }
34
35  class Dechiffrer
36  {
37      Dechiffrement src;
38      Writer dest;
39
40      Dechiffrer(String src, String dest) throws IOException
41      {
42          this.src = new Dechiffrement(new FileReader(src));
43          this.dest = new BufferedWriter(new FileWriter(dest));
44      }
45
46      void décoderTexte() throws IOException
47      {
48          int tmp;
49
50          while ((tmp = src.read()) != -1) {
51              dest.write(tmp);
52          }
53          src.close();
54          dest.close();
55      }
56  }
57
58  class Chiffrement extends FilterWriter
59  {
60      final static char alphabetDécaléMin[] =
61          "fghijklmnopqrstuvwxyzabcde".toCharArray();
62      final static char alphabetDécaléMaj[] =
63          "FGHIJKLMNOPQRSTUVWXYZABCDE".toCharArray();
64
65      Chiffrement(Writer sortie)
66      {
67          super(sortie);
68      }
69
70      public void write(int caractère) throws IOException
71      {
72          if ((caractère >= 'a') && (caractère <= 'z'))
73              caractère = alphabetDécaléMin[caractère - 'a'];
74          else if ((caractère >= 'A') && (caractère <= 'Z'))
75              caractère = alphabetDécaléMaj[caractère - 'A'];
76          super.write(caractère);
77      }
78  }
79
80  class Dechiffrement extends FilterReader
81  {
82      final static char alphabetDécaléMin[] =
83          "abcdefghijklmnopqrstuvwxyz".toCharArray();
84      final static char alphabetDécaléMaj[] =
85          "ABCDEFGHIJKLMNOPQRSTUVWXYZ".toCharArray();
86
87      Dechiffrement(Reader entrée)
88      {
89          super(entrée);
90      }
91      public int read() throws IOException
92      {
93          int caractère = super.read();
94          if ((caractère >= 'a') && (caractère <= 'z'))
95              caractère = alphabetDécaléMin[caractère - 'f'];
96          else if ((caractère >= 'A') && (caractère <= 'Z'))
97              caractère = alphabetDécaléMaj[caractère - 'F'];
98          return caractère;
99      }
100 }
```

```
# cat txt
```

```
il fait beau aujourd'hui !
N'est-il pas ?
# java ChiffrerDechiffrer txt
# cat txt~ txt~~
nq kfny gjfz fzotzwi'mzn !
S'jxy-nq ufx ?
il fait beau aujourd'hui !
N'est-il pas ?
```

3.3.7 Récapitulatif de la hiérarchie des classes relatives aux entrées-sorties

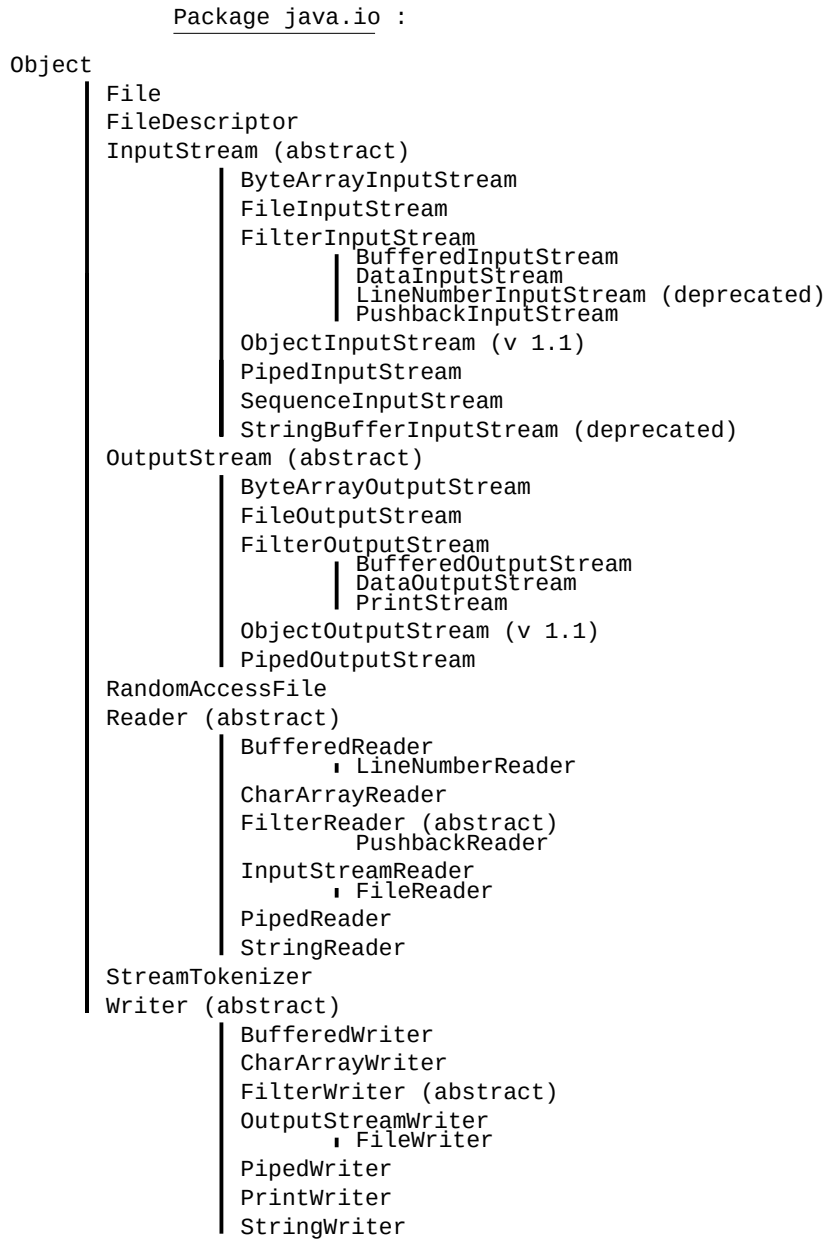


FIG. 3.5 – Version simplifiée du *package java.io*.

Chapitre 4

L'*API* réseau standard, le *multithreading* et le parallélisme en *Java*™

Sommaire

4.1 L'*API* réseau standard : *java.net*

Modèle *OSI* (*Open Systems Interconnection*) à 7 couches : la couche physique, la couche de liaison de données, la couche réseau, la couche transport, la couche session, la couche présentation et la couche application. En pratique, pour schématiser le réseau des réseaux, l'Internet donc, on utilise couramment un modèle à 4 couches : la couche physique et de liaison fait référence aux protocoles de sous-réseau que sont Ethernet, Fast Ethernet, FDDI, LocalTalk, Token Ring..., la couche réseau *IP* (*Internet Protocol*) se charge du regroupement des données en paquets et de l'identification des postes distants, la couche transport offre des services de base permettant le transport des messages (elle correspond essentiellement aux protocoles *TCP*, *Transmission Control Protocol*, et *UDP*, *User Datagram Protocol*), la couche application, enfin, se charge de mettre en forme pour l'utilisateur les données transmises.

Java™ ne se préoccupe pas de la première couche "physique et de liaison". Il ne prend en charge que les 3 dernières couches du modèle et ce uniquement dans le cas de l'*IP* : ainsi, par exemple, les protocoles AppleTalk (Macintosh) et IPX (Netware) de la couche réseau lui sont inconnus.

▶ **L'adressage *IP* :** Le protocole *IP* est basé sur une stratégie de commutation de paquets ; il véhicule des "paquets" d'octets en mode non connecté (ce qui signifie qu'il n'y a pas nécessité d'une négociation préalable de connexion). La conversion d'une adresse logique en une adresse physique se fait pas le biais du protocole *ARP*, *Address Resolution Protocol*.

En *Java*™, c'est la classe *java.net.InetAddress* qui se charge de la manipulation des adresses *IP* et de la résolution de nom.

```
1  import java.net.*;
2
3  class NslookupSimple
4  {
5      public static void main ( String [] arguments)
```

```
6       {
7           try {
8               InetAddress tmp = InetAddress.getByName(arguments[0]);
9               System.out.println ("Name :    " + tmp.getHostName() +
10                                  "\nAddress :    " + tmp.getHostAddress());
11          }
12          catch (ArrayIndexOutOfBoundsException uneException) {
13              System.err.println ("Usage : java NslookupSimple (NomDeMachine|AdresseIP)");
14              System.exit (1);
15          }
16          catch (UnknownHostException uneException) {
17              System.err.println ("Machine inconnue");
18              System.exit (2);
19          }
20      }
21  }
```

► **Le concept de port de la couche transport :** Tandis que le protocole *IP* permet l'acheminement des paquets de données d'une machine vers une autre, les deux protocoles *TCP* et *UDP* de la couche transport permettent à deux processus résidant sur des machines distantes de pouvoir dialoguer. Or, pour pouvoir communiquer avec un processus distant, il est nécessaire de pouvoir l'identifier précisément et de le distinguer des autres processus qui s'exécutent sur la machine distante (celle-ci peut, en effet, être simultanément serveur de fichiers, serveur d'annuaires, serveur de temps...).

En fait, la couche transport privilégie plus exactement la notion de service dispensé par un hôte distant à celle de processus s'exécutant sur l'hôte distant, partant du principe qu'un même service peut être offert indifféremment par plusieurs processus de la même machine (c'est exactement le cas d'un serveur Web tel qu'Apache lorsqu'il s'exécute sur une machine *multithreads*). Le port[1], qui est un nombre entier de l'intervalle $\{1,...,65535\}$, est le mécanisme retenu pour désigner le point d'émission ou réception d'un service sur une machine hôte. La donnée d'une adresse *IP* et d'un numéro de port permet de caractériser un service donné sur une machine hôte.

► **Les protocoles *UDP* et *TCP* :** Le protocole *UDP* n'enrichit essentiellement le protocole IP sous-jacent qu'en termes de multiplexage des échanges entre deux ou plusieurs hôtes. Il offre à une application la possibilité d'envoyer des messages structurés (l'unité de données échangées s'appelle un datagramme), mais ne garantit pas leur bonne réception par le destinataire ni l'ordre d'arrivée des datagrammes. Il n'enrichit donc d'aucune qualité de service le protocole réseau et se révèle très simple et efficace mais peu fiable.

À l'opposé, *TCP* est un protocole fiable qui permet à deux applications résidant sur des machines distantes ou non, d'échanger des données (on parle dans ce cas de segments). Une connexion *TCP* s'effectue de la manière suivante :

- procédure d'établissement de la connexion entre les deux extrémités par mécanisme de *three-way handshake* (initialisation en trois temps),
- échange de données en mode *full duplex* (communication simultanément bidirectionnelle), avec garantie de l'acheminement des données dans l'ordre et sans redondance des caractères qui auraient été réémis par le protocole de liaison,
- procédure de fin de connexion respectant les qualités de service énoncées dans le point précédent.

Bien que ces deux protocoles soient très couramment utilisés sur l'Internet, nous développerons essentiellement dans la suite de ce chapitre des applications basées sur le protocole *TCP*.

► **Interaction de processus distants, *socket* et modèle client-serveur :** Comme nous l'avons vu précédemment avec les threads, il est tout à fait possible pour deux ou plusieurs tâches distinctes sur une machine mono-processeur (ou sur une machine multi-processeurs à mémoire partagée) d'accéder et de manipuler un même ensemble de données. Par contre, dans le cas de processus ne partageant pas un espace d'adressage commun (typiquement dans le cas de communications entre machines distantes), la manipulation concurrente d'un même ensemble de données nécessite de faire appel à des mécanismes de communication par messages.

L'un des mécanismes que propose *Java*™ pour l'échange de données entre deux processus s'exécutant sur des hôtes distants est à base de *socket*. Les *sockets* ont à l'origine été développées sur l'Unix de Berkeley, mais le paquetage *java.net* de l'*API* standard permet à toute machine virtuelle *Java*™ de reproduire leur comportement quelle que soit la plate-forme réelle sur laquelle elle s'exécute. Une *socket* est donc une "extrémité d'un canal de communication" par laquelle un processus peut émettre ou réceptionner des informations d'un autre processus, qu'il soit distant ou non.

Dans ce cadre précis d'applications réparties, les processus impliqués sont classés en deux catégories (mais celles-ci peuvent évoluer au cours du temps et les rôles sont tout à fait susceptibles de s'inverser). Il y a d'un côté le(s) serveur(s) et de l'autre le(s) client(s). Les processus peuvent ou non résider sur une même machine hôte, puisqu'en fait les notions de client et de serveur font référence non pas à des processeurs mais bien à des processus.

[1] Attention, il s'agit d'un port logique, qui n'a rien à voir avec les ports série, parallèle ou PS/2 de votre machine !

Il existe deux types bien distincts de serveurs. Ceux-ci peuvent être concurrents ou itératifs. Dans le premier cas, chaque nouvelle sollicitation entraîne la création d'une tâche[2] chargée de prendre en charge la connexion (ce qui permet de contourner le goulot d'étranglement que constitue le serveur unique), dans le second, c'est le même processus qui répond successivement aux divers clients.

4.1.1 Le point de vue du client

```
1  import java.net.*;
2  import java.io.*;
3
4  class ClientReseauSimpliste
5  {
6      public static void main ( String [] arguments)
7          throws UnknownHostException,IOException
8      {
9          String ligne;
10         // instanciation de la socket cliente. arguments[0] correspond
11         // à l'adresse du serveur et arguments[1] correspond au port
12         // sur lequel est distribué le service (ce second argument du
13         // constructeur de la socket doit au préalable être convertit
14         // sous forme d'une valeur entière) :
15         Socket maSocket = new Socket(arguments[0], Integer.parseInt (arguments [1]));
16         // association d'un flux de lecture (avec tampon) à la socket
17         // courante à l'aide de la méthode d'intance getInputStream de
18         // la classe Socket :
19         BufferedReader entrée =
20             new BufferedReader(new InputStreamReader(maSocket.getInputStream ()));
21         // tant qu'il reste des "lignes" de données à lire dans le
22         // flux de lecture, les lire puis les afficher sur la sortie
23         // standard du poste client :
24         while (( ligne = entrée.readLine ()) != null)
25             System.out.println ( ligne );
26         // fermer le flux d'entrée lié à la socket et la socket elle-même :
27         entrée.close ();
28         maSocket.close ();
29     }
30 }
```

```
# java ClientReseauSimpliste localhost 13
Tue Jul 4 23 :42 :21 2000
```

Nous pouvons aussi la tester avec le port 19 (auquel est associé le service *chargen*).

Un scanner de ports *TCP*

```
1  import java.net.*;
2  import java.io.*;
3
4  class ScannerDePortsTCP
5  {
6      private String adresse;
7      private int portMax;
8
9      ScannerDePortsTCP(String adresse, int portMax)
10     {
11         this.adresse = adresse;
12         this.portMax = portMax;
13     }
14
15     public void lancer ()
16     {
17         Socket maSocket;
18
19         for( int port =1; port <= portMax ; port ++) {
20             try {
21                 maSocket = new Socket(adresse, port );
22                 System.out.println ( port + "/tcp open sur " + adresse );
23             }
24             catch ( UnknownHostException uneException) {
25                 System.err.println ("La machine " + adresse + " est inaccessible !" );
26                 System.exit (1);
27             }
28             catch ( ConnectException uneException) {
29             }
30             catch ( IOException uneException) {
31                 System.err.println ("Erreur de lecture du flux d'entrée." );
32                 System.exit (2);
33             }
34             catch ( IllegalArgumentException uneException) {
35                 System.err.println ("Vous êtes sorti de l'intervalle 1, ..., 65535." );
36                 System.exit (3);
37             }
38         }
39     }
40
41     public static void main ( String [] arguments) throws InterruptedException
42     {
43         String cible = arguments[0];
44         int portMax = Integer.parseInt (arguments[1]);
45
46         long dateDébut = System.currentTimeMillis (); // mesure du temps
47
48         ScannerDePortsTCP monScan = new ScannerDePortsTCP(cible,portMax);
49         monScan.lancer();
50
51         long dateFin = System.currentTimeMillis (); // mesure du temps
```

[2]Elle peut aussi être prise en charge par l'une des tâches créées à l'initialisation et qui n'est pas détruite à la fin de chaque sollicitation.

