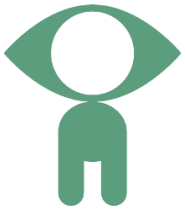
Aide mémoire UML & Java
1ère partie : Introduction

marc.lemaire@u-cergy.fr

12 septembre 2019



DÉPARTEMENT
Sciences Informatiques

Table des matières

1. Ce paragraphe présente la notion de polymorphisme **dynamique** en relation avec la notion d'héritage.
2. *Remarque* : les éléments de ce paragraphe sont extraits de [Boo00].
3. A ne pas confondre avec la **classe** Collections (avec "s") qui propose un ensemble de méthodes *static* permettant d'implémenter une version synchronisée de n'importe quelle collection (dans le cas d'accès concurrents par plusieurs Threads). Elle permet également de créer des versions en lecture seule et un ensemble de manipulations (`sort( )`, `binarySearch( )`, `fill( )`, `max( )`, `min( )`, `reverse( )`, ...).

Chapitre 1

Généralités

1.1 Notations utilisées

Les exemples de code sont en police de caractères à pas fixe :

```
System.out.println("hello world");
```

Remarque : Il arrive parfois qu'une coupure dans une ligne de programme soit nécessaire :

```
  System.out.println("ligne d'instructions trop longue pour te-
nir sur une seule ligne et qui déborde sur la ligne suivante");
```

Les commentaires de code et de description des classes et méthodes sont en italique

```
// commentaires
```

Les éléments principaux à retenir sont signalés par le pictogramme : 

1.2 Historique et introduction

Bref historique

Le langage de programmation JAVA est issue des recherches menées par SUN MICROSYTEMS [1], dans le but de concevoir un langage de haut niveau, facile à apprendre, portable et ouvert. La première version du langage est apparue en 1995, la version 1.2 (appelée Java 2) en 1998 (suivie des versions 1.3 à 1.5 toujours appelée Java 2), les versions ultérieures sont dénommée 6, 7, 8, etc. Les versions actuelles sont appelée Java SE [2] 12 et Java EE 8 (J2E 8).

Introduction

Le langage JAVA est un langage orienté objet (comme tous les langages de programmation modernes), qui exploite abondamment la syntaxe des langages C et C++ et est de ce fait relativement facile à apprendre si on connaît déjà ces langages. Il est fourni avec un très grand nombre de classes (composants logiciels) permettant des réalisations rapides.

1. En 2009, Sun a été racheté par Oracle.
2. Le présent document traite de Java SE (Standard Edition) mais n'aborde pas Java EE (Entreprise Edition)

Sa notoriété initiale est lié au succès d'Internet (des pages Web) par l'intermédiaire des Applets - même si cet aspect est devenu aujourd'hui extraimement marginal. Java est un langage à part entière qui permet de réaliser des applications complexes indépendantes du Web. Les Applets permettent d'insérer des programmes à l'intérieur d'une page Web.

1.3 Les spécificités de Java : la JVM

JAVA a été conçu pour être un langage portable multi-plate-formes. Il est en effet possible (contrairement à la plupart des langages tels que C ou C++) d'exécuter un même programme JAVA sur WINDOWS, LINUX, MAC OS ou SOLARIS sans recompilation (cette portabilité est résumée dans la devise de SUN concernant JAVA : *"write once, run anywhere"*). Cette simplicité est due à l'existence d'une couche logicielle intermédiaire entre le code compilé (ou ***byte-code***) et le système d'exploitation (*Operating System*). Cette couche logicielle est appelée machine virtuelle Java ou **JVM** (***Java Virtual Machine***). De ce fait l'ensemble des navigateurs Web actuels intègrent une machine virtuelle afin de pouvoir exécuter les Applets. Les programmes Java sont donc partiellement compilés et partiellement interprétés.

Les phases de compilation et d'exécution diffèrent donc sensiblement du langage C puisqu'aucun fichier exécutable (de type ".exe" sous WINDOWS) n'est généré.

Remarque : en JAVA, l'édition de liens est dynamique : la JVM charge les différentes classes dont l'application a besoin à l'exécution.

1.4 Compilation et exécution (les commandes javac et java)

Important : pour des précisions sur l'installation du JDK sous WINDOWS, consulter le § 13 page 90.

Un premier exemple de programme Java

exemple 1 : "hello world" simple - fichier Essai.java

```
/* Exemple de code Java */
public class Essai {
    public static void main (String[] args) {
        System.out.println("hello world");
    }
}
```

Important : le nom du fichier doit être le même que le nom de la classe publique (ici Essai) en respectant la casse (les minuscules et les majuscules).

La **compilation** de ce fichier se fera sur la ligne de commande (dans une fenêtre Terminal) par appel du compilateur javac (*java compiler*) sur le fichier à compiler, en respectant la casse par :

```
javac Essai.java
```

L'**exécution** de ce fichier se fera par appel de la machine virtuelle java sur la classe considérée (ici Essai) en respectant la casse : cette classe doit être **public** et doit disposer d'une méthode **main**, point d'entrée du programme.

```
java Essai
```

exemple 2 : "hello world" avec paquetage - fichier EssaiPaq.java

```
/* Exemple de code Java avec paquetage */
package test;
public class EssaiPaq {
    public static void main (String[] args) {
        System.out.println("hello world - with package");
    }
}
```

La compilation se fera par :

```
javac EssaiPaq.java
```

L'exécution se fera par le lancement de la machine virtuelle sur la classe considérée en précisant son paquetage d'appartenance (java paquetage.Classe) et en respectant la casse :

```
java test.EssaiPaq
```

Quelques remarques concernant les exemples précédents :

- ▷ Tout code doit appartenir à une classe (cf. § 3.2 page 19 pour la définition de cette notion).
- ▷ La méthode main a une signature unique (`public static void main (String[] args)`), contrairement au langage C où plusieurs formes sont tolérées.
- ▷ Dans le cas de plusieurs classes à compiler simultanément, on peut utiliser (et on doit le faire si ces classes sont interdépendantes) la commande suivante :

```
javac *.java
```

javac

l'option -d (*directory*) permet d'indiquer le nom du répertoire racine de la hiérarchie des classes pour les fichiers .class créés.

1.5 Ligne de commande ou interface graphique ?

Concernant les possibilités de saisie et d'affichage en ligne de commande (scanf( ) et printf( ) en C ou << et >> en C++), le choix en JAVA a été de privilégier la programmation d'interfaces graphiques actuelles (fenêtrées) sans fournir au départ [3] de solution simple pour la saisie (scanf( ) et <<) et en offrant par contre une grande simplicité pour l'affichage. Pour cette raison, nous n'aborderons pas les aspects de saisie au clavier dans un Terminal (obsolète) en privilégiant l'apprentissage de la programmation d'interfaces graphiques. De ce fait, les premiers exemples ne seront pas interactifs et les informations en entrée seront codées (très temporairement) "en dur" dans les premiers programmes de tests (le § . 10.9 page 82 présente une classe permettant la saisie d'informations depuis une fenêtre terminal)

Par contre, contrairement à la fonction printf( ) du langage C, l'affichage en ligne de commande est extrêmement simple et nous l'utiliserons fréquemment y compris à des fins de recherche d'erreurs (*debug* ou analyse) en mode graphique.

3. depuis la version 1.5 du langage, la classe java.util.Scanner offre une solution simple à la saisie interactive en mode console.

Quelques exemples d'utilisation de System.out.println()

Pour afficher une chaîne de caractères à l'écran il suffit de la passer à la méthode (fonction) println( ) :

```
System.out.println("mon texte");
```

Pour afficher un texte sans retour à la ligne on utilise print( ) :

```
System.out.print("mon texte");
```

On peut aussi comme en C indiquer explicitement un retour à la ligne par \n :

```
System.out.println("première ligne \n deuxième ligne");
```

Pour afficher le contenu d'une variable, il suffit de la passer à la méthode (fonction) d'affichage :

```
int i = 3;
System.out.println(i);
```

Pour afficher texte et variable(s) simultanément, il suffit d'utiliser l'opérateur de concaténation "+" autant de fois que nécessaire :

```
int i = 3;
System.out.println(" variable i = " + i);
```

Exemple de concaténations multiples :

```
int i = 3;
float f = 12.7f;
System.out.println(" i = " + i + " et f = " + f );
```

Chapitre 2

Éléments de syntaxe

Remarque : Il existe de grandes similitudes de syntaxe entre JAVA et le langage C, nous rappelons brièvement ci-après la syntaxe usuelle.

