LE LANGAGE JAVA

TODO :

- chapitre 3.1 : visibilité des attributs et méthodes
- chapitre 6.4 : exclusion mutuelle et synchronisation des threads
- chapitre 6.4 : déprécation des méthodes stop(), suspend(), resume() et procédés alternatifs
- chapitre 8 (jdbc) : à faire
- autres chapitres (rmi, jni, jsp, servlet, …) : à faire
- classes internes (+visibilité) ?
- XML ?

1.1.0.1 – 07/05/2010
peignotc(at)arqendra(dot)net / peignotc(at)gmail(dot)com

TABLE DES MATIÈRES

TABLE DES ANNEXES

TABLE DES ILLUSTRATIONS

0 PROLOGUE

Le contenu du présent document se focalise principalement sur les aspects de Programmation Orientée Objet appliqués au langage Java et tient pour acquis un certain nombre de concepts et d'éléments de base de la programmation et communs à différents langages, tels que le langage C, C++, C#, etc. (nda : voir le cours *Le langage C* ou *Le langage C++* du même auteur).

La maîtrise de notions telles que les principes de « mot-clef », d'« instruction », et de « variable », les structures de contrôle, les tableaux, les fonctions, les pointeurs et les types de variables complexes – principalement les structures – est impérative pour appréhender les nouveaux éléments amenés.

1 INTRODUCTION AU LANGAGE JAVA

Java est un langage de programmation orienté objet qui a été inventé au début des années 90 et dont les droits sont détenus par la société Sun Microsystems [1]. La création du langage provient des résultats obtenus avec le C++ dans l'élaboration de programmes de domotique qui furent jugés insatisfaisants.

Très vite, pendant sa conception, il s'est avéré que Java convenait aussi parfaitement à la programmation multimédia, et au monde de l'internet qui commençait à éclore et à se faire connaître du grand public.

Depuis sa création, le langage a subi de nombreuses évolutions et corrections. Les mises-à-jour régulières [2] ont permis au fur et à mesure d'inclure de nombreuses bibliothèques, permettant de développer plus facilement des applications complexes.

La manière d'exploiter un code source Java est singulière : le code source est d'abord compilé en utilisant un compilateur Java ; le résultat de cette compilation est ensuite interprété par la machine cible, au moyen d'un interpréteur Java.

Ce concept d'exploitation [3] découle de la volonté de produire un langage le plus indépendant possible des spécificités du hardware, tout en conservant les avantages d'un langage comme le C++ ; sont ainsi conjugués les avantages d'un langage compilé ainsi que ceux d'un langage interprété.

Pour exécuter un programme Java, il est donc nécessaire et suffisant d'installer sur la machine cible un interpréteur Java, plus communément appelé *machine virtuelle Java* ou JVM [4], qui a en charge l'interfaçage entre les réquisitions de l'application Java et les spécificités hardware et/ou software de la plate-forme d'exécution [5].

Une fois installée, celle-ci permet alors l'exécution de toute application développée en Java, qu'il s'agisse d'applications de type « console » (simple texte) ou d'applications graphiques ; elle peut aussi être intégrée au navigateur et permettre alors l'exécution d'applets [6].

La machine virtuelle Java est généralement intégrée et disponible sous la forme d'un environnement d'exécution complet [7] comprenant la JVM et un ensemble de bibliothèques Java.

Les avantages du langage Java sont :

- langage tout objet (à l'exception des types primitifs) ;
- grande robustesse (langage fortement typé, gestion automatique de la mémoire, gestion des erreurs, etc.) ;
- portabilité (aucune recompilation nécessaire : une fois le code Java compilé, son exécution sans erreurs ne dépend que de la JVM [8]) ;
- gestion de la sécurité ;
- forte capacité d'intégration aux environnements web ;
- facilité d'écriture d'interfaces graphiques professionnelles.

Les inconvénients du langage Java sont :

- vitesse d'exécution généralement inférieure à un langage compilé en natif (tel que C++) ;
- difficultés d'accès aux ressources matérielles.

---

[1] Les concepteurs du langage Java sont James Gosling et Patrick Naughton, employés de Sun Microsystems.

[2] La version actuelle (juillet 2009) est la « 6 update 13 ».

[3] Concept repris par le langage C# et le framework .Net.

[4] JVM : Java Virtual Machine (eng) ≡ Machine Virtuelle Java (fr).

[5] Un même code source Java, une fois compilé, peut donc être exécuté sous Windows, Linux, MacOS, Solaris, etc.

[6] Applet : application graphique pouvant s'exécuter dans une page web.

[7] Appelé JRE (Java Runtime Environment (eng) ≡ environnement d'exécution Java (fr)), parfois abrégé en JSE ou J2SE (Java Standard Edition / Java2 Standard Edition (eng) ≡ Édition Standard de Java / Édition Standard de Java2 (fr)). Pour les devéloppeurs, il faut se tourner vers le JDK (Java Development Kit (eng) ≡ Kit de Développement Java (fr)) qui comprend le JRE et l'environnement de développement.

[8] « Write once, run everywhere » (eng) ≡ « Écrire une fois, s'exécuter partout » (fr).

2 NOTIONS DE BASE

2.1 LA PROGRAMMATION ORIENTÉE OBJET

Java est un langage de programmation **tout objet**, à la syntaxe proche du C++. On ne manipule donc que des objets.

2.1.1 Généralités sur l'objet

Le terme **objet** désigne une entité informatique spécifique, qui est une représentation abstraite simplifiée d'un objet concret et abstrait du monde réel ou virtuel. Un objet appartient à une famille d'objets et possède des variables associées et des fonctions associées ; en terminologie objet, on parle respectivement de **classe** (famille d'objets), **attributs** [1] (variables) et **méthodes** (fonctions).

Lorsque l'on fait référence à une classe, le terme de **membres** désigne les attributs et les méthodes de cette classe.

Ex. : Soit la classe `Vehicule` représentant la notion de véhicule du monde réel.

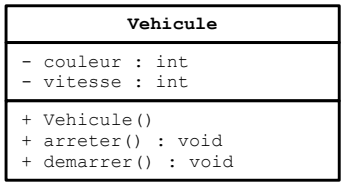


Figure 2.1 : exemple d'un objet (classe, attributs et méthodes, visibilité des membres)

La programmation orientée objet (POO) peut être définie par les trois grands principes suivants :

- l'encapsulation ;
- l'héritage de classe ;
- le polymorphisme.

2.1.2 Principes

2.1.2.1 L'encapsulation

L'**encapsulation** consiste à préserver la valeur des attributs de l'objet ; ceux-ci ne sont accessibles de l'extérieur de la classe que par l'intermédiaire des méthodes de la classe. Les attributs sont dits alors *privés* (accès direct restreint à la classe seulement) ou *protégés* (accès direct restreint à la classe et aux éventuelles sous-classes), alors que les méthodes sont généralement *publiques* [2] (accès possible à l'intérieur et à l'extérieur de la classe).

Ex. : Seules les méthodes `demarrer()` et `arreter()` peuvent modifier l'attribut `vitesse`.

Pour tout attribut qui doit être accessible de l'extérieur de la classe, on écrit en général 2 méthodes spécifiques appelées *accesseur*s ; on distingue les accesseurs en lecture, permettant de connaître la valeur d'un attribut (prototype usuel : `type_attribut getAttribut()`) et les accesseurs en écriture permettant d'affecter une valeur à un attribut (prototype usuel : `void setAttribut(type_attribut newvaleur)`).

L'attribut peut alors être considéré comme étant *public*, mais la différence par rapport à une réelle définition publique est que les accesseurs peuvent contrôler la lecture ou l'écriture de l'attribut (notamment éviter une valeur d'écriture erronée, ou bien transmettre une valeur de lecture mise à l'échelle, etc.)

[1] Parfois appelées aussi *propriétés* dans certains langages.

[2] Une méthode peut cependant être privée si elle est destinée à être appelée uniquement par d'autres méthodes de la classe.

Ex. : Les attributs de la classe `Vehicule` sont protégés grâce à la visibilité privée (−), alors que les méthodes sont publiques (+). Néanmoins, pour permettre une collaboration avec l'extérieur de la classe, on associe à l'attribut `couleur` un accesseur en lecture (`int getCouleur()`) et un accesseur en écriture (`void setCouleur(int cc)`).

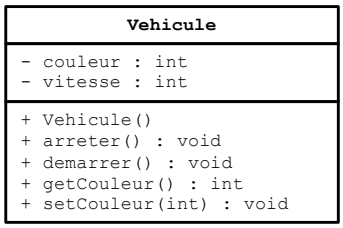


Figure 2.2 : exemple de respect de l'encapsulation grâce aux accesseurs

2.1.2.2 L'héritage de classe

L'**héritage** de classe permet de créer une nouvelle classe à partir d'une classe déjà existante. Cela permet ainsi de préciser, ou mettre à jour une classe. Du fait de l'héritage, la nouvelle classe possède automatiquement les mêmes membres que la classe déjà existante [1], lesquels peuvent être complétés par de nouveaux attributs et nouvelles méthodes. Dans le cadre de cet héritage, la classe déjà existante est appelée la *super-classe*, et la nouvelle classe est appelée la *sous-classe*.

Ex. : Soit une classe qui représente un véhicule, et deux autres représentant une voiture et un vélo. Une voiture « est une sorte de » véhicule ; un vélo est aussi « une sorte de » de véhicule. Dans les relations `Vehicule`/`Voiture` et `Vehicule`/`Velo`, `Vehicule` est donc la super-classe ; `Voiture` et `Velo` sont donc les sous-classes.

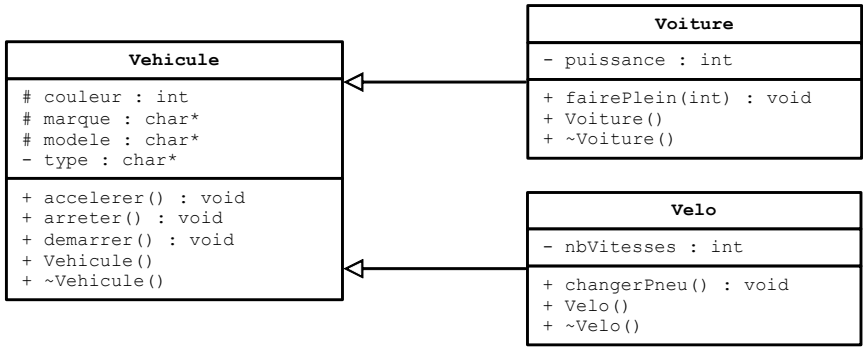


Figure 2.3 : exemple d'un héritage de classe

2.1.2.3 Le polymorphisme

Le **polymorphisme** [2] est caractérisé par la définition de plusieurs méthodes homonymes, mais dont le comportement et/ou la signature [3] diffèrent, ou bien qui s'appliquent à des types d'objets distincts.

Il existe plusieurs types de polymorphismes :

- polymorphisme ad hoc : 2 classes différentes possèdent chacune une méthode homonyme (nom et signature identiques) mais dont le comportement est différent (adapté à chaque classe) ; aussi appelé *surcharge* [4] ;

  Ex. : Les classes `Vehicule` et `Ordinateur` possèdent toutes les deux une méthode `demarrer()` ; mais les actions de cette méthode sont différentes pour chacune des 2 classes (par exemple `demarrer()` pour `Vehicule` inclut `insererClef()`, etc. alors que `demarrer()` pour `Ordinateur` inclut `allumerEcran()`, etc.).

- polymorphisme d'héritage : 1 sous-classe hérite d'une méthode [5] de la super-classe (nom et signature de la méthode identiques), mais dont le comportement est différent (adapté à la sous-classe par redéfinition de la méthode) ; aussi appelé *spécialisation* [6] ;

[1] Uniquement si ceux-ci ont été déclarés publics ou protégés.
[2] Mot d'étymologie grecque signifiant « peut prendre plusieurs formes ».
[3] La signature d'une méthode correspond au nombre et au type d'arguments (paramètres d'entrée) de la méthode.
[4] Surcharge (fr) ≡ overloading (eng).
[5] Parmi un ensemble d'autres membres.
[6] Spécialisation (fr) ≡ overriding (eng).

Ex. : La classe `Velo` hérite de la méthode `demarrer()` de la classe `Vehicule` ; mais celle-ci est redéfinie pour être adaptée à la classe `Velo` (par exemple `demarrer()` pour la sous-classe `Velo` inclut `pedaler()`, etc., alors que `demarrer()` pour la super-classe `Vehicule` inclut `insererClef()`, etc.).

- polymorphisme paramétrique : 1 classe possède plusieurs définitions d'une méthode homonyme mais dont la signature est différente ; aussi appelé *généricité* [1].

  Ex. : La classe `Voiture` possède plusieurs définitions de la méthode `fairePlein(): void fairePlein(int nbLitres), void fairePlein(int nbLitres, String typeCarburant), void fairePlein(float pourXEuros).`

2.2 DÉVELOPPEMENT JAVA

2.2.1 Environnement de développement

Pour développer en Java, un éditeur de texte associé à un compilateur Java peut suffire. Mais on peut aussi utiliser des outils de développement, comme des AGL [2] (NetBeans, Eclipse, JBuilder, etc.) qui incluent les outils de base nécessaires (éditeur de texte et compilateur) et proposent un certain nombre de fonctionnalités qui simplifient la construction d'applications (débogueur, outils graphiques RAD, etc.).

Le code source Java (fichier *.java*) est compilé dans un premier temps ; le résultat de cette compilation est un fichier (*.class*) et constitue ce qu'on appelle alors le **bytecode** [3]. Ce bytecode est alors interprété par la machine cible, au moyen de la machine virtuelle Java, propre à la plate-forme en cours d'utilisation sur cette machine.

Dans son exploitation générale, Java est donc un langage compilé et interprété ; mais on peut aussi développer de manière classique pour une plate-forme spécifique, en effectuant une compilation native [4].

La création d'un fichier exploitable à partir d'un fichier source Java suit les étapes suivantes :

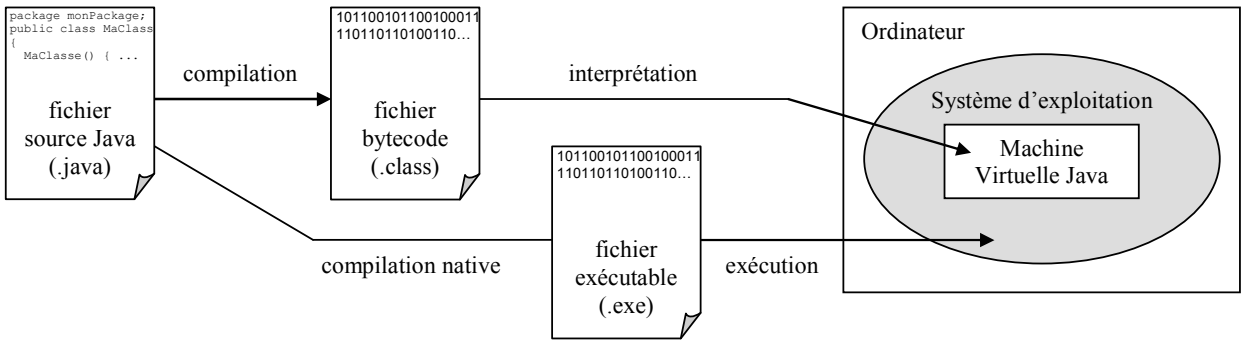


Figure 2.4 : exploitation du code source Java

2.2.2 Structure d'un fichier source

La syntaxe Java est dérivée de la syntaxe C++ ; néanmoins, on note les différences suivantes :

- Un fichier source Java porte l'extension *.java* ;
- Un fichier source ne contient qu'une seule classe [5] et il porte le nom de la classe (*NomClasse.java*) ; dans un projet Java, on a donc autant de fichiers que de classes ;
- Les méthodes sont implémentées à l'intérieur de la classe ;
- La relation de type « objet » est marquée par le caractère '`.`' (point), la notation pointée ('`->`') n'existe pas ;
- Toutes les instanciations d'objet se font implicitement de manière dynamique ; de fait, en tant que langage tout objet, il ne reste plus que les variables de type primitif [6] qui ne sont pas instanciés ;
- La destruction des objets est réalisée de manière automatique dès lors que ceux-ci ne sont plus référencés [7] ;

[1] Généricité (fr) ≡ template (eng).
[2] Atelier de Génie Logiciel.
[3] Appelé aussi *p-code*, pour pseudo-language.
[4] Cependant on perd alors l'une des grandes spécificités du langage Java par rapport à ces concurrents : la portabilité.
[5] Une seule classe publique, laquelle peut contenir des classes privées (utilisables uniquement par la classe publique).
[6] `int`, `long`, `float`, `double`, `char`, etc.
[7] C'est un mécanisme appelé *ramasse-miettes* (fr) ≡ garbage collector (eng), inclus dans la JVM, qui s'occupe de cette tâche.

- Pas de préprocesseur (`#include`, etc.), pas de macros, pas de variables globales, pas d'héritage multiple, pas de surcharge d'opérateurs [1].

La structure d'un fichier source Java est donc la suivante :

```
package nom_paquetage;  // nom du paquetage (projet) auquel appartient la classe

import chemin.bibliotheque;     // importation d'une bibliothèque/paquetage

public class NomClasse
{
   // attributs
   type_attribut attribut1;
   ...

   // méthodes
   NomClasse() { ... }     // constructeur
   type_méthode nomMethode1(arguments) { ... }
   ...

   // méthode main() – permet d'exécuter la classe (1 seule par projet en général)
   public static void main(String[] args) { ... }

} // pas de ';' final
```

Ex. :

UneClasse.java

```
package MonPackage;

public class UneClasse
{
   private int i;

   public UneClasse(int ii) // constructeur avec 1 argument de type int
   {
      this.i = ii; // initialisation de l'attribut i à l'aide de l'argument
   }

   public void uneMethode()
   {
      System.out.println(i);
   }
}
```

MonApplication.java

```
package MonPackage;

public class MonApplication
{
   private int x;
   private int tab[];
   private UneClasse unObjet;

   public MonApplication() {}

   public static void main(String[] args)
   {
      this.x = 5; // utilisation de this car x est attribut, pas variable
      tab = new int[4]; // instanciation du tableau (un tableau est un objet)

      unObjet = new UneClasse(x); // instanciation d'un objet de UneClasse
      unObjet.uneMethode(); // appel d'une méthode de UneClasse
      System.out.println(tab.length); // affichage de la longueur du tableau
   }
}
```

[1] À l'exception de la surcharge de l'opérateur + pour la classe String, qui permet de concaténer des chaînes.