```
52              System.err . println ("Temps d'exécution = "  + ( dateFin−dateDébut) + " ms");
53          }
54      }
```

4.1.2 Le point de vue du serveur

Schéma de fonctionnement d'un serveur *TCP*

Un serveur *TCP* a un rôle relativement passif dans l'établissement de la connexion avec le client. En effet, après avoir informé la machine virtuelle et le système qu'il est en mesure de recevoir les connexions des clients, le serveur se place simplement en position d'attente de demande de connexion sur le port concerné (à l'aide de la méthode *accept*). En *Java*™, la création d'une *socket* serveur, son rattachement à un port donné et l'ouverture du service sont pris en charge par la classe *ServerSocket* du paquetage *java.net*[3].

En pratique, une simple instanciation de la classe *ServerSocket* permet de disposer d'une *socket* d'écoute attachée au port correspondant au service proposé. Cette écoute se "matérialise" dans le code de votre serveur par invocation de la méthode d'instance *accept* de la classe *ServerSocket*. Lorsqu'une demande de connexion *TCP* en provenance d'un client arrive sur le port concerné, le serveur en prend connaissance par *accept* et instancie immédiatement une nouvelle *socket* dédiée uniquement à cette connexion avec le client. On appelle généralement cette *socket* la *socket* de service. La connexion du client avec le serveur est alors effective et les méthodes d'instances *getLocalAddress* et *getLocalPort* de la classe *Socket* permettent de récupérer respectivement l'adresse *IP* et le port du serveur, tandis que les méthodes d'instance *getInetAddress* et *getPort* de la classe *Socket* permettent de récupérer respectivement l'adresse *IP* et le port du client. La donnée de ce 4-uplet permet d'identifier complètement la connexion.

La classe `ServeurReseauSimpliste` ci-après met en place un serveur sur le port donné en argument de la ligne de commande et se positionne en attente de connexion. Lorsqu'un client le sollicite, il affiche les paramètres de la connexion (c'est-à-dire le 4-uplet qui permet de l'identifier), puis ferme celle-ci. Nous sommes exactement dans le cas d'un serveur qui répond par simple sollicitation sur son port d'attache :

```
1   import java . net .*;
2   import java . io .*;
3
4   class  ServeurReseauSimpliste
5   {
6       private  int  port ;
7       private  ServerSocket  leServeur ;
8       private  Socket  uneConnexionEtablie;
9
10      ServeurReseauSimpliste ( int  port )
11      {
12          this . port  =  port ;
13      }
14
15      void  lancer  ()  throws IOException
16      {
17          // Lançons le serveur sur le port identifié par le numéro "port" :
18          leServeur  =  new ServerSocket( port );
19          System.out. println ("Un serveur  est  lancé  sur  le  port  " + leServeur . getLocalPort ());
20          // Mettons le serveur en attente d'une connexion sur ce port :
21          uneConnexionEtablie = leServeur . accept ();
22          System.out. println ("Contact  établi  entre  :\ n− le  serveur  " +
23                               uneConnexionEtablie.getLocalAddress () +  ",  sur  le  port  " +
24                               uneConnexionEtablie. getLocalPort () +
25                               "\n− et  le  client  " + uneConnexionEtablie. getInetAddress () +
26                               ",  sur  le  port  " + uneConnexionEtablie. getPort ());
27      }
28
29      void  arrêter  ()  throws IOException
30      {
31          uneConnexionEtablie. close ();
32          leServeur . close ();
33      }
34
35      public  static  void main ( String []  arguments)  throws IOException
36      {
37          // instanciation d'un nouveau serveur sur le port passé en argument de
38          // la ligne de commande :
39          ServeurReseauSimpliste  monServeur =
40              new ServeurReseauSimpliste( Integer . parseInt (arguments [0]));
41          monServeur.lancer ();
42          monServeur. arrêter ();
43      }
44  }
```

4.1.3 Échange de données en *UDP* : exemple de remise non fiable

La totalité des clients et serveurs que nous avons développés jusqu'à présent communiquent à base de *socket TCP*. Il est cependant tout a fait possible d'utiliser le protocole *UDP*, moins fiable mais beaucoup plus rapide, pour permettre à deux processus distants de communiquer.

Ce protocole, défini dans la RFC 768, garantit essentiellement que, s'il y a réception du datagramme[4], alors celui-ci est reçu dans son intégralité sans perte de données. Par contre, comme nous allons le constater avec l'exemple

[3] Dans un langage plus traditionnel comme le langage C, cette simple mise en place se décompose en trois étapes : création de la *socket* par la primitive *socket*, rattachement de la *socket* à un port par *bind* et ouverture du service par *listen*. En *perl*, une application serveur basée sur les *sockets* s'initie aussi à l'aide de trois primitives portant les mêmes noms.

[4] Le datagramme est l'unité de transmission en *UDP*.

ci-après, il ne garantit absolument pas l'entière réception de la totalité des datagrammes (ni même d'ailleurs leur ordre d'arrivée…). C'est au protocole applicatif de s'en charger, si nécessaire.

```
1  import java.net.*;
2  import java.io.*;
3
4  class EnvoiUDP
5  {
6      private final static int NOMBRE_D_ENVOIS = 240;
7      InetAddress adresseCible;
8      int portCible;
9
10     EnvoiUDP(String adresseCible, int portCible) throws UnknownHostException
11     {
12         this.adresseCible = InetAddress.getByName(adresseCible);
13         this.portCible = portCible;
14     }
15
16     private byte[] forgerLeMessage(int tailleDatagramme)
17     {
18         // cette méthode remplit un tableau d'octets par des séquences de
19         // chiffres décimaux stockés sous forme d'octets (la taille totale du
20         // tableau vaut "tailleDatagramme") :
21         byte[] résultat = new byte[tailleDatagramme];
22         for (int i=0; i<tailleDatagramme; i++)
23             // Rappel : 48 est le code ASCII du chiffre 0, 49 celui du chiffre
24             // 1,... et 57 celui du chiffre 9 :
25             résultat[i] = (byte)(48 + i%10);
26         return résultat;
27     }
28
29     void envoyer(int tailleDatagramme) throws SocketException, IOException
30     {
31         // la méthode envoyer génère un datagramme à partir d'un tableau de
32         // "tailleDatagramme" octets, puis boucle un nombre de fois égal à
33         // "NOMBRE_D_ENVOIS" sur son envoi par la méthode send d'une instance
34         // "anonyme" de DatagramSocket :
35         DatagramSocket laSocketUDP = new DatagramSocket();
36         DatagramPacket leDatagramme =
37             new DatagramPacket(forgerLeMessage(tailleDatagramme),tailleDatagramme,
38                                adresseCible, portCible);
39
40         for(int i=1; i<=NOMBRE_D_ENVOIS; i++) {
41             laSocketUDP.send(leDatagramme);
42             System.out.println("datagramme " + i + "/" + tailleDatagramme + " octets");
43         }
44     }
45
46     public static void main(String[] arguments)
47         throws UnknownHostException, SocketException, IOException
48     {
49         new EnvoiUDP(arguments[0],Integer.parseInt(arguments[1])).
50             envoyer(Integer.parseInt(arguments[2]));
51     }
52 }
```

Pour valider l'émission de ces datagrammes par le client, nous avons aussi écrit une classe `ReceptionUDP` qui se charge de créer un serveur *UDP* pour recenser le nombre et la taille respective de chaque datagramme reçu. En pratique, cette classe instancie initialement un *DatagramSocket* et l'attache au port dont le numéro est indiqué en argument de la ligne de commande. Ensuite, elle instancie un *DatagramPacket* à partir d'un tableau de 65316 octets (la taille est en fait une valeur que l'on peut passer en argument sur la ligne de commande) et boucle indéfiniment sur la réception de datagrammes à l'aide de la méthode *receive* appliquée au *DatagramSocket* courant.

```
1  import java.net.*;
2  import java.io.*;
3
4  class ReceptionUDP
5  {
6      int portCible;
7      DatagramSocket leServeur;
8
9      ReceptionUDP(int portCible) throws SocketException,IOException
10     {
11         this.portCible = portCible;
12         leServeur = new DatagramSocket(portCible);
13     }
14
15     void réception(int tailleDatagramme) throws IOException
16     {
17         DatagramPacket leDatagrammeDeRéception =
18             new DatagramPacket(new byte[tailleDatagramme],tailleDatagramme);
19         int i=1;
20
21         System.out.println("Le serveur est à l'écoute ...");
22         while (true) {
23             leServeur.receive(leDatagrammeDeRéception);
24             System.out.println("Le serveur a reçu " + tailleDatagramme +
25                                " octets – réception numéro " + i++);
26             // System.out.println(new String(tampon));
27         }
28
29     }
30
31     public static void main(String[] arguments) throws SocketException,IOException
32     {
33         new ReceptionUDP(Integer.parseInt(arguments[0])).
34             réception(Integer.parseInt(arguments[1]));
35     }
36 }
```

À titre d'information, nous avons utilisé les utilitaires `ReceptionUDP` et `EnvoiUDP` entre deux machines distantes d'un même réseau local commuté, à plat, en *Fast Ethernet*. Nous nous sommes rendu compte que, sur un tel réseau, en dehors des heures d'affluence, il n'y a quasiment pas de perte de datagramme (celles-ci sont de l'ordre de 1‰ à quelques unités près dans certaines configurations). Par contre, lorsque la couche physique se "dégrade" (si, par exemple, on repasse dans une architecture en bus, ou si l'on cascade les éléments de réseau…), on peut fréquemment

perdre deux tiers des datagrammes au cours du transfert, ce qui n'a plus rien de négligeable !

4.2 *Java*™ et le *multithreading*

Java™ est un langage multitâches ⇔ il propose "nativement" des fonctionnalités permettant d'exécuter concurremment (et simultanément sur une machine multiprocesseurs[5]) plusieurs séquences d'instructions d'une même classe ou non. Chaque séquence, que l'on appelle aussi flot de contrôle, processus léger ou contexte d'exécution, est un *thread*. Les *threads* sont des processus légers qui s'exécutent, en parallèle[6], dans le même espace d'adressage, et ce, de manière concurrente.

Toute application *Java*™ ne dispose, lors de son lancement, que d'un seul *thread*, le *thread* initial. Pour en créer de nouveaux, elle doit impérativement instancier un nouvel objet de la classe *java.lang.Thread* pour le lancer ensuite. Il existe, pour ce faire, deux façons de procéder. La première consiste à considérer que le code qui doit être lancé dans la nouvelle *thread* est membre d'une classe qui hérite de la classe *java.lang.Thread*. La seconde consiste à placer le code qui doit être lancé dans la nouvelle *thread* dans une classe qui implémente l'interface *java.lang.Runnable*[7]. La première solution a l'avantage de ne manipuler qu'un objet mais elle a l'inconvénient d'empêcher tout nouvel héritage pour la classe considérée.

4.2.1 Création explicite d'un *thread*

⇔ soit dériver la classe *java.lang.Thread* et instancier ensuite la classe dérivée, soit créer une classe implémentant l'interface *Runnable*.

Création par héritage de la classe *Thread*

Pour que la classe courante puisse opérer au sein d'un contexte d'exécution différent du contexte d'exécution initial, elle doit instancier un nouvel objet de la classe *Thread*. Pour ce faire, elle peut, assez simplement, étendre la classe *Thread* et surcharger la méthode *run* en y insérant le code de la tâche qui doit être exécutée concurremment au reste de l'application.

La méthode *run* (qui est définie, mais vide de contenu, dans la classe mère *Thread*) ne doit jamais être appelée directement pour lancer un *thread*. Il faut systématiquement invoquer la méthode *start* sur l'instance courante. Cette dernière méthode, qui est aussi définie dans la classe *Thread*, a la particularité de lancer la méthode *run* et de rendre la main au processus léger appelant sans attendre que la méthode *run* ait achevé sa tâche.

```
1  public class Factoriel1 extends Thread
2  {
3      private int valeur;
4
5      Factoriel1 (int valeur)
6      {
7          this.valeur = valeur;
8      }
9
10     public void run()
11     {
12         long résultat = 1;
13         System.out.println("Thread de calcul : démarrage du calcul");
14         while (valeur > 1)
15             résultat *= valeur--;
16         System.out.println("Thread de calcul : le résultat vaut " + résultat);
17     }
18
19     public static void main (String [] arguments)
20     {
21         new Factoriel1 (Integer.parseInt(arguments[0])).start();
22         System.out.println("Thread principal : je reprends la main");
23     }
24 }
```

Création par implémentation de l'interface *Runnable*

Il existe une autre manière de rendre une classe *multithreads*. Il faut pour cela qu'elle implémente l'interface *Runnable* et son unique méthode *run*. L'avantage est que cela permet à la classe courante de pouvoir hériter d'une autre classe que la classe *Thread*. L'inconvénient tient au fait qu'il faut explicitement instancier la classe *Thread* et passer l'objet à manipuler en argument de constructeur (pour ensuite invoquer la méthode *start* sur le *thread* lui-même).

```
1  public class Factoriel2 implements Runnable
2  {
3      private int valeur;
4
5      Factoriel2 (int valeur)
6      {
```

[5] À condition d'utiliser non pas les *green threads* mais les *native threads* lors du lancement de la machine virtuelle *Java*™...

[6] Ce parallélisme est simulé dans le cas d'une machine mono-processeur.

[7] Cette interface ne définit que la méthode abstraite *run*.

```
 7              this . valeur  =  valeur ;
 8          }
 9
10          public  void  run ()
11          {
12              long  résultat   = 1;
13              System.out. println ("Thread de  calcul  :  démarrage du  calcul ");
14              while  ( valeur  > 1)
15                  résultat  *= valeur --;
16              System.out. println ("Thread de  calcul  :  le  résultat  vaut  " +  résultat );
17          }
18
19          public  static  void  main ( String []  arguments)
20          {
21              new Thread(new Factoriel2( Integer . parseInt (arguments [0]))).  start  ();
22              System.out. println ("Thread  principal  :  je  reprends  la  main");
23          }
24      }
```

4.2.2 Notion de priorité et contrôle des *threads*

Notion de priorité

Du point de vue du développeur et de l'utilisateur, l'exécution des *threads* n'est pas vraiment déterministe. Pour pouvoir les utiliser à bon escient, il est donc important de les contrôler.

Pour donner l'illusion à un utilisateur que sa machine mono-processeur est capable de gérer plusieurs tâches simultanément, la *JVM* procède par *scheduling*. C'est-à-dire qu'elle alloue une fraction de temps processeur à chaque *thread* actif et le préempte (le suspend provisoirement) au bout d'un certain laps de temps pour rendre la main à un autre processus léger actif. L'algorithme de *scheduling* est déterministe (même si ce n'est pas forcément perceptible pour l'utilisateur) et à base de notion de priorité.

La priorité est un attribut propre à chaque *thread* qui lui permet d'être, ou non, traité avant ses congénères. C'est une propriété qui est fortement prise en compte dans l'algorithme de *scheduling*. Ainsi, si un *thread* `T1` est lancé avec une priorité plus élevée que celle d'un *thread* `T2` actif, le *scheduler* suspend l'exécution de `T2` et donne la main à `T1`. Si ces deux processus légers ont un niveau de priorité équivalent, le *scheduler* leur alloue respectivement et alternativement un peu de temps processeur.

Lors de sa création, un *thread* hérite de la priorité du *thread* appelant (celui qui l'a instancié). Au lancement d'une application *Java*™, la priorité initiale du *thread* principal vaut par défaut *Thread.NORM_PRIORITY*. Le niveau de priorité d'un *thread* se modifie dynamiquement avec la méthode *setPriority* qui prend un entier en argument (cet entier doit être compris entre *Thread.MIN_PRIORITY* et *Thread.MAX_PRIORITY* ou une exception de type *IllegalArgumentException* est levée).

► **Important :** Dans le cadre d'une interface graphique, il ne faut pas oublier de rendre le *thread* chargé de la gestion des événements plus prioritaire que celui qui est chargé des calculs ! Si tel n'est pas le cas, votre interface ne pourra pas réellement piloter les traitements puisqu'elle n'aura pas la possibilité de reprendre la main au moment où vous la solliciterez…

Cycle de vie d'un *thread*

un *thread* (après instanciation) est activé par invocation de la méthode *start*. Le *scheduler* est alors en mesure de lui donner la main pour qu'il s'exécute, en fonction des critères de la "règle du tourniquet" (*setPriority* permet de modifier sa priorité). Lorsqu'il est élu et qu'il passe donc du stade *runnable* au stade *run*, il peut décider de lui-même de rendre la main au *scheduler* par la méthode *yield*. Sinon, il s'exécute tant que le *scheduler* ne le préempte pas. Un *thread* peut sortir de l'état *runnable* en invoquant la méthode *wait*, la méthode *sleep* ou en restant bloqué sur une opération d'entrée-sortie. Dans chacun de ces trois cas, le *thread* retourne en état *runnable* respectivement, lorsqu'il reçoit un *notify* ou un *notifyAll* (cas du *wait*), lorsque la période d'endormissement s'achève ou lorsque l'opération d'entrée-sortie se termine. Un *thread* meurt de sa "belle mort" lorsque sa méthode *run* s'achève.

Contrôle des *threads*

Quand plusieurs processus légers sont lancés concurremment, vous pouvez par exemple souhaiter les synchroniser, les obliger à rendre la main, les endormir temporairement, les détruire… *Java*™ propose pour cela un certain nombre de primitives. Sachez que, dans la version 1.2 du langage, les méthodes *resume*, *stop* et *suspend* ont été déclarées obsolètes. Nous ne les présenterons donc pas. Les méthodes d'instances qui suivent permettent de piloter le *thread* auquel elles s'appliquent :

Méthode	Signification

destroy	Détruit le *thread* courant
interrupt	Interrompt le déroulement du *thread* courant
join	Positionne une barrière en attendant la fin du *thread* courant
sleep	Endort le *thread* courant pendant un nombre de millisecondes égal à l'argument
yield	Rend la main au *scheduler*

4.2.3 Positionner un verrou avec les *threads*

En *Java*™, on verrouille (du point de vue du *multithreading*) un objet ou une méthode à l'aide du modificateur *synchronized*. L'application de ce mot clef à une entité implique que, si celle-ci est accédée au sein d'un *thread*, alors elle est assurée de l'être exclusivement : tout autre *thread* doit attendre la fin de cet accès pour l'accéder à son tour.

Pour préciser cette notion de verrou, nous proposons ci-après un exemple d'accès concurrent à une même ressource (non verrouillée) entraînant une erreur de calcul, et deux façons différentes de résoudre ce problème. En effet, normalement, au cours de son exécution, chaque *thread* est indépendant des autres *threads* de la même application, mais tous ont accès au même espace d'adressage (aux mêmes données). Il peut donc arriver qu'une même donnée soit accédée en modification par deux ou plusieurs *threads* simultanément... ce qui génère immanquablement des conflits et des erreurs ! C'est exactement ce qui se produit dans le cas de la classe `AccesConcurrent1` où deux instances de `Fournisseur` `a` et `b` accèdent concurremment en écriture à la même variable d'instance `taille`.

De l'intérêt de la méthode *yield*. Après avoir activé les méthodes *run* des deux *threads* `a` et `b` (en invoquant leurs méthodes *start*), le constructeur de la sur-classe `AccesConcurrent1` attend leurs morts respectives pour afficher la valeur de *taille* résultat de leurs calculs. Pendant ce temps, `a` ou `b` lance la méthode `incrémenter` et commence par stocker la valeur de la variable `taille` avant de repasser la main au *scheduler* lorsqu'il arrive sur l'instruction comprenant le *yield*. L'autre *thread* se lance alors, commence par stocker la valeur de la variable `taille`, puis repasse la main au *scheduler* lorsqu'il arrive sur l'instruction comprenant le *yield*. Le premier des deux *threads* reprend alors la main et incrémente `taille` de 10 unités (qui prend alors la valeur 10) avant de se terminer et de "rendre le processeur" à son confrère. **Le problème est que ce dernier a stocké une valeur nulle pour la variable `taille` et que celle-ci n'est plus à jour**. Il incrémente alors à son tour `taille` de 10 unités (qui prend alors encore la valeur 10) avant de se terminer et de "rendre le processeur" au *thread* initial.