2.1 Types de données (les types de base)

Comme pour tout langage de programmation, on utilise en JAVA des variables : celles-ci sont repérées par leur nom (on rappelle que le nom d'une variable est sensible à la casse) et appartiennent à un type (entier, réel...). Il existe comme en C et C++, des types de base prédéfinis qui sont des types simples, il existe quelque différences sensibles avec le langage C, en particulier l'apparition d'un vrai type booléen (non assimilable à 0 et 1) et d'un vrai type caractère (codé au format **unicode** et non assimilable à un entier), il n'existe pas de type *unsigned*. Certaines variables ont une valeur par défaut. [1]

type	contenu	taille	min	max	défaut
`boolean`	booléen	1 bit	`false`	`true`	`false`
`char`	caractère unicode [2]	2 octets (16 bits)	`\u0000`	`\uFFFF`	`\u0000`
`byte`	entier signé en complément à 2	1 octet (8 bits)	`-128`	`+127`	`0`
`short`	entier signé en complément à 2	2 octets (16 bits)	`-32768`	`+32767`	`0`
`int`	entier signé en complément à 2	4 octets (32 bits)	`-2 147 483 648`	`+2 147 483 647`	`0`
`long`	entier signé en complément à 2	8 octets (64 bits)	`-2^63`	`2^63 - 1`	`0L`
`float`	réel simple précision : IEEE 754	4 octets (32 bits)	± m^e (m<2^24 et -144<e<104)		`0.0f`
`double`	réel double précision : IEEE 754	8 octets (64 bits)	±m^e (m<2^53 et -1045<e<1000)		`0.0`

TABLE 2.1 – Les types prédéfinis

1. Plus précisément, les variables d'instance ou de classe ont les valeurs par défaut mentionnées dans le tableau 2.1 mais les variables locales doivent impérativement être initialisées car elles n'ont pas de valeur par défaut et le compilateur empêche l'utilisation de variables indéterminées.

2.1.1 Déclaration

Toute variable doit être déclarée avant d'être utilisée. Cette déclaration se fait en mentionnant son type :

```
int i;
char c;
float f;
```

2.1.2 Initialisation

Une variable peut être initialisée à une valeur de départ différente de sa valeur par défaut au moment de sa déclaration :

```
int i = 3;
char c = '\u0041'; // ou char c = 'A';
float f = 12.7f;
```

on peut aussi réaliser déclaration et initialisation en deux instructions :

```
int i;
i = 3;
```

 N.B. : A la différence des attributs (variables d'instance) qui ont des valeurs par défaut (cf. § 2.1 page précédente), les variables locales (i.e. de méthode) **doivent** être initialisées avant d'être utilisées (dans le cas contraire, une erreur de compilation est générée).

2.1.3 Affectation

Une variable peut être modifiée à tout moment (dans son domaine de visibilité) par :

```
i = 7;
c = 'B';
f = 35.4f;
```

Valeur d'affectation pour les caractères :

```
'a' // caractère a
'\n' // caractère de retour à la ligne ('\' est le caractère d'échappe-
ment).
'\u263a' // caractère unicode 263A
```

Valeurs d'affectation pour les entiers :

Sauf indication contraire, les valeurs entières manipulées directement (i.e. sans variable associée) dans un programme sont de type **int** :

```
12 // entier décimal 12 (de type int)
012 // entier octal 12 (de type int), soit 10 en décimal
0x12 // entier hexadécimal 12 (de type int), soit 18 en décimal
12L // entier décimal 12 (de type long)
```

N.B. : certaines conversions implicites sont invalides :

```
byte octet = 12; // erreur
```

Cette instruction provoque une erreur à la compilation, l'entier 12 (de type int) doit être converti explicitement par transtypage (ou cast) :

```
byte octet = (byte)12; // ok
```

Par contre toute conversion implicite vers un type plus important est valide :

```
long entier = 13; // ok
```

Valeurs d'affectation pour les réels :

Sauf indication contraire, les valeurs réelles manipulées directement (i.e. sans variable associée) dans un programme sont de type **double** :

```
73.9 // réel 73,9 en notation décimale (de type double)
7.39e+1 // réel 73,9 en notation scientifique (de type double)
73.9f // réel 73,9 en notation décimale (de type float)
7.39e+1F // réel 73,9 en notation scientifique (de type float)
```

N.B. : certaines conversions implicites sont invalides :

```
float reel = 73.9; // erreur
```

Cette instruction provoque une erreur à la compilation : le réel 73,9 (de type double) doit être suffixé par la lettre F (ou f) pour indiquer qu'il s'agit d'un float :

```
float reel = 73.9f; // ok
```

On peut aussi, comme dans le cas des entiers, réaliser une opération de cast (transtypage ou forçage de type explicite) :

```
float reel = (float)73.9; // ok
```

2.2 Les opérateurs

2.2.1 Présentation

JAVA utilise tous les opérateurs du C, à part les opérateurs sur les pointeurs (&, * et ->). En effet, bien que JAVA ne manipule pratiquement que des pointeurs, ces derniers ne sont pas accessibles au programmeur. En particulier, il n'y a pas de moyen de récupérer l'adresse machine d'une variable ou d'un objet[3].

3. On peut néanmoins connaître la valeur d'une référence au sein de la JVM

Il y a trois grandes catégories d'opérateurs :

- ▷ les opérateurs arithmétiques :
  `+, -, *, /, %` (modulo)
  `++` (incrémentation), `--` (décrémentation) : raccourcis syntaxiques pour `var = var+1` et `var=var-1`
  `+=, -=, *=, /=, %=` : raccourcis syntaxiques : "variable opérateur= valeur" est équivalents à "variable = variable opérateur valeur", exemple i += 3 est équivalent à i = i + 3
- ▷ les opérateurs logiques et de comparaison :
  `!` (NON logique), `&&` (ET logique), `||` (OU logique)
  `==` (égal), `!=` (différent), `<, <=, >, >=` (inférieur et supérieur, strictement ou non)
- ▷ les opérateurs de bas niveau (manipulations sur les bits) :
  `&` (ET bit à bit), `|` (OU bit à bit), `^` (XOR : OU exclusif), `~` (complément à 1)
  `<<` (décalage à gauche), `>>` (décalage arithmétique à droite avec extension du bit de signe), `>>>` (décalage logique à droite avec insertion de zéros (0) pour les bits de poids fort).

Les autres opérateurs importants sont :

- ▷ le cast : (type), ou forçage de type permettant de considérer, **le temps de l'instruction**, une variable comme appartenant à un autre type :

```
float f;
int num = 5, den = 2;
f = (float)num / (float)den;
// ici f vaudra 2.5 : en l'absence de cast, f vaudrait 2.0 !
```

- ▷ `+` : cet opérateur a une deuxième signification (en plus de l'addition mathématique) : il permet la concaténation de chaînes de caractères, cf. § 1.5 page 9 : exemple d'utilisation avec println( ), on dit que l'opérateur + est surchargé.
- ▷ `.` (point), `new, instanceof, [ ]` : qui sont détaillés plus loin

2.2.2 La priorité des opérateurs

Le tableau 2.2 présente une liste exhaustive des différents opérateurs de JAVA et leurs priorités (décroissantes) en précisant le sens de l'associativité (–> : de gauche à droite ; ou <– : de droite à gauche).

2.3 Les structures de contrôle

La plupart des éléments présentés dans ce chapitre sont communs avec le langage C.

2.3.1 Tests

Test conditionnel if (SI)

Le test de la valeur d'une variable se fait par la structure conditionnelle if :

```
if (condition) { ... }
```

exemples

Opérateurs	Associativité
( ) [ ] . ++ (postfixé) – (postfixé)	–>
+ (unaire) - (unaire) ++ (préfixé) – (préfixé) ~ (unaire) ! cast new	<–
* / %	–>
+ -	–>
+ (concaténation)	–>
<< >> >>>	–>
< <= > >= instanceof	–>
== !=	–>
&	–>
^	–>
|	–>
&&	–>
||	–>
? :	–>
= += -= *= /= %= <<= >>= >>>= &= |= ^=	<–

TABLE 2.2 – Les opérateurs de JAVA et leurs priorités

```
if ( i < 7 ) { ... } // SI i est inférieur strictement à 7
if ( i == 3 ) { ... } // SI i est égal à 3
if ( i >= 4 ) { ... } // SI i est supérieur ou égal à 4
if ( i != 5 ) { ... } // SI i est différent de 5
```

Test conditionnel if ... else (SI ... SINON)

Le test peut avoir une alternative par un else (sinon) optionnel :

```
if (condition) { ... } else { ... }
```

exemples

```
if ( i==3) { ... }
else { ... } // tous les cas où i est différent de 3
```

Remarque : une structure "if else" peut parfois être remplacée par l'opérateur conditionnel ternaire ? :

exemple :

```
if ( i==3 ) {
    var = 5;
}
else {
    var = 19;
}
```

est équivalent à

```
var = ( i==3 )? 5 : 19;
```

Remarque : dans le cas de plusieurs if / else imbriqués, il est possible d'utiliser l'idiome else if qui permet une écriture plus lisible

 **Attention** : Il ne faut pas confondre affectation (=) et comparaison (==) :

2.3.2 Boucles

Les boucles permettent de réaliser plusieurs fois un même bloc d'instructions, il en existe trois formes :

for (boucle POUR)

Remarque : l'usage des boucles "for" est à réserver aux traitements dont on connaît à l'avance (en entrant dans la structure for) le nombre d'itérations.

syntaxe :

```
for( traitement initial ; condition ; itération ) { ... }
```

la zone "traitement initial" est exécutée **une seule fois** avant toutes les boucles et est généralement le lieu de l'initialisation de la variable de boucle.

la zone "condition" est testée **avant** chaque itération de la boucle, de ce fait une boucle for peut éventuellement ne jamais être exécutée ...

la zone "itération" (en général réservée à l'incrémentation d'indice de la variable de boucle) est exécutée **à la fin** de chaque itération de la boucle.

exemple :

```
for ( i = 0 ; i < 10 ; i++ ) { ... }
```

Cas particulier en Java : il est possible de **déclarer** la variable de boucle à **l'intérieur** de la structure for :

```
for (int i=0 ; i < 10 ; i++) { ... }
```

le domaine de visibilité de i, dans ce cas est la boucle for.

 **Attention :** Lorsqu'une boucle ou un test ne contient qu'une seule instruction, il est possible d'omettre les accolades. Il est toutefois recommandé de les conserver car le point virgule (;), lorsqu'il ne constitue pas le délimiteur de fin de ligne correspond à une instruction vide (le NOP de l'assembleur) ainsi la boucle suivante exécute 10 fois l'instruction vide puis affiche (une seule fois) le message "hello world" ... :

```
for (int i=0 ; i < 10 ; i++);
   System.out.println("hello world");
```

do ... while (boucle FAIRE ... TANT QUE)

Le bloc d'instructions délimité par do ... while sera **au moins exécuté une fois**.

Syntaxe :

```
do { ... } while (condition); // attention au ";" final
```

exemple :

```
do { ... }
while ( i < 10 ); // i doit évoluer dans le bloc d'instructions si-
non la boucle est infinie...
```

while (boucle TANT QUE)

La boucle "while" est à réserver aux cas de figures où on ne connaît pas à l'avance le nombre d'itérations de boucle à réaliser. Le bloc d'instruction peut ne jamais être exécuté (si la condition est fausse dès le départ).

Syntaxe :

```
while ( condition ) { ... }
```

exemple

```
while ( i < 10 ) { ... } // i doit évoluer dans le bloc d'instruc-
tions sinon la boucle est infinie...
```

Interrompre une boucle

Quelle que soit la structure de contrôle retenue, il est possible d'interrompre partiellement (`continue`) ou définitivement (`break`) une boucle.

le mot-clef **continue** permet de passer directement à l'itération suivante de la boucle et est en général utilisé dans une boucle for.

le mot-clef **break** permet d'interrompre définitivement une boucle et de sortir du bloc d'instructions (l'exécution reprendra juste après la boucle).

2.3.3 Sélection

La structure de contrôle "switch case" permet de réaliser des sélections multiples en fonction de la valeur d'une expression de type simple. Elle peut dans certains cas remplacer avantageusement des tests imbriqués [4].

Syntaxe :

```
switch ( var ) { // var peut être une expression de type simple
    case val1 :
    ...
    break;
    case val2 :
    ...
    break;
    default :
    ...
    break; // ce dernier break est optionnel
}
```

Exemple :

```
switch ( var ) { // ici on suppose var de type entier
    case 3 : // si var == 3
    ...
    break;
```

4. Remarque : n'est présentée ici que la forme la plus commune, d'autres alternatives telles que la succession de case ou l'omission volontaire de break sont également possibles.

```
        case 17 : // si var == 17
        ...
        break;
        default : // toute autre valeur de var
        ...
        break; // ce dernier break est optionnel
    }
```

Remarque : l'expression associée à une instruction switch doit avoir une valeur de type byte, char, short ou int.

2.4 Les autres éléments de base

2.4.1 Les commentaires

Il existe trois types de commentaires en JAVA (les deux premiers sont communs à d'autres langages (C, C++,...), la troisième forme fait l'objet d'une présentation détaillée au § 10.2 page 79)

- ▷ le commentaire sur plusieurs lignes
  si on souhaite mettre en commentaire plusieurs lignes, il suffit (comme en C) de les encadrer par les délimiteurs /* et */

```
/*
première ligne de commentaire
deuxième ligne de commentaire
*/
```

- ▷ le commentaire en fin de ligne
  le commentaire en fin de ligne est indiqué par la double barre de division //

```
int i = 0; // commentaire
```

- ▷ le commentaire pour le générateur automatique de la documentation au format HTML (cf § 10.2 page 79).

```
/**
 * commentaire pour le documenteur Javadoc
 * @author ml
 * @version 1.5
 */
```

2.4.2 Les mots du langage

Le tableau 2.3 présente l'ensemble des mots réservés du langage [5].

2.4.3 Conventions d'écriture

Voir chapitre 3 page 19, pour une description détaillée des notions abordées.

Paquetages

Les noms des paquetages sont en minuscules (ex. : ceci.est.un.paquetage).

5. Depuis le JDK 1.4, le mot-clef "assert" complète ce tableau.

abstract	boolean	break	byte	case
catch	char	class	const	continue
default	do	double	else	extends
final	finally	float	for	goto
if	implements	import	instanceof	int
interface	long	native	new	package
private	protected	public	return	short
static	super	switch	synchronized	this
throw	throws	transient	try	void
volatile	while			

TABLE 2.3 – les mots-clés du langage JAVA

Classes

Les noms des classes commencent par une majuscule (ex. : CeciEstUneClasse).

Attributs et méthodes

Les noms des attributs et des méthodes commencent par une minuscule (ex. : ceciEstUnAttribut ; ceciEstUneMethode( )).

Les méthodes accesseurs suivent les conventions de noms suivantes (pour un attribut xxx) :

1. Cas général (*getter* et *setter*)

```
// <type> désigne l'un des types de base ou une classe ou une interface
<type> getXxx()
setXxx(<type> <valeur>)
```

2. Cas d'un attribut booléen

```
boolean isXxx()
setXxx(boolean b)
```

3. Cas d'une collection (en complément des deux méthodes du cas général)

```
getXxx(int index)
setXxx(int index, <type élément> <valeur élément>)
```

Constantes

Les constantes sont indiquées en majuscules (ex. : CECI_EST_UNE_CONSTANTE).

Chapitre 3

Programmation Orientée Objet (P.O.O.)

3.1 Les concepts d'objet et d'encapsulation

Le langage C est un langage structuré fonctionnelement : en programmation structurée fonctionnelle, un programme est formé de la réunion de différentes fonctions et de différentes structures de données généralement indépendantes de ces fonctions.

Le langage Java (comme le C++) est un langage orienté objet. En Programmation Orientée Objet (P.O.O.), les fonctions, appelées **méthodes**, et les structures de données, appelées **attributs**, sont regroupées dans une unique entité logicielle. Ce concept de programmation permet d'optimiser les aspects modulaires de composants logiciels en vue de leurs réutilisation par des tiers. La P.O.O. permet une encapsulation des données, de sorte qu'il n'est pas possible d'agir directement sur les données d'un objet : il est nécessaire de passer par ses méthodes, qui jouent ainsi le rôle d'interface obligatoire.

Le grand mérite de l'encapsulation est que, vu de l'extérieur, un objet se caractérise uniquement par les spécifications de ses méthodes, la manière dont sont réellement implantées les données étant sans importance (les détails concrets d'implémentation sont cachés).

3.2 Classe

Une classe est la définition (le "moule" ou "modèle") du composant logiciel. Elle permet de décrire les données hétérogènes (c'est une extension de la notion de structure en C) mais également les fonctions associées à ces variables, dans une seule entité logicielle. Une **classe** définit un nouveau type de données élaboré réunissant à la fois des données (les **attributs**) et les fonctions liées à ces données (les **méthodes**).

 *A retenir* : Un objet est une **instance** d'une classe. Alors qu'une classe est une "déclaration d'intentions", un objet est une réalisation concrète, un exemplaire de cette classe : il existe en mémoire.

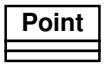
Une classe est définie par deux composantes :

- ▷ Une composante statique, définissant la **structure de données** commune à tous les objets appartenant à la classe, elle est composée d'un ensemble d'**attributs**.
- ▷ Une composante dynamique, définissant l'ensemble des **services** pouvant être demandés aux instances, ces services prennent le nom de **méthodes**.

Important : seule la structure est commune, les valeurs des attributs étant propre à chaque objet. En revanche, les méthodes sont communes à l'ensemble des objets d'une même classe.

Important : en Java, tout code appartient nécessairement à une classe, mot-clef **class**.

Exemple : nous définissons une classe Point permettant de représenter les points du plan.



 N.B. : par convention, le nom d'une classe commence toujours par une majuscule.

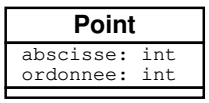
3.3 Attribut

Première composante d'une classe, les attributs[1] permettent de décrire la structure de donnée (cf. "struct" en C). Les attributs peuvent être soit d'un des types de base (cf. tableau 2.1 page 10), soit d'un autre type structuré c'est-à-dire d'une autre classe.

Exemple d'attribut :

```
int age; // attribut de type de base int
String nom; // attribut de type com-
posé, ici de la classe String (chaîne de caractères)
```

Attributs de notre classe Point (un point est défini par deux attributs entiers : son abscisse et son ordonnée).

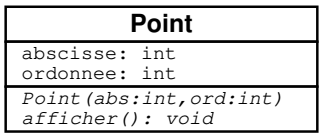


3.4 Constructeur et méthode

Les méthodes[2] sont l'interface qui permet de manipuler les attributs, on peut les classer en deux catégories : les constructeurs et les autres méthodes.

- ▷ Les **constructeurs** sont des méthodes particulières qui sont appelée **une seule fois** au moment de la construction de l'objet (l'instanciation). Le but principal d'un constructeur est de permettre l'**initialisation** des attributs.
- ▷ les **autres méthodes** peuvent être appelées depuis n'importe quel endroit du code et à tout moment. Elles s'appliquent à un objet précis (instance de la classe considérée)[3].

Nous rajoutons un constructeur et une méthode à notre classe Point :



 *A retenir* : un constructeur porte obligatoirement le nom de la classe (en respectant la casse, il commence donc par une majuscule), on peut lui passer des paramètres, **on ne précise pas son type de retour** (et surtout pas "void"!!), en effet l'appel d'un constructeur est obligatoirement employé en combinaison avec l'opérateur ***new*** qui renverra une référence sur l'objet créé et initialisé par le constructeur spécifié.

Remarque : Par convention, les noms des autres méthodes commencent par une **minuscule**, on peut leur passer des paramètres et elles peuvent retourner tout type de valeur de retour.

Exemple de la classe Point (version 0)

1. Les attributs sont parfois désignés sous les dénominations "propriété", "état", "variable d'instance" ou "donnée membre".
2. Les méthodes sont parfois désignées sous les dénominations "comportement" , "opération" ou "fonction".
3. Les méthodes de classes sont traitées au § 5.1.1 page 31

```
/*
programme Java d'exemple de la classe Point (version 0)
*/
class Point {
    int abscisse;
    int ordonnee;
    // un constructeur
    Point (int abs, int ord) {
        abscisse = abs;
        ordonnee = ord;
    }
    // une méthode d'affichage
    // remarque : la concaténation des éléments se fait par l'opéra-
teur +
    void afficher() {
        System.out.println("je suis un point d'abscisse : " + abs-
cisse + " et d'ordonnée : " + ordonnee);
    }
} // fin de la classe Point (version 0)
```

3.5 Référence, instance, objet

Un objet est une instance d'une classe, on le crée par l'opérateur **new** qui alloue la mémoire nécessaire pour cet objet (cf. fonction malloc( ) du C) et l'appel du constructeur spécifié. Pour pouvoir accéder à un objet, il faut que celui-ci soit référencé par une variable particulière appelée **référence** (l'équivalent d'un pointeur en C). Un même objet peut être affecté à plusieurs références, par contre l'inverse n'est pas vrai (une référence ne peut désigner plusieurs objets simultanément). Néanmoins, une même variable référence peut désigner successivement des objets différents au cours de l'exécution du programme.

Exemple de **déclaration** d'une référence p1 de **type** Point (pouvant éventuellement référencer un des types dérivés [4] de Point) :

```
Point p1;
```

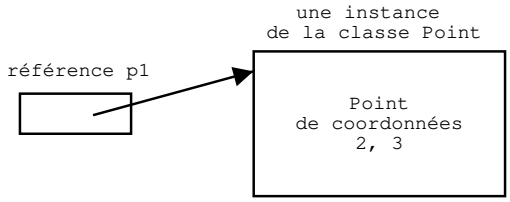
Rappel : lors de sa déclaration, une référence est initialisée par défaut à la valeur **null**. [5]

Instanciation d'un nouvel objet de la classe Point (de coordonnées 2, 3) référencé par p1 : emploi de new (allocation dynamique de mémoire) combiné à l'appel d'un constructeur (initialisation du nouvel objet).

```
p1 = new Point(2, 3);
```

Remarque : les opérations de déclaration, instanciation et affectation peuvent s'écrire sur une même ligne :

```
Point p1 = new Point(2, 3);
```



4. L'héritage et les classes dérivées sont traités au § 3.7 page 24.

5. Cette remarque n'est valable que pour les attributs, puisque les variables locales de méthode doivent obligatoirement être initialisées explicitement.

Exemple de la classe TestPoint (version 0)

```
/*
programme Java d'exemple de la classe TestPoint (version 0)
*/
public class TestPoint {
    public static void main(String[] args) {
        Point p1 = new Point(2, 3);
        p1.afficher();
    }
}
```

Dans cet exemple, on appelle la méthode afficher de la classe Point sur l'instance de cette classe référencée par la référence p1. C'est **l'opérateur "."** (point) qui permet d'appeler une méthode sur un objet. Les objets (instances) communiquent entre eux à l'aide de messages : un message est une demande d'exécution d'une méthode à un objet (par l'intermédiaire d'une référence sur cet objet, seul moyen d'y accéder).

Remarque : la comparaison de référence (ex. : p1 == p2) permet de savoir si p1 et p2 référencent le même objet mais ne permet pas de comparer le contenu des objets référencés.

3.6 Encapsulation et niveaux de visibilité

L'encapsulation des données permet de garantir que les données ne sont pas accessibles de l'extérieur (de la classe), et ne peuvent être manipulées que par l'interface proposée (les méthodes).

Tout attribut, méthode ou classe possède un niveau d'accessibilité.

Pour les classes (et les interfaces cf. § 5.3 page 37), il existe deux niveaux différents[6] (tableau 3.1) :

modificateur	signification pour une classe ou une interface
public	accès toujours possible
	accès possible depuis les classes du même paquetage

TABLE 3.1 – Les 2 niveaux de visibilité des classes et interfaces

Pour les attributs et les méthodes, il existe quatre niveaux distincts[7] (tableau 3.2) :

modificateur	signification pour un attribut ou une méthode
public	accès possible partout où sa classe est accessible
protected	accès possible depuis les classes dérivée (à l'intérieur ou à l'extérieur du paquetage) et depuis toutes les classes du même paquetage
	accès possible depuis toutes les classes du même paquetage
private	accès restreint à la classe où est faite la déclaration

TABLE 3.2 – les 4 niveaux de visibilité des attributs et méthodes

D'une manière générale :

- ▷ les classes sont déclarées "public" sauf si ce sont des "*inner-class*" ou classe internes (cas où une classe est imbriquée dans une autre classe). Il ne peut y avoir qu'une seule classe "public" par fichier .java, de ce fait on crée généralement un fichier par classe.

6. la notion de paquetage est détaillée au § 5.2 page 34
7. Remarque : La visibilité "private protected" n'existe plus.

- ▷ les attributs sont déclarés : private, “ ” (paquetage) ou protected selon ce que l'on souhaite en faire : empêcher leur accès depuis l'extérieur, réserver cet accès aux classes du même paquetage ou prévoir leur accessibilité par les classes dérivées.
- ▷ les méthodes sont habituellement de niveau “public”.

Comme les attributs ne sont pas “public”, on prévoit généralement des méthodes particulières appelées accesseurs permettant de les manipuler, ces accesseurs peuvent être en lecture ou en écriture et porte par convention des nom commençant par “get” pour les accesseurs en lecture [8] et par “set” pour les accesseurs en écriture [9], suivi du nom de l'attribut lui-même (cf. § 2.4.3 page 17 pour les conventions d'écriture). (Ex. : un accesseur “getAbscisse( )” de notre classe Point retournera la valeur de l'abscisse du point considéré).

Notre classe point devient “public” et s'enrichit des précisions sur les niveaux de visibilité de ses attributs et méthodes (avec comme convention d'écriture : “-” pour “private”, “#” pour “protected”, “”(rien) pour le niveau paquetage et “+” pour “public”). Nous y avons rajouté un deuxième constructeur (sans argument), appelé constructeur par défaut, une méthode “deplacer(int dx, int dy)” permettant de translater le point, et des accesseurs en lecture et en écriture pour les deux attributs.

Nous avons à présent deux méthodes constructeurs portant le même nom mais ayant des paramètres différents, on parle alors de **surdéfinition** ou de **surcharge** [10] de méthode.

Point
-abscisse: int -ordonnee: int
+Point() *+Point(abs:int,ord:int)* *+deplacer(dx:int,dy:int): void* *+afficher(): void* *+getAbscisse(): int* *+getOrdonnee(): int* *+setAbscisse(abs:int): void* *+setOrdonnee(ord:int): void*

Exemple de la classe Point (version 1)

```
/*
programme Java d'exemple de la classe Point version 1
*/
// une seule classe "public" par fichier
public class Point {
    // deux attributs privés
    // remarque : ils pourraient être protégés (protected)
    private int abscisse;
    private int ordonnee;
    // constructeur sans argument (par défaut)
    public Point() {
        abscisse = 0;
        ordonnee = 0;
    }
    // constructeur avec arguments
    public Point (int abs, int ord) {
        abscisse = abs;
        ordonnee = ord;
    }
    // méthode permettant de déplacer le point dans le plan (transla-
tion)
```

8. Il existe également des accesseurs en lecture commençant par “is” pour les attributs booléens (exemple : méthode “isEmpty( )” de la classe ArrayList) .

9. Ces méthodes accesseurs sont parfois désignées sous les appellations ***getter*** et ***setter***.

10. Cette forme est parfois appelée “polymorphisme paramétrique”, la méthode pouvant prendre plusieurs formes différentes en fonction des paramètres passés.

```
    public void deplacer(int dx, int dy) {
        abscisse += dx;
        ordonnee += dy;
    }
    // méthode d'affichage
    public void afficher() {
        System.out.println("je suis un point d'abscisse : " + abs-
cisse + " et d'ordonnée : " + ordonnee);
    }
    // deux accesseurs en lecture
    public int getAbscisse() {
        return abscisse;
    }
    public int getOrdonnee() {
        return ordonnee;
    }
    // deux accesseurs en écriture
    public void setAbscisse(int abs) {
        // on placerait ici tous les contrôles sur la validité du para-
mètre
        abscisse = abs;
    }
    public void setOrdonnee(int ord) {
        // idem
        ordonnee = ord;
    }
} // fin de la classe Point (version 1)
```

3.7 Héritage et polymorphisme

3.7.1 Héritage

L'héritage[11] est une relation entre classes (alors que l'instanciation est une relation entre une classe et un objet). Elle est définie par deux propriétés : on dit qu'une classe D (classe dérivée) hérite (ou dérive) d'une classe B (classe de base) si :

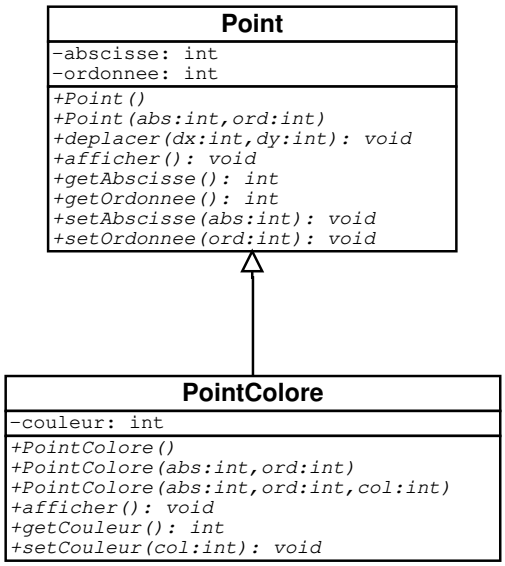
1. L'ensemble des attributs et / ou des méthodes de D est un sur-ensemble des attributs et méthodes de B
2. Le langage JAVA considère que D est *compatible avec* B, dans le sens où toute instance de D est admise comme étant un "cas particulier" de B dans un programme quelconque.

Une classe dérivée (ou classe fille[12]) n'a pas à mentionner les attributs et méthodes de sa classe de base (ou classe mère ou super-classe) : ceux-ci sont implicitement présents. On fait donc une définition incrémentale de la nouvelle classe, en ne mentionnant que ce qui la différencie de sa classe mère.

Remarque : lorsqu'un objet reçoit une demande de service, si la méthode correspondante est définie dans sa classe, elle est exécutée ; sinon la classe mère est consultée, et ainsi de suite jusqu'à trouver la méthode dans l'une des classes de la hiérarchie de classes de la classe initiale. La méthode est alors exécutée.

11. La relation d'héritage entre classes peut être comparée à la taxinomie en botanique ou en zoologie (classification d'éléments).
12. Les classes dérivée sont parfois appelées "sous-classe".

Dans notre exemple de classe Point nous définissons une classe dérivée “PointColore”, ayant les mêmes propriétés qu’un Point auquel on ajoute de la couleur.



 En JAVA, la relation d’héritage entre deux classes est indiquée par le mot-clef **extends**.

Exemple de la classe PointColore

```
/*
exemple d'héritage en Java
programme Java d'exemple de la classe PointColore (qui dérive de Point)
*/
public class PointColore extends Point {
    private int couleur;
    public PointColore() {
    // l'appel au constructeur de la classe de base doit être la pre-
mière instruction
        super();
        couleur = 0;
    }
    public PointColore(int abs, int ord) {
        super(abs, ord);
        couleur = 0;
    }
    public PointColore(int abs, int ord, int col) {
        super(abs, ord);
        setCouleur(col); // appel de l'accesseur en écriture
    }
    public void afficher() {
        super.afficher();
        System.out.println(" >>> et de couleur : " + couleur );
    }
    // accesseur en lecture sur l'attribut couleur :
    public int getCouleur() {
        return couleur;
    }

    // accesseur en écriture (contrôlée) sur l'attribut couleur :
    public void setCouleur(int col) {
```

```
            if (couleur < 0) {
                throw new IllegalArgumentException("la cou-
leur ne peut pas être négative");
            }
            else { // le else est optionnel puisqu'une exception est le-
vée cf.  5.6
                couleur = col;
            }
        }
} // fin de la classe PointColore
```

Remarques sur ce programme : le mot-clef **super** :

Lorsque l'on souhaite utiliser explicitement des constructeurs ou méthodes d'une classe de base depuis une classe dérivée, il faut utiliser le mot-clef **super**. celui-ci permet de faire référence à la classe directement supérieure (dans la hiérarchie de classes) dont dérive la classe considérée. Ainsi, on a dans la classe PointColore la possibilité d'appeler explicitement des constructeurs et méthodes de la classe Point.

- ▷ Pour les *constructeurs*, l'appel à un constructeur de la classe de base doit être la première ligne du constructeur de la classe dérivée. On peut exploiter les différentes formes de constructeurs : dans notre exemple, super( ) fait référence au constructeur sans paramètre de la classe Point et super(abs, ord) fait référence au constructeur avec deux paramètres entiers de la même classe Point. Si la première instruction d'un constructeur n'est pas super( ) ou this( ) [13], avec ou sans arguments, un appel à super( ) [14] est automatiquement ajouté avant la première instruction.
- ▷ Pour les *méthodes*, l'appel peut se faire depuis n'importe quel endroit du code, on appelle une méthode de la classe de base en faisant précéder son nom du mot-clef super : dans notre exemple super.afficher( ) fait référence à la méthode afficher( ) de la classe Point.
- ▷ Pour les *attributs*, le mécanisme est identique aux méthodes : on fait précéder le nom de l'attribut du mot-clef super. l'accès à un attribut d'une classe de base par une méthode d'une classe dérivée est conditionné par le niveau de visibilité de cet attribut cf. § 3.6

Remarque : la méthode afficher( ) est présente dans les deux classes (classe de base et classe dérivée) : on dit qu'il y a **redéfinition** de la méthode afficher( ).

3.7.2 Polymorphisme [15]

Remarques préalables :

Principe : tout objet d'une classe dérivée peut être utilisé partout où un objet de la classe de base est attendu (*"Qui peut le plus peut le moins"*) :

```
Point p = new PointColore();
```

Cette instruction est valide, p référence un nouvel objet de type PointColore (classe dérivée de la classe Point) car un PointColore est (aussi) un Point.

N.B. : par contre l'inverse n'est pas vrai :

```
PointColore pc = new Point(); // erreur
```

Cette instruction provoque une erreur lors de la compilation en effet, la méthode getCouleur( ) par exemple n'a aucun sens pour un Point.

13. cf. § 5.1.3 page 33
14. Appel du constructeur sans paramètre de la classe de base, aussi appelé constructeur par défaut.
15. Ce paragraphe présente la notion de polymorphisme **dynamique** en relation avec la notion d'héritage.

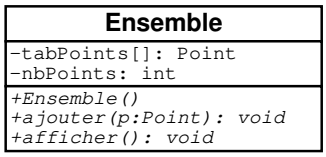
Polymorphisme

Littéralement, le polymorphisme désigne la capacité à prendre plusieurs formes. Ramené au contexte de la P.O.O., il s'agit d'une propriété qui va permettre à l'émetteur d'un message (appel de méthode), de l'envoyer à son destinataire sans connaître avec précision la nature de ce destinataire.

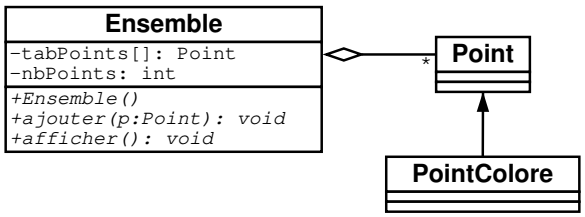
L'intérêt de cette notion vient de ce que l'héritage introduit dans un langage, même s'il est typé, des situations où l'on n'a pas à la compilation les informations nécessaires à la détermination de la classe d'appartenance d'un objet : les classes dérivées étant compatibles avec leur classe de base, une instance d'une classe dérivée peut être fournie lorsque le programme a besoin d'une instance de la classe de base. Dans ce cas, l'objet en question n'est plus traité que comme une instance de sa classe de base, on a perdu l'information de son type précis.

Pour parer à ce problème, on a recours à un mécanisme dit de ***liaison dynamique***. Ce mécanisme est ainsi nommé parce qu'il retarde au moment de l'exécution le choix de la méthode à déclencher lors de la réception d'un message (les méthodes redéfinies sont sélectionnées dynamiquement lors de l'exécution). Lorsqu'une méthode est redéfinie dans une classe dérivée, c'est celle-ci qui prévaut sur la méthode (de même signature) de la classe de base. La méthode correspondante est recherchée dynamiquement par la JVM au moment de l'exécution en fonction du type réel référencé.

Prenons l'exemple d'un ensemble de points (classe Point) du plan constitué de points (classe Point) et de points colorés (classe PointColore) :



Cela est possible car un PointColore **est aussi** un Point (N.B. : l'inverse n'est pas vrai). Si on souhaite afficher notre ensemble, on appellera la méthode "afficher( )" de chacun de nos objets. Si la liaison entre le message envoyé (méthode afficher) et la méthode déclenchée pour y répondre est faite à la compilation, on voit que c'est la version de la méthode afficher( ) de "Point" qui sera systématiquement utilisée, puisque l'ensemble considère ses éléments comme des points (classe de base). Si en revanche on applique la liaison dynamique, alors chaque point s'affichera en fonction de sa nature réelle (coloré ou non). C'est cela que recouvre en P.O.O., la notion de polymorphisme (les appels de méthode sont réalisés en fonction de la classe d'appartenance de l'objet référencé).



3.8 Classe et méthode abstraite (*abstract*)

Une **méthode** est dite abstraite (virtuelle pure du C++) si elle n'est que déclarée dans la classe, et non définie (i.e. seul son prototype est fourni, pas le corps de la méthode). En Java, une méthode abstraite est précédé du mot-clef **abstract.**

L'avantage des méthodes abstraites est qu'elle peuvent déclarer un comportement global commun à un ensemble de classes, et ceci très haut dans la hiérarchie de l'héritage.

Une **classe** abstraite est une classe non instanciable : on ne peut pas lui appliquer l'opérateur new. Une classe abstraite possède généralement au moins une méthode abstraite (mais pas nécessairement). Comme pour une méthode abstraite, le nom de la classe sera précédé du mot-clef **abstract**.

Exemple : si on souhaite modéliser tous les véhicules motorisés qui circulent sur une route (voiture, moto, scooter, camion, tracteur...), on peut regrouper dans une même classe de base tous les éléments communs à cette catégorie (attribut nbRoues, méthode accesseur getNbRoues( ) ...). Si on est sûr qu'il existe bien une méthode accélérer pour l'ensemble de cette catégorie, on est incapable à ce stade d'en préciser les modalités (poignée ou pédale d'accélération ...) : on indiquera donc l'existence de cette méthode sans pouvoir en préciser le code, que seules les classes dérivées seront en mesure d'implémenter :

```
public abstract class Vehicule {
    ...
    public abstract accelerer() ;
    ...
}
```

Remarque : même si une classe abstraite est non instanciable, cela n'empêche pas qu'elle ait des constructeurs qui seront appelés par les classes dérivées (dans notre exemple, on pourrait initialiser l'attribut nbRoues à une valeur par défaut).

Chapitre 4

UML et modélisation

4.1 Pourquoi utiliser une modélisation [1] ?

La modélisation permet de représenter une simplification de la réalité de façon à faciliter la compréhension du système à développer.

- ▷ Un modèle est une simplification de la réalité.
- ▷ Les modèles permettent de mieux comprendre le système que l'on développe.
- ▷ Nous construisons des modèles pour les systèmes complexes parce que nous ne sommes pas en mesure d'appréhender de tels systèmes dans leur intégralité.

Les quatre principes de la modélisation :

1. Le choix des modèles à créer a une très forte influence sur la manière d'aborder un problème et sur la nature de sa solution.
2. Tous les modèles peuvent avoir différents niveaux de précision.
3. Les meilleurs modèles ne perdent pas le sens de la réalité.
4. Parce qu'aucun modèle n'est suffisant à lui seul, il est préférable de décomposer un système important en un ensemble de petits modèles presque indépendants.

4.2 UML : *Unified Modeling Language*

Historiquement, UML est issue du regroupement de plusieurs méthode d'analyse dont **Booch** et **OMT** (*Object Modeling Technology*). Ses concepteurs sont Grady Booch, James Rumbaugh et Ivar Jacobson. La méthode **UML** (*Unified Modeling Language*) est devenue le standard industriel des méthodes d'analyse pour la modélisation objet. La notation UML est devenue le standard de modélisation visuelle adopté par le comité ANSI [2] ainsi que par l'OMG [3].

N.B. : Un mémento UML est disponible à la fin de ce document : cf. § 12 page 87.

Il existe différents diagrammes (différentes vues) possibles d'un même système : aspects statiques et dynamiques, niveaux de détails, etc... Les diagrammes sont organisés en paquetages.

UML propose quatre types de diagrammes statiques pour la modélisation structurelle (en fonction du niveau de détail désiré et de la complexité du système à modéliser).

1. *Remarque* : les éléments de ce paragraphe sont extraits de [Boo00].
2. ANSI : *American National Standard Institute*
3. OMG : *Object Management Group*

1. Les diagrammes de déploiement dans lesquels chaque noeud est constitué de un ou plusieurs composants.
2. Les diagrammes de composants dans lesquels chaque composant est constitué de une ou plusieurs classes.
3. Les diagrammes de classes qui sont les plus fréquent en P.O.O. dans lesquels on trouve les classes, les interfaces et leurs relations.
4. Les diagrammes d'objets qui permettent d'illustrer les structures de données et de représenter les instances dans un contexte précis (cas réel ou prototype).

Dans cette introduction, nous nous intéresserons uniquement aux diagrammes de classes.

4.2.1 Les diagrammes de Classes

Ce sont les diagrammes les plus fréquemment utilisés pour débuter en P.O.O. Il permettent de représenter les différentes classes du modèle, les attributs et les méthodes, les types, les niveaux de visibilité, les relations entres classes.

Dans la pratique, un diagramme de classes doit constituer un ensemble équilibré, ce qui signifie qu'une classe ne doit être ni trop grande (la diviser en plusieurs classes si besoin) ni trop petite (procéder à des regroupements si nécessaire). Chaque classe doit correspondre à une abstraction conceptuelle dans le contexte modélisé.

Chapitre 5

Java : compléments essentiels

5.1 static, final, this et Garbage Collector

5.1.1 attributs et méthodes de classes : le mot-clef static

attribut de classe

Le mot-clef **static** permet de définir des attributs de classe, c'est à dire des attributs qui ne seront pas liés à une instance de cette classe (à un objet) mais qui seront communs à tous les objets de cette classe. Il est possible d'accéder à ces attributs même si aucun objet de cette classe n'a été instancié.

Ces attributs particuliers sont précédés du mot-clef **static**. On peut y accéder indifféremment (sous réserve que leur niveau de visibilité le permette) par le nom de la classe ou par une référence sur une instance de cette classe.

Exemple : on pourrait imaginer un attribut de classe de notre classe Point représentant le nombre de point créés : nbPoints ; cet attribut serait incrémenté au niveau des constructeurs et décrémenté au niveau de la méthode finalize( ) [1]. Un autre exemple déjà utilisé est l'attribut out de la classe System (`System.out`).

Remarque : pour les attributs, **static** s'utilise généralement en combinaison avec **final** afin de réaliser des constantes de classe (cf. § 5.1.2).

méthode de classe

Le mot-clef **static** permet également de définir des méthodes de classe. Nous rappelons que même si les méthodes sont communes à tous les objets d'une classe (le code des méthodes n'est pas dupliqué), elles ne peuvent s'appliquer que sur une instance de cette classe (par le biais d'une référence). Dans le cas de méthodes statiques, celles-ci peuvent être appelées même si aucun objet de cette classe n'a été créé : on accède à ces méthodes en faisant précéder leur nom du nom de la classe.

Ce mécanisme de méthode de classe s'avère très utile.

Exemple 1 : double java.lang.Math.sqrt(double a) ; Comme toutes les méthodes de la classe java.lang.Math, sqrt( ) est une méthode static (de classe) de telle sorte qu'il est possible de l'utiliser sans avoir à créer un objet de la classe Math :

```
// affichage de la racine carrée de 2
System.out.println(java.lang.Math.sqrt(2.0));
```

1. En réalité, cette utilisation reste fictive car le "nettoyage" de la mémoire par le ramasse-miettes se fait de manière asynchrone et non prioritaire par rapport à l'exécution du programme, la valeur de l'attribut nbPoints ne pourrait donc être fiable...

Exemple 2 : String java.lang.String.valueOf(int i) ; Cette méthode de classe (static) permet de convertir un entier en une chaîne de caractères de type String.

Exemple 3 : méthode parseInt( ) de la classe java.lang.Integer permettant de convertir une chaîne de caractères en un entier :

```
int val = Integer.parseInt("12");
```

Exemple 4 : méthode permettant de calculer la factorielle d'un nombre :

```
package outils;

public class Math2 {
   public static long factorielle(int n) throws IllegalArgumentExcep-
tion {
      long fact = 1L;
      if (n<0) {
         throw new IllegalArgumentException();
      }
      else {
         for (int i=n ; i>1 ; i--) {
            fact *= i;
         }
      }
      return fact;
   }
}
```

utilisation de cette méthode dans un programme :

```
int a = 5;
long resultat = outils.Math2.factorielle(a);
```

Remarque : une méthode "static" (de classe) est implicitement "final" (cf. § 5.1.2) et ne peut donc être redéfinie.

5.1.2 Constantes : le mot-clef final

Constantes

Le mot-clef **final** permet de définir des constantes. celles-ci doivent être initialisées au moment de leur définition et ne peuvent être ultérieurement modifiées (elles ne peuvent être réaffectées à une nouvelle valeur) [2].

Constantes de classe

En combinant les propriétés **static** (cf. § 5.1.1) et **final**, on peut créer des constantes de classe, c'est-à-dire des constantes indépendantes des instances de cette classe.

Exemple de constante de classe : java.lang.Math.PI ; Cette constante de classe permet d'utiliser (une valeur approchée de) π (PI) sans instance de la classe Math :

2. Ce mécanisme est équivalent au mot-clef **const** du C++.

```
// affichage de la circonférence d'un cercle de rayon r
System.out.println(2 * java.lang.Math.PI * r);
```

Remarque : De même qu'en C, il est recommandé d'utiliser les majuscules pour les noms des constantes.

final

le mot-clef final peut être utilisé dans d'autres contexte que les constantes :

- ▷ appliqué à une classe, il indique que cette classe ne peut être dérivée (ex. : public final class String).
- ▷ appliqué à une méthode, il indique que cette méthode ne peut être redéfinie.

5.1.3 Auto-référence : le mot clef this

Le mot-clef **this** permet de faire référence à l'objet courant **à l'intérieur** du code le décrivant. Cette facilité se révèle très utile dans le cas de plusieurs constructeurs. En effet, les constructeurs d'une classe ont généralement des codes très similaires avec selon les cas une initialisation par défaut d'un attribut ou une initialisation explicite à partir des paramètres passés. Afin de ne pas dupliquer des lignes d'instructions identiques, il est possible d'appeler un autre constructeur de la même classe à partir du constructeur courant par l'utilisation du mot clef **this( )** auquel on passe éventuellement des paramètres.

Dans l'exemple de notre classe PointColore, nous pouvons remplacer le code des deux constructeurs avec arguments par :

```
public PointColore(int abs, int ord) {
    this(abs, ord, 0); // appel du constructeur avec 3 paramètres
}
public PointColore(int abs, int ord, int col) {
    super(abs, ord);
    couleur = col;
}
```

Une deuxième utilisation du mot-clef **this** est mise en oeuvre lors du passage de paramètres à une méthode, dans les cas ou le paramètre passé désigne un des attributs. JAVA offre la possibilité de nommer le paramètre passé exactement comme l'attribut (puisqu'ils désignent la même notion conceptuelle) : dans ce cas il faudra pouvoir différencier le paramètre passé de l'attribut (puisqu'ils portent le même nom) cette différenciation se fera par le mot-clef **this.** qui précédera l'attribut.

Dans l'exemple de notre classe Point, nous pouvons récrire le code de l'accesseur en écriture setAbscisse par :

```
public void setAbscisse(int abscisse) {
    // le paramètre passé porte le même nom que l'attribut
    this.abscisse = abscisse; // attribut = paramètre passé
}
```

5.1.4 Tableau récapitulatif

5.1.5 Le Garbage Collector (ou ramasse-miettes)

La destruction des objets en JAVA n'est pas comme en C++ explicite (il n'existe pas en JAVA d'instruction *delete* ni de méthode destructeur), mais est assurée par le ***Garbage Collector*** (ou

	classe	interface	attribut	méthode
`abstract`	la classe ne peut être instanciée, contient au moins une méthode abstraite.	par définition, une interface est abstraite		seule la signature est indiquée : la méthode doit être redéfinie dans une classe dérivée.
`final`	la classe ne peut être dérivée.	l'interface ne peut être dérivée.	l'attribut est constant.	la méthode ne peut être redéfinie dans une classe dérivée.
`static`			attribut de classe : commun à tous les objets de la classe.	méthode de classe : peut être invoquée sans nécessiter d'instance de la classe.

TABLE 5.1 – tableau récapitulatif "abstract, final, static"

ramasse-miettes) qui détruit en tâche de fond tous les objets qui ne sont plus référencés [3]. C'est la machine virtuelle Java qui gère le *Garbage Collector*.

Toutefois, il est possible de réaliser des actions spécifiques au moment de la destruction d'un objet par la méthode **finalize**( ). Cette méthode particulière est appelée (invoquée) directement par le *Garbage Collector* avant la libération de la mémoire associée à cet objet (la destruction de l'objet). Elle est définie dans la classe Object (cf. § 5.5 page 41) et peut être redéfinie au sein d'une classe. On peut donc placer dans cette méthode le code nécessaire à la restitution des ressources externes occupées par l'objet (fermeture d'un canal de communication (*socket*) ou d'un fichier par exemple). Sa redéfinition est optionnelle et n'est nécessaire que pour la libération de ressources autre que la mémoire.

N.B. : Lors de cette redéfinition de la méthode finalize( ), ne pas oublier d'appeler la méthode finalize( ) de la classe Object (ou de la classe de base) par super.finalize( ).

5.2 package (paquetage)

Préambule : il existe une classe Point de l'AWT (cf. § 11 page 84) : java.awt.Point. De ce fait, il est nécessaire qu'il n'y ait pas de confusion entre cette classe existante et notre classe d'exemple. Ce problème est résolu en JAVA par le mécanisme des **paquetages** (***packages***) qui permettent de regrouper les classes, ayant des corrélations fonctionnelles, par famille.

Il existe plusieurs dizaines de paquetages fournis avec le langage [4] :

Tout paquetage doit être importé (sauf pour le paquetage *java.lang* [5]) : mot-clef **import**. Les principaux paquetages sont présentés dans le tableau 5.3.

Toute classe JAVA, appartient nécessairement à un paquetage (sans indication particulière, une classe appartient au paquetage par défaut [6]). L'appartenance à un paquetage doit être la première ligne d'un

3. Parfois appelé ébouleur, le *Garbage Collector* n'est pas synchrone avec l'exécution du programme mais est exécuté en parallèle et de façon cyclique : lorsqu'un objet n'est plus référencé, il sera à terme détruit (lors de la scrutation suivante de la mémoire par le *Garbage Collector*), et la mémoire qu'il occupait sera libérée.

4. L'API de la version 1.2 du JDK comportait plus de 7000 méthodes.

5. Le paquetage *java.lang* est "indispensable" et est **implicitement** importé dans tout programme JAVA.

6. Si le nom du paquetage n'est pas spécifié, la classe appartient alors au paquetage par défaut ou paquetage anonyme (*unnamed*). Cette utilisation n'est pas recommandée et il est toujours préférable de spécifier un paquetage d'appartenance.

version du JDK	paquetages	classes	améliorations majeures
1.0	8	212	version initiale (core)
1.1	23	504	classes internes, performances de la JVM
1.2	59	1520	API Swing, API des collections, appellation plate-forme Java 2
1.3	76	1842	amélioration de la stabilité et des performances
1.4	135	2991	API d'E/S java.nio, expressions régulières, XML, SSL
1.5 / 5.0	166	3278	collections paramétrées, entrées-sorties améliorées
6	202	3780	amélioration du support XML, de JDBC, des Services Web
7	209	4024	amélioration du support unicode et de la sécurité, Java FX
8	217	4240	amélioration du support des annotations, de la sécurité, TLS
9			
10			
11			
12			

TABLE 5.2 – versions du JDK et paquetages

fichier source est se déclare par le mot-clef **package** suivi du nom du paquetage. L'identification complet d'une classe (ou d'une interface) est défini par le nom (éventuellement composé) de son paquetage d'appartenance **et** le nom de la classe (de l'interface) elle-même. ex. : java.lang.Math ("java.lang" est le nom du paquetage d'appartenance et "Math" est le nom de la classe elle-même).

Afin de ne pas rencontrer de conflits dans les noms des classes utilisées, les noms des paquetages suivent une règle s'appuyant sur les noms DNS (*Domain Name Services*) inverses.

Exemple : package fr.u-cergy.monpaquetage ;

 N.B. : Par convention, et afin de le différencier d'une classe, un nom de paquetage est toujours en minuscules.

Exemple de la classe Point (version 2) - fichier Point.java

```
/*
programme Java d'exemple de la classe Point avec paquetage (version 2)
*/
// l'appartenance a un paquetage doit être la première ligne du fi-
chier source
package geometrie;
// le nom complet pourrait être : package fr.u-cergy.ml.geometrie;
// une seule classe publique par fichier
public class Point {
    // les attributs
    protected int abscisse;
    protected int ordonnee;
    // les constructeurs
    public Point() {
        this(0, 0);
    }
    public Point (int abscisse, int ordonnee) {
        this.abscisse = abscisse;
        this.ordonnee = ordonnee;
    }
    // les méthodes
    public void deplacer(int dx, int dy) {
```

nom du paquetage	regroupement de classes
java.applet	Contient essentiellement la classe Applet (Applet, AudioClip...).
java.awt	(*abstract windowing toolkit*) Regroupe l'ensemble des classes de composants graphiques AWT (cf. § 11) (Graphics, Component, Color...).
java.awt.event	Regroupe l'ensemble des interfaces écouteurs et l'ensemble des classes d'événement et d'Adapter (cf. § .8.4)
java.io	(*input-output*) Regroupe l'ensemble des éléments nécessaires aux entrées-sorties sur fichiers (cf. § 9) (File...).
java.lang	(*language*) Regroupe toutes les classes, interfaces et exceptions "fondamentales" à tout programme JAVA (System, Math ...), est importé implicitement et automatiquement pour tout programme JAVA (String, Thread, System...).
java.net	(*network*) Regroupe tous les éléments nécessaires à la programmation réseau (URL, Socket...).
java.rmi	(*remote method invocation*) Regroupe les principaux éléments nécessaires à la programmation d'applications distribuées.
java.sql	(*structured query language*) Regroupe les classes permettant l'interfaçage avec des bases de données dont JDBC (*Java DataBase Connectivity*).
java.util	(*utilities*) Regroupe toutes les classes "utilitaires", en particulier les collections (cf. § 6.2) (ArrayList, HashMap ...).
javax.swing	Regroupe l'ensemble des composants graphiques "légers" (écrit en JAVA et complètement indépendants de la plate-forme) de type swing (cf. § 8) (JFrame, JButton...).

TABLE 5.3 – Les principaux paquetages JAVA.

```
        abscisse += dx;
        ordonnee += dy;
    }
    public void afficher() {
        System.out.println("je suis un point d'abscisse : " + abs-
cisse + " et d'ordonnée : " + ordonnee);
    }

    // accesseurs en lecture
    public int getAbscisse() {
        return abscisse;
    }
    public int getOrdonnee() {
        return ordonnee;
    }

    // accesseurs en écriture
    public void setAbscisse(int abscisse) {
        this.abscisse = abscisse;
    }
    public void setOrdonnee(int ordonnee) {
        this.ordonnee = ordonnee;
    }
} // fin de la classe Point (version 2)
```

Exemple de la classe TestPoint (version 2)

```
/*
programme Java d'exemple de la classe TestPoint (version 2)
*/
package test; // la classe de test appartient habituelle-
ment a un autre paquetage, ou éventuellement au paquetage par dé-
faut (si on ne précise rien).
import geometrie.Point; // on peut aussi sélectionner collective-
ment les classes du paquetage, ici par import geometrie.*;
public class TestPoint {
    public static void main(String[] args) {
        Point p1 = new Point(2, 3);
        p1.afficher();
        p1.deplacer(4, 1);
        p1.afficher();
    }
}
```

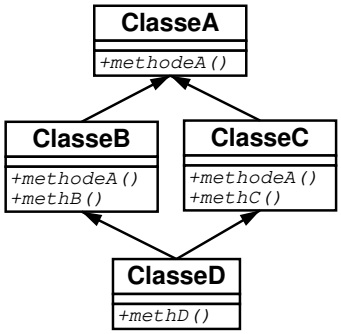
Rappel (cf. § 1.4 page 7) : la compilation se fera par : javac Point.java et javac TestPoint.java (ou javac *.java) et l'exécution par : java test.TestPoint (lancement de la classe publique disposant d'une méthode main( ) en précisant son paquetage d'appartenance).

5.3 Interface

5.3.1 Présentation

Les problème liés à l'héritage multiple (cas du losange) :

Si on suppose l'existence d'une classe ClasseA définissant une méthode methodeA( ) et de deux classes dérivée classeB et classeC redéfinissant chacune la méthode methodeA( ) pour leurs propres besoins. Si l'héritage multiple est possible, cela signifie que l'on peut créer une classe ClasseD dérivée à la fois de ClasseB et de ClasseC (on souhaite que ClasseD puisse bénéficier des services methB( ) et methC( ) respectivement définis dans ces deux classes). Dans ce cas, quelle est la méthode methodeA( ) disponible dans ClasseD ? (celle héritée de ClasseB ? celle héritée de ClasseC ? ou bien celle d'origine héritée de ClasseA ?) :



C'est pour cette raison qu'en JAVA, seul l'**héritage simple** est possible.

5.3.2 Définition

Une **interface** est une extension de la notion de classe abstraite (cf. § 3.8 page 27), puisque dans le cas d'une interface, toutes les méthodes sont abstraites et seuls des attributs constants de classe (*static* **et** *final*) peuvent être définis. Une interface peut donc être constituée d'attributs constants,

de méthodes abstraites (dans ce cas, les qualificateurs *public* et *abstract* peuvent être omis car ils sont implicites), de méthodes statiques ou encore (depuis la version 7 de Java) de méthode par défaut (mot-clef *default*).

L'héritage multiple (contrairement au C++), n'existant pas en Java, les interfaces sont le mécanisme qui permet de réaliser un pseudo-héritage multiple.

Une interface se définie par le mot-clef **interface** (en lieu et place du mot-clef **class**), une interface est implicitement abstraite (on ne précise pas le mot-clef *abstract*).

Une classe ne peut hériter que d'une seule classe mère mais peut implémenter (mot-clef **implements**) une ou plusieurs interfaces. Pour être instanciable, une classe doit définir l'ensemble des méthodes abstraites de l'interface qu'elle implémente.

Une interface définit un type : on peut déclarer des variables de ce type.

5.3.3 Syntaxe

En reprenant l'exemple du § 5.3.1, si on souhaite créer une classe ClasseD bénéficiant à la fois des services de ClasseB et de ClasseC, il faut en JAVA créer une **interface** pour l'une de ces deux classes (on choisira pour l'interface, la classe demandant le moins de travail de réécriture).

L'interface créée contiendra les signatures (les prototypes) des méthodes de la classe (pour une interface, il n'est pas nécessaire d'indiquer que les méthodes sont abstraites) :

```
public interface InterfaceC {
    public void methC();
}
```

On pourra alors réaliser une structure équivalente à l'héritage multiple :

```
public class ClasseD extends ClasseB implements InterfaceC {
    ...
}
```

Remarque : ClasseD doit définir methC( ) sinon elle est abstraite.

Voir aussi § 6.3.2 page 51 pour d'autres exemples d'interfaces.

5.4 La manipulation des chaînes de caractères : les classes String et StringBuffer

Il existe en JAVA, deux classes permettant de manipuler les chaînes de caractères : la classe **String** pour les chaînes constantes (immuables) et la classe **StringBuffer** pour les chaînes de contenu et de longueur variables (modifiables).

5.4.1 La classe String

La classe String constitue un cas particulier unique : c'est la seule classe qui autorise une initialisation de référence sans instanciation d'un objet de cette classe (opérateur new) :

```
String s1 = "POO : UML et Java";
```

Cette instruction est valide : la chaîne de caractères “POO : UML et Java” est stockée par le compilateur dans la zone de code et non dans le tas (*heap*), de façon différente à toutes les autres classes.

N.B. : on peut néanmoins procéder comme pour toute autre classe (dans ce cas l'objet correspondant sera créé dans le tas) :

```
String s1 = new String("POO : UML et Java");
```

Concaténation de chaînes de caractères

La surcharge de l'opérateur “+” :

En JAVA, (contrairement au C++), il n'est pas possible d'effectuer une surcharge des opérateurs arