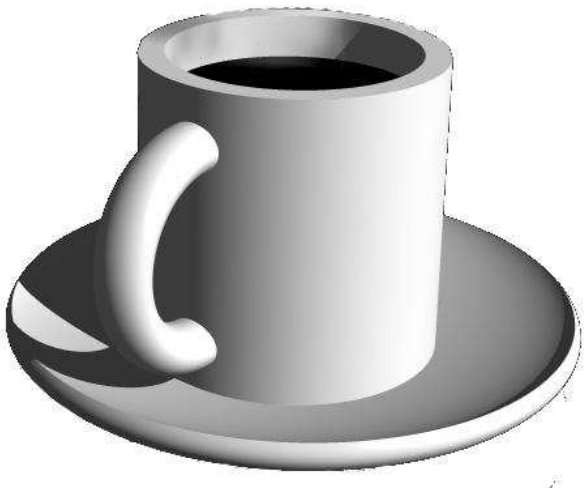
Lancelot Pecquet

Programmation Orientée Objet

Introduction illustrée en *Java*



Université de Poitiers, Licence d'Informatique
Version 1.1, du 19 janvier 2006

Qu'attendre de ce document ?

Ce document a pour objectif principal de faire découvrir, à un public débutant en la matière, les fondements de la programmation orientée objet. Le célèbre langage *Java* est utilisé pour illustrer les concepts théoriques. Ces quelques pages ne constituent aucunement un manuel de référence du langage *Java* et il conviendra de se reporter pour cela au manuel en ligne, disponible, par exemple, sur `http://java.sun.com/docs/`. Des lectures complémentaires pour approfondir les sujets abordés dans cette introduction sont suggérées dans la conclusion, p. 75.

Table des matières

Liste des tableaux

Table des figures

Chapitre 1

Introduction à la Programmation Orientée Objet

1.1 Qu'est-ce que la Programmation Orientée Objet (POO) ?

On classe souvent les langages de programmation en trois familles :

1. impératifs (comme les assembleurs, C, Perl, ...) ;
2. fonctionnels (comme lisp, scheme,...) ;
3. objets (comme C++, Java,...).

En fait, cette classification doit plus s'entendre comme des philosophies différentes dans l'approche de la programmation et les langages implémentent même souvent plusieurs de ces aspects. Ces trois philosophies sont complémentaires en fonction des situations.

1.2 Intérêt de la POO

Pour permettre à une entreprise de concevoir et développer un logiciel sur une durée de plusieurs années, il est nécessaire de structurer les différentes parties du logiciel de manière efficace. Pour cela, des règles précises de programmation sont établies, afin que les différents groupes de personnes intervenant sur un même logiciel puissent échanger leurs informations. En pratique, on a constaté que malgré l'instauration de règles, l'adaptation ou la réutilisation d'une partie programme nécessitait souvent une modification très importante d'une grande partie du code. La programmation orientée objet est donc apparue avec, pour objectifs principaux :

1. de concevoir l'organisation de grands projets informatiques autour d'entités précisément structurés, mêlant données et fonctions (les objets) facilitant la modélisation de concepts sophistiqués ;
2. d'améliorer la sûreté des logiciels en proposant un mécanisme simple et flexible des données sensibles de chaque objet en ne les rendant accessibles que par le truchement de certaines fonctions associées à l'objet (encapsulation) afin que celles-ci ne soient pas accessibles à un programmeur inattentif ou malveillant.
3. de simplifier la réutilisation de code en permettant l'extensibilité des objets existants (héritage) qui peuvent alors être manipulés avec les même fonctions (polymorphisme).

1.3 Vue informelle des concepts fondamentaux de la POO

1.3.1 Classes et méthodes

Une *classe* est une structure informatique regroupant les caractéristiques et les modes de fonctionnements d'une famille d'*objets*, appelés *instances* de la classe. Par exemple, une classe `voiture` aurait pour caractéristiques, (appelés *champs* en POO), ses dimensions, sa couleur, sa puissance, ...,

pour modes de fonctionnement (appelés *méthodes* en POO et notées suivies de parenthèses) des actions telles que `démarrer_moteur()`, `couper_moteur()`, `atteindre(v)` où `v` est la vitesse à laquelle on veut aller. La méthode `atteindre(v)` pourraît elle-même être décomposée en `débrayer()`, `embrayer()`, `accélérer()`, `freiner()`, `change_rapport(i)`, où `i` est le nouveau rapport choisi, *etc.* Chaque exemplaire sortant de l'usine serait alors une instance de la classe classe `voiture` et partagerait les même caractéristiques techniques.

1.3.2 Encapsulation

Si toutes les pièces électroniques et mécaniques servent au fonctionnement du véhicule, il n'est pas souhaitable, en général, que le conducteur puisse agir sur l'un d'entre-eux isolément car il risquerait de compromettre la sécurité de son véhicule (et par conséquent celle des autres usagers) en changeant les choses à l'aveuglette (par exemple en augmentant la puissance sans penser que les contraintes mécaniques résultantes risqueraient de provoquer la rupture de telle ou telle pièce qui n'avait pas été prévue pour cela).

Du point de vue du concepteur, l'action `démarrer_moteur()` peut être décomposée en des centaines de contrôles électromécaniques, puis en opérations de préchauffage des bougies, d'injection de carburant, d'allumage, de stabilisation du régime moteur au ralenti,... Ces opérations seront considérées comme *privées* pour le concepteur.

En revanche, du point de vue du conducteur, `démarrer_moteur()` consiste uniquement à mettre la clé de contact et la tourner et cette méthode sera considérée comme *publique*, autrement dit, utilisable par tout conducteur.

Le mécanisme de protection correspondant au fait que des champs et des méthodes privées ne soient accessibles par l'utilisateur d'une classe que par certaines méthodes publiques s'appelle l'*encapsulation*.

1.3.3 Héritage et polymorphisme

Toutes les voitures, bien qu'ayant des caractéristiques communes, se déclinent en plusieurs gammes : coupés, berlines, 4×4, cabriolets, *etc*, ayant chacune des caractéristiques communes. Par exemple, on imagine bien, dans la gamme des cabriolets, une méthode `met_toît(b)` où `b` sera un booléen indiquant si l'on veut que le toît (capote, hard-top), soit mis en place ou pas. On pourrait alors définir les classes `coupé`, `4x4`, `cabriolet`,... en tant que cas particulier de `voiture`, c'est-à-dire héritant de toutes les caractéristiques déjà définies : toutes ont les champs `dimensions`, `couleur`, `puissance`,... et les méthodes `démarrer_moteur()`, `couper_moteur()`... Pour la classe `cabriolet`, héritière de `voiture`, il s'agirait donc juste de définir un nouveau champ `toît`, qui vaudrait `vrai` ou `faux` selon que le toît est mis ou pas et la méthode `met_toît(b)` évoquée plus haut.

Le fait que les instances des classes héritières soient puissent être vues comme des instances de la classe mère de manière transparente s'appelle le *polymorphisme d'héritage*. Ainsi, si la la classe `voiture` définit une méthode `affiche_immatriculation()`, on pourra l'utiliser telle quelle pour n'importe quel véhicule 4×4.

Chapitre 2

Fonctionnement général de Java

2.1 Historique

La programmation orientée (POO) objet débute en Norvège, à la fin des années 1960. Le langage *Simula* est conçu pour simuler des événements discrets d'une manière plus pratique. Les années 1970 ont vu naître d'autres langages tels que *Smalltalk* (développé par *Xérox*, pour le premier système multi-fenêtrage avec écran bitmap et souris, repris plus tard par l'*Apple Macintosh*, puis copié par *Microsoft Windows*. Au cours des années 1980, avec l'arrivée des stations de travail et des ordinateurs personnels, l'intérêt pour les interfaces graphiques s'est accru et on a vu apparaître des langages « purement orienté objet » (*Eiffel*) et des extensions « orientées objet » de langage existants (*C++*, étendant le C, et des versions « objet » de *Pascal* (*Delphi*), *Prolog*, *Lisp*,...). Pendant les années 1990, de nouveaux langages orientés objet sont apparus. On citera, par exemple : *Java*, *Objective Caml*, *Python*.

En ce qui concerne plus précisément Java :

V0.x Développement de Java par l'équipe de Patrick Naughton et James Gosling, chez *Sun Microsystems* (1991) ;

V1.0 première version de Java (1995) ;

V1.02 gère la connectivité, les bases de données et les objets distribués (1996) ;

V1.1 gestionnaires d'événements, internationalisation, bibliothèque *Beans* (1997) ;

V1.2 améliorations et bibliothèque *Swing*(1998) ;

V1.3 améliorations (2000) ;

V1.5 améliorations (2004) : types énumérés, collections mieux gérées, arguments variables de méthodes

2.2 Avantages et inconvénients

Java est un langage qui présente beaucoup d'avantages :

1. **propre :** il préserve un typage assez fort des données en présence, les opérations arithmétiques sont très bien spécifiées ;
2. **interruptible :** il dispose d'un gestionnaire d'exceptions améliorant la robustesse et la lisibilité des programmes ;
3. **à ramasse-miette :** la mémoire est gérée automatiquement par le langage ;
4. **sécurisé :** de nombreux garde-fous (encapsulation, exceptions, gestionnaire de sécurité,...) contribuent à la robustesse des programmes, tant à la compilation qu'à l'exécution ;
5. **multithread :** des processus légers peuvent s'exécuter en parallèle, ce qui est particulièrement utile dans le cas de la programmation de serveurs ou de la parallélisation multiprocesseurs ;
6. **surchargeable :** les fonctions peuvent avoir le même nom et des arguments de type différent, ce qui permet de rapprocher la conception abstraite de l'implémentation ;

7. **portable :** grâce à l'existence d'une machine virtuelle et à une conception globale propice à la programmation graphique et événementielle utilisable sur toute plateforme et à l'existence d'une multitude de bibliothèques standard ;
8. **proche de C et C++ :** la syntaxe de Java ne désoriente pas les programmeurs habitués à ces langages ; l'interfaçage des programmes étant, par ailleurs, possible ;
9. **international :** les programmes Java supportent les normes internationales Unicode pour le codage des caractères de toutes les langues ;
10. **riche :** il existe de nombreuses bibliothèques dans tous les domaines ; celles-ci ont l'avantage considérable d'être standardisées ;
11. **interconnectable :** certains programmes Java (les *applets*) peuvent être directement intégrées dans les pages Web afin de les rendre dynamiques ; de nombreuses bibliothèques standard permettent de travailler en réseau ;
12. **populaire :** en particulier en entreprise, le langage Java est un investissement pour celui qui l'apprend.

Java souffre toutefois d'un certain nombre d'inconvénients, en particulier :

1. **lent :** même lors de l'utilisation d'un compilateur natif ;
2. **absence d'héritage multiple :** à la différence de C++, bien que cette carence soit compensée par un mécanisme d'interfaces ;
3. **pas de généricité :** à la différence de C++ (*templates*) et d'Objective Caml (classes paramétrées, polymorphisme complet), on dispose toutefois de la pseudo-généricité due à l'héritage.

2.3 Compilation et exécution des programmes Java

2.3.1 Machine Virtuelle Java

Les programmes Java ont été conçus pour être exécutables sur n'importe quelle architecture. Cela passe par le respect de norme strictes concernant la représentation des données comme des types primitifs portables, décrits dans la Tab. 3.1, p. 22 mais, malgré cela, la façon d'interagir avec un ordinateur ou un autre reste susceptible de différer, du fait de la multiplicité des processeurs et des systèmes d'exploitation. Le langage C, dans sa norme ISO C90 (ANSI), est portable sur un grand nombre d'architectures mais, afin que le destinataire puisse faire fonctionner le programme, il faut fournir :

1. soit le source du programme ;
2. soit une version exécutable par architecture supportée.

Dans le premier cas, le destinataire doit préalablement compiler le programme pour sa machine, en gérant tous les problèmes de compatibilité des bibliothèques standard (ou alors au prix d'une version alourdie si elle est compilée statiquement), ce qui peut prendre un certain temps et ne s'accomode pas des situations où les auteurs du code ne souhaitent pas révéler leurs sources[1]. Dans le second cas, il faut se limiter à quelques architectures standard car le nombre de possibilités est gigantesque.

Dans toutes les situations, certains aspects restent difficilement portables car les mécanismes qui gèrent ces événéments sont très différents d'un système à l'autre. C'est, en particulier, le cas de la programmation graphique.

Puisque la compilation directe vers les différents langages machines posait problème, l'idée des concepteurs de Java a été de proposer une **machine virtuelle java (JVM :*Java Virtual Machine*)**, c'est-à-dire un programme se comportant comme un « processeur logiciel » qui interprète des instructions portables en les traduisant dans le langage machine du processeur sur lequel le programme s'exécute.

[1] au risque de contrarier les partisans de l'OpenSource...

L'un des objectifs des concepteurs était de pouvoir proposer des petits programmes intégrés de manière transparente à des pages Web dynamiques, permettant à l'utilisateur d'interagir avec ces pages et ce, quelle que soit la plateforme sur laquelle le navigateur s'exécuterait. Ces programmes s'appellent les **applets** ; nous y reviendrons à la Section 12.5, p. 71.

2.3.2 Compilation

D'une manière générale, une *compilation*, en informatique, désigne la traduction d'un programme d'un langage dans un autre. On sous-entend souvent que le langage de destination de cette compilation est le langage machine de l'ordinateur sur lequel on va exécuter le programme. L'exécution d'un programme compilé consiste alors uniquement à déverser les instructions directement dans le processeur pour qu'il les applique.

En l'absence de compilation ou si celle-ci n'a pas eu lieu vers le langage machine, mais vers un autre langage, un *interprète* convertira chaque instruction en langage machine, au moment de l'exécution.

Le programme `javac` compile un programme `HelloWorld.java` en `HelloWorld.class` qui est un fichier contenant ce qu'on appelle du **code octet (*bytecode*)**. `HelloWorld.class` est constitué de code exécutable par une « *machine virtuelle* », en l'occurence, un programme appelé `java`. En invoquant `java HelloWorld`, les instructions du fichier `HelloWorld.class` seront, une par une traduites à destination du processeur de l'ordinateur sur lequel s'exécute le programme.

Un autre mode d'exécution peut avoir lieu si le programme `java` est associé à un **compilateur *Just In Time* (JIT)**. Dans ce cas, au lieu de faire d'interpréter tour à tour chaque instruction, le code-octet est compilé en code natif (exécutable par le processeur), puis ce code natif est exécuté, ce qui est souvent plus rapide que lors d'une interprétation.

2.3.3 Assembleur Java

Comme tout compilateur, `javac` utilise un langage intermédiaire : **l'assembleur Java** qui est l'équivalent, lisible par un être humain, du code-octet, lisible par la machine virtuelle Java. Le programme standard `javap` permet de désassembler un fichier `HelloWorld.class`, par exemple celui fabriqué par le source de la Section 2.4, p. 17, en invoquant : `javap HelloWorld.class`. Le résultat, au demeurant toujours assez éloigné d'un assembleur pour les processeurs classiques, est :

HelloWorld.ksm

```
// compiler version: 45.3

@source "HelloWorld.java"
@signature "Ljava/lang/Object;"
class HelloWorld extends java.lang.Object {

  /**
   * main
   *
   * stack 2
   * locals 1
   */
  @signature "([Ljava/lang/String;)V"
  public static void main(java.lang.String[]) {
    @line 3
    @getstatic java.io.PrintStream java.lang.System.out
    @const  "Hello World"
    @invokevirtual void java.io.PrintStream.println(java.lang.String)
    @return
  }

  /**
   * <init>
   *
   * stack 1
   * locals 1
```

```
   */
  @signature "()V"
  void <init>() {
    @line 1
    @aload  0
    @invokespecial void java.lang.Object.<init>()
    @return
  }
}
```

En fait, peut-être afin d'éviter la multiplication de fichiers contenant un code-octet ne respectant pas les même règles que ceux produits par `javac` et, par là même, potentiellement « dangereux », il n'y a pas d'assembleur livré avec la suite d'applications Java et *Sun* n'a même pas défini de standard officiel pour l'assembleur de sa machine virtuelle. On trouve toutefois plusieurs programmes d'assemblage comme, par exemple, *jasmin* [5]. Voici un programme `HelloWorld.j` que `jasmin` assemble en un fichier `HelloWorld.class` lisible par la JVM et qui affiche `Hello World` à l'écran.