```
1  class AccesConcurrent1
2  {
3      static int taille = 0;
4
5      AccesConcurrent1()
6      {
7          Thread a,b;
8          (a = new Thread(new Fournisseur ())). start ();
9          (b = new Thread(new Fournisseur ())). start ();
10         // on attend la fin des threads a et b pour procéder à l'affichage de
11         // la valeur de "taille" dans le thread principal :
12         try {
13             a. join ();
14             b. join ();
15         } catch ( InterruptedException uneException) {}
16         System.out. println (" Taille totale : " + taille );
17     }
18
19     class Fournisseur implements Runnable
20     {
21         public void run()
22         {
23             int tmp = AccesConcurrent1. taille ;
24             // l'instruction yield qui suit rend la main au scheduler : c'est ce
25             // qui va permettre à la valeur de la variable d'instance "taille" de
26             // changer entre temps dans l'autre thread...
27             Thread. currentThread (). yield ();
28             AccesConcurrent1. taille = tmp + 10;
29         }
30     }
31
32     public static void main ( String [] arguments)
33     {
34         new AccesConcurrent1();
35     }
36 }
```

L'exemple est un peu surfait mais il a l'intérêt de mettre en évidence ce problème d'accès concurrents à une même ressource non verrouillée. Voici le résultat obtenu à l'exécution :

```
# java AccesConcurrent1
Taille totale : 10
```

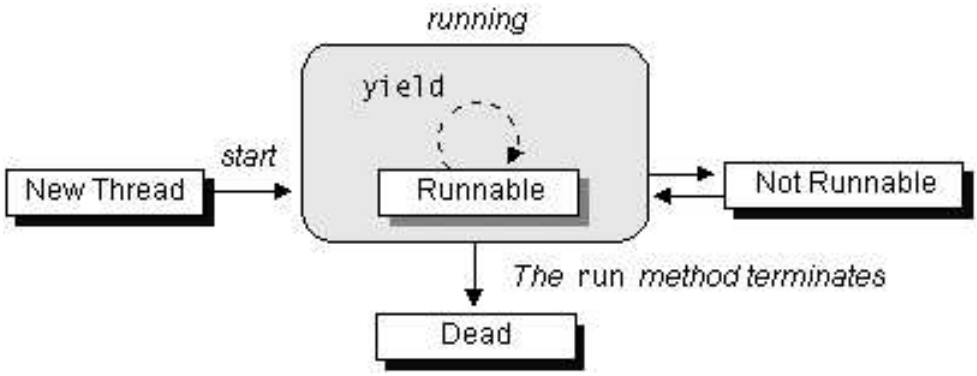


FIG. 4.1 – Cycle de vie d'un *thread*

▶ **Première solution : verrouiller l'objet au moment où il est modifié !** Un premier remède consiste à verrouiller l'objet instance de la surclasse pendant le laps de temps où l'on incrémente sa variable d'instance. De cette façon, la portion de code qui contient l'appel à la méthode `incrémenter` est non plus "parallèle" mais devient forcément séquentielle. Elle ne peut être exécutée que par un seul *thread* simultanément. C'est exactement ce que l'on réalise ci-après avec la classe `AccesConcurrent2` :

```
1  class AccesConcurrent2
2  {
3      static int taille = 0;
4
5      AccesConcurrent2()
6      {
7          Thread a,b;
8          (a = new Thread(new Fournisseur(this ))). start ();
9          (b = new Thread(new Fournisseur(this ))). start ();
10         try {
11             a. join ();
12             b. join ();
13         } catch ( InterruptedException uneException) {}
14         System.out. println (" Taille totale : " + taille );
15     }
16
17     class Fournisseur implements Runnable
18     {
19         AccesConcurrent2 objet;
20
21         Fournisseur (AccesConcurrent2 objet)
22         {
23             this . objet = objet;
24         }
25
26         public void run()
27         {
28             synchronized(objet ) {
29                 // on "séquentialise" cette portion de code par la
30                 // pose d'un verrou relatif à "objet" :
31                 int tmp = AccesConcurrent2. taille ;
32                 Thread. currentThread (). yield ();
33                 AccesConcurrent2. taille = tmp + 10;
34             }
35         }
36     }
37
38     public static void main ( String [] arguments)
39     {
40         new AccesConcurrent2();
41     }
42 }
```

Les *threads* `a` et `b` posent tour à tour un verrou sur l'objet instance de la surclasse et l'incrémentation de `taille` est donc faite en séquence et non simultanément. On obtient donc le bon résultat :

```
# java AccesConcurrent2
Taille totale : 20
```

▶ **Deuxième solution : verrouiller la méthode par laquelle le problème arrive !** Le second remède, plus simple à mettre en œuvre, consiste à poser directement un verrou sur la méthode qui accède à la valeur de la variable d'instance en lecture et en écriture. En l'occurence, cela signifie qu'il faut verrouiller directement `incrémenter`, comme nous le faisons ci-après dans la classe `AccesConcurrent3` :

```
1  class AccesConcurrent3
2  {
3      static int taille = 0;
4
5      AccesConcurrent3()
6      {
7          Thread a,b;
8          (a = new Thread(new Fournisseur ())). start ();
9          (b = new Thread(new Fournisseur ())). start ();
10         try {
11             a. join ();
12             b. join ();
13         } catch ( InterruptedException uneException) {}
14         System.out. println (" Taille totale : " + taille );
15     }
16
17     synchronized static void incrémenter( int unEntier)
18     {
19         // la méthode incrémenter est synchronized... Elle n'est donc invoquée
20         // que par un unique thread au plus.
21         int tmp = AccesConcurrent3. taille ;
22         Thread. currentThread (). yield ();
23         AccesConcurrent3. taille = tmp + 10;
24     }
25
26     class Fournisseur implements Runnable
27     {
28         public void run()
29         {
30             AccesConcurrent3.incrémenter (10);
31         }
32     }
33
34     public static void main ( String [] arguments)
35     {
36         new AccesConcurrent3();
37     }
38 }
```

Les *threads* `a` et `b` posent tour à tour un verrou sur la méthode `incrémenter`, l'incrémentation de `taille` est donc faite en séquence et non simultanément. On obtient le bon résultat :

```
# java AccesConcurrent3
Taille totale : 20
```

► **Remarque :** Il est équivalent de verrouiller une méthode d'instance ou de verrouiller le bloc d'instructions de la même méthode d'instance sur l'objet courant. En clair, les deux portions de code qui suivent ont la même incidence.

```
1  synchronized void laMéthodeAVerrouiller () {
2    // bloc d'instructions ...
3  }
```

Ce qui précède équivaut en pratique à ce qui suit :

```
1  void laMéthodeAVerrouiller () {
2    synchronized ( this ) {
3      // bloc d'instructions ...
4    }
5  }
```

4.2.4 *Threads* et interface graphique : exemple pratique avec l'*API SWING*

Le *thread* principal se charge donc d'instancier un *thread* préposé à la gestion de la date au format français et un autre préposé à la gestion de la date au format allemand. Le constructeur de la classe principale `Horloge`, qui s'exécute dans le *thread* initial, commence par instancier une fenêtre (un objet *javax.swing.JFrame*), puis instancie la classe `HorlogeInterne` en attachant cette dernière instance à un composant graphique de la fenêtre pré-définie. Enfin, après avoir affiché l'ensemble à l'écran, il rend la main.

Entre-temps, le constructeur de l'instance d'`HorlogeInterne` s'est chargé de définir le format d'affichage de l'horloge en fonction de la "locale" qui lui est passée en argument, d'instancier un nouveau *thread* préposé au rafraîchissement de l'affichage et de le lancer en invoquant la méthode *start*. La méthode *run* du *thread* lié à l'instance d'`HorlogeInterne` se contente de boucler indéfiniment en mettant à jour l'affichage[8] en fonction de l'heure courante.

```
1  import javax.swing.*;
2  import java.awt.*;
3  import java. util .*;
4  import java. text .DateFormat;
5
6  class Horloge
7  {
8      // définition d'une classe interne qui n'a de raison d'être qu'au
9      // sein de la sur-classe Horloge :
10     class HorlogeInterne extends JPanel implements Runnable
11     {
12         private Thread threadDeMiseAJour = null;
13         private Date dateCourante = null;
```

[8] La méthode *repaint* invoque implicitement la méthode *paintComponent* que nous avons surchargée. Celle-ci, qui est une méthode héritée de la classe *javax.swing.JComponent*, utilise la technique du *double buffering* pour préparer l'affi chage en arrière-plan et le rendre visible à l'écran d'un seul coup.

```
14              private Font fonteHeure = new Font("Helvetica",Font.BOLD,40);
15              private Font fonteJour = new Font("Helvetica",Font.PLAIN,12);
16              private DateFormat formatHeure = null;
17              private DateFormat formatJour = null;
18
19              HorlogeInterne(Locale choixDeLaLocale)
20              {
21                  // On redimensionne le composant courant avant de fixer
22                  // les formats d'affichage de la date et de
23                  // l'heure. Enfin, on lance le thread de rafraîchissement
24                  // de l'horloge :
25                   setPreferredSize (new Dimension(280,80));
26                  formatHeure
27                       = DateFormat.getTimeInstance(DateFormat.MEDIUM,choixDeLaLocale);
28                  formatJour
29                       = DateFormat.getDateInstance(DateFormat.LONG,choixDeLaLocale);
30                  threadDeMiseAJour = new Thread(this);
31                  threadDeMiseAJour.start ();
32              }
33
34              public void paintComponent(Graphics g) {
35                  super.paintComponent(g);
36                  // affichage de l'heure et de la date courante
37                  // respectivement dans des fontes de grosse et petite
38                  // tailles :
39                  dateCourante = Calendar.getInstance().getTime();
40                  g.setFont(fonteHeure);
41                  g.drawString(formatHeure.format(dateCourante),40,50);
42                  g.setFont(fonteJour);
43                  g.drawString(formatJour.format(dateCourante),170,74);
44              }
45
46              public void run()
47              {
48                  // boucle infinie :
49                  while (true) {
50                      // mise à jour du composant graphique (repaint lance
51                      // paintComponent) :
52                      repaint ();
53                      try {
54                          // le thread "sommeille" pendant 1 seconde avant
55                          // de boucler :
56                          threadDeMiseAJour.sleep(1000);
57                      } catch (InterruptedException uneException) {}
58                  }
59              }
60          }
61
62          Horloge(Locale choixDeLaLocale)
63          {
64              // on commence par instancier la fenêtre qui correspond à
65              // l'horloge, puis on définit son comportement à la fermeture
66              // de la fenêtre :
67              JFrame laFenêtre = new JFrame("Mon horloge...");
68              laFenêtre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
69              // on insère ensuite le composant permettant l'affichage de
70              // l'horloge dans la fenêtre :
71              laFenêtre.getContentPane().add(new HorlogeInterne(choixDeLaLocale));
72              // on procède enfin à l'affichage de l'ensemble :
73              laFenêtre.pack();
74              laFenêtre.setVisible(true);
75          }
76
77          public static void main(String[] arguments)
78          {
79              new Horloge(Locale.FRANCE);
80              new Horloge(Locale.GERMAN);
81          }
82      }
```

► **Remarque :** Pour implémenter ces deux horloges nous aurions aussi pu instancier la classe *javax.swing.Timer*. Toute instance de cette classe offre, en effet, la possibilité d'envoyer, à intervalles de temps réguliers, la méthode *actionPerformed* à ses différents *ActionListeners* (il s'agit des objets qui doivent être informés de chaque fin de période de "sommeil" du *Timer*). L'emploi d'un *Timer* permet alors d'éviter d'utiliser explicitement les *threads* dans ce cas précis.

4.2.5 De l'utilité d'une barrière de synchronisation des *Threads*

```
1   public class ThreadJoin extends Thread
2   {
3       static int[] tab = {101,102,103,104,105,106,107};
4
5       static void affi cher ()
6       {
7           System.out.print("Affi chage global :");
8           for (int i=0; i<tab.length ; i++)
9               System.out.print(tab[i] + ",");
10          System.out.println ();
11      }
12
13      public static void main(String[] arguments) throws InterruptedException
14      {
15          Auxiliaire [] tabThread = new Auxiliaire[tab.length];
16          for (int i=0; i<tab.length ; i++) {
17              tabThread[i] = new Auxiliaire(i);
18              tabThread[i].start ();
19          }
20
21          // synchronisation "obligatoire" (à commenter pour voir...) :
22          for (int i=0; i<tab.length ; i++) tabThread[i].join ();
23
24          ThreadJoin.affi cher ();
25      }
26  }
27
```

```
28  class Auxiliaire extends Thread
29  {
30      private int i;
31
32      Auxiliaire (int i)
33      {
34          this.i=i;
35      }
36      public void run()
37      {
38          System.out.println (getName() + ":: opération sur élément indice " +
39                              i + ":" + (ThreadJoin.tab[i]--));
40      }
41  }
```

Avec la barrière de synchronisation :

```
# javac ThreadJoin.java
# java ThreadJoin
Thread-0 : : opération sur élément indice 0 : 101
Thread-1 : : opération sur élément indice 1 : 102
Thread-2 : : opération sur élément indice 2 : 103
Thread-3 : : opération sur élément indice 3 : 104
Thread-4 : : opération sur élément indice 4 : 105
Thread-5 : : opération sur élément indice 5 : 106
Thread-6 : : opération sur élément indice 6 : 107
Affi chage global : 100,101,102,103,104,105,106,
```

Sans la barrière de synchronisation (sur machine mono-processeur) :

```
# javac ThreadJoin.java
# java ThreadJoin
Thread-0 : : opération sur élément indice 0 : 101
Thread-1 : : opération sur élément indice 1 : 102
Thread-2 : : opération sur élément indice 2 : 103
Thread-3 : : opération sur élément indice 3 : 104
Thread-4 : : opération sur élément indice 4 : 105
Thread-5 : : opération sur élément indice 5 : 106
Affi chage global : 100,101,102,103,104,105,107,
Thread-6 : : opération sur élément indice 6 : 107
```

Sans la barrière de synchronisation (sur machine multi-processeurs) :

```
# javac ThreadJoin.java
# java -native ThreadJoin
Thread-4 : : opération sur élément indice 4 : 105
Thread-5 : : opération sur élément indice 5 : 106
Thread-0 : : opération sur élément indice 0 : 101
Affi chage global : Thread-6 : : opération sur élément indice 6 : 107
Thread-3 : : opération sur élément indice 3 : 104
Thread-2 : : opération sur élément indice 2 : 103
Thread-1 : : opération sur élément indice 1 : 102
100,101,102,103,104,105,106,
```

4.3 JOMP : une implémentation de OpenMP en *Java*™

Fin des années 90 : retour en force des machines multi-processeurs à mémoire partagée. Chaque construteur a son propre jeu de directives de parallélisation semi-implicite $\Rightarrow$ nécessité d'un consensus. Le 28 octobre 1997[9], un ensemble d'industriels et constructeurs adoptent OpenMP[10] (*Open Multi Processing*) comme "standard industriel". OpenMP est "un modèle multitâches dont le mode de communication entre les tâches est implicite (la gestion des communications est à la charge du compilateur)" contrairement aux bibliothèques d'échanges de messages telles que PVM, MPI...

[9] Extrait de : `http ://www.idris.fr/data/cours/parallel/openmp/`
[10] `http ://www.openmp.org/`

Principes de fonctionnement : l'introduction de directives OpenMP est à la charge du développeur. Le programme OpenMP est exécuté par un processus unique qui peut activer des processus légers (*threads*) à l'entrée d'une région parallèle - sur un modèle type "*fork and join*". Chaque *thread* exécute alors une tâche composée d'un ensemble d'instructions avant de se terminer pour retrouver la "région séquentielle" (on peut aussi parfois introduire la notion de synchronisation). Le partage du travail consiste essentiellement à : exécuter une boucle par répartition des itérations entre les tâches, exécuter plusieurs sections de code mais une seule par tâche, exécuter plusieurs occurences d'une même portion de code par différentes tâches...

Ce standard possède une interface pour les langages Fortran, C et C++. Une implémentation expérimentale a été écrite en *Java*™ par des chercheurs de l'université d'Edinbourgh : JOMP[11]

4.3.1 Premier exemple : `HelloWorld`

Pendant l'exécution d'une tâche, une variable peut-être consultée. Elle est soit définie dans l'espace mémoire du processus léger, on parle alors de variable privée (*private*), soit définie dans un espace mémoire partagé par tous les processus légers, on parle alors de variable partagée (*shared*). Ce premier exemple constitue un cas typique de réplication d'un même code sur différentes tâches. Contenu du fichier `HelloWorld.jomp` :

```
1  import jomp.runtime.*;
2
3  public class HelloWorld
4  {
5      public static void main(String [] arguments)
6      {
7          int rang,nbth,nbpe;
8
9          //omp parallel private(rang,nbth)
10         {
11             rang = OMP.getThreadNum();
12             nbth = OMP.getNumThreads();
13
14             System.out.println("thread " + rang + "/" + nbth);
15         }
16     }
17 }
```

Ce fichier `HelloWorld.jomp` est "pré-traité" :

```
# java -classpath jomp1.0b.jar jomp.compiler.Jomp HelloWorld
Jomp Version 1.0.beta.
Compiling class HelloWorld....
Parallel Directive Encountered
```

... c'est ce qui permet de générer automatiquement un fichier `HelloWorld.java` :

```
1  import jomp.runtime.*;
2
3  public class HelloWorld {
4
5      public static void main(String [] arguments)
6      {
7          int rang,nbth,nbpe;
8
9  // OMP PARALLEL BLOCK BEGINS
10 {
11   __omp_Class0 __omp_Object0 = new __omp_Class0();
12   // shared variables
13   __omp_Object0.arguments = arguments;
14   // firstprivate variables
15   try {
16     jomp.runtime.OMP.doParallel(__omp_Object0);
17   } catch(Throwable __omp_exception) {
18     System.err.println("OMP Warning: Illegal thread exception ignored!");
19     System.err.println(__omp_exception);
20   }
21   // reduction variables
22   // shared variables
23   nbpe = __omp_Object0.nbpe;
24   arguments = __omp_Object0.arguments;
25 }
26 // OMP PARALLEL BLOCK ENDS
27
28     }
29
30 // OMP PARALLEL REGION INNER CLASS DEFINITION BEGINS
31 private static class __omp_Class0 extends jomp.runtime.BusyTask {
32   // shared variables
33   int nbpe;
34   String [] arguments;
35   // firstprivate variables
36   // variables to hold results of reduction
37
38   public void go(int __omp_me) throws Throwable {
39   // firstprivate variables + init
40   // private variables
41   int rang;
42   int nbth;
43   // reduction variables, init to default
44     // OMP USER CODE BEGINS
```

[11] `http ://www.epcc.ed.ac.uk/research/jomp/`

```
45
46              {
47                  rang = OMP.getThreadNum();
48                  nbth = OMP.getNumThreads();
49
50                  System.out.println("thread " + rang + "/" + nbth);
51              }
52          // OMP USER CODE ENDS
53        // call reducer
54        // output to _rd_ copy
55        if (jomp.runtime.OMP.getThreadNum(__omp_me) == 0) {
56        }
57        }
58      }
59      // OMP PARALLEL REGION INNER CLASS DEFINITION ENDS
60
61      }
```

```
# javac -classpath jomp1.0b.jar : HelloWorld.java
# java -native -Djomp.threads=5 -classpath jomp1.0b.jar : HelloWorld
thread 3 / 5
thread 1 / 5
thread 2 / 5
thread 4 / 5
thread 0 / 5
```

4.3.2 Second exemple : découpage de boucle dans le cas d'un produit de matrices

```
1   import jomp.runtime.*;
2   import java.util.Random;
3   import java.text.DecimalFormat;
4
5   public class MatriceCarreeMP
6   {
7       static Random aléa = new Random(0L);
8       static DecimalFormat leFormat = new DecimalFormat("###0.0000");
9
10      static void initialisation (double[] T,int n)
11      {
12          double nn = n*n;
13
14          for(int i=0; i<nn; i++)
15              T[i] = aléa.nextDouble();
16      }
17
18      static double trace(double[] T,int n)
19      {
20          int i;
21          double sommeTermesDiagonaux = 0;
22          for(i=0; i<n; i++)
23              sommeTermesDiagonaux += T[i*(n+1)];
24          return sommeTermesDiagonaux;
25      }
26
27      static void produitMatrice(double[] A,double[] B,double[] C,int n)
28      {
29          int i,j,k,in,kn;
30          double tmp;
31
32          //omp parallel private(j,k,in,kn,tmp) shared(A,B,C)
33          {
34              //omp for
35              for(i=0; i<n; i++) {
36                  in = i*n;
37                  for(j=0; j<n; j++) {
38                      tmp = 0;
39                      for(k=0, kn=0; k<n; k++, kn += n)
40                          tmp += A[in+k]*B[kn+j];
41                      C[in+j] = tmp;
42                  }
43              }
44          }
45      }
46
47      public static void main(String[] arguments)
48      {
49          int n = Integer.parseInt(arguments[0]);
50
51          double[] A = new double[n*n];
52          double[] B = new double[n*n];
53          double[] C = new double[n*n];
54
55            initialisation (A,n);
56            initialisation (B,n);
57
58          long dateDébut = System.currentTimeMillis(); // mesure du temps
59           produitMatrice(A,B,C,n);
60          long dateFin = System.currentTimeMillis(); // mesure du temps
61
62          System.out.println("trace(C) = " + leFormat.format(trace(C,n)));
63          System.err.println("Temps d'exécution = " + ((dateFin-dateDébut)/1000.) + "s");
64      }
65  }
```

```
# java -classpath jomp1.0b.jar jomp.compiler.Jomp MatriceCarreeMP
# javac -classpath jomp1.0b.jar : MatriceCarreeMP.java
# java -native -Djomp.threads=20 -classpath jomp1.0b.jar : MatriceCarreeMP 400
trace(C) = 40047.3507
```

```
Temps d'exécution = 1.242 s
# java -native -Djomp.threads=5 -classpath jomp1.0b.jar : MatriceCarreeMP 400
trace(C) = 40047.3507
Temps d'exécution = 1.897 s
# java -native -Djomp.threads=2 -classpath jomp1.0b.jar : MatriceCarreeMP 400
trace(C) = 40047.3507
Temps d'exécution = 3.497 s
# java -native -Djomp.threads=1 -classpath jomp1.0b.jar : MatriceCarreeMP 400
trace(C) = 40047.3507
Temps d'exécution = 6.308 s
```

Comparons avec la version séquentielle pure :

```
# grep -v omp MatriceCarreeMP.jomp > ! MatriceCarreeMP.java
# javac MatriceCarreeMP.java
# java MatriceCarreeMP 400
trace(C) = 40047.3507
Temps d'exécution = 4.354 s
```

► **Note :** pour l'échange de message explicite, on peut par exemple consulter le site : `http://aspen.csit.fsu.edu/pss/HPJava/mpiJava.html`.

Chapitre 5

Interfaces graphiques, programmation événementielle et manipulation d'images

Sommaire

5.1 Concepts principaux liés aux interfaces graphiques en *Java*™

Dans une interface graphique réalisée en *Java*™, on imbrique généralement trois niveaux d'objets graphiques.

► **Le conteneur principal** ou *top-level container* est le premier de ces niveaux. Il permet d'encapsuler toutes les entités des deux autres niveaux. Il peut s'agir soit d'une fenêtre, instance de *javax.swing.JFrame*, dans le cadre d'une application autonome, soit d'une boîte de dialogue, instance de *javax.swing.JDialog*, soit d'une instance de *javax.swing.JApplet* dans le cas d'une *applet*, soit enfin d'une instance de *javax.swing.JWindow* dans le cas d'une fenêtre sans bordure. Un conteneur est donc un objet qui dérive de la classe *java.awt.Container* mais qui ne dérive pas de sa classe fille *javax.swing.JComponent*.

► **Un composant-conteneur intermédiaire** ou *intermediate container* est, au titre de composant, un objet qui dérive non seulement de la classe *java.awt.Container* mais encore de sa classe fille *javax.swing.JComponent*. Un tel composant est en fait aussi un conteneur puisqu'il se charge de regrouper en son sein des composants "atomiques". Un objet instance des classes *javax.swing.JPanel*, *javax.swing.JScrollPane* ou *javax.swing.JTabbedPane* constitue, par exemple, un tel composant-conteneur intermédiaire.

► **Un composant "atomique"** ou *atomic component* hérite des propriétés de la classe *javax.swing.JComponent*. Il s'agit des éléments de base (que l'on appelle aussi *widgets* ou contrôles dans d'autres environnements de développements d'applications graphiques) tels que le bouton instance de *javax.swing.JButton*, la case à cocher instance de *javax.swing.JCheckBox*, le bouton radio instance de *javax.swing.JRadioButton*, la zone de saisie de texte sur une ligne (instance de *javax.swing.JTextField*) ou sur plusieurs (instance de *javax.swing.JTextArea*), la liste de choix déroulant (instance de *javax.swing.JComboBox*), l'étiquette (instance de *javax.swing.JLabel*)… Ces composants sont ajoutés (ou insérés) dans le conteneur courant en invoquant la méthode *add* de la classe *Container*.

```
1  import javax.swing.*;
2
3  class ConteneurEtComposant
4  {
5      ConteneurEtComposant()
6      {
7          // on commence par instancier le conteneur principal (il s'agit d'une
8          // fenêtre instance de JFrame) :
9          JFrame laFenêtre = new JFrame("Le titre de la fenêtre");
10         // on instancie ensuite le composant-conteneur intermédiaire (il
11         // s'agit d'une instance de JPanel) :
12         JPanel leConteneurIntermédiaire = new JPanel();
13         // on lie ce conteneur intermédiaire à la fenêtre courante :
14         laFenêtre.setContentPane( leConteneurIntermédiaire );
15         // on insère, par le biais de la méthode add, deux composants
16         // "atomiques" dans "leConteneurIntermédiaire" :
17         leConteneurIntermédiaire.add(new JButton("Voici un bouton"));
18         leConteneurIntermédiaire.add(new JLabel("Voici une étiquette "));
19         // on termine le constructeur en provoquant l'affichage de la fenêtre
20         // et de son contenu :
21         laFenêtre.pack();
22         laFenêtre. setVisible (true);
23     }
24
25     public static void main ( String [] arguments)
26     {
27         // on lance l'interface en instanciant la classe ConteneurEtComposant :
28         new ConteneurEtComposant();
29     }
30 }
```

5.1.1 Positionnement des composants

Pour contrôler le placement et la taille des composants au sein de la zone d'affichage d'un conteneur, *Java*™ propose d'utiliser un gestionnaire de placement. Celui-ci permet de mettre au point une politique de positionnement de chaque composant dans le conteneur courant.

► **Un gestionnaire de placement** ou *layout manager* est donc un objet associé au conteneur courant par le biais de la méthode *setLayout*, qui se charge de disposer les composants en vue de leur affichage. L'*API* graphique du langage propose, de base, plusieurs gestionnaires de placement allant du plus simple avec les gestionnaires *java.awt.FlowLayout* ou *java.awt.GridLayout*, au plus souple (moins immédiat en ce qui concerne leur mise en œuvre) avec les gestionnaires *java.awt.GridBagLayout* ou *javax.swing.BoxLayout*, en passant par des gestionnaires spécifiques tels que *java.awt.BorderLayout* ou *java.awt.CardLayout*.

Le gestionnaire *BorderLayout* est le gestionnaire par défaut des conteneurs principaux. Il permet de positionner les composants en fonction des cinq critères suivants : *BorderLayout.NORTH* (en haut), *BorderLayout.EAST* (à droite), *BorderLayout.SOUTH* (en bas), *BorderLayout.WEST* (à gauche) et *BorderLayout.CENTER* (au centre). La fenêtre de la figure 5.1 a été obtenue en utilisant un tel gestionnaire de placement.

FIG. 5.1 – Rendu visuel de la classe `PositionnementDesComposants` en utilisant un gestionnaire de placement de composants de type *BorderLayout*.

Le gestionnaire *BoxLayout* permet de placer les composants en ligne (avec la contrainte *BoxLayout.X_AXIS*) ou en colonne (avec la contrainte *BoxLayout.Y_AXIS*). Chaque composant garde sa taille par défaut, c'est le plus grand d'entre eux qui fixe la taille de la fenêtre d'affichage du conteneur. Ce gestionnaire propose un certain nombre de fonctionnalités permettant de fixer l'alignement respectif des composants et leur comportement lors de l'agrandissement du conteneur. La fenêtre de la figure 5.2 a été obtenue en utilisant un tel gestionnaire de placement.

Le gestionnaire *FlowLayout* est le gestionnaire par défaut du composant-conteneur intermédiaire de type *JPanel*. Ce gestionnaire se contente de disposer chaque composant, en ligne, de gauche à droite, en fonction de leur ordre d'insertion, en passant à la ligne suivante si nécessaire. La fenêtre de la figure 5.3 a été obtenue en utilisant un tel gestionnaire de placement.

Le gestionnaire *GridLayout* aligne la taille de chaque composant du conteneur sur celle du plus grand d'entre eux et les dispose tous dans une grille dont le nombre de colonnes et le nombre de lignes sont fixés à l'instanciation du gestionnaire. La fenêtre de la figure 5.4 a été obtenue en utilisant un tel gestionnaire de placement.

FIG. 5.2 – Rendu visuel de la classe `PositionnementDesComposants` en utilisant un gestionnaire de placement de composants de type *BoxLayout* selon l'axe des ordonnées.

FIG. 5.3 – Rendu visuel de la classe `PositionnementDesComposants` en utilisant un gestionnaire de placement de composants de type *FlowLayout*.

Le gestionnaire *GridBagLayout* est le plus évolué des gestionnaires de placement que nous présentons. Il permet de positionner des composants sur une grille virtuelle et offre pour cela un certain nombre de possibilités : un composant peut s'étaler sur plusieurs cellules, les rangées et les colonnes de composants peuvent être de taille variable... Pour ce faire, il faut utiliser la méthode *setConstraints* du gestionnaire courant et un objet de la classe *GridBagConstraints* qui permet d'expliciter l'ensemble des contraintes à appliquer lors du placement d'un composant donné. Nous utilisons un gestionnaire *GridBagLayout* au sein de la classe `MiniCalculatrice` de la section 5.2.1. La fenêtre de la figure 5.7 a été obtenue en utilisant un tel gestionnaire de placement.

Le gestionnaire *CardLayout* se distingue des gestionnaires qui précèdent dans la mesure où il permet de positionner différents ensembles de composants au sein d'un même conteneur, de manière à ce qu'un seul de ces ensembles soit visible à un moment donné. Il offre ainsi une fonctionnalité relativement similaire à celle du composant-conteneur intermédiaire *JTabbedPane* qui propose, au sein d'une même fenêtre d'affichage de conteneur, de varier les présentations de composants en fonction d'un système à base d'onglets.

```
1  import javax.swing.*;
2  import java.awt.*;
3
4  class PositionnementDesComposants
5  {
6      PositionnementDesComposants(int choix)
7      {
8          JFrame laFenêtre = new JFrame("Placement de composants");
9          JPanel leConteneurIntermédiaire = new JPanel();
10         laFenêtre.setContentPane( leConteneurIntermédiaire );
11         JButton un = new JButton("un"), deux = new JButton("deux"),
12             trois = new JButton("trois"), quatre = new JButton("quatre"),
13             cinq = new JButton("cinq");
14
15         switch (choix) {
16         case 0:
17             // choix du gestionnaire de placement par défaut (FlowLayout dans
18             // le cas de composant-conteneur intermédiaire de type JPanels) :
19             leConteneurIntermédiaire.setLayout(new FlowLayout());
20             // ajout des composants à proprement parler :
21             leConteneurIntermédiaire.add(un);
22             leConteneurIntermédiaire.add(deux);
23             leConteneurIntermédiaire.add(trois);
24             leConteneurIntermédiaire.add(quatre);
25             leConteneurIntermédiaire.add(cinq);
26             break;
27         case 1:
28             // choix d'un gestionnaire de placement de type BorderLayout :
29             leConteneurIntermédiaire.setLayout(new BorderLayout());
30             // ajout des composants à proprement parler :
31             leConteneurIntermédiaire.add(un,BorderLayout.NORTH);
32             leConteneurIntermédiaire.add(deux,BorderLayout.WEST);
33             leConteneurIntermédiaire.add(trois,BorderLayout.CENTER);
34             leConteneurIntermédiaire.add(quatre,BorderLayout.EAST);
35             leConteneurIntermédiaire.add(cinq,BorderLayout.SOUTH);
36             break;
37         case 2:
38             // choix d'un gestionnaire de placement de type BoxLayout :
39             leConteneurIntermédiaire.
40                 setLayout(new BoxLayout(leConteneurIntermédiaire,BoxLayout.Y_AXIS));
41             un.setAlignmentX(Component.CENTER_ALIGNMENT);
42             deux.setAlignmentX(Component.LEFT_ALIGNMENT);
43             trois.setAlignmentX(Component.CENTER_ALIGNMENT);
```

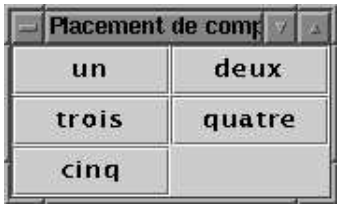


FIG. 5.4 – Rendu visuel de la classe `PositionnementDesComposants` en utilisant un gestionnaire de placement de composants de type *GridLayout* sur 3 rangées de 2 colonnes.

```
44                  quatre .setAlignmentX(Component.RIGHT_ALIGNMENT);
45                  cinq .setAlignmentX(Component.CENTER_ALIGNMENT);
46                  // ajout des composants à proprement parler :
47                  leConteneurIntermédiaire .add(un);
48                  leConteneurIntermédiaire .add(deux);
49                  leConteneurIntermédiaire .add( trois );
50                  leConteneurIntermédiaire .add(quatre );
51                  leConteneurIntermédiaire .add(cinq );
52                  break;
53              case 3:
54                  // choix d'un gestionnaire de placement de type GridLayout :
55                  leConteneurIntermédiaire .setLayout(new GridLayout(3,2));
56                  // ajout des composants à proprement parler :
57                  leConteneurIntermédiaire .add(un);
58                  leConteneurIntermédiaire .add(deux);
59                  leConteneurIntermédiaire .add( trois );
60                  leConteneurIntermédiaire .add(quatre );
61                  leConteneurIntermédiaire .add(cinq );
62                  break;
63              }
64              // affichage de la fenêtre :
65              laFenêtre .pack();
66              laFenêtre . setVisible (true);
67          }
68
69          public static void main ( String [] arguments)
70          {
71              // on lance l'interface en instanciant la classe courante :
72              new PositionnementDesComposants(Integer. parseInt (arguments [0]));
73          }
74      }
```

5.1.2 Gestion des événements

À chaque clic souris, à chaque frappe de caractère, un événement se produit, c'est-à-dire en fait qu'un "signal" est émis par un objet source (un composant graphique généralement). Pour pouvoir le prendre en compte, il faut cependant associer un contrôleur d'événement à l'objet source, en invoquant l'une des méthodes *add...Listener*. Cette invocation de méthode a pour effet de déléguer la gestion de l'événement à un *Listener* ou un *Adapter*.

En fait, tout objet peut être informé d'un événement survenu à un composant, à condition qu'il s'enregistre expressément (avec l'une des méthodes *add...Listener*) comme étant à l'écoute des événements susceptibles de survenir sur le composant en question.

▶ **Un délégué ou *listener*** est donc une classe qui implémente une interface *Listener* (c'est-à-dire une interface qui dérive de *java.util.EventListener*) et qui prend ainsi en charge le traitement de l'événement en son sein. Dans le cas de la classe `ListenerDEvenement`, l'instruction :

```
leBouton.addMouseListener(this );
```

permet d'enregistrer l'objet courant comme délégué au traitement des clics souris sur le composant `leBouton`. Pour que cet objet puisse être assimilé complètement à un délégué, il lui faut cependant aussi implémenter l'interface *java.awt.event.MouseListener* et ses méthodes publiques *mouseClicked* (cette méthode est invoquée sur réception d'un clic souris), *mouseEntered* (cette méthode est invoquée lorsque le curseur de la souris entre dans le composant), *mouseExited* (cette méthode est invoquée lorsque le curseur de la souris sort du composant), *mousePressed* (cette méthode est invoquée lorsqu'un bouton de la souris est enfoncé) et *mouseReleased* (cette méthode est invoquée lorsqu'un bouton de la souris est relâché).

```
1   import javax.swing.*;
2   import java.awt.event.*;
3
4   class ListenerDEvenement implements MouseListener
5   {
6       JButton leBouton;
7       JLabel leLabelCompteur;
8       int leCompteur = 0;
9
10      ListenerDEvenement()
11      {
12          JFrame laFenêtre = new JFrame("Délégué ou Listener");
13          JPanel leConteneurIntermédiaire = new JPanel();
14          laFenêtre .setContentPane( leConteneurIntermédiaire );
15          // instanciation et ajout des deux composants :
16          leLabelCompteur = new JLabel("Aucun clic enregistré");
```

```
17              leConteneurIntermédiaire .add(leLabelCompteur);
18              leBouton = new JButton("Le bouton");
19              leConteneurIntermédiaire .add(leBouton);
20              // l'objet courant s'enregistre comme délégué au traitement des clics
21              // souris sur le composant "leBouton" :
22              leBouton.addMouseListener(this);
23              laFenêtre .pack();
24              laFenêtre . setVisible (true);
25          }
26
27          public void mouseClicked (MouseEvent événement)
28          {
29              leLabelCompteur.setText((++leCompteur < 2) ?
30                                      leCompteur + " clic enregistré " :
31                                      leCompteur + " clics enregistrés ");
32          }
33          public void mouseEntered (MouseEvent événement) {}
34          public void mouseExited (MouseEvent événement) {}
35          public void mousePressed (MouseEvent événement) {}
36          public void mouseReleased (MouseEvent événement) {}
37
38          public static void main ( String [] arguments)
39          {
40              new ListenerDEvenement();
41          }
42      }
```

FIG. 5.5 – Rendu visuel de la classe `ListenerDEvenement` après cinq clics souris sur le composant `leBouton`.

► **Un adaptateur ou *adapter*** est une classe qui hérite des classes abstraites *Adapter* et qui prend ainsi en charge le traitement de l'événement en son sein. Dans le cas de la classe `ListenerDEvenement`, nous avons dû impérativement, pour mettre en œuvre le délégué à la gestion des clics souris sur le composant `leBouton`, implémenter les cinq méthodes *mousePressed*, *mouseReleased*, *mouseEntered*, *mouseExited* et *mouseClicked* de l'interface *MouseListener*. Or, pour quatre de ces méthodes, nous n'avons défini aucune action associée, puisque seul l'événement réceptionné par *mouseClicked* nous intéresse. Pour éviter d'avoir à définir de telles méthodes, le langage *Java*™ et son *API* graphique prévoient d'associer à chaque interface *Listener* (ayant plus d'une méthode), une classe abstraite *Adapter* où toutes les méthodes de l'interface correspondante sont implémentées mais laissées vides de traitement. Ainsi, comme nous allons le voir avec la classe `AdaptateurDEvenement`, la classe abstraite *java.awt.event.MouseAdapter* implémente, pour gérer la souris, l'interface *java.awt.event.MouseListener* ; il nous suffit donc d'en hériter et de redéfinir le corps de la méthode *mouseClicked* pour obtenir un traitement équivalent à celui de la classe `ListenerDEvenement`.

```
1   import javax.swing.*;
2   import java.awt.event.*;
3
4   class AdaptateurDEvenement extends MouseAdapter
5   {
6       JButton leBouton;
7       JLabel leLabelCompteur;
8       int leCompteur = 0;
9
10      AdaptateurDEvenement()
11      {
12          JFrame laFenêtre = new JFrame("Adaptateur ou Adapter");
13          JPanel leConteneurIntermédiaire = new JPanel();
14          laFenêtre.setContentPane( leConteneurIntermédiaire );
15          // instanciation et ajout des deux composants :
16          leLabelCompteur = new JLabel("Aucun clic enregistré ");
17          leConteneurIntermédiaire.add(leLabelCompteur);
18          leBouton = new JButton("Le bouton");
19          leConteneurIntermédiaire.add(leBouton);
20          // on ajoute un contrôleur d'événements souris au bouton :
21          leBouton.addMouseListener(this);
22          laFenêtre.pack();
23          laFenêtre.setVisible (true);
24      }
25
26      public void mouseClicked (MouseEvent événement)
27      {
28          leLabelCompteur.setText((++leCompteur < 2) ?
29                                  leCompteur + " clic enregistré " :
30                                  leCompteur + " clics enregistrés ");
31      }
32
33      public static void main ( String [] arguments)
34      {
35          new AdaptateurDEvenement();
36      }
37  }
```

► **Adaptateur en classe interne :** Comme vous pouvez le constater, la classe `AdaptateurDEvenement` qui précède, hérite de la classe abstraite *MouseAdapter*. Comme il n'y a pas d'héritage multiple en *Java*™, il peut donc sembler impossible d'utiliser un tel adaptateur lorsque la classe courante dérive déjà d'une autre classe. Pour contourner cette

limite, il faut alors utiliser, par exemple, une classe interne qui prend en charge le caractère d'adaptateur d'événement. C'est exactement ce que nous réalisons au sein de la classe `AdaptateurInterne` qui, parce qu'elle hérite déjà de la classe *JFrame*, ne peut pas, en plus, dériver la classe abstraite *MouseAdapter*. Pour s'affranchir de cette restriction, nous avons donc choisi de créer la classe interne `AdaptateurEnClasseInterne` qui dérive de la classe *MouseAdapter* et redéfinit sa méthode *mouseClicked*. `AdaptateurEnClasseInterne` est instanciée au moment même où son instance est enregistrée comme adaptateur délégué au traitement des clics souris sur le composant `leBouton`.

```
1  import javax.swing.*;
2  import java.awt.event.*;
3
4  class AdaptateurInterne extends JFrame
5  {
6      JButton leBouton;
7      JLabel leLabelCompteur;
8      int leCompteur = 0;
9
10     // la classe interne "AdaptateurEnClasseInterne" est préposée à la gestion
11     // des événements souris survenus sur le composant "leBouton" :
12     class AdaptateurEnClasseInterne extends MouseAdapter
13     {
14         public void mouseClicked (MouseEvent événement)
15         {
16             leLabelCompteur.setText((++leCompteur < 2) ?
17                                     leCompteur + " clic enregistré " :
18                                     leCompteur + " clics enregistrés ");
19         }
20     }
21
22     AdaptateurInterne ()
23     {
24         super("Adaptateur ou Adapter");
25         JPanel leConteneurIntermédiaire = new JPanel();
26         setContentPane( leConteneurIntermédiaire );
27         // instanciation et ajout des deux composants :
28         leLabelCompteur = new JLabel("Aucun clic enregistré ");
29         leConteneurIntermédiaire.add(leLabelCompteur);
30         leBouton = new JButton("Le bouton");
31         leConteneurIntermédiaire.add(leBouton);
32         // on ajoute un contrôleur d'événements souris au bouton :
33         leBouton.addMouseListener(new AdaptateurEnClasseInterne());
34         pack();
35         setVisible (true);
36     }
37
38     public static void main ( String [] arguments)
39     {
40         new AdaptateurInterne ();
41     }
42 }
```

▶ **Adaptateur en classe interne anonyme :** La classe `AdaptateurEnClasseInterne`, parce qu'elle déporte le traitement des événements au sein d'une classe interne de type adaptateur, constitue un premier moyen pour concilier les effets de l'absence d'héritage multiple en *Java*™ et du traitement des exceptions par la voie d'un adaptateur. Comme il peut cependant sembler un peu lourd dans certains cas, ponctuels et limités, de déclarer une classe interne de type adaptateur délégué au traitement des événements, le langage propose l'alternative de la classe interne anonyme. C'est exactement ce que nous réalisons au sein de la classe `AdaptateurInterneAnonyme`, où le traitement circonstancié (c'est-à-dire le corps de la méthode *mouseClicked* de la classe interne anonyme qui hérite de la classe abstraite *MouseAdapter*) est défini au moment même de l'enregistrement comme adaptateur délégué au traitement des clics souris sur le composant `leBouton` :

```
1  import javax.swing.*;
2  import java.awt.event.*;
3
4  class AdaptateurInterneAnonyme extends JFrame
5  {
6      JButton leBouton;
7      JLabel leLabelCompteur;
8      int leCompteur = 0;
9
10     AdaptateurInterneAnonyme()
11     {
12         super("Adaptateur ou Adapter");
13         JPanel leConteneurIntermédiaire = new JPanel();
14         setContentPane( leConteneurIntermédiaire );
15         // instanciation et ajout des deux composants :
16         leLabelCompteur = new JLabel("Aucun clic enregistré ");
17         leConteneurIntermédiaire.add(leLabelCompteur);
18         leBouton = new JButton("Le bouton");
19         leConteneurIntermédiaire.add(leBouton);
20         // on ajoute un contrôleur d'événements souris au bouton :
21         leBouton.addMouseListener(new MouseAdapter() {
22             public void mouseClicked (MouseEvent événement)
23             {
24                 leLabelCompteur.setText((++leCompteur < 2) ?
25                                         leCompteur + " clic enregistré " :
26                                         leCompteur + " clics enregistrés ");
27             }
28         });
29         pack();
30         setVisible (true);
31     }
32
33     public static void main ( String [] arguments)
34     {
35         new AdaptateurInterneAnonyme();
36     }
37 }
```

► **Enregistrer un délégué ou un adaptateur :** On enregistre un adaptateur ou un délégué au traitement d'un événement sur un composant, à l'aide d'une des méthodes *addXXXListener*.

5.2 Quelques exemples...

5.2.1 Une mini-calculatrice

comme l'application est "relativement" conséquente, nous l'avons décomposée sous forme d'une imbrication de classes internes. Ainsi, la classe principale `MiniCalculatrice` englobe à la fois une classe interne `BarreDeMenus` qui se charge d'implémenter la barre de menus et une classe interne `PavéNumérique` qui se charge d'implémenter le pavé numérique et la zone d'affichage de la calculatrice. La classe interne `BarreDeMenus` se compose elle-même d'une nouvelle classe interne appelée `ChoixDuMenu` qui se charge d'instancier les choix de l'unique menu déroulant. Cette nouvelle classe interne permet de factoriser le code associé à la création d'un choix de menu et rend ainsi la lecture du code plus aisée. De son côté, la classe interne `PavéNumérique` englobe aussi une nouvelle classe interne appelée `Bouton` qui est préposée à la mise au point d'un bouton de la calculatrice.

D'un point de vue pratique, la classe principale `MiniCalculatrice`, qui implémente l'interface *java.awt.event.ActionListener*, comprend un constructeur qui se charge de la mise en place des composants graphiques dans la fenêtre, une méthode statique *main* qui lance l'application en instanciant cette classe et une méthode *actionPerformed*. Cette dernière, qui associe un comportement à chaque événement, qu'il soit issu des boutons du pavé numérique ou des choix du menu déroulant, réalise l'opération d'addition à proprement parler.

En ce qui concerne la programmation événementielle de notre application, nous avons centralisé le traitement des événements au niveau de la méthode *actionPerformed* de la classe englobante principale `MiniCalculatrice`, qu'il s'agisse des événements générés par la barre de menus (ou plus précisément par les objets instance de `ChoixDuMenu`) ou des événements générés par le pavé numérique (ou plus précisément par les objets instance de `Bouton`). Dans ces deux cas, nous enregistrons, avec la méthode *addActionListener*, la classe `MiniCalculatrice` comme déléguée au traitement des événements liés aux composants de types `ChoixDuMenu` ou `Bouton`.

```
1  import javax.swing.*;
2  import java.awt.*;
3  import java.awt.event.*;
4
5  /* la classe MiniCalculatrice englobe une première classe interne
6   * "BarreDeMenus" (chargée d'implémenter la barre de menus) et une
7   * seconde "PavéNumérique" (chargée d'implémenter le pavé numérique et
8   * la zone d'affichage de la calculatrice) : */
9  class MiniCalculatrice implements ActionListener
10 {
11     JFrame laFenêtre;
12     JLabel zoneDAffichage;
13     long opérandeGauche=0,opérandeDroit=0;
14     char clicPrécédent ='';
15     /* la classe interne "BarreDeMenus", qui dérive de la classe
16      * JMenuBar, englobe elle-même une classe interne "ChoixDuMenu"
17      * qui permet d'implémenter les trois choix de l'unique menu
18      * déroulant de notre application : */
19     class BarreDeMenus extends JMenuBar
20     {
21         class ChoixDuMenu extends JMenuItem
22         {
23             ChoixDuMenu(MiniCalculatrice xcalc, String étiquette, char raccourciClavier)
24             {
25                 super(étiquette, raccourciClavier);
26                 setActionCommand(String.valueOf(raccourciClavier));
27                 addActionListener(xcalc);
28             }
29         }
30
31         BarreDeMenus(MiniCalculatrice xcalc)
32         {
33             JMenu uniqueMenu = new JMenu("Application");
34             uniqueMenu.add(new ChoixDuMenu(xcalc,"Pour information",'i'));
35             uniqueMenu.add(new ChoixDuMenu(xcalc,"Ré-initialiser",'r'));
36             uniqueMenu.add(new ChoixDuMenu(xcalc,"Quitter",'q'));
37             add(uniqueMenu);
38         }
39     }
40     /* la classe interne "PavéNumérique", qui dérive de la classe
41      * JPanel, englobe elle-même une classe interne "Bouton", qui
42      * dérive de JButton et est chargée de l'implémentation de chacun
43      * des boutons de la calculatrice. */
44     class PavéNumérique extends JPanel
45     {
46         GridBagLayout grilleDePlacement;
47         GridBagConstraints contraintesDePlacement;
48         Font laFonte = new Font("Monospaced",Font.BOLD,36);
49
50         class Bouton extends JButton
51         {
52             Bouton(MiniCalculatrice xcalc,PavéNumérique leConteneur,
53                    String étiquette, int contrainteEnRangée)
54             {
55                 super(étiquette);
56                 setFont(laFonte);
57                 contraintesDePlacement.gridwidth = contrainteEnRangée;
58                 grilleDePlacement.setConstraints(this,contraintesDePlacement);
59                 leConteneur.add(this);
60                 setActionCommand(étiquette);
61                 addActionListener(xcalc);
62             }
63         }
```

```
64
65          PavéNumérique(MiniCalculatrice xcalc)
66          {
67              /* pour mettre en place le pavé numérique, nous avons
68               * utilisé un gestionnaire de placement de type
69               * GridBagLayout. Ce gestionnaire permet de disposer les
70               * boutons de la calculatrice et sa zone d'affichage sur
71               * une grille de 5 rangées de 3 colonnes (disposition
72               * habituelle d'une mini-calculatrice). Ce placement
73               * s'effectue en fonction d'une instance de
74               * GridBagConstraints, celle-ci est chargée d'exprimer les
75               * contraintes de positionnement de chaque composant au
76               * sein de la grille. */
77              grilleDePlacement = new GridBagLayout();
78              contraintesDePlacement = new GridBagConstraints();
79              // le gestionnaire de placement est rattaché au composant
80              // courant dérivé de JPanel :
81              setLayout( grilleDePlacement );
82              /* quand le composant (le bouton ou la zone d'affichage)
83               * est plus petit que l'emplacement de la grille qui lui
84               * est alloué, celui-ci est "requalibré" dans les deux
85               * dimensions pour occuper toute la place disponible : */
86              contraintesDePlacement.fill = GridBagConstraints.BOTH;
87              // instanciation de la zone d'affichage :
88              zoneDAffichage = new JLabel("0", JLabel.RIGHT);
89              zoneDAffichage.setFont(laFonte);
90              // la constante "GridBagConstraints.REMAINDER" permet de
91              // spécifier que le composant courant est le dernier de sa
92              // rangée (si elle est affectée à l'attribut "gridwidth"
93              // du moins) :
94              contraintesDePlacement.gridwidth = GridBagConstraints.REMAINDER;
95              grilleDePlacement.setConstraints(zoneDAffichage,contraintesDePlacement);
96              add(zoneDAffichage);
97              // instanciation des boutons :
98              new Bouton(xcalc, this, "7",1);
99              new Bouton(xcalc, this, "8",1);
100             new Bouton(xcalc, this, "9", GridBagConstraints.REMAINDER);
101
102             new Bouton(xcalc, this, "4",1);
103             new Bouton(xcalc, this, "5",1);
104             new Bouton(xcalc, this, "6", GridBagConstraints.REMAINDER);
105
106             new Bouton(xcalc, this, "1",1);
107             new Bouton(xcalc, this, "2",1);
108             new Bouton(xcalc, this, "3", GridBagConstraints.REMAINDER);
109
110             new Bouton(xcalc, this, "0",1);
111             new Bouton(xcalc, this, "=",1);
112             new Bouton(xcalc, this, "+", GridBagConstraints.REMAINDER);
113         }
114     }
115     // constructeur de la classe englobante :
116     MiniCalculatrice()
117     {
118         // on commence par créer la fenêtre et lui définir un
119         // comportement à la fermeture :
120         laFenêtre = new JFrame("XCalc");
121         laFenêtre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
122         // on instancie une barre de menus que l'on rattache à
123         // "laFenêtre" :
124         laFenêtre.setJMenuBar(new BarreDeMenus(this));
125         // ainsi qu'un "PavéNumérique" :
126         laFenêtre.setContentPane(new PavéNumérique(this));
127         // enfin, on procède à l'affichage de l'ensemble :
128         laFenêtre.pack();
129         laFenêtre.setVisible(true);
130     }
131     // méthode de gestion des événements survenus au cours de
132     // l'exécution de la classe MiniCalculatrice :
133     public void actionPerformed(ActionEvent événement)
134     {
135         char résultatDuClic = événement.getActionCommand().charAt(0);
136
137         switch ( résultatDuClic ) {
138         case 'i':
139             // création de la boîte de dialogue modale d'information :
140             JOptionPane leContenu =
141                 new JOptionPane("Il s'agit d'une Mini-calculatrice ;-)",
142                                 JOptionPane.INFORMATION_MESSAGE,
143                                 JOptionPane.DEFAULT_OPTION);
144             JDialog boîteDeDialogue =
145                 leContenu.createDialog(laFenêtre,"Pour information ...");
146             boîteDeDialogue.setVisible(true);
147             break;
148         case 'r':
149             // ré-initialisation de la calculatrice :
150             zoneDAffichage.setText("0");
151             opérandeGauche = 0;
152             opérandeDroit = 0;
153             clicPrécédent = ' ';
154             break;
155         case 'q':
156             // fin de l'application :
157             laFenêtre.setVisible(false);
158             laFenêtre.dispose();
159             System.exit(0);
160             break;
161         case '+':
162             if ((clicPrécédent != '=') && (clicPrécédent != '+')) {
163                 opérandeDroit = Long.parseLong(zoneDAffichage.getText());
164                 opérandeGauche += opérandeDroit;
165                 zoneDAffichage.setText(String.valueOf(opérandeGauche));
166             }
167             break;
168         case '=':
169             if (clicPrécédent == '+') {
170                 long tmp = opérandeGauche;
171                 opérandeGauche += opérandeDroit;
172                 opérandeDroit = tmp;
173             }
174             else if (clicPrécédent == '=')
175                 opérandeGauche += opérandeDroit;
176             else {
177                 opérandeDroit = Long.parseLong(zoneDAffichage.getText());
178                 opérandeGauche += opérandeDroit;
```

```
179                 }
180                 zoneDAffi chage. setText ( String . valueOf(opérandeGauche));
181                 break;
182             default :
183                 String  affi chage ;
184                 if  (( clicPrécédent  == '=') || (  clicPrécédent  == '+'))
185                     zoneDAffi chage. setText ("" +  résultatDuClic );
186                 else  {
187                     affi chage  = zoneDAffi chage. getText ();
188                     if  ( affi chage . equals ("0"))
189                         zoneDAffi chage. setText ("" +  résultatDuClic );
190                     else  if  ( affi chage . length () < 8)
191                         zoneDAffi chage. setText ( affi chage  +  résultatDuClic );
192                 }
193             }
194             clicPrécédent  =  résultatDuClic ;
195         }
196
197         public  static  void main ( String [] arguments) throws Exception
198         {
199             // on lance l'application en instanciant la classe englobante
200             // MiniCalculatrice :
201             new MiniCalculatrice ();
202         }
203     }
```

FIG. 5.6 – La boîte de dialogue modale de l'application.

Les deux copies d'écran des figures 5.6 et 5.7 présentent respectivement la boîte de dialogue modale de notre application d'une part et l'unique menu déroulant d'autre part. Dans les deux cas, nous pouvons constater les effets du gestionnaire de placement *GridBagLayout* utilisé pour la mise en forme des boutons du pavé numérique. Ainsi, tandis que le composant `zoneDAffichage` (instance de la classe *JPanel*) s'étale, seul, sur la largeur de 3 composants "normaux" de notre interface graphique, tous les autres composants (instances de la classe `Bouton` dérivée de *JButton*) n'occupent qu'une seule "case" de la grille virtuelle associée au gestionnaire *GridBagLayout*. Nous marquons les retours à la ligne pour le positionnement des composants en affectant la valeur *GridBagConstraints.REMAINDER* à l'attribut *gridwidth* de l'objet `contraintesDePlacement` (instance de *GridBagConstraints*), alors que celui-ci a traditionnellement la valeur 1.

5.2.2 Un client web

```
1   import java .awt .*;
2   import java .awt. event .*;
3   import javax .swing .*;
4   import javax .swing. event .*;
5   import java . net .*;
6   import java . io .*;
7
8   public  class  ClientHttp  extends JFrame
9   {
10      private  URL urlCourante = null ;
11      private  JEditorPane  uneArdoiseNonÉditable = null ;
12      private  ArdoiseDuClientHttp uneArdoiseDuClientHttp = null ;
13
14      ClientHttp ()
15      {
16          super("Mini-client  HTTP");
17          // comportement de la fenêtre à la fermeture :
18          addWindowListener(new WindowAdapter() {
19                  public  void  windowClosing(WindowEvent événement)
20                  {
21                      quitter ();
```

FIG. 5.7 – Le menu déroulant de l'interface graphique.

```
22                      }
23                  });
24              // l'"ardoise" d'affichage de la ressource donnée en URL :
25              uneArdoiseNonÉditable = new JEditorPane();
26              /* on insère l'"ardoise" d'affichage de la ressource dans un conteneur
27               * intermédiaire (et on attache ce dernier à la fenêtre courante) : */
28              uneArdoiseDuClientHttp = new ArdoiseDuClientHttp(this);
29              getContentPane(). add(uneArdoiseDuClientHttp);
30              pack();
31               setVisible (true);
32          }
33
34          ClientHttp ( String  urlCourante )
35          {
36              this ();
37              setUrlCourante( urlCourante );
38              uneArdoiseDuClientHttp.setUnChampDeSaisie(urlCourante);
39              chargerPage ();
40          }
41
42          public void setUrlCourante( String  urlCourante )
43          {
44              try {
45                  this . urlCourante = new URL(urlCourante);
46              }
47              catch ( IOException ex ) {
48                  JOptionPane.showMessageDialog(this,"Ressource " + urlCourante + "  inaccessible  !",
49                                                "Erreur au téléchargement",JOptionPane.ERROR_MESSAGE);
50              }
51          }
52
53          public void setUrlCourante(URL urlCourante)
54          {
55              this . urlCourante = urlCourante;
56          }
57
58          public JEditorPane getUneArdoiseNonÉditable()
59          {
60              return uneArdoiseNonÉditable;
61          }
62
63          public void chargerPage()
64          {
65              try {
66                  uneArdoiseNonÉditable.setPage( urlCourante );
67              }
68              catch ( IOException ex ) {
69                  JOptionPane.showMessageDialog(this,"Ressource " + urlCourante + "  inaccessible  !",
70                                                "Erreur au téléchargement",JOptionPane.ERROR_MESSAGE);
71              }
72          }
73
74          public void quitter ()
75          {
76               setVisible ( false );
77              dispose ();
78              System. exit (0);
79          }
80
81          public static void main(String [] arguments)
82          {
83              new ClientHttp(arguments[0]);
84          }
85      }
86
87      class ArdoiseDuClientHttp extends Panel
88      {
89          private JTextField unChampDeSaisie = null;
90          private DéléguéChangementURL gestionnaireDÉvénements = null;
91
```

```
92          ArdoiseDuClientHttp( ClientHttp  leClientHttp )
93          {
94              gestionnaireDÉvénements = new DéléguéChangementURL(leClientHttp,this);
95
96              GridBagLayout grilleDePlacement = new GridBagLayout();
97              GridBagConstraints  contraintesDePlacement = new GridBagConstraints();
98              /* le gestionnaire de placement est rattaché au composant courant
99               * dérivé de JPanel : */
100             setLayout( grilleDePlacement );
101             /* quand le composant (le bouton ou la zone d'affichage) est plus
102              * petit que l'emplacement de la grille qui lui est alloué, celui-ci
103              * est "requalibré" dans les deux dimensions pour occuper toute la
104              * place disponible : */
105             contraintesDePlacement. fill = GridBagConstraints.BOTH;
106             /* ajout de l'étiquette : */
107             JLabel uneÉtiquette = new JLabel(" Url :" );
108             contraintesDePlacement.gridwidth = 1;
109             grilleDePlacement. setConstraints ( uneÉtiquette , contraintesDePlacement );
110             add( uneÉtiquette );
111             /* la constante "GridBagConstraints.REMAINDER" permet de spécifier que
112              * le composant courant est le dernier de sa rangée (si elle est
113              * affectée à l'attribut "gridwidth" du moins) : */
114             unChampDeSaisie = new JTextField();
115             unChampDeSaisie.addActionListener(gestionnaireDÉvénements);
116             contraintesDePlacement.gridwidth = GridBagConstraints.REMAINDER;
117             grilleDePlacement. setConstraints (unChampDeSaisie,contraintesDePlacement);
118             add(unChampDeSaisie);
119             /* ajout de l'ardoise déroulante : */
120             JEditorPane uneArdoiseNonÉditable = leClientHttp .getUneArdoiseNonÉditable();
121             uneArdoiseNonÉditable. setPreferredSize (new Dimension(500,400));
122             uneArdoiseNonÉditable. setEditable ( false );
123             uneArdoiseNonÉditable.addHyperlinkListener(gestionnaireDÉvénements);
124             JScrollPane  ardoiseDéroulante = new JScrollPane(uneArdoiseNonÉditable);
125             ardoiseDéroulante . setPreferredSize (new Dimension(500,400));
126             contraintesDePlacement.gridwidth = GridBagConstraints.REMAINDER;
127             grilleDePlacement. setConstraints ( ardoiseDéroulante , contraintesDePlacement );
128             add( ardoiseDéroulante );
129         }
130
131         public void setUnChampDeSaisie(String uneUrl)
132         {
133             unChampDeSaisie.setText(uneUrl);
134         }
135     }
136
137     class DéléguéChangementURL implements ActionListener,HyperlinkListener
138     {
139         private ClientHttp  leClientHttp = null;
140         private ArdoiseDuClientHttp uneArdoiseDuClientHttp = null;
141
142         DéléguéChangementURL(ClientHttp leClientHttp,ArdoiseDuClientHttp uneArdoiseDuClientHttp)
143         {
144             this. leClientHttp = leClientHttp ;
145             this.uneArdoiseDuClientHttp = uneArdoiseDuClientHttp;
146         }
147
148         public void actionPerformed( ActionEvent événement)
149         {
150             leClientHttp . setUrlCourante (événement.getActionCommand());
151             leClientHttp .chargerPage ();
152         }
153
154         public void hyperlinkUpdate(HyperlinkEvent événement)
155         {
156             if (événement.getEventType() == HyperlinkEvent.EventType.ACTIVATED) {
157                 leClientHttp . setUrlCourante (événement.getURL());
158                 leClientHttp . chargerPage ();
159                 // mettre à jour le champ de saisie de l'url :
160                 uneArdoiseDuClientHttp.setUnChampDeSaisie(événement.getURL().toString());
161             }
162         }
163     }
```

5.2.3 Génération d'image

L'ensemble de Mandelbrot est l'ensemble des points du plan complexe d'affixes $C = R_c + i \times I_c$ tels que la suite des $(Z_n)_{n\in\mathbb{N}}$ avec $Z_0 = 0$ et $Z_{n+1} = Z_n \times Z_n + C$ ne diverge pas à l'infini[1].

```
1   import java. util . Properties ;
2   import java .awt.image.BufferedImage;
3   import java .awt.image. WritableRaster ;
4   import com.sun.image.codec.jpeg .*;
5   import java. io .*;
6
7   class MandelbrotSequentiel
8   {
9       /*
10       * Determiner l'ensemble des points C du plan complexe tels que la suite
11       * complexe des (Z(n)) converge, avec :
12       *       Z(0) = 0
13       *       Z(n+1) = Z(n)*Z(n) + C
14       */
15      private double xMin,xMax,yMin,yMax,xPas,yPas;
16      private int  xResolution , yResolution , profondeur;
17      private String  nomFichierImage;
18      private Properties  valeursInitiales ;
19      private BufferedImage monImage;
20      private WritableRaster  tamponImage;
21
22      MandelbrotSequentiel( String  nomDeFichierDInitialisation )
23      {
24          valeursInitiales =new Properties ();
25          try {
26              InputStream  entrée=
27                  new FileInputStream(new File( nomDeFichierDInitialisation ));
28              // chargement de la table des valeurs initiales depuis un fichier
29              // du disque :
30              valeursInitiales . load( entrée );
31              System.err . println ("Chargement du fichier d' initialisation terminé." );
32          } catch ( IOException e ) {
```

[1] C'est-à-dire que Z_n doit admettre une limite finie quand n tend vers l'infini.

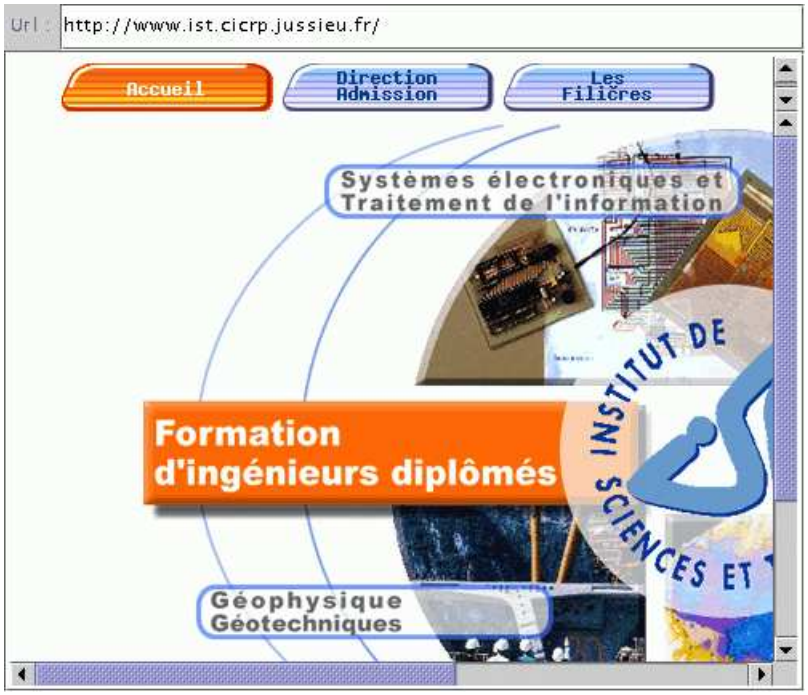


FIG. 5.8 – Mini-client web.

```
33                System.err . println ("Pas de fichier d' initialisation ." );
34            }
35            xMin = affecterDouble("xMin","-2.0");
36            xMax = affecterDouble("xMax","0.5");
37            yMin = affecterDouble("yMin","-1.25");
38            yMax = affecterDouble("yMax","1.25");
39            xResolution = affecterInt ("xResolution","300");
40            yResolution = affecterInt ("yResolution","300");
41            profondeur = affecterInt ("profondeur","55");
42            nomFichierImage = affecterString ("nomDuFichierImage","Mandelbrot.jpg");
43            // vérification du bon chargement des valeurs initiales :
44              valeursInitiales . list (System.err );
45            // affectation des dernières valeurs initiales :
46            xPas = (xMax - xMin) / xResolution;
47            yPas = (yMax - yMin) / yResolution;
48            monImage = new BufferedImage(xResolution,yResolution,
49                                         BufferedImage.TYPE_INT_RGB);
50            tamponImage = monImage.getRaster();
51        }
52
53        String affecterString (String clef, String valeurParDéfaut)
54        {
55            String tmp = valeursInitiales . getProperty (clef, valeurParDéfaut );
56              valeursInitiales . setProperty (clef ,tmp);
57            return tmp;
58        }
59
60        int affecterInt (String clef, String valeurParDéfaut)
61        {
62            int tmp;
63            try {
64                tmp = Integer . parseInt ( valeursInitiales .
65                                           getProperty ( clef , valeurParDéfaut ));
66            } catch (NumberFormatException e) {
67                System.err . println ("La valeur initiale pour l' attribut "
68                                      + clef + " doit être de type entier !" );
69                tmp = Integer . parseInt ( valeurParDéfaut );
70            }
71              valeursInitiales . setProperty ( clef , Integer . toString (tmp));
72            return tmp;
73        }
74
75        double affecterDouble (String clef, String valeurParDéfaut)
76        {
77            double tmp;
78            try {
79                tmp = Double.parseDouble( valeursInitiales .
80                                           getProperty ( clef , valeurParDéfaut ));
81            } catch (NumberFormatException e) {
82                System.err . println ("La valeur initiale pour l' attribut "
83                                      + clef + " doit être de type fbttant !" );
84                tmp = Double.parseDouble( valeurParDéfaut );
85            }
86              valeursInitiales . setProperty ( clef ,Double. toString (tmp));
87            return tmp;
88        }
89
90        void calculer ()
91        {
92            // La primitive de calcul de la fractale à proprement parler :
93            double rC,iC; // Parties reelles et imaginaires de C
94            double rZ,iZ; // Parties reelles et imaginaires de Z
95            double rZ2,iZ2,rZxiZ;
```

```
96              int iter , i , j ;
97              double couleur ;
98
99              for( j =0 , iC=yMin ; j<yResolution ; j ++ , iC+=yPas)
100                 for ( i =0 , rC=xMin ; i<xResolution ; i ++ , rC+=xPas) {
101                     // pour chaque point de la portion de plan complexe d'affixe
102                     // C=rC+i*iC, on calcule la suite des Zn tels que Z(n+1) =
103                     // Z(n)*Z(n)+C et Z(0)=0. En pratique, après initialisation de
104                     // Z à 0, on itère la boucle suivante un nombre de fois au
105                     // plus égal à la valeur du paramètre "profondeur" (la boucle
106                     // s'arrête entre-temps si la valeur du module de Z dépasse 2) :
107                     rZ = 0.0;
108                     iZ = 0.0;
109                     iter = 0;
110                     do {
111                         rZ2 = rZ * rZ;
112                         iZ2 = iZ * iZ;
113                         rZxiZ = rZ * iZ;
114                         rZ = rZ2 - iZ2 + rC;
115                         iZ = 2 * rZxiZ + iC;
116                         iter ++;
117                     } while (( iter < profondeur) && ((rZ2 + iZ2 ) <= 4));
118                     // la couleur du pixel correspondant est obtenue par rapport
119                     // du nombre d'itérations effectifs avant "divergence" sur le
120                     // nombre total fixé initialement d'itérations. Les
121                     // composantes rouge, verte et bleue ont toutes la même valeur
122                     // puisque l'on cherche à générer une image dans les dégradés
123                     // de gris.
124                     couleur = ( iter *255)/profondeur;
125                     tamponImage.setPixel( i , j , new double[]{couleur, couleur, couleur });
126                 }
127         }
128
129         void sauverImageRésultat () throws java . io . IOException
130         {
131             // cette primitive permet de sauver le tableau de pixels "tamponImage"
132             // sur un fichier du disque après encodage au format JPEG :
133             FileOutputStream sortie =new FileOutputStream(new File(nomFichierImage));
134             JPEGImageEncoder monEncodeur = JPEGCodec.createJPEGEncoder(sortie);
135             JPEGEncodeParam param = monEncodeur.getDefaultJPEGEncodeParam(monImage);
136             // l'encodage JPEG se fait sans perte de qualité avec un faible taux
137             // de compression (valeur 1.0f passée en premier argument dans la
138             // méthode setQuality) :
139             param. setQuality (1.0 f , false );
140             monEncodeur.setJPEGEncodeParam(param);
141
142             monEncodeur.encode(monImage);
143              sortie . close ();
144         }
145
146         void enregistrerLesPropriétés ( String nomDeFichierDInitialisation )
147         {
148             try {
149                 OutputStream sortie =
150                     new FileOutputStream(new File( nomDeFichierDInitialisation ));
151                 // enregistrement de la table des valeurs initiales dans un
152                 // fichier sur le disque :
153                  valeursInitiales .
154                     store ( sortie ," Valeurs initiales - ensemble de Mandelbrot");
155                 System.err . println ("Fin de l' enregistrement des valeurs initiales ." );
156             }
157             catch ( IOException e ) {
158                 System.err . println ("Problème à l' enregistrement des valeurs initiales ..." );
159             }
160         }
161
162         public static void main ( String [] arguments) throws java . io . IOException
163         {
164             long dateDébut = System. currentTimeMillis (); // mesure du temps
165             MandelbrotSequentiel bdpd = new MandelbrotSequentiel("Mandelbrot. ini " );
166             bdpd. calculer ();
167             bdpd.sauverImageRésultat ();
168             long dateFin = System. currentTimeMillis (); // mesure du temps
169             bdpd. enregistrerLesPropriétés ("Mandelbrot. ini " );
170             System.err . println ("Temps d'exécution = " + ( dateFin -dateDébut) + " ms");
171         }
172     }
```

Le constructeur de la classe `MandelbrotSequentiel` commence par charger la table des valeurs initiales depuis un fichier du disque : ce fichier précise la résolution demandée en abscisse et en ordonnée, les valeurs minimum et maximum des abscisses et ordonnées de la fenêtre d'étude et le nom du fichier image. Ce constructeur vérifie aussi le bon chargement des dites valeurs initiales depuis le fichier, puis calcule le pas de résolution utilisé et instancie la classe *java.awt.image.BufferedImage* pour stocker l'image après calcul.

La méthode `calculer` réalise le calcul de la fractale à proprement parler. Pour chaque point de la portion de plan complexe d'affixe $C = rC + i \times iC$ (rC et iC étant les parties réelles et imaginaires de C), on calcule la suite des Z_n tels que $Z(n+1) = Z(n) \times Z(n) + C$ et $Z(0) = 0$. En pratique, après initialisation de Z à 0, on itère la boucle suivante un nombre de fois au plus égal à la valeur du paramètre `profondeur` (la boucle s'arrête entre-temps si la valeur du module de Z dépasse 2). Puis, pour égayer un peu l'ensemble, on affecte une couleur à chaque point calculé. La couleur du pixel correspondant est obtenue par rapport du nombre d'itérations effectives avant "divergence" sur le nombre total maximal, fixé initialement, d'itérations. Les composantes rouge, verte et bleue ont toutes la même valeur puisque l'on cherche à générer une image dans les dégradés de gris.

En fait, l'ensemble des points à calculer est stocké dans un tableau `tamponImage` qui est une instance de *java.awt.image.WritableRaster*. Ce *Raster* est assimilable à un tableau de triplets de réels (les composantes de couleur RVB) de `xResolution` colonnes et `yResolution` lignes. L'appel de la méthode *setPixel* sur ce `tamponImage` permet, pour chaque pixel d'indice de colonne i et d'indice de ligne j, de stocker le code couleur du point d'affixe $rC + i \times iC$ (où $rC = xMin + i \times xPas$ et $Ic = yMin + j \times yPas$).

La méthode `sauverImageRésultat` permet de sauver le tableau de pixels `tamponImage` sur un fichier du disque après encodage au format JPEG.

Enfin, la méthode `enregistrerLesPropriétés` permet l'enregistrement de la table des valeurs initiales dans un fichier sur le disque.

Chapitre 6

L'*API Reflection* et la *Java™ 2 Enterprise Edition* platform (J2EE) - comparatif *Java™* et C++

Sommaire

En plus de la *Java™ 2 Standard Edition* (J2SE), *Sun Microsystems Inc.* propose une *Java™ 2 Enterprise Edition* (J2EE) et une *Java™ 2 Micro Edition* (J2ME - dédiée aux pagers, cellulaires et autres systèmes embarqués. . .).

Java™ 2 Enterprise Edition englobe les technologies suivantes : Enterprise JavaBeans[tm] (EJB) technology, java[tmp] Servlets, JavaServer Pages[tm] (JSP) technology, Java Remote Method Invocation (RMI), Java Naming and Directory Interface[tm] services (JNDI), JavaMail[tm] API, Java Database Connectivity (*JDBC™*) API, *Java™* Message Service (JMS), *Java™* Transaction (JTA) et *Java™* Transaction Service (JTS). . . D'autres API relativement spécifiques, comme la *Java™* Telephony API (JTAPI) ou le *Java™* Web Service Developer Pack (JWSDP) ne font actuellement partie d'aucun de ces trois "lots".

6.1 L'*API* Reflection - paquetage java.lang.reflect

Reflection ou plutôt introspection puisque cette librairie de classes permet de scruter le contenu d'objets et de classes en *Java™*, pour en déterminer la structure sans disposer des sources, à condition que le gestionnaire de sécurité (SecurityManager) le permette. Ce paquetage - qui date de la version 1.1 du *JDK* - offre la possibilité de :

- construire de nouvelles classes
- d'accéder et de modifier dynamiquement les attributs d'objets existants,
- d'invoquer des méthodes statiques ou dynamiques. . .

Intérêt de disposer d'un tel éventail de possibilités :

- les browsers de classes, les débogueurs, les interpréteurs et les décompilateurs, les mécanismes de sérialisation/désérialisation, doivent pouvoir accéder à la totalité des champs d'une classe ou d'un objet pour pouvoir les interfacer ou les présenter. . .
- les composants réutilisables (tels que les Beans) doivent pouvoir être scrutés de l'extérieur - notamment par des environnements de développement visuels - afin de présenter leurs propriétés statiques et dynamiques qui permettront à un développeur de les intégrer.

. . . cette *API* est utile pour tous les développements génériques en *Java™*.

```
1  class ScrutateurDeClasses
2  {
3      public static void main(String [] arguments) throws ClassNotFoundException
4      {
5          java.lang.reflect.Field [] lesVariables = null;
```

```
6               java.lang. reflect .Constructor [] lesConstructeurs = null;
7               java.lang. reflect .Method[] lesMéthodes = null;
8
9               java.lang.Class laClasse = Class.forName(arguments[0]);
10
11              try{
12                   lesVariables = laClasse. getDeclaredFields ();
13                   lesConstructeurs = laClasse. getDeclaredConstructors ();
14                  lesMéthodes = laClasse.getDeclaredMethods();
15              } catch (java.lang. SecurityException uneException) {
16                  System.err. println ("Problème d'accès aux propriétés de la classe :" + arguments[0]);
17                  System.exit (1);
18              }
19
20              for(int i =0 ; i< lesVariables. length ; i++)
21                  System.out. println ("Champ[" + i + "] == " + lesVariables [i ]);
22              for(int i =0 ; i< lesConstructeurs. length ; i++)
23                  System.out. println ("Constructeur [" + i + "] == " + lesConstructeurs [i ]);
24              for(int i =0 ; i<lesMéthodes.length ; i++)
25                  System.out. println ("Méthode[" + i + "] == " + lesMéthodes[i ]);
26          }
27      }
```

```
# java ScrutateurDeClasses java.lang.Boolean
Champ[0] == public static final java.lang.Boolean java.lang.Boolean.TRUE
Champ[1] == public static final java.lang.Boolean java.lang.Boolean.FALSE
Champ[2] == public static final java.lang.Class java.lang.Boolean.TYPE
Champ[3] == private boolean java.lang.Boolean.value
Champ[4] == private static final long java.lang.Boolean.serialVersionUID
Constructeur[0] == public java.lang.Boolean(java.lang.String)
Constructeur[1] == public java.lang.Boolean(boolean)
Méthode[0] == public boolean java.lang.Boolean.booleanValue()
Méthode[1] == public boolean java.lang.Boolean.equals(java.lang.Object)
Méthode[2] == public static boolean java.lang.Boolean.getBoolean(java.lang.String)
Méthode[3] == public int java.lang.Boolean.hashCode()
Méthode[4] == private static boolean java.lang.Boolean.toBoolean(java.lang.String)
Méthode[5] == public java.lang.String java.lang.Boolean.toString()
Méthode[6] == public static java.lang.Boolean java.lang.Boolean.valueOf(java.lang.String)
```

6.2 L'interface native de *Java*™ (JNI)

de l'intégration épisodique de fonctions C à *Java*™ comme langage de coordination et composition de composants natifs. *Java*™ dispose d'*API* de programmation concurrente (*threads*), distribuée (RMI), de programmation d'interfaces graphiques… le C est performant, bas niveau… L'*API* JNI existe depuis la version 1.1 du langage (elle assure une relative compatibilité ascendante avec l'*API* Native Method Interface, NMI, de la version 1.0 du langage).

```
1   public class HelloWorldJNI
2   {
3       private native void Affi cherUnMessage(String leMessage);
4
5       static {
6           System.loadLibrary("RessourcesNatives");
7       }
8
9       public static void main(String [] arguments) {
10          new HelloWorldJNI().Affi cherUnMessage("Bonjour tout le monde !");
11      }
12  }
```

```
# javac HelloWorldJNI.java
# javah -jni HelloWorldJNI
```

Ces deux instructions permettent de générer le fichier `HelloWorldJNI.h` suivant :

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class HelloWorldJNI */

#ifndef _Included_HelloWorldJNI
#define _Included_HelloWorldJNI
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     HelloWorldJNI
 * Method:    AfficherUnMessage
 * Signature: (Ljava/lang/String;)V
 */
JNIEXPORT void JNICALL Java_HelloWorldJNI_Affi cherUnMessage
  (JNIEnv *, jobject, jstring);

#ifdef __cplusplus
```

```
}
#endif
#endif
```

Nous venons ainsi d'obtenir la signature de la méthode native :

```
JNIEXPORT void JNICALL Java_HelloWorldJNI_AfficherUnMessage
    (JNIEnv*,jobject,jstring);
 --> HelloWorldJNI : nom de la classe
 --> AfficherUnMessage : méthode native
 --> JNIEnv* : pointeur sur l'interface JNI (pour l'accès aux méthodes du JNI)
 --> jobject : l'objet courant instance de HelloWorldJNI (this)
```

Il reste alors à écrire le code C natif `RessourcesNatives.c` qui commence par inclure le fichier d'en-tête précédent :

```
#include <jni.h>
#include <stdio.h>
#include "HelloWorldJNI.h"

JNIEXPORT void JNICALL Java_HelloWorldJNI_AfficherUnMessage
(JNIEnv *environnement, jobject objetCourant, jstring leMessage)
{
  const char* msg = (*environnement)->GetStringUTFChars(environnement,leMessage,0);
  printf("%s\n",msg);
  (*environnement)->ReleaseStringUTFChars(environnement,leMessage,msg);
}
```

et à compiler ce fichier source comme librairie partagée sous le nom `RessourcesNatives.so` :

```
# gcc -shared -o RessourcesNatives.so RessourcesNatives.c
-I/usr/local/jdk1.2/include -I/usr/local/jdk1.2/include/linux
-D_REENTRANT -D_GNU_SOURCE
# java -Djava.library.path=. HelloWorldJNI
Bonjour tout le monde !
```

► **Attention :** les chemins indiqués dans la directive de compilation, pour la génération de la librairie partagée `RessourcesNatives.so`, sont à adapter à votre propre plate-forme.

► **Conclusion :**
- extrait de *The JavaTM Tutorial* (*Sun Microsystems Inc.*) : "The JNI is for programmers who must take advantage of platform-specific functionality outside of the Java Virtual Machine. Because of this, it is recommended that only experienced programmers should attempt to write native methods or use the Invocation API !",
- le JNI est un mécanisme complet et simple d'interfaçage bidirectionnel entre : deux modèles de programmation (l'impératif ou le modulaire et l'objet), deux modes d'exploitation (performances séquentielles ou vectorielles et convivialité de la vision haut niveau de *Java*™, comme par exemple celle du calcul distribué), deux classes d'application (calcul scientifique et environnements graphiques, réseaux, sécurité, base de données)...
- pour limiter le sur-coût du mécanisme d'appel et d'accès, le code natif n'est pas un objet mais une librairie chargée dynamiquement (différence avec CORBA et COM) localement par un bloc initialiseur statique,
- on appelle *callback* le fait d'invoquer une méthode *Java*™ depuis le code natif,
- attention : la cohabitation *Java*™ - natif (Fortran, C, C++, Assembleur...) peut créer des erreurs de très bas niveau dépendantes des installations et difficiles à prévoir (librairies non réentrantes en Fortran),
- attention : du point de vue de *Java*™, le code natif n'est pas vérifiable.

6.3 *JDBC*™

L'*API JDBC*™ ⇔ ensemble de classes et d'interfaces permettant, à partir d'une application écrite en *Java*™, de communiquer avec une gestionnaire de bases de données relationnelles quelconque, pour peu que l'on charge le

pilote adéquat[1]. Cette interface de programmation représente une alternative tout à fait intéressante aux solutions propriétaires puisqu'elle fournit un frontal d'accès homogène aux SGBDR[2].

JDBC™ est une *API* définie par strates. La couche apparente pour le programmeur est celle qui est représentée par l'ensemble des fonctionnalités du paquetage *java.sql*. Ces fonctionnalités de "haut niveau" nous permettent de nous affranchir des problèmes d'implémentation de bas niveau comme, par exemple, l'établissement d'une connexion avec le SGBDR. Les autres couches inférieures sont destinées à faciliter l'implémentation de pilotes. Ceux-ci, s'ils souhaitent respecter la version 2.0 de l'*API*, doivent implémenter une bonne quinzaine d'interfaces définies dans le paquetage *java.sql*. Dans le cas de la version 1.0 de l'*API*, il leur faut essentiellement implémenter les interfaces : *CallableStatement* (exécution de procédures stockées), *Connection* (ouverture d'une session), *DatabaseMetaData* (accès aux informations générales concernant la base de données elle-même), *Driver* (chargement du pilote), *PreparedStatement* (exécution de requêtes précompilées paramètrées), *ResultSet* (représentation du résultat d'une requête), *ResultSetMetaData* (accès aux types et attributs des champs d'un *ResultSet*) et *Statement* (enveloppe permettant de soumettre des requêtes SQL statiques).

▶ **Note** : l'API 3.0 (qui fait suite aux versions 1.2.1, 2.0 et 2.1) de JDBC fait partie de la version 1.4 du *Java™ 2 Standard Edition*. En plus du paquetage *java.sql*, elle introduit un nouveau paquetage *javax.sql* relatif au côté serveur. Par ailleurs, une nouvelle API complémentaire émerge, il s'agit de *Java™ Data Objects* (JDO) qui permet de s'affranchir du choix du SGBD et de manipuler les données de façon générique.

6.3.1 Le SQL - *Standard Query Language*

⇔ langage normalisé qui permet de passer des ordres à un gestionnaire de bases de données relationnelles. Il se compose de commandes de définition (ALTER, CREATE et DROP), de manipulation (DELETE, INSERT et UPDATE) et d'interrogation de données (SELECT). Toutes ces instructions ont une syntaxe relativement élémentaire. Nous allons vous présenter, en créant effectivement nos tables, celle du moteur *MySQL* (qui n'est pas compatible en totalité avec la norme SQL 92) car c'est du SGBDR dont nous disposons[3].

base de données ⇔ outil spécialisé de stockage d'unités d'informations ⇔ ensemble de tables composées elles-mêmes de rangées d'éléments (les enregistrements ou tuples) ayant un nombre de colonnes (les champs) fixé et identique pour tous les enregistrements de la table.

base de données de gestion d'un parc de voiliers est composée de trois tables :

```
mysql> CREATE TABLE bateaux (
    ->         identifiantBateau TINYINT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    ->         nomBateau VARCHAR(50) NOT NULL,
    ->         longueurHT FLOAT,
    ->         couchages SMALLINT,
    ->         tarifParPersonne FLOAT);

mysql> CREATE TABLE clients (
    ->         identifiantClient TINYINT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    ->         nomClient VARCHAR(50) NOT NULL,
    ->         prenomClient VARCHAR(50),
    ->         adresse VARCHAR(100),
    ->         telephone VARCHAR(15));

mysql> CREATE TABLE reservations (
    ->         identifiantBateau TINYINT NOT NULL,
    ->         identifiantClient TINYINT NOT NULL,
    ->         dateDeDebut TIMESTAMP(8) NOT NULL,
    ->         PRIMARY KEY(identifiantBateau,identifiantClient,DateDeDebut));
```

Dans la première table, le champ `identifiantBateau` est une clef primaire unique (c'est-à-dire un élément qui ne doit apparaître qu'une fois au plus dans la table et qui permet d'identifier précisément l'enregistrement concerné), mais c'est aussi un compteur entier auto-incrémentable (il suffit d'insérer la valeur `NULL` en guise de champ pour que le SGBDR lui affecte automatiquement une valeur adéquate). Les autres champs, relativement classiques, permettent respectivement de renseigner le nom du bateau, sa longueur hors tout, le nombre de places à bord et le tarif par personnes embarquées.

Dans la seconde table, le champ `identifiantClient` est aussi une clef primaire unique de type compteur auto-incrémentable. Il permet de distinguer un client donné. Les autres champs, toujours très classiques, permettent respectivement de stocker son nom, son prénom, son adresse et son numéro de téléphone.

[1] La liste des pilotes disponibles est consultable sur l'Internet à l'adresse `http ://industry.java.sun.com/products/jdbc/drivers`.
À la date du 18 février 2002, il y a 155 pilotes répertoriés sur le site.

[2] Systèmes de Gestion de Bases de Données Relationnelles.

[3] Ce SGBDR est téléchargeable à l'adresse `http ://www.mysql.com/`. Il est passé sous licence GPL (*GNU General Public License*) depuis sa version 3.23.19 !

La troisième table est plus spécifique dans la mesure où elle définit une clef primaire composite formée de l'`identifiantBateau`, de l'`identifiantClient` et d'un champ `DateDeDebut`. Cette table recense l'ensemble des locations de voiliers et permet par jointure avec les deux premières tables de retrouver, par exemple, le nom du client qui a loué le bateau à telle ou telle autre date.

Une fois les structures de données définies, nous pouvons maintenant les enrichir par des données. C'est exactement ce que réalisent les trois instructions SQL INSERT ci-après :

```
INSERT INTO bateaux VALUES
        (NULL,'Jeanneau Sun Fast 40','12','8','4500'),
        (NULL,'Beneteau Oceanis Clipper','11.5','8','4000'),
        (NULL,'Beneteau First 53F5','16','10','5000'),
        (NULL,'Jeanneau Sun Oddyssey','11','6','3500'),
        (NULL,'Jeanneau Sun 2000','11.5','6','4500'),
        (NULL,'Jeanneau Sun Fast 40','12','8','4500');

INSERT INTO clients VALUES
        (NULL,'Dupont','Emmanuel','Rue de la Paix','01.12.34.56.78'),
        (NULL,'Renaud','Pierre','Rue de Stockholm','06.12.34.56.78'),
        (NULL,'Martin','Fred','Avenue de la République','03.12.34.56.78'),
        (NULL,'Dupond','Patricia','Rue Pasteur','04.12.34.56.78'),
        (NULL,'Francis','Océane','Rue Benoît Guichon','03.09.87.65.43');

INSERT INTO reservations VALUES
        (1,1,2000-7-3),
        (1,3,2000-10-23),
        (2,2,2000-11-20),
        (5,5,2000-8-7),
        (6,3,2000-8-7),
        (6,4,2000-8-7),
        (4,2,2000-11-27);
```

Nous avons donc inséré respectivement 6 enregistrements dans la table `bateaux`, 5 dans la table `clients` et 7 dans la table `reservations`. Pour retrouver le nom, le prénom et le numéro de téléphone des clients qui ont utilisé, utilisent ou utiliseront l'un des voiliers "Jeanneau Sun Fast 40" de l'agence, nous pouvons procéder par requête SELECT de la manière suivante :

```
mysql> SELECT clients.nomClient,clients.prenomClient,clients.telephone
    ->         FROM clients,bateaux,reservations
    ->         WHERE bateaux.nomBateau LIKE 'Jeanneau Sun Fast 40'
    ->         AND bateaux.identifiantBateau = reservations.identifiantBateau
    ->         AND clients.identifiantClient = reservations.identifiantClient
    ->         GROUP BY clients.identifiantClient;
+-----------+-------------+----------------+
| nomClient | prenomClient | telephone      |
+-----------+-------------+----------------+
| Dupont    | Emmanuel     | 01.12.34.56.78 |
| Martin    | Fred         | 03.12.34.56.78 |
| Dupond    | Patricia     | 04.12.34.56.78 |
+-----------+-------------+----------------+
3 rows in set (0.00 sec)
```

6.3.2 Accès au SGBDR via *JDBC*™

Pour interroger notre gestionnaire de base de données MySQL, nous avons au préalable téléchargé le pilote *mm.mysql.jdbc-2.0pre4* sur le site Web :

`http://www.worldserver.com/mm.mysql/`.

Cette première étape est naturellement fonction de votre propre gestionnaire de bases de données et nous ne pouvons que vous inviter, une fois encore, à visiter le site qui répertorie l'ensemble des pilotes à l'adresse :

`http://industry.java.sun.com/products/jdbc/drivers`.

► **Une fois le pilote installé,** il s'agit de passer au développement de notre application à proprement parler. Pour ce faire, nous commençons par importer le paquetage *java.sql* en en-tête de notre classe (il ne faut pas importer le paquetage relatif au pilote, celui-ci est chargé dynamiquement à l'exécution en invoquant la méthode *newInstance* sur une instance de *java.lang.Class* obtenue par application au préalable de sa méthode de classe *forName*).

Le pilote étant chargé, il s'agit ensuite d'établir la connexion et d'ouvrir une session avec le SGBDR. Ceci est réalisé en invoquant la méthode statique *getConnection* sur la classe *DriverManager* et en lui passant en argument l'URL qui permet de localiser le SGBDR à contacter et la base de données à charger.

Cette étape étant réalisée, on instancie un objet de la classe *Statement* relatif à la session courante par invocation de la méthode *createStatement* sur l'objet *connexion* courant. On peut alors soumettre des requêtes de manipulation, de définition ou d'interrogation au SGBDR. C'est ce que réalise l'appel à la méthode *executeQuery* sur cette instance de *Statement*, qui place le résultat de la requête dans l'objet `résultats` instance de *ResultSet*. Avant de clore la

connexion, on affiche les tuples constitutifs du *ResultSet* `résultats`, en les parcourant un à un par invocation de la méthode *next*. C'est exactement ce que réalise la classe `LocationDeVoiliers` ci-après :

```
1  import java.sql.*;
2
3  public class LocationDeVoiliers
4  {
5      public static void main ( String [] arguments)
6      {
7          String nomClient,prenomClient,telephone;
8          try {
9              // chargement dynamique (à l'exécution et non à la compilation) du
10             // pilote MySQL :
11             Class.forName("org.gjt.mm.mysql.Driver").newInstance();
12             // établissement de la connexion avec le SGBDR :
13             Connection connexion =
14                 DriverManager.getConnection("jdbc:mysql://127.0.0.1:3306/voile","leduc","");
15             // établissement et soumission de la requête statique :
16             Statement déclaration = connexion.createStatement();
17             ResultSet résultats = déclaration.executeQuery(
18                 "SELECT clients.nomClient, clients.prenomClient, clients.telephone " +
19                 "FROM clients,bateaux, reservations " +
20                 "WHERE bateaux.nomBateau LIKE 'Jeanneau Sun Fast 40' " +
21                 "AND bateaux.identifiantBateau = reservations.identifiantBateau " +
22                 "AND clients.identifiantClient = reservations.identifiantClient " +
23                 "GROUP BY clients.identifiantClient ");
24             // lecture des résultats :
25             while ( résultats.next())
26                 {
27                     nomClient = résultats.getString("nomClient");
28                     prenomClient = résultats.getString("prenomClient");
29                     telephone = résultats.getString("telephone");
30                     System.out.println (nomClient.toUpperCase() + " " +
31                                         prenomClient + " : " + telephone);
32                 }
33             résultats.close();
34             déclaration.close();
35             connexion.close();
36         }
37         catch ( SQLException e) {
38         }
39         catch ( java.lang.Exception e) {
40         }
41     }
42 }
```

Enfin, après simple compilation de notre classe (sans enrichissement du CLASSPATH puisque le pilote est chargé dynamiquement à l'exécution), nous lançons l'exécution de notre classe en indiquant à la machine virtuelle, à l'aide de l'option *-classpath*, l'endroit (le répertoire) où se trouvent les classes du pilote :

```
# javac LocationDeVoiliers.java
# java -classpath /home/leduc/pilotes/ : LocationDeVoiliers
DUPONT Emmanuel : 01.12.34.56.78
MARTIN Fred : 03.12.34.56.78
DUPOND Patricia : 04.12.34.56.78
```

6.4 RMI - *Remote Method Invocation*

protocole de communication évolué pour permettre à des objets distants, résidant dans des machines virtuelles distinctes, de communiquer dans le cadre d'une application distribuée (ie une application qui s'exécute sur plusieurs machines reliées en réseau). Plusieurs tendances :

- CORBA (Common Object Request Broker Application) de l'Object Management Group,
- DCOM (Distributed Component Object Model) de Microsoft,
- RMI de *Sun Microsystems Inc.*.

Un programme qui utilise les RMI se décompose en un objet distant (qui propose plusieurs méthodes à l'invocation à distance), le serveur, et un client qui récupère une référence du serveur et peut activer alors ses méthodes de la même façon que pour un objet local. En fait, un objet client ne référence pas directement l'objet serveur distant mais une interface qu'implémente la classe de l'objet distant. Ceci permet de distinguer les services proposés "en local" des services proposés à distance, mais cela permet aussi de compiler le client sans disposer de l'implémentation de l'objet serveur.

Entre l'applicatif client et l'applicatif serveur, se trouvent empilées, dans l'ordre, une couche "amorce (stub) client" (qui se charge des mécanismes de sérialisation/désérialisation, d'obtention de référence de l'objet distant...), une couche "Remote Reference Layer" (qui se charge de l'aspect fonctionnel de l'invocation, de la persistance de la connexion...) et une couche transport ("Transport Layer", qui aiguille les invocations, suit les connexions...) qui repose sur TCP mais pourrait aussi utiliser UDP ou SSL ("Secure Socket Layer").

L'API RMI se compose de 5 paquetages : *java.rmi* (côté client), *java.rmi.activation*, *java.rmi.dgc* ("Distributed Garbage Collector"), *java.rmi.registry* (utilisation de services d'enregistrement), *java.rmi.server* (côté serveur).

Pour commencer, l'objet "distant" doit implémenter l'interface *java.rmi.Remote*.

```
1  import java.rmi.*;
2
3  public interface ServeurRMI extends Remote
4  {
5      public int maFonction(int n) throws RemoteException;
6  }
```

Implémentons le serveur :

```
1  import java.rmi.*;
2  import java.rmi.server.*;
3
4  public class ServeurRMIImplementation
5      extends UnicastRemoteObject implements ServeurRMI
6  {
7      public ServeurRMIImplementation(String idtf) throws RemoteException
8      {
9          try{
10             Naming.rebind(idtf, this);
11         }
12         catch (Exception uneException) {
13             System.err.println(uneException);
14         }
15     }
16
17     public int maFonction(int n) throws RemoteException
18     {
19         return n+1;
20     }
21
22     public static void main(String[] arguments) throws Exception
23     {
24         if (System.getSecurityManager() == null) {
25             System.setSecurityManager(new RMISecurityManager());
26         }
27         ServeurRMIImplementation leServeur
28             = new ServeurRMIImplementation("toto123");
29         System.out.println("ok c'est tout bon...");
30     }
31 }
```

Implémentons le client :

```
1  import java.rmi.*;
2
3  public class ClientRMI
4  {
5      public static void main(String arguments[])
6      {
7          try {
8              if (System.getSecurityManager() == null) {
9                  System.setSecurityManager(new RMISecurityManager());
10             }
11             ServeurRMI leServeur
12                 = (ServeurRMI) Naming.lookup("rmi://localhost/toto123");
13             System.out.println(leServeur.maFonction(13));
14         }
15         catch (Exception uneException) {
16             System.err.println(uneException);
17         }
18     }
19 }
```

Compilons et générons les amorces :

```
# javac ClientRMI.java ServeurRMI.java ServeurRMIImplementation.java
# rmic ServeurRMIImplementation
```

Cette dernière commande permet d'obtenir les amorces client et serveur :
`ServeurRMIImplementation_Skel.class` et
`ServeurRMIImplementation_Stub.class`.
Inscription de l'objet serveur dans le registre :

```
# rmiregistry &
```

Génération d'un fichier de gestion des accès (ici, on choisit de l'appeller `policy.all`) :

```
1  grant {
2    permission java.security.AllPermission "","";
3  };
```

Lancement du serveur :

```
# java -Djava.security.policy=policy.all ServeurRMIImplementation
ok c'est tout bon...
```

Lancement du client :

```
# java -Djava.security.policy=policy.all ClientRMI
14
```

6.5 Bref comparatif entre *Java*™ et C++

Java™	C++
Généralités	
sans préprocesseur	avec préprocesseur
sans opérateur **,**	avec opérateur **,**
sans fonction à nombre variable d'arguments	avec fonction à nombre variable d'arguments
avec étiquette sur le *break* et le *continue*	sans étiquette sur le *break* et le *continue*
sans *const*	avec *const*
sans *goto*	avec *goto*
sans variable globale	avec variable globale
À propos de l'objet	
langage objet "pur"	langage orienté objets
toute fonction est une méthode d'instance ou de classe	il peut-y avoir des fonctions non rattachées à une classe
sans héritage multiple	avec héritage multiple
sans type paramétré	avec types paramétrés (*templates*)
sans surcharge d'opérateurs	avec surcharge d'opérateurs
"liaison dynamique" de toutes les méthodes (sauf celles déclaréees *final*)	seules les *virtual* fonctions sont liées dynamiquement
À propos des types primitifs	
tout est objet sauf les types primitifs	il existe aussi des types *struct*, *union*, *enum*, tableaux…
les types primitifs sont portables (big-Endian)	les types primitifs sont "plateforme-dépendants"
caractère 16-bits Unicode	caractère 8-bits ASCII
avec type booléen	sans type booléen
toute conditionnelle est une expression booléenne	un résultat entier est assimilé à une expression booléenne
initialisation automatique	initialisation à la charge du programmeur
À propos des pointeurs et structures de données	
les objets sont manipulés par référence, mais il n'y a pas de manipulation explicite ni d'arithmétique de pointeur	il existe les opérateurs $*$, $->$ et &
les références sur les tableaux ne peuvent pas être manipulées comme des pointeurs	les tableaux peuvent être manipulés avec l'arithmétique sur les pointeurs
avec test automatique de débordement de tableau	sans test automatique de débordement de tableau
tableaux multidimensionnels pouvant être non réguliers (les lignes d'une "matrice" peuvent être de longueurs variables)	tableaux multidimensionnels réguliers dont la taille est fixée à la déclaration
objet *String*	les chaînes de caractères sont des tableaux de caractères terminées par un zéro
concaténation via l'opérateur $+$	concaténation via une primitive
sans *typedef*	avec *typedef*
En vrac	
API réseau et support du *multithreading* en natif	pas d'API réseau ni de support du *multithreading* en natif
avec *garbage collector*	sans *garbage collector*
pré-compilé puis interprété	compilé
portable	architecture dépendant