# Initiation à la programmation orientée-objet avec le langage Java

**Pôle Informatique 2013-2014**


158 cours Fauriel
42023 Saint-Étienne Cedex 02

***Gauthier Picard***

gauthier.picard@emse.fr
Institut Henri Fayol, ENSM.SE

***Laurent Vercouter***

laurent.vercouter@insa-rouen.fr
LITIS, INSA Rouen

# Table des matières


**1 Introduction au langage Java** **1**
- 1.1 Environnement Java . . . 1
  - 1.1.1 Compilation . . . 2
  - 1.1.2 Interprétation . . . 2
- 1.2 Programmation orientée-objet . . . 2
  - 1.2.1 Classe . . . 3
  - 1.2.2 Objet . . . 4

**2 Syntaxe du langage** **7**
- 2.1 Types de données . . . 7
  - 2.1.1 Types primitifs . . . 7
  - 2.1.2 Tableaux et matrices . . . 8
  - 2.1.3 Chaînes de caractères . . . 9
- 2.2 Opérateurs . . . 9
- 2.3 Structures de contrôle . . . 9
  - 2.3.1 Instructions conditionnelles . . . 9
  - 2.3.2 Instructions itératives . . . 10
  - 2.3.3 Instructions `break` et `continue` . . . 12

**3 Éléments de programmation Java** **13**
- 3.1 Premiers pas . . . 13
  - 3.1.1 Classe `HelloWorld` . . . 13
  - 3.1.2 Packages . . . 14
- 3.2 Variables et méthodes . . . 15
  - 3.2.1 Visibilité des champs . . . 15
  - 3.2.2 Variables et méthodes de classe . . . 16

**4 Héritage** **19**
- 4.1 Principe de l'héritage . . . 19
  - 4.1.1 Redéfinition . . . 20
  - 4.1.2 Polymorphisme . . . 21
- 4.2 Interfaces . . . 23
- 4.3 Classes abstraites . . . 24
- 4.4 Classes et méthodes génériques . . . 25

**5 Gestion des exceptions** **27**
- 5.1 Déclaration . . . 27
- 5.2 Interception et traitement . . . 29
- 5.3 Classes d'exception . . . 30
- 5.4 Classification des erreurs en Java . . . 30

**6 Gestion des entrées/sorties simples** **33**
6.1 Flux d'entrée . . . 33
6.1.1 Lecture des entrées clavier . . . 34
6.1.2 Lecture à partir d'un fichier . . . 34
6.1.3 Lecture d'objets enregistrés . . . 36
6.2 Flux de sortie . . . 37
6.2.1 Ecriture sur la sortie standard "écran" . . . 37
6.2.2 Ecriture dans un fichier . . . 37
6.2.3 Ecriture d'objets . . . 39

**A Applications graphiques (package `swing`)** **43**
A.1 Le schéma Modèle-Vue-Contrôleur . . . 43
A.2 Composants graphiques . . . 44
A.2.1 Composants . . . 44
A.2.2 Containers . . . 45
A.2.3 Exemple d'interface . . . 47
A.3 Contrôleurs d'événements . . . 51
A.3.1 Evénements . . . 51
A.3.2 Interface Listener . . . 51
A.3.3 Exemple de contrôleur . . . 51

**B Diagramme de classes UML** **55**
B.1 Représentation des classes et interfaces . . . 55
B.1.1 Les classes . . . 55
B.1.2 Les membres de classe . . . 55
B.1.3 Les classes abstraites . . . 56
B.1.4 Les interfaces . . . 56
B.2 Les relations . . . 57
B.2.1 L'héritage . . . 57
B.2.2 La réalisation . . . 57
B.3 Les associations . . . 57
B.3.1 Direction des associations . . . 58
B.3.2 Agrégation et composition . . . 59
B.4 Correspondance UML-Java . . . 59
B.4.1 Classes et membres . . . 59
B.4.2 Classes abstraites . . . 59
B.4.3 Interfaces . . . 60
B.4.4 Héritage . . . 60
B.4.5 Réalisation . . . 60
B.4.6 Associations . . . 60

La plupart des programmes donnés en exemple dans ce cours sont téléchargeables à l'URL :
http://www.emse.fr/~picard/cours/1A/java

# Chapitre 1

# Introduction au langage Java

Le langage Java est un langage généraliste de programmation synthétisant les principaux langages existants lors de sa création en 1995 par *Sun Microsystems*. Il permet une programmation orientée-objet (à l'instar de SmallTalk et, dans une moindre mesure, C++), modulaire (langage ADA) et reprend une syntaxe très proche de celle du langage C.

Outre son orientation objet, le langage Java a l'avantage d'être **modulaire** (on peut écrire des portions de code génériques, c-à-d utilisables par plusieurs applications), **rigoureux** (la plupart des erreurs se produisent à la compilation et non à l'exécution) et **portable** (un même programme compilé peut s'exécuter sur différents environnements). En contre-partie, les applications Java ont le défaut d'être plus lentes à l'exécution que des applications programmées en C par exemple.

## 1.1 Environnement Java

Java est un langage interprété, ce qui signifie qu'un programme compilé n'est pas directement exécutable par le système d'exploitation mais il doit être interprété par un autre programme, qu'on appelle interpréteur. La figure 1.1 illustre ce fonctionnement.

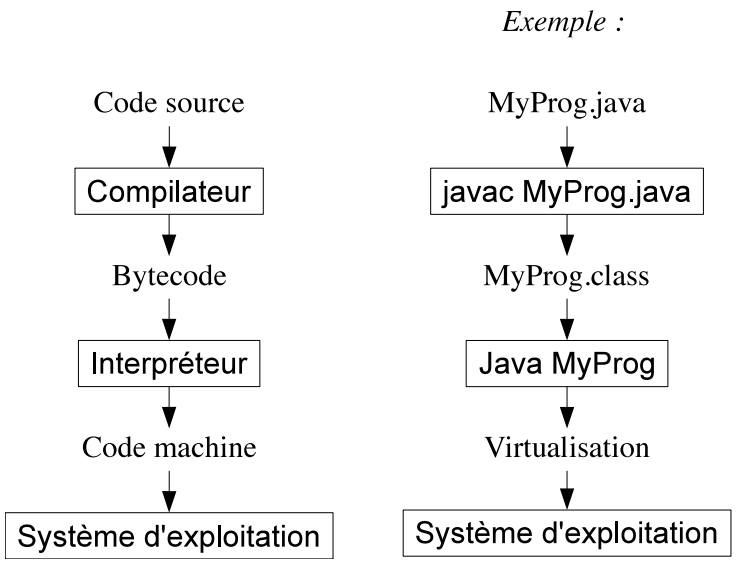



FIGURE 1.1 – Interprétation du langage

Un programmeur Java écrit son code source, sous la forme de classes, dans des fichiers dont l'extension est `.java`. Ce code source est alors compilé par le compilateur `javac` en un langage appelé *bytecode* et enregistre le résultat dans un fichier dont l'extension est `.class`. Le bytecode

ainsi obtenu n'est pas directement utilisable. Il doit être interprété par la *machine virtuelle* de Java qui transforme alors le code compilé en code machine compréhensible par le système d'exploitation. C'est la raison pour laquelle Java est un langage portable : le bytecode reste le même quelque soit l'environnement d'exécution.

En 2009, Sun Microsystems est racheté par Oracle Corporation qui fournit dorénavant les outils de développement Java SE (*Standard Edition*) contenus dans le *Java Development Kit* (JDK). Au moment où est écrit ce livret, la dernière version stable est le JDK 1.7.

### 1.1.1 Compilation

La compilation s'effectue par la commande `javac` suivie d'un ou plusieurs nom de fichiers contenant le code source de classes Java. Par exemple, `javac MyProg.java` compile la classe `MyProg` dont le code source est situé dans le fichier `MyProg.java`. La compilation nécessite souvent la précision de certains paramètres pour s'effectuer correctement, notamment lorsque le code source fait référence à certaines classes situées dans d'autres répertoires que celui du code compilé. Il faut alors ajouter l'option `-classpath` suivie des répertoires (séparés par un ; sous Windows et : sous Unix) des classes référencées. Par exemple :

```
javac -classpath /prog/exos1:/cours MyProg.java
```

compilera le fichier `MyProg.java` si celui-ci fait référence à d'autres classes situées dans les répertoires `/prog/exos1` et `/cours`. Le résultat de cette compilation est un fichier nommé `MyProg.class` contenant le bytecode correspondant au source compilé. Ce fichier est créé par défaut dans le répertoire où la compilation s'est produite. Il est cependant fortement souhaitable de ne pas mélanger les fichiers contenant le code source et ceux contenant le bytecode. Un répertoire de destination où sera créé le fichier `MyProg.class` peut être précisé par l'option *-d*, par exemple :

```
javac -d /prog/exos1 -classpath /cours MyProg.java
```

### 1.1.2 Interprétation

Le bytecode obtenu par compilation ne peut être exécuté qu'à l'aide de l'interpréteur. L'exécution s'effectue par la commande `java` suivie du nom de la classe à exécuter (sans l'extension `.class`). Comme lors de la compilation, il se peut que des classes d'autres répertoires soient nécessaires. Il faut alors utiliser l'option `-classpath` comme dans l'exemple qui suit :

```
java -classpath /prog/exos1:/cours MyProg
```

## 1.2 Programmation orientée-objet

Chaque langage de programmation appartient à une "famille" de langages définissant une approche ou une méthodologie générale de programmation. Par exemple, le langage C est un langage de programmation *procédurale* car il suppose que le programmeur s'intéresse en priorité aux *traitements* que son programme devra effectuer. Un programmeur C commencera par identifier ces traitements pour écrire les fonctions qui les réalisent sur des données prises comme paramètres d'entrée.

La programmation *orientée-objet* (introduite par le langage SmallTalk) propose une méthodologie centrée sur les données. Le programmeur Java va d'abord identifier un ensemble d'**objets**, tel que chaque objet représente un élément qui doit être utilisé ou manipulé par le programme, sous la forme d'ensembles de données. Ce n'est que dans un deuxième temps, que le programmeur va écrire les traitements, **en associant chaque traitement à un objet donné**. Un objet peut

être vu comme une entité regroupant un ensemble de données et de **méthodes** (l'équivalent d'une fonction en C) de traitement.

### 1.2.1 Classe

Un objet est une variable (presque) comme les autres. Il faut notamment qu'il soit déclaré avec son type. Le type d'un objet est un type complexe (par opposition aux types primitifs entier, caractère, …) qu'on appelle une **classe**.

Une classe regroupe un ensemble de données (qui peuvent être des variables primitives ou des objets) et un ensemble de méthodes de traitement de ces données et/ou de données extérieures à la classe. On parle d'**encapsulation** pour désigner le regroupement de données dans une classe.

Par exemple, une classe *Rectangle* utilisée pour instancier des objets représentant des rectangles, encapsule 4 entiers : la longueur et la largeur du rectangle ainsi que la position en abscisse et en ordonnée de l'origine du rectangle (par exemple, le coin en haut à gauche). On peut alors imaginer que la classe *Rectangle* implémente une méthode permettant de déplacer le rectangle qui nécessite en entrée deux entiers indiquant la distance de déplacement en abscisse et en ordonnée. L'accès aux positions de l'origine du rectangle se fait directement (i.e. sans passage de paramètre) lorsque les données sont encapsulées dans la classe où est définie la méthode. Un exemple, écrit en Java, de la classe *Rectangle* est donné ci-dessous :

```
class Rectangle {

    int longueur;
    int largeur;
    int origine_x;
    int origine_y;

    void deplace(int x, int y) {
        this.origine_x = this.origine_x + x;
        this.origine_y = this.origine_y + y;
    }

    int surface() {
        return this.longueur * this.largeur;
    }
}
```

Pour écrire un programme avec un langage orienté-objet, le programmeur écrit uniquement des classes correspondant aux objets de son système. Les traitements à effectuer sont programmés dans les méthodes de ces classes qui peuvent faire appel à des méthodes d'autres classes. En général, on définit une classe, dite "exécutable", dont une méthode peut être appelée pour exécuter le programme.

#### Encapsulation

Lors de la conception d'un programme orienté-objet, le programmeur doit identifier les objets et les données appartenant à chaque objet mais aussi des droits d'accès qu'ont les autres objets sur ces données. L'encapsulation de données dans un objet permet de cacher ou non leur existence aux autres objets du programme. Une donnée peut être déclarée en accès :

- **public** : les autres objets peuvent accéder à la valeur de cette donnée ainsi que la modifier ;

- **privé** : les autres objets n'ont pas le droit d'accéder directement à la valeur de cette donnée (ni de la modifier). En revanche, ils peuvent le faire indirectement par des méthodes de l'objet concerné (si celles-ci existent en accès public).

Les différents droits d'accès utilisables en Java sont détaillés dans la section 3.2.1.

#### Méthode constructeur

Chaque classe doit définir une ou plusieurs méthodes particulières appelées des **constructeurs**. Un **constructeur** est une méthode invoquée lors de la création d'un objet. Cette méthode, qui peut être vide, effectue les opérations nécessaires à l'initialisation d'un objet. Chaque constructeur doit avoir le même nom que la classe où il est défini et n'a aucune valeur de retour (c'est l'objet créé qui est renvoyé). Dans l'exemple précédent de la classe rectangle, le constructeur initialise la valeur des données encapsulées :

```
class Rectangle {
    ...
    Rectangle(int lon, int lar) {
        this.longueur = lon;
        this.largeur = lar;
        this.origine_x = 0;
        this.origine_y = 0;
    }
    ...
}
```

Plusieurs constructeurs peuvent être définis s'ils acceptent des paramètres d'entrée différents.

### 1.2.2 Objet

#### Instanciation

Un objet est une instance (anglicisme signifiant « cas » ou « exemple ») d'une classe et est référencé par une variable ayant un état (ou valeur). Pour créer un objet, il est nécessaire de déclarer une variable dont le type est la classe à instancier, puis de faire appel à un constructeur de cette classe. L'exemple ci-dessous illustre la création d'un objet de classe `Cercle` en Java :

```
Cercle mon_rond;
mon_rond = new Cercle();
```

L'usage de parenthèses à l'initialisation du vecteur, montre qu'une méthode est appelée pour l'instanciation. Cette méthode est un constructeur de la classe `Cercle`. Si le constructeur appelé nécessite des paramètres d'entrée, ceux-ci doivent être précisés entre ces parenthèses (comme lors d'un appel classique de méthode). L'instanciation d'un objet de la classe `Rectangle` faisant appel au constructeur donné en exemple ci-dessous pourra s'écrire :

```
Rectangle mon_rectangle = new Rectangle(15,5);
```

☛ **Remarque importante :** en Java, la notion de pointeur est transparente pour le programmeur. Il faut néanmoins savoir que **toute variable désignant un objet est un pointeur**. Il s'ensuit alors que le passage d'objets comme paramètres d'une méthode est **toujours** un passage par référence. A l'inverse, le passage de variables primitives comme paramètres est toujours un passage par valeur.

### Accès aux variables et aux méthodes

Pour accéder à une variable associée à un objet, il faut préciser l'objet qui la contient. Le symbole '`.`' sert à séparer l'identificateur de l'objet de l'identificateur de la variable. Une copie de la longueur d'un rectangle dans un entier `temp` s'écrit :

```
int temp = mon_rectangle.longueur;
```

La même syntaxe est utilisée pour appeler une méthode d'un objet. Par exemple :

```
mon_rectangle.deplace(10,-3);
```

Pour qu'un tel appel soit possible, il faut que trois conditions soient remplies :

1. La variable ou la méthode appelée existe !
2. Une variable désignant l'objet visé existe et soit instanciée.
3. L'objet, au sein duquel est fait cet appel, ait le droit d'accéder à la méthode ou à la variable (cf. section 3.2.1).

Pour référencer l'objet "courant" (celui dans lequel se situe la ligne de code), le langage Java fournit le mot-clé `this`. Celui-ci n'a pas besoin d'être instancié et s'utilise comme une variable désignant l'objet courant. Le mot-clé `this` est également utilisé pour faire appel à un constructeur de l'objet courant. Ces deux utilisations possibles de `this` sont illustrées dans l'exemple suivant :

```
class Carre {

    int cote;
    int origine_x;
    int origine_y;

    Carre(int cote, int x, int y) {
        this.cote = cote;
        this.origine_x = x;
        this.origine_y = y;
    }

    Carre(int cote) {
        this(cote, 0, 0);
    }
}
```

# Chapitre 2

# Syntaxe du langage

Le langage C a servi de base pour la syntaxe du langage Java :

- le caractère de fin d'une instruction est " `;` "

```
a = c + c;
```

- les commentaires (non traités par le compilateur) se situent entre les symboles "`/*`" et "`*/`" ou commencent par le symbole "`//`" en se terminant à la fin de la ligne

```
int a; // ce commentaire tient sur une ligne
int b;
```

ou

```
/*Ce commentaire nécessite
 2 lignes*/
int a;
```

- les identificateurs de variables ou de méthodes acceptent les caractères {a..z}, {A..Z}, `$`, _ ainsi que les caractères {0..9} s'ils ne sont pas le premier caractère de l'identificateur. Il faut évidemment que l'identificateur ne soit pas un mot réservé du langage (comme `int` ou `for`).
  Ex : `mon_entier` et `ok4all` sont des identificateurs valides mais
  `mon-entier` et `4all` ne sont pas valides pour des identificateurs.

## 2.1 Types de données

### 2.1.1 Types primitifs

Le tableau 2.1 liste l'ensemble des types primitifs de données de Java.

En plus de ces types primitifs, le terme `void` est utilisé pour spécifier le retour vide ou une absence de paramètres d'une méthode. On peut remarquer que chaque type primitif possède une classe qui encapsule un attribut du type primitif. Par exemple, la classe `Integer` encapsule un attribut de type `int` et permet ainsi d'effectuer des opérations de traitement et des manipulations qui seraient impossibles sur une simple variable de type `int`.

A l'inverse du langage C, Java est un langage très rigoureux sur le typage des données. Il est interdit d'affecter à une variable la valeur d'une variable d'un type différent [1] si cette seconde variable n'est pas explicitement transformée. Par exemple :

1. exception faite des types associés par une relation d'héritage, cf. section 4

TABLE 2.1 – Type primitifs de données en Java

| Type | Classe éq. | Valeurs | Portée | Défaut |
|---|---|---|---|---|
| `boolean` | `Boolean` | `true` ou `false` | N/A | `false` |
| `byte` | `Byte` | entier signé | {-128..128} | o |
| `char` | `Character` | caractère | {\uoooo..\uFFFF} | \uoooo |
| `short` | `Short` | entier signé | {-32768..32767} | 0 |
| `int` | `Integer` | entier signé | {-2147483648..2147483647} | 0 |
| `long` | `Long` | entier signé | $\{-2^{31}..2^{31}-1\}$ | 0 |
| `float` | `Float` | réel signé | $\{-3,4028234^{38}..3,4028234^{38}\}$<br>$\{-1,40239846^{-45}..1,40239846^{-45}\}$ | 0.0 |
| `double` | `Double` | réel signé | $\{-1,797693134^{308}..1,797693134^{308}\}$<br>$\{-4,94065645^{-324}..4,94065645^{-324}\}$ | 0.0 |

```
int a;
double b = 5.0;
a = b;
```

est interdit et doit être écrit de la manière suivante :

```
int a ;
double b = 5.0 ;
a = (int)b ;
```

### 2.1.2 Tableaux et matrices

Une variable est déclarée comme un tableau dès lors que des crochets sont présents soit après son type, soit après son identificateur. Les deux syntaxes suivantes sont acceptées pour déclarer un tableau d'entiers (même si la première, non autorisée en C, est plus intuitive) :

```
int[] mon_tableau;
int mon_tableau2[];
```

Un tableau a toujours une taille fixe [2] qui doit être précisée avant l'affectation de valeurs à ses indices, de la manière suivante :

```
int[] mon_tableau = new int[20];
```

De plus, la taille de ce tableau est disponible dans une variable `length` appartenant au tableau et accessible par `mon_tableau.length`. On peut également créer des matrices ou des tableaux à plusieurs dimensions en multipliant les crochets (ex : `int[][] ma_matrice;`). À l'instar du C, on accède aux éléments d'un tableau en précisant un indice entre crochets (`mon_tableau[3]` est le quatrième entier du tableau) et un tableau de taille `n` stocke ses éléments à des indices allant de `0` à `n-1`.

2. pour utiliser des ensembles à taille variable, la classe `java.util.Vector` est très utile

### 2.1.3 Chaînes de caractères

Les chaînes de caractères ne sont pas considérées en Java comme un type primitif ou comme un tableau. On utilise une classe particulière, nommée `String`, fournie dans le package `java.lang`. Les variables de type String ont les caractéristiques suivantes :

- leur valeur ne peut pas être modifiée
- on peut utiliser l'opérateur + pour concaténer deux chaînes de caractères :

```
String s1 = "hello";
String s2 = "world";
String s3 = s1 + " " + s2;
//Après ces instructions s3 vaut "hello world"
```

- l'initialisation d'une chaîne de caractères s'écrit :

```
String s = new String(); //pour une chaine vide
String s2 = new String("hello world");
// pour une chaîne de valeur "hello world"
```

- un ensemble de méthodes de la classe `java.lang.String` permettent d'effectuer des opérations ou des tests sur une chaîne de caractères (voir la documentaion de la classe `String`).

## 2.2 Opérateurs

Une liste des opérateurs disponibles en Java est présentée par ordre de priorité décroissante dans le tableau 2.2.

## 2.3 Structures de contrôle

Les structures de contrôle permettent d'exécuter un **bloc d'instructions** soit plusieurs fois (instructions itératives) soit selon la valeur d'une expression (instructions conditionnelles ou de choix multiple). Dans tous ces cas, un bloc d'instruction est

- soit une instruction unique ;
- soit une suite d'instructions commençant par une accolade ouvrante "`{`" et se terminant par une accolade fermante "`}`".

### 2.3.1 Instructions conditionnelles

Syntaxe :

`if (<condition>) <bloc1> [else <bloc2>]`

ou

`<condition>?<instruction1>:<instruction2>`

`<condition>` doit renvoyer une valeur booléenne. Si celle-ci est vraie c'est `<bloc1>` (resp. `<instruction1>`) qui est exécuté sinon `<bloc2>` (resp. `<instruction2>`) est exécuté. La partie `else <bloc2>` est facultative.

Exemple :

TABLE 2.2 – Opérateurs Java

| Pr. | Opérateur | Syntaxe | Résultat | Signification |
|---|---|---|---|---|
| 1 | `++` | `++<ari>` | `<ari>` | pré incrémentation |
| | | `<ari>++` | `<ari>` | post incrémentation |
| | `--` | `--<ari>` | `<ari>` | pré décrémentation |
| | | `<ari>--` | `<ari>` | post décrémentation |
| | `+` | `+<ari>` | `<ari>` | signe positif |
| | `-` | `-<ari>` | `<ari>` | signe négatif |
| | `!` | `!<boo>` | `<boo>` | complément logique |
| | `(type)` | `(type)<val>` | `<val>` | changement de type |
| 2 | `*` | `<ari>*<ari>` | `<ari>` | multiplication |
| | `/` | `<ari>/<ari>` | `<ari>` | division |
| | `%` | `<ari>%<ari>` | `<ari>` | reste de la division |
| 3 | `+` | `<ari>+<ari>` | `<ari>` | addition |
| | `-` | `<ari>-<ari>` | `<ari>` | soustraction |
| | `+` | `<str>+<str>` | `<str>` | concaténation |
| 4 | `<<` | `<ent> << <ent>` | `<ent>` | décalage de bits à gauche |
| | `>>` | `<ent> >> <ent>` | `<ent>` | décalage de bits à droite |
| 5 | `<` | `<ari> < <ari>` | `<boo>` | inférieur à |
| | `<=` | `<ari> <= <ari>` | `<boo>` | inférieur ou égal à |
| | `>` | `<ari> > <ari>` | `<boo>` | supérieur à |
| | `>=` | `<ari> >= <ari>` | `<boo>` | supérieur ou égal à |
| | `instanceof` | `<val>instanceof<cla>` | `<boo>` | test de type |
| 6 | `==` | `<val>==<val>` | `<boo>` | égal à |
| | `!=` | `<val>!=<val>` | `<boo>` | différent de |
| 7 | `&` | `<ent>&<ent>` | `<ent>` | ET bit à bit |
| | | `<boo>&<boo>` | `<boo>` | ET booléen |
| 8 | `^` | `<ent>^<ent>` | `<ent>` | OU exclusif bit à bit |
| | | `<boo>^<boo>` | `<boo>` | OU exclusif booléen |
| 9 | `\|` | `<ent>\|<ent>` | `<ent>` | OU bit à bit |
| | | `<boo>\|<boo>` | `<boo>` | OU booléen |
| 10 | `&&` | `<boo>&&<boo>` | `<boo>` | ET logique |
| 11 | `\|\|` | `<boo>\|\|<boo>` | `<boo>` | OU logique |
| 12 | `?:` | `<boo>?<ins>:<ins>` | `<ins>` | si...alors...sinon |
| 13 | `=` | `<var>=<val>` | `<val>` | assignation |

**Légende**

`<ari>` valeur arithmétique
`<boo>` valeur booléenne
`<cla>` classe
`<ent>` valeur entière
`<ins>` instruction
`<str>` chaîne de caractères (`String`)
`<val>` valeur quelconque
`<var>` variable

```java
if (a == b) {
    a = 50;
    b = 0;
} else {
    a = a - 1;
}
```

### 2.3.2 Instructions itératives

Les instruction itératives permettent d'exécuter plusieurs fois un bloc d'instructions, et ce, jusqu'à ce qu'une condition donnée soit fausse. Les trois types d'instruction itératives sont les suivantes :

**TantQue...Faire...** L'exécution de cette instruction suit les étapes suivantes :

1. la condition (qui doit renvoyer une valeur booléenne) est évaluée. Si celle-ci est vraie on passe à l'étape 2, sinon on passe à l'étape 4 ;
2. le bloc est exécuté ;

3. retour à l'étape 1 ;
4. la boucle est terminée et le programme continue son exécution en interprétant les instruction suivant le bloc.

Syntaxe :
```
while (<condition>) <bloc>
```

Exemple :

```
while (a != b) a++;
```

**Faire...TantQue...** L'exécution de cette instruction suit les étapes suivantes :

1. le bloc est exécuté ;
2. la condition (qui doit renvoyer une valeur booléenne) est évaluée. Si celle-ci est vraie on retourne à l'étape 1, sinon on passe à l'étape 3 ;
3. la boucle est terminée et le programme continue son exécution en interprétant les instruction suivant le bloc.

Syntaxe :
```
do <bloc> while (<condition>);
```

Exemple :

```
do a++
while (a != b);
```

**Pour...Faire** Cette boucle est constituée de trois parties : (i) une initialisation (la déclaration de variables locales à la boucle est autorisée dans cette partie) ; (ii) une condition d'arrêt ; (iii) un ensemble d'instructions à exécuter après chaque itération (chacune de ces instructions est séparée par une virgule). L'exécution de cette instruction suit les étapes suivantes :

1. les initialisations sont effectuées ;
2. la condition (qui doit renvoyer une valeur booléenne) est évaluée. Si celle-ci est vraie on passe à l'étape 2, sinon on passe à l'étape 6 ;
3. le bloc principal est exécuté ;
4. les instructions à exécuter après chaque itération sont exécutées ;
5. retour à l'étape 2 ;
6. la boucle est terminée et le programme continue son exécution en interprétant les instructions suivant le bloc principal.

Syntaxe :
```
for (<init>;<condition>;<instr_post_itération>) <bloc>
```

Exemple :

```
for (int i = 0, j = 49; (i < 25) && (j >= 25); i++, j--)  {
   if (tab[i] > tab[j]) {
       int tampon = tab[j];
```

➲ **Question** : que se passe-t-il si le `break` situé après le `case 'c'` est omis ?

### 2.3.3 Instructions `break` et `continue`

L'instruction `break` est utilisée pour sortir immédiatement d'un bloc d'instructions (sans traiter les instructions restantes dans ce bloc). Dans le cas d'une boucle on peut également utiliser l'instruction `continue` avec la différence suivante :

**`break` :** l'exécution se poursuit après la boucle (comme si la condition d'arrêt devenait vraie) ;

**`continue` :** l'exécution du bloc est arrêtée mais pas celle de la boucle. Une nouvelle itération du bloc commence si la condition d'arrêt est toujours vraie.

Exemple :

```
for (int i = 0, j = 0; i < 100 ; i++) {
   if (i > tab.length) {
      break;
   }
   if (tab[i] == null) {
      continue;
   }
   tab2[j] = tab[i];
   j++;
}
```

# Chapitre 3

# Éléments de programmation Java

## 3.1 Premiers pas

Un programme écrit en Java consiste en un ensemble de classes représentant les éléments manipulés dans le programme et les traitements associés. L'exécution du programme commence par l'exécution d'une classe qui doit implémenter une méthode particulière "`public static void main(String[] args)`". Les classes implémentant cette méthode sont appellées classes *exécutables*.

### 3.1.1 Classe `HelloWorld`

Une classe Java `HelloWorld` qui affiche la chaîne de caractères "`Hello world`" s'écrit :

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

L'exécution (après compilation) de cette classe se fait de la manière suivante :

```
C:\>java HelloWorld
Hello world
C:\>
```

☛ **Remarque :** le tableau de chaînes de caractères `args` qui est un paramètre d'entrée de la méthode `main` contient des valeurs précisées à l'exécution. Si la classe avait été exécutée par la ligne de commande "`java HelloWorld 4 toto _`", ce tableau contiendrait 3 éléments dont les valeurs seraient respectivement "4", "`toto`" et "`_`".

Dans ce premier programme très simple, une seule classe est utilisée. Cependant, la conception d'un programme orienté-objet nécessite, pour des problèmes plus complexes, de créer plusieurs classes et la classe exécutable ne sert souvent qu'à instancier les premiers objets. La classe exécutable suivante crée un objet en instanciant la classe `Rectangle` (cf. section 1.2.1) et affiche sa surface :

```
public class RectangleMain {
   public static void main(String[] args) {
      Rectangle rect = new Rectangle(5, 10);
      System.out.println(”La surface est ” + rect.surface());
   }
}
```

☛ **Remarque importante :** il est obligatoire d'implémenter ***chaque classe publique dans un fichier séparé*** et il est indispensable que ***ce fichier ait le même nom que celui de la classe***. Dans le cas précédent, deux fichiers ont ainsi été créés : `RectangleMain.java` et `Rectangle.java`.

### 3.1.2 Packages

Un grand nombre de classes, fournies par Java SE, implémentent des données et traitements génériques utilisables par un grand nombre d'applications. Ces classes forment l'API (*Application Programmer Interface)* du langage Java. Une documentation en ligne pour l'API java est disponible à l'URL :

http://docs.oracle.com/javase/7/docs/api/

Toutes ces classes sont organisées en *packages* (ou bibliothèques) dédiés à un thème précis. Parmi les packages les plus utilisés, on peut citer les suivants :

| Package | Description |
|---|---|
| java.awt | Classes graphiques et de gestion d'interfaces |
| java.io | Gestion des entrées/sorties |
| java.lang | Classes de base (importé par défaut) |
| java.util | Classes utilitaires |
| javax.swing | Autres classes graphiques |

Pour accéder à une classe d'un package donné, il faut préalablement importer cette classe ou son package. Par exemple, la classe `Date` appartenant au package `java.util` qui implémente un ensemble de méthodes de traitement sur une date peut être importée de deux manières :

- une seule classe du package est importée :

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

- toutes les classes du package sont importées (même les classes non utilisées) :

```
import java.util.*;
```

Le programme suivant utilise cette classe pour afficher la date actuelle :

```
import java.util.Date;

public class DateMain {
   public static void main(String[] args) {
      Date today = new Date();
      System.out.println(”Nous sommes le ” + today.toString());
   }
}
```

Il est possible de créer vos propres packages en précisant, avant la déclaration d'une classe, le package auquel elle appartient. Pour assigner la classe précédente à un package, nommé `fr.emse`, il faut modifier le fichier de cette classe comme suit :

```
package fr.emse;

import java.util.Date;

public class DateMain {
    ...
}
```

Enfin, il faut que le chemin d'accès du fichier `DateMain.java` corresponde au nom de son package. Celui-ci doit donc être situé dans un répertoire `fr/emse/DateMain.java` accessible à partir des chemins d'accès définis lors de la compilation ou de l'exécution (voir section 1.1.1).

## 3.2 Variables et méthodes

### 3.2.1 Visibilité des champs

Dans les exemples précédents, le mot-clé `public` apparaît parfois au début d'une déclaration de classe ou de méthode sans qu'il ait été expliqué jusqu'ici. Ce mot-clé autorise n'importe quel objet à utiliser la classe ou la méthode déclarée comme publique. La portée de cette autorisation dépend de l'élément à laquelle elle s'applique (voir le tableau 3.1).

Table 3.1 – Portée des autorisations

| **Élément** | **Autorisations** |
|---|---|
| Variable | Lecture et écriture |
| Méthode | Appel de la méthode |
| Classe | Instanciation d'objets de cette classe et accès aux variables et méthodes de classe |

Le mode `public` n'est, bien sûr, pas le seul type d'accès disponible en Java. Deux autres mot-clés peuvent être utilisés en plus du type d'accès par défaut : `protected` et `private`. Le tableau 3.2 récapitule ces différents types d'accès (la notion de sous-classe est expliquée dans la section 4).

Table 3.2 – Autorisations d'accès

| | `public` | `protected` | défaut | `private` |
|---|---|---|---|---|
| **Dans la même classe** | Oui | Oui | Oui | Oui |
| **Dans une classe du même package** | Oui | Oui | Oui | Non |
| **Dans une sous-classe d'un autre package** | Oui | Oui | Non | Non |
| **Dans une classe quelconque d'un autre package** | Oui | Non | Non | Non |

Si aucun mot-clé ne précise le type d'accès, celui par défaut est appliqué. En général, il est souhaitable que les types d'accès soient limités et le type d'accès `public`, qui est utilisé systématiquement par les programmeurs débutants, ne doit être utilisé que s'il est indispensable. Cette restriction permet d'éviter des erreurs lors d'accès à des méthodes ou de modifications de variables sans connaître totalement leur rôle.

### 3.2.2 Variables et méthodes de classe

Dans certains cas, il est plus judicieux d'attacher une variable ou une méthode à une classe plutôt qu'aux objets instanciant cette classe. Par exemple, la classe `java.lang.Integer` possède une variable `MAX_VALUE` qui représente la plus grande valeur qui peut être affectée à un entier. Or, cette variable étant commune à tous les entiers, elle n'est pas dupliquée dans tous les objets instanciant la classe `Integer` mais elle est associée directement à la classe `Integer`. Une telle variable est appelée **variable de classe**. De la même manière, il existe des **méthodes de classe** qui sont associées directement à une classe. Pour déclarer une variable ou méthode de classe, on utilise le mot-clé `static` qui doit être précisé avant le type de la variable ou le type de retour de la méthode.

La classe `java.lang.Math` nous fournit un bon exemple de variable et de méthodes de classes.

```
public final class Math {
   ...
   public static final double PI = 3.14159265358979323846;
   ...
   public static double toRadians(double angdeg) {
      return angdeg / 180.0 * PI;
   }
   ...
}
```

La classe `Math` fournit un ensemble d'outils (variables et méthodes) très utiles pour des programmes devant effectuer des opération mathématiques complexes. Dans la portion de classe reproduite ci-dessus, on peut notamment y trouver une approximation de la valeur de π et une méthode convertissant la mesure d'un angle d'une valeur en degrés en une valeur en radians. Dans le cas de cette classe, il est tout a fait inutile de créer et d'instancier un objet à partir de la classe `Math`. En effet, la valeur de π ou la conversion de degrés en radians ne vont pas varier suivant l'objet auquel elles sont rattachées. Ce sont des variables et des méthodes de classe qui peuvent être invoquées à partir de toute autre classe (car elles sont déclarées en accès public) de la manière suivante :

```
public class MathMain {
   public static void main(String[] args) {
      System.out.println("pi = " + Math.PI);
      System.out.println("90° = " + Math.toRadians(90));
   }
}
```

➲ **Question :** Dans les sections précédentes, nous avons déjà utilisé une variable de classe et une méthode de classe. Pouvez-vous trouver lesquelles ?

➡ **Réponse :**

- la méthode `main` des classes exécutables est une méthode de classe car elle est appelée directement à partir d'une classe ;
- lors de l'affichage d'une chaîne de caractères à l'écran par l'instruction `System.out.println(…)`, on fait appel à la variable `out` de la classe `java.lang.System` qui est un objet représentant la sortie standard (l'écran) et sur laquelle on appelle la méthode `println` permettant d'afficher une chaîne de caractères.

# Chapitre 4

# Héritage

Dans certaines applications, les classes utilisées ont en commun certaines variables, méthodes de traitement ou même des signatures de méthode. Avec un langage de programmation orienté-objet, on peut définir une classe à différent niveaux d'abstraction permettant ainsi de factoriser certains attributs communs à plusieurs classes. Une classe générale définit alors un ensemble d'attributs qui sont partagés par d'autres classes, dont on dira qu'elles *héritent* de cette classe générale.

Par exemple, les classes `Carre` et `Rectangle` peuvent partager une méthode `surface()` renvoyant le résultat du calcul de la surface de la figure. Plutôt que d'écrire deux fois cette méthode, on peut définir une relation d'héritage entre les classes `Carre` et `Rectangle`. Dans ce cas, seule la classe `Rectangle` contient le code de la méthode `surface()` mais celle-ci est également utilisable sur les objets de la classe `Carre` si elle hérite de `Rectangle`.

## 4.1 Principe de l'héritage

L'idée principale de l'héritage est d'organiser les classes de manière hiérarchique. La relation d'héritage est unidirectionnelle et, si une classe B hérite d'une classe A, on dira que B est une sous-classe de A. Cette notion de sous-classe signifie que la classe B est un cas particulier de la classe A et donc que les objets instanciant la classe B instancient également la classe A.

Prenons comme exemple des classes `Carre`, `Rectangle` et `Cercle`. La figure 4.1 propose une organisation hiérarchique de ces classes telle que `Carre` hérite de `Rectangle` qui hérite, ainsi que `Cercle`, d'une classe `Forme`.

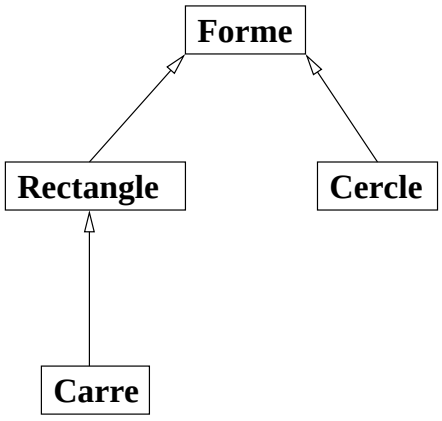



FIGURE 4.1 – Exemple de relations d'héritage

Pour le moment, nous considérerons la classe `Forme` comme vide (c'est-à-dire sans aucune variable ni méthode) et nous nous intéressons plus particulièrement aux classes `Rectangle` et `Carre`.

La classe `Rectangle` héritant d'une classe vide, elle ne peut profiter d'aucun de ses attributs et doit définir toutes ses variables et méthodes. Une relation d'héritage se définit en Java par le mot-clé `extends` utilisé comme dans l'exemple suivant :

```
public class Rectangle extends Forme {

   private int largeur;
   private int longueur;

   public Rectangle(int x, int y) {
      this.largeur = x;
      this.longueur = y;
   }

   public int getLargeur() {
      return this.largeur;
   }

   public int getLongueur() {
      return this.longueur;
   }

   public int surface() {
      return this.longueur * this.largeur;
   }

   public void affiche() {
      System.out.println("rectangle " + longueur + "x" + largeur);
   }
}
```

En revanche, la classe `Carre` peut bénéficier de la classe `Rectangle` et ne nécessite pas la réécriture de ces méthodes si celles-ci conviennent à la sous-classe. Toutes les méthodes et variables de la classe `Rectangle` ne sont néanmoins pas accessibles dans la classe `Carre`. Pour qu'un attribut puisse être utilisé dans une sous-classe, il faut que son type d'accès soit `public` ou `protected`, ou, si les deux classes sont situées dans le même package, qu'il utilise le type d'accès par défaut. Dans cet exemple, les variables `longueur` et `largeur` ne sont pas accessibles dans la class `Carre` qui doit passer par les méthodes `getLargeur()` et `getLongueur()`, déclarées comme publiques.

### 4.1.1 Redéfinition

L'héritage intégral des attributs de la classe `Rectangle` pose deux problèmes :

1. il faut que chaque carré ait une longueur et une largeur égales ;
2. la méthode `affiche` écrit le mot "`rectangle`" en début de chaîne. Il serait souhaitable que ce soit "`carré`" qui s'affiche.

De plus, les constructeurs ne sont pas hérités par une sous-classe. Il faut donc écrire un constructeur spécifique pour `Carre`. Ceci nous permettra de résoudre le premier problème en écrivant un constructeur qui ne prend qu'un paramètre qui sera affecté à la longueur et à la largeur. Pour attribuer une valeur à ces variables (qui sont privées), le constructeur de `Carre` doit faire appel au

constructeur de `Rectangle` en utilisant le mot-clé `super` qui fait appel au constructeur de la classe supérieure comme suit :

```
public Carre(int cote) {
  super(cote,cote);
}
```

☛ **Remarques :**

- L'appel au constructeur d'une classe supérieure doit toujours se situer dans un constructeur et toujours en tant que première instruction ;
- Si aucun appel à un constructeur d'une classe supérieure n'est fait, le constructeur fait appel implicitement à un constructeur vide de la classe supérieure (comme si la ligne `super()` était présente). Si aucun constructeur vide n'est accessible dans la classe supérieure, une erreur se produit lors de la compilation.

Le second problème peut être résolu par une *redéfinition* de méthode. On dit qu'une méthode d'une sous-classe redéfinit une méthode de sa classe supérieure, si elles ont la même signature mais que le traitement effectué est ré-écrit dans la sous-classe. Voici le code de la classe `Carre` où sont résolus les deux problèmes soulevés :

```
public class Carre extends Rectangle {

   public Carre(int cote) {
      super(cote, cote);
   }

   public void affiche() {
      System.out.println("carré " + this.getLongueur());
   }
}
```

Lors de la redéfinition d'une méthode, il est encore possible d'accéder à la méthode redéfinie dans la classe supérieure. Cet accès utilise également le mot-clé `super` comme préfixe à la méthode. Dans notre cas, il faudrait écrire `super.affiche()` pour effectuer le traitement de la méthode `affiche()` de `Rectangle`.

Enfin, il est possible d'interdire la redéfinition d'une méthode ou d'une variable en introduisant le mot-clé `final` au début d'une signature de méthode ou d'une déclaration de variable. Il est aussi possible d'interdire l'héritage d'une classe en utilisant `final` au début de la déclaration d'une classe (avant le mot-clé `class`).

### 4.1.2 Polymorphisme

Le polymorphisme est la faculté attribuée à un objet d'être une instance de plusieurs classes. Il a une seule classe "réelle" qui est celle dont le constructeur a été appelé en premier (c'est-à-dire la classe figurant après le `new`) mais il peut aussi être déclaré avec une classe supérieure à sa classe réelle. Cette propriété est très utile pour la création d'ensembles regroupant des objets de classes différentes comme dans l'exemple suivant :

```
Forme[] tableau = new Forme[4];
tableau[0] = new Rectangle(10,20);
tableau[1] = new Cercle(15);
tableau[2] = new Rectangle(5,30);
tableau[3] = new Carre(10);
```

L'opérateur `instanceof` peut être utilisé pour tester l'appartenance à une classe comme suit :

```
for (int i = 0 ; i < tableau.length ; i++) {
   if (tableau[i] instanceof Forme)
      System.out.println("element " + i + " est une forme");
   if (tableau[i] instanceof Cercle)
      System.out.println("element " + i + " est un cercle");
   if (tableau[i] instanceof Rectangle)
      System.out.println("element " + i + " est un rectangle");
   if (tableau[i] instanceof Carre)
      System.out.println("element " + i + " est un carré");
}
```

L'exécution de ce code sur le tableau précédent affiche le texte suivant :

```
element[0] est une forme
element[0] est un rectangle
element[1] est une forme
element[1] est un cercle
element[2] est une forme
element[2] est un rectangle
element[3] est une forme
element[3] est un rectangle
element[3] est un carré
```

L'ensemble des classes Java, y compris celles écrites en dehors de l'API, forme une hiérarchie avec une racine unique. Cette racine est la classe `Object` dont hérite toute autre classe. En effet, si vous ne précisez pas explicitement une relation d'héritage lors de l'écriture d'une classe, celle-ci hérite par défaut de la classe `Object`. Grâce à cette propriété, des classes génériques [1] de création et de gestion d'un ensemble, plus élaborées que les tableaux, regroupent des objets appartenant à la classe `Object` (donc de n'importe quelle classe).

Une des propriétés induites par le polymorphisme est que l'interpréteur Java est capable de trouver le traitement à effectuer lors de l'appel d'une méthode sur un objet. Ainsi, pour plusieurs objets déclarés sous la même classe (mais n'ayant pas la même classe réelle), le traitement associé à une méthode donné peut être différent. Si cette méthode est redéfinie par la classe réelle d'un objet (ou par une classe située entre la classe réelle et la classe de déclaration), le traitement effectué est celui défini dans la classe la plus spécifique de l'objet et qui redéfinie la méthode.

Dans notre exemple, la méthode `affiche()` est redéfinie dans toutes les sous-classes de `Forme` et les traitements effectués sont :

```
for (int i = 0 ; i < tableau.length ; i++) {
   tableau[i].affiche();
}
```

1. voir par exemple les classes `java.util.Vector`, `java.util.Hashtable`, …

Résultat :

```
rectangle 10x20
cercle 15
rectangle 5x30
carré 10
```

Dans l'état actuel de nos classes, ce code ne pourra cependant pas être compilé. En effet, la fonction `affiche()` est appelée sur des objets dont la classe déclarée est `Forme` mais celle-ci ne contient aucune fonction appelée `affiche()` (elle est seulement définie dans ses sous-classes). Pour compiler ce programme, il faut transformer la classe `Forme` en une interface ou une classe abstraite tel que cela est fait dans les sections suivantes.

## 4.2 Interfaces

Une interface est un type, au même titre qu'une classe, mais abstrait et qui donc ne peut être instancié (par appel à `new` plus constructeur). Une interface décrit un ensemble de signatures de méthodes, sans implémentation, qui doivent être implémentées dans toutes les classes qui *implémentent* l'interface. L'utilité du concept d'interface réside dans le regroupement de plusieurs classes, tel que chacune implémente un ensemble commun de méthodes, sous un même type. Une interface possède les caractéristiques suivantes :

- elle contient des signatures de méthodes ;
- elle ne peut pas contenir de variables ;
- une interface peut hériter d'une autre interface (avec le mot-clé `extends`) ;
- une classe (abstraite ou non) peut implémenter plusieurs interfaces. La liste des interfaces implémentées doit alors figurer après le mot-clé `implements` placé dans la déclaration de classe, en séparant chaque interface par une virgule.

Dans notre exemple, `Forme` peut être une interface décrivant les méthodes qui doivent être implémentées par les classes `Rectangle` et `Cercle`, ainsi que par la classe `Carre` (même si celle-ci peut profiter de son héritage de `Rectangle`). L'interface `Forme` s'écrit alors de la manière suivante :

```
public interface Forme {
   public int surface() ;
   public void affiche() ;
}
```

Pour obliger les classes `Rectangle`, `Cercle` et `Carre` à implémenter les méthodes `surface()` et `affiche()`, il faut modifier l'héritage de ce qui était la classe `Forme` en une implémentation de l'interface définie ci-dessus :

```
public class Rectangle implements Forme {
   ...
}
```

et

```
public class Cercle implements Forme {
   ...
}
```

Cette structure de classes nous permet désormais de pouvoir compiler l'exemple donné dans la section 4.1.2 traitant du polymorphisme. En déclarant un tableau constitué d'objets implémentant l'interface `Forme`, on peut appeler la méthode `affiche()` qui existe et est implémentée par chaque objet.

Si une classe implémente une interface mais que le programmeur n'a pas écrit l'implémentation de toutes les méthodes de l'interface, une erreur de compilation se produira sauf si la classe est une classe abstraite.

## 4.3 Classes abstraites

Le concept de classe abstraite se situe entre celui de classe et celui d'interface. C'est une classe qu'on ne peut pas directement instancier car certaines de ses méthodes ne sont pas implémentées. Une classe abstraite peut donc contenir des variables, des méthodes implémentées et des signatures de méthode à implémenter. Une classe abstraite peut implémenter (partiellement ou totalement) des interfaces et peut hériter d'une classe ou d'une classe abstraite.

Le mot-clé `abstract` est utilisé devant le mot-clé `class` pour déclarer une classe abstraite, ainsi que pour la déclaration de signatures de méthodes à implémenter.

Imaginons que l'on souhaite attribuer deux variables, `origine_x` et `origine_y`, à tout objet représentant une forme. Comme une interface ne peut contenir de variables, il faut transformer `Forme` en classe abstraite comme suit :

```
public abstract class Forme {
   private int origine_x;
   private int origine_y;

   public Forme() {
      this.origine_x = 0;
      this.origine_y = 0;
   }

   public int getOrigineX() {
      return this.origine_x;
   }

   public int getOrigineY() {
      return this.origine_y;
   }

   public void setOrigineX(int x) {
      this.origine_x = x;
   }

   public void setOrigineY(int y) {
      this.origine_y = y;
   }

   public abstract int surface();

   public abstract void affiche();
}
```

De plus, il faut rétablir l'héritage des classes `Rectangle` et `Cercle` vers `Forme` :

```
public class Rectangle extends Forme {
    ...
}
```

et

```
public class Cercle extends Forme {
    ...
}
```

Lorsqu'une classe hérite d'une classe abstraite, elle doit :

- soit implémenter les méthodes abstraites de sa super-classe en les dotant d'un corps ;
- soit être elle-même abstraite si au moins une des méthodes abstraites de sa super-classe reste abstraite.

## 4.4 Classes et méthodes génériques

Il est parfois utile de définir des classes paramétrées par un type de données (ou une classe). Par exemple, dans le package `java.util`, de nombreuses classes sont génériques et notamment les classes représentant des ensembles (`Vector`, `ArrayList`, etc.). Ces classes sont génériques dans le sens où elles prennent en paramètre un type (classe ou interface) quelconque `E`. `E` est en quelque sorte une variable qui peut prendre comme valeur un type de donné. Ceci se note comme suit, en prenant l'exempe de `java.util.ArrayList` :

```
package java.util ;

public class ArrayList<E>
      extends AbstractList<E>
      implements List<E>, ...
{
   ...
   public E set(int index, E element) {
      ...
   }

   public boolean add(E e) {
      ...
   }
   ...
}
```

Nous pouvons remarquer que le type passé en paramètre est noté entre chevrons (ex : `<E>`), et qu'il peut ensuite être réutilisé dans le corps de la classe, par des méthodes (ex : la méthode `set` renvoie un élément de classe `E`).

Il est possible de définir des contraintes sur le type passé en paramètre, comme par exemple une contrainte de type `extends` [2] :

2. Ici, on utilise `T extends E` pour signaler que le type `T` est un sous type de `E`, que `E` soit une classe ou une interface (on n'utilise pas `implements`).

```
public class SortedList<T extends Comparable<T>> {
    ...
}
```

Ceci signifie que la classe `SortedList` (liste ordonnée que nous voulons définir) est paramétrée par le type `T` qui doit être un type dérivé (par héritage ou interfaçage) de `Comparable<T>`. En bref, nous définissons une liste ordonnée d'éléments comparables entre eux (pour pouvoir les trier), grâce à la méthode `int compareTo(T o)` de l'interface `Comparable`[3] qui permet de comparer un `Comparable` à un élément de type `T`.

3. Voir `java.lang.Comparable` pour plus d'information sur cette interface.

# Chapitre 5

# Gestion des exceptions

Lors de l'écriture d'un programme, la prise en compte d'erreurs prend une place très importante si l'on souhaite écrire un programme robuste. Par exemple, la simple ouverture d'un fichier peut provoquer beaucoup d'erreurs telles que l'inexistence du fichier, un mauvais format, une interdiction d'accès, une erreur de connexion au périphérique, … Pour que notre programme soit robuste, il faut que toutes les erreurs possibles soient détectées et traitées.

Certains langages de programmation, dont le langage Java, proposent un mécanisme de prise en compte des erreurs, fondé sur la notion d'*exception*. Une exception est un objet qui peut être émis par une méthode si un événement d'ordre "exceptionnel" (les erreurs rentrent dans cette catégorie) se produit. La méthode en question ne renvoie alors pas de valeur de retour, mais émet une exception expliquant la cause de cette émission. La propagation d'une émission se déroule selon les étapes suivantes :

1. Une exception est générée à l'intérieur d'une méthode ;
2. Si la méthode prévoit un traitement de cette exception, on va au point 4, sinon au point 3 ;
3. L'exception est renvoyée à la méthode ayant appelé la méthode courante, on retourne au point 2 ;
4. L'exception est traitée et le programme reprend son cours après le traitement de l'exception.

La gestion d'erreurs par propagation d'exception présente deux atouts majeurs :

- **Une facilité de programmation et de lisibilité** : il est possible de regrouper la gestion d'erreurs à un même niveau. Cela évite des redondances dans l'écriture de traitements d'erreurs et encombre peu le reste du code avec ces traitements.
- **Une gestion des erreurs propre et explicite** : certains langages de programmation utilisent la valeur de retour des méthodes pour signaler une erreur à la méthode appelante. Etant donné que ce n'est pas le rôle de la valeur de retour de décrire une erreur, il est souvent impossible de connaître les causes réelles de l'erreur. La dissociation de la valeur de retour et de l'exception permet à cette dernière de décrire précisément la ligne de code ayant provoqué l'erreur et la nature de cette erreur.

## 5.1 Déclaration

Il est nécessaire de déclarer, pour chaque méthode, les classes d'exception qu'elle est susceptible d'émettre. Cette déclaration se fait à la fin de la signature d'une méthode par le mot-clé `throws` à la suite duquel les classes d'exceptions (séparées par une virgule s'il en existe plusieurs) qui peuvent être générées sont précisées. La méthode `parseInt` de la classe Integer est un bon exemple :

```
public static int parseInt(String s) throws NumberFormatException {
    ...
}
```

Cette méthode convertit une chaîne de caractères, qui doit contenir uniquement des chiffres, en un entier. Une erreur peut se produire si cette chaîne de caractères ne contient pas que des chiffres. Dans ce cas une exception de la classe `NumberFormatException` est émise.

Une exception peut être émise dans une méthode de deux manières : (i) par une autre méthode appelée dans le corps de la première méthode ; (ii) par la création d'un objet instanciant la classe `Exception` (ou la classe `Throwable`) et la levée explicite de l'exception en utilisant le mot-clé `throw`.

L'exemple ci-dessous illustre ce second cas :

```
public class ExempleException {
   /*
    * Cette méthode renvoie le nom du mois
    * correspondant au chiffre donné en paramètre.
    * Si celui-ci n'est pas valide une exception de classe
    * IndexOutOfBoundsException est levée.
    */

   public static String month(int mois)
           throws IndexOutOfBoundsException {
      if ((mois < 1) || (mois > 12)) {
         throw new IndexOutOfBoundsException(
                 "le numero du mois qui est "
                    + mois
                    + " doit être compris entre 1 et 12");
      }
      if (mois == 1)
         return "Janvier";
      else if (mois == 2)
         return "Février";
      ...
      else if (mois == 11)
         return "Novembre";
      else
         return "Décembre";
   }
}
```

La signification des exceptions de la classe `IndexOutOfBoundsException` est qu'un index donné dépasse les bornes minimum et maximum qu'il devrait respecter. Si une méthode demande la chaîne de caractères correspondant à des mois inexistants (inférieur à 1 ou supérieur à 12) une exception signalera cette erreur. On peut remarquer dans cet exemple que la détection et la formulation de l'erreur sont codées dans la méthode mais pas son traitement.

## 5.2 Interception et traitement

Avant de coder le traitement de certaines exceptions, il faut préciser l'endroit où elles vont être interceptées. Si une méthode `A` appelle une méthode `B` qui appelle une méthode `C` qui appelle une méthode `D`, et que cette méthode `D` lève une exception, celle-ci est d'abord transmise à `C` qui peut l'intercepter ou la transmettre à `B`, qui peut aussi l'intercepter ou la transmettre à `A`.

L'interception d'exceptions se fait par une sorte de "mise sur écoute" d'une portion de code. Pour cela on utilise le mot-clé `try` suivi du bloc à surveiller. Si aucune exception ne se produit dans le bloc correspondant, le programme se déroule normalement comme si l'instruction `try` était absente. Par contre, si une exception est levée, le traitement de l'exception est exécuté puis l'exécution du programme reprend son cours après le bloc testé.

Il est également nécessaire de préciser quelles classes d'exception doivent être interceptées dans le bloc testé. L'interception d'une classe d'exception s'écrit grâce au mot-clé `catch` suivi de la classe concerné, d'un nom de variable correspondant à l'objet exception, puis du traitement. Si une exception est levée sans qu'aucune interception ne soit prévue pour sa classe, celle-ci est propagée à la méthode précédente.

Dans l'exemple ci-dessous, le programme demande à l'utilisateur de saisir le numero d'un mois et affiche à l'écran le nom de ce mois. Les exceptions qui peuvent être levées par ce programme sont traitées.

```
public class ExempleTraitementException {

   public static void main(String[] args) {
      System.out.print("Entrez le numero d'un mois : ");
      try {
         BufferedReader input = new BufferedReader(
                  new InputStreamReader(System.in));
         String choix = input.readLine();
         int numero = Integer.parseInt(choix);
         System.out.println(ExempleException.month(numero));
      } catch (IndexOutOfBoundsException e) {
         System.err.println("Numero incorrect : "
            + e.getMessage());
      } catch (NumberFormatException e) {
         System.err.println("Entrée incorrecte : "
            + e.getMessage());
      } catch (IOException e) {
         System.err.println("Erreur d'accès : "
            + e.getMessage());
      }
   }
}
```

Trois classes d'exception sont ici traitées :

- `IndexOutOfBoundsException` (levé par la méthode `month`) se produit si le numero entré par l'utilisateur est inférieur à 1 ou supérieur à 12 ;
- `NumberFormatException` (levé par la méthode `parseInt`) qui se produit si le texte entré par l'utilisateur n'est pas convertible en entier ;
- `IOException` (levé par la méthode `readLine`) qui se produit si il y a eu une erreur d'accès au périphérique d'entrée.

Dans chacun de ces cas, le traitement consiste à afficher le message d'erreur associé à l'exception.

## 5.3 Classes d'exception

Une classe est considérée comme une classe d'exception dès lors qu'elle hérite de la classe `Throwable`. Un grand nombre de classes d'exception sont proposées dans l'API pour couvrir les catégories d'erreurs les plus fréquentes. Les relations d'héritage entre ces classes permettent de lever ou d'intercepter des exceptions décrivant une erreur à différents niveaux de précision. Les classes d'exception les plus fréquemment utilisées sont récapitulées dans le tableau 5.1.

TABLE 5.1 – Classes d'exception fréquentes

| Classe | Description |
|---|---|
| `AWTException` | Les exceptions de cette classe peuvent se produire lors d'opérations de type graphique. |
| `ClassCastException` | Signale une erreur lors de la conversion d'un objet en une classe incompatible avec sa vraie classe. |
| `FileNotFoundException` | Signale une tentative d'ouverture d'un fichier inexistant. |
| `IndexOutOfBoundsException` | Se produit lorsque l'on essaie d'accéder à un élément inexistant dans un ensemble. |
| `IOException` | Les exceptions de cette classe peuvent se produire lors d'opérations d'entrées/ sorties. |
| `NullPointerException` | Se produit lorsqu'un pointeur `null` est reçu par une méthode n'acceptant pas cette valeur, ou lorsque l'on appelle une méthode ou une variable à partir d'un pointeur `null`. |

☛ **Remarque :** Il n'est pas indispensable d'intercepter les exceptions héritant de la classe `RuntimeException` (dans le tableau ci-dessus, les classes qui en héritent sont `ClassCastException`, `IndexOutOfBoundsException` et `NullPointerException`). Celles-ci peuvent être propagées jusqu'à la machine virtuelle et n'apparaître que pendant l'exécution. Ces exceptions sont appelées *exception non vérifiées* ou *unchecked exceptions*.

Si aucune des classes d'exception ne correspond à un type d'erreur que vous souhaitez exprimer, vous pouvez également écrire vos propres classes d'exception. Pour cela, il suffit de faire hériter votre classe de la classe `java.lang.Exception`.

## 5.4 Classification des erreurs en Java

On peut finalement distinguer quatre types de situations d'erreurs en Java :

**Erreurs de compilation.** Avant même de pouvoir exécuter le programme, notre code source génère des erreur par le compilateur. Il faut alors réviser et corriger le code pour ne plus avoir d'erreurs.

**Erreurs d'exécution.** Alors que notre programme est en cours d'exécution, la JVM étant mal configurée ou corrompue, le programme s'arrête ou se gèle. A priori, c'est une erreur non pas due à notre programme, mais à la configuration ou l'état de l'environnement d'exécution de notre programme.

**Exception non vérifiée.** Alors que notre programme est en cours d'exécution, une trace de la pile des exceptions est affichée, pointant vers une partie de notre code sans gestion d'exception. Visiblement, nous avons utilisé du code qui est capable de lever un exception non vérifiée (comme `NullPointerException`). Il faut modifier le programme pour que cette situation ne survienne pas.

**Exception vérifiée.** Alors que notre programme est en cours d'exécution, une trace de la pile des exceptions est affichée, pointant vers une partie de notre code avec gestion d'exception. Visiblement, nous avons produit du code qui est capable de lever un exception vérifiée (comme `FileNotFoundException`) mais les données passées à notre programme ne valide pas ces exceptions (par exemple, lorsque l'on essaie d'ouvrir un fichier qui n'existe pas). Il faut alors revoir les données passées en paramètre du programme. Notre code a bien détecté les problèmes qu'il fallait détecter. Le chapitre suivant sur les entrées/sorties présentent de nombreux exemples relatifs à ce cas.

# Chapitre 6

# Gestion des entrées/sorties simples

Le package `java.io` propose un ensemble de classes permettant de gérer la plupart des entrées/sorties d'un programme. Cette gestion consiste à créer un objet *flux* dans lequel transitent les données à envoyer ou à recevoir. Un flux connecte un objet Java à un autre élément. Deux cas sont illustrés dans ce chapitre : les interactions avec un utilisateur (entrée clavier et sortie écran) et les accès en lecture ou écriture à un fichier.

## 6.1 Flux d'entrée

Un flux d'entrée est une instance d'une sous-classe de `InputStream`. Chaque classe de flux d'entrée a son propre mode d'échange de données qui spécifie un format particulier de données ou un accès particulier. Les classes les plus couramment utilisées sont :

- **ByteArrayInputStream** permet de lire le flux d'entrée sous la forme d'octets (*byte*) ;
- **DataInputStream** permet de lire le flux d'entrée sous la forme de types de données primitifs de Java. Il existe des méthodes pour récupérer un entier, un réel, un caractère,…
- **FileInputStream** est utilisé pour lire le contenu d'un fichier. Les objets de cette classe sont souvent encapsulés dans un autre objet de classe `InputStream` qui définit le format des données à lire.
- **ObjectInputStream** permet de lire des objets (c-à-d des instances de classes Java) à partir du flux d'entrée, si ces objets implémentent les interfaces `java.io.Serializable` ou `java.io.Externalizable`.
- **Reader** n'est pas une sous-classe de `InputStream` mais représente un flux d'entrée pour chaînes de caractères. Plusieurs sous-classes de `Reader` permettent la création de flux pour chaînes de caractères.
- **Scanner** n'est pas une sous-classe de `InputStream`, mais un `Iterator` qui permet de lire un flux (fichier ou chaîne de caractères par exemple) "mot" par "mot" en définissant le délimiteur entre les mots (espace par défaut).

La lecture de données à partir d'un flux d'entrée suit le déroulement suivant :

1. **Ouverture du flux :** Elle se produit à la création d'un objet de la classe `InputStream`. Lors de l'appel au constructeur, on doit préciser quel élement externe est relié au flux (par exemple un nom de fichier ou un autre flux).
2. **Lecture de données :** Des données provenant du flux sont lues au moyen de la méthode `read()`) ou d'une méthode équivalente. La méthode précise à employer dépend du type de flux ouvert.
3. **Fermeture du flux :** Quand le flux n'est plus nécessaire, il doit être fermé par la méthode `close()`.

### 6.1.1 Lecture des entrées clavier

Les données provenant de l'utilisation du clavier sont transmises dans un flux d'entrée créé automatiquement pour toute application Java. On accède à ce flux par la variable statique de la classe `java.lang.System` qui s'appelle `in`. Ce flux est alors utilisé comme paramètre d'entrée du constructeur d'un autre flux d'entrée. Pour cet autre flux, on utilise généralement une sous-classe de `Reader` pour récupérer les entrées de l'utilisateur sous la forme d'une chaîne de caractères. La classe `Clavier` en donne un exemple :

```
import java.io.*;

public class Clavier {
  public static void main(String[] args) {
    try {
      BufferedReader flux = new BufferedReader(
         new InputStreamReader(System.in));
      System.out.print("Entrez votre prenom : ");
      String prenom = flux.readLine();
      System.out.println("Bonjour " + prenom);
      flux.close();
    } catch (IOException ioe) {
      System.err.println(ioe);
    }
  }
}
```

### 6.1.2 Lecture à partir d'un fichier

Un fichier est représenté par un objet de la classe `java.io.File`. Le constructeur de cette classe prend en paramètre d'entrée le chemin d'accès du fichier. Le flux d'entrée est alors créé à l'aide de la classe `FileInputStream` sur lequel on peut lire caractère par caractère grâce à la méthode `read()`. L'exemple suivant présente une méthode pour afficher le contenu d'un fichier :

```
import java.io.*;

public class LectureFichier {

  public static void main(String[] args) {
    try {
      File fichier = new File("monFichier.txt");
      FileInputStream flux = new FileInputStream(fichier);
      int c;
      while ((c = flux.read()) > -1) {
        System.out.write(c);
      }
      flux.close();
    } catch (FileNotFoundException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Il arrive souvent d'enregistrer des données dans des fichiers textes. Il peut alors être utile d'utiliser un `BufferedReader` ou un `Scanner` pour effectuer la lecture. Dans les exemples suivants, on considère une matrice $10 \times 10$ enregistrée dans un fichier texte `matrice.txt` ligne par ligne, avec les colonnes séparées par des espaces :

```
import java.io.*;

public class LectureMatrice {

  public static void main(String[] args) {
    try {
      FileReader fileReader = new FileReader("matrice.txt");
      BufferedReader reader = new BufferedReader(fileReader);
      while (reader.ready()) {
        String[] line = reader.readLine().split(" ");
        for (String s  : line) {
          System.out.print(s);
        }
        System.out.println();
      }
      reader.close();
      fileReader.close();
    } catch (FileNotFoundException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

On peut effectuer un traitement similaire avec un `Scanner` :

```
import java.io.*;
import java.util.Scanner;

public class LectureMatriceScanner {
  public static void main(String[] args) {
    try {
      Scanner fileScanner = new Scanner(new File("matrice.txt"));
      while (fileScanner.hasNextLine()) {
        Scanner lineScanner = new Scanner(fileScanner.nextLine());
        while (lineScanner.hasNext())
           System.out.print(lineScanner.next());
        System.out.println();
      }
    } catch (FileNotFoundException e) {
      e.printStackTrace();
    }
  }
}
```

### 6.1.3 Lecture d'objets enregistrés

Il est parfois utile d'enregistrer l'état d'objet (de n'importe quelle classe implémentant les interfaces `java.io.Serializable` ou `java.io.Externalizable`) pour des exécutions futures. Le flux d'entrée est encore créé à l'aide de la classe `FileInputStream` et est ensuite encapsulé dans un autre flux spécifiant le format des données à lire. L'exemple suivant illustre la lecture d'un objet de la classe `Date` dans un fichier nommé `monFichier.dat` :

```
import java.io.*;
import java.util.Date;

public class LectureDate {
  public static void main(String[] args) {
    try {
      File fichier = new File("monFichier.dat");
      ObjectInputStream flux = new ObjectInputStream(
      new FileInputStream(fichier));
      Date laDate = (Date) flux.readObject();
      System.out.println(laDate);
      flux.close();
    } catch (IOException ioe) {
      System.err.println(ioe);
    } catch (ClassNotFoundException cnfe) {
      System.err.println(cnfe);
    }
  }
}
```

L'objet qui est lu dans le fichier doit être une instance de la classe `java.util.Date`.

## 6.2 Flux de sortie

Un flux de sortie est une instance d'une sous-classe de `OutputStream`. Comme pour les flux d'entrée, chaque classe de flux de sortie a son propre mode d'écriture de données. Les classes les plus couramment utilisées sont :

- **ByteArrayOutputStream** permet d'écrire des octets vers le flux de sortie ;
- **DataOutputStream** permet d'écrire des types de données primitifs de Java vers le flux de sortie.
- **FileOutputStream** est utilisé pour écrire dans un fichier. Les objets de cette classe sont souvent encapsulés dans un autre objet de classe `OutputStream` qui définit le format des données à écrire.
- **ObjectOutputStream** permet d'écrire des objets (c-à-d des instances de classes Java) vers le flux de sortie, si ces objets implémentent les interfaces `Serializable` ou `Externalizable`.
- **Writer** n'est pas une sous-classe de `OutputStream` mais représente un flux de sortie pour chaînes de caractères. Plusieurs sous-classes de `Writer` permettent la création de flux pour chaînes de caractères.

L'écriture de données vers un flux de sortie suit le même déroulement que la lecture d'un flux d'entrée :

1. **Ouverture du flux :** Elle se produit lors de la création d'un objet de la classe `OutputStream`.

2. **Ecriture de données :** Des données sont écrites vers le flux au moyen de la méthode `write()` ou d'une méthode équivalente. La méthode précise à employer dépend du type de flux ouvert.

3. **Fermeture du flux :** Quand le flux n'est plus nécessaire, il doit être fermé par la méthode `close()`.

### 6.2.1 Ecriture sur la sortie standard "écran"

Comme pour les entrées du clavier, l'écriture vers l'écran fait appel à la variable statique `out` de la classe `System`. On appelle généralement la méthode `System.out.print` ou `System.out.println` comme cela a été fait dans de nombreux exemples de ce livret.

### 6.2.2 Ecriture dans un fichier

L'écriture dans un fichier se fait par un flux de la classe `FileOutputStream` qui prend en entrée un fichier (instance de la classe `File`). Ce flux de sortie permet d'écrire des caractères dans le fichier grâce à la méthode `write()`. L'exemple suivant présente une méthode pour écrire un texte dans un fichier :

```
import java.io.*;

public class EcritureFichier {

  public static void main(String[] args) {
    try {
      File fichier = new File("monFichier.txt");
      FileOutputStream flux = new FileOutputStream(fichier);
      String texte = "Hello World!";
      for (int i = 0; i < texte.length(); i++) {
        flux.write(texte.charAt(i));
      }
      flux.close();
    } catch (FileNotFoundException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

On peut également utiliser un `Writer` comme un `FileWriter` pour écrire des chaînes de caractères dans un fichier assez simplement. Dans l'exemple suivant, on écrit une série de 10 lignes de 10 entiers aléatoires séparés par des espaces dans un fichier pouvant être lu par la classe `LectureMatrice` :

```
import java.io.*;
import java.util.Random;

public class EcritureMatrice {

  public static void main(String[] args) {
    try {
      FileWriter writer = new FileWriter("random.txt");
      Random generator = new Random(System.currentTimeMillis());
      for (int i = 0; i < 9; i++) {
        for (int j = 0; j < 9; j++)
           writer.write(generator.nextInt() + " ");
        writer.write("\n");
      }
      writer.close();
    } catch (FileNotFoundException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

### 6.2.3 Ecriture d'objets

Le flux de sortie peut également être encapsulé dans un flux de type `ObjectOutputStream`, comme le montre l'exemple suivant pour écrire la date courante dans un fichier :

```
import java.io.*;
import java.util.Date;

public class EcritureDate {

  public static void main(String[] args) {
    try {
      File fichier = new File("monFichier.dat");
      ObjectOutputStream flux = new ObjectOutputStream(
      new FileOutputStream(fichier));
      flux.writeObject(new Date());
      flux.close();
    } catch (IOException ioe) {
      System.err.println(ioe);
    }
  }
}
```

Ce fichier peut ensuite être lu par la classe `LectureDate`.

☛ **Remarque importante :** Nous avons présenté dans ce chapitre plusieurs exemples d'entrée/-sortie utilisant différents modes de lecture/écriture (avec ou sans flux). Nous conseillons toutefois d'utiliser en *priorité* un `Scanner` pour la lecture dans un fichier, et un `FileWriter` pour l'écriture dans un fichier.

# Annexes

# Annexe A

# Applications graphiques (package `swing`)

Annexe

Lors des premières versions du langage Java, le seul package fourni par défaut par Java SE permettant de construire des interfaces graphiques était le package `java.awt`. Depuis la version 1.1 du JDK, il est très fortement recommandé d'utiliser les classes du package `javax.swing` pour écrire des interfaces graphiques. En effet, le package swing apporte deux avantages conceptuels par rapport au package awt :

- Les composants swing sont dits "légers" (*lightweight*) contrairement aux composants "lourds" (*heavyweight*) d'awt. L'apparence graphique d'un composant dit lourd dépend du système d'exploitation car elle fait appel à un composant correspondant dans le système d'exploitation. Avec un composant léger, son apparence (*look-and-feel*) est fixée et peut être modifiée dans le code Java et il est donc possible de donner à une fenêtre une apparence à la Windows tout en utilisant Linux.
- Il applique complètement le schéma Modèle-Vue-Contrôleur (cf. section A.1).

La plupart des classes de composants du package swing héritent de classes du package awt en redéfinissant certaines méthodes pour assurer les deux avantages précédemment cités. Cependant, le package awt n'est pas entièrement obsolète et certaines de ces classes sont encore utilisées pour la gestion d'événements (cf. section A.3) ou la disposition des composants (classes implémentant l'interface *LayoutManager*).

## A.1 Le schéma Modèle-Vue-Contrôleur

Le schéma Modèle-Vue-Contrôleur (MVC) est un mode de conception d'applications graphiques préconisant la distinction des données, de leur apparence graphique et des traitements qui peuvent être effectués. Un composant graphique est décomposé en trois parties :

- le **modèle** contenant ses données et son état courant ;
- la **vue** qui correspond à son apparence graphique ;
- le **contrôleur** associant des traitements à des événements pouvant se produire au sein du composant.

La figure A.1 donne un exemple d'une interface graphique simple contenant un bouton "Quitter".

la décomposition MVC de ce bouton (instance de la classe `JButton`) est constituée : (i) du modèle contenant la chaîne de caractères "`Quitter`" ; (ii) de la vue constituée d'un rectangle gris à bord noir, d'une taille donnée, à une position donnée et au sein duquel est écrite la chaîne de caractères du modèle ; (iii) du contrôleur implémentant le traitement à effectuer lors d'un click sur le bouton, c'est-à-dire l'appel à l'instruction de fermeture de la fenêtre.

Tous les composants graphiques du package swing ont un modèle, une vue et peuvent avoir plusieurs contrôleurs. Pour faciliter la programmation, certaines classes (tel que `JButton`) encapsulent le modèle et la vue dans une seule classe disposant de méthodes d'accès et de modifications

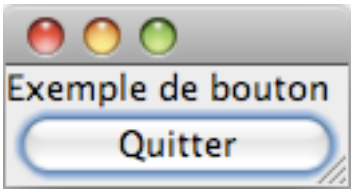



FIGURE A.1 – Exemple de `JButton`

des données ou de l'apparence. Dans ces cas-là, un modèle par défaut est utilisé.

## A.2 Composants graphiques

Dans un programme Java, une classe de composant graphique est une classe qui hérite de `java.awt.Component`. Il est indispensable d'hériter de cette classe car elle implémente de nombreuses méthodes nécessaires à l'affichage du composant, en renvoyant la dimension du composant, sa police de caractères, sa couleur de fond, etc.

La seconde classe essentielle pour l'écriture d'interfaces graphiques est la classe `Container`. Un container est un composant (qui hérite donc de `Component`) mais avec la particularité de pouvoir contenir d'autres composants. Ainsi, une interface graphique est un objet d'une classe de type `java.awt.Container` (par exemple une fenêtre ou une boîte de dialogue) regroupant des simples composants et d'autres containers qui, eux aussi, contiennent d'autres simples composants et containers, et ainsi de suite.

### A.2.1 Composants

Nous ne présentons pas ici toutes les classes de composants du package swing pour nous intéresser plutôt à la vue et au modèle de chaque composant. L'utilisation de certains composants est illustrée dans la section A.2.3.

#### Vue

L'apparence graphique d'un composant est le résultat de l'implémentation de la méthode `paint()` de la classe `Component`. Quasiment toutes les classes héritant de `Component` redéfinissent cette méthode afin de définir l'affichage du composant. Si vous écrivez une classe héritant d'une classe de composant existante, il est donc fortement déconseillé de redéfinir la méthode `paint` pour la plupart des composants existants car vous empêcheriez son affichage normal.

Si vous souhaitez définir l'affichage d'un composant personnalisé (par exemple pour afficher une image ou des formes géométriques), vous devrez faire hériter votre classe de composant de la classe `java.awt.Canvas` qui représente un composant vide. Dans ce cas-là, il est souhaitable de redéfinir la méthode `paint()` qui est vide dans la classe `Canvas`.

#### Modèle

Il existe pour chaque classe de composant, une seconde classe qui gère le modèle de ce composant (par exemple, `ButtonModel` représente un modèle de bouton). En général un modèle par défaut est créé dans le constructeur du composant. L'accès à ce modèle est très souvent masqué car il existe des méthodes, telles que `setText()` pour changer le texte d'un bouton ou `setSelected()` pour changer l'état d'une case à cocher, de la classe de composant qui peuvent être directement appelées pour changer les données du modèle. Cependant, pour des composants plus complexes

tels que des tables (`JTable`) ou des arborescences (`JTree`), il est parfois indispensable de créer un modèle personnalisé.

### A.2.2 Containers

En premier lieu, il convient de rappeler qu'un container est un composant, et, par conséquent, que tout ce qui a été présenté précédemment est aussi vrai pour les containers. En plus des caractéristiques d'un composant simple, un container contient un ensemble de composants ainsi que des contraintes sur leur disposition. L'ajout d'un composant à un container se fait par une méthode `add()`. Il existe plusieurs méthodes `add()` acceptant des paramètres d'entrée différents. La méthode à utiliser dépend du type de disposition choisi.

#### Disposition des composants

L'objet qui gère la disposition des composants est une instance d'une classe qui doit implémenter l'interface `java.awt.LayoutManager`. Cet objet doit être ajouté au container, soit lors de l'appel à son constructeur, soit par la méthode `setLayout()`. Un seul `LayoutManager` est autorisé par container. Quelques classes utilisées pour la disposition des composants sont présentées ci-dessous.

**BorderLayout.** Le container est découpé en cinq cases : une case par côté (gauche, droite, haut et bas) et une au centre. L'ajout d'un composant se fait en précisant la case à utiliser grâce à des variables de classe de `BorderLayout` :

```
MyContainer.add(MyComponent,BorderLayout.NORTH);
MyContainer.add(MyComponent2,BorderLayout.CENTER);
```

La taille de chaque case dépend : (i) de la largeur maximale entre la case du nord, du sud et l'addition des largeurs des cases du centre, de l'est et de l'ouest ; (ii) de la hauteur maximale entre la case à l'est, à l'ouest et l'addition des hauteurs des cases du centre, du nord et du sud. La case du centre est étirée pour remplir tout l'espace restant.

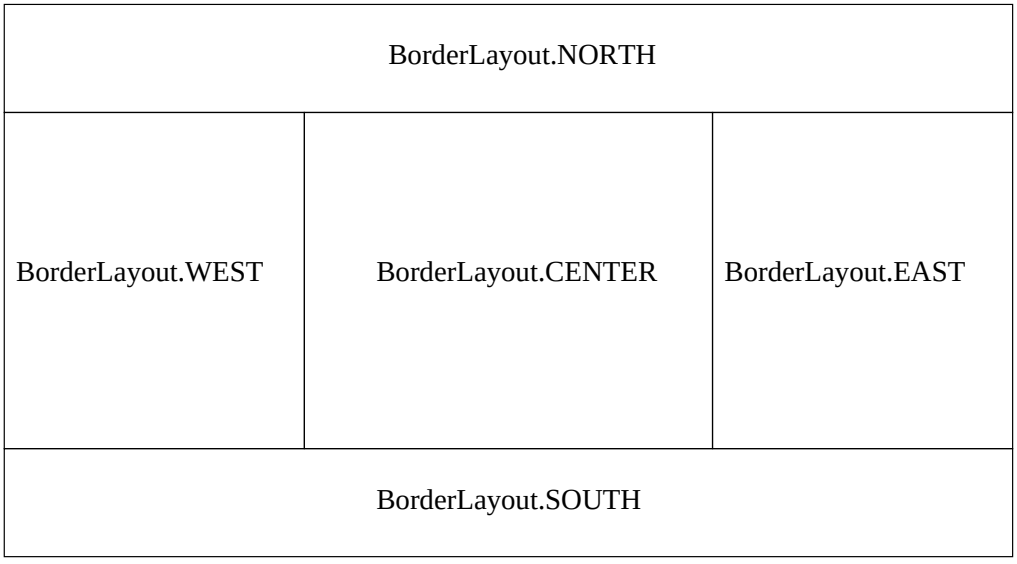



FIGURE A.2 – Découpage du BorderLayout

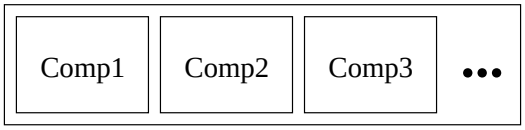



FIGURE A.3 – Découpage du FlowLayout

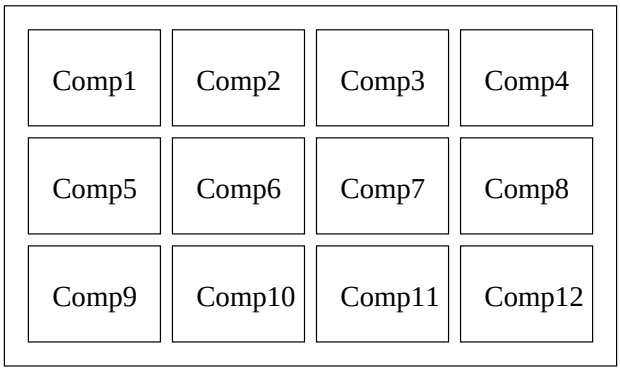



FIGURE A.4 – Découpage d'un GridLayout 3x4

**FlowLayout.** Les composants sont ajoutés les uns à la suite des autres et de la gauche vers la droite. Dès qu'une ligne est remplie de composants (c'est-à-dire dès que la largeur du container est atteinte, les composants restants sont ajoutés sur une ligne en dessous. Il n'y a pas d'autres contraintes de disposition :

```
MyContainer.add(MyComponent);
```

**GridLayout.** Le container est découpé en une grille composée de n cases de taille égale. Le constructeur du GridLayout requiert en paramètres d'entrée le nombre de lignes et le nombre de colonnes de la grille. Les composants sont ajoutés les uns à la suite des autres en remplissant d'abord la première ligne, puis la seconde, etc. Il n'y a pas d'autres contraintes de disposition :

```
MyContainer.add(MyComponent);
```

### Containers classiques

La classe de container la plus simple est `javax.swing.JPanel`. Un panel est un composant initialement vide auquel on va ajouter des composants suivant la disposition choisie. La disposition peut être spécifiée lors de l'appel au constructeur du panel (par ex, `JPanel myPanel = new JPanel(new BorderLayout());`) ou par un appel explicite à la méthode `setLayout`, par exemple :

```
myPanel.setLayout(new GridLayout(4,5));
```

Les composants peuvent ensuite être ajoutés avec la méthode `add()` appropriée.

Une interface graphique consiste donc en une hiérarchie de composants telle que certains composants sont des containers spécifiant la disposition d'autres composants. La racine de cette

hiérarchie est également un container qui peut s'afficher librement (c-à-d. sans que sa disposition soit précisée dans un container). Parmi ces containers de haut niveau, on trouve la classe `javax.swing.JFrame` qui représente une fenêtre et `javax.swing.JDialog` qui représente une boîte de dialogue. La création d'une fenêtre suit généralement les étapes suivantes :

1. Création d'une instance de `JFrame` (ou d'une sous-classe).
2. Ajout des composants au panel de la fenêtre. Ce panel est accessible par la méthode `getContentPane()`. On peut également attribuer à la fenêtre un nouveau panel par la méthode `setContentPane()`.
3. Calcul de la taille de la fenêtre à l'aide de la méthode `pack()`. Ce calcul est automatique et prend en compte la dimension préférée de chaque composant ainsi que leur disposition.
4. Affichage de la fenêtre par la méthode `setVisible()`.

Par exemple, la figure A.1 (p. 44) est une instance d'une sous-classe de `JFrame` :

```
import java.awt.*;
import javax.swing.*;

public class ExBouton extends JFrame {

   public ExBouton() {
      super();
      JLabel label = new JLabel("Exemple de bouton");
      JPanel panel = new JPanel(new BorderLayout());
      panel.add(label, BorderLayout.NORTH);
      JButton b = new JButton(" Quitter ");
      panel.add(b, BorderLayout.SOUTH);
      this.setContentPane(panel);
      this.pack();
   }

   public static void main(String[] args) {
      ExBouton bouton = new ExBouton();
      bouton.setVisible(true);
   }
}
```

### A.2.3 Exemple d'interface

Le meilleur moyen d'apprendre à écrire des interfaces graphiques est de s'inspirer d'exemples d'interfaces existantes. La figure A.5 est un exemple de fenêtre produit avec le code suivant la figure.

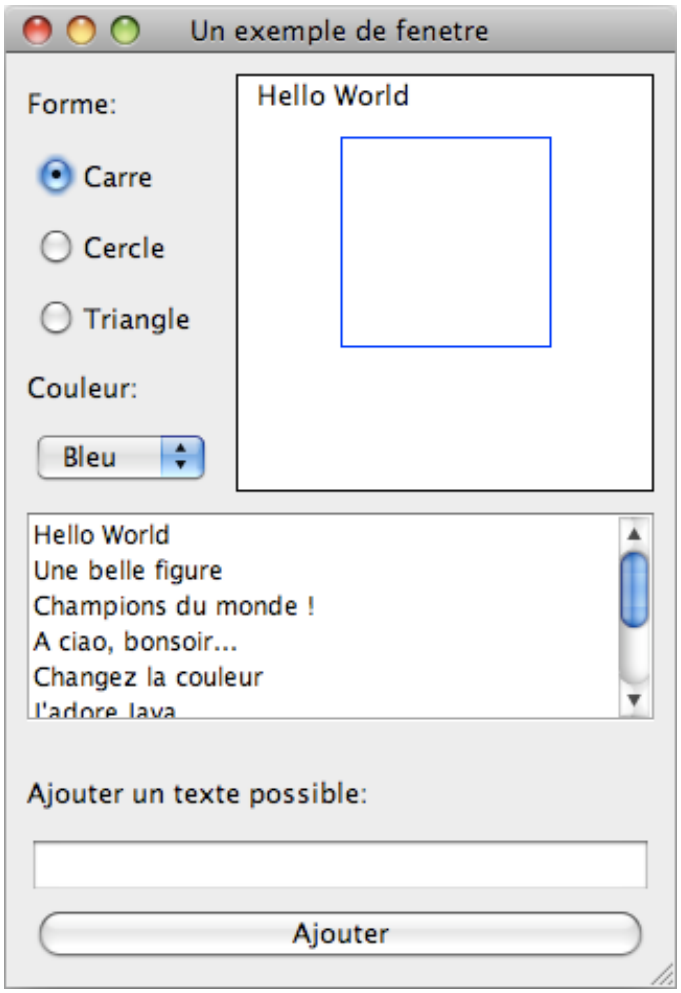



FIGURE A.5 – Exemple d'interface graphique

```java
import java.awt.*;
import java.util.Vector;
import javax.swing.*;
import javax.swing.border.EmptyBorder;

public class ExInterface extends JFrame {

   /**
    * Les variables pointant chaque composant sont declarees
    * en variables d'instance et en type protected. Elles
    * auraient tres bien pu etre des variables locales au
    * constructeur mais, pour des raisons pedagogiques,
    * la gestion des evenements n'est presentee que plus
    * tard (section events) dans une classe heritant
    * de ExInterface. Les variables sont de type protected
    * afin de pouvoir y accéder dans cette sous-classe.
    */
   protected JRadioButton button1,  button2,  button3;
   protected JComboBox couleur;
   protected JTextField text;
   protected JButton ajout;
   protected JList textList;
   protected Vector<String> listData;
   protected Dessin dessin;

   public ExInterface() {
      super("Un exemple de fenetre");

      //le programme doit se terminer quand la fenetre est fermee
      this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      // creation du panel gauche
      JPanel gauche = new JPanel(new GridLayout(6, 1, 0, 5));

      // creation des cases a cocher
      JLabel label = new JLabel("Forme:");
      button1 = new JRadioButton("Carre", true);
      button2 = new JRadioButton("Cercle", false);
      button3 = new JRadioButton("Triangle", false);
      JLabel label2 = new JLabel("Couleur:");
      couleur = new JComboBox();
      couleur.addItem("Bleu");
      couleur.addItem("Rouge");
      couleur.addItem("Vert");

      // le groupe de bouton permet d'interdire la selection
      // de plusieurs cases en meme temps
      ButtonGroup group = new ButtonGroup();
      group.add(button1);
      group.add(button2);
      group.add(button3);

      // remplissage du panel gauche
      gauche.add(label);
      gauche.add(button1);
      gauche.add(button2);
      gauche.add(button3);
      gauche.add(label2);
      gauche.add(couleur);

      // creation du panel bas
      JPanel bas = new JPanel(new BorderLayout(0, 10));

      // creation et remplissage du panel d'ajout de texte
      JPanel textPanel = new JPanel(new GridLayout(3, 1, 0, 5));
      label = new JLabel("Ajouter un texte possible:");
      textPanel.add(label);
      text = new JTextField(16);
      textPanel.add(text);
      ajout = new JButton("Ajouter");
      textPanel.add(ajout);

      // creation du modele de la liste
      listData = new Vector<String>();
      listData.addElement("Hello World");
      listData.addElement("Une belle figure");
      listData.addElement("Champions du monde !");
      listData.addElement("A ciao, bonsoir...");
      listData.addElement("Changez la couleur");
      listData.addElement("J'adore Java");
      listData.addElement("Allez les verts !");
      listData.addElement("Ca marche");
      listData.addElement("blablabla");

      // creation de la liste
      textList = new JList(listData);
      textList.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);

      // l'inclusion de la liste dans un JScrollPane permet
      // de la faire defiler par des ascenceurs
      JScrollPane scrollPane = new JScrollPane(textList);
      scrollPane.setPreferredSize(new Dimension(200, 100));

      // remplissage du panel bas
```

```
        bas.add(scrollPane, BorderLayout.NORTH);
        bas.add(textPanel, BorderLayout.SOUTH);

        // remplissage du panel principal
        JPanel mainPanel = (JPanel) this.getContentPane();
        mainPanel.setLayout(new BorderLayout(10, 10));
        mainPanel.add(gauche, BorderLayout.WEST);
        dessin = new Dessin();
        mainPanel.add(dessin, BorderLayout.CENTER);
        mainPanel.add(bas, BorderLayout.SOUTH);

        // une bordure permet d'aerer l'affichage
        mainPanel.setBorder(new EmptyBorder(10, 10, 10, 10));

        // calcul de la dimension de la fenetre
        this.pack();
    }

    public static void main(String[] args) {
        ExInterface myInterface = new ExInterface();
        myInterface.setVisible(true);
    }
}

// Cette classe herite de canvas pour redéfinir la
// methode paint. La vue est codee dans cette classe
class Dessin extends Canvas {

    String figure;
    Color couleur;
    String text;

    // le constructeur de la classe
    public Dessin() {
        figure = "Carre";
        couleur = Color.blue;
        text = "Hello World";
        this.setBackground(Color.white);
    }

    public void paint(Graphics g) {
        // un rectangle noir encadre le composant
        g.setColor(Color.black);
        g.drawRect(0, 0, this.getWidth() - 1, this.getHeight() - 1);
        // le texte est affiche
        g.drawString(text, 10, 15);

        // la couleur est choisie
        g.setColor(couleur);

        // la figure est affichee
        if (figure.equals("Carre")) {
            g.drawRect(50, 30, 100, 100);
        } else if (figure.equals("Cercle")) {
            g.drawOval(50, 30, 100, 100);
        } else {
            g.drawLine(50, 130, 100, 30);
            g.drawLine(100, 30, 150, 130);
            g.drawLine(50, 130, 150, 130);
        }
    }

    // ces methodes permettent de changer le texte,
    // la couleur ou la figure a afficher
    public void setText(String txt) {
        text = txt;
    }

    public void setCouleur(String coul) {
        if (coul.equals("Bleu")) {
            couleur = Color.blue;
        } else if (coul.equals("Rouge")) {
            couleur = Color.red;
        } else {
            couleur = Color.green;
        }
    }

    public void setFigure(String fig) {
        figure = fig;
    }

    // La redéfinition de cette methode permet de specifier
    // la taille preferree du composant
    public Dimension getPreferredSize() {
        return new Dimension(200, 200);
    }
}
```

## A.3 Contrôleurs d'événements

Le rôle des contrôleurs est d'intercepter certains événements et d'effectuer un traitement associé au type de l'événement. Un événement peut être produit par un click sur un bouton, la sélection d'un élément d'une liste, un déplacement de souris, la pression d'une touche du clavier, etc.

### A.3.1 Evénements

Un événement graphique est représenté dans le langage Java comme un objet dont la classe hérite de `java.awt.AWTEvent`. Parmi les sous-classes de `AWTEvent`, on peut citer les plus couramment utilisées :

- `ActionEvent` : Se produit lorsqu'une action est effectuée sur un composant. Ex : click sur un bouton.
- `ItemEvent` : Se produit lorsqu'une sélection a été effectuée sur un composant. Ex : cochage d'une case.
- `KeyEvent` : Se produit lorsque un événement provient du clavier. Ex : pression d'une touche.
- `MouseEvent` : Se produit lorsque un événement provient de la souris. Ex : déplacement de la souris.
- `WindowEvent` : Se produit lorsqu'une action est effectuée sur une fenêtre. Ex : click sur l'icone de fermeture d'une fenêtre.

Des méthodes sont attachées à chacune de ces classes pour avoir accès à plus de détails sur l'événement. On peut, par exemple, récupérer le composant source de l'événement, la position de la souris lors du click, etc.

### A.3.2 Interface Listener

Le contrôleur qui intercepte un certain type d'événement doit implémenter une des interfaces héritant de `java.util.EventListener`. L'interface à implémenter dépend du type d'événement à intercepter. La table A.1 présente les interfaces correspondant aux événements décrits ci-dessus.

Certaines de ces interfaces demandent qu'un grand nombre de méthodes soient implémentées (par ex, `WindowListener`). Des classes, appelées *adapter*, implémentant ces interfaces sont proposées dans l'API, pour lesquelles toutes les méthodes ont des implémentations vides. Cette facilité de programmation permet de n'implémenter que la méthode souhaitée. Par exemple, on utilisera la classe `WindowAdapter` pour implémenter un traitement à effectuer à la fermeture d'un fenêtre (méthode `windowClosing()`) sans avoir à écrire des méthodes vides pour tous les autres cas où aucun traitement n'est requis.

Après la création d'un objet contrôleur, il est nécessaire de le rattacher à un ou plusieurs composants. Le contrôleur intercepte uniquement les événements des composants auquel il est rattaché. Cette opération se fait par l'appel à une méthode du composant de la forme `add...Listener`. Par exemple, le rattachement d'un contrôleur `myActionListener` à un bouton s'écrit :

```
myButton.addActionListener(myActionListener);
```

### A.3.3 Exemple de contrôleur

Dans l'exemple donné plus loin, des traitements ont été associés aux différents composants de l'interface présentée en section A.2.3. Les contrôleurs ont été volontairement créés de manière différente pour illustrer plusieurs cas :

Table A.1 – Quelques interfaces pour contrôleur

| Contrôleur | Evénement | Méthodes à implémenter |
|---|---|---|
| `ActionListener` | `ActionEvent` | - `actionPerformed(ActionEvent)` |
| `ItemListener` | `ItemEvent` | - `itemStateChanged(ItemEvent)` |
| `KeyListener` | `KeyEvent` | - `keyPressed(KeyEvent)`<br>- `keyReleased(KeyEvent)`<br>- `keyTyped(KeyEvent)` |
| `MouseListener` | `MouseEvent` | - `mouseClicked(MouseEvent)`<br>- `mouseEntered(MouseEvent)`<br>- `mouseExited(MouseEvent)`<br>- `mousePressed(MouseEvent)`<br>- `mouseReleased(MouseEvent)` |
| `WindowListener` | `WindowEvent` | - `windowActivated(WindowEvent)`<br>- `windowClosed(WindowEvent)`<br>- `windowClosing(WindowEvent)`<br>- `windowDeactivated(WindowEvent)`<br>- `windowDeiconified(WindowEvent)`<br>- `windowIconified(WindowEvent)`<br>- `windowOpened(WindowEvent)` |

- le **contrôleur de la fenêtre** est un objet instancié à partir d'une classe `WindowControler` qui hérite de la classe `WindowAdapter`. Le seul traitement qui est précisé est de quitter le programme quand l'utilisateur clique sur le bouton de fermeture.
- le **contrôleur des boutons de choix** est utilisé sur plusieurs composants. Il traite les événements provenant des boutons `JRadioButton` et `JComboBox`. Ce contrôleur est une instance de `ChoiceControler` qui implémente l'interface `ItemListener`. Le traitement effectué est une mise à jour du dessin.
- le **contrôleur de la liste** est en fait le même objet que celui représentant la fenêtre. C'est la classe `ExControleur` qui implémente l'interface `ListSelectionListener` et met à jour le dessin à chaque fois que la sélection de la liste change.
- le **contrôleur de la zone de texte** est aussi implémenté par l'objet fenêtre. Si la touche "entrée" est pressée dans la zone de texte, la liste est mise à jour.
- le **contrôleur du bouton** a une forme un peu spéciale mais fréquemment utilisée. Il est possible, en Java, de redéfinir une méthode d'une classe pendant une instanciation. Lors de l'instanciation du contrôleur (`new ActionListener()`), la méthode `actionPerformed()` est redéfinie directement avant le point virgule de fin d'instruction. C'est le seul cas où l'instanciation d'une interface est permise si toutes ses méthodes sont implémentées.

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

public class ExControleur extends ExInterface
   implements ListSelectionListener, ActionListener {

   public ExControleur() {
      // appel au constructeur de ExInterface pour creer la
      // fenetre et ses composants
      super();

      // ajout d'un contrôleur à la fenêtre
      this.addWindowListener(new WindowControler());

      // Un contrôleur unique est créé pour les boutons
      // de choix
      ChoiceControler controleur = new ChoiceControler(dessin);
      button1.addItemListener(controleur);
      button2.addItemListener(controleur);
      button3.addItemListener(controleur);
      couleur.addItemListener(controleur);

      // le role de contrôleur pour la liste et le champ de
      // texte est assure par l'objet courant ExControleur
      textList.addListSelectionListener(this);
      text.addActionListener(this);

      // Le controleur du bouton est ecrit directement dans
      // la classe
      ajout.addActionListener(new ActionListener() {

         public void actionPerformed(ActionEvent e) {
            String str = text.getText();
            if (str.length() > 0) {
               listData.addElement(text.getText());
               text.setText("");
               textList.setListData(listData);
            }
         }
      });
   }

   // Cette methode est appelee si la selection de la liste
   // change
   public void valueChanged(ListSelectionEvent e) {
      String selected = (String) textList.getSelectedValue();
      // Le texte affiche est mis a jour en fonction de la
      // nouvelle selection
      if (selected != null) {
         dessin.setText(selected);
      } else {
         dessin.setText("");
      }
      dessin.repaint();
   }

   // Cette methode est appelee si la touche entree est
   // pressee dans le champ de texte
   public void actionPerformed(ActionEvent e) {
      String str = text.getText();
      // Le texte du champ est ajoute à la liste des textes
      if (str.length() > 0) {
         listData.addElement(text.getText());
         text.setText("");
         textList.setListData(listData);
      }
   }

   // Le main doit etre redefinie pour creer un objet de la
   // classe ExControleur et non ExInterface
   public static void main(String[] args) {
      ExControleur myControleur = new ExControleur();
      myControleur.setVisible(true);
   }
}

// Une classe de controleur de fenetre
class WindowControler extends WindowAdapter {

   public void windowClosing(WindowEvent e) {
      System.exit(0);
   }
}

// Une classe pour le controle des boutons de choix.
// Cette classe est utilisee pour les JRadioButton et}
// le JComboBox
class ChoiceControler implements ItemListener {

   Dessin dessin;

   // Il est necessaire que le dessin soit passe en parametre
   // pour sa mise a jour apres un evenement.
```

```
    public ChoiceControler(Dessin d) {
        dessin = d;
    }

    // Cette méthode est appelee si la selection d'un bouton
    // change
    public void itemStateChanged(ItemEvent e) {
        // On teste que l'evenement est bien une selection
        if (e.getStateChange() == ItemEvent.SELECTED) {
            // traitement à effectuer si l'evenement s'est
            // produit sur un JRadioButton
            if (e.getSource() instanceof JRadioButton) {
                dessin.setFigure(((JRadioButton) e.getItem()).getText());
            } // traitement a effectuer si l'evenement s'est
            // produit sur un JComboBox
            else if (e.getSource() instanceof JComboBox) {
                dessin.setCouleur((String) e.getItem());
            } else {
                System.err.println("Erreur - composant non valide");
            }
            dessin.repaint();
        }
    }
}
```

# Annexe B

# Diagramme de classes UML

UML (*Unified Modeling Language*) a été créé en 1997 pour être le langage standard de modélisation orienté-objet. UML contient différents diagrammes utilisés pour décrire de nombreux aspects du logiciel. Dans le cadre de ce cours, nous utiliserons uniquement le **diagramme de classes** [1].

Le diagramme de classes représente la structure statique du logiciel. Il décrit l'ensemble des classes qui sont utilisées ainsi que leurs associations. Il est inspiré des diagrammes Entité-Relation utilisés en modélisation de bases de données, en y ajoutant les aspects opérationnels (les méthodes) et quelques subtilités sémantiques (la composition, par exemple).

## B.1 Représentation des classes et interfaces

### B.1.1 Les classes

En UML, une classe est au minimum décrite par un nom. Graphiquement, elle est représentée par un rectangle, éventuellement divisé en 3 parties : son nom, ses attributs et ses opérations. On appelle **membres** de la classe ses attributs et méthodes (ou opérations).

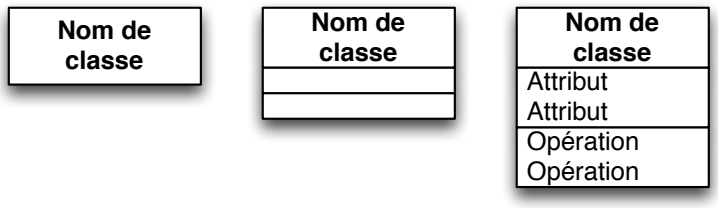



FIGURE B.1 – Représentations UML possible d'une classe, avec ou sans les membres

### B.1.2 Les membres de classe

Chaque attribut est décrit au moins par un `nom` (unique dans une même classe) et par un `type de données`.

Une opération, ou méthode, est décrite au moins par un `nom`, par un ensemble d'`arguments` nécessaires à son invocation et par un `type de retour`. Chaque `argument` est décrit par un nom et un type de données.

1. le langage UML pourra être étudié en profondeur lors du cours Analyse et Conception de Systèmes Informatiques de l'axe `Ingénierie des Systèmes Informatiques`

Un niveau de **visibilité** est également attribué à chaque membre. La visibilité d'un membre d'une classe définit quelles autres classes y ont accès (en terme de lecture/écriture). UML utilise 3 niveaux de visibilité :

- `public` (noté par +), le membre est visible par toutes les classes
- `privé` (noté par -), le membre n'est visible par aucune classe sauf celle qui le contient
- `protégé` (noté par #), le membre est visible par toutes les sous-classes de celle qui le contient (cette visibilité est expliquée ultérieurement)

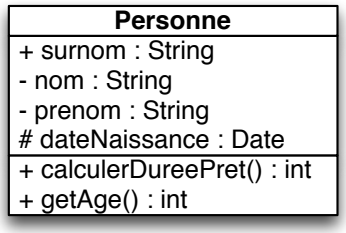



FIGURE B.2 – Exemple de classe `Personne` avec attributs et opérations

La figure B.2 est un exemple d'une classe `Personne` avec toutes les possibilités de visibilité.

### B.1.3 Les classes abstraites

En UML, le mot-clé `{abstrait}` (ou `{abstract}`) est accolé aux classes et méthodes abstraites. Une autre manière souvent usitée de représenter ces méthodes ou classes est d'écrire leur nom en italique. Les *attributs et méthodes de classe* sont soulignés. La classe `Personne` décrite dans la figure ci-dessous montre un exemple d'attribut et de méthode de classe.

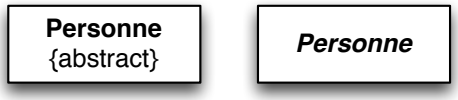



FIGURE B.3 – Deux représentations possibles pour une classe abstraite `Personne`

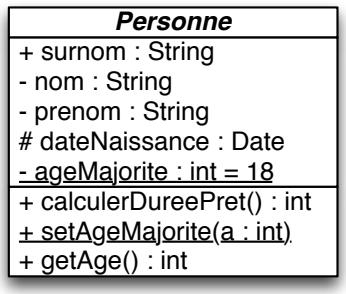



FIGURE B.4 – Exemple d'un attribut et d'une méthode de classe pour la classe `Personne`

### B.1.4 Les interfaces

En UML, une interface est décrite par le mot-clé «`interface`» dans le bloc d'entête, comme présenté dans la figure B.5.


FIGURE B.5 – Exemple d'interface

## B.2 Les relations

En conception orientée objet, les relations englobent notamment les relations d'héritage et de réalisation d'interface.

### B.2.1 L'héritage

En UML, l'héritage se représente par une flèche à la pointe creuse. La figure B.6 décrit deux classes `Super-classe` et `Sous-classe`. La classe `Sous-classe` **hérite** de la classe `Super-classe`.

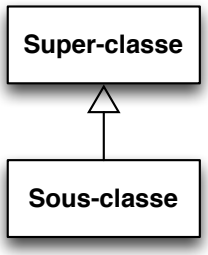



FIGURE B.6 – Exemple d'héritage

### B.2.2 La réalisation

La réalisation d'une interface par une classe se représente par une flèche pointillée à pointe creuse, comme illustré dans la figure B.7.

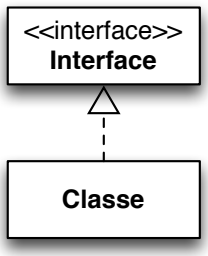



FIGURE B.7 – Exemple de réalisation de l'interface `Interface` par la classe `Classe`

## B.3 Les associations

Certaines relations entre classes d'un même diagramme sont représentées en UML sous la forme d'**associations**. Le plus souvent une association ne relie que deux classes. Une association peut être identifiée par un nom et chacune de ses extrémités définit le nombre d'instances

des classes reliées qui sont impliquées dans cette association. On appelle **multiplicité** ce nombre d'instances qui peut prendre les valeurs suivantes :

| Multiplicité | Interprétation |
|---|---|
| 1 | un et un seul |
| 0..1 | zéro ou un |
| N | exactement N |
| M..N | de M à N |
| * | zéro ou plus |
| 0..* | zéro ou plus |
| 1..* | un ou plus |

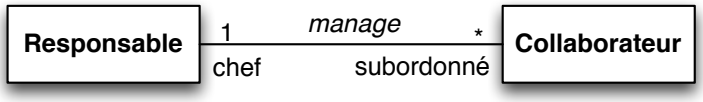



FIGURE B.8 – Exemple d'association `manage`

L'exemple de la figure B.8 décrit une association `manage` entre la classe `Responsable` et la classe `Collaborateur`. Un responsable gère plusieurs collaborateurs, ses `subordonnés`. Un collaborateur est géré par un seul responsable, son `chef`. Les éléments `subordonné` et `chef` sont des *rôles* d'association.

### B.3.1 Direction des associations

Les associations peuvent être *dirigées*, ce qui contraint la visibilité et la navigation dans le modèle. La direction se représente par une flèche classique. Par défaut, s'il n'y a pas de flèche, l'association est bidirectionnelle (comme s'il y avait une flèche à chaque extrémité de l'association).

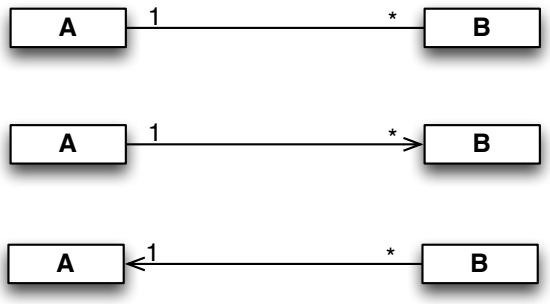



FIGURE B.9 – Exemples d'associations dirigées

La figure B.10 présente des exemples de directions, dont voici les interprétations. La première ligne signifie que `A` connaît tous les `B` auxquels elle est associée, et réciproquement, un `B` connaît le `A` auquel il est associé. La deuxième ligne signifie que seul le `A` connaît les `B` auxquels il est associé, mais pas l'inverse. Finalement, dans la troisième ligne, un `B` connaît le `A` auquel il est associé, mais pas l'inverse. En fait, ceci va impliquer la présence ou non d'un attribut `a` de type `A` dans la classe `B` ou `b` de type `B` dans la classe `A` en fonction de la direction. Par exemple, pour la deuxième ligne, `A` possède une liste d'objet de type `B` mais `B` ne possède pas d'attribut de type `A`.

### B.3.2 Agrégation et composition

Deux sous-types d'associations permettent de préciser un sens particulier à ces relations : l'agrégation et la composition. Elles peuvent également être dirigées.

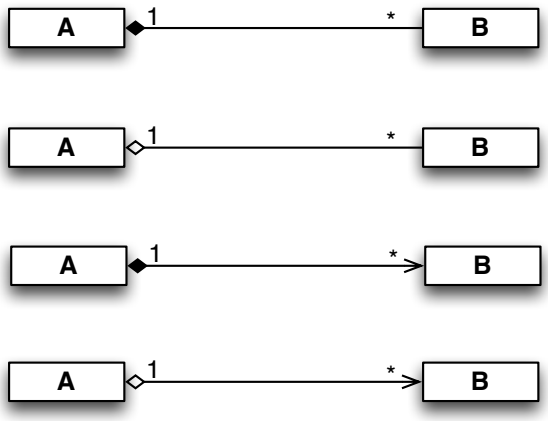



FIGURE B.10 – Exemples de compositions et d'agrégations.

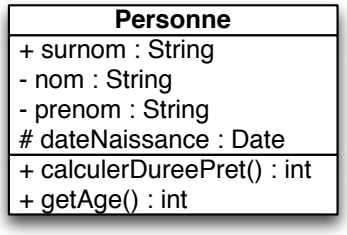
L'**agrégation** est une association avec relation de subordination, souvent nommée `possède` représentée par un trait reliant les deux classes et dont l'origine se distingue de l'autre extrémité (la classe subordonnée) par un *losange creux*. Une des classes "regroupe" d'autres classes. On peut dire que l'objet `A` utilise ou possède une instance de la classe `B`.

La **composition** est une association liant le cycle de vie des deux classes concernées. Une association de composition s'interprète comme une classe est composée de un ou plusieurs élément de l'autre classe. Elle est représentée par un trait reliant les deux classes et dont l'origine se distingue de l'autre extrémité (la classe composant) par un *losange plein*. On peut dire que l'objet `A` est composé instance de la classe `B`, et donc si l'objet de type `A` est détruit, les objets de type `B` qui le composent sont également détruit. Ce sera également souvent les objets de type `A` qui créeront les objets de type `B`.

## B.4 Correspondance UML-Java

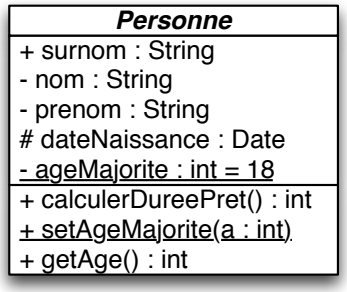
Java permet de programmer tout modèle sous forme de diagramme de classe UML tel que présenté ci-dessus. Voici quelques exemples de correspondance entre le modèle UML et le codage Java.

### B.4.1 Classes et membres



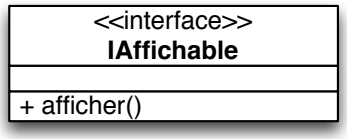

```java
public class Personne {
  public String surnom;
  private String prenom;
  private String nom;
  protected Date dateNaissance;
  public int calculerDureePret() {...}
  public int getAge() {...}
}
```

### B.4.2 Classes abstraites



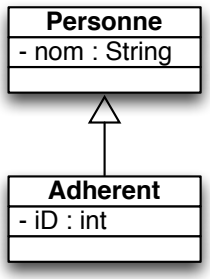

```java
public abstract class Personne {
  public String surnom;
  private String prenom;
  private String nom;
  protected Date dateNaissance;
  private static int ageMajorite = 18;
  public int calculerDureePret() {...}
  public static void setAgeMajorite(int a) {...}
  public int getAge() {...}
}
```

### B.4.3 Interfaces



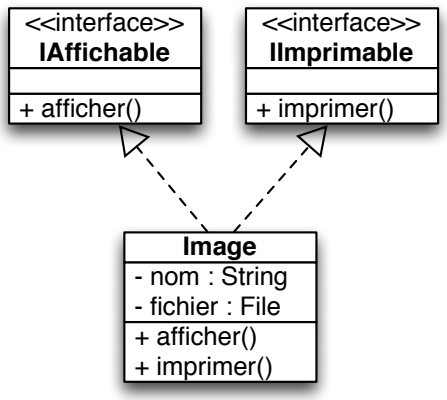

```java
interface IAffichable {
  void afficher();
}
```

### B.4.4 Héritage



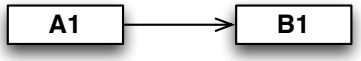

```java
public class Adherent extends Personne {
  private int iD;
}
```

### B.4.5 Réalisation

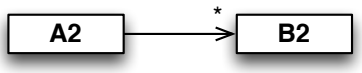



```java
public class Image implements IAffichable, IImprimable {
  private String nom;
  private File fichier;
  public void afficher(){...}
  public void imprimer(){...}
}
```

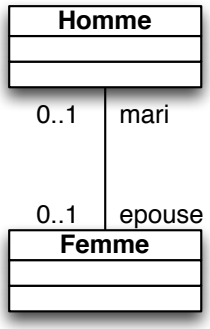
### B.4.6 Associations




```java
public class A1 {
  private B1 b1;
  ...
}
```

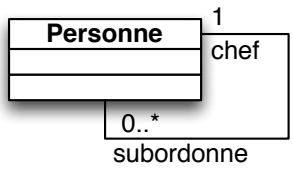



```java
public class A2 {
  private ArrayList<B2> b2s;
  ...
}
```




```java
public class Homme {
  private Femme epouse;
  ...
}
```

```java
public class Femme {
  private Homme mari;
  ...
}
```




```java
public class Personne {
  private ArrayList<Personne> subordonnes;
  private Personne chef;
  ...
}
```

# Références


- **Livres :**
  - *Programmer en Java, 7e Edition*, Claude Delannoy, Eyrolles, 2011
  - *The Java Tutorial : A Short Course on the Basics, 4th Edition*, Collectif, Prentice Hall, 2006
  - *Effective Java, 2nd Edition*, Joshua Bloch, Prentice Hall, 2008
  - *Java in a nutshell, 5th edition*, David Flanagan, O'Reilly, 2005
- **Sites web :**
  - Le site officiel Java, `http://www.oracle.com/technetwork/java/index.html`
  - Le tutorial Java, `http://docs.oracle.com/javase/tutorial/`
  - l'API du JDK 1.7, `http://docs.oracle.com/javase/7/docs/api/`
  - Un site (français) de développeurs, `http://www.javafr.com/`
  - Le site JavaWorld, `http://www.javaworld.com`
- **Quelques autres liens :**
  - des liens en rapport avec Java, `http://www.javamug.org/mainpages/Java.html`
  - Tutoriaux jGuru, `http://java.sun.com/developer/onlineTraining/`
  - plein d'autres cours, `http://java.developpez.com/cours/`