HelloWorld.j

```
; --- Copyright Jonathan Meyer 1996. All rights reserved. -----------------
; File:      jasmin/examples/HelloWorld.j
; Author:    Jonathan Meyer, 10 July 1996
; Purpose:   Prints out "Hello World!"
; ------------------------------------------------------------------------

.class public examples/HelloWorld
.super java/lang/Object

;
; standard initializer
.method public <init>()V
   aload_0

   invokenonvirtual java/lang/Object/<init>()V
   return
.end method

.method public static main([Ljava/lang/String;)V
   .limit stack 2

   ; push System.out onto the stack
   getstatic java/lang/System/out Ljava/io/PrintStream;

   ; push a string onto the stack
   ldc "Hello World!"

   ; call the PrintStream.println() method.
   invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V

   ; done
   return
.end method
```

En deux mots, chaque *thread* dispose d'un exemplaire propre du *registre PC*, qui stocke l'adresse de l'instruction courante (indéfini si c'est une méthode native) dans une zone mémoire est réservée aux méthodes en cours d'exécution. Il peut lire et/ou écrire sur sa *pile (*stack*)*. L'interruption `StackOverFlowError` est levée en cas de débordement d'une pile, tandis que l'interruption `OutOfMemoryError` signale que la taille des piles ne peut plus être augmentée. Le *tas (heap)* est partagé par tous les *threads* et est parcouru par le ramasse-miette qui détermine les objets non-référencés à éliminer.

2.4 Hello World

Éditons un fichier `HelloWorld.java` contenant :

HelloWorld.java

```
class HelloWorld{
    public static void main(String[] args){
        System.out.println("Hello World");
    }
}
```

Évidemment, cela peut paraître un peu compliqué là où d'autres langages se contenteraient de `print "Hello World"` (*cf.* l'évolution du programmeur sur `http://www-rocq.inria.fr/~pecquet/jokes.html`) mais cela nous permet de voir, avec un exemple simple, le différentes caractéristiques du langage Java.

Le mot `class` introduit la classe `HelloWorld`. Cette classe ne contient pas de champ mais une unique méthode appelée `main`. Le compilateur, invoqué sur le fichier `HelloWorld.class`, cherchera, dans la classe `HelloWorld` une méthode main pour en faire la procédure principale.

La méthode `main` ne renvoie pas de valeur comme le mot `void` qui précède son nom l'indique. Elle a un argument (dont elle ne fait rien ici et qui serait instancié par une suite de paramètres passés en ligne de commande, le cas échéant), qui est un tableau nommé `args` de chaînes de caractères dont le type `String` est prédéfini en Java.

Le mot `public` signifie que cette méthode peut être invoquée de partout (ici, depuis la machine virtuelle Java). Le mot `static` signifie, quant à lui, que la méthode ne dépend pas d'une instance particulière de la classe mais est attachée à la classe elle-même.

La méthode `main` n'a qu'une instruction : elle invoque le méthode `println` du champ `out` de la classe `System` qui a pour effet l'impression sur la sortie standard de la chaîne passée en argument de `println`, c'est-à-dire `Hello World`, suivie d'un retour à la ligne (le « `ln` » de `println` sigifie *lign* : ligne).

On compilera avec la commande `javac HelloWorld.java` qui produit le fichier bytecode `HelloWorld.class`. Ensuite, il faut lancer la machine virtuelle à laquelle on donnera en argument le nom de ce fichier en bytecode afin qu'elle l'exécute. Cela se fait avec la commande `java HelloWorld` (attention, ne pas écrire `java HelloWorld.class`, sinon on obtient une erreur qui ressemble à ça :

```
java HelloWorld.java
java.lang.ClassNotFoundException: HelloWorld/java
        at kaffe.lang.SystemClassLoader.findClass0(SystemClassLoader.java:native)
        at kaffe.lang.SystemClassLoader.findClass(SystemClassLoader.java:150)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:149)
        at java.lang.Class.forName(Class.java:74)
        at java.lang.Class.forName(Class.java:43)
Caused by: <null cause>
```

2.5 Environnement de travail

Pour développer en Java sous Unix, on dispose de différents outils :

1. un mode standard pour *GNU emacs* gérant la colorisation des mots-clés et l'indentation automatique, comme on le voit dans la Fig. 2.1, p. 18. Il est mis en marche automatiquement dès le chargement d'un fichier dont le nom se termine par `.java`.
2. le projet GNU propose, en *open source*, un compilateurs et la machine virtuelle *kaffe* (`http://www.kaffe.org/`);
3. *Sun Microsystems* distribue des kits de développement sur son site web : `http://www.sun.com`.

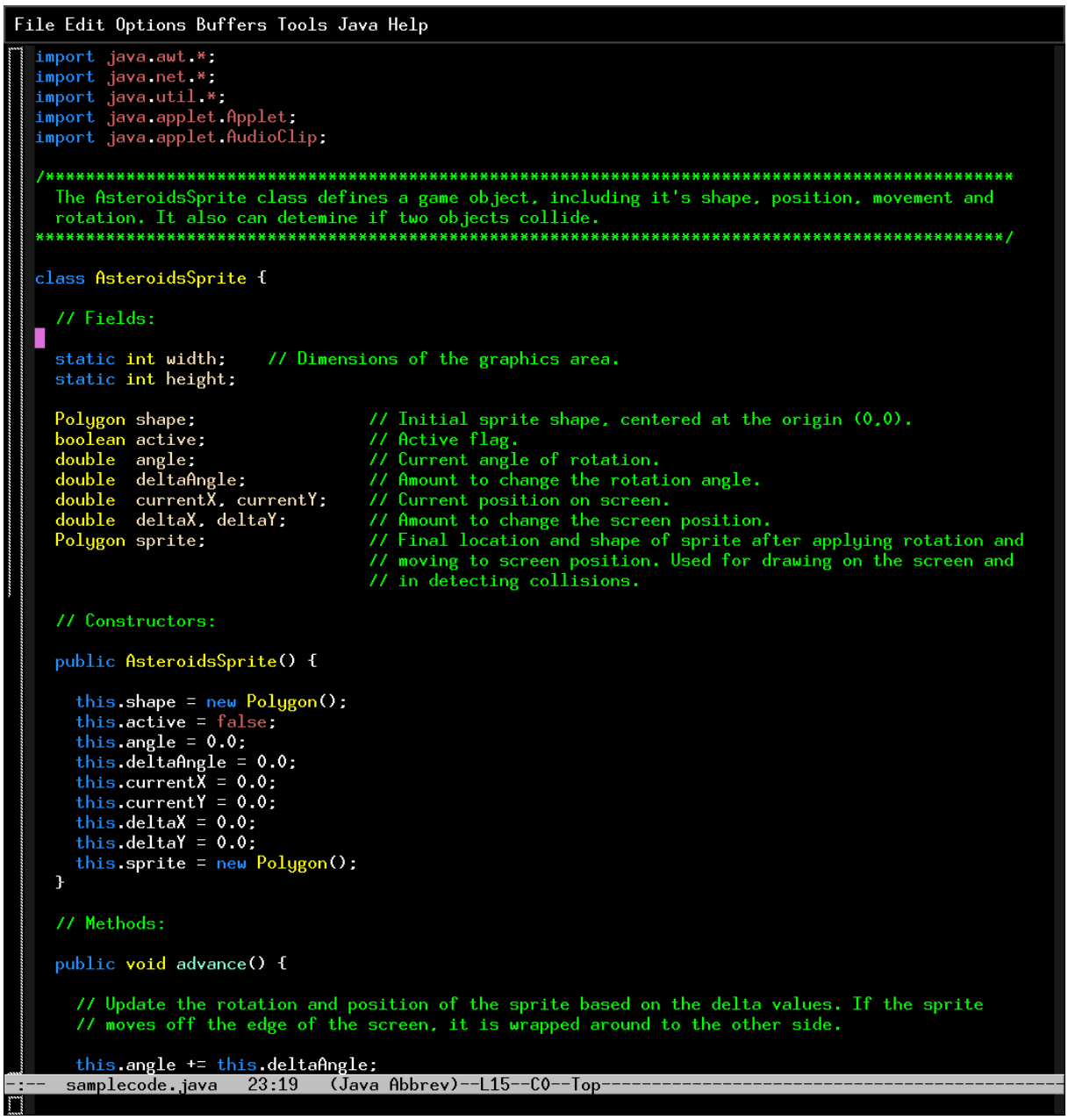


FIG. 2.1 – Le `java-mode` de *GNU emacs*.

2.6 Documentation en ligne

Pour connaître les caractéristiques techniques des différentes classes, interfaces, *etc*, on pourra se reporter à l'aide en ligne comme on peut la voir dans la Fig. 2.2.

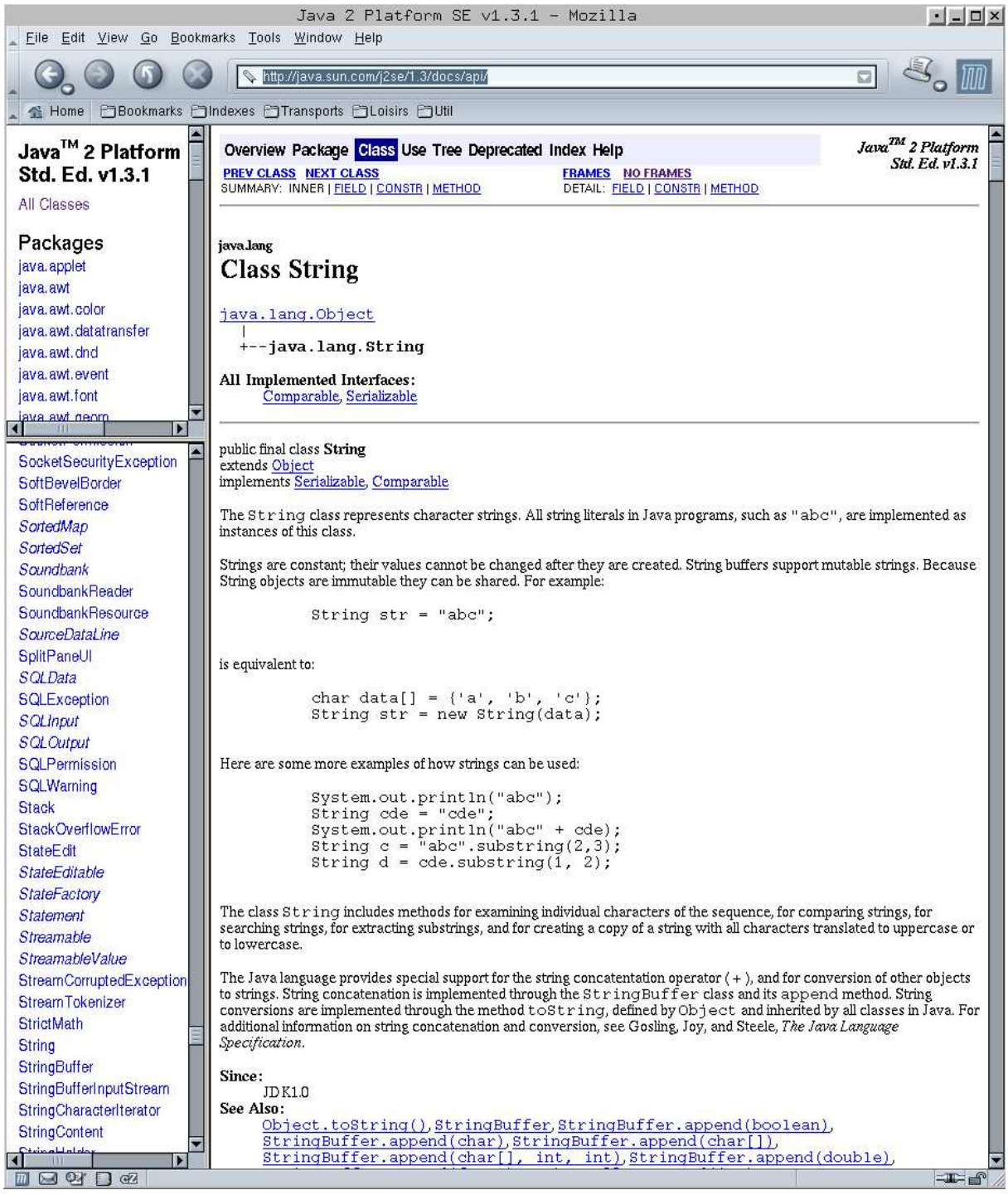


FIG. 2.2 – La documentation en ligne sur `http://java.sun.com/j2se/1.3/docs/api/`.

Chapitre 3

Éléments simples de Java

3.1 Commentaires

Comme dans tout langage, les zones de commentaires sont cruciales pour permettre la compréhension du code. On distinguera trois types de commentaires en java :

```
// les commentaires de fin de ligne

/* Les commentaires susceptibles de
s'étendre sur plusieurs lignes */

/** Les commentaires que le logiciel
javadoc pourra utiliser pour générer une
documentation HTML
*/
```

3.2 Types primitifs

Les types primitifs de Java sont, en quelque sorte, les « composants atomiques » avec lesquels on construit des classes en Java. Ceux-ci sont définis dans la Table 3.1 .

Les types non-primitifs sont désignés implicitement par leur référence en mémoire. Il s'agit des instances de classes, dont les tableaux de type primitif, ou non, font partie ainsi que nous le verrons plus tard. La valeur `null` désigne une référence qui ne correspond à rien.

En ce qui concerne les `float` et les `double`, certains processeurs (Intel) réalisant leurs opérations flottantes sur 80 bits et tronquant finalement le résultat sur 64 bits peuvent donner des résultats différents des processeurs réalisant toutes leurs opérations sur 64 bits. Par défaut, la précision maximale des processeurs est utilisée pour ne pas pénaliser la vitesse d'exécution mais en spécifiant qu'une méthode est `strictfp` (*strict floating point*), tous les calculs sont tronqués à 64 bits pour en garantir la portabilité, au prix d'une perte de précision et d'un ralentissement. Préfixer le nom d'une classe par `strictfp` a pour effet de préfixer implicitement toutes les méthodes de cette classe par `strictfp`.

Le standard Unicode [6] permet de représenter $1\,114\,112 = 2^{20} + 2^{16}$ caractères par différents codages à longueur variable. Actuellement 96 000 caractères sont effectivement définis[1]. Dans la norme UTF-16 utilisée par Java (et pas totalement supportée d'ailleurs), les 256 premiers caractères sont codés sur un octet et sont les caractères ISO-8859-1 (ASCII 8 bit, *Latin 1*, utilisé dans les pays occidentaux). Les caractère suivants codent le latin, le grec, le cyrillique, le copte, l'arménien, l'hébreu, les caractères arabes, puis des caractères chinois, japonais, coréens,...

La sémantique de l'affectation des types primitifs est la copie bit-à-bit et celle de l'égalité est l'égalité bit-à-bit. Les valeurs de type primitifs ne requièrent pas de demande de mémoire explicite avec le mot-clé `new` comme ce sera le cas pour les types référencés. Seule la déclaration de type suffit, comme par exemple :

[1] Dont environ 50 000 caractères chinois, quoique seuls 5000 soient usuels...

TAB. 3.1 – Types primitifs de Java. L'arithmétique entière est toujours réduite modulo et ne déborde jamais. Conformément au standard IEEE754, les flottants peuvent déborder et valoir `infinity`

Nom	représentation
`boolean`	1 bit
`char`	16 bits, Unicode UTF-16, 2.1
`byte`	8 bits, signé
`short`	16 bits, signé
`int`	32 bits, signé
`long`	64 bits, signé
`float`	32 bits, signé IEEE754/SP/1985
`double`	64 bits, signé IEEE754/DP/1985

```
int x = 1;
```

Quoiqu'il ne s'agisse plus de types primitifs, il est intéressant de savoir que Java dispose de bibliothèques de calcul en précision arbitraire : `java.Math.BigInteger` pour les entiers et `java.Math.BigDecimal` pour les nombres décimaux.

3.3 Opérateurs

On retrouve les opérateurs classiques des langages de programmation, essentiellement équivalent à ceux qu'on a en C, dans la Table 3.2

3.4 Affichage polymorphe

Java permet de donner à plusieurs méthodes le même nom, à condition que le type ou le nombre des arguments de ces méthodes soient différents. Ainsi, pour afficher une valeur x, on utilise la méthode `System.out.println(`x`)` qui est *surchargée*, comme le montre l'exemple suivant :

```
System.out.println(1);   // int
System.out.println(2.5); // double
System.out.println('A'); // char
```

3.5 Portée des identificateurs

Un identificateur est connu dans une zone délimitée par des accolades. Les variables locales sont prioritaires sur les variables globales, ce qui est le cas le plus souvent car c'est assez logique. En effet, dans le monde entier, on connaît Henry DUNANT comme le fondateur de la Croix-Rouge. Cependant, si, dans un village, le garagiste est un homonyme et s'appelle aussi Henry DUNANT, tout le monde comprendra que « faire réparer sa voiture chez Henry DUNANT » ne signifie pas qu'on fait remorquer le véhicule jusqu'au siège du Comité International de la Croix-Rouge à Genève...

```
class test{
  public static void main(String[] args){
    int x = 1;
    System.out.println(x);   // Affiche 1
    {
      double x = 2.5;
```

TAB. 3.2 – Les différents opérateurs de Java. Attention, l'affectation, l'égalité et la différence ont une sémantique de valeur pour les types primitifs mais une sémantique de référence pour les types référencés. Soit `i` une variable valant k, l'application de l'opérateur de pré-incrémentation `++i` incrémente (resp. pré-décrémentation `--i` décrémente) la valeur de `i` et renvoie la valeur après incrémentation, *i.e.* $k+1$ (resp. décrémentation, *i.e.* $k-1$), tandis que l'opérateur de post-incrémentation `i++` incrémente (resp. post-décrémentation `i--` décrémente) la valeur de `i` et renvoie la valeur avant incrémentation (resp. décrémentation), *i.e.* k. L'opérateur `=` est associatif à droite et renvoie la valeur de ses opérandes. Par exemple, si l'on exécute `int i,j ; i=j=5 ;`, l'instruction `j=5` est effectuée en premier et renvoie `5`.

syntaxe	sémantique
x `=` y	x prend la même valeur que celle de y (affectation)
x `==` y	vrai si $x = y$, faux sinon (test d'égalité)
x `!=` y	faux si $x = y$, vrai sinon (test de différence)
x `<` y	$x < y$ (inférieur)
x `>` y	$x < y$ (supérieur)
x `<=` y	$x \leq y$ (inférieur ou égal)
x `>=` y	$x \leq y$ (supérieur ou égal)
x `+` y	$x + y$ (addition)
x `-` y	$x - y$ (soustraction)
x `*` y	$x \times y$ (multiplication)
x `/` y	x/y (division)
x `%` y	$x \mod y$ (modulo)
`++`x	pré-incrémentation de x
`--`x	pré-décrémentation de x
x`++`	post-incrémentation de x
x`--`	post-décrémentation de x
x `+=` y	équivaut à x `=` $x + y$
x `*=` y	équivaut à x `=` $x * y$
x `-=` y	équivaut à x `=` $x - y$
x `/=` y	équivaut à x `=` x/y
x `%=` y	équivaut à x `=` $x \mod y$
x `&&` y	conjonction logique de booléen
x `||` y	disjonction logique de booléen
`!`x	négation logique d'un booléen
x `&` y	conjonction logique bit à bit d'entiers
x `|` y	conjonction logique bit à bit d'entiers
`~`x	négation logique bit à bit d'entier
`^`x	disjonction exclusive (xor) bit à bit d'entier
x`<<`n	décalage à gauche des bits de x (0 à droite)
x`>>`n	décalage à droite des bits de x (bit de signe propagé)
x`>>>`n	décalage à droite des bits de x (0 à gauche)
`(`T`)`x	transtype x vers le type T (si possible)

```
      double y = -1;
      System.out.println(x); // Affiche 2.5
      System.out.println(y); // Affiche -1
    }
    System.out.println(x);   // Affiche 1
    System.out.println(y);   // Provoque une erreur
 }
}
```

3.6 Instructions conditionnelles

3.6.1 Introduction

Afin de distinguer diverses situations, comme dans tous les langages, on dispose de l'instruction conditionnelle `if` et, pour certains types primitifs discrets, du `switch`. Un test d'assertion, utilisable avec l'instruction `assert`, facilite le débogage.

3.6.2 `if`

Le syntaxe de `if` est la même que celle du C :

`if(`*test*`){`
partie à exécuter si test *est vrai* ;
`}`

avec la variantes :

`if(`*test*`){`
partie à exécuter si test *est vrai* ;
`}else{`
partie à exécuter si test *est faux* ;
`}`

et :

`if(`$test_1$`){`
partie à exécuter si $test_1$ *est vrai* ;
`}else if(`$test_2$`){`
partie à exécuter si $test_1$ *est faux et* $test_2$ *est vrai* ;
$\vdots$
`}else if(`$test_n$`){`
partie à exécuter si $test_i$ *est faux pour* $1 \leq i < n$ *et si* $test_n$ *est vrai* ;
`}else{`
partie à exécuter si $test_i$ *est faux pour* $1 \leq i \leq n$;
`}`

3.6.3 `switch`

Pour une variable x dont le type est `byte`, `char`, `short` et `int`, Java dispose aussi de l'instruction `switch` :

```
switch(x){
    case a_1:
        partie à exécuter si x = a_1 ;
        break;
      ⋮
    case a_n:
        partie à exécuter si x = a_n ;
        break;
    default: // Optionnel mais fortement recommandé
        partie à exécuter si x ∉ {a_1, ..., a_n} ;
```

3.6.4 Assertions

Pour débugguer, on peut utiliser le mécanisme d'assertions, comme dans :

```
assert (x == 1);

// ou encore:

assert (x == 1): "x devrait être égal à 1";
```

La chaîne suivant le `:` est passée au constructeur d'une `AssertionError`. À compiler avec `javac -source 1.4 toto.java` car ce mot-clé apparaît avec la version 1.4. On peut activer/désactiver les assertions classe par classe ou, par package, avec l'option `-ea` (*enable assertions*) et de `-da` (*disable assertions*).

3.7 Boucles

3.7.1 `for`

Le syntaxe est la même que celle du C. Étant donnée une variable de boucle i :

```
for(état initial de i ; test sur i ; transformation de i){
    corps de la boucle à exécuter si test est vrai ;
}
```

La transformation sur i est souvent une opération d'incrémentation : i`++` (resp. `++`i), faite après (resp. avant) le passage dans la boucle, ou de décrémentation : i`--` (resp. `--`i), faite après (resp. avant) le passage dans la boucle. On peut définir localement la variable de boucle : par exemple `for(int i=0;i<10;i++)`.

3.7.2 `while`

Le syntaxe est la même que celle du C :

```
while(test){
    corps de la boucle à exécuter si test est vrai ;
}
```

3.7.3 `do... while`

`dowhile@do... while` Le syntaxe est la même que celle du C :

```
do{
      corps de la boucle à exécuter ;
}while(test);
```

qui permet d'exécuter le corps une première fois avant de faire le test, ce qui est pratique dans certains cas.

3.7.4 Distinction fondamentale entre `for` et `while`

En principe, un `for` est un cas particulier de `while` car les conditions initiales, finales et le test ne portent que sur la variable de boucle. On sait, en particulier, que la boucle terminera toujours. Ce n'est pas le cas avec une boucle `while`. Si Java, comme C, donne une facilité d'écriture qui permet d'écrire un `while` avec la syntaxe d'un `for`, c'est, cependant, une très bonne habitude à prendre que de distinguer ces deux instructions.

3.7.5 Sortie de boucle

Hormis la situation normale où le test de fin de boucle est vrai, il arrive que, suite à un test interne à la boucle, on souhaite quitter la boucle prématurément ou reprendre une nouvelle itération sans poursuivre jusqu'au bout l'itération en cours. Dans le premier cas, on utilisera `break`, dans le second, `continue`.

```
boolean egalite(int[] x, int[] y){
  boolean b = true;

  for(int j=0; j<n;j++){
    if (x[j] != y[j]){
      b = false;
      break;
    }
  }
  return b;
}
```

Ce mécanisme est possible pour plusieurs boucles imbriquées en nommant l'entrée de boucle. Ainsi :

```
boolean egalite(int[][] A, int[][] B){
  boolean b = true;

  all_loops:
  for(int i=0;i<k;i++){
    for(int j=0; j<n;j++){
      if (A[i][j] != B[i][j]){
        b = false;
        break all_loops;  // sort des deux boucles en meme temps
      }
    }
  }
  return b;
}
```

3.8 Tableaux primitifs

3.8.1 Tableaux unidimensionnels

Les indices des tableaux commencent à 0 et tout accès en dehors de la taille qui a été définie déclenche une `ArrayIndexOutOfBoundsException`. La mémoire requise pour stocker les éléments d'un tableau doit être demandée, comme pour tout autre objet référencé (donc hors valeurs de type primitif), par le mot-clé `new`, comme dans :

```
int [] tab = new int[30]; // tableau d'indices numérotés de 0 à 29
```

La mémoire réservée pour `tab` est désallouée automatiquement par le ramasse-miette lorsque l'objet devient obsolète. On peut se passer de `new` dans le cas où le tableau est initialisé explicitement :

```
int [] tab = {1, 2, 3};   // tab[0] = 1, tab[1] = 2, tab[2] = 3
```

L'objet `null` désigne un tableau vide.

3.8.2 Tableaux multidimensionnels

L'allocation mémoire se fait simultanément dans les différentes dimensions, comme dans :

```
int [][] tab = new int[k][n];
for(int i=0;i<k;i++){
 for(int j=0;j<n;j++){
  tab[i][j] = 0;
 }
}
```

Chapitre 4

Fondements objet de Java

4.1 Structure des classes en Java

4.1.1 Syntaxe d'une classe

Une classe java est composée de données, appelés *champs* et de fonctions appelées méthodes, voire d'autres classes, dites *internes*, dont nous verrons plus tard l'utilisation.

L'accès à ces *attributs* est limité par un certain nombre de mots-clé que nous verrons un peu plus loin. Ces éléments sont définis dans une zone délimitée par des accolades, précédée par le mot `class`,. Pour éviter les ambiguïtés, on peut préciser qu'on fait référence à l'instance courante en déréférençant les attributs à partir de `this`, comme dans l'exemple suivant :

```
class point{
  // abscisse et ordonnée entières, privées:
  private int x,y;

  // accesseurs publics, première version:
  public int get_x(){return x;}
  public int get_y(){return y;}

  // mutateurs publics, première version:
  public void set_x(int x){this.x=x;} // ambiguïté => this
  public void set_y(int y){this.y=y;} // ambiguïté => this
}
```

Les instances des classes sont accédées par références. Attention, donc, au sens des opérateurs d'affectation et d'égalité. À toute classe est attaché un objet `null` qui n'est, toutefois, pas considéré comme une instance de cette classe.

Le type des champs, ainsi que les prototypes des méthodes qui seront utilisées, sont typés statiquement (à la compilation). À cause du polymorphisme, le corps de la méthode qui sera utilisée est décidé à l'exécution (liaison retardée).

4.1.2 Modes d'encapsulation

Divers mots-clé paramètrent le niveau d'encapsulation des attributs :

1. `private` : accès depuis la classe uniquement ;
   - pour un champ : c'est un choix restrictif car il empêche aux classes héritières de l'utiliser directement ; l'accès, toujours possible par des accesseurs et des mutateurs, est plus sûr mais moins efficace ;
   - pour une méthode : c'est une pratique courante, par exemple, pour des méthodes auxilliaires qui n'ont pas vocation à être visibles de l'extérieur de la classe, ni même par les héritières qui utiliseront une autre méthode qui leur sera accessible ;

2. `package` : accès depuis les classes du package uniquement (c'est la valeur par défaut) ;
   – pour un champ : c'est un choix pratique, à condition de bien délimiter ses packages ;
   – pour une méthode : idem ;
3. `protected` : accès depuis toutes les classes du package et des héritières, uniquement ;
   – pour un champ : c'est un choix pratique qui va dans le sens de l'efficacité mais qui peut poser des problèmes de sécurité si un utilisateur de la classe détruit sa cohérence interne en agissant directement sur les champs sans passer par les accesseurs et les mutateurs ;
   – pour une méthode : c'est un paramétrage minimal pour que les classes héritières puissent en bénéficier directement ;
4. `public` : accès depuis partout où le paquetage est accessible :
   – pour un champ : c'est un choix qui doit rester très exceptionnel car il va à l'encontre de la philosophie d'encapsulation ;
   – pour une méthode : c'est le paramétrage des méthodes qui permettent à la classe d'interagir avec le reste des objets, quels qu'ils soient ;
   – pour une classe : c'est la façon de rendre la classe utilisable en dehors du paquetage (*cf.* Section 7.1, p. 47).

4.1.3 Déréférencement des attributs

Les champs c et les méthodes f d'un objet M sont identifiés par l'expression $M.c$ et $M.f$. Dans l'exemple précédent, si l'on a une instance `p` de la classe `point`, on peut écrire :

```
int a,b;
a = p.get_x();
b = p.get_y();
```

En revanche, du fait que `x` et `y` sont des champs `private` de `point`, on ne pourrait pas écrire :

```
int a,b;
a = p.x; // provoque une erreur à la compilation
b = p.y; // provoque une erreur à la compilation
```

Les programmeurs C++ comprendront que, du fait que les objets sont des références, on n'a pas besoin, comme lorsqu'il s'agit de pointeurs, d'utiliser une notation telle que M`->`c.

4.1.4 Attributs `final`

La présence de l'attribut `final` signifie :

1. pour un champ : que celui-ci est constant et tenter une modification engendre une erreur de compilation ;
2. pour une méthode : qu'elle ne pourra pas être surchargée dans une classe héritière ;
3. pour une classes : qu'elle ne pourra pas avoir d'héritière.

```
final int x = 1; // constante entière
x = 2;           // erreur à la compilation
```

4.1.5 Attributs `static`

La présence de l'attribut `static` signifie :

1. pour un champ : que celui-ci ne dépend pas d'une instance de la classe et peut être, par là même, stocké en un seul exemplaire ; toutes les instances peuvent y faire référence et on peut même y faire référence en dehors de toute instance (*cf.* la section sur les classes enveloppantes) ;

2. pour une méthode : qu'elle ne fait appel qu'à des champ statiques et peut ainsi être invoquée en dehors de toute instance.

Voici un exemple d'utilisation :

```
class votant{
  private static int nombre_votants;
  static int getNombreVotants(){return nombre_votants;}

  boolean a_vote = false;

  static void vote(){if(!a_vote){nombre_votants++;} a_vote = true;}
}

...

votant A;
votant B;

votant.getNombreVotants() // 0

A.vote();
B.vote();

votant.getNombreVotants() // 2
```

Il existe d'autres modificateurs (`abstract`, `synchronized`,...) que nous verrons plus tard.

4.1.6 Constructeurs

Les classes définies précédemment étaient presque totalement définies, restait à exprimer comment créer une nouvelle instance de la classe. Cela est fait avec un *constructeur*. Un constructeur est une méthode portant le nom de la classe dont il permet de créer des instances. Les constructeurs n'ont pas de type de retour (même pas `void` : ils « renvoient moins que rien »). On appelle *constructeur par défaut* le constructeur qui ne prend pas d'argument et *constructeur de copie* un constructeur utilisant un argument du même type dont il fait une copie (cela pose des problèmes en Java et on préfère utiliser un mécanisme de clonage, *cf.* Section 8.4, p. 55). Le mot-clé `this` peut également être utilisé pour invoquer un autre constructeur. Ainsi :

```
class point{
  // abscisse et ordonnée entières, privées:
  private int x,y;

  // accesseurs et mutateurs publics (voir la note ci-dessous pour le ‘‘final’’):
  public final int get_x(){return x;}
  public final int get_y(){return y;}
  public final void set_x(int x){this.x=x;}
  public final void set_y(int y){this.y=y;}

  // constructeurs (this désigne un autre constructeur):
  point(int x,int y){set_x(x.get_x()); set_y(x.get_y());}
  point(){this(0,0);}                              // constructeur par défaut
  point(point p){this(p.get_x(),p.get_y())}        // constructeur de copie

}
```

La syntaxe de création d'une instance est :

```
point O = new point();    // l'origine
point p = new point(3,4); // un nouveau point p
point q = new point(p);   // une copie q de p
```

4.1.7 Notes sur les constructeurs

Constructeurs, accesseurs et mutateurs `final`

En toute logique, le principe d'encapsulation des données voudrait que les constructeurs utilisent les accesseurs et les mutateurs. Cela est possible à condition que ceux-ci soient `final`. En effet, si une classe C' dérive d'une classe C (cf. *infra*), lors de la création d'une instance de C', l'appel à `super()` invoque un constructeur de C. Si ce dernier utilise une méthode f de C qui n'est pas `final` et que celle-ci est redéfinie en f' dans C', c'est la version f' qui sera utilisée (contrairement au C++) alors que l'instance dérivée n'est pas encore construite. Moralité : il ne faut pas utiliser de méthode non `final` dans un constructeur : en particulier, il faudra donc idéalement rendre `final` les accesseurs et mutateurs.

Utilisation de `this()`

En Java, dans un constructeur de la classe C, on ne peut invoquer un autre constructeur de C avec `this()` que si cette instruction est la première.. Enfin, remarquons que si `this` désigne bien une référence à l'objet courant, il n'est pas possible de la manipuler explicitement ; en d'autres termes, `this` n'est pas une *lvalue* : on ne peut pas écrire `this =`...

4.1.8 Le destructeur `finalize()`

Lorsque le moment le plus propice est venu, le ramasse-miette de Java est lancé automatiquement (il n'est pas nécessaire de l'invoquer dans votre programme, sauf situations exceptionnelles) par la méthode baptisée `System.gc()`. Chaque objet devenu inutile car plus référencé invoque alors son destructeur appelé `finalize()` qui va provoquer son élimination de la mémoire. Cette méthode n'a pas à être définie explicitement (car elle est héritée d'`Object`, que nous verrons lors de l'héritage) mais peut, si nécessaire, être redéfinie pour effectuer une opération de nettoyage (effacement de fichiers temporaires par exemple) avant que l'objet soit définitivement détruit. L'exemple suivant contient pourra être relu après avoir lu les sections sur l'héritage et les exceptions.

```
class point{
  // compteur du nombre de points créés:
  private static nb_points = 0;
  public static getNbPoints(){return nb_points;}

  // abscisse et ordonnée entières, privées:
  private int x,y;

  // accesseurs et mutateurs publics
  public final int get_x(){return x;}
  public final int get_y(){return y;}
  public final void set_x(int x){this.x=x;}
  public final void set_y(int y){this.y=y;}

  // constructeurs (this désigne un autre constructeur):
  point(int x,int y){set_x(x.get_x()); set_y(y.get_y()); nb_points++;} // incrémente
  point(){this(0,0);}                      // constructeur par défaut
  point(point p){this(p.get_x(),p.get_y())}            // constructeur de copie

  // destructeur décrémentant le compteur de points:
  protected void finalize throws Throwable(){
          try{nb_pts--:}finally{super.finalize();}
  }
}
```

4.1.9 Conversion des objets en chaîne de caractères

Le mécanisme de l'héritage permet de définir d'une manière très simple la façon dont les objets vont être affichés. Il suffit de (re)définir une méthode `public String toString()` dans la classe. Par exemple, pour la classe `point` vue plus haut, il suffit de rajouter la ligne :

```
class point{

...

  // convertisseur en chaîne de caractères:
  public String toString(){return ("(" + x + "," + y + ")");}

...

}
```

pour que l'on puisse invoquer directement `System.out.println(p)` où `p` est un point.

4.2 Classes enveloppantes des types primitifs

Les types primitifs disposent de classes dites *enveloppante* permettant de stocker certaines valeurs utiles, telles `Float.NaN` (*not a number*) et `Float.POSITIVE_INFINITY` (définies par le standard IEEE754) ou encore `Long.MAX_VALUE` (le plus grand entier `long`), ou de réaliser certaines opérations comme, par exemple, `Integer.parseInt(s)` qui renvoie un entier issu de la chaîne de caractères `s`.

4.3 Classes internes

Une classe peut être définie à l'intérieur d'une autre classe. Elle est alors dite *interne*. Cette classe connaît tous les attributs de la classe dans laquelle elle est définie (y compris les privés). Si la classe interne est déclarée `private`, par exemple, sa définition n'est, en revanche, pas visible par les autres classes du paquetage.

Chapitre 5

Héritage, classes abstraites et interfaces

5.1 Héritage

5.1.1 Introduction

L'héritage est le mécanisme central de la programmation orientée objet. Il permet de d'enrichir une classe C en une classe C' en ne redéfinissant que les attributs spécifiques à C' ou en redéfinissant dans la définition de C', des méthodes déjà définies dans C. Comme on pouvait faire référence à l'objet courant ou à ses constructeurs avec le mot-clé `this`, on peut faire référence à la classe mère ou à ses constructeurs avec le mot-clé `super`.

Notons que les constructeurs ne s'héritent pas et que les classes héritières peuvent diminuer l'encapsulation mais pas l'augmenter.

Dans l'exemple suivant, on définit une classe `point` :

```
class point{
  // données:
  private int x,y;

  // accesseurs/mutateurs:
  public final void set_x(int x){this.x=x;}
  public final void set_y(int y){this.y=y;}
  public final int get_x(){return x;}
  public final int get_y(){return y;}

  // constructeurs:
  public point(int a, int b){set_x(a);set_y(b);}

  // affichage:
  public String toString(){return ("(" + x + "," + y + ")");}
}
```

dont on dérive une classe `pixel` avec le mot-clé `extends` :

```
class pixel extends point{
  // données:
  protected int c;

  // accesseurs / mutateurs:
  public final int get_c(int c){return c;}
  public final void set_c(int c){this.c=c;}

  // constructeurs:
  public pixel(point A, int c){super(A.x,A.y); set_c(0);}
```

```
  public pixel(int a, int b, int c){super(a,b); set_c(0);}

  public String toString(){return super.toString() + ", couleur=" + c;}
}
```

L'héritage permet de réaliser un polymorphisme dit *d'héritage* qui signifie que les instances des classes filles peuvent invoquer des méthodes des classes mères de manière transparente. Ainsi, la méthode `set_x()` peut être invoquée depuis un `pixel p` en écrivant `p.set_x(10)`, par exemple. Nous reverrons ce mécanisme dans la Section 8.1, p. 51. Voici un exemple de ce qu'on peut faire avec les classes définies plus haut :

image.java

```
class image{
  public static void main(String[] args){
    point A = new point(320,200);
    System.out.println(A); // Invocation automatique du toString() de point
    int noir = 0, blanc = 255;
    pixel p = new pixel(A,noir);
    System.out.println(p); // Invocation automatique du toString() de pixel
    p.set_x(10);       // Polymorphisme d'héritage: p est ici identifié à un point
    p.set_y(200);      // Polymorphisme d'héritage: p est ici identifié à un point
    p.set_c(blanc);
    System.out.println(p);
  }
}
```

5.1.2 Les classes `Object` et `Class`

En Java, toutes les classes dérivent d'une classe appelée `Object`. Lorsque l'on définit une nouvelle classe, celle-ci bénéficie des propriétés d'`Object` bien qu'on n'y fasse pas explicitement référence (on écrit pas `extends Object`, par exemple), en général. Chaque classe est représentée par une instance de la classe `Class`.

Voici les méthodes définies dans `Object` :

1. `public boolean equals(Object x)` désigne l'égalité par référence et équivaut, de ce fait, à l'opérateur `==`. Elle est destinée à être redéfinie pour que la sémantique de l'égalité soit conforme à ce qu'on veut. Par exemple, le `equals()` de la classe `String` teste l'égalité caractère par caractère.
2. `public int hashCode()` définit une fonction de hachage utilisée par les tables de hachage. Elle est destinée à être redéfinie pour que la sémantique du hachage soit conforme à ce qu'on veut. Par exemple, le `hashCode()` de la classe `String` renvoie le même code pour deux chaînes égales, caractère par caractère.
3. `protected Object clone() throws CloneNotSupportedException` permet de créer des copies de l'objet courant dans certaines conditions (sur lesquelles nous reviendrons). Le `throws CloneNotSupportedException` signifie que la méthode peut déclencher une exception du type `CloneNotSupportedException` en cas de problème de clonage.
4. `public final Class getClass()` renvoie la classe exacte de l'objet courant (celle du constructeur ayant servi à créer l'objet, même si celui a été casté). Cette méthode étant `final`, elle ne peut pas être redéfinie.
5. `public void finalize() throws Throwable` est le destructeur qui nettoie la mémoire de l'objet courant lors de l'appel par le ramasse-miette. Cette méthode peut être redéfinie comme nous l'avons fait dans la Section 4.1.8, p.32. Le `throws Throwable` signifie que la méthode peut déclencher une interruption en cas de problème.
6. `public String toString()` renvoie une chaîne de caractères représentant l'objet courant par le nom de la classe suivie du caractère `@`, suivie de son `hashCode()`. Cette méthode peut être

redéfinie pour que la sémantique de représentation soit conforme à ce qu'on veut, comme nous l'avons fait pour la classe `point`, par exemple. La méthode `public String toString()` est appelée par `System.out.println()`.

En Java, l'héritage des classes est *simple*, ce qui signifie qu'une classe peut avoir plusieurs descendantes mais une seule ascendante (sauf `Object` qui n'en a pas). Java propose, de surcroît, une notion d'héritage multiple des interfaces, comme nous le verrons à la section consacrée à ce sujet.

5.2 Méthodes et classes abstraites

Dans la définition d'une classe C, il peut être souhaitable de laisser la définition d'une méthode f aux héritières de C tout en permettant aux autres méthodes de C d'utiliser f. Cela est fait en préfixant la définition de f par le mot-clé `abstract`. Dans ce cas, la définition de la classe C doit également être préfixé par `abstract`, ce qui signifie que la classe n'est pas instancible. Les arguments de f doivent apparaître avec leur type *et* le nom des variables muettes qu'on utilisera dans les classes héritières pour leur définition. Une classe abstraite peut dériver d'une classe non-abstraite : c'est, en particulier, le cas de toutes les classes abstraites dérivant d'`Object`.

Le mécanisme des classes abstraites est très utile lorsque la méthode f telle qu'elle sera définie dans les classes héritières sera censée utiliser une représentation de l'objet qui n'est pas encore définie à ce stade. On définit ainsi :

```
abstract class parisien{
    private final int age_max = (int)(75.0 + new java.util.Random().nextGaussian()*10.0);
    private final void jeunesse(){System.out.println("Ouin!!! <grandit>, <etudes>");}
    private final void metro(){System.out.println("beeeeep clac pshhh VRRRRR...");}
    private abstract void boulot();
    private void dodo(){System.out.println("Zzzzz...");}
    private abstract void bilan();
    private final void meurt(){System.out.println("Argh!");}
    public void vit(){
        jeunesse();
        for(int i=0;i<age_max*365;i++){metro();  boulot();  dodo();}
        meurt(); bilan();
    }
}
```

dont on peut dériver les professionnels suivants[1] :

```
class commercial extends parisien{
    private int pecule = 0;
    private void boulot(){System.out.println("blabla"); pecule += 1000;}
    private void bilan(){System.out.println("Le défunt avait gagné " + pecule);}
}

class chercheur extends parisien{
    private int questions = 0, reponses = 0;
    private void boulot(){System.out.println("grat grat"); reponses++; questions += 100;}
    private void bilan(){System.out.println("Le défunt avait trouvé "
                + (float)reponses/(float)questions)*10 + "% des réponses";}
}
```

on peut alors observer la vie de ces créatures :

```
commercial p = new commercial(); p.vit();
chercheur q = new chercheur(); q.vit();
```

[1] dont on trouve des specimens en dehors de Paris mais ceux-là ont alors un style de vie moins stressant.

5.3 Interfaces

5.3.1 Introduction

On peut voir une *interface* comme une classe abstraite pure, c'est-à-dire ne possédant que des méthodes abstraites et, éventuellement, des champs constants. Tout champ d'une interface est supposé `public final` et toutes ses méthodes sont supposées `public abstract`. On définit une interface I en déclarant `interface` I comme on l'aurait fait pour une classe. Par défaut, à l'instar de ce qui se passe pour une classe, une interface est visible depuis un paquetage. On peut la rendre visible de partout en préfixant sa définition par le mot-clé `public`.

Définissons une interface `Animal` dans laquelle on ne se préoccupe que de la fonction d'alimentation :

```
interface Animal{void Mange();}
```

5.3.2 Héritage multiple des interfaces

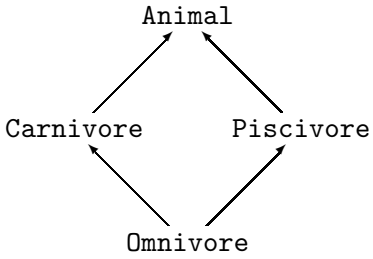
Un mécanisme d'héritage existe également pour les interfaces et l'on peut définir :

```
interface Carnivore extends Animal{void Traque();}
interface Piscivore extends Animal{void Traque();}
interface Frugivore extends Animal{void Cueille();}
```

Là où les choses changent par rapport aux classes, c'est qu'on peut avoir un héritage multiple :

```
interface Omnivore extends Carnivore, Piscivore, Frugivore{}
```

Les méthodes `Mange()` et `Traque()` étant abstraites, par définition, dans toutes les interfaces qu'on considère, on n'a pas à s'inquiéter du fait que l'invocation de ces méthodes par un `Omnivore` utiliserait les versions d'`Animal`, `Carnivore` ou `Piscivore`. En effet, le corps des méthodes n'est pas encore défini à ce stade. Seul le respect de leur prototype est pertinent. Nous avons ici un héritage dit « en diamant » ou « en losange » :



Un conflit peut toutefois survenir si on a un héritage multiple de plusieurs interfaces dans lesquelles sont définies des méthodes portant le même nom, ayant des arguments de même type, et dont seul le type de retour diffère. Ce type de conflit est détecté à la compilation :

```
interface I1{int f(int i, int j);}
interface I2{int f(int i, int j);}
interface I3{double f(int i, int j);}
class C implements I1,I2; // OK
class D implements I1,I3; // erreur de compilation
```

5.3.3 Implémentation d'une interface

Lorsqu'une interface I est définie, on explicite qu'une classe C satisfait aux conditions de I en écrivant `class` C `implements` I.

Ainsi, on peut définir le `guepard`, qui est `Carnivore` :

```
class guepard implements Carnivore{
    private String s;
                                public final void set_s(String s){this.s=s;}
    public guepard(String s){set_s(s);}
    public void Traque(){System.out.println(s +" court après la proie,"
                        + "lui saute dessus et la mord à la gorge"
                        + "jusqu'à ce que mort s'ensuive");};
    public void Mange(){System.out.println(s +" déchiquète goulûment la proie");}
}
```

et l'`humain`, qui est (en général) `Omnivore` :

```
class humain implements Omnivore{
    private String s;
                                public final void set_s(String s){this.s=s;}
    humain(String s){set_s(s);}
    public void Traque(){System.out.println(s + " va au supermarché");};
    public void Cueille(){Traque();};
    public void Mange(){System.out.println(s +" déplace la fourchette en alternance "
                                           +\"entre sa bouche et son assiette");}
}
```

On remarque ici que la méthode `Traque()` implémente simultanément la version de `Carnivore` et de `Piscivore`. On pourra ensuite invoquer :

```
humain alfred = new humain("alfred"); alfred.Traque(); alfred.Mange();
guepard groarrr = new guepard("groarrr"); groarrr.Traque(); groarrr.Mange();
```

5.3.4 Interfaces indicatrices

On peut utiliser une interface comme mécanisme de balisage. Considérons une instance M d'une classe C. La classe C implémente une interface I ssi l'expression (*M* `instanceof` *I*) est vraie. C'est précisément ce qui se passera lors de l'invocation de M`.clone()` (la méthode `clone()` est définie dans `Object` – pas dans l'interface `Cloneable` – et peut potentiellement être invoquée par n'importe quel objet) : si C n'implémente pas l'interface I = `Cloneable`, une `CloneNotSupportedException` sera levée. C'est un mécanisme de protection pour garantir que le concepteur de la classe C n'invoquera pas `clone()` par mégarde sans avoir compris ce qu'il faisait. On reviendra sur le clonage dans la Section 8.4, p. 55.

On récupère la liste des interfaces implémentées par une classe C donnée en invoquant C`.getInterfaces()` qui renvoie un `Class[]`. Cela permet de détecter qu'une interface est, ou pas, implémentée par une classe et d'agir en fonction.

Chapitre 6

Interruptions

6.1 La classe `Throwable`

Les interruptions donnent la possibilité d'écrire des programmes plus clairs dont le code se focalise sur la situation typique en délégant les situations exceptionnelles à des gestionnaires particuliers. Ainsi, pour faire la photocopie d'un original qu'on a placé sur la vitre, il est plus clair d'expliquer :

> « premièrement : on appuie sur le bouton copie ;
> deuxièmement : on récupère la copie et l'original.
>
> si un problème de bourrage survient, on retire la feuille qui bourre ;
> si un problème autre problème survient, on fait réparer le copieur. »

plutôt que :

> « premièrement : on appuie sur le bouton copie ;
> si un problème de bourrage survient, on retire la feuille qui bourre ;
> si un problème autre problème survient, on fait réparer le copieur ;
> sinon (c'est le deuxièmement) : on récupère la copie et l'original. »

En effet, dans la première formulation, on a pu séparer la situation typique, décrite dans les deux premières lignes, de la situation atypique, envisagée dans les deux dernières, tandis que dans la seconde formulation, l'action principale est mélangée avec le traitement des situations atypiques, ce qui rend plus difficile la compréhension de l'action principale. Le mécanisme des interruptions permet donc d'aller à l'essentiel sans se perdre dans les détails.

Une *interruption* est définie comme une rupture du déroulement normal des instructions. Elle est matérialisée par l'émission d'un signal de type `Throwable` émis par une méthode grâce à l'instruction `throw`.. De la classe `Throwable` dérive la classe `Error` qui sert à signaler les erreurs graves (*cf. infra*), et la classe `Exception` pour les autres événements. C'est cette dernière classe que nous étudierons particulièrement.

Tous les `Throwable` ont un champ de type `String` qui est initialisé par un constructeur ayant cette chaîne pour argument. Il sert à stocker un message d'erreur qu'on récupère avec la méthode `getMessage()`. La méthode `toString()` d'une exception renvoie le nom de l'exception suivi de ce message.

6.2 Syntaxe des gestionnaires d'interruptions

Pour gérer une interruption, on utilise la syntaxe suivante :

```
try{
      code susceptible de lever une interruption ;
}catch(type T_1 d'une interruption e_1){
      traitement de l'interruption e_1 ;
                ⋮
}catch(type T_n d'une interruption e_n){
      traitement de l'interruption e_n ;
}finally{ // optionnel
      instructions effectuées systématiquement ;
      une fois terminés l'exécution du bloc try et des blocs catch éventuels,
      y compris en présence de return, continue ou break,
      (sauf, en cas de System.exit()) ;
}
```

Lorsqu'un exception d'un type T non assimilable à $T_1, \ldots, T_n$ est levée, celle-ci est propagée dans les blocs `try` dans lequel la zone de code est imbriquée, jusqu'à ce qu'un gestionnaire d'exception de type T sache la traiter. En dernier recours, l'exécution est traitée par la la méthode `main` qui provoque un affichage de la pile d'appels des méthodes depuis la méthode ayant déclenché l'interruption et sort inopinément du programme.

6.3 La classe `Exception`

Afin d'illustrer l'exemple que nous avons donné, définissons les exceptions :

```
class ProblemePhotocopieuseException extends Exception{
  ProblemePhotocopieuseException(){super();}
  ProblemePhotocopieuseException(String s){super(s);}
}

class ProblemeBourragePhotocopieuseException extends ProblemePhotocopieuseException{
  ProblemeBourragePhotocopieuseException(){super();}
  ProblemeBourragePhotocopieuseException(String s){super(s);}
}
```

Créons ensuite une classe `utilisateur`, munie d'un champ `boolean copie_ok` et d'un générateur pseudo-aléatoire `java.util.Random R` qui nous servira à simuler l'activité d'une photocopieuse. On définit les méthodes :

```
  void place_original(int i){
    System.out.println("Met l'original de " + i + " sur la vitre"); copie_ok=false;
  }

  void recupere(int i){
    System.out.println("Récupère l'original " + i + " et sa copie");
  }
```

Nous voulons ensuite définir une méthode `bouton_copie()`, susceptible de propager une `ProblemePhotocopi` (cela est matérialisé par le `throws` dans la déclaration de la méthode). Dans 75% des cas, la copie aura lieu normalement, dans les 25% restants, une `ProblemePhotocopieuseException` sera levée. Si tel est le cas, il s'agira, dans les 3/4 des situations, d'une `ProblemeBourragePhotocopieuseException`.

```
  void bouton_copie() throws ProblemePhotocopieuseException{
      System.out.println("Appuie sur copie");
      if(R.nextDouble()>0.75){
          if(R.nextDouble()<0.75){
```

```
            throw new ProblemeBourragePhotocopieuseException(">>>Bourrage papier");
        }else{
            throw new ProblemePhotocopieuseException(">>>Autre problème");
        }
    }else{copie_ok = true;}
}
```

On pourra traiter ces situations problématiques en utilisant les méthodes :

```
void traiter_bourrage(){System.out.println("Enlève la feuille qui bourre sous le capot");}

void reparation(){System.out.println("Fait réparer la photocopieuse");}
```

La photocopie `n` pages de texte consiste désormais à placer, pour chaque `i` entre `0` et `n-1`, l'original `i` sur la vitre, puis à essayer de copier cet original avec la méthode `bouton_copie()` et à récupérer le résultat et l'original avec `recupere(i)`. On s'occupera des, bourrages avec `traiter_bourrage()`, sinon, on se résoudra à utiliser `reparation()`.

```
void photocopie(int n){
  for(int i=0;i<n;i++){
    place_original(i);
    while(!copie_ok){
      try{bouton_copie(); recupere(i);}
      catch(ProblemeBourragePhotocopieuseException e){
        System.out.println(e.getMessage());
        traiter_bourrage();
      }
      catch(ProblemePhotocopieuseException e){
        System.out.println(e.getMessage());
        reparation();
      }
      finally{ // exécuté systématiquement une fois les blocs try et/ou catch terminés
        System.out.println("-----------------------------------");
      }
    }
  }
}
```

On remarquera que la méthode `photocopie()` ne propage aucune exception puisque celles qui sont lancées à l'intérieur de cette méthode sont rattrapées par les blocs `catch` définis plus haut. Une exécution de :

```
utilisateur u = new utilisateur();
u.photocopie(10);
```

est :

```
Met l'original de 0 sur la vitre
Appuie sur copie
Récupère l'original 0 et sa copie
-----------------------------------
Met l'original de 1 sur la vitre
Appuie sur copie
>>>Autre problème
Fait réparer la photocopieuse
-----------------------------------
Appuie sur copie
Récupère l'original 1 et sa copie
```

```
-----------------------------------
Met l'original de 2 sur la vitre
Appuie sur copie
Récupère l'original 2 et sa copie
-----------------------------------
Met l'original de 3 sur la vitre
Appuie sur copie
Récupère l'original 3 et sa copie
-----------------------------------
Met l'original de 4 sur la vitre
Appuie sur copie
Récupère l'original 4 et sa copie
-----------------------------------
Met l'original de 5 sur la vitre
Appuie sur copie
Récupère l'original 5 et sa copie
-----------------------------------
Met l'original de 6 sur la vitre
Appuie sur copie
Récupère l'original 6 et sa copie
-----------------------------------
Met l'original de 7 sur la vitre
Appuie sur copie
Récupère l'original 7 et sa copie
-----------------------------------
Met l'original de 8 sur la vitre
Appuie sur copie
Récupère l'original 8 et sa copie
-----------------------------------
Met l'original de 9 sur la vitre
Appuie sur copie
>>>Bourrage papier
Enlève la feuille qui bourre sous le capot
-----------------------------------
Appuie sur copie
Récupère l'original 9 et sa copie
-----------------------------------
```

6.4 La classe `Error`

Les interruptions de la classe `Error` signalent des problèmes graves de Java qui sont, en général, irrécupérables et terminent l'exécution du programme inopinément. On pourra citer `AssertionError` qui est levée lorsqu'une assertion est fausse, `AbstractMethodError` qui signale une tentative d'invocation de méthode abstraite, `StackOverFlowError` qui signifie que la taille de la pile est insuffisante ou encore `OutOfMemoryError` qui signifie que le ramasse-miette n'arrive plus à libérer la mémoire nécessaire.

6.5 Interruptions vérifiées et non-vérifiées, `RuntimeException`

Dans l'exemple précédent, le prototype de la méthode représentant la copie d'une page était `void bouton_copie() throws ProblemePhotocopieuseException` car l'exception levée dans cette méthode n'était pas traitée localement, mais par une autre méthode (la méthode `photocopie()` en l'occurence) qui surveille les appels à `bouton_copie()` en plaçant cette instruction dans un bloc `try` suivi d'un `catch` gérant les `ProblemePhotocopieuseException`.

Il est évident, toutefois, qu'on ne peut pas prévoir (par définition) tous les problèmes susceptibles de survenir lorsqu'on écrit une méthode. C'est le cas des `Error` qui sont, pour cette raison, des interruptions dites *non-vérifiées*

Si, dans le cas des `Error`, on ne peut pas faire grand-chose en général sinon quitter le système, il est des situations moins graves dans lesquelles, suite à une erreur de programmation, une erreur survient. Cette erreur peut, dans un certain nombre de cas, être traitée sans pour autant mettre fin au programme en cours. C'est le rôle des `RuntimeException` (elle-même héritée d'`Exception`), ainsi baptisées car on ne découvre qu'au moment de l'exécution qu'elles sont effectivement levées et ne sont pas considérées comme prévisibles.

Les `RuntimeException` et leurs héritières n'ont donc pas à être déclarées dans la liste des exceptions après le mot `throws`, dans le prototype d'une méthode. Il en va ainsi de `ArrayIndexOutOfBoundsException` signalant un accès à un tableau en dehors de ses bornes ou de `NullPointerException` signalant qu'on tente une opération sur l'objet `null`.

Du fait de leur nature imprévisible, les `RuntimeException` n'ont pas à apparaître dans la liste des interruptions qui suit le mot `throws`. La syntaxe serait, en effet, autrement plus pénible s'il fallait signaler, à chaque déclaration de méthode, que certains objets étant susceptibles d'être `null`, une `NullPointerException` pourrait être lancée !

Toutes les exceptions n'étant pas des `RuntimeException` sont donc *vérifiées*. C'est, par exemple, le cas des `IOException`, lancées lors de problèmes relatifs aux entrées-sorties, telles que `FileNotFoundException` signalant qu'un fichier demandé n'existe pas ou encore que `EOFException` annonçant la fin d'un fichier.

Chapitre 7

Paquetages et documentation

7.1 Paquetages (`package`)

Il est utile de réunir des classes relevant d'une même logique dans un *paquetage* (*package*). Un paquetage est défini en mettant la commande `package`, suivi du nom du package. Celui-ci est composé d'une succession de noms de répertoires séparés par un point. Le programme appelant devra chercher à l'origine de ce répertoire. Considérons ainsi l'arborescence :

```
$ tree
.
|-- Recettes
|   |-- Plats
|   |   |-- Couscous.class
|   |   |-- Couscous.java
|   |   |-- Nem.class
|   |   |-- Nem.java
|   |   |-- PotAuFeu.class
|   |   `-- PotAuFeu.java
|   `-- Tartes
|       |-- Fraises.class
|       |-- Fraises.java
|       |-- Poires.class
|       |-- Poires.java
|       |-- Pommes.class
|       `-- Pommes.java
|-- dejeuner.class
`-- dejeuner.java
```

Si le fichier `Pommes.java` est défini par :

Pommes.java

```
package Recettes.Tartes;

// la classe est publique, pour pouvoir être utilisée en dehors du package:
public class Pommes{
    public String toString(){return "Tarte aux pommes";}
}
```

et que le fichier `PotAuFeu.java` est défini par :

PotAuFeu.java

```
package Recettes.Plats;

// la classe est publique, pour pouvoir être utilisée en dehors du package:
public class PotAuFeu{
    public String toString(){return "Pot au feu";}
}
```

alors, on peut accéder aux définitions de ces paquetage depuis le programme `dejeuner.java` de trois manières :

1. en faisant `import Recettes.Plats.*`, on rend connue la définition de toutes les classes définies dans le paquetage `Recettes.Plats` ;
2. plus précisément, en faisant `import Recettes.Plats.PotAuFeu`, on rend connue la définition de la classe `PotAuFeu` ;
3. sans `import`, en préfixant le nom de la classe `Pomme` par le chemin correspondant au paquetage, c'est-à-dire en y faisant référence par l'expression `Recettes.Tartes.Pommes`.

dejeuner.java

```
import Recettes.Plats.PotAuFeu;

class dejeuner{
  public static void main(String[] args){
      PotAuFeu plat = new PotAuFeu();

      // en utilisant le nom complet:
      Recettes.Tartes.Pommes dessert = new Recettes.Tartes.Pommes();
      System.out.println("Votre plat: " + plat + "votre dessert: " + dessert);
  }
}
```

7.2 Javadoc

7.2.1 Génération automatique de documentation

L'environnement de développement Java vient avec un compilateur appelé `javadoc` qui va lire le fichier `.java` pour en extraire un commentaire et générer des pages de documentation en HTML.

Pour extraire la documentation `foo.html` du fichier source `foo.java`, il suffit de taper :

```
javadoc foo.java
```

en ayant pris le soin de commenter le fichier `foo.java` d'une façon semblable à :

foo.java

```
/** Cette classe sert à donner un exemple de <b>javadoc</b>
    @author <a href="http://www-rocq.inria.fr/~pecquet">Lancelot Pecquet</a>
    @version 1.0 (2004-04-06)
*/
public class exemple{
...
}
```

Comme on peut le remarquer, ce commentaire :
- commence par `/**` et se termine par `*/` ;
- précède immédiatement la définition de la zone qu'il commente (ici une classe, cela pourrait aussi bien être une méthode, un paquetage ou une interface) ;
- contient des balises HTML (comme par exemple la balise `<b> ... </b>` qui mettra le mot « javadoc » en gras dans la version HTML, ou encore l'hyperlien `<a href= ... >... </a>` qui pointera sur la page web de l'auteur ;
- des balises spécifiques à Javadoc, comme `@version` ou `@author`.

7.2.2 Définition de quelques balises

Voici quelques mots-clé utiles pour baliser les commentaires de vos sources :
- `@author` A : définit A comme l'auteur de la classe qui suit ;
- `@version` V : définit V comme étant le numéro de version de la classe qui suit ;

- `@return` x S : associe le commentaire S à la variable x retournée par la méthode qui suit ;
- `@deprecated` S : signale que la zone de code qui suit est devenue obsolète ;
- `@exception` e S : associe le commentaire S à l'exception e levée par la méthode qui suit ;
- `@see` L : signale un lien vers la documentation de la classe L si L est un nom de classe, vers la documentation de la méthode f de la classe a si L est de la forme a`#`f où a est le nom d'une classe et f le nom d'une de ses méthodes ;
- `@param` : x S : associe le commentaire S à l'argument x de la méthode qui suit ;
- `@since` d : signale que le code qui suit a été ajouté à la date d.

La documentation officielle en ligne de Java est générée de la sorte. Une capture d'écran de celle-ci est visible sur la Fig. 2.2, p. 20.

Chapitre 8

Transtypage, affectation, égalité, comparabilité, clonage

8.1 Transtypage (*cast*)

8.1.1 Introduction

Dans un langage de programmation, il arrive, en permanence, de vouloir convertir une donnée x d'un type T vers une donnée x' d'un type T'. L'application de la fonction $\varphi : x \longmapsto x'$ s'appelle un *transtypage (cast)*.

Par exemple, il y a une fonction φ : `int` $\longrightarrow$ `long` qui envoie un `int` x représentant un entier n sur un `long` x' représentant également l'entier n. On noterait `long` x' `= (long)`x.

Cette fonction est injective et, de ce fait, on peut considérer qu'il s'agit d'une identification naturelle ; à ce titre, il n'est pas nécessaire de faire la conversion explicitement et on peut écrire directement `long` x' `=` x. On parle alors d'*upcast*.

On pourrait souhaiter définir une application réciproque phi^{-1} (*downcast*) qui transforme tout `long` en `int`. Or φ n'est inversible que dans le sous-ensemble $I = \varphi($`int`$)$ des `long` qui étaient de la forme $\varphi(x)$ où x était un x, c'est-à-dire les « petits » `long`.

Le concepteur de langage est alors face à un choix : établir une convention pour représenter les images réciproques des éléments x' qui sont des `long` mais qui ne sont pas dans I ou interdire purement et simplement la conversion d'un `long` en `int` s'il n'est pas dans I.

En tout état de cause, un *downcast* n'est pas anodin et Java nécessitera toujours d'y faire mention explitement.

Toujours est-il que ces deux approches coexistent, en Java. Une valeur de type primitif sera toujours downcastable (nous y reviendrons plus loin), ce downcast donnant lieu, le cas échéant à une perte d'information (par troncature de bits, par exemple). Dans l'exemple précédant, il faudra donc écrire `int` x `= (int)`x'. Un objet M d'une classe C ne pourra, quant à lui, être downcasté en un objet M' d'une classe héritière C' que si M provient d'un précédent *upcast* d'une classe assimilable à C'.

En réalité, dans ce cas, l'*upcast* d'un objet M' de classe C' vers un objet de classe C ne détruit pas les attributs de M' qui ne sont pas dans C mais se contente de les masquer. Le *downcast* d'un objet M aura pour effet de démasquer ces attributs, si M provient d'une instance de la classe C' et lèvera une `ClassCastException`, sinon.

8.1.2 L'opérateur `instanceof`

Afin de s'assurer qu'un *downcast* d'une valeur x de type T vers une valeur de type T' est autorisé (ces types sont donc non-primitifs, tableaux inclus), on peut utiliser l'opérateur `instanceof` en testant l'expression x `instanceof` T' et en réalisant la conversion, dans le cas où cette expression est évaluée à `true`, par l'affectation T x' `=` $(T)x$. Si l'on tente une conversion mais que celle-ci n'est pas possible, une `ClassCastException` (non-vérifiée) est lancée. Voici quelques exemples commentés :

```
System.out.println("foo" instanceof String); // true
System.out.println("foo" instanceof Object); // true
```

Le résultat est bien celui attendu ; en revanche, les lignes :

```
System.out.println(1 instanceof Object); // ne compile pas:
System.out.println(1 instanceof int);    // ne compile pas:
```

ne compile pas et provoque les messages d'erreur respectifs `error :Cannot apply operator "instanceof" to "int" and * "java.lang.Object"` et `error :Cannot apply operator "instanceof" to "int" and "int"`, ce qui est normal puisque `instanceof` ne s'applique qu'aux types primitifs.

On peut, en revanche, utiliser les classes enveloppantes :

```
System.out.println(new Integer(1) instanceof Integer); // true
```

cependant, certaines vérifications, effectuées à la compilation empêchent d'écrire le code suivant, qui semble pourtant raisonnable :

```
System.out.println((new Integer(1)) instanceof String); // ne compile pas:
```

et le compilateur indique dans son message : `error :Cannot apply operator "instanceof" to "java.lang.Integer" and "java.lang.String"`. Le fait de remonter dans la hiérarchie des objets permet de passer l'étape de compilation mais `instanceof` fait bien la différence entre un `Object` qui provient d'un `Integer` et un `Object` provenant d'une `String` :

```
System.out.println((Object)"foo" instanceof String);            // true
System.out.println((Object)(new Integer(1)) instanceof String); // false
```

Comme on l'a dit, l'opérateur `instanceof` fonctionne pour les tableaux, y compris de type primitif :

```
System.out.println(new int[]{1} instanceof int[]);    // true
System.out.println(new int[]{1} instanceof Object);   // true
```

Enfin, nous l'avons dit, `null` n'est l'instance d'aucune classe :

```
System.out.println(null instanceof Object);            // false:
```

Donnons maintenant un exemple de problème de `downcast` à l'exécution :

```
Object x = (Object)(new Integer(1));
String y = (String) x;
```

L'appel de la seconde ligne provoque la levée d'une `ClassCastException` et l'affichage du message d'erreur : `can't cast 'java/lang/Integer' to 'java/lang/String'`.

8.1.3 Upcast implicite des types primitifs

Nous l'avons vu plus haut, les types primitifs sont automatiquement upcastés lors des affectations, selon la hiérarchie définie dans la Fig. 8.1, mais il y a également *upcast* automatique lors d'opérations arithmétiques mettant en jeu des opérandes de type différent et ce, *juste avant* que l'opération soit effectuée. Il faut noter que les valeurs de type `char`, `byte` et `short` seront immédiatement convertis en `int` car ces types ne supportent pas les opérations arithmétiques.

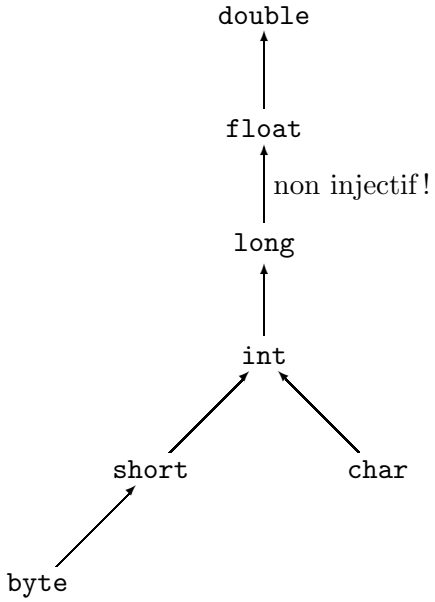


FIG. 8.1 – Transtypage des types primitifs : `char` est converti en `int` car le résultat de sa conversion doit être un entier positif et que les `short` sont signés. Attention, la conversion `long` $\longrightarrow$ `float` n'est pas injective : les `long` sont codés sur 64 bits tandis que les `float` ne le sont que sur 32 bits. Elle permet toutefois de faire des opérations approchées dans des intervalles plus vastes que ceux autorisés, en calcul exact, avec les `long`.

8.1.4 Downcast explicite des types primitifs

Nous l'avons déjà signalé, mais nous donnons ici, un exemple de ce qui se passe lors d'un *downcast* avec perte d'information.

```
int n = 1000;
byte b = (byte)n;  // -24
```

La variable `n` se code par la chaîne binaire $0^{20}\,0011\,1110\,1000$ dont les 8 bits les moins significatifs, choisis par la convention de *downcast* pour coder le byte `b`, sont 1110 1000. Le `byte` b représente désormais, en complément à 2, l'entier -24 qui est loin d'être une approximation satisfaisante de `n` $=$ 1000.

8.1.5 Upcast implicite des types non-primitifs

Ce phénomène est sans-cesse utilisé par le polymorphisme d'héritage. Nous pouvons revisiter, à la lumière de ce qui précède, l'exemple de la p. 35. On avait défini une classe `point` et une classe dérivée `pixel`. Lorsqu'on exécute la commande `p.set_x(10);`, alors que `p` est un `pixel` et que `set_x()` est une méthode de la classe point, en l'absence d'un *upcast* implicite on devrait écrire `((point)p).set_x(10)`.

8.1.6 Downcast explicite des types non-primitifs

Toujours avec l'exemple précédent, l'expression `((pixel)A).set_c(noir)` ne pourrait être valide car `A` n'est pas assimilable à un `pixel` : il n'a pas de couleur ! Par contre, si l'on définit un nouveau `point B = new point(200,200)`, on peut fabriquer un tableau de `point` contenant `p` et `B` :

```
point[] nuage = new point[]{p,B}; // p est vu comme un point ici
```

mais pour modifier la couleur de l'élément d'indice 0 de `nuage`, il faut le downcaster en pixel :

```
((pixel)nuage[0]).set_c(blanc); // downcast explicite obligatoire: ok
((pixel)nuage[1]).set_c(blanc); // ClassCastException: ce n'est pas un pixel!
```

8.1.7 Tableaux de types non-primitifs

Pour élargir l'exemple précédent, voici ce qu'on peut faire et ne pas faire avec des tableaux de type non-primitif :

```
Object o = new int[5]; // OK
class B extends A{};
A[] S;
B[] T;

S=T; // oui
T=S; // non: downcast explicite nécessaire!
```

8.2 Égalité

Une fois réglés les problèmes de transtypage qui peuvent réserver des surprise, on peut se demander ce que signifie l'*égalité* entre deux instances.

L'opérateur `==` désigne, on l'a vu, l'égalité bit-à-bit entre valeurs de type primitif et l'égalité bit-à-bit des références entre valeurs de type référencé.

Lorsqu'on définit une classe, on souhaite donner une sémantique différente à la notion d'égalité. Pour cela, on redéfinit la méthode `equals()` définie dans la classe `Object` (attention, pas *surcharger*, le type de la valeur passée en paramètre est bien `Object` : ceci est une convention utilisée dans un certain nombre de bibliothèques avec lesquelles il faut rester compatible). En respectant ce principe, toutes les méthodes utilisant un test d'égalité pour leurs calculs (par exemple les méthodes testant l'appartenance à un ensemble, comme on le verra au Chapitre 9, p. 59) auront la sémantique attendue.

Le contrat d'implémentation d'`equals()` est que :

- `equals()` est une relation d'équivalence (réflexive, symétrique, transitive) ;
- `equals()` renvoie toujours la même valeurs lors d'appels successifs si les objets n'ont pas été modifiés ;
- `x.equals(null)` renvoie toujours `false` pour toute valeur `x` $\neq$ `null`.

Par exemple, pour la classe point, il est pertinent de définir une méthode exprimant que deux points sont égaux si et seulement si leurs abscisses sont égales et leurs ordonnées aussi :

```
public boolean equals(Object Q){
  if ((x!=null) && (Q instanceof point)){
    point q = (point)Q;
    b = ((x == q.get_x()) && (y == q.get_y()));
  }else{
    b = false;
  }
  return b;
}
```

Attention : la notion d'égalité dans un contexte où il y a du polymorphisme peut receler bien des surprises : si `Q` est un `pixel`, que se passe-t-il ? y a-t-il vraiment symétrie ? Ces questions sont assez subtiles à régler et il n'y a pas qu'une réponse à ce problème.

8.3 Comparabilité

Il est fréquent d'avoir des classes dans lesquelles une notion d'ordre entre les instances existe. Cela permet, par exemple, de faire le tri d'un tableau de telles instances par rapport à l'ordre que l'on considère.

En Java, la convention utilisée consiste à l'implémenter l'interface `Comparable`[1], qui est pourvue d'une méthode `public int compareTo(Object o)` telle que, si une instance M_1 d'une classe C_1 implémentant cette interface est comparé à une instance M_2 d'une classe C_2 par M_1`.compareTo(`M_2`)` :

- si $M_1 < M_2$ pour l'ordre choisi, M_1`.compareTo(`M_2`)` renvoie un entier négatif ;
- si $M_1 = M_2$ pour l'ordre choisi, M_1`.compareTo(`M_2`)` renvoie zéro ;
- si $M_1 > M_2$ pour l'ordre choisi, M_1`.compareTo(`M_2`)` renvoie un entier positif ;
- dans le cas où M_1 et M_2 ne sont pas de classes comparables, une `ClassCastException` est levée.

La classe `String` implémente `Comparable` en définissant la méthode `compareTo()` comme représentant l'ordre lexicographique (celui du dictionnaire). Ainsi, on a :

Par exemple :

```
String[] S = new String[]{"foo","bar","gee"};
System.out.println("foo".compareTo("bar")); // renvoie 4 (nombre de lettres entre c et f)
System.out.println("foo".compareTo("foo")); // renvoie 0
System.out.println("bar".compareTo("foo")); // renvoie -4
Array.sort(S);                             // trie le tableau dans l'ordre alphabétique
```

8.4 Copie et clonage

Les constructeurs de copie que nous avons définis jusqu'à présent avaient pour vocation de créer une nouvelle instance d'une classe à partir d'une instance existante. Cependant, à la différence de C++ dans lequel ce mécanisme est satisfaisant, Java ne permet pas de manipuler directement des objet mais uniquement des références sur ces objets. Il s'ensuit un certain nombre de subtilités qui perturbent le polymorphisme si l'on utilise un constructeur de copie.

Le mécanisme prévu par Java est donc de redéfinir la méthode `clone()` d'`Object` avec, pour objectif, le contrat d'implémentation suivant :

1. `x.clone().getClass() == x.getClass()` doit être `true` (du même type) ;
2. `x.clone().equals(x)` doit être `true` (sémantiquement équivalentes) ;
3. on peut modifier x et x`.clone()` indépendemment.

Si x est une instance de la classe C, et que `clone()` n'a pas été redéfinie, x`.clone()` renvoie une *copie superficielle* de x de la manière suivante :

- tous les champs de type primitif de x sont dupliqués dans la nouvelle instance ;
- tous les champs de type non-primitif référencent les instances de ces champs définies dans x.

Cela permet de garantir les conditions 1 et 2 du contrat d'implémentation mais pas toujours la condition 3. En effet, si la classe n'a que des champs dont les valeurs sont de type primitif ou immuable (c'est-à-dire dont la valeur ne peut être modifiée, comme les `String`), cela ne pose pas de problème. Pour les autres valeurs, deux cas se présentent :

1. les références partagées entre objets clonés, qui sont modifiées simultanément pour toutes les instances ;
2. les références à des objets qui doivent être clonés également afin que la modification d'une instance n'affecte pas ses clones.

Trois conditions sont nécessaires pour pouvoir utiliser `clone()` convenablement :

- implémenter l'interface `Cloneable`[2] car dans `Object`, il y a un mécanisme de protection contre le clonage « par hasard » :

[1] On peut également utiliser un `Comparator`...

[2] On devrait écrire `Clonable` ! Le standard est en cours d'évolution en ce sens.

```
public class Object{...
  protected Object clone() throws CloneNotSupportedException{
    if (!(this instanceof Cloneable)){throw new CloneNotSupportedException();}
  ...}
...}
```

– redéfinir la méthode `clone()` pour la classe `C` en la rendant `public` (pour pouvoir la cloner de n'importe-où : elle est initialement `protected`) et en captant l'exception `CloneNotSupportedException` qui ne sera pas levée puisqu'on implémente `Cloneable` :

```
class A{...
  public Object clone(){
                                A y = null;
    try{
      y = (A)super.clone();
      // Début du clonage des champs à dupliquer dans y
      ...
      // Fin du clonage des champs: y est prêt
    }catch(CloneNotSupportedException e){}
    return y;
  }
...}
```

– Downcaster systématiquement les instances de type `Object` retournées par `clone()` (la rédéfinition impose à cette méthode de renvoyer un `Object`)

Pour prendre un exemple, définissons, dans un premier temps, un point clonable :

```
class point implements Cloneable{
  private int x,y;

  public final void set_x(int x){this.x=x;}
  public final void set_y(int y){this.y=y;}

  public point(int x,int y){set_x(x); set_y(y);}
  public point(){this(0,0);}

  public Object clone(){
    point q = null;
    try{
      q = (point)super.clone();
    }
    catch(CloneNotSupportedException e){}
    return q;
  }
  public String toString(){return "(" + x + ", " + y + ")";}
}
```

Désormais, il est facile de fabriquer un rectangle clonable en définissant la façon de cloner les différents attributs (ici deux sommets sud-ouest et nord-est, ce qui caractérise un rectangle dont les côtés sont parallèles aux axes), à partir des méthodes de clonage de chacun d'entre-eux :

```
class rectangle implements Cloneable{
  private point SO, NE;

  public final void setSO(point A){SO=A;}
  public final void setNE(point A){NE=A;}
  public rectangle(){
    setSO(new point());
    setNE(new point());
  }
```

```
  public Object clone(){
    rectangle S=null;
    try{
      S = (rectangle)super.clone();
      S.setSO((point)SO.clone());
      S.setNE((point)NE.clone());
    }
    catch(CloneNotSupportedException e){}
    return S;
  }

  rectangle(point SO, point NE){
    setSO(SO); setNE(NE);
  }

  public String toString(){return "" + SO + " - " + NE;}

}
```

On peut s'assurer en exécutant :

```
point SO = new point(10,20);
point NE = new point(30,40);
point A = (point)SO.clone();
A.set_x(50);
System.out.println(SO);
System.out.println(A);
rectangle R = new rectangle(SO,NE);
System.out.println(R);
rectangle S = (rectangle)R.clone();
S.setSO(A);
System.out.println(R);
System.out.println(S);
```

que la sortie attendue est bien :

```
(10, 20)
(50, 20)
(10, 20) - (30, 40)
(10, 20) - (30, 40)
(50, 20) - (30, 40)
```

Chapitre 9

Conteneurs et itérateurs

9.1 L'interface `Collection`

Lorsqu'on souhaite regrouper différentes valeurs au sein d'un même aggrégat, plusieurs choix sont possibles. Les tableaux sont, bien entendu, les plus fréquemment utilisés mais Java propose, en standard, un certain nombre de *collections* ayant chacune des propriétés adaptées à telle ou telle situation. Ces collections sont représentées par une interface `Collection`[1] dont les méthodes sont indiquées dans la Table 9.1. Dérivent de cette interface les interfaces :

- `Set` désignant une collection dans laquelle chaque élément n'apparaît qu'une fois et qui possède également une interface fille : `SortedSet` pour les ensembles dont les éléments sont ordonnés ;
- `List` désignant une collection dans laquelle chaque élément peut apparaître plusieurs fois, dans un ordre donné.
- `Map` désignant une application définie par un tableau associatif de couples (clé,valeur), et qui possède également une interface fille : `SortedMap` pour les tableaux associatifs ordonnés ;

TAB. 9.1 – Méthodes de l'interface `Collection`. Les différents éléments des collections sont accessibles par une interface d'*itérateurs*, du type `Iterator`. Pour les autres méthodes, les noms sont suffisamment explicites pour ne pas nécessiter une explication.

`void boolean add(Object o)`
`void boolean addAll(Collection c)`
`void clear()`
`boolean contains(Object o)`
`boolean containsAll(Collection c)`
`boolean equals(Object o)`
`int hashCode()`
`boolean isEmpty()`
`Iterator iterator()`
`boolean remove(Object o)`
`boolean removeAll(Collection c)`
`boolean retainAll(Collection c)`
`int size()`
`Object[] toArray()`
`Object[] toArray(Object[] t)`

[1] La gestion de ces collections a changé dans la version 1.5, le propos de ce paragraphe utilise encore les versions antérieures.

9.2 L'interface `Iterator`

Cette interface sert à se déplacer dans la `Collection` associée, en utilisant les méthodes :
- `boolean hasNext()` qui renvoie vrai si l'on n'a pas atteint la fin de l'énumération ;
- `Object next()` qui renvoie l'élément suivant (qu'il conviendra de downcaster, le cas échéant) si `hasNext() == true` et qui lève une `NoSuchElementException` sinon.

On peut aussi s'en servir pour modifier l'objet localement, à la position courante, en invoquant :
- `void add(Object o)`
- `void remove()`

Deux appels successifs à `remove()` sans `next()` lèvent une `IllegalStateException`. L'utilisation classique d'un itérateur, sur une collection `K` d'objets de type `A`, est la suivante :

```
Iterator it = K.iterator();
while(it.hasNext()){
  x = (A)it.next();        // downcast ou ClassCastException
  System.out.println(x);   // par exemple
}
```

9.3 Représentation des ensembles

9.3.1 L'interface `Set` et sa dérivée `SortedSet`

Comme on l'a vu, les interfaces `Set` et sa dérivée `SortedSet` représentent des collections où chaque élément n'apparaît qu'une fois.

9.3.2 L'implémentation `HashSet` de `Set`

La classe `HashSet` implémente l'interface `Set` avec une table de hachage. Celle-ci permet de tester l'appartenance, d'ajouter et d'enlever un élément très rapidement. Pour réaliser le test d'appartenance, il est nécessaire que les méthodes `hashCode()` et `equals()` des éléments de cet ensemble aient une sémantique convenable.

On donne ici un exemple d'utilisation de `HashSet`, les autres collections s'utilisent de la même façon.

hash.java

```
import java.util.*;
class hash{
  public static void main(String[] args){
    HashSet X = new HashSet();

    X.add("Liberté");
    X.add("Égalité");
    X.add("Fraternité");

    Iterator it = X.iterator();
    while(it.hasNext()){
      System.out.println((String)(it.next()));
    }
  }
}
```

La sortie, comme prévu, ne respecte pas l'ordre dans lequel on a entré les données :

```
Fraternité
Égalité
Liberté
```

9.3.3 L'implémentation `TreeSet` de `SortedSet`

La classe `TreeSet` implémente l'interface `SortedSet` avec un arbre binaire équilibré. Celui-ci permet de tester l'appartenance, d'ajouter et d'enlever un élément un peu moins rapidement que dans un `HashSet`, mais de toujours ranger les éléments dans l'ordre. Pour réaliser le tri des éléments, il est nécessaire que l'interface `Comparable` soit implémentée et que la méthode `compareTo()` des éléments de cet ensemble ait une sémantique convenable.

9.4 Représentation des listes

9.4.1 L'interface `List`

Comme on l'a vu, l'interface `List` représente les collections où chaque élément peut apparaître plusieurs fois, à une position différente. Cette interface propose également la série de méthodes définie dans la Table. 9.2.

TAB. 9.2 – Méthodes de l'interface `List`. Les différents éléments des listes sont accessibles par une interface du type `ListIterator`, héritée d'`Iterator`. Pour les autres méthodes, les noms sont suffisamment explicites pour ne pas nécessiter une explication.

`void add(int index, Object element)`
`boolean addAll(int index, Collection c)`
`int indexOf(Object o)`
`int lastIndexOf(Object o)`
`ListIterator listIterator()`
`ListIterator listIterator(int index)`
`Object remove(int index)`
`Object set(int index, Object element)`
`List subList(int fromIndex, int toIndex)`

9.4.2 L'implémentation `ArrayList` de `List`

La classe `ArrayList` implémente l'interface `List` avec un tableau. Celui-ci permet d'accéder à l'élément i très rapidement mais sera plus lent lorsqu'il s'agira d'insérer ou de supprimer un élément en début de liste (copie de la fin du tableau).

9.4.3 L'implémentation `LinkedList` de `List`

La classe `LinkedList` implémente l'interface `List` avec une liste doublement chaînée. Celle-ci permet d'accéder à l'élément i assez lentement si celui-ci est au milieu de la liste (parcours de toute la demi-liste), mais permettra d'insérer ou de supprimer très rapidement un nouvel élément.

9.5 Représentation des applications

9.5.1 L'interface `Map` et sa dérivée `SortedMap`

Comme on l'a vu, les interfaces `Map` et `SortedMap` représentent les applications définies par un tableau associatif de couples (clé,valeur), triée, ou non.

9.5.2 Les implémentations `HashMap` (et `WeakHashMap`) de `Map`

Les classes `HashMap` (et `HashWeakMap`) implémentent l'interface `Map` par une table de hachage des clés avec des propriétés comparables à ce qui se passe pour les `HashSet`.

9.5.3 Les implémentations `TreeMap` de `SortedMap`

La classe `TreeMap` implémente l'interface `SortedMap` par un arbre binaire équilibré dont les nœuds sont étiquetés par les clés, avec des propriétés comparables à ce qui se passe pour les `TreeSet`.

Chapitre 10

Étude détaillée de la classe String

10.1 `String`, `char[]`, `StringBuffer`

s`.atChar(i)` désigne le i ième caractère de la `String` s.

10.2 Fonctions de base

On a `length()` et `concat()` qui équivaut à `+`. `+=` retourne un nouvel objet et ne peut pas modifier l'instance.

10.3 Conversions

On peut passer de `String` à un tableau de `char` (ce dernier étant modifiable) de la manière suivante :

```
char[] T = {'a','b','c','d','e'};
String S = new String(T);
char[] U = S.toCharArray();
```

la méthode `equals` et `equalsIgnoreCase` renvoient des booléens et la méthode `compareTo()` renvoie 0 si les chaînes sont égales, -1 si la première est inférieure à la seconde et 1 si la première est supérieure à la seconde, l'ordre étant lexicographique.

La méthode `valueOf()` utilise la méthode `toString()` des objets.

```
String un_deux_trois = String.valueOf(123);
int cent_vingt_trois = Integer.valueOf(un_deux_trois).intValue();
```

```
StringBuffer t1 = new StringBuffer("toto");
StringBuffer t2 = t1;
System.out.println(s1 == s2); // true
t1.append("!");  // l'objet pointé par s1 est modifié
```

10.4 Tests d'égalité

```
StringBuffer t1 = new StringBuffer("toto");
StringBuffer t2 = t1;
System.out.println(t1 == t2); // true
t1.append("!");  // l'objet pointé par s1 est modifié
System.out.println(t1 == t2); // true
System.out.println(t1); // "toto!"
System.out.println(t2); // "toto!"
```

```
String s1 = "toto";
String s2 = s1;
System.out.println(s1 == s2); // true
s1 += "!";                    // s1 pointe sur une nouvelle chaîne "toto!"
System.out.println(s1 == s2); // false
System.out.println(s1); // "toto!"
System.out.println(s2); // "toto"

String u1 = "toto";
String u2 = "toto";
System.out.println(u1 == u2); // true!!! la chaîne est détectée comme présente et partagée
```

10.5 Recherche de motifs (*pattern matching*)

On a les méthodes suivantes :

TAB. 10.1 – Recherche de motifs avec `String`

méthode Java	spécification
`boolean startsWith(String` s`)`	retourne `true` ssi la chaîne commence par s
`boolean endsWith(String` s`)`	retourne `true` ssi la chaîne finit par s
`int indexOf(String` s`)`	renvoie la position de la chaîne s
`int indexOf(char` c`)`	renvoie la position du caractère c
`String replace(char` c`, char` d`)`	renvoie la chaîne après remplacement des c en d

10.6 Manipulation de chaînes

`trim` supprime les espaces `toUpperCase` met la chaîne en majuscules `toLowerCase` met la chaîne en minuscules `substring` retourne une chaîne située entre deux positions

10.7 Analyseur lexical

```
String s = "Prospere, youpla boume!";
java.until.StringTokenizer st = new java.until.StringTokenizer(s);
while (st.hasMoreTokens()){System.out.println(st.nextToken());}
/* affiche:
Prospere,
youpla
boume!
*/
st = new java.until.StringTokenizer(s,","); // le séparateur est ","
/* affiche:
Prospere,
 youpla boume!
*/
```

10.8 Récupération d'une page web

Une page web se récupère avec :

```
String page = (String)new URL("http://www-rocq.inria.fr/~pecquet").getContent();
```

Chapitre 11

Threads et interface `Runnable`

11.1 Processus légers = fils = *threads*

La notion de processus correspond à du parallélisme à mémoire distribuée : chaque processus ayant sa propre zone de mémoire pour travailler (lors d'un `fork()`, toute la zone mémoire d'un processus est ainsi copiée).

Au sein d'un même processus, il est possible de définir plusieurs *threads* (chacun suivant le *fil* de son exécution, d'où ce choix terminologique).

Pour créer un nouveau thread T, il suffit d'étendre la class `Thread` et de redéfinir sa méthode `public void run()`. Le début de la vie du thread est provoqué par l'invocation de T`.start()`. On peut suspendre celui-ci par la méthode `wait()` et le réveiller par la méthode `notify()` (et réveiller tous les threads endormis avec `notifyAll`).

Un thread peut également être endormi pendant une durée déterminée avec l'instruction `Thread.sleep(`m`)` où m désigne un nombre de microsecondes donné.. On notera qu'il n'y a pas d'autre moyen, en Java, d'attendre une durée déterminée, sauf à faire de l'attente active en exécutant une boucle de quelques dizaines de milliers d'instructions inutiles en attendant.

Ces *threads* disposent chacun des même variables et il est nécessaire de disposer de mécanismes de protection pour éviter qu'ils y accèdent simultanément. Cela est fait en mettant le mot-clé *synchronized* devant chaque méthode critique, ce qui empêche à deux méthodes `synchronized` de s'exécuter simultanément.

11.2 Exemple : le dîner des philosophes

Voici comment implémenter le célèbre dîner des philosophes de Dijkstra en quelques lignes de Java. Il s'agit de faire manger cinq philosophes alternativement sachant que chacun d'eux a besoin de sa fourchette de gauche et de sa fourchette de droite et qu'il n'y a qu'une fouchette intercalée entre chaque philosophes sur la table ronde qui les accueille.

Chaque philosophe sera contrôlé par un moniteur, c'est-à-dire un mini-gestionnaire de tâches de type `Monitor` que nous allons définir. La forme d'un philosophe sera la suivante :

```
class Philosophe extends Thread{
  private int i;
  private Monitor M;

  Philosophe(int i, Monitor M){this.i=i; this.M=M;}

  public void run(){
    try{
      while(true){
        M.set_affame(i);
        System.out.println(Thread.currentThread().getName() + " mange");
        System.out.println(M);
```

```
        Thread.sleep(600000*(long)Math.random());
        M.set_repus(i);
        System.out.println(Thread.currentThread().getName() + " pense");
        System.out.println(M);
        Thread.sleep(600000*(long)Math.random());
      }
    }catch(InterruptedException e){System.out.println(e + ": réveil prématuré");}
  }
}
```

où l'on a préalablement défini le moniteur comme étant :

```
class Monitor{
  private static final int libre = -1;
  private int[] fourchette;
  private boolean[] mange;

  Monitor(){
    fourchette = new int[5];
    for(int i=0;i<5;i++){
      fourchette[i] = libre;
    }
    mange = new boolean[5];
    for(int i=0;i<5;i++){
      mange[i] = false;
    }
  }

  public String toString(){
    String s = "fourchettes = [";
    for(int i=0;i<4;i++){
      s += (fourchette[i] != libre)?("" + fourchette[i] + ", "):"*, ";
    }
    s += ((fourchette[4] != libre)?("" + fourchette[4]):"*") + "]";

    s += ", convives = [";
    for(int i=0;i<4;i++){
      s += (mange[i])?("M, "):"P, ";
    }
    s += (mange[4])?("M]"):"P]";
    return s;
  }

  // le moniteur rend le philosophe affamé:
  public synchronized void set_affame(int i) throws InterruptedException{
    while(fourchette[i] != libre || fourchette[(i+1)%5] != libre){wait();}
    System.out.println("Le philosophe " + i + " saisit les fourchettes "
                       + i + " et " + ((i+1)%5));
    mange[i] = true; fourchette[i] = fourchette[(i+1)%5] = i;
  }

  // le moniteur rend le philosophe repus:
  public synchronized void set_repus(int i) throws InterruptedException{
    mange[i] = false; fourchette[i] = fourchette[(i+1)%5] = libre;
    System.out.println("Le philosophe " + i + " libère les fourchettes "
                       + i + " et " + ((i+1)%5));
    notifyAll();
  }
}
```

On utilise ce qui précède ainsi :

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

```
    Monitor M = new Monitor();
    Philosophe[] convives = new Philosophe[5];
    for(int i=0;i<5;i++){
      convives[i] = new Philosophe(i,M);
      convives[i].setName("Le Philosophe " + i);
      convives[i].start();
    }
  }
}
```

dont le résultat est :

```
$ javac philosophes.java; java philosophes
Le philosophe 3 saisit les fourchettes 3 et 4
Le Philosophe 3 mange
fourchettes = [*, *, *, 3, 3], convives = [P, P, P, M, P]
Le philosophe 0 saisit les fourchettes 0 et 1
Le Philosophe 0 mange
fourchettes = [0, 0, *, 3, 3], convives = [M, P, P, M, P]
Le philosophe 3 libère les fourchettes 3 et 4
Le Philosophe 3 pense
fourchettes = [0, 0, *, *, *], convives = [M, P, P, P, P]
Le philosophe 0 libère les fourchettes 0 et 1
Le Philosophe 0 pense
fourchettes = [*, *, *, *, *], convives = [P, P, P, P, P]
Le philosophe 1 saisit les fourchettes 1 et 2
Le Philosophe 1 mange
fourchettes = [*, 1, 1, *, *], convives = [P, M, P, P, P]
```

Chapitre 12

Application Programming Interface (API) Java

12.1 Introduction

L'API Java est gigantesque. Nous n'en verrons que quelques aspects ici.

12.2 Entrées/sorties `java.io`

12.2.1 Entrée et sortie standard

Si la sortie standard a été largement utilisée avec `System.out.println()`, force est de constater que l'entrée standard est nettement plus difficile à manipuler en Java. Voici un petit exemple :

```
import java.io.*;

BufferedReader console = new BufferedReader(new InputStreamReader(System.in));
System.out.println("vous vous prénommez? ");
String nom = null;
try{nom = console.readLine();}catch(IOException){} // Pas d'erreur en théorie...
System.out.println("Bonjour " + nom);
```

12.2.2 La classe `File`

La classe `File` bénéficie des méthodes définies dans la Table 12.1.

12.2.3 Lecture d'un fichier texte

La lecture d'un fichier texte se fait facilement :

```
import java.io.*;

BufferedReader console = new BufferedReader(new FileReader("toto.txt"));
String ligne = null;
try{
    while((ligne = in.readLine()) != null){
      System.out.println(ligne);
    }
    in.close();
  }catch(IOException){} // Pas d'erreur en théorie...
```

TAB. 12.1 – Méthodes de la classe `File`

méthode Java	spécification
`boolean exists()`	`true` ssi le fichier/répertoire existe
`boolean canRead()`	`true` ssi il est possible de lire dans le fichier
`boolean canWrite()`	`true` ssi il est possible d'écrire dans le fichier
`void delete()`	efface le fichier
`String getAbsolutePath()`	le chemin absolu du fichier/répertoire
`String getName()`	le chemin absolu du fichier/répertoire
`String getParent()`	le chemin absolu du répertoire parent
`boolean IsDirectory()`	`true` ssi c'est un répertoire
`boolean IsFile()`	`true` ssi c'est un fichier
`long lastModified()`	date de dernière modification
`long length()`	taille
`long list()`	liste des fichiers du répertoire

12.2.4 Écriture dans un fichier texte

L'écriture dans un fichier texte se fait facilement :

```
import java.io.*;

File f = new File("toto.txt");

PrintWriter out = new PrintWriter(new FileWriter(f));
try{
      out.println("Coucou");
    }
    out.close();
  }catch(IOException){} // Pas d'erreur en théorie...
```

12.2.5 Lecture/écriture dans un fichier à accès aléatoire

On peut également lire et/ou écrire dans des fichiers à accès aléatoire avec : :

```
try{
     RandomAccessFile fichier = new RandomAccessFile("foo.txt","rw");
     fichier.writeUTF("Coucou!");
}catch(Exception e){
     System.out.println(e);
}
```

12.3 Mathématiques `java.math`

Les méthodes mathématiques de base sont définies dans la classe `Math` et rappelées dans la Table 12.2.

12.4 Paquetages utilitaires `java.util`

12.4.1 Nombres aléatoires `java.util.Random`

On crée un générateur pseudo-aléatoire `R` en écrivant :

TAB. 12.2 – Méthodes de la classe `Math`

Méthode java	fonction mathématique
`Math.abs(`x`)`	$\lvert x\rvert$
`Math.sqrt(`x`)`	$\sqrt{x}$
`Math.sin(`x`)`	$\sin x$
`Math.cos(`x`)`	$\cos x$
`Math.tan(`x`)`	$\tan x$
`Math.asin(`x`)`	$\arcsin x$
`Math.acos(`x`)`	$\arccos x$
`Math.atan(`x`)`	$\arctan x$
`Math.exp(`x`)`	$\exp x$
`Math.log(`x`)`	$\log x$ en (néperien)
`Math.max(`x`,`y`)`	$\max(x, y)$
`Math.pow(`x`,`y`)`	$x^y = \exp(y \log x)$
`Math.floor(`x`)`	$\lfloor x \rfloor = \max\{n \ : n \in \mathbb{N} \mid n \leq x\}$
`Math.ceil(`x`)`	$\lceil x \rceil = \min\{n \ : n \in \mathbb{N} \mid n \geq x\}$
`Math.round(`x`)`	entier le plus proche avec `round(0.5)` $= 1$
`Math.random()`	*cf.* `nextDouble()` Section 12.4.1, p. 70

```
Random R = new Random();
```

Il est initialisé sur l'heure courante (mais on peut préciser une valeur au constructeur pour servir de graine) et on dispose ensuite les variables aléatoires suivantes : `R.nextInt()` et `R.nextLong()` qui suivent la loi uniforme sur les `int` et les `long`, `R.nextFloat()` et `R.nextDouble()` qui suivent la loi uniforme sur les `float` et les `double` représentant des réels de l'intervalle $[0, 1]$ et `R.nextGaussian()` qui suit la loi gaussienne d'espérance 0.0 et de variance 1.0 sur les `double`.

12.4.2 Analyse lexicale `java.util.StringTokenizer`

On peut très facilement définir un analyseur syntaxique avec `StringTokenizer`

```
String s = "Prospere, youpla boume!";
java.util.StringTokenizer st = new java.util.StringTokenizer(s);
while (st.hasMoreTokens()){System.out.println(st.nextToken());}
/* affiche:
Prospere,
youpla
boume!
*/
st = new java.until.StringTokenizer(s,","); // le séparateur est ","
/* affiche:
Prospere,
 youpla boume!
*/
```

12.5 Applets `java.applets`

Une applet est destinée à être vue à travers un navigateur web, suite à une connexion réseau. Une applet a donc des restrictions en matière de sécurité et ne peuvent pas :

1. accéder au système de fichier ;

2. communiquer avec un autre site Internet que celui sur lequel l'applet a été chargé ;
3. exécuter un autre programme que le sien ;

Il faut construire un fichier HTML qui appellera le code de l'applet, comme par exemple :

hello_world.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html lang="en">

  <head>
  <meta http-equiv="Content-Type" content="text/html"; charset="iso-8859-1">
  <title>Hello World Applet</title>
  </head>

  <body>

    <div style="text-align:center;">
    <!-- HelloWorldApplet.class est située dans le répertoire mes_applets: -->
    <applet code="HelloWorldApplet.class" codebase="mes_applets" width="600" height="400">

      <!-- on peut passer une option au programme (souvent issue d'un formulaire): -->
      <param name=taille_police value=40>

      <!-- le message ci-dessous ne sera visible que si Java est inactif -->
      Java est inactif: le programme ne peut être exécuté.
    </applet>
    </div>

  </body>
</html>
```

Simultanément, on écrit le code du fichier `HelloWorldApplet.java` :

HelloWorldApplet.java

```
import java.awt.Graphics;
import java.awt.Font;
import java.awt.FontMetrics;
import java.awt.Color;
import java.util.Random;

public class HelloWorldApplet extends java.applet.Applet{

    final int x_max = 600, y_max = 400;

    // Emplacement où l'on va écrire depuis le coin en haut à gauche:
    int x=0,y=0;
    Font f = null;
    String le_texte="Hello World!";
    Random r = new Random();

    // On redéfinit la méthode init qui organise la création de l'applet:
    public void init(){
        setBackground(Color.white);
        int taille = Integer.parseInt(getParameter("taille_police"));
        if (taille <= 0){taille = 10;}
        f = new Font("Courier",Font.BOLD,taille);
        FontMetrics fm = getFontMetrics(f); // Informations sur la police
        x = (x_max - fm.stringWidth(le_texte))/2;
        y = (y_max - fm.getHeight())/2;
    }

    /*
    // On pourrait aussi redéfinir:
    public void start(){}   // définit le comportement à chaque chargement de la page web
    public void stop(){}    // définit le comportement à chaque départ de la page web
```

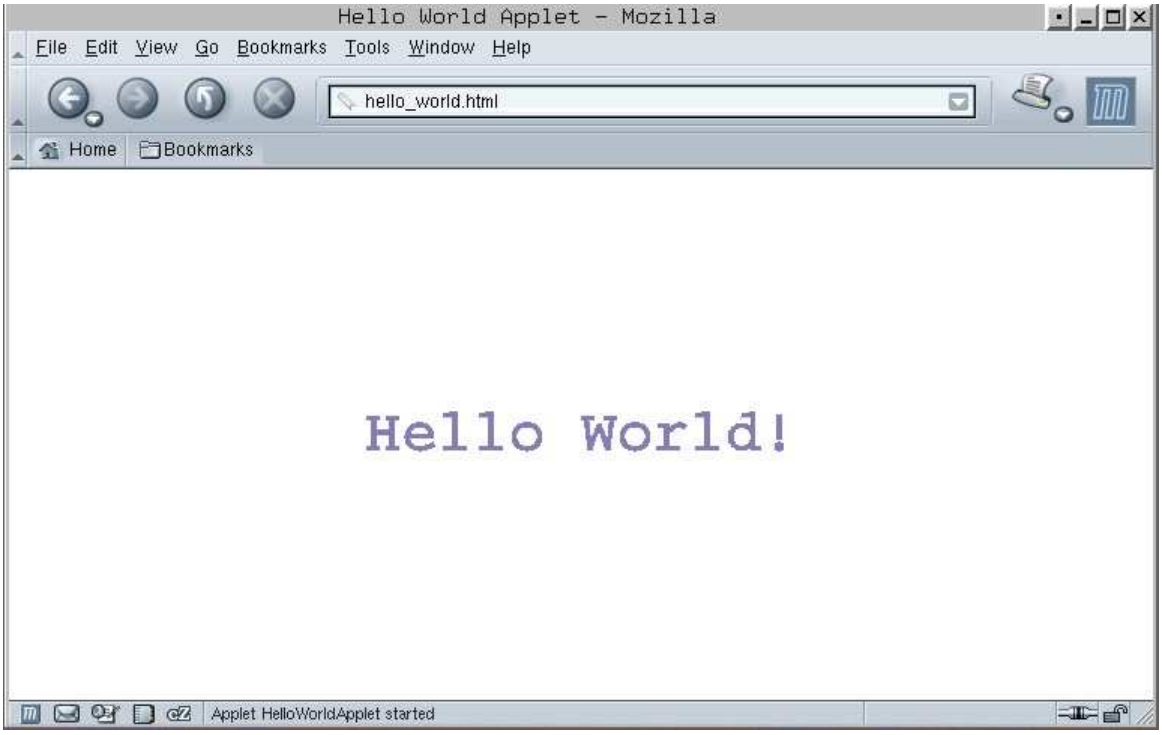


FIG. 12.1 – Visualisation du fichier `hello_world.html` appelant l'applet `HelloWorldApplet.class` dans le navigateur Mozilla. Le texte prend une couleur 32 bits aléatoire toutes les 200 ms environ.

```
    public void destroy(){} // organise la destruction de l'applet
    */

    // La méthode suivante inclut le "main"
    public void paint(Graphics screen){
        screen.setFont(f);
        while(true){
            screen.setColor(new Color((int)(r.nextDouble()*255),(int)(r.nextDouble()*255),
                                      (int)(r.nextDouble()*255)));
            screen.drawString(le_texte,x,y);
            try{Thread.sleep(200);} // dort au moins 200ms entre chaque tour de boucle
            catch (InterruptedException e){System.out.println("Réveil inattendu");}

        }
    }
}
```

Le résultat est visible dans la Fig. 12.1.

Chapitre 13

Conclusion et lectures suggérées

Le très bon [1] constitue une introduction pédagogique et approfondie au langage Java. Les manuels de référence officiels de Sun sont [3] et [4]. Une référence exhaustive se trouve dans [2].

Index

R

S

T

U

V

W

X

Bibliographie

[1] Ken Arnold, James Gosling, and David Holmes. *Le Langage Java.* Vuibert, 3ème edition, 2001.

[2] David Flanagan. *Java in a nutshell, manuel de référence.* O'Reilly, 4ème edition, 2002.

[3] Cay S. Horstmann and Gary Cornell. *Au cœur de Java 2, Notions fondamentales*, volume 1. Campus Press, Sun Microsystems, 2ème edition, 2001.

[4] Cay S. Horstmann and Gary Cornell. *Au cœur de Java 2, Fonctions avancées*, volume 2. Campus Press, Sun Microsystems, 2ème edition, 2002.

[5] Jon Meyer and Troy Downing. *Java Virtual Machine.* O'Reilly, 1997.

[6] Unicode. Wikipedia, the Free Encyclopedia, `http://en.wikipedia.org/wiki/Unicode`.