INTRODUCTION AU LANGAGE JAVA

Ce document est destiné aux personnes confirmées en C++ désireuses d'acquérir rapidement les notions de la programmation Java. Ce langage a été conçu dans le but de permettre une programmation orientée objet simple, accessible par le plus grand nombre. L'apprentissage du langage pour des personnes ayant déjà programmé avec un langage orienté objet est donc rapide. Se voulant un langage simple, Java est privé de certaines fonctionnalités, telles que les patrons (*templates* en C++) et l'héritage multiple (qui est remplacé par le concept d'*interface* beaucoup moins ambigu). Le langage est certes simple, mais l'utilisation des nombreuses bibliothèques l'est beaucoup moins. L'objectif de ce document n'est en aucun cas d'apprendre ces bibliothèques. Le lecteur est renvoyé pour cela à des documents plus complets, notamment le tutoriel Java de Sun Microsystems. Nous proposons plutôt ici une rapide introduction aux concepts de base du langage.

- **Structure générale**
  Présente l'organisation des fichiers et des classes en Java, les *packages*, la notion de machine virtuelle, la compilation en pseudo-code...

- **Classes et héritage**
  Explique comment manipuler des objets, définir des classes et leur relation d'héritage, implémenter des méthodes virtuelles...

- **Structures de données**
  Décrit la syntaxe des structures de données classiques comme les tableaux, les chaînes de caractères, les types primitifs et les conteneurs.

- **Flux d'entrée et de sortie**
  Explique comment manipuler les flux d'entrée et de sortie tels que le clavier, l'écran et les fichiers.

- **Graphisme avec l'AWT**
  Présente quelques fonctionnalités de base de l'AWT (*Abstract Window Toolkit*) pour réaliser des opérations graphiques simples et une interaction minimale avec l'utilisateur.

- **Concepts avancés**
  Introduit en vrac des notions plus avancées telles que les interfaces, les exceptions, les *threads*, les applets...

1. STRUCTURE GENERALE

ORGANISATION DES FICHIERS

Fichiers sources

Les fichiers sources en Java portent l'extension `.java`. Contrairement au C++, il n'est pas nécessaire de séparer dans des fichiers différents l'interface (`.hpp`) et l'implémentation (`.cpp`). En revanche, dans un fichier source, il ne peut y avoir qu'une seule classe visible de l'extérieur du fichier et c'est celle qui porte le même nom que le fichier. Un exemple, si l'on veut créer la classe `Liste`, elle doit être écrite dans le fichier `Liste.java`. Si d'autres classes comme `Cellule` sont définies dans ce même fichier, elles ne seront visibles que dans le fichier. Le mot-clé `public` qui désigne une classe visible de l'extérieur ne peut être utilisé que pour la classe `Liste`. Le fichier `Liste.java` contiendra par exemple:

```
public class Liste { ... }
class Cellule { ... }
```

Packages

Plusieurs classes peuvent être regroupées pour former un *package*. En C++, cela peut être fait par l'intermédiaire des *namespaces*, aucune contrainte n'est faite sur l'organisation physique des fichiers qui composent un *package*. En Java, le mot-clé `package` est employé pour indiquer à quel *package* appartient un fichier. Reprenons l'exemple de `Liste.java`, voici sa structure lorsqu'il fait partie du *package* `conteneur`.

```
package conteneur;
public class Liste { ... }
class Cellule { ... }
```

L'organisation physique des fichiers en Java est très importante. Prenons l'exemple suivant d'une arborescence de fichiers.

- Répertoire `maths/`
  - Répertoire `matrice/`
    - Fichier `Matrice.java`
    - Fichier `Vecteur.java`
  - Répertoire `grandnombre/`

Cela signifie que le *package* `maths` en contient deux autres, `matrice` et `grandnombre`. Le *package* `matrice` contient deux classes `Matrice` et `Vecteur`. La convention en Java est de nommer les classes avec la première lettre en majuscule et les *packages* avec la première lettre en minuscule. Pour accéder directement à la classe `Matrice`, il faut spécifier le chemin complet `maths.matrice.Matrice`. La commande

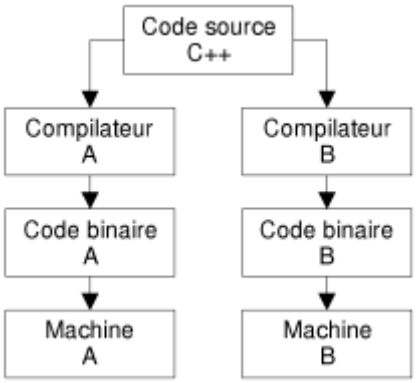
```
import maths.matrice.*;
```

au début d'un fichier permet d'utiliser n'importe quelle classe de `maths.matrice`, en l'occurrence `Matrice`, directement dans ce fichier. Cette commande est équivalente à `using namespace` en C++.

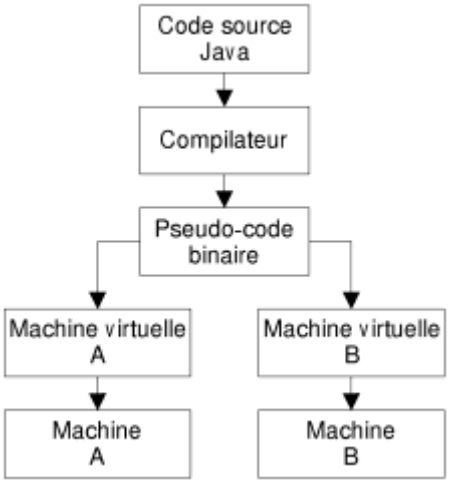
COMPILATION ET EXECUTION

Machine virtuelle Java et pseudo-code binaire

Dans la plupart des langages classiques, la compilation signifie la traduction de code source dans le langage du processeur, i.e. le language machine. Cela signifie que si l'on prend des fichiers binaires (i.e. des fichiers résultants d'une compilation), et qu'on les place sur une machine différente de celle pour laquelle ils ont été compilés, ces fichiers sont totalement inexploitables. La figure suivante illustre les processus de compilation et d'exécution d'un programme C++.



Java a été conçu pour être totalement portable. Au lieu de compiler le code source Java dans un langage machine spécifique (il existe quand même des compilateurs qui peuvent le faire), le code source est compilé dans un pseudo-langage machine, appelé souvent *pseudo-code binaire*, qu'un interpréteur logiciel, la *machine virtuelle Java* (*Java Virtual Machine*, JVM), peut décoder et exécuter. Ainsi, toute machine qui dispose d'une JVM peut exécuter des fichiers binaires Java. La figure suivante illustre les processus de compilation et d'exécution d'un programme Java.



Au cours de leur exécution, les fichiers binaires Java sont interprétés, cela signifie qu'en temps réel, la machine virtuelle Java traduit du code binaire Java en code binaire dédié pour la machine sur laquelle elle s'exécute. Cela implique nécessairement un ralentissement dans l'exécution des programmes, en comparaison avec les programmes C++ par exemple qui sont directement compilés pour la machine. Mais un mécanisme appelé *Just In Time* (JIT) permet d'optimiser cette phase de traduction du code binaire Java en code binaire dédié, ce qui améliore grandement les performances des programmes Java et les rapproche des performances des compilations dédiées. Les mauvaises performances que l'on rencontre souvent avec les programmes Java proviennent plutôt du fait que

l'héritage est trop utilisé, simplement parce que le langage est privé de la notion de patron (ce qui est en train d'être corrigé). En effet, l'héritage, par l'appel à une méthode virtuelle ou par le *downcast* (conversion d'un objet d'une classe en l'une de ses sous-classes), peut entraîner des ralentissements conséquents.

Compiler un code source

Voici maintenant la commande qui permet de compiler un programme Java. Considérons le fichier `Liste.java`, s'il dépend d'autres fichiers Java, le compilateur va se charger automatiquement de les compiler ou de les recompiler si nécessaire. Ainsi, la commande

```
javac -classpath <classpath> Liste.java
```

compile le fichier `Liste.java` et tous les fichiers dont il dépend si nécessaire (i.e. s'ils ont été modifiés ou s'ils n'ont jamais été compilés). Pour trouver ces fichiers, le compilateur regarde dans les répertoires listés par `<classpath>`. Pour des programmes simples, il suffit de préciser le répertoire courant (i.e. `<classpath>` = `.`). Pour éviter de préciser à chaque fois `<classpath>`, il existe la variable d'environnement `CLASSPATH` qui peut être initialisée à partir du *shell*. Il faut noter que les répertoires qui constituent `CLASSPATH` ou `<classpath>` doivent être séparés par `:` (sous Unix) ou `;` (sous Windows).

Exécuter un pseudo-code binaire

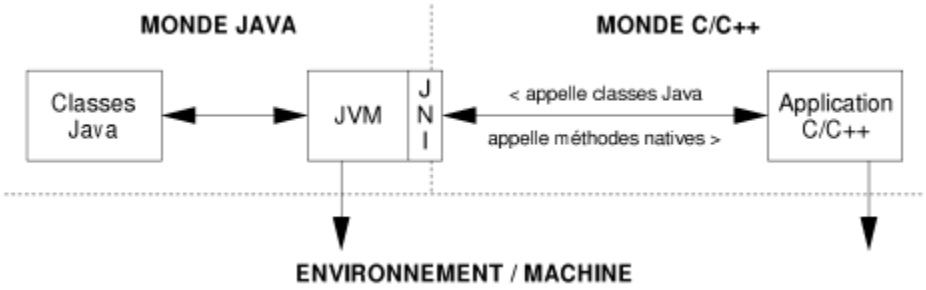
La compilation fournit pour chaque fichier `.java` compilé un fichier avec l'extension `.class` de même nom. Ce fichier peut être placé sur n'importe quelle machine avec une JVM. Pour l'exécuter, il suffit de taper la commande suivante.

```
java -classpath <classpath> Liste
```

Elle ordonne à la JVM d'interpréter le fichier `Liste.class`. Là aussi, il faut préciser éventuellement le chemin des fichiers dont dépend `Liste.class`, à l'aide de `<classpath>`. Nous verrons par la suite que tous les fichiers `.class` ne peuvent pas être exécutés directement, il faut qu'ils disposent d'une méthode particulière, équivalente de la fonction `main` en C++.

L'interface JNI

Pour diverses raisons, un programme implémenté en Java peut être plus lent que son implémentation en C++. L'efficacité peut être un critère important pour un logiciel, notamment ceux dédiés au calcul scientifique. En revanche, Java offre une portabilité intéressante au niveau du graphisme. Même si cela est souvent déconseillé, il est possible de combiner du code C++ avec du code Java. Typiquement, pour une application scientifique, les calculs seront effectués en C++ et l'interface en Java. Le mécanisme qui permet cela est JNI (*Java Native Interface*), une interface entre Java et C++. Le lecteur intéressé pourra consulter le livre: *The Java Native Interface: Programmer's Guide and Specification*, Sheng Liang, 1999, Addison-Wesley. Une partie de ce livre est disponible dans le tutoriel Java de Sun Microsystems. JNI permet à Java d'appeler des méthodes dites *natives* (i.e. fonctions C++ compilées dans un langage machine dédié), il permet également à C++ d'appeler des méthodes des classes Java. La figure suivante illustre sommairement ce mécanisme.



2. CLASSES ET HERITAGE

VARIABLE OBJET

Comme en C++, il existe en Java deux types de variables, celles qui font référence à des types primitifs (`int`, `char`, `double`...) et celles qui représentent des objets. La sémantique concernant les variables de type primitif en Java est identique à celle du C++. En revanche, les variables objets ont une signification bien différente.

Référence ou pointeur ?

Il est dit qu'en Java il n'existe pas de pointeurs et que toutes les variables objets sont des références sur des objets. Pour des personnes programmant en C++, le terme "référence" a une signification bien précise qui le différencie du terme "pointeur". En Java, le terme "référence" correspond en fait au terme "pointeur" du C++, à la seule différence qu'il est impossible pour le programmeur de manipuler explicitement l'adresse encapsulée dans le pointeur (i.e. impossible d'afficher l'adresse ou d'effectuer une opération arithmétique). Considérons l'exemple suivant qui déclare une variable `s` de la classe `String`.

```
String s;
```

La variable est initialisée implicitement à `null`, indiquant qu'elle ne référence aucun objet. A ce moment donc, aucun objet de la classe `String` n'est créé.

Création d'un objet

La création d'un objet doit être demandée explicitement par le programmeur, grâce au mot-clé `new`.

```
s = new String("hello !");
```

Comme en C++, `new` permet de passer des paramètres au constructeur de la classe `String`. L'exemple suivant montre également que les variables en Java sont des pointeurs.

```
void traduire(String t) { t=new String("Bonjour !"); }
...
String s = new String("Hello !");
traduire(s);
```

Les arguments des méthodes sont toujours passés par copie, ainsi le pointeur `s` (ou la référence `s`) est copiée, et c'est sa copie locale `t` qui référence la chaîne `"Bonjour !"`. La variable `s` reste donc inchangée après l'appel à la méthode `traduire`.

Accès aux propriétés d'un objet

Toutes les variables objets étant des pointeurs en Java, il n'est plus nécessaire, contrairement à C++, de faire la différence entre `->` et `.` pour appeler une méthode. En Java, seul le symbole `.` est utilisé pour appeler une méthode. L'exemple suivant appelle la méthode `length` de l'objet `s`.

```
int taille = s.length();
```

Destruction d'un objet

Toutes les variables étant finalement allouées dynamiquement, il serait fastidieux d'effectuer manuellement toutes les libérations de mémoire. Un mécanisme automatique, le *ramasse-miettes* (ou *garbage collector*), se charge de libérer les objets une fois qu'ils ne sont plus référencés par aucune variable. L'inconvénient ou l'avantage, selon les personnes, de cet automatisme est que le programmeur n'a aucune idée du moment où les objets sont libérés et donc de l'appel au destructeur. En effet, le ramasse-miettes n'est pas obligé de libérer la mémoire tout de suite, il peut très bien attendre d'avoir besoin de ressources pour effectuer un nettoyage.

DEFINITION D'UNE CLASSE

Syntaxe

La syntaxe pour définir une classe en Java est très proche de celle employée en C++. Le plus simple pour la présenter est tout simplement de montrer un exemple.

```
class Personne {
 // Attributs privés d'instance
 protected int numero;
 protected String nom;
 protected String prenom;

 // Attribut privé de classe
 protected static int compteur = 0;

 // Constante
 public static final int MAX = 100;

 // Méthodes publiques d'instance
 public int getNumero() { return numero; }
 public String getNom() { return nom; }
 public String getPrenom() { return prenom; }

 // Constructeur
 public Personne(String n,String p) {
  nom=n;
  prenom=p;
  numero=++compteur;
 }

 // Destructeur
 protected void finalize() {}

 // Méthodes publiques de classe
 public static int getNbPersonne() { return compteur; }
 public static void main(String args[]) { ... }
}
```

Les mots clés `public` et `protected` permettent respectivement de déclarer une propriété publique ou privée. Comme en C++, `static` permet de définir des propriétés de classe. La combinaison `static final` permet de définir des constantes. Le constructeur porte le nom de la classe, et bien entendu, il peut y en avoir plusieurs. Le destructeur lui porte le nom `finalize`. Il est conseillé de le déclarer uniquement s'il est nécessaire de l'employer, ce qui est très rare du fait que la libération de la mémoire est gérée automatiquement par le ramasse-miettes.

La méthode *main*

Lors de l'exécution d'un fichier, la JVM appelle la méthode `main` de la classe publique du fichier. Si cette méthode n'existe pas, l'exécution échoue. En paramètre, la méthode `main` reçoit un tableau de chaînes de caractères qui correspondent aux arguments de la ligne de commande, de manière similaire à C++ (cf. l'exemple de la classe `Personne`).

La super-classe *Object*

Avec Java, une classe définie sans super-classe hérite systématiquement de la classe `Object` qui se trouve ainsi au sommet de la hiérarchie d'héritage. Toute classe en Java hérite, directement ou indirectement, de cette classe `Object`. Nous verrons l'intérêt de cette classe au chapitre suivant.

HERITAGE

Héritage simple et méthodes virtuelles

Java n'autorise que l'héritage simple grâce au mot clé `extends`. L'héritage multiple n'est pas autorisé, à cause des nombreuses ambiguïtés qu'il soulève. Pour permettre néanmoins cette modélisation, la notion d'*interface* a été introduite, nous la présentons dans le dernier chapitre. Considérons l'exemple suivant d'héritage où la classe `Cercle` hérite de la classe `Forme`.

```
class Forme {
 protected float x;
 protected float y;

 public Forme(float a,float b) {
  x=a;
  y=b;
 }

 public float aire() { return 0.0; }
}

class Cercle extends Forme {
 protected float rayon;
 protected static final float PI = 3.1415;

 public Cercle(float a,float b,float r) {
  super(a,b);
  rayon=r;
 }

 public float aire() { return (PI*rayon*rayon); }
}
```

Contrairement au C++, les méthodes sont virtuelles par défaut en Java. Dans l'exemple, la méthode `aire` est donc virtuelle et surchargée dans la classe `Cercle`.

Appel à la super-classe

Le mot-clé `super` représente la super-classe de la classe dans laquelle on se trouve (il n'y en a qu'une puisqu'il n'y a pas d'héritage multiple). Ainsi, dans l'exemple avec la classe `Forme`, le constructeur `Cercle` appelle le constructeur de la classe `Forme` grâce à `super(...)`. Il est également possible pendant la surcharge d'une méthode d'appeler la version de la super-classe, comme le montre l'exemple suivant.

```
class Personne {
 protected String nom;
```

```
 protected String prenom;
 ...
 public String getInfo() { return (nom+" "+prenom); }
}

class Employe extends Personne {
 protected String poste;
 ...
 public String getInfo() { return (super.getInfo()+" "+poste); }
}
```

Méthodes finales

Le mot-clé `final` permet de rendre une méthode *finale*, i.e. non virtuelle, elle ne peut donc pas être surchargée. Il est conseillé de rendre les accesseurs (i.e. les méthodes `get...` et `set...` qui permettent de manipuler les attributs) finaux, l'appel à une méthode virtuelle étant une opération très coûteuse en comparaison avec l'appel à une méthode finale. Ainsi, dans l'exemple de la section précédente, il faut plutôt déclarer:

```
 public final int getNumero() { return numero; }
 public final String getNom() { return nom; }
 public final String getPrenom() { return prenom; }
```

Méthodes abstraites

Il est possible de définir des méthodes abstraites, i.e. sans corps, grâce au mot-clé `abstract`. L'exemple de la classe `Forme` est modifié pour proposer la méthode `aire` abstraite.

```
abstract class Forme {
 protected float x;
 protected float y;

 public Forme(float a,float b) {
  x=a;
  y=b;
 }

 public abstract float aire();
}

class Cercle extends Forme {
 protected float rayon;
 protected static final float PI = 3.1415;

 public Cercle(float a,float b,float r) {
  super(a,b);
  rayon=r;
 }

 public float aire() { return (PI*rayon*rayon); }
}
```

Dans ce cas, la classe `Forme` est abstraite également, ce qui signifie qu'elle ne peut pas être instanciée, aucun objet purement de cette classe ne peut être créé.

3. STRUCTURES DE DONNEES

TABLEAUX

Un tableau en Java est un objet, donc comme toute variable objet, une variable tableau est une référence. Lors de sa déclaration, il n'est pas possible d'indiquer la taille du tableau.

```
int t[];
```

La taille n'est fournie qu'au moment de l'allocation.

```
t=new int[10];
```

Pour connaître la taille d'un tableau, il suffit de consulter l'attribut `length` de celui-ci.

```
int taille = t.length;
```

Il est possible d'allouer des tableaux à plusieurs dimensions, comme le montre l'exemple suivant qui construit une matrice 3x4 de chaînes de caractères.

```
String m[][] = new String[3][4];
int i = 0;
int j;

while (i<3) {
 j=0;

 while (j<4) {
  m[i][j]=new String("case "+i+";"+j);
  ++j;
 }

 ++i;
}
```

Attention, lors de l'allocation d'un tableau d'éléments de type primitif, les cases du tableau contiennent ces éléments. Lors de l'allocation d'un tableau d'objets d'une classe quelconque, les cases du tableau contiennent des références d'objets initialisées à `null`.

Un tableau à plusieurs dimensions est en fait un tableau de tableaux. Et il n'est pas obligatoire que ces derniers soient tous de même taille.

```
int m[][] = new int[3][];
int i = 0;

while (i<3) {
 m[i]=new int[5*(i+1)];
 j=0;

 while (j<5*(i+1)) {
  m[i][j]=i+j;
  ++j;
 }

 ++i;
}
```

Il est possible, comme en C++, d'initialiser directement un tableau à sa déclaration.

```
int a[] = {1,2,3};
```

CHAINE DE CARACTERES

Contrairement au C++, les chaînes de caractères en Java ne sont pas des tableaux de caractères. Le seul moyen d'obtenir une chaîne de caractères est d'utiliser la classe `String`.

```
String r = new String("bonjour");
String t = "bonjour";
```

Pour connaître la taille d'une chaîne de caractères, il suffit d'appeler la méthode `length`.

```
int taille = t.length();
```

Il est possible de concaténer deux chaînes de caractères grâce à l'opérateur `+`.

```
String s = t+" toi";
```

Un objet de la classe `String` n'étant pas un tableau de caractères, il n'est pas possible d'utiliser l'opérateur `[]` pour accéder à un caractère de la chaîne. Pour cela, il faut utiliser la méthode `charAt`.

```
char j = s.charAt(3);
```

L'opérateur `==` appliqué directement sur deux variables compare leurs références et non pas les objets qu'elles pointent. Pour comparer réellement deux objets, il faut utiliser leur méthode `equals`, héritée de la classe `Object`, et normalement surchargée dans leur classe véritable.

```
if (s.equals("bonjour")) ...
```

La méthode `indexOf` permet de déterminer la position d'une chaîne de caractères dans une autre. Si aucune sous-chaîne n'est trouvée, alors la méthode renvoie `-1`.

```
int position = s.indexOf("toi");
```

Enfin, la méthode `substring` permet d'extraire une partie d'une chaîne de caractères, en indiquant la position du premier caractère et la position après le dernier caractère de la sous-chaîne.

```
String toi = s.substring(position,position+3);
```

TYPES PRIMITIFS

Voici la liste des types primitifs reconnus par Java. La plupart sont identiques à ceux du C++. La différence importante est que la norme Java impose un codage standard des types.

Type	Description
`byte`	entier signé, 8 bits
`boolean`	booléen, 1 bit, `true` / `false`
`char`	caractère, 16 bits, norme Unicode
`short`	entier signé, 16 bits
`int`	entier signé, 32 bits

`long`	entier signé, 64 bits
`float`	flottant, 32 bits, norme IEEE 754
`double`	flottant, 64 bits, norme IEEE 754

Les variables de type primitif ne sont pas reconnues comme des objets, ce qui est très gênant lorsqu'on ne dispose pas des patrons de composant. Comment proposer par exemple une liste chaînée qui puisse à la fois contenir des entiers de type `int` et des objets de type `String` ? La solution est de transformer les variables de type primitif en objets. Comme toutes les classes en Java héritent de la classe `Object`, il est alors possible de proposer une liste chaînée d'objets de type `Object`. Tous les types primitifs ont leur équivalent en objet.

Type primitif	Classe équivalente
`byte`	`Byte`
`boolean`	`Boolean`
`char`	`Character`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`

Nous présentons quelques fonctionnalités de la classe `Integer`, celles des autres classes sont très similaires. Tout d'abord, il est possible de créer à partir d'un entier un objet `Integer`.

```
Integer i = new Integer(4);
```

D'une chaîne de caractères, il est possible d'extraire un entier grâce à la méthode `parseInt` de la classe `Integer`. A noter que l'accès aux propriétés de classes en Java se fait en appliquant l'opérateur `.` sur la classe, alors qu'en C++ il s'agit de l'opérateur `::`.

```
int i = Integer.parseInt("4");
```

La méthode lève une exception (cf. le dernier chapitre) si la conversion de la chaîne en entier est impossible. Naturellement, d'un objet `Integer`, il est possible de récupérer sa valeur sous la forme d'un entier.

```
int j = i.intValue();
```

CONTENEURS

A l'instar de C++ avec la STL (*Standard Template Library*), Java fournit en standard un certain nombre de conteneurs très utiles pour accélérer le développement des applications. Cependant, comme la notion de patron n'existe pas encore en Java, les conteneurs sont conçus de manière à stocker des objets de la classe `Object`, à laquelle tout objet appartient. Mais nous verrons dans cette section que cela impose au programmeur d'employer le *downcast*, mécanisme lent puisqu'il doit détecter en temps réel la classe de l'objet. En outre, pour les types primitifs, les conteneurs ne sont pas utilisables directement, il faut transformer les variables en des objets. En résumé, ces classes conteneurs sont très utiles, mais ne sont pas d'une très grande efficacité. Nous ne citerons ici que deux conteneurs, le *vecteur* et le *dictionnaire*, qui sont assez représentatifs. Nous aborderons également l'*énumération*, qui permet un parcours séquentiel des éléments d'un conteneur. Notons que tous ces éléments font partie du *package* `java.util`.

Vecteur

Le vecteur, la classe `Vector`, équivalente à la classe `vector` de la STL, représente un tableau redimensionnable, autrement dit quand on lui ajoute des éléments, il adapte sa taille et réalloue de la mémoire si nécessaire. Voici un exemple simple qui ajoute dix entiers dans un vecteur.

```
Vector v = new Vector();
int i = 0;

while (i<10) {
 v.addElement(new Integer(i));
 ++i;
}
```

Le tableau suivant propose une liste non exhaustive de méthodes qui permettent de manipuler un vecteur.

Méthode	Description
`v.lastElement()`	Retourne le dernier élément de `v`.
`v.elementAt(i)`	Retourne l'élément de `v` à la position `i`.
`v.setElementAt(o,i)`	Affecte `o` à l'élément de `v` à la position `i`.
`v.insertElementAt(o,i)`	Insère l'élément `o` dans `v` à la position `i`.
`v.removeElement(i)`	Supprime l'élément de `v` à la position `i`.
`v.removeAllElements()`	Supprime tous les éléments de `v`.
`v.size()`	Retourne le nombre d'éléments de `v`.
`v.setSize(n)`	Redimensionne `v` à la taille `n`.
`v.indexOf(o)`	Retourne la position où se trouve l'élément de valeur `o`, si aucun n'est trouvé, `-1` est retourné.
`v.elements()`	Retourne une énumération des éléments de `v` (cf. la dernière section).

Dictionnaire

Le dictionnaire, la classe `Dictionary`, équivalente à la classe `map` de la STL, représente un conteneur associatif, où un élément est identifié par une clé. Voici un exemple simple qui associe à des entiers leur forme textuelle.

```
Dictionary d = new Hashtable();

d.put(new Integer(1),"un");
d.put(new Integer(2),"deux");
d.put(new Integer(3),"trois");
...
```

La classe `Dictionary` est abstraite, la sous-classe que nous utilisons ici est `Hashtable`. Pour récupérer ensuite la forme textuelle d'un nombre, il suffit de fournir la clé, i.e. le nombre, au dictionnaire.

```
String s = (String)d.get(new Integer(3));
```

La méthode `get` renvoie `null` si la clé n'est pas présente dans le dictionnaire. Il faut noter également que `get` retourne une référence sur un objet de type `Object`, et qu'il est nécessaire de le convertir, dans notre exemple en une référence sur un objet de type `String`, la syntaxe pour cela est identique à celle du C++.

Il est également possible de supprimer un élément du dictionnaire, en utilisant sa clé.

```
d.remove(new Integer(3));
```

La méthode `size` permet de connaître la taille d'un dictionnaire.

```
int taille = d.size();
```

Suivant les circonstances, il peut être intéressant de parcourir tous les éléments du dictionnaire ou toutes ses clés. Pour cela, le dictionnaire dispose de deux méthodes: `keys` qui renvoie une énumération des clés, et `elements` qui renvoie une énumération des éléments. La notion d'énumération est présentée dans la section suivante.

Enumération

Les conteneurs ne fournissent pas d'interface commune pour parcourir tous les éléments qu'ils contiennent, ce qui est nécessaire pour que les algorithmes soient indépendants des structures de données qu'ils manipulent. En outre, un conteneur comme le dictionnaire possède deux ensembles d'éléments, les clés et les valeurs associées. Pour disposer d'une manière commune de parcourir un ensemble d'éléments, indépendamment du type de conteneur, il existe la classe (ou plus exactement l'interface) `Enumeration` qui représente un ensemble ordonné d'éléments. Un conteneur dispose de méthodes permettant de le convertir en un objet de la classe `Enumération` (cf. les méthodes `keys` et `elements` des conteneurs `Vector` et `Dictionary`). Reprenons l'exemple de la section précédente pour parcourir les clés du dictionnaire.

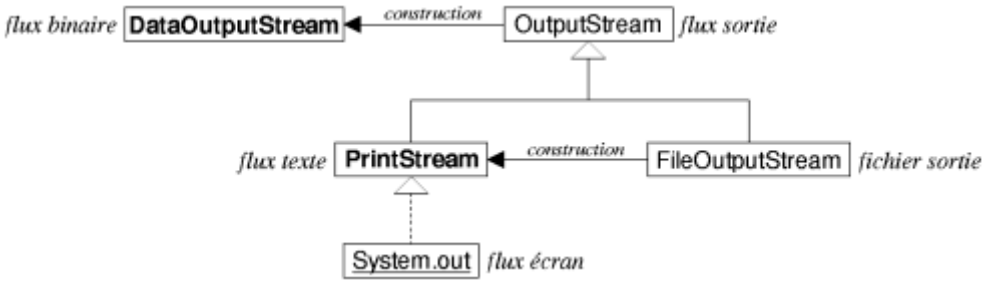
```
Enumeration e = d.keys();

while (e.hasMoreElements()) {
 i=(Integer)e.nextElement();
 ...
}
```

4. FLUX D'ENTREE ET DE SORTIE

FLUX DE SORTIE

Le schéma suivant représente la hiérarchie des flux de sortie (fichiers en mode écriture, écran) considérés dans ce chapitre. Ils héritent tous de la classe `OutputStream` du *package* `java.io`.



Mode texte

Le flux écran, l'objet `System.out`, appartient à la classe `PrintStream` qui représente un flux texte. Il est possible de construire un objet de cette classe à partir d'un fichier de sortie de la classe `FileOutputStream`.

```
FileOutputStream fichier = new FileOutputStream("sortie.txt");
PrintStream flux = new PrintStream(fichier);
```

Ainsi, les fichiers de sortie et l'écran appartiennent à la même classe qui dispose des méthodes `print` et `println`. Celles-ci permettent d'écrire en mode texte un objet (ou un élément de type primitif) dans le flux. La particularité de la méthode `println` est de retourner à la ligne après avoir écrit l'objet.

```
String s = new String("Hello !");
Date d = new Date();
int i = 5;

flux.println(s);
flux.println(d);
flux.println(i);
fichier.close();
```

Lorsqu'un objet est passé en paramètre à `print` ou `println` (e.g. l'objet `d`), sa méthode `toString` héritée de la classe `Object` est appelée et retourne une forme texte de l'objet. Il faut penser à fermer un flux de fichier, quel qu'il soit, dès qu'il n'est plus utile, à l'aide de la méthode `close`.

Mode binaire

Un objet de la classe `FileOutputStream` peut être manipulé en mode binaire. Pour cela, il faut construire un objet de la classe `DataOutputStream` qui permet d'écrire le contenu d'éléments primitifs dans le flux, grâce aux méthodes `writeBoolean`, `writeByte`, `writeChar`, `writeDouble`, `writeFloat`, `writeInt`, `writeLong`, `writeShort`... L'exemple suivant écrit un flottant et une chaîne de caractères en binaire dans un fichier.

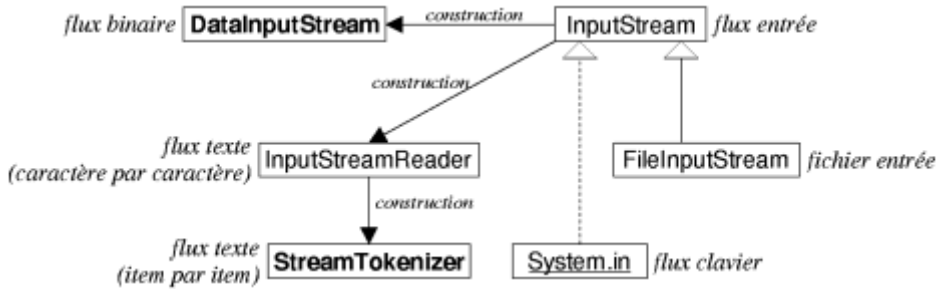
```
FileOutputStream fichier = new FileOutputStream("sortie.dat");
DataOutputStream flux = new DataOutputStream(fichier);
double d = 3.5;

flux.writeDouble(d);
flux.writeChars("hello !");
```

La méthode `writeChars` permet d'écrire en binaire tous les caractères d'une chaîne.

FLUX D'ENTREE

Le schéma suivant représente la hiérarchie des flux d'entrée (fichiers en mode lecture, clavier) considérés dans ce chapitre. Ils héritent tous de la classe `InputStream` du *package* `java.io`.



Mode texte

Le flux clavier, l'objet `System.in`, appartient directement à la classe `InputStream`, alors qu'un fichier d'entrée est un objet de la classe `FileInputStream`, héritant de `InputStream`. A partir de cette super-classe, il est possible de construire un flux `InputStreamReader` qui représente un flux d'entrée texte pouvant être manipulé caractère par caractère. De cet objet, il est nécessaire de construire un flux d'entrée texte de la classe `StreamTokenizer` si l'on désire extraire directement les items (entiers, flottants, mots...), le séparateur entre ces éléments étant le caractère espace par défaut.

```
FileInputStream fichier = new FileInputStream("entree.txt");
InputStreamReader lecteur = new InputStreamReader(fichier);
// ou lecteur = new InputStreamReader(System.in);
StreamTokenizer flux = new StreamTokenizer(lecteur);
```

Ainsi, les fichiers d'entrée et le clavier appartiennent à la même classe qui dispose de méthodes permettant de lire dans un flux texte un mot (i.e. une chaîne de caractères sans espace, grâce à la méthode `sval`) ou un flottant double précision (grâce à la méthode `nval`). L'exemple suivant lit un flottant puis un mot dans le flux.

```
double d;
String s;

if (flux.nextToken()==StreamTokenizer.TT_NUMBER)
 d=flux.nval;

if (flux.nextToken()==StreamTokenizer.TT_WORD)
 s=flux.sval;
```

La méthode `nextToken` avance d'un item chaque fois qu'elle est exécutée. Il est impératif de l'appeler avant `nval` ou `sval` la première fois, afin de placer le flux sur le premier item. La méthode `nextToken` retourne un entier qui indique sur quoi pointe le flux. Des constantes symboliques, de la classe `StreamTokenizer`, permettent d'analyser ce retour. `TT_WORD` signifie que le flux pointe sur un mot, `TT_NUMBER` sur un nombre, `TT_EOF` que la fin de fichier est atteinte...

L'exemple suivant lit un fichier rempli d'entiers.

```
int i;

while (flux.nextToken()!=StreamTokenizer.TT_EOF) {
 i=(int)flux.nval;
 System.out.println(i);
}
```

Mode binaire

Un objet de la classe `FileInputStream` peut être manipulé en mode binaire. Pour cela, il faut construire un objet de la classe `DataInputStream` qui permet de lire des éléments de type primitif dans le flux, grâce aux méthodes `readBoolean`, `readByte`, `readChar`, `readDouble`, `readFloat`, `readInt`, `readLong`, `readShort`... L'exemple suivant lit les flottants contenus dans un fichier binaire et les affiche à l'écran.

```
FileInputStream fichier = new FileInputStream("entree.dat");
DataInputStream flux = new DataInputStream(fichier);
double d;

while (flux.available()!=0) {
 d=flux.readDouble();
 System.out.println(d);
}
```

La méthode `available` permet de détecter la fin d'un flux binaire puisqu'elle renvoie le nombre d'octets encore disponibles dans ce flux.

5. GRAPHISME AVEC L'*AWT*

Il existe actuellement deux bibliothèques standards en Java pour le graphisme, l'AWT (*Abstract Window Toolkit*) et Swing. La seconde est plus récente, mais plus délicate à manipuler. Nous proposons donc de présenter plutôt des composants de l'AWT qui se trouvent dans le *package* `java.awt`.

COMPOSANTS GRAPHIQUES

La première chose à faire dans toute interface graphique est de fournir une ou plusieurs fenêtres qui vont accueillir les composants graphiques de l'application. Nous nous intéressons simplement ici à l'ouverture d'une fenêtre dans laquelle on peut dessiner simplement. Mais il faut savoir que l'AWT propose une hiérarchie de nombreux composants graphiques comme des boutons, des menus déroulants..., tous héritant de la super-classe `Component`. L'exemple qui suit ouvre deux fenêtres, chacune contenant un composant simple, de la classe `Panel`, qui prend tout l'espace de la fenêtre, et dans lequel est affiché un rectangle.

```
import java.awt.*;

public class Fenetre extends Frame {
 protected Panneau panneau;

 public Fenetre(String titre) {
  super(titre); // Appelle le constructeur de la super-classe.
  panneau=new Panneau(); // Crée un panneau.
  setLayout(new BorderLayout()); // Crée le gestionnaire de placement.
  add(panneau,BorderLayout.CENTER); // Place le panneau dans la fenêtre.
  pack(); // Dimensionne la fenêtre et ses composants.
  setResizable(false); // Indique que la fenêtre n'est pas redimensionnable.
  setVisible(true); // Affiche la fenêtre.
 }

 public Dimension getPreferredSize() // Utilisée par la méthode pack pour
 { return (new Dimension(200,200)); } // connaître la taille de la fenêtre.

 public static void main(String args[]) {
  new Fenetre("Fenetre 1");
  new Fenetre("Fenetre 2");
 }
}

class Panneau extends Panel {
 public Panneau() { super(); }

 public void paint(Graphics g) {
  super.paint(g);
  g.drawRect(10,10,getWidth()-20,getHeight()-20);
 }
}
```

La classe `Fenetre` hérite de `Frame`, dont le constructeur reçoit une chaîne de caractères qui est le titre de la fenêtre. Dans le constructeur de `Fenetre`, un composant `Panel` est créé et positionné sur la fenêtre. La méthode `setLayout` permet de spécifier quel est le gestionnaire chargé de positionner les composants sur la fenêtre. Nous utilisons ici un gestionnaire de la classe `BorderLayout`, qui permet de positionner les composants de manière relative, mais il en existe d'autres avec des

politiques de placement différentes. La méthode `add` ajoute le composant `Panel` dans la fenêtre, en demandant de le centrer (grâce à `BorderLayout.CENTER`). Le gestionnaire le positionnera ensuite automatiquement, mais étant le seul composant, il occupera tout l'espace de la fenêtre. La méthode `pack` redimensionne la fenêtre et les composants qu'elle contient, et pour connaître la taille de la fenêtre, elle interroge sa méthode `getPreferredSize`. Les méthodes `setResizeable` et `setVisible` indiquent respectivement que la fenêtre n'est pas redimensionnable et qu'elle est visible.

La méthode `paint` dessine un rectangle dans le composant `Panel`, mais cette opération sera détaillée plus tard. Notons simplement que la méthode `paint` est appelée chaque fois que l'on a besoin d'afficher le composant (i.e. la première fois qu'il apparaît à l'écran, lorsqu'une fenêtre qui cachait le composant le dévoile à nouveau...). Il est d'usage de mettre dans cette méthode le code correspondant à l'affichage du composant.

GESTION DES EVENEMENTS

En exécutant l'exemple précédent, on s'aperçoit qu'il est impossible de fermer les fenêtres, simplement parce que l'événement engendré par le clic de la souris sur le bouton de fermeture de la fenêtre n'a pas été récupéré et traité par le programme. Pour cela, il faut mettre en place des objets, les *listeners*, chargés de cette écoute. Ils appartiennent à la classe (ou plus exactement à l'interface) `EventListener`. Par exemple, lorsque l'événement de fermeture de la fenêtre est levé, la méthode `windowClosing` du *listener* attaché à la fenêtre est exécutée.

Evénements d'une fenêtre

Pour traiter cet événement, il faut donc créer un *listener* de fenêtre, de la classe `WindowAdapter`, et surcharger sa méthode `windowClosing`. Il suffit ensuite de le rattacher à la fenêtre par la méthode `addWindowListener`. L'exemple précédent est complété pour que la fermeture des fenêtres soit possible. L'une des deux fenêtres, la fenêtre maîtresse terminera l'application lorsqu'elle se fermera, alors que l'autre, la fenêtre esclave, ne stoppera pas l'application à sa fermeture.

```
import java.awt.*;
import java.awt.event.*;

public class Fenetre extends Frame {
 protected boolean maitre;
 protected Panneau panneau;

 protected class EvenementFenetre extends WindowAdapter {
  public void windowClosing(WindowEvent e) {
   dispose(); // Ferme la fenêtre.
   if (maitre) System.exit(0);
  }
 }

 public Fenetre(String titre,boolean m) {
  super(titre);
  panneau=new Panneau();
  maitre=m;
  setLayout(new BorderLayout());
  add(panneau,BorderLayout.CENTER);
  addWindowListener(new EvenementFenetre());
  pack();
  setResizable(false);
  setVisible(true);
 }

 public Dimension getPreferredSize()
 { return (new Dimension(200,200)); }
```

```
 public static void main(String args[]) {
  new Fenetre("Fenetre maitresse",true);
  new Fenetre("Fenetre esclave",false);
 }
}
```

Toutes les classes permettant la gestion des événements des composants graphiques se trouvent dans le *package* `java.awt.event`.

Evénements du clavier

Pour gérer les événements du clavier, le mécanisme est identique à celui des événements d'une fenêtre. La différence ici est la classe du *listener*, qui est `KeyAdapter`, et donc les méthodes à surcharger. Le plus simple pour introduire tous les événements importants est un exemple. Voici un *listener* pour les événements du clavier.

```
protected class EvenementClavier extends KeyAdapter {
 public void keyPressed(KeyEvent e) {
  if (e.getKeyChar()==KeyEvent.CHAR_UNDEFINED)
   System.out.println(getTitle()+", touche de controle "
                      +e.getKeyCode()+" enfoncee...");
  else
   System.out.println(getTitle()+", touche '"
                      +e.getKeyChar()+"' enfoncee...");
 }

 public void keyReleased(KeyEvent e) {
  if (e.getKeyChar()==KeyEvent.CHAR_UNDEFINED)
   System.out.println(getTitle()+", touche de controle "
                      +e.getKeyCode()+" relachee...");
  else
   System.out.println(getTitle()+", touche '"
                      +e.getKeyChar()+"' relachee...");
 }
}
```

Le *listener* capture entre autres deux types d'événements: "une touche est enfoncée" (la méthode `keyPressed` est appelée) et "une touche est relâchée" (la méthode `keyReleased` est appelée). Les méthodes appelées lors d'un événement du clavier reçoivent un paramètre de la classe `KeyEvent`. Deux méthodes `getKeyChar` et `getKeyCode` de cet objet permettent respectivement de récupérer soit le caractère associé à la touche impliquée dans l'événement, soit directement son code. La méthode `getKeyChar` retourne `KeyEvent.CHAR_UNDEFINED` si la touche n'a pas de caractère associé. La méthode `getKeyCode` retourne un entier, mais des constantes symboliques sont fournies pour déterminer plus facilement quelle touche a été appuyée ou relâchée. Voici un extrait de ces codes.

Constante	Touche
`VK_LEFT`	Flèche gauche
`VK_RIGHT`	Flèche droite
`VK_UP`	Flèche haut
`VK_DOWN`	Flèche bas
`VK_PAGE_UP`	Flèche page haut
`VK_PAGE_DOWN`	Flèche page bas
`VK_TAB`	Tabulation
`VK_BACK_SPACE`	*Backspace*
`VK_SHIFT`	*Shift*

VK_CONTROL	*Ctrl*
VK_ALT	*Alt*
VK_ALT_GRAPH	*Alt Gr*

Il ne faut pas oublier de rattacher le *listener* de clavier à la fenêtre, pour cela il faut rajouter la ligne suivante dans le constructeur de la fenêtre, de manière semblable au *listener* de fenêtre.

```
addKeyListener(new EvenementClavier());
```

Evénements de la souris

Pour gérer les événements de la souris, le mécanisme est identique aux autres types d'événements. La classe du *listener* à surcharger dans ce cas est `MouseAdapter`. Voici un exemple de *listener* pour les événements de la souris. Ce *listener* est placé sur le composant `Panel`. C'est sur lui que l'on clique, il reçoit donc les événements de la souris.

```
class Panneau extends Panel {
 protected class EvenementSouris extends MouseAdapter {
  public void mousePressed(MouseEvent e)
  { getGraphics().drawRect(e.getX(),e.getY(),1,1); }

  public void mouseReleased(MouseEvent e)
  { System.out.println("Souris relachee..."); }
 }

 public Panneau() {
  super();
  addMouseListener(new EvenementSouris());
 }

 public boolean isFocusTraversable() { return false; }

 public void paint(Graphics g) {
  super.paint(g);
  g.drawRect(10,10,getWidth()-20,getHeight()-20);
 }
}
```

Lorsque l'on clique sur le composant `Panel`, le focus du clavier lui est attribué par défaut. Ce qui signifie que le *listener* de clavier attaché à la fenêtre ne reçoit plus les événements du clavier. Pour éviter cette perte de focus, il suffit d'indiquer que le composant `Panel` ne désire pas obtenir le focus, en surchargeant la méthode `isFocusTraversable` pour qu'elle retourne `false`.

DESSINER

Il est possible de dessiner dans la plupart des composants héritant de la classe `Component`, mais cela ne se fait pas directement, il faut utiliser un intermédiaire, un *contexte graphique*, un objet de la classe `Graphics`. Pour l'obtenir, il suffit d'appeler la méthode `getGraphics` du composant, elle retourne alors un objet de cette classe (cf. la méthode `mousePressed` de l'exemple précédent). Mais certaines méthodes, comme `paint`, reçoivent directement cet objet en paramètre (cf. l'exemple précédent). Un contexte graphique possède des méthodes pour afficher des formes dans le composant qu'il référence. Le tableau suivant récapitule les plus importantes.

Méthode	Description
`drawArc(x,y,l,h,a1,a2)`	Trace un arc de cercle dans le rectangle de coin (`x`;`y`), de largeur `l` et de hauteur `h`. L'arc commence à l'angle `a1` et se termine à l'angle `a2`,

	exprimés en degré.
`fillArc(x,y,l,h,a1,a2)`	La même chose avec une forme pleine.
`drawLine(x1,y1,x2,y2)`	Trace une ligne entre les coordonnées (`x1`;`y1`) et (`x2`;`y2`).
`drawRect(x,y,l,h)`	Trace un rectangle de coin (`x`;`y`), de largeur `l` et de hauteur `h`.
`fillRect(x,y,l,h)`	La même chose avec une forme pleine.
`drawString(s,x,y)`	Dessine la chaîne de caractères `s` aux coordonnées (`x`;`y`).

Il est possible de spécifier la couleur des formes dessinées, pour cela il faut utiliser la méthode `setColor` du contexte graphique. Elle attend en paramètre une couleur qui est un objet de la classe `Color`. Des couleurs sont prédéfinies dans la classe, par exemple `Color.red`, `Color.green`... Mais il est possible de définir ses propres couleurs en indiquant leur code RGB (*Red / Green / Blue*), c'est-à-dire leur niveau de rouge, de vert et de bleu, ces valeurs étant des flottants entre 0 et 1. L'exemple suivant dessine un rectangle orange dans le contexte graphique du composant courant.

```
Graphics g = getGraphics();
g.setColor(new Color((float)1.0,(float)0.5,(float)0.0));
g.fillRect(50,50,100,100);
```

6. CONCEPTS AVANCES

Ce chapitre propose un rapide aperçu de notions plus avancées du langage. Il présente le concept d'interface très utilisé en Java, le mécanisme des exceptions similaire à celui du C++, la manipulation des *threads* qui est fondamentale dans la conception d'une interface graphique, et la notion d'applet qui permet d'embarquer un programme Java dans une page Web.

INTERFACE

Java ne permet pas l'héritage multiple. Il existe plusieurs raisons à ce choix des concepteurs du langage. Tout d'abord, un héritage multiple dans une modélisation est très souvent une erreur de conception et peut être remplacé par une agrégation ou une composition. Ensuite, l'héritage multiple peut conduire à des ambiguïtés. La principale raison étant qu'une classe qui dérive d'une autre hérite à la fois de son interface (i.e. le prototype des méthodes publiques de la classe), mais également de son implémentation (i.e. les propriétés privées et le corps des méthodes). C'est justement l'héritage de l'implémentation qui est ambigu. Par exemple, une classe `C` hérite des classes `A` et `B`, toutes les deux disposant d'une méthode `m`. Quelle implémentation de la méthode `m` est appelée pour un objet de la classe `C` ?

Pour éviter tout problème lié à l'héritage multiple, mais en fournissant tout de même un mécanisme similaire, Java propose la notion d'*interface* correspondant à notre définition du paragraphe précédent. Une interface est définie de la même manière qu'une classe en Java. La différence est seulement qu'une interface ne possède pas d'attributs et toutes ses méthodes sont abstraites (i.e. sans corps). Cela revient donc à peu de chose près à une classe abstraite pure en C++. Une classe ne peut hériter que d'une seule classe, mais elle peut dériver de plusieurs interfaces. L'héritage d'une interface signifie qu'on ajoute à une classe la possibilité de répondre à l'appel de méthodes supplémentaires. Mais il faut bien entendu implémenter ces méthodes, l'héritage d'interface ne fournissant aucune implémentation. Considérons maintenant l'exemple suivant.

```
class Forme {
 protected int x1;
 protected int y1;
 public Forme(int a,int b) { x1=a; y1=b; }
}

interface Affichable {
 public void afficher(java.awt.Graphics g);
}

class Ligne extends Forme implements Affichable {
 protected int x2;
 protected int y2;

 public Ligne(int a,int b,int c,int d)
 { super(a,b); x2=c; y2=d; }

 public void afficher(java.awt.Graphics g)
 { g.drawLine(x1,y1,x2,y2); }
}

class Rectangle extends Forme implements Affichable {
 protected int l;
 protected int h;

 public Rectangle(int a,int b,int c,int d)
 { super(a,b); l=c; h=d; }
```

```
 public void afficher(java.awt.Graphics g)
 { g.drawRect(x1,y1,l,h); }
}
```

Les classes `Ligne` et `Rectangle` héritent de la classe `Forme` et de l'interface `Affichable`. Elles profitent donc des fonctionnalités de `Forme` et implémentent l'interface `Affichable`, c'est-à-dire qu'elles proposent la méthode `afficher`. Notons que l'héritage d'une classe est spécifié par le mot-clé `extends` (la classe héritée est étendue), alors que l'héritage d'une ou plusieurs interfaces est spécifié par le mot-clé `implements` (l'interface héritée est implémentée). Le terme *implémenter* est préféré au terme *hériter* lorsqu'on parle d'une interface. De retour à notre exemple, il est ainsi possible de faire un tableau d'éléments "affichables" et de les traiter en masse.

```
Affichable t[] = { new Ligne(10,10,100,100),
                   new Rectangle(40,40,20,20) };
int i = 0;

while (i<t.length) t[i++].afficher(getGraphics());
```

GESTION DES EXCEPTIONS

La gestion des exceptions est très proche de celle du C++: lorsqu'une méthode est susceptible de lever une exception, elle est surveillée par un bloc `try` et l'instruction `catch` prépare la réception de l'erreur éventuelle. Les bibliothèques standards Java lèvent des exceptions appartenant toutes à la classe `Exception` (ou à l'une de ses sous-classes). L'exemple suivant prévoit la récupération de l'exception pouvant être levée par la création d'un objet de la classe `FileInputStream`.

```
java.io.FileInputStream f;
try { f=new java.io.FileInputStream("entree.txt"); }
catch (Exception e) { System.out.println(e.getMessage()); }
```

Un objet de la classe `Exception` possède la méthode `getMessage` qui retourne une chaîne de caractères décrivant l'erreur. Ainsi, pour proposer ses propres exceptions, il suffit de dériver une classe de la super-classe `Exception` et de surcharger la méthode `getMessage`.

```
class ExceptionDebordement extends Exception {
 public String getMessage()
 { return "Debordement de tableau !"; }
}
```

Pour jeter une exception lorsqu'on détecte un problème, on utilise le mot-clé `throw`.

```
public int getElement(int i) throws ExceptionDebordement {
 if (i>=tab.length) throw new ExceptionDebordement();
 return tab[i];
}
```

En Java, il est obligatoire d'indiquer qu'une méthode peut lever une exception, à l'aide du mot-clé `throws`. Il est également obligatoire lorsqu'on appelle une méthode susceptible de jeter une exception de récupérer cette exception par un bloc `try`, la compilation est impossible sinon. On peut décider ensuite de traiter l'exception ou de la transmettre à la méthode appelante. Il est également possible de jeter d'autres objets que ceux de la classe `Exception`, mais dans ce cas il faut que l'objet appartienne à une classe dérivant de `Throwable`.

MANIPULATION DES *THREADS*

Un *thread* est un processus léger dans un programme. Contrairement aux processus qui ne

partagent par défaut aucune donnée (par exemple la fonction `fork` en C dédouble un programme, i.e. son code et ses données, en deux processus indépendants), les *threads* d'un même programme partagent leurs données (par exemple, un même objet peut être manipulé dans des *threads* différents). Les *threads* en Java appartiennent à la classe `Thread`. Voici un exemple de *thread* qui prend en charge la mise à jour de l'affichage d'une horloge toutes les minutes, pendant que l'application effectue des opérations sur le *thread* principal.

```
class Horloge implements Runnable {
 protected Thread thread;
 protected boolean fin;
 public void setThread(Thread t) { thread=t; }
 public Thread getThread() { return thread; }
 public Horloge() { ... }
 public void afficher() { ... }
 public void stop() { fin=true; }

 public void run() {
  fin=false;

  while (!fin) {
   afficher();
   try { thread.sleep(1000); }
   catch(Exception e) {}
  }
 }
}
...
Horloge h = new Horloge();
h.setThread(new Thread(h));
h.getThread().start();
...
h.stop();
```

Lorsqu'un *thread* est créé, il faut lui attacher l'objet qui va s'exécuter dessus. Ce dernier doit implémenter l'interface `Runnable` pour disposer de la méthode `run`. Le *thread* démarre lorsque sa méthode `start` est appelée, celle-ci appelle alors la méthode `run` de l'objet attaché. Dans cette méthode, le développeur place le code qu'il souhaite exécuter sur le *thread*. Pour notre exemple, la classe `Horloge` implémente l'interface `Runnable` et la méthode `run` qui effectue en boucle un affichage de l'horloge suivi d'une pause d'une seconde. La méthode `sleep` d'un *thread* le stoppe pendant un certain nombre de millisecondes. Si la méthode `run` se termine, le *thread* s'arrête naturellement. Pour forcer l'arrêt d'un `thread`, il est possible d'appeler sa méthode `destroy`, mais il s'agit d'un arrêt brutal qui peut entraîner des complications. Il est plutôt conseillé de gérer l'arrêt du *thread* à l'aide d'une variable comme le montre l'exemple avec l'attribut `fin` et la méthode `stop`.

Un objet peut être exécuté en même temps sur des *threads* différents, i.e. en même temps plusieurs de ses méthodes peuvent être appelées. Cela peut conduire à de sérieux problèmes. Imaginons que deux *threads* appellent en même temps les méthodes `reserveTicket` et `numeroTicket` d'un même objet dans l'exemple suivant.

```
public int reserveTicket(String nom) {
 if (n>=N) return -1;
 ticket[n]=nom;
 return n++;
}

public int numeroTicket(String nom) {
 int i = 0;

 while (i<n && !ticket[i].equals(nom)) ++i;
 if (i==n) return -1;
 return i;
}
```

Pendant que l'un modifie des attributs de l'objet, le second les consulte. Ce dernier peut alors se

retrouver avec des informations invalides (i.e. des données anciennes mélangées à des données nouvelles). Pour empêcher l'accès multiple aux attributs d'un objet, il est possible d'utiliser le mot-clé `synchronized`. Devant une méthode, ce mot signifie que deux *threads* ne peuvent pas utiliser en même temps la méthode. Si plusieurs méthodes possèdent ce mot-clé, cela signifie que seulement l'une des méthodes peut être appelée à la fois (et par un seul *thread* en même temps). De telles méthodes sont dites *synchronisées*. Dans l'exemple suivant, `reserveTicket` et `numeroTicket` ne peuvent pas être appelées en même temps.

```
public synchronized int reserveTicket(String nom) { ... }
public synchronized int numeroTicket(String nom) { ... }
```

Pour une synchronisation encore plus évoluée, il existe les méthodes `wait()` et `notify()`. La méthode `wait()` suspend le *thread* courant, celui-ci rend alors les méthodes synchronisées accessibles à d'autres *threads*. Le *thread* suspendu attend ensuite qu'un autre le réveille. Cela ne peut être fait qu'en appelant la méthode `notify` de l'objet qui a suspendu le *thread*. Prenons l'exemple suivant.

```
public synchronized void prendreJeton() {
 while (n==0) {
  try { wait(); }
  catch (Exception e) {}
 }

 --n;
}

public synchronized void rendreJeton() {
 ++n;
 notify();
}
```

Un *thread* suspendu avec la méthode `wait` n'est réellement réactivé avec `notify` qu'une fois que le *thread* qui a appelé `notify` a libéré la méthode synchronisée dans laquelle il se trouve.

CREATION D'UNE APPLET

Pour exécuter un programme Java, il est nécessaire d'appeler une JVM, ce qui peut être fait avec la commande `java`. Mais il est possible de créer un type d'application embarquable dans du code HTML, la JVM est alors appelée implicitement par le navigateur. Ce genre d'application est appelée *applet*, ou *appliquette*, car elle est normalement plus légère qu'une application fonctionnant seule (dite *autonome* ou *standalone*). Il est néanmoins possible de faire à peu près les mêmes choses que dans une application Java classique, les restrictions majeures concernent la sécurité de l'ordinateur. En effet, une applet dans une page HTML est démarrée automatiquement lorsqu'un utilisateur visualise la page, il est évident qu'une applet ne peut alors pas effectuer n'importe quelle opération, comme consulter ou modifier des fichiers un peu partout sur le disque-dur. Voici comment embarquer une applet dans une page Web.

```
<APPLET CODE="MonApplet" CODEBASE=".">
 <PARAM NAME="couleur" VALUE="red">
 <PARAM NAME="taille" VALUE="25">
</APPLET>
```

Il suffit d'utiliser la balise `<APPLET>` dans laquelle les balises `<PARAM>` correspondent à des paramètres que l'on peut transmettre à l'applet. L'attribut `CODE` indique le nom de la classe de l'applet et `CODEBASE` l'endroit où trouver le fichier de cette classe. Les attributs `WIDTH` et `HEIGHT` peuvent aussi être utilisés pour spécifier les dimensions de l'applet. Dans l'exemple, les paramètres `couleur` et `taille` avec respectivement les valeurs `red` et `25` sont transmis à l'applet. Le code suivant montre comment récupérer ces paramètres et la manière dont s'organise une applet.

```
import java.applet.*;
import java.awt.*;

public class MonApplet extends Applet {
 protected String couleur;
 protected int taille;

 public void init() {
  couleur=getParameter("couleur");
  taille=Integer.parseInt(getParameter("taille"));
 }

 public void start() { ... }
 public void stop() { ... }
 public void paint(Graphics g) { ... }
}
```

Dans l'exemple HTML précédent, le navigateur crée automatiquement un objet de la classe `MonApplet`, qui doit appartenir à la classe `Applet` du package `java.applet`. Il n'est pas recommandé d'intervenir dans le constructeur de l'objet. A la place, la méthode `init` est proposée pour initialiser l'applet. Cette méthode est appelée par le navigateur quand il le juge nécessaire. Ensuite, une fois l'applet initialisée, la méthode `start` est exécutée, également par le navigateur, elle correspond à la méthode `main` d'une application classique. C'est dans cette méthode que le développeur démarre l'application. Enfin, une fois qu'elle ne sert plus, l'applet est stoppée, la méthode `stop` est alors appelée par le navigateur, avant la destruction de l'objet. Dans cette méthode peuvent être placées des instructions pour libérer des ressources comme des *threads*. La classe `Applet` hérite de `java.awt.Component`, elle possède donc la méthode `paint` qui est gérée de la même manière que n'importe quel autre composant graphique.