2.2.3 Les bibliothèques de classes Java

Les classes sont regroupées en bibliothèques de classes, appelées **paquetages** [1] ; lesquelles permettent d'utiliser les diverses ressources de la machine (capacités graphiques, réseau, base de données, etc.). Elles représentent ce qu'on appelle l'API [2] de Java. Ainsi, lorsque le langage Java subit un changement de version, c'est principalement l'API qui est modifié, avec la mise à jour et/ou le rajout de classes.

Chaque paquetage se rapporte à un domaine précis : composants graphiques, composants réseaux, etc.

L'API de Java est hiérarchisé :

- `java.lang`, `java.util` : classes de base et utilitaires du langage (importées par défaut) ;
- `java.io` : entrées/sorties ;
- `java.net` : réseau ;
- `java.awt`, `java.swing` : graphique ;
- etc.

Pour utiliser une classe, ou bien tout ou partie d'un paquetage déjà existant, il faut importer la/les classe(s) grâce au mot-clef `import` suivant la syntaxe `import chemin.bibliotheque;`.

Ex. :
```
import java.awt.Button;    // importation de la classe Button
import java.awt.font.*;    // importation d'une partie du paquetage java.awt
import java.awt.*;         // importation du paquetage complet java.awt
```

Toute classe appartient à un paquetage. Si on n'en spécifie pas un explicitement, un paquetage par défaut est attribué, qui correspond au nom du projet auquel appartient la classe.

On peut aussi définir ses propres paquetages et les utiliser par la suite dans ses propres programmes.

2.3 LES VARIABLES

2.3.1 Les types de variables prédéfinis

Le langage Java utilise les types classiques suivants, dits aussi types « primitifs » car non-objets :

- `boolean` : booléen (`true`/`false`) ;
- `byte`, `short`, `int`, `long` : entier sur 8, 16, 32, 64 bits ;
- `float`, `double` : réel sur 32, 64 bits (pour `float`, on suffixe les littéraux avec '`f`', ex : `float x = 1.4f;`) ;
- `char` : caractère sur 16 bits au standard Unicode [3] (ex. : `char c = '\u0040';` ou `char c = 'A';`) ;
- `void` : type « vide », aucun type.

Mais il existe aussi des types prédéfinis objet :

- `String` : chaîne de caractères.

2.3.2 Les constantes

Java ne possède pas de macros [4]. Pour définir une constante, on utilise le modificateur `final` [5] suivant la syntaxe `final type_variable nomVariable`.

Ex. : `final int NB = 5;`

2.3.3 Les chaînes de caractères

Le type `String` représente une chaîne de caractères. Ce type est en réalité un objet et doit donc être instancié dynamiquement.

[1] Paquetage (fr) ≡ package (eng).
[2] Application Programming Interface : Interface de programmation d'application.
[3] Le standard Unicode, codé sur 16 bits, permet de coder 65536 caractères différents, et comprend ainsi une partie des alphabets cyrillique, hébreux, arabe, chinois, grec, etc., ainsi que les caractères accentués (non inclus dans le standard ASCII, codé sur 7 bits : 128 caractères).
[4] #define en langage C/C++.
[5] Le mot-clef final est un modificateur qui peut aussi être appliqué à une méthode, ou bien une classe, mais il a alors une autre signification.

Ex. : `String chn = new String();`

Les chaînes de caractères doivent être encadrées par le caractère '`"`' (guillemets).

Ex. : `String chn = new String("ceci est une chaîne de caractères");`

On peut concaténer les chaînes facilement avec l'opérateur '+' (plus).

Ex. :
```
String chn, chn2;
chn = new String("ceci est une ");
chn2 = new String(chn + "chaîne de caractères");
```

Les caractères spéciaux connus peuvent aussi être utilisés : `\r` (retour chariot), `\n` (saut de ligne), `\t` (tabulation), `\b` (retour arrière), `\f` (saut de page).

Le caractère `\` est le caractère d'échappement et permet par exemple d'insérer le caractère '`"`' (guillemet) dans une chaîne.

En tant qu'objet, le type `String` possède donc des attributs et des méthodes. On notera par exemple les méthodes `length()` et `equals()`.

2.3.4 Les tableaux

Les tableaux sont considérés comme des objets et doivent donc être instanciés dynamiquement.
Il existe deux types de déclarations possibles pour les tableaux :

- `type nom_tableau[]` : classique, comme en C++ ;
- `type[] nom_tableau` : le type `type[]` est considéré comme un objet.

Ex. :
```
int tab1[] = new int[10];
int[] tab2 = new int[10];
```

En tant qu'objet, un tableau possède des attributs et des méthodes. On notera par exemple l'attribut `length`, comme les méthodes `toString()` et `equals()`.

On peut évidemment créer des tableaux à plusieurs dimensions.

Ex. : `double[][] nombres = new double[10][20];`

Nb : Les objets `String` (objet de type `String`) et `String[]` (tableau d'objets de type `String`) sont deux objets complètements différents ; ils n'ont donc pas les mêmes attributs et les mêmes méthodes.

2.4 LES STRUCTURES DE CONTRÔLE

Elles sont identiques au langage C++.

2.4.1 Les opérateurs d'évaluation d'expression

Pour évaluer une expression, on utilise un ou plusieurs **opérateurs**, parmi 2 types différents :

- opérateurs de tests :
  - == : égal ;
  - != : différent ;
  - > : strictement supérieur ;
  - < : strictement inférieur ;
  - >= : supérieur ou égal ;
  - <= : inférieur ou égal.
- opérateurs logiques :
  - `&&` : ET ;
  - `||` : OU ;
  - `!` : NON.

Le résultat obtenu est un booléen (`true`/`false`).

La priorité des opérateurs est la suivante (du plus prioritaire au moins prioritaire) :
- parenthèses ;
- incrémentation, décrémentation ;
- multiplication, division, modulo ;
- addition, soustraction ;
- comparaison ;
- affectation.

2.4.2 Les tests

2.4.2.1 Test simple « si… sinon… »

Pour réaliser un test simple « si… sinon… », on utilise l'instruction `if ()` éventuellement associée avec l'instruction `else`, suivant la syntaxe :

```
if (expression) {
   /* instructions à exécuter si expression vraie */
}
else {
   /* instructions à exécuter si expression fausse */
}
```

2.4.2.2 Test multiple « au cas où… faire… »

Pour réaliser un test multiple « au cas où… faire… », on utilise les instructions `switch ()` et `case` éventuellement associées avec l'instruction `default`, suivant la syntaxe :

```
switch (variable) {
   case valeur1 :  /* instructions si variable vaut valeur1 */
                   break;
   case valeur2 :  /* instructions si variable vaut valeur2 */
                   break;
   ...
   default :  /* instructions si variable ne vaut aucune des valeurs */
}
```

Nb : Seules des variables de type *entier* ou *caractère* peuvent être utilisées.

2.4.3 L'opérateur ternaire

L'opérateur ternaire peut être utilisé, à la place d'une structure de test à base de `if`, pour affecter rapidement une valeur à une variable. La syntaxe est la suivante :

```
variable = (expression) ? valeur_si_expression_vraie : valeur_si_expression_fausse;
```

2.4.4 Les boucles

2.4.4.1 Boucle « tant que… faire… »

Pour réaliser une répétition « tant que… faire… » avec aucune exécution au minimum, on utilise l'instruction `while ()`, suivant la syntaxe :

```
while (expression) {
   /* instructions à répéter si expression vraie */
}
```

2.4.4.2 Boucle « répéter… tant que… »

Pour réaliser une répétition « répéter… tant que… » avec au moins 1 exécution, on utilise les instructions `do` et `while ()`, suivant la syntaxe :

```
do {
   /* instructions à répéter si expression vraie */
}
while (expression);
```

2.4.4.3 Boucle « pour… faire… »

Pour réaliser une répétition contrôlée « pour… faire… », on utilise l'instruction `for ()`, suivant la syntaxe :

```
for (initialisation ; expression ; modification) {
   /* instructions à répéter si expression vraie */
}
```

2.5 GESTION DES ÉVÉNEMENTS

Java intègre la notion de gestion d'événements. C'est-à-dire que l'on programme la réaction de l'application en fonction des actions de l'utilisateur sur tout objet graphique de l'application ; ces actions sont appelées **événements**.

En réalité, un événement est un message qu'un objet envoie à un autre objet en lui signalant l'occurrence d'un fait remarquable (ex. : un clic de souris sur un bouton).

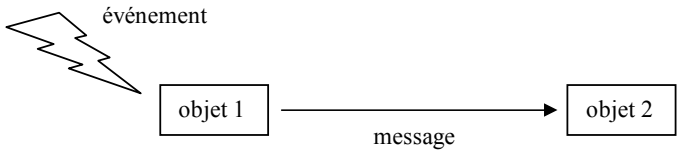


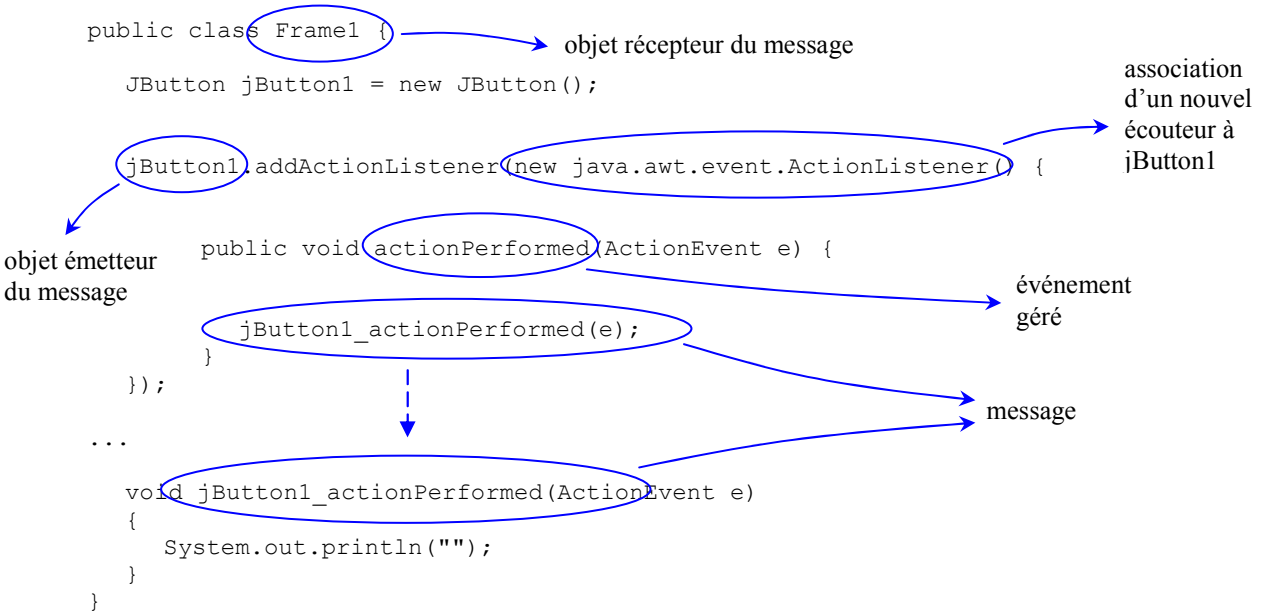
Figure 2.5 : envoi de message entre deux objets lors d'un événement

Dans un environnement graphique, il y a une grande quantité d'événements possibles, et un objet ne sera intéressé que par une partie d'entre eux. Par conséquent il faut spécifier les événements que chaque objet doit prendre en considération ; on utilise pour cela un objet qui est appelé un **écouteur** [1].

Un même objet peut être intéressé par plusieurs événements ; réciproquement, un même type d'événements peut intéresser plusieurs objets différents.

La gestion d'événements revient donc à définir une communication entre deux objets grâce à un écouteur ; l'un des objets est donc émetteur du message, alors que l'autre est récepteur.

Ex. : Soit un objet `JButton` auquel on désire associer l'événement `actionPerformed` (clic sur le bouton).



Il existe plusieurs types d'écouteurs ; chacun d'entre eux est spécialisé et est défini par une classe :

- `class ActionListener` : gestion des événements « standard » : clic sur un bouton, double-clic sur un élément de liste, choix d'un élément d'un menu, validation d'un champ texte ;
  Méthode : `actionPerformed()`.

[1] Écouteur (fr) ≡ listener (eng) ; il faut comprendre « écouteur d'événements », c'est-à-dire « être à l'écoute d'un événement ».

- `class MouseListener` : gestion des clics de souris ;
  Méthodes : `mouseClicked()`, `mousePressed()`, `mouseReleased()`, `mouseEntered()`, `mouseExited()`.
- `class MouseMotionListener` : gestion des déplacements de souris ;
  Méthodes : `mouseDragged()`, `mouseMoved()`.
- `class KeyListener` : gestion du clavier ;
  Méthodes : `keyTyped()`, `keyPressed()`, `keyReleased()`.
- `class WindowListener` : gestion de la fenêtre ;
  Méthodes : `windowOpened()`, `windowClosing()`, `windowClosed()`, `windowIconified()`, `windowDeiconified()`, `windowActivated()`, `windowDeactivated()` ;
- `class FocusListener` : gestion des événements lorsqu'un composant devient actif ou passif (gain ou perte de focus) ;
  Méthodes : `focusGained()`, `focusLost()`.
- `class TextListener` : gestion d'un composant de type texte ;
- …

2.6 LES EXCEPTIONS

Le langage Java offre un mécanisme pour palier la gestion des erreurs d'exécution d'un programme ; le but est de prévoir une poursuite ou une sortie du programme de manière correcte lorsqu'une erreur survient.

On appelle **exception** le signal envoyé par une méthode ayant détecté l'occurrence d'une erreur d'exécution ou d'une situation anormale.

On dit que la méthode déclenche (`throws`) une exception, qui est interceptée (`catch`) par le code. Les appels de méthodes susceptibles de déclencher une exception se font à l'intérieur d'un bloc spécifique (`try`).

C'est dans le bloc `catch` qu'est défini le traitement de l'exception.

Enfin, le bloc optionnel `finally` permet de définir des traitements qui sont exécutés qu'il y ait eu exception ou pas.

Ex. :
```
public class Frame1 extends Frame
{
   public Frame1()
   {
      try {
         jbInit();
      }
      catch(Exception e) {
         System.out.println("erreur");
      }
      finally {
         ...
      }
   }

   ...

   private void jbInit() throws Exception
   {
      JButton jButton1 = new JButton();
      JTextField jTextField1 = new JTextField();
      ...
   }
}
```

`jbInit()` peut générer une erreur d'exécution, il faut donc en tenir compte

code à exécuter si le bloc `try` associé (`jbInit()` donc) déclenche une exception

code à exécuter une fois le bloc `try` traité, qu'il y ait eu exception ou pas

`jbInit()` (initialisation de l'IHM) est susceptible de déclencher une exception

Le bloc `catch ()` doit être mentionné même si aucun traitement particulier de l'exception n'est défini (`catch (Exception e) {}`).

Nb : Lors du traitement d'une exception dans une méthode, on peut faire le choix suivant : soit traiter l'exception là où cela est nécessaire, en utilisant pour cela `try` / `catch ()` ; soit ne pas traiter l'exception directement, et laisser alors la méthode appelante s'en occuper, en le lui faisant savoir en utilisant la déclaration `throws`.

Ex. : Dans l'exemple précédent, il a été décidé que `jbInit()` ne traiterait pas l'exception, et cette méthode a donc été déclaré ainsi : `void jbInit() throws Exception {...}`. Ainsi, c'est à la méthode appelant `jbInit()`, en l'occurrence `Frame1()`, de gérer l'exception, en utilisant `try` / `catch ()`.

Dans le cas d'un bloc d'instructions ou bien d'une méthode susceptible de générer plus d'un seul type d'exceptions différentes, la gestion des exceptions peut se faire de 2 manières différentes :

- On traite chaque exception indépendamment : on aura alors 1 bloc `try {}`, et autant de blocs `catch () {}` spécialisés que d'exceptions différentes à gérer.

  Ex. :
  ```
  try {
     // bloc d'instructions ou méthode susceptible de générer plusieurs
     // exceptions différentes (exemple avec NullPointerException et IOException)
  }
  catch (NullPointerException npe) {
     // instructions à exécuter si une exception du type
     // NullPointerException est déclenchée
  }
  catch (IOException ioe) {
     // instructions à exécuter si une exception du type
     // IOException est déclenchée
  }
  ```

- On traite toutes les exceptions de la même manière : on aura alors 1 bloc `try {}`, et 1 seul bloc `catch() {}` gérant tout type d'exception possible (tout type d'exception hérite de la classe `Exception` (ex. : signature de la classe `IOException` : `public class IOException extends Exception {...}`)).

  Ex. :
  ```
  try {
     // bloc d'instructions ou méthode susceptible de générer plusieurs
     // exceptions différentes
  }
  catch (Exception e) {
     // instructions à exécuter si n'importe quel type d'exception
     // (NullPointerException, IOException, etc.) est déclenchée
  }
  ```

2.7 LES DIFFÉRENTS TYPES D'APPLICATIONS JAVA

2.7.1 Les applications « console »

On appelle **application console** [1] une application qui ne met en jeu qu'une fenêtre de type texte et qui pourra nous permettre d'afficher différents résultats.
La fenêtre texte est ce qu'on appelle la *console*, qui sera donc ici une console Java.

La structure générale du code Java pour une application console est donc la suivante :

UneClasse.java

```
package MonPackage;

public class UneClasse
{
   // attributs
   ...

   // méthodes
   public UneClasse() // constructeur par défaut (sans argument)
   {
      ...
   }

   public void uneMethode()
   {
      System.out.println("méthode uneMethode() de la classe UneClasse");
   }
   ...
}
```

[1] Dite aussi CUI : Console User Interface (eng) ≡ Interface Utilisateur Console (fr).

MonApplication.java

```
package MonPackage;

public class MonApplication // classe principale
{
   // attributs
   ...

   // méthodes
   public MonApplication() // constructeur par défaut (sans argument)
   {
      ...
   }

   public static void main(String[] args)
   {
      MonApplication monApp; // déclaration d'un objet de la classe elle-même
      monApp = new MonApplication(); // instanciation d'un objet de la classe

      System.out.println("bonjour\n"); // affichage console

      UneClasse unObjet;
      unObjet = new UneClasse(); // instanciation d'un objet de UneClasse
      unObjet.uneMethode(); // appel d'une méthode de UneClasse
      ...
   }
}
```

La machine virtuelle Java exécute la méthode principale `main()`. Celle-ci doit absolument être publique (`public`), pour pouvoir être appelée (exécutée) de l'extérieur de la classe ; elle doit aussi absolument être statique (`static`) pour pouvoir être appelée sans que l'on dispose d'une instance de la classe [1] ; enfin, la signature doit absolument faire apparaître un tableau de chaînes en tant qu'arguments d'entrée (`String[] arguments`). La déclaration complète de la méthode `main()` est donc : `public static void main(String[] args)`.

La méthode `main()` exécute ensuite un code qui doit alors [2] instancier un objet de la classe, afin que les méthodes non statiques de la classe principale puissent être invoquées.

Les affichages console se font en utilisant la classe `System`, qui possède des méthodes statiques ; celles-ci peuvent donc être appelées sans disposer d'une instance de cette classe. Cette classe permet notamment de gérer les flux d'entrées/sorties standard :

- entrée standard : clavier (`System.in.*`) ;
- sortie standard : écran (`System.out.*`) ;
- erreur standard : écran (`System.err.*`).

2.7.2 Les applications graphiques

Une **application graphique** [3] est une application console à laquelle on a rajouté une interface graphique [4]. L'application possède alors 2 fenêtres :

- fenêtre graphique (interface graphique) : application graphique ;
- fenêtre texte (console java) : permet d'afficher les messages d'exceptions interceptées.

Par conséquent, toute application java (graphique ou texte) comprend toujours une console, qui n'est pas affichée automatiquement dans le cas d'une application graphique, mais qui est disponible si besoin est.

La partie « application graphique » dérive [5] de la classe `Frame` (ou `JFrame`).

La structure générale du code Java pour une application graphique est donc la suivante :

[1] Comme en Java les méthodes, et donc la méthode `main()`, sont déclarées à l'intérieur de la classe, il est impossible d'avoir pu créer un objet de la classe elle-même au moment de la définition de la méthode `main()` ; celle-ci est donc déclarée statique, et peut alors être appelée sans que l'on ait instancié un objet de la classe.
[2] Entre autres.
[3] Dite aussi GUI : Graphical User Interface (eng) ≡ Interface Utilisateur Graphique (fr).
[4] Dite aussi IHM : Interface Homme-Machine.
[5] Au sens objet.

Cadre1.java

```
package MonPackage;

import java.awt.*; // importation de librairies graphique
import javax.swing.*; // importation de librairies graphique

public class Cadre1 extends Frame
{
   // attributs
   ...

   // méthodes
   public Cadre1()
   {
      ...
   }

   public void uneMethode()
   {
      ...
   }
   ...
}
```

MonApplication.java

```
package MonPackage;

public class MonApplication // classe principale
{
   // attributs
   ...

   // méthodes
   public MonApplication()
   {
      ...
   }

   public static void main(String[] args)
   {
      MonApplication monApp = new MonApplication();

      Cadre1 monCadre = new Cadre1();
      ...
   }
}
```

En général on tend à respecter la séparation traitements / graphique [1], à l'image de la séparation fenêtre texte / fenêtre graphique, et on utilise alors une classe (voire plusieurs), autre que l'interface graphique, pour réaliser les traitements.

2.7.3 Les applets

On appelle **applet** une application graphique qui s'exécute à l'intérieur de la fenêtre d'un navigateur internet.

Pour créer une applet, il faut dériver de la classe `Applet`.

La structure générale du code Java pour une applet est donc la suivante :

[1] Modèle MVC : Model – View – Controller (eng) ≡ Modèle – Vue – Contrôleur (fr), séparation des traitements et de l'IHM.

Applet1.java

```
package MonPackage;

import java.applet.*; // importation du paquetage applet

public class Applet1 extends Applet
{
   // attributs
   ...

   // méthodes
   public void init()
   {
      ...
   }
   ...
}
```

Lorsque le navigateur charge une page HTML contenant une applet, il charge donc l'applet elle-même, puis instancie un objet de la classe principale (dérivée de la classe `Applet`). C'est la fenêtre du navigateur qui sert de conteneur graphique à l'applet.

Si on veut que l'applet puisse aussi être exécutée en dehors d'un navigateur web, comme une simple application graphique, il suffit d'écrire la méthode `main()` qui sera appelée lors d'une exécution classique.

3 L'HÉRITAGE

3.1 L'HÉRITAGE SIMPLE

3.1.1 Principes

En tant que langage objet, Java implémente le concept d'héritage de classes. En revanche, contrairement au C++, Java ne permet pas d'héritage multiple, et on ne peut donc dériver que d'une seule classe [1].

Pour définir une sous-classe à partir d'une super-classe, on utilise le mot-clef `extends`, suivant la syntaxe `public class SousClasse extends SuperClasse {...}`

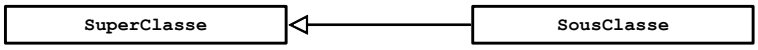


Figure 3.1 : héritage entre deux classes

La sous-classe hérite ainsi des attributs et des méthodes de la super-classe ; les méthodes peuvent bien évidemment être redéfinies par surcharge le cas échéant.

3.1.2 Hiérarchie de classes

En Java, toutes les classes dérivent implicitement de la classe `Object`, qui est appelée **classe-mère**. Cet héritage peut être direct, ou bien peut provenir de plusieurs relations d'héritages successives.

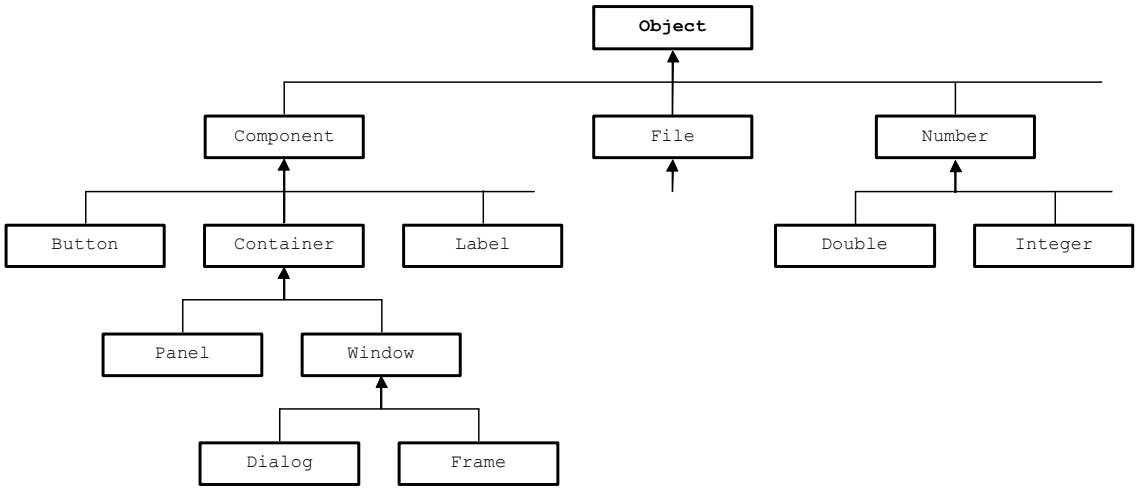


Figure 3.2 : hiérarchie des classes Java

L'ensemble des classes et des relations d'héritage entre ces classes forment alors une arborescence (hiérarchie) au sommet de laquelle se situe la classe `Object`.
Cette hiérarchie unique permet de définir des comportements communs pour toute classe Java.

Ainsi, une classe qui n'hérite apparemment d'aucune autre, hérite en fait implicitement de la classe `Object`.

[1] Ceci pourrait apparaître comme une faiblesse de Java par rapport au C++. En réalité cette particularité a le mérite d'éviter les difficultés de programmation liées à l'héritage multiple ; la notion d'interface (cf. 3.2) permet de palier en partie ceci.

```
public class UneClasse extends Object        ⇔      public class UneClasse
{                                                   {
   ...                                                 ...
}                                                   }
```

Une classe quelconque créée par l'utilisateur hérite donc toujours, même indirectement, de la classe `Object`.

La conséquence est que les méthodes [1] de la classe `Object` se retrouvent alors dans toute classe Java. Par exemple, la méthode `toString()` est dérivée dans toutes les classes Java ; cette méthode affiche le nom de la classe sous forme de chaîne, sauf bien évidemment pour les classes qui ont redéfinies cette méthode [2].

Comme le constructeur d'une classe fait un appel implicite au constructeur de la classe dont il dérive, une autre conséquence est que l'on fera donc toujours appel, au final, au constructeur de la classe `Object` en enchaînant les différents constructeurs des relations d'héritage existantes entre la classe utilisateur et la classe `Object`.

3.1.3 Accès aux attributs et méthodes dans une relation d'héritage

Pour être accessible par une sous-classe, les attributs et les méthodes d'une super-classe doivent être déclarées protégés (`protected` : seules les sous-classes peuvent y accéder), voire publics (`public` : tout le monde y a accès).

On peut faire appel à un attribut ou bien une méthode de la super-classe en utilisant l'opérateur `super` qui fonctionne à l'identique de l'opérateur `this`.

Pour accéder à un attribut de la super-classe, on écrit donc `super.nomAttribut` et pour une méthode `super.nomMethode()`.

Ex. :

SuperClasse.java

```
package MonPackage;

public class SuperClasse
{
   protected int i;

   public SuperClasse()
   {
      ...
   }
}
```

SousClasse.java

```
package MonPackage;

public class SousClasse extends SuperClasse // SousClasse dérive de SuperClasse
{
   private int i;

   public SousClasse()
   {
      ...
   }

   public uneMethode
   {
      i = 0;

      this.i = 1;

      super.i = 5;
   }
}
```

[1] La classe `Object` ne possède pas d'attribut.

[2] Cas des classes `Double`, `Integer`, etc. par exemple.

La première action réalisée par le constructeur d'une sous-classe est d'exécuter le constructeur par défaut de la super-classe dont il dépend, et ce de manière automatique, sans que cela soit visible dans le code ; en revanche, si un appel explicite à l'une des surcharges du constructeur de la super-classe est fait, alors il n'y a plus besoin d'invoquer le « super »-constructeur par défaut.

Lorsque l'on désire faire appel au constructeur de la super-classe, on écrit `super()`. Cette méthode publique est surchargée autant de fois que le constructeur de la super-classe est lui-même surchargé, en respectant alors les différentes signatures existantes.

Ex. :

SuperClasse.java

```
package MonPackage;

public class SuperClasse
{
   public SuperClasse()
   {
      ...
   }

   public SuperClasse(int a)
   {
      ...
   }

   public SuperClasse(double db, int b)
   {
      ...
   }

}
```

SousClasse.java

```
package MonPackage;

public class SousClasse extends SuperClasse
{
   public SousClasse()
   {
      super(); // appel implicite, donc inutile
   }

   public SousClasse(int aa)
   {
      super(aa); // appel du constructeur surchargé sinon appel implicite à super()
   }

   public SousClasse(double ddb, int bb)
   {
      super(ddb, bb); // appel du constructeur surchargé (sinon appel à super())
   }
}
```

Nb : On peut empêcher la surcharge future d'une méthode en utilisant le mot-clef `final`.

3.2 LES INTERFACES

3.2.1 Introduction : méthode et classe abstraite

On dit qu'une méthode d'une classe est **abstraite** si seul son prototype est défini, et qu'aucun code n'est donné. Son implémentation complète devra en fait être écrite par chaque sous-classe qui dérivera de cette classe.
Une méthode abstraite d'une classe doit donc obligatoirement être surchargée dans chaque sous-classe dérivée.

La « super-méthode » ne constitue alors que la définition d'un concept, que chaque sous-classe sera libre d'implémenter selon ses besoins.

Une méthode abstraite se déclare en utilisant le mot-clef `abstract`.

Ex. :

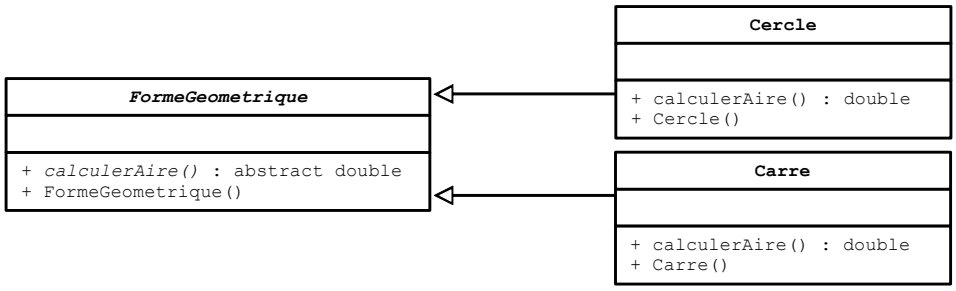


Figure 3.3 : exemple d'utilisation d'une méthode abstraite

FormeGeometrique.java

```
package Geometrie;

public abstract class FormeGeometrique
{
   public FormeGeometrique()
   {
      ...
   }

   public abstract double calculerAire(); // définition du prototype uniquement
}
```

Cercle.java

```
package Geometrie;

public class Cercle extends FormeGeometrique
{
   private double rayon;

   public Cercle(double r)
   {
      this.rayon = r;
   }

   public double calculerAire() // implémentation de la méthode abstraite
   {
      return (java.lang.Math.PI * this.rayon * this.rayon);
   }
}
```

Carre.java

```
package Geometrie;

public class Carre extends FormeGeometrique
{
   private double cote;

   public Carre(double c)
   {
      this.cote = c;
   }
```

```
        public double calculerAire() // implémentation de la méthode abstraite
        {
           return (this.cote * this.cote);
        }
     }
```

Si une classe comprend une méthode abstraite, alors la classe elle-même doit être déclarée abstraite (`public abstract class MaClasse { ... }`).

La conséquence directe du caractère « abstrait » de la classe est qu'on ne peut pas instancier d'objets de cette classe [1]. La classe est donc pleinement destinée à demeurer une classe « concept », c'est-à-dire une classe qui constitue un modèle pour d'autres classes.

Lors de la définition d'une sous-classe à partir d'une super-classe abstraite, toute méthode abstraite doit obligatoirement être codée ; sinon elle doit être déclarée abstraite, et de fait, la classe à laquelle elle appartient doit l'être aussi. La sous-classe constitue alors elle aussi un modèle pour d'autres classes [2].

3.2.2 Définition d'une interface

En Java, on appelle **interface** une classe abstraite dont toutes les méthodes sont abstraites (`abstract`) et tous les attributs sont constants (`final`).

On déclare une interface avec le mot-clef `interface`, qui vient en remplacement du mot-clef `class`.

```
public interface MonInterface {
   // attributs
   int i = 0; // initialisation obligatoire des attributs (doit être constant)
   ...

   // méthodes
   void uneMethode(); // définition du prototype des méthodes
   ...
}
```

De par sa nature, une interface a implicitement et automatiquement des attributs publics (`public`), statiques (`static`) et constants (`final`), ainsi que des méthodes publiques (`public`) et abstraites (`abstract`) ; il est donc inutile de préciser ces différents modificateurs.
De la même manière, une interface ne définit pas de constructeur.

En général, on assimile une interface à une classe de services, c'est-à-dire une classe qui est utilisée par d'autres classes pour implémenter une fonctionnalité qu'elles doivent proposer ; ces fonctionnalités sont appelées services [3].

Ex. : Une classe qui « utilise » l'interface `Runnable` doit implémenter la méthode `run()` qui autorise la fonctionnalité *exécution en tant que thread* (multi-tâches).

En d'autres termes, une interface ne fait que représenter, sans l'implémenter, un comportement que doit suivre certaines classes, laissant ainsi la liberté à chaque classe de l'implémenter selon ses besoins.

3.2.3 Implémentation

Une fois le type de service défini pour un groupe de classes à travers une interface, on utilise cette définition de service en dérivant de l'interface.

Plus spécifiquement, on dit qu'une classe **implémente** une interface si cette classe fournit les définitions de toutes les méthodes déclarées dans l'interface.
Pour marquer ce principe d'implémentation réalisé par la sous-classe, on utilise le mot-clef `implements` suivant la syntaxe `public class MaClasse implements MonInterface {...}`.

[1] Ce qui est logique étant donné qu'elle possède au moins une méthode dont le code n'est pas donné (la méthode abstraite).
[2] On peut ainsi imaginer une hiérarchie de classes abstraites, servant de modèles lors d'une implémentation.
[3] Usuellement, le nom d'une interface possède un suffixe en –ible ou –able, marquant ainsi la capacité de service offerte (ex. Runnable).

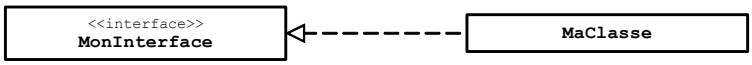


Figure 3.4 : implémentation d'une interface

MonInterface.java

```
package MonPackage;

public interface MonInterface
{
   ...
}
```

MaClasse.java

```
package MonPackage;

public class MaClasse implements MonInterface
{
   ...
}
```

Contrairement à l'héritage, une classe peut implémenter plusieurs interfaces [1].

Ex. :

```
public class MaClasse implements MonInterface, MonAutreInterface
{
   ...
}
```

Une interface peut hériter d'une autre interface. Cependant la limitation à l'héritage simple est identique à celles des classes.

Ex. :

MonInterface.java

```
package MonPackage;

public interface MonInterface
{
   ...
}
```

MonAutreInterface.java

```
package MonPackage;

public interface MonAutreInterface extends MonInterface // héritage d'interface
{
   ...
}
```

La hiérarchie des relations d'héritage et celle des implémentations d'interfaces sont totalement indépendantes. Ainsi, une classe peut héritée d'une autre classe et elle peut aussi, en même temps, implémenter une ou plusieurs interfaces.

Ex. :

```
public class MaClasse extends SuperClasse implements MonInterface
{
   ...
}
```

[1] De fait, le concept d'interface est souvent vu comme la possibilité de réaliser un héritage multiple en Java.

3.2.4 Exemple d'utilisation d'une interface

Soit la définition de l'interface `FormeGeometrique` suivante, et des classes `Cercle` et `Carre` qui implémentent cette interface.

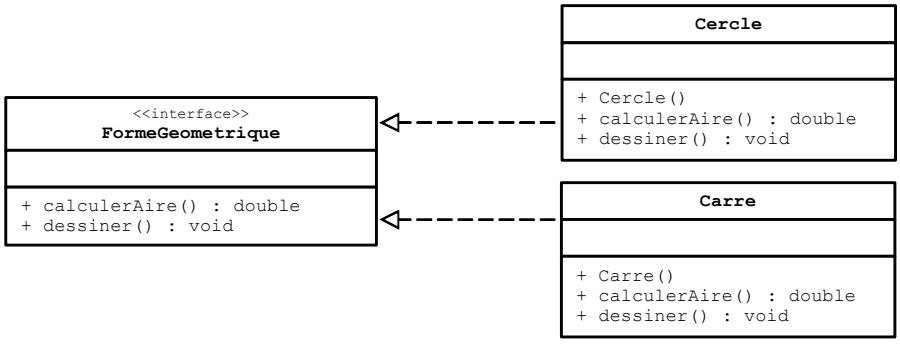


Figure 3.5 : exemple d'utilisation d'une interface

FormeGeometrique.java

```
package Geometrie;

public interface FormeGeometrique
{
   public void dessiner();
   public double calculerAire();
}
```

Cercle.java

```
package Geometrie;

public class Cercle implements FormeGeometrique // implémente l'interface
{
   private int rayon;

   public Cercle(int r)
   {
      this.rayon = r;
   }

   public void dessiner() // implémentation adaptée au cercle
   {
      g.drawOval(0, 0, 2 * this.rayon,2 * this.rayon);
   }

   public double calculerAire() // implémentation adaptée au cercle
   {
      return (java.lang.Math.PI * this.rayon * this.rayon);
   }
}
```

Carre.java

```
package Geometrie;

public class Carre implements FormeGeometrique
{
   private int cote;

   public Carre (int c)
   {
      this.cote = c;
   }
```

```
   public void dessiner() // implémentation adaptée au carré
   {
      g.drawLine(0, 0, 0, cote);
      g.drawLine(0, cote, cote, cote);
      g.drawLine(cote, cote, cote, 0);
      g.drawLine(cote, 0, 0, 0);
   }

   public double calculerAire() // implémentation adaptée au carré
   {
      return (this.cote * this.cote);
   }
}
```

On peut alors définir une liste de formes géométriques différentes auxquelles on peut appliquer le même traitement, puisque toutes les formes implémentent l'interface `FormeGeometrique`.

Ex. : Une classe `Dessin`, qui permet de gérer un ensemble de formes géométriques, de les dessiner, ainsi que de calculer l'aire de chacune des formes.

Dessin.java

```
package Geometrie;

public class Dessin
{
   private FormeGeometrique[] listeFormes; // déclaration d'un tableau de formes

   public Dessin()
   {
      listeFormes = new FormeGeometrique[] { // initialisation du tableau
         new Cercle(100), // Cercle est une sorte de FormeGeometrique
         new Carre(300) // Carre est une sorte de FormeGeometrique
      };
   }

   public void toutDessiner() // dessine toutes les formes
   {
      for(int i=0 ; i<listeFormes.length ; i++)
         listeFormes[i].dessiner(); // appel de dessiner() adaptée
   }

   public double[] toutCalculerAire() // calcule l'aire de toutes les formes
   {
      double[] aires = new double[listeFormes.length];

      for(int i=0 ; i<listeFormes.length ; i++)
         aire[i] = listeFormes[i].calculerAire();

      return (aires);
   }
}
```

`listeFormes[0]`, bien qu'étant un objet déclaré comme étant du type `FormeGeometrique` (`private FormeGeometrique[] listeFormes;`), a été instancié à l'aide du constructeur `Cercle()` (`new Cercle(100)`), par conséquent, il s'agit bien là d'une instance de la classe `Cercle` [1]. Donc lorsque la méthode `dessiner()` est appelée pour cet objet (`listeFormes[i].dessiner();` pour `i` = 0), on appelle bien la méthode `dessiner()` de la classe `Cercle()`, donc celle adaptée à ce que représente l'objet.

C'est la même chose pour `i` = 1, où on fait référence alors à un objet déclaré du type `FormeGeometrique` mais instancié avec le constructeur `Carre()` ; bien évidemment mêmes remarques pour la méthode `calculerAire()`.

Nb : Par souci de clarté, le code des différentes classes donné ci-dessus a été simplifié ; par conséquent il est inexact, et ne peut pas aboutir à une exécution correcte.

[1] On met ainsi en avant le fait qu'il est possible en POO, qu'un objet soit d'un type X à la déclaration, et d'un type Y à l'exécution ; il suffit pour cela que Y soit une sous-classe de X ; ex. : en Java, pour toute classe il est possible d'écrire : Object monObjet = new MaClasse();.

4 LES FLUX D'ENTRÉES/SORTIES

4.1 INTRODUCTION

4.1.1 Notion de flux

Une application peut avoir à faire appel à des entités informatiques externes [1] afin de lire ou d'écrire des informations ou des données.
La réalisation d'une action de lecture ou d'écriture constitue ce qu'on appelle un **flux**.

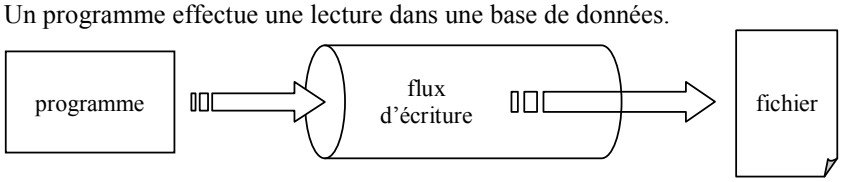
Ex. :
- lorsqu'une application sauvegarde des données dans un fichier, elle utilise un flux d'écriture ;
- lorsqu'une application lit des informations à partir d'une base de données, elle utilise un flux de lecture.

4.1.2 Définitions

Un **flux** a pour but principal de transférer des données ; il s'agit d'un outil de communication.
Il constitue un canal de transfert de données unidirectionnel et séquentiel entre deux entités informatiques. Dans une communication utilisant un flux, l'une est donc l'émetteur des données, et l'autre le récepteur.

Lorsqu'une communication de ce type est mise en œuvre par une application Java, l'une de ces entités est donc un programme Java, alors que l'autre pourra être un fichier, une base de données, un socket, un périphérique, …

Un programme effectue une lecture dans une base de données.



Un programme effectue une écriture dans un fichier.

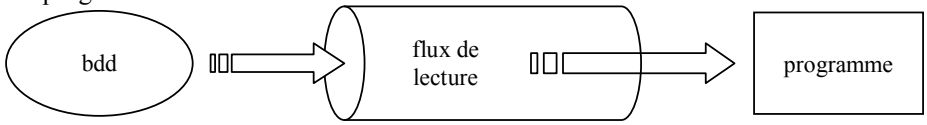


Figure 4.1 : exemples de flux de lecture et d'écriture

Un flux peut transférer les données de manière brute, soit donc octet par octet, ou bien de manière formatée, en transférant par exemples des `double`, des `int`, …

On considère que le programme Java est le point central des communications, et on peut alors parler de *flux d'entrée* pour un flux en lecture et de *flux de sortie* pour un flux en écriture.

4.1.3 Principe de mise en œuvre des flux

La retranscription algorithmique de la mise en œuvre des flux est la suivante :

[1] Externes à l'application elle-même, donc pouvant être localisées sur la même machine, ou d'autres dans un réseau informatique.

- flux d'écriture :
```
Ouvrir le flux
Tant qu'il ya des données
   Écrire les données
Fermer le flux
```
- flux de lecture :
```
Ouvrir le flux
Tant qu'il ya des données
   Lire les données
Fermer le flux
```

4.2 LES FLUX STANDARD

L'interfaçage entre l'application Java et le système d'exploitation, qui est une entité externe à l'application, met en œuvre des flux d'entrées/sorties.

Comme ces flux servent de manière récurrente lors de l'utilisation de la machine, on les appelle **flux standard** [1] ou flux d'entrées/sorties standard. Ces flux sont gérés par la classe `java.lang.System` :

- entrée standard : par défaut le clavier, utilise un flux géré par l'objet `System.in` (stdin) ;
- sortie standard : par défaut la console (l'écran), utilise un flux géré par l'objet `System.out` (stdout) ;
- erreur standard : par défaut est aussi la console, utilise un flux géré par l'objet `System.err` (stderr).

L'objet `System.in` gère donc un flux d'entrée ; il s'agit en fait d'une instance de la classe abstraite `InputStream`.

Les objets `System.out` et `System.err` gèrent eux des flux de sortie ; il s'agit en fait d'instances d'une sous-classe de la classe abstraite `OutputStream`.

4.3 GESTION DES FLUX

En Java, on peut distinguer 4 types de flux :

- flux d'octets : pour lire ou écrire octet par octet ;
- flux de données : pour lire ou écrire des données de type primitif (entier, flottant, booléen, chaîne, etc.) ;
- flux de caractères : pour lire ou écrire des caractères, similaire au flux d'octets en utilisant des caractères Unicode (codage sur 2 octets (16 bits)) ;
- flux d'objets : pour lire ou écrire des objets.

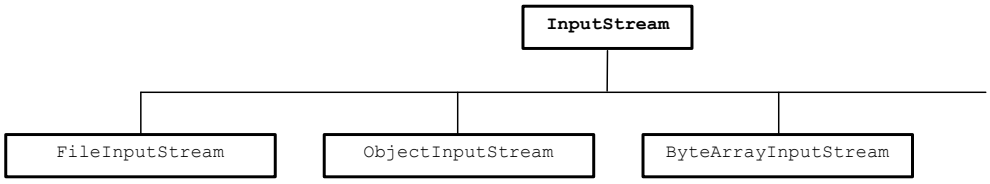
Les classes permettant de gérer ces différents flux d'entrées/sorties sont regroupés dans le paquetage `java.io` [2].

4.3.1 Les flux d'octets

Pour gérer des flux d'octets, on dispose d'un ensemble de classes dérivant des 2 classes abstraites suivantes :

- classe `InputStream` : définition des principales méthodes de gestion de flux de lecture d'octets ;
- classe `OutputStream` : définition des principales méthodes de gestion de flux d'écriture d'octets.

Ces 2 classes étant abstraites, elles ne peuvent donc pas être instanciées. Elles permettent de préciser des comportements généraux pour les classes dérivées.



[1] Les flux standards sont énormément utilisés dans le monde Unix, au niveau du shell.

[2] À l'exception des classes de gestion de flux sur sockets, qui font partie du paquetage java.net.

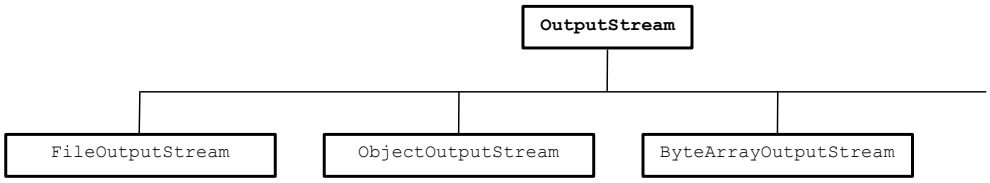


Figure 4.2 : classes de gestion de flux d'octets

Ces classes manipulent des octets, c'est-à-dire des données non formatées ; elles ne sont donc pas forcément des plus pratiques à utiliser dans tous les cas.

4.3.2 Les flux de caractères

Reposant sur les mêmes principes que les classes `InputStream` et `OutputStream`, on dispose des classes `Reader` et `Writer` qui réalisent des flux de caractères.

Ce sont aussi des classes abstraites qui servent donc de modèles pour l'ensemble des classes de gestion de flux de caractères.

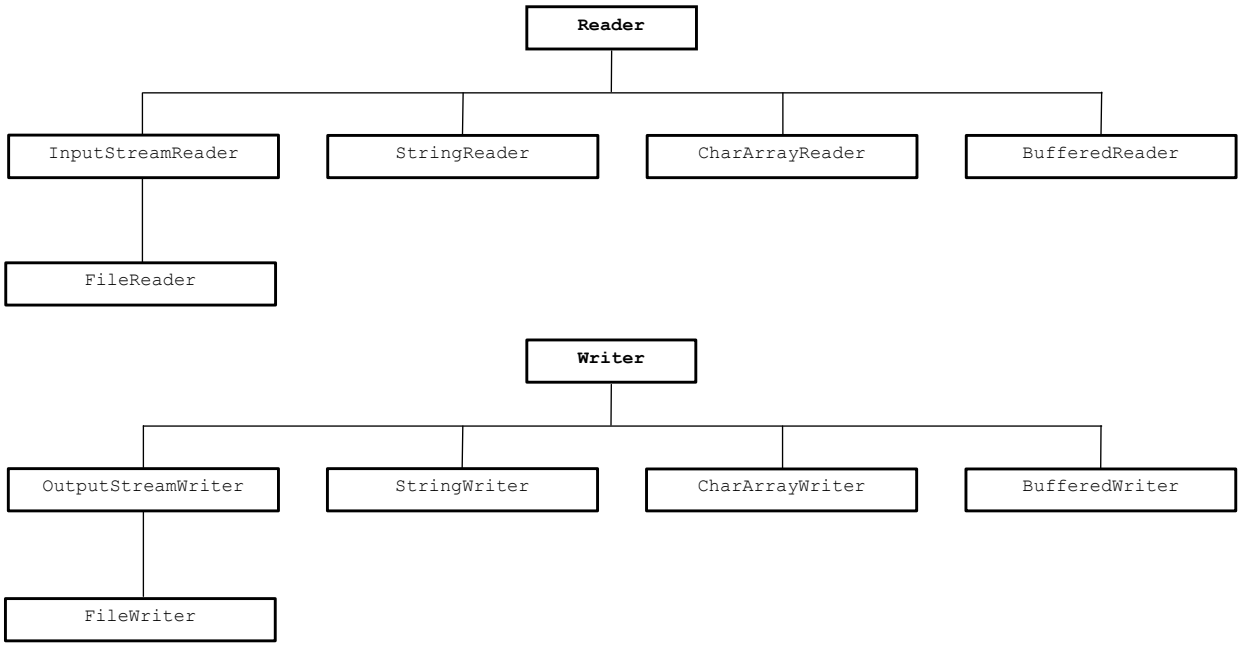


Figure 4.3 : classes de gestion de flux de caractères

4.4 GESTION DES FLUX SUR FICHIER TEXTE

4.4.1 Enregistrement des données

Pour enregistrer des données, il faut d'abord déterminer si l'on doit manipuler des octets ou des caractères ; ce qui revient à choisir entre les classes `OutputStream` et `Writer`.

On utilise ensuite la classe héritée correspondante qui permet de gérer des flux sur fichier : `FileOutputStream` ou `FileWriter`.

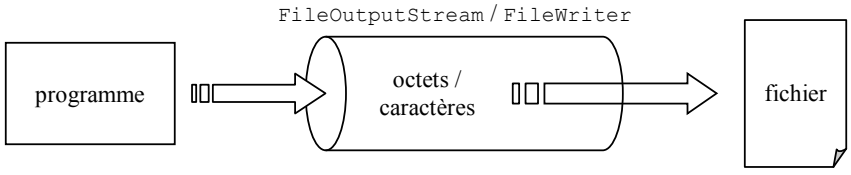


Figure 4.4 : flux de sortie sur fichier

On crée alors le flux de sortie en instanciant un objet de la classe et en précisant en paramètre le nom du fichier destination du flux.

Ex. : `FileWriter outFile = new FileWriter("fichier.txt");`

On peut ensuite écrire des données dans le flux grâce à la méthode `write()` de la classe utilisée :

- `FileOutputStream` : `write(byte[])` ;
- `FileWriter` : `write(String)`.

Ex. : `outFile.write("ceci est un test");`

Une fois toutes les données désirées écrites dans le flux (/enregistrées dans le fichier), on referme le flux en utilisant la méthode `close()`, présente dans chacune des classes `OutputStream` et `FileWriter`.

Ex. : `outFile.close();`

Lors du travail avec des fichiers, de nombreuses erreurs peuvent survenir (fichier protégé en écriture, plus d'espace disque, pas de droit d'écriture dans le répertoire, …) ; il est donc nécessaire d'intercepter toute erreur éventuelle d'entrées/sorties (`IOException`).

Ex. :

```
try {
   FileWriter outFile = new FileWriter("fichier.txt"); // ouverture du flux
   outFile.write("ceci est un test"); // écriture dans le flux
   outFile.close(); // fermeture du flux
}
catch (IOException e) { // traitement de l'exception du type IO
   System.out.println("erreur écriture fichier");
}
```

4.4.2 Lecture des données

Pour lire des données, il faut d'abord savoir si l'on doit manipuler des octets (`InputStream`) ou des caractères (`Reader`). On utilise ensuite la classe héritée correspondante qui permet de gérer des flux sur fichier : `FileInputStream` ou `FileReader`.

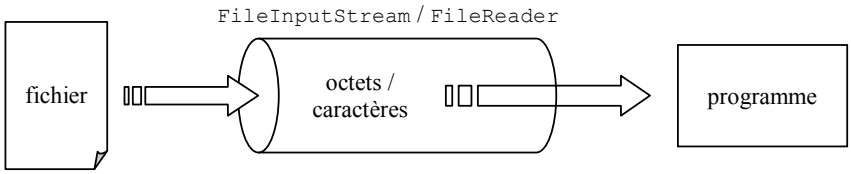


Figure 4.5 : flux d'entrée sur fichier

On crée alors le flux d'entrée en instanciant un objet de la classe et en précisant en paramètre le nom du fichier source du flux.

Ex. : `FileReader inFile = new FileReader("fichier.txt");`

On peut ensuite lire les données à partir du flux grâce à la méthode `read()` de la classe utilisée :

- `FileInputStream` : `read(byte[])` ;
- `FileReader` : `read(char[], int, int)`.

Ex. :
```
char[] texte = new char[100];
int debut = 200;
inFile.read(texte, debut, texte.length);
```

Une fois toutes les données lues à partir du flux, on referme le flux en utilisant la méthode `close()`, présente dans chacune des classes `InputStream` et `FileReader`.

Ex. : `inFile.close();`

Là encore, de nombreuses erreurs peuvent survenir (fichier illisible, protégé en lecture, …) ; il est donc nécessaire d'intercepter toute erreur éventuelle d'entrées/sorties (`IOException`).

Ex. :

```
try {
   char[] texte = new char[100]; // tableau pour stocker la lecture
   int debut = 200; // indice de début de lecture

   FileReader inFile = new FileReader("fichier.txt"); // ouverture du flux
   inFile.read(texte, debut, texte.length); // lecture de 100 caractères
   inFile.close(); // fermeture du flux
}
catch (IOException e) { // traitement de l'exception du type IO
   System.out.println("erreur lecture fichier");
}
```

4.4.3 Cas pratique

Les classes `FileOutputStream`, `FileWriter`, `FileInputStream` et `FileReader` manipulent les données individuellement ; ce qui a pour effet d'effectuer d'incessantes écritures ou lectures sur le fichier. Il est donc plus judicieux de regrouper les données puis d'effectuer une écriture dans le fichier en une seule fois, ou bien d'effectuer une lecture complète du fichier puis d'en extraire le contenu [1].

Pour effectuer cela, on dispose des classes `BufferedOutputStream`, `BufferedWriter`, `BufferedInputStream` et `BufferedReader` qui permettent d'opérer des écritures ou des lectures par lots, notamment basées sur des lignes de texte complète.

Ces classes ne permettent pas de gérer un flux directement et doivent en fait être rattachées à un flux sur fichier préexistant [2].

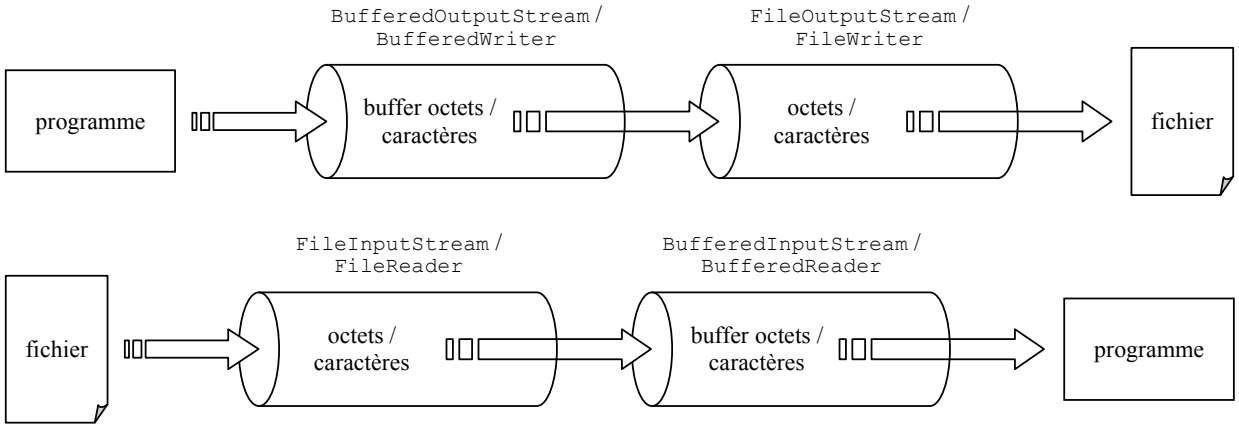


Figure 4.6 : buffer de flux d'entrée/sortie sur fichier texte

Méthodes de la classe `BufferedWriter` :

- `BufferedWriter(Writer)` : constructeur à partir d'un flux de sortie de caractères ;
- `write(String)` : écriture d'une chaîne dans le flux ;
- `newLine()` : écriture du caractère *fin de ligne* ;
- `close()` : fermeture du flux.

Méthodes de la classe `BufferedReader` :

- `BufferedReader(Reader)` : constructeur à partir d'un flux d'entrée de caractères ;
- `String readLine()` : lecture d'une ligne complète (jusqu'au caractère *fin de ligne*) ;
- `close()` : fermeture du flux.

[1] En réalité, la lecture ou l'écriture ne se fait pas qu'en une seule fois ; le buffer utilisé pour lire ou écrire dans le fichier a une taille prédéfinie (512 octets par défaut) ; néanmoins l'utilisation d'un tel procédé réduit considérablement le nombre d'accès au fichier.

[2] On dit qu'on réalise alors un chaînage de flux.

Ex. :

Ecriture.java

```
try {
   FileWriter outFile = new FileWriter("fichier.txt"); // ouverture du flux
   BufferedWriter out = new BufferedWriter(outFile); // chaînage du flux
   out.write("ligne 1"); // écriture dans le flux
   out.newLine(); // insertion d'un saut de ligne
   out.write("ligne 2"); // écriture dans le flux
   out.close(); // fermeture du flux
}
catch (IOException e) {
   System.out.println("erreur écriture fichier");
}
```

Lecture.java

```
try {
   String texte;

   FileReader inFile = new FileReader("fichier.txt"); // ouverture du flux
   BufferedReader in = new BufferedReader(inFile); // chaînage du flux
   while ((texte = in.readLine()) != null) // lecture ligne par ligne
      System.out.println(texte);
   in.close(); // fermeture du flux
}
catch (IOException e) {
   System.out.println("erreur lecture fichier");
}
```

4.5 GESTION DES FLUX D'OBJETS

4.5.1 La sérialisation

On appelle **sérialisation** la gestion de flux qui lit ou écrit des objets [1]. Un flux d'objets est aussi un flux de données (soit donc formaté), mais adapté aux objets.

Les informations lues ou écrites sont transférées comme des séries d'octets, d'où les termes de sérialisation (écriture d'un objet) et désérialisation (lecture d'un objet).

Pour pouvoir (dé)sérialiser un objet, il faut que celui-ci implémente l'interface `Serializable` (paquetage `java.io`) ; cette interface ne définit par ailleurs aucune méthode, l'objet n'a donc pas à être modifié pour pouvoir être lu ou écrit dans un flux.

Pour lire ou écrire des objets, on utilise alors les classes `ObjectInputStream` et `ObjectOutputStream` qui proposent respectivement les méthodes `readObject()` et `writeObject()`.

Ex. : Lecture/écriture d'un objet dans un fichier ; soit l'objet `Etudiant` suivant :

Etudiant.java

```
class Etudiant implements java.io.Serializable // peut être gérée par un flux
{
   String nom;
   float moyenne;

   public Etudiant(String sonnom, float samoyenne) // constructeur
   {
      this.nom = sonnom;
      this.moyenne = samoyenne;
   }
}
```

[1] Au sens POO (Programmation Orientée Objet).

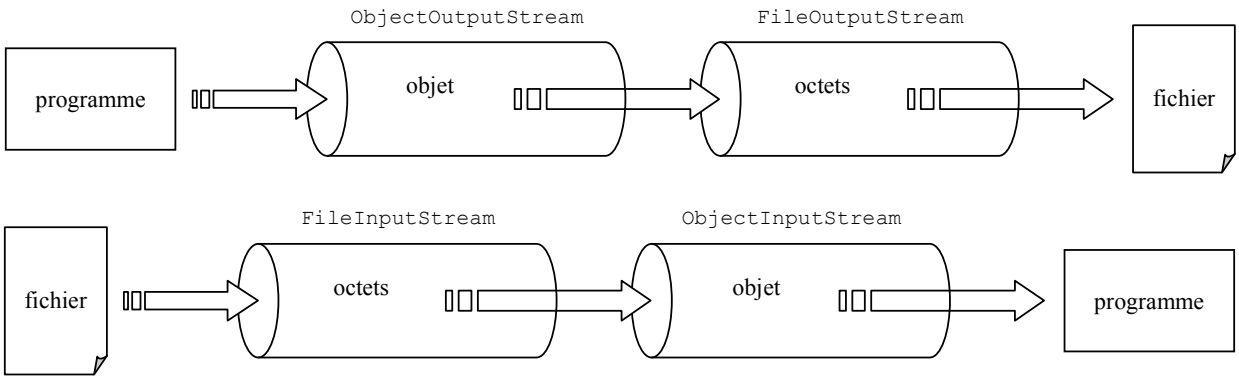


Figure 4.7 : exemple de sérialisation sur fichier

Serialisation.java

```
...
try {
   FileOutputStream fos = new FileOutputStream("obj.dat"); // ouverture du flux
   ObjectOutputStream oos = new ObjectOutputStream(fos); // chaînage du flux

   Etudiant etud1 = new Etudiant("Gérard",12f); // instanciation de l'objet
   oos.writeObject(etud1); // écriture de l'objet dans le flux (sauvegarde)
   oos.close(); // fermeture du flux
}
catch (IOException e) {
   System.out.println("erreur écriture objet");
}
...
```

Deserialisation.java

```
...
try {
   FileInputStream fis = new FileInputStream("obj.dat"); // ouverture du flux
   ObjectInputStream ois = new ObjectInputStream(fis); // chaînage du flux

   Etudiant e1 = (Etudiant)ois.readObject(); // relecture de l'objet (cast)
   ois.close(); // fermeture du flux
}
catch (IOException e) {
   System.out.println("erreur lecture objet");
}
catch (ClassNotFoundException cnfe) {
   System.out.println("erreur transtypage");
}
...
```

4.5.2 Persistance

On dit qu'un objet est **persistant** si son état est conservé entre deux instances d'une même application ou d'applications différentes. Le principe consiste à sauvegarder l'état de l'objet, généralement dans un fichier, pour pouvoir reconstruire ultérieurement un objet identique.

Le mécanisme de sérialisation est parfaitement adapté à la persistance d'un objet. On peut ainsi sauvegarder dans un fichier l'état d'un objet en mémoire lors de la fermeture d'une application ; cet état est ensuite relu lors du lancement de l'application [1].

Ex. : Sauvegarde des paramètres de la fenêtre graphique.

[1] Pensez à la mise en veille prolongée proposée par Windows…

ParametresIHM.java

```
import java.awt.*;

class ParametresIHM implements java.io.Serializable
{
   Point position;
   Dimension dimensions;

   public ParametresIHM(Point pos, Dimension dims)
   {
      this.position = pos;
      this.dimensions = dims;
   }
}
```

Cadre.java

```
Import java.io.*;

class Cadre extends Jframe
{
   ParametresIHM params;

   // constructeur récupérant les paramètres sauvegardés
   // si leur valeur ne peut être récupérée, alors des valeurs par défaut
   // sont attribuées
   public Cadre()
   {
      // relecture des paramètres à partir du fichier de sauvegarde
      try {
         FileInputStream fis = new FileInputStream("params.dat");
         ObjectInputStream ois = new ObjectInputStream(fis);
         params = (ParametresIHM)ois.readObject();
         ois.close();
      }
      catch (IOException e) { // affectation de valeurs par défaut en cas d'erreur
         params = new ParametresIHM(new Point(0,0), new Dimension(50,50));
      }
      catch (ClassNotFoundException cnfe) {
         System.out.println("Erreur: transtypage");
      }

      ...

      // utilisation des valeurs des paramètres récupérées
      this.setLocation(params.position);
      this.setSize(params.dimensions);
   }

   ...

   // sauvegarde de la valeur des paramètres de la fenêtre avant
   // fermeture de l'application
   void this_windowClosing(WindowEvent e)
   {
      // récupération des valeurs courantes des paramètres
      params.position = this.getLocation();
      params.dimensions = this.getSize();
```

```
        // sauvegarde des paramètres dans le fichier de sauvegarde
        try {
           FileOutputStream fos = new FileOutputStream("params.dat");
           ObjectOutputStream oos = new ObjectOutputStream(fos);
           oos.writeObject(params);
           oos.close();
        }
        catch (IOException ioe) {
           System.out.println("Erreur: création fichier paramètres");
        }

        System.exit(0);
     }
  }
```

---

5 LES APPLETS

5.1 GÉNÉRALITÉS

5.1.1 Définition

On appelle **applet** une application graphique qui s'exécute à l'intérieur de la fenêtre d'un navigateur internet.

Pour créer une applet, il faut dériver de la classe `Applet` contenue dans le package `java.applet` :

```
package MonPackage;

import java.applet.*; // importation du paquetage applet

public class Applet1 extends Applet // nouvel applet
{
   ...
}
```

5.1.2 Fonctionnement

Lorsque le navigateur charge une page HTML contenant une applet, il charge l'applet elle-même, puis instancie un objet de la classe principale (celle dérivée de la classe `Applet`). C'est alors la fenêtre du navigateur qui sert de conteneur graphique à l'applet.
Une applet s'exécute donc sous le contrôle du navigateur, et non pas sous celui de la machine virtuelle Java [1].

Une applet étant une forme d'application graphique (fenêtre graphique + console), la partie graphique s'affiche dans la fenêtre du navigateur alors que la partie texte s'affiche dans la console [2].

5.2 CYCLE DE VIE

Une applet suit un **cycle de vie**, c'est-à-dire que son exécution suit une logique bien précise, décrite par le schéma suivant :

[1] En réalité, tout navigateur capable d'exécuter une applet fait appel à une machine virtuelle Java, qu'elle soit intégrée sous forme de plugin (et souvent propriétaire de fait, ex. : Internet Explorer), ou bien installée sur le système comme une JVM classique (ex. : Opera).
[2] Par défaut, la console est généralement invisible, mais elle peut être affichée par simple paramétrage du navigateur.

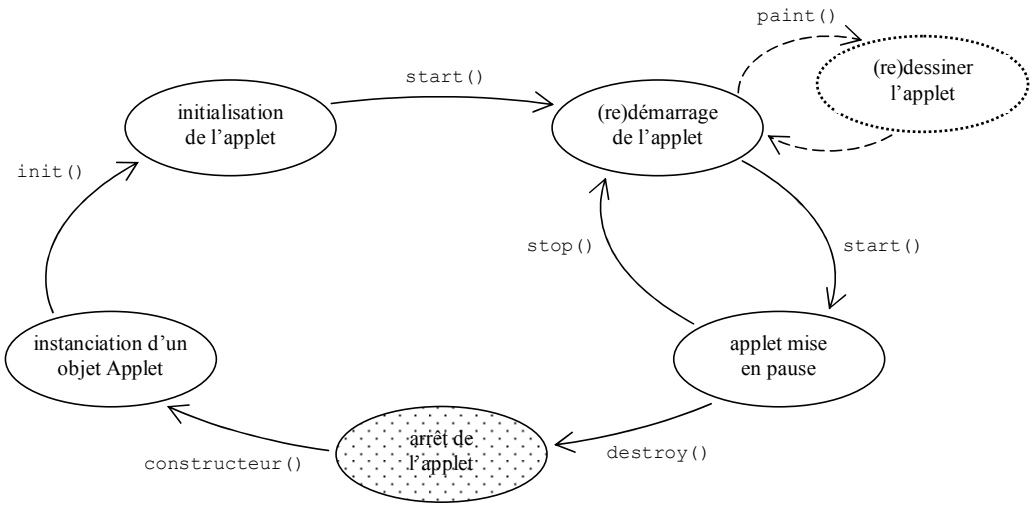


Figure 5.1 : cycle de vie d'une applet

Voici le détail de ce cycle de vie :

- `constructeur()` : appelée par la machine virtuelle du navigateur au tout début (méthode induite inaccessible au niveau du code) ;
- `init()` : appelée à la création de l'applet ;
- `start()` : appelée à la suite de `init()`, et appelée à nouveau à chaque fois que l'applet a été mise en pause par `stop()` ;
- `stop()` : appelée lorsque le navigateur est iconifié, lorsqu'une autre page est chargée dans le navigateur, ou lorsque le navigateur passe en arrière-plan (du point de vue des fenêtres du système d'exploitation) ;
- `destroy()` : appelée lorsque l'on ferme complètement la page ;
- `paint()` : appelée à chaque fois qu'il faut (re)dessiner l'applet dans le navigateur (au tout début, et à chaque fois que la page du navigateur repasse au premier plan).

Ces méthodes sont héritées de la classe `Applet`, mais elles peuvent être surchargées si besoin est.

Si on veut que l'applet puisse aussi être exécutée en dehors d'un navigateur web, comme une simple application graphique, il suffit d'écrire la méthode `main()` qui sera appelée lors d'une exécution classique [1].

5.3 EXÉCUTION D'UNE APPLET

5.3.1 Principes

Une fois l'applet mise au point, notamment en surchargeant les diverses méthodes héritées (`init()`, `start()`, etc.) en fonction de ses besoins, il faut permettre de charger l'applet dans le navigateur pour qu'elle puisse être exécutée.

Ceci ne peut pas être fait directement, et l'applet ne peut être chargée dans le navigateur que via une page HTML. Il faut « insérer » l'applet dans le code HTML, grâce à l'utilisation de la balise `<applet>`.

Ex. :

Bonjour.java

```
package MonApplet;

import java.applet.*;

public class Bonjour extends Applet
{
   public void paint(java.awt.Graphics g)
   {
      g.drawString("bonjour",100,100);
   }
}
```

[1] La méthode main() est utile aussi pour l'écriture et la mise au point de l'applet elle-même en phase de développement.

Applet.html

```
<html>
   <head></head>
   <body>
      <applet code="MonApplet.bonjour.class" width=400 height=200>
      </applet>
   </body>
</html>
```

5.3.2 La balise HTML <applet>

La forme générale de l'utilisation de la balise `<applet>` est la suivante :

```
<applet
   code = nom_fichier_.class (incluant le nom du paquetage)
   width = largeur_en_pixels
   height = hauteur_en_pixels
>
   message affiché par les navigateurs incapables de gérer la balise applet
</applet>
```

Le nom du fichier *.class* doit être complet, et doit donc contenir le nom du package (avec son arborescence).

Les 3 attributs de balise `code=`, `width=` et `height=` sont obligatoires. Il existe par contre des attributs optionnels :

- `codebase = adresse_base_URL` : précise le chemin du fichier *.class* de l'applet (par défaut c'est le chemin du document HTML qui est utilisé) ;
- `alt = message` : affiche un message pour les navigateurs comprenant la balise `<applet>` mais étant dans l'impossibilité d'exécuter l'applet (problème de droits par exemple) ;
- `name = instance` : précise le nom de l'instance d'une autre applet de la page HTML qui peut alors être utilisé pour établir une communication entre les 2 applets ;
- `archive = nom_archive_jar` : précise le nom de l'archive à charger qui contient le fichier *.class* mentionné par l'attribut `code=` ;
- `align = left / right / top / bottom` ;
- `hspace = espace_horizontal_en_pixels` ;
- `vspace = espace_vertical_en_pixels`.

Si on veut passer des paramètres à l'applet, il faut utiliser pour cela la balise `<param>` suivant la syntaxe : `<param name = nom_parametre value = valeur_parametre>` que l'on insère entre les balises `<applet>`.

On récupère alors la valeur `valeur_parametre` du paramètre `nom_parametre` dans le code de l'applet sous forme de chaîne en utilisant la méthode `getParameter()` héritée de la classe `Applet` suivant la syntaxe : `String str = this.getParameter("nom_parametre");`.

5.3.3 Exemple récapitulatif

Soit une applet à laquelle on va passer deux paramètres que l'on va afficher à l'écran (en les « dessinant ») :

Exemple1.java

```
package MonExempleApplet;

import java.awt.Graphics;

public class Exemple1 extends java.applet.Applet
{
   String s1, s2;

   public void init()
   {
      s1 = this.getParameter("chaine1");
      s2 = this.getParameter("chaine2");
   }
```

```
   public void paint(Graphics g)
   {
      g.drawString(s1 + "\n" + s2 + "\n",50,50);
   }
}
```

Applet.html

```
<html>
   <head>
      <title>exemple Applet</title>
   </head>
   <body>
      <h1>exemple Applet</h1>
      <applet
         code="MonExempleApplet.Exemple1.class"
         width=200
         height=100
         alt="votre navigateur ne peut pas exécuter cette applet"
         align="top"
         vspace=10
         hspace=10
      >
         <param name="chaine1" value="Bonjour">
         <param name="chaine2" value="à tous">
         votre navigateur ne peut pas exécuter d'applets Java
      </applet>
   </body>
</html>
```

5.4 EMPAQUETAGE DES APPLETS

Une applet est au départ pleinement destinée à être exécutée dans un cadre client/serveur et a pour portée principale internet. Par conséquent la question de la taille de l'applet est primordiale [1].

Pour réduire le temps de chargement, on peut empaqueter les fichiers *.class* nécessaires à l'exécution de l'applet en les archivant dans un fichier de type *.jar* [2].

Il faut alors préciser le nom de l'archive à charger en utilisant l'attribut `archive=` de la balise `<applet>`.

Ex. :
```
<applet
   code="MonApplet.bonjour.class"
   width=200
   height=100
   archive="bonjour.jar"
>
</applet>
```

L'archivage s'opère en ligne de commande où on utilise la commande `jar`, qui suit une syntaxe très proche de la commande d'archivage `tar` [3]. Cependant la commande `jar` fait plus qu'un simple archivage ; en effet elle rajoute un fichier appelé *fichier manifest* qui contient des informations sur l'archive *.jar* créée.

Lors de l'archivage sous forme d'archive jar, il convient de respecter scrupuleusement l'arborescence mettant en avant le nom du paquetage.

Ex. : Pour empaqueter tous les fichiers *.class* du sous-répertoire *MonApplet/* dans l'archive *bonjour.jar* :

```
jar cfv bonjour.jar MonApplet/*.class
```

[1] Comprendre : la taille en octets du fichier ou des fichiers .class constituant l'applet.
[2] Jar : Java ARchive – format de compression identique au format zip, même si un fichier .jar et un fichier .zip restent différents.
[3] La commande tar, issue du monde Unix, permet d'archiver (pas de compresser) des fichiers.

5.5 APPLETS ET SÉCURITÉ

L'exécution d'une applet est une opération sensible puisqu'une machine cliente exécute un programme provenant d'une autre machine (le serveur qui héberge l'applet). L'environnement d'exécution est donc très sécurisé afin de protéger la machine cliente des erreurs ou des malversations d'exécution de l'applet.

En conséquence, une applet n'accède qu'à un nombre restreint de ressources de la machine cliente. La définition de ces ressources accessibles constitue la politique de sécurité de l'exécution d'applets Java, laquelle est définie par le navigateur, et peut donc ainsi être paramétrée par l'utilisateur.

Cependant, certains éléments de sécurité sont généralement communs ; parmi ceux-ci, on notera en général qu'une applet ne peut pas :

- charger une bibliothèque ;
- définir de méthode native [1] ;
- avoir accès au système de fichiers d'une quelconque manière ;
- établir une connexion autre qu'avec la machine hébergeant l'applet ;
- exécuter un programme ;
- accéder aux propriétés du système ;
- fermer l'interpréteur Java ;
- lancer une impression ;
- etc.

Contrairement aux idées reçues, l'environnement d'exécution des applets est donc extrêmement sécurisé [2].

[1] Méthode compilée spécifiquement pour le type de processeur utilisé et n'étant donc pas exécutée par la JVM.

[2] En comparaison, les composants ActiveX (développés par Microsoft) sont nettement moins sécurisés et sont susceptibles d'être générateur de bugs, vecteur de trojans, etc. (voir http://www.microsoft.com/france/securite).

6 LE MULTITHREADING

6.1 INTRODUCTION

Les ordinateurs actuels utilisent des systèmes d'exploitation dits **multitâches** [1], c'est-à-dire que l'on peut exécuter plusieurs actions en même temps : éditer un fichier texte, écrire des données sur le disque dur, effectuer des calculs mathématiques, etc.

Chacune de ces actions est mise en œuvre par une application stockée en mémoire morte [2], et qui est chargée en mémoire vive [3] lors de son exécution ; l'exécution d'un programme est appelée **tâche** ou **processus**.

En réalité, ces tâches ne sont pas exécutées en même temps. En effet, un processeur ne peut gérer qu'une seule tâche à la fois à un instant *t*. En revanche, chaque tâche pouvant être décomposée en une liste de traitements successifs, le processeur peut s'interrompre entre deux traitements d'une même tâche, et reprendre une autre tâche au niveau du traitement où il s'était précédemment interrompu [4].

Le processeur se « partage » ainsi entre ces différentes tâches, en passant successivement d'un traitement à un autre traitement, d'une manière répétée et très rapide : l'utilisateur a ainsi l'impression que plusieurs tâches s'exécutent en même temps [5] car les traitements décomposant chacune des tâches évoluent en parallèle ; on parle alors d'*exécution pseudo-parallèle* [6].

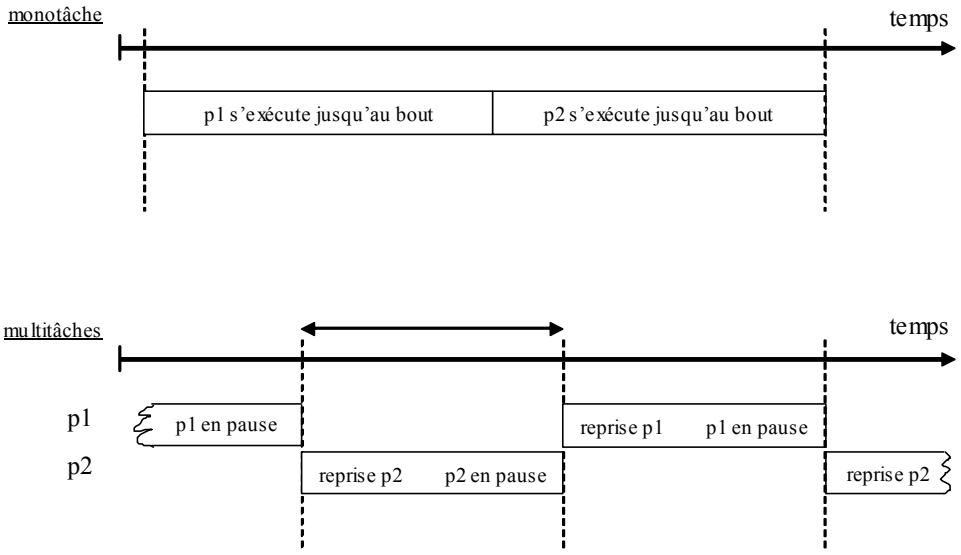


Figure 6.1 : comparaison de l'exécution de deux processus en monotâche et multitâches

[1] Le premier OS multitâches est UNIX, créé en 1969 par Ken Thompson ; chez Microsoft, ça commence en 1993 avec Windows NT 3.1.
[2] Par mémoire morte (=ROM), on entend tout support d'informations non volatile (disque dur, disquette, cédérom, clé usb, etc.).
[3] Par mémoire vive, on entend tout support d'informations volatile (RAM, mémoire cache, etc.).
[4] On parle là de *multitâches préemptif*, c'est-à-dire que le partage du temps processeur entre les tâches est géré par le système d'exploitation ; cette notion s'oppose au *multitâches coopératif* dans lequel ce sont les tâches elle-même qui se répartissent le temps processeur.
[5] On pourrait faire l'analogie avec le cinéma, où une succession très rapide de photos renvoie une impression de mouvement.
[6] *Pseudo*-parallèle, car les exécutions ne se font pas réellement en parallèle ; il faudrait en effet disposer de plusieurs processeurs pour cela.

6.2 DÉFINITIONS

Dans le principe, le **multithreading** est identique au multitâches [1], à ceci près que c'est le processus lui-même qui est découpé en sous-traitements parallèles, appelés **threads** [2], permettant ainsi au processeur de se partager entre ces différents traitements.

Cependant, à la différence du multitâches, pour que le processeur gère différents threads, chaque application doit avoir été développée dans une optique multithreads.

Les threads d'un même processus sont internes à ce processus et partagent ainsi un même espace mémoire, un même code, ainsi que les mêmes ressources.

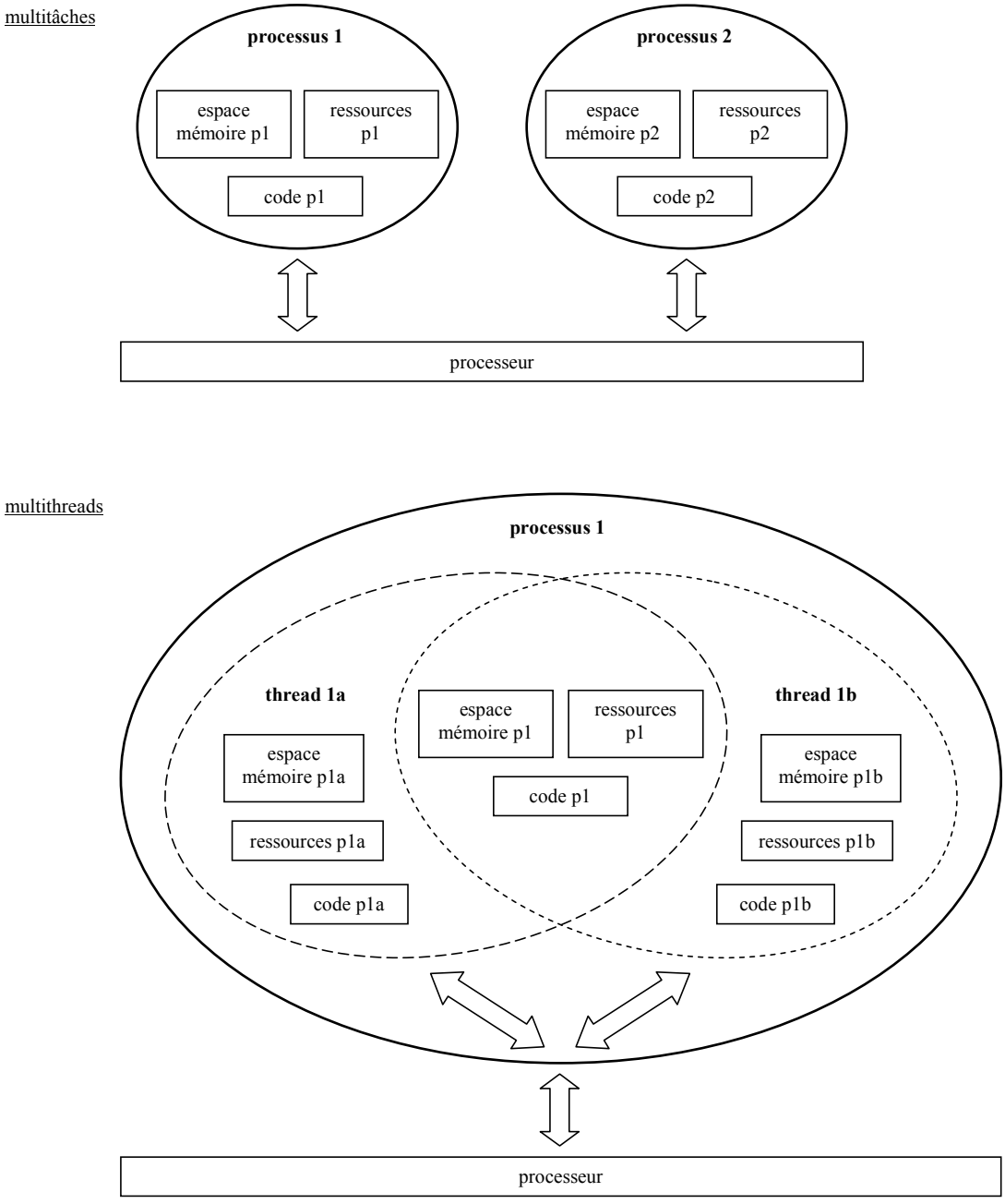


Figure 6.2 : comparaison entre multitâches et multithreading

[1] Un système d'exploitation multithreads est aussi multitâches.
[2] Ou processus légers, ou fils d'exécution (fil(s) d'exécution).

6.3 IMPLÉMENTATION

6.3.1 La classe Thread

Pour créer un **thread**, il faut écrire une classe qui dérive de la classe `Thread`, contenue dans le paquetage `java.lang`. On hérite ainsi de toutes les méthodes nécessaires à la gestion du thread.

Le point d'entrée du thread est la méthode `run()` [1] ; celle-ci doit donc définir les actions à réaliser par ce nouveau processus.

Pour lancer le thread au niveau du système, et permettre ainsi son exécution en parallèle du reste de l'application, il faut appeler la méthode héritée `start()`, laquelle se charge alors d'appeler implicitement la méthode `run()`.

Ex. : Une application qui utilise un thread.

MonThread.java

```
package ExempleThread;

public class MaClasse extends Thread // nouveau thread
{
   public void run() // surcharge de la méthode run()
   {
      ...
   }

   ...
}
```

Application.java

```
package ExempleThread;

public class Application
{
   ...

   public static void main(String[] args)
   {
      MaClasse th1 = new MaClasse(); // instanciation d'un thread
      th1.start(); // lancement du thread
      ...
   }
}
```

Une même application peut être décomposée en autant de threads qu'il en ait besoin [2].

Ex. : Une application qui utilise 2 threads (donc 3 exécutions pseudo-parallèles).

ThreadA.java

```
package ExempleThread;

public class ThreadA extends Thread
{
   public void run()
   {
      while (true)
         System.out.println("A");
   }
}
```

[1] Et peut donc être considérée comme l'équivalent de la méthode main() du thread.

[2] La gestion des threads est toujours coûteuse en ressources processeur (sans compter les problèmes de synchronisation et d'accès concurrents), il convient donc de créer des threads uniquement lorsque l'application le nécessite.

ThreadB.java

```
package ExempleThread;

public class ThreadB extends Thread
{
   public void run()
   {
      while (true)
         System.out.println("B");
   }
}
```

Application.java

```
package ExempleThread;

public class Application
{
   ...

   public static void main(String[] args)
   {
      ThreadA th1 = new ThreadA();
      ThreadB th2 = new ThreadB();
      th1.start();
      th2.start();
      ...
   }
}
```

6.3.2 L'interface Runnable

L'inconvénient de la méthode précédente (dérivation de la classe `Thread`), est que, comme l'héritage multiple est impossible en Java, on ne peut pas créer une classe multithreads qui dérive d'une autre classe quelconque. Pour pallier ce problème, il faut utiliser l'interface `Runnable` du paquetage `java.lang`.

La classe devant offrir le service de pouvoir être exécutée en tant que thread, doit alors implémenter l'interface `Runnable`, qui définit le prototype de la méthode `run()`. Toute instance d'une classe qui implémente cette interface devient ainsi un thread potentiel.

Ex. :

MonThread.java

```
package ExempleThread;

public class MaClasse implements Runnable // classe pouvant être "threadée"
{
   public void run() // implémentation de la méthode run()
   {
      ...
   }

   ...
}
```

Application.java

```
package ExempleThread;

public class Application
{
   ...
      public static void main(String[] args)
   {
      MaClasse mc1 = new MaClasse(); // instanciation d'un objet "runnable"
      Thread th1 = new Thread(mc1); // création d'un thread sur l'instance
      th1.start(); // lancement du thread
      ...
   }
}
```

On peut aussi décider de démarrer directement le thread dans le constructeur-même de la classe ; une simple instanciation suffit alors pour lancer le thread.

Ex. :

MonThread.java

```
package ExempleThread;

public class MaClasse implements Runnable // classe pouvant être "threadée"
{
   public MaClasse()
   {
      (new Thread(this)).start();
   }

   public void run() // implémentation de la méthode run()
   {
      ...
   }

   ...
}
```

Application.java

```
package ExempleThread;

public class Application
{
   ...
      public static void main(String[] args)
   {
      MaClasse mc1 = new MaClasse(); // instanciation d'un objet "runnable"
      ...
   }
}
```

Nb : L'inconvénient de cette dernière solution, est qu'en terme de ré-utilisabilité, il faut être certain que la classe, dans sa définition, adhère totalement à une exécution parallèle [1].

6.4 GESTION DES THREADS

6.4.1 Cycle de vie

En réalité, même si on ne crée pas de thread, toute application Java contient toujours au moins un thread, appelé *thread principal*, qui est en fait le processus correspondant à l'exécution de l'application. Les autres threads éventuels sont appelés alors *threads secondaires*.

[1] Par exemple, un composant de type « horloge » permettant d'afficher l'heure dans une application devra toujours être exécuté en parallèle du reste de l'application, la solution est donc bien appropriée ici.

Toute exécution de code Java est donc apparentée à un thread ; ainsi, une application prend fin lorsque tous ses threads sont terminés.

Un thread suit un **cycle de vie**, c'est-à-dire que son exécution suit une logique bien précise, décrite par le schéma suivant :

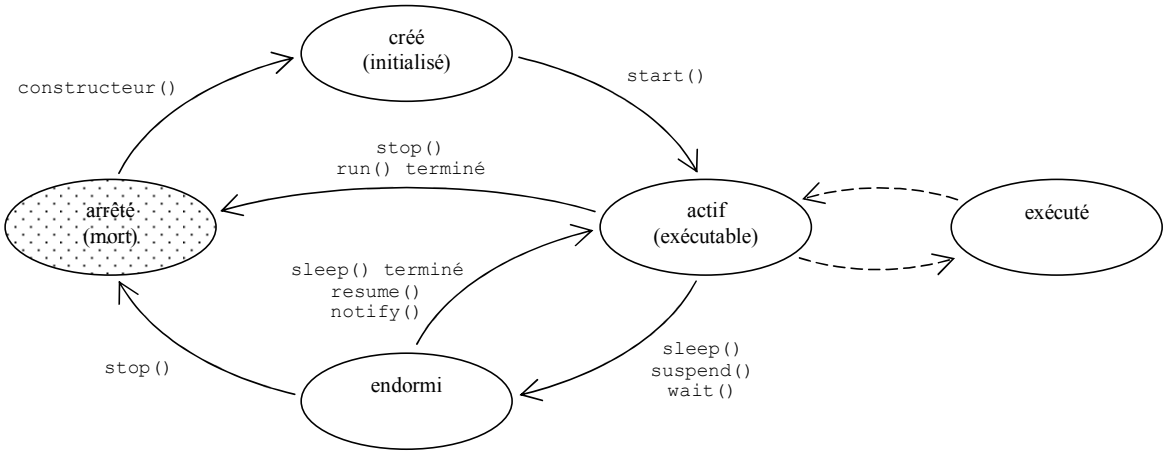


Figure 6.3 : cycle de vie d'un thread

Voici le détail de ce cycle de vie et des méthodes afférentes :

- `constructeur()` : construit une instance du thread ;
- `start()` : démarre le thread en exécution pseudo-parallèle, et appelle la méthode `run()` ;
- `stop()` : stoppe l'exécution du thread de manière définitive ;
- `sleep()` : place le thread dans l'état endormi pendant une durée déterminée ;
- `suspend()` : suspend l'exécution du thread en le plaçant dans l'état endormi ;
- `resume()` : reprend l'exécution d'un thread endormi ;
- `wait()` : place le thread dans un état endormi, en attente d'un signal provenant d'un autre thread avec la méthode `notify()` ou `notifyAll()` ;
- `notify()` : permet à un thread endormi avec la méthode `wait()` de reprendre son exécution.

6.4.2 Autres outils de gestion

Il est possible d'assigner une priorité à un thread en utilisant la méthode `setPriority()` [1].

Un thread peut être défini comme étant un thread *daemon*, qui sera alors terminé automatiquement si tous les threads « normaux » sont terminés. Pour cela on utilise la méthode `setDaemon()` [2].

Afin de faciliter la gestion des threads, il est possible de réunir plusieurs threads au sein d'un groupe de thread, avec la classe `ThreadGroup`. On peut alors gérer tous les threads de ce groupe en faisant appel à des méthodes reprises de la classe `Thread` : `suspend()`, `resume()`, `stop()`, etc.

Le développement d'une application multithreads soulève de nombreuses questions dont il faut impérativement tenir compte :

- définition de priorités et de dépendances (priorité, thread daemon) ;
- exclusion mutuelle pour l'accès à une ressource partagée (verrou, sémaphore, etc.).
- synchronisation entre threads (méthodes `wait()`, `notify()`, `notifyAll()`) ;

[1] Dix niveaux de priorités existent en tout, inclus dans la plage [1 ; 10] (1 : MIN_PRIORITY, 5 : NORM_PRIORITY, 10 : MAX_PRIORITY).

[2] En général, un thread accédant à une ressource partagée est positionné comme un thread daemon, afin d'éviter le blocage de l'accès à la ressource en cas d'arrêt ou de sortie anormale de l'application lors d'un dysfonctionnement.

7 LES SOCKETS

7.1 INTRODUCTION

Un **socket** est un moyen de communication de type point-à-point entre deux applications différentes, s'exécutant généralement chacune sur un ordinateur différent [1]. Basé sur le protocole TCP, la fiabilité de la communication est donc ainsi assurée, tout comme la réception des données dans l'ordre de l'émission.

Dans une communication par socket, on a donc une application qui est serveur, et l'autre qui est cliente :

- serveur : attend les connexions des clients, et reçoit les données des clients, ou bien leur en transmet ;
- client : initie une communication avec le serveur, afin de transmettre des données, ou bien d'en recevoir.

La communication par socket utilise les flux pour émettre ou recevoir les données.

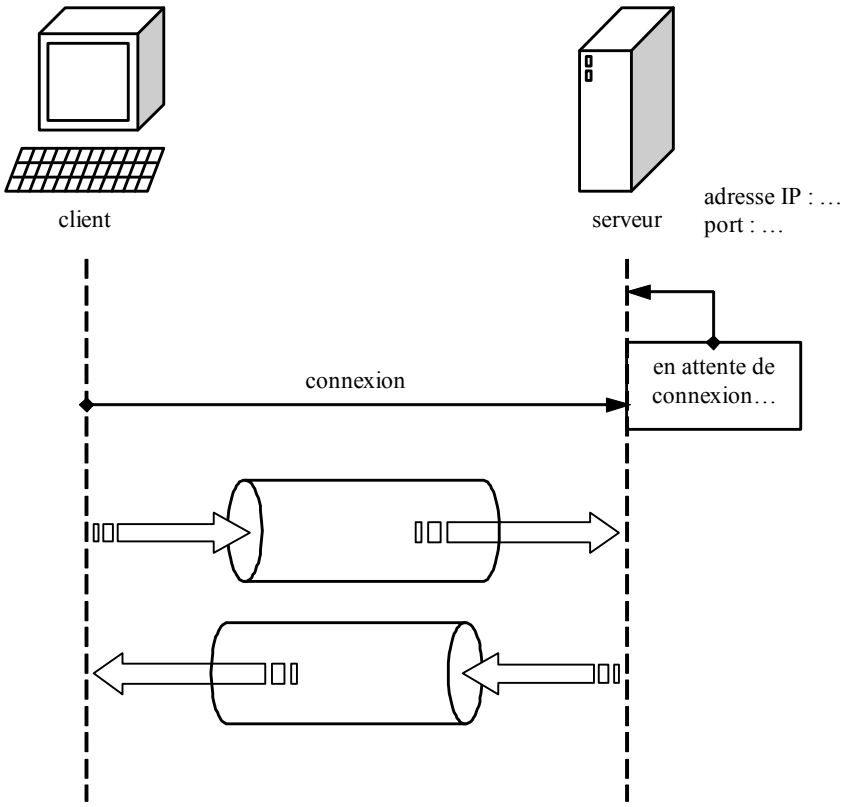


Figure 7.1 : communication client/serveur par socket

Les flux étant unidirectionnels, il faut créer deux flux distincts, à la fois au niveau du serveur et du client, pour émettre et recevoir des données [2]. Si l'on a uniquement besoin d'émettre des données dans un seul sens, alors un seul flux suffit.

Un socket utilise la notion de *port de connexion* du protocole TCP afin de permettre à d'autres communications d'être établies sur la même machine ; ce port correspond à un nombre entier codé sur 16 bits.

[1] Il est cependant tout à fait possible de faire communiquer 2 applications exécutées sur le même ordinateur en utilisant un socket.
[2] Un flux en écriture au niveau du serveur correspondant à un flux en lecture au niveau de client, et vice-versa.

Les ports 0 à 1023 sont réservés par convention [1] (appelés *well known ports* [2]) ; les ports 1024 à 49151 sont enregistrés (*registered ports* [3]) ; les ports 49152 à 65535 sont les ports publics et dynamiques.

7.2 IMPLÉMENTATION

Toutes les classes et méthodes nécessaires à la gestion des sockets font partie du paquetage `java.net`. Une fois la communication établie, on utilise des flux, donc des classes du paquetage `java.io`.

7.2.1 La classe Socket

Pour créer un socket, il faut utiliser la classe `Socket`, dont voici les principales méthodes utiles :

- `Socket(InetAddress adresseIpDistante, int portDistant)` : crée un nouveau socket connecté au serveur d'adresse IP `adresseIpDistante` sur le port `portDistant` ;
- `Socket(String nomDistant, int portDistant)` : crée un nouveau socket connecté au serveur dont le nom DNS est `nomDistant` sur le port `portDistant` ;
- `getInputStream() : InputStream` : renvoie un flux de lecture associé au socket ;
- `getOutputStream() : OutputStream` : renvoie un flux d'écriture associé au socket ;
- `close() : void` : ferme le socket.

7.2.2 Serveur socket

Le serveur socket permet de fournir un service accessible par toute machine sur le réseau. Il attend donc les connexions des clients sur un port précis, et répond aux connexions initiées sur ce port par les clients. Cette « réponse » se traduit par l'émission et/ou la réception de données en utilisant des flux.

Le principe du serveur est le suivant :

- création du serveur socket sur un port précis ;
- boucle sans fin ou contrôlée.
  - attente de connexion (fonction bloquante) ;
  - récupération du socket créé par le serveur socket suite à une connexion ;
  - récupération des flux d'entrée et/ou de sortie associés au socket ;
  - lecture et/ou écriture à partir des flux ;
  - fermer les flux ;
  - fermer le socket.

Pour créer un serveur socket, on utilise la classe `ServerSocket` ; parmi ses méthodes, on retiendra :

- `ServerSocket(int portLocal)` : crée un nouveau serveur socket en écoute sur le port `portLocal` ;
- `accept() : Socket` : attend une connexion, et une fois celle-ci établie, renvoie le nouveau socket associé ;
- `close() : void` : ferme le serveur socket.

Ex. : Un serveur socket en écoute sur le port 7474.

MonServeur.java

```
package ExempleSocket;

import java.net.*;
import java.io.*;

public class MonServeur
{
   private final int srvPort = 7474; // port local d'écoute
   private ServerSocket srvSock = null;
```

[1] C'est l'organisme international IANA (Internet Assign Numbers Authority), chargé de coordonner un certain nombre de standards sur l'internet (adressage IP, noms de domaines, etc.), qui gère ces spécifications (http://www.iana.org/assignments/port-numbers).

[2] Exemples : ports standards multi-plateformes (FTP : 20-21, HTTP : 80, SMTP : 25, POP3 : 110, HTTPS : 443, ...), ou ports reconnus comme tels (NetBios : 137-139, partage de fichiers et d'imprimante Windows / SaMBa : 445, …)

[3] Généralement utilisés temporairement par le système d'exploitation (aussi utilisables pour le domaine privé).

```
   public MonServeur() throws IOException
   {
      this.srvSock = new ServerSocket(this.srvPort); // instanciation du serveur
   }

   public void atWork() throws IOException
   {
      Socket sock = null;
      String msg = null;

      while (true) {
         sock = this.srvSock.accept(); // attente d'une connexion
         System.out.println("Connexion établie");

         InputStream is = sock.getInputStream(); // ouverture d'un flux d'entrée
         DataInputStream dis = new DataInputStream(is); // chaînage du flux
         msg = dis.readUTF(); // lecture du flux
         System.out.println("Message reçu: " + msg);

         OutputStream os = sock.getOutputStream();// ouverture d'un flux de sortie
         DataOutputStream dos = new DataOutputStream(os); // chaînage du flux
         dos.writeUTF("Message bien reçu"); // écriture dans le flux
         System.out.println("Message envoyé");

         dis.close(); // fermeture du flux d'entrée
         dos.close(); // fermeture du flux de sortie
         sock.close(); // fermeture du socket
      }
   }

   public static void main(String[] args)
   {
      try {
         MonServeur srv = new MonServeur(); // instanciation d'un serveur socket
         srv.atWork(); // lancement du serveur
      }
      catch (IOException ioe) {
         System.out.println("erreur serveur");
      }
   }
}
```

7.2.3 Client socket

Le client socket se connecte à un serveur bien précis, connaissant son adresse IP, ou son nom DNS, et le port de connexion. Une fois connecté, il communique avec le serveur en utilisant des flux, ce qui permet d'émettre ou de recevoir des données.

Pour créer un client socket, il faut utiliser la classe `Socket`.

Ex. : Un client socket se connectant sur un serveur en écoute sur le port 7474.

MonClient.java

```
package ExempleSocket;

import java.net.*;
import java.io.*;

public class MonClient
{
   final static int portSrv = 7474; // port de connexion distant

   public MonClient()
   {
      ...
   }
```

```
   public void receiveFrom(String ipSrv) throws IOException
   {
      Socket sock = new Socket(ipSrv,portSrv); // création d'un socket et connexion

      if (sock != null) {
         OutputStream os = sock.getOutputStream(); // ouverture d'un flux de sortie
         DataOutputStream dos = new DataOutputStream(os); // chaînage du flux
         dos.writeUTF("msg " + sock.getLocalPort()); // écriture dans le flux
         InputStream is = sock.getInputStream(); // ouverture d'un flux d'entrée
         DataInputStream dis = new DataInputStream(is); // chaînage du flux
         System.out.println("réponse srv: " + dis.readUTF()); // lecture du flux
         dos.close(); // fermeture du flux d'entrée
         dis.close(); // fermeture du flux d'entrée
         sock.close(); // fermeture du socket
      }
   }

   public static void main(String[] args)
   {
      try {
         MonClient clt = new MonClient(); // instanciation du client
         clt.receiveFrom("localhost"); // lancement du client
      }
      catch (IOException ioe) {
         System.out.println("erreur client");
      }
   }
}
```

7.3 CAS PRATIQUE

Si on observe le fonctionnement du serveur socket, on se rend compte d'une faiblesse dans son implémentation : lorsqu'une connexion d'un client a été initié, le serveur est alors dans l'incapacité de répondre à d'autres clients tant que la communication avec le premier client n'est pas terminé. Il faudrait pouvoir gérer plusieurs traitements en parallèle.

La solution est donc d'utiliser un thread secondaire qui va s'occuper de la communication avec le client, pendant que le thread principal continue d'être à l'écoute. Lorsque le serveur reçoit une connexion, il lance un thread secondaire, puis se repositionne en écoute [1].

Ex. : Un serveur socket multi-connexions en écoute sur le port 7474.

MonServeur.java

```
package ExempleSocket;

import java.net.*;
import java.io.*;

public class MonServeur {
   private final int srvPort = 7474; // port local d'écoute
   private ServerSocket srvSock = null;

   public MonServeur() throws IOException
   {
      this.srvSock = new ServerSocket(this.srvPort); // instanciation du serveur
   }

   public void atWork() throws IOException
   {
      Socket sock = null;
```

[1] Bien évidemment, durant le laps de temps très court où le thread principal lance le thread secondaire, un autre client peut éventuellement se connecter. Ceci est pallié car le serveur socket possède une file d'attente, qui par défaut est de 50 (nda : compulser les différentes formes du constructeur de la classe ServerSocket pour plus de détails).

```
      while (true) {
         sock = this.srvSock.accept(); // attente d'une connexion
         System.out.println("Connexion établie");

         new ServeurWork(sock); // lancement de la communication avec le client
      }
   }

   public static void main(String[] args)
   {
      try {
         MonServeur srv = new MonServeur(); // instanciation d'un serveur socket
         srv.atWork(); // lancement du serveur
      }
      catch (IOException ioe) {
         System.out.println("erreur serveur");
      }
   }
}
```

ServeurWork.java

```
package ExempleSocket;

import java.net.*;
import java.io.*;

public class ServeurWork implements Runnable // classe "threadable"
{
   private Socket sock;

   public ServeurWork(Socket sock)
   {
      this.sock = sock;
      (new Thread(this)).start(); // démarrage du thread
   }

   public void run()
   {
      String msg = null;

      try {
         InputStream is = sock.getInputStream(); // ouverture d'un flux d'entrée
         DataInputStream dis = new DataInputStream(is);
         msg = dis.readUTF(); // lecture du flux
         System.out.println("Message reçu: "+msg);

         OutputStream os = sock.getOutputStream(); // ouverture d'un flux de sortie
         DataOutputStream dos = new DataOutputStream(os);
         dos.writeUTF("Message bien reçu"); // écriture dans le flux
         System.out.println("Message envoyé");

         dis.close(); // fermeture du flux d'entrée
         dos.close(); // fermeture du flux de sortie
         sock.close(); // fermeture du socket
      }
      catch (IOException ioe) {
         System.out.println("erreur serveur thread");
      }
   }
}
```

7.4 DÉTAILS SUR LA COMMUNICATION CLIENT-SERVEUR

Les différents échanges d'une communication client-serveur par socket peuvent être résumés de la manière suivante :

MonServeur.java

```
public void atWork() throws IOException
{
   Socket sock = null;
   String msg = null;

   while (true) {
      sock = this.srvSock.accept();

      System.out.println("Connexion établie");

      InputStream is = sock.getInputStream();
      DataInputStream dis = new DataInputStream(is);

      msg = dis.readUTF();

      System.out.println("Message reçu: "+msg);

      OutputStream os = sock.getOutputStream();
      DataOutputStream dos = new DataOutputStream(os);

      dos.writeUTF("Message bien reçu");

      System.out.println("Message envoyé");

      dis.close();
      dos.close();
      sock.close();
   }
}
```

MonClient.java

```
public void receiveFrom(String ipSrv) throws IOException
{
   Socket sock = new Socket(ipSrv,portSrv);

   if (sock!= null) {
      OutputStream os = sock.getOutputStream();
      DataOutputStream dos = new DataOutputStream(os);
      dos.writeUTF("msg " + sock.getLocalPort());

      InputStream is = sock.getInputStream();
      DataInputStream dis = new DataInputStream(is);
      System.out.println("réponse srv: " + dis.readUTF());

      dos.close();
      dis.close();
      sock.close();
   }
}
```

① : Le serveur est en attente de connexion (la méthode `accept()` est bloquante) ; le client se connecte au serveur lorsque qu'un objet socket client est instancié ; le serveur poursuit alors son exécution.
② : Le serveur veut lire des données émises par le client, en utilisant donc un flux d'entrée (l'ensemble des méthodes du type `read()` des flux d'entrée sont bloquantes) ; le client envoie une donnée au serveur, en utilisant donc un flux de sortie (une méthode du type `write()`) ; la présence de données dans le flux permet alors au serveur de poursuivre son exécution.
③ : Le client veut lire des données émises par le serveur ; le serveur envoie une donnée au client via le flux ; le client poursuit alors son exécution.

A MOTS-CLEFS DU LANGAGE JAVA

A.1 ORDRE ALPHABÉTIQUE

A	abstract	assert				
B	boolean	break	byte			
C	case	catch	char	class	const	continue
D	default	do	double			
E	else	extends				
F	false	final	finally	float	for	
G	goto					
I	if	implements	import	instanceof	int	interface
L	long					
N	native	new	null			
P	package	private	protected	public		
R	return					
S	short	static	strictfp	super	switch	synchronized
T	this	throw	throws	transient	true	try
V	void	volatile				
W	while					

Les mots-clefs sont réservés, donc inutilisables comme noms de classe, d'interface, méthode, attribut ou variable [1].

A.2 CATÉGORIES

A.2.1 Paquetages, classes, membres et interfaces

class	Déclaration d'une classe.
extends	Dérivation d'une classe pour une sous-classe, ou d'une interface pour une sous-interface ; héritage des membres publics et protégés.
implements	Implémentation d'une interface par une classe.
import	Importation d'une classe ou d'une bibliothèque de classes (/paquetage).
instanceof	Opérateur de test d'appartenance d'un objet à une classe.
interface	Déclaration d'une interface.
new	Instanciation d'un objet (réservation de l'espace mémoire).
package	Déclaration d'un nom de paquetage (pour regrouper diverses classes).
super	À l'intérieur d'une classe, référence à la super-classe dont elle dérive.
this	À l'intérieur d'une classe, référence à l'objet de la classe elle-même.

[1] Même si Java distingue la casse, il est déconseillé d'utiliser les mots-clef, même avec une casse distincte.

A.2.2 Modificateurs de visibilité de membres

C : Classe, A : Attribut, M : Méthode, [1] : si interne.

	C	A	M	
`(package)`	X	X	X	Visibilité limitée aux classes du même paquetage que la classe du membre ; visibilité par défaut (à ne pas utiliser de manière explicite).
`private`	X [1]	X	X	Visibilité limitée à la propre classe du membre (une classe interne est considérée comme un membre de la classe).
`protected`	X [1]	X	X	Visibilité limitée à la propre classe du membre, aux sous-classes de la classe du membre et aux classes du même paquetage que la classe du membre.
`public`	X	X	X	Aucune restriction de visibilité.

A.2.3 Autres modificateurs

C : Classe, A : Attribut, M : Méthode, [1] : si interne.

	C	A	M	
`abstract`	X		X	M : méthode dont seul le prototype est défini, doit être implémentée par surcharge lors de la dérivation ; conséquence directe : classe abstraite. C : classe contenant une ou plusieurs méthodes abstraites ; ne peut pas être instanciée, utilisable seulement par dérivation.
`final`	X	X	X	A : attribut constant. M : méthode ne pouvant être surchargée lors de la dérivation. C : classe ne pouvant être dérivée.
`native`			X	M : méthode implémentée dans un autre langage (souvent C++) et accédée via une librairie dynamique (recherche de rapidité).
`static`	X [1]	X	X	A : attribut disponible même sans disposer d'instance de la classe ; commun à tous les objets de la classe. M : méthode disponible même sans disposer d'instance de la classe. C : classe interne disponible même sans disposer d'instance de la classe.
`synchronized`			X	M : méthode exécutable par un seul thread à la fois (applicable aussi pour un bloc de code de la méthode plutôt que la méthode entière).
`transient`		X		A : attribut non sauvegardé lors de la sérialisation de l'objet (sécurité).
`volatile`		X		A : attribut sauvegardé directement en RAM, sans être bufferisé, assurant ainsi la fiabilité de sa valeur instantanée lors de l'accès par divers membres.

A.2.4 Types primitifs

`boolean`	Booléen (`true`/`false`).
`byte`	Entier sur 1 octet.
`char`	Caractère sur 2 octets (standard Unicode).
`double`	Réel sur 8 octets.
`float`	Réel sur 4 octets.
`int`	Entier sur 4 octets.
`long`	Entier sur 8 octets.
`short`	Entier sur 2 octets.
`void`	Type « vide » (utilisé pour les déclarations de méthodes uniquement).

A.2.5 Structures de contrôle

`assert`	Vérification d'une condition nécessaire à l'exécution d'une instruction.
`break`	Sortie de boucle ou de bloc de code.
`case`	Test multiple – sélection (cf. `default`, `switch`).
`continue`	Poursuite de l'exécution de la boucle (branchement à l'itération suivante (test de continuité)).
`default`	Test multiple – sélection par défaut (cf. `case`, `switch`).
`do`	Boucle avec une itération minimale (cf. `while`).
`else`	Test simple – test conditionnel inverse (cf. `if`).
`for`	Boucle avec itération.

`if`	Test simple – test conditionnel (cf. `else`).
`return`	Sortie du bloc de code avec ou sans valeur de retour.
`switch`	Test multiple – test de sélection (cf. `case`, `default`).
`while`	Boucle avec une itération minimale (cf. `do`) ou aucune.

A.2.6 Gestion des exceptions

`catch`	Capture et traitement de l'exception (cf. `try`, `finally`).
`finally`	Déclaration d'un bloc de code « final » exécuté à la sortie du bloc `try` / `catch ()`, qu'une exception ait été signalée ou pas (cf. `try`, `catch`).
`throw`	Lancement explicite d'une exception (cf. `throws`).
`throws`	Déclaration des types d'exceptions qu'une méthode est susceptible de déclencher (cf. `throw`).
`try`	Déclaration d'un bloc de code susceptible de signaler une exception (cf. `catch`, `finally`).

A.2.7 Autres

`const`	Réservé (usage futur).
`goto`	Réservé (usage futur).
`false`	Valeur booléenne équivalente à FAUX / 0.
`null`	Pointeur nul (pas d'espace mémoire réservé).
`strictfp`	Calcul en virgule flottante avec respect du standard IEEE, assurant ainsi l'unicité du résultat du calcul quelque soit la JVM utilisée.
`true`	Valeur booléenne équivalente à VRAI / 1.

B RÉFÉRENCE DU LANGAGE

B.1 BIBLIOTHÈQUES DES CLASSES JAVA

L'API [1] Java est constitué de l'ensemble des bibliothèques des classes Java, et constitue toute la richesse du langage.

`java.lang`	Langage : Classes de base du langage Java.
`java.util`	Utilitaires : Structures de données (notamment dynamiques).
`java.io`	Entrées/sorties : Gestion des divers types d'entrées/sorties.
`java.text`	Texte : Gestion du texte, des dates, des nombres et des messages.
`java.math`	Maths : Calcul en précision entière et en virgule flottante.
`java.awt`	AWT (Abstract Window Toolkit) : Conception d'IHM et gestion d'événements (composants graphiques « lourds » [2]).
`javax.swing`	Swing : Conception d'IHM et gestion d'événements (composants graphiques « légers » [3]).
`java.applet`	Applet : Création et gestion d'applets (application intégrées à un navigateur).
`java.net`	Réseau : Accès et communication via le réseau.
`java.sql`	SQL : Accès aux bases de données, traitement des requêtes.
`java.beans`	Beans : Gestion des JavaBeans.
`java.lang.reflect`	Réflexion : Gestion et manipulation de concepts liés aux objets ou à la programmation en général.
`java.rmi`	RMI (Remote Method Invocation) : Programmation d'applications distribuées (accès à des objets distants via le réseau) [4].
`java.security`	Sécurité : Gestion de la sécurité par cryptographie.

B.2 ÉDITIONS DE LA PLATE-FORME JAVA

L'une des vocations principales du langage Java étant de développer une portabilité maximale, toute application Java est orientée multi-plateformes (Windows/Linux/Mac, x86/x64, etc.). Il existe donc autant de machines virtuelles différentes que de plates-formes d'exécutions différentes ; en sus, pour une plate-forme donnée, la JVM est déclinée en 3 éditions :

- J2SE – Standard Edition : classes pour le développement d'applications et d'applets ;
- J2EE – Enterprise Edition : J2SE + classes pour le développement d'applications d'entreprise (JDBC, Servlet/JSP, etc.) ;
- J2ME – Micro Edition : sous-ensemble de classes du J2SE destinées aux produits électroniques grand public mobiles (PDA, téléphones mobiles, etc.).

[1] Application Programming Interface (eng) ≡ Interface de Programmation d'Application (fr).

[2] Un composant graphique « lourd » (heavyweight GUI component (eng)) est dessiné en faisant appel au gestionnaire de fenêtres du système d'exploitation, c'est-à-dire qu'il aura le *look & feel* (apparence) du thème utilisé par le gestionnaire de fenêtres de l'OS (on parle parfois de *skin*). Pour dessiner le composant ainsi, la JVM communique avec l'IHM de l'OS en utilisant des méthodes natives (JNI : Java Native Interface) ; l'appel à ces méthodes est coûteuse en ressources, d'où le terme « lourd ». Ce choix a été fait au départ pour assurer une portabilité maximale aux applications Java. AWT ne comprend ainsi que des composants graphiques « lourds ».

[3] Un composant graphique « léger » (lightweight GUI component (eng)) est dessiné directement par la JVM selon un *look & feel* décidé par le développeur. Le composant graphique est ainsi complètement indépendant du gestionnaire de fenêtres de l'OS, et autant son *look & feel* peut être émulé pour ressembler à celui de l'OS, autant il peut être complètement différent puisqu'il est interchangeable à volonté avec extrêmement peu de modification du code. Une application Java peut ainsi avoir exactement la même apparence d'une plateforme à une autre. Swing, censé supplanter AWT, possède des équivalents aux composants graphiques AWT, dont le nouveau nom reprend l'ancien nom précédé d'un 'J' (Frame → JFrame). Swing ne comprend que des composants graphiques « légers » (codés intégralement en Java), exception faite des composants ayant besoin d'interagir directement avec le gestionnaire de fenêtres de l'OS, à savoir : JFrame, JApplet, JDialog et JWindow.

[4] Identique dans les fonctionnalités à CORBA, mais limité à un développement en Java (à l'inverse de RMI-IIOP).

Dans chaque édition, deux versions [1] sont disponibles :

- Java Runtime Environment (J2RE ou JRE) : Environnement d'exécution Java, nécessaire et suffisant pour exécuter un programme Java ; composé de la JVM et de l'API [2] ;
- Java Development Kit (J2DK ou JDK) : Kit de développement d'applications Java ; composé du JRE, de tous les outils nécessaires au développement (compilateur, etc.), et d'annexes (code source, démos, etc.).

B.3 VERSIONS DE LA PLATE-FORME JAVA

Depuis ses débuts en 1995, le langage Java a fortement évolué afin de voir sa stabilité s'améliorer, ainsi que de proposer de nouvelles classes et/ou bibliothèques de classes.

Nom officiel	Versions et dates de sortie	Nom de code	Principales évolutions	Version testée	Nombre de paquetages / classes	Taille JRE / JDK / Doc JDK (Mo)
Java 1.0	1995			n.d.	8 / 211	n.d.
Java 1.1	1997 (1.1.4)	Sparkler	Sérialisation, classes internes, JavaBeans, archives Jar, RMI, JDBC, internationalisation, JNI, …	1.1.8_10	22 / 477	6 / 20 / 12
	1997 (1.1.5)	Pumpkin				
	1998 (1.1.6)	Abigail				
	1998 (1.1.7)	Brutus				
	1998 (1.1.8)	Chelsea				
J2SE 1.2	1998 (1.2.0)	Playground	Swing + JFC, Java IDL (CORBA), JDBC 2.0, collections, compilateur JIT, réflexion, …	1.2.2_17	59 / 1524	20 / 65 / 86
	1999 (1.2.1)	(aucun)				
	1999 (1.2.2)	Cricket				
J2SE 1.3	1999 (1.3.0)	Kestrel	JNDI, HotSpot JVM, JPDA, amélioration performances, JavaSound, …	1.3.1_20	76 / 1840	18 / 82 / 108
	2000 (1.3.1)	Ladybird				
J2SE 1.4	2001 (1.4.0)	Merlin	JAXP (XML), JDBC 3.0, journalisation, expressions régulières, WebStart, JCE, …	1.4.2_19	135 / 2723	42 / 199 / 177
	2002 (1.4.1)	Hopper				
	2003 (1.4.2)	Mantis				
J2SE 5.0	2004 (1.5.0)	Tiger	Programmation générique, JAXP 1.3, JMX management, SASL, autoboxing, enum, metadata, …	1.5.0_18	166 / 3279	70 / 189 / 224
JSE 6	2006 (6.0)	Mustang		6.0_13	203 / 3793	86 / 295 / 265
JSE 7	2010 ?	Dolphin				

Nb : Les tests de taille ont été effectués avec la version Windows de la plate-forme ; les nombres de paquetages et de classes ont été comptabilisés par rapport à ceux référencés dans la documentation officielle (Doc JDK).

---

[1] Les 2 versions (JRE et JDK) sont libres d'utilisation, et sont téléchargeables gratuitement sur http://java.sun.com/.

[2] Un certain nombre de navigateurs utilisent directement le JRE pour l'exécution des applets (Opera, Mozilla Firefox), ce n'est pas le cas de Internet Explorer qui intègre sa propre machine virtuelle, mais que l'installation du JRE peut supplanter (hautement conseillé !).

C BIBLIOGRAPHIE

Wazir Dino, *Cours Java*, TS IRIS – LEGT Louis-Modeste Leroy – Évreux, 2002 ;

Sun Microsystems, *Documentation JDK API J2SE 1.4.2*, http://java.sun.com/j2se/1.4.2/docs/api/index.html, 2003 ;

Doudoux Jean-Michel, *Développons en Java v0.80.2*, http://perso.wanadoo.fr/jm.doudoux/java/tutorial/, 2004 ;

Borland, *Apprendre Java avec Jbuilder 7*, 2002 ;

Infini.fr, *Java*, http://www.infini-fr.com/Sciences/Informatique/Langages/Imperatifs/Java/java.html, 2005 ;

Clavel Gilles, **Mirouze Nicolas**, **Munerot Sandrine**, **Pichon Emmanuel**, **Soukal Mohamed**,
Java : la synthèse – vers la maturité avec le JDK 1.2 (2è édition), InterEditions, 1998 ;

Touraïvane, *Introduction à Java*, http://pages.univ-nc.nc/~touraivane/main/Enseignement.html,
Ecole Supérieure d'Ingénieurs de Luminy – Département Génie Bio Médical, 1998 ;

Eteks.com, *Du C/C++ à Java*, http://www.eteks.com/coursjava/tdm.html, 2005 ;

Hardware.fr, *Forum Java*, http://forum.hardware.fr/hardwarefr/Programmation/Java/liste_sujet-1.htm, 2005 ;

Niedermair Elke et Michael, *Programmation Java 2*, MicroApplication, 2000 ;

Delannoy Claude, *Exercices en Java*, Eyrolles, 2001 ;

Boichat Jean-Bernard, *Apprendre Java et C++ en parallèle*, Eyrolles, 2000 ;

CommentÇaMarche.net, *Java*, http://www.commentcamarche.net/java/, 2005 ;

Wikipedia – l'encyclopédie libre, *Java*, http://fr.wikipedia.org/, 2009 ;

Développez.com, *Introduction à Java*, http://java.developpez.com/, 2005 ;

Oplog, *Formation internet : Java*, OpLog – l'Opportunité Logicielle, 2002 ;

Longuet Patrick, *Livre d'or Java – votre guide complet de développement*, Sybex, 1996 ;

Peignot Christophe, *Langage UML*, http://info.arqendra.net/, 2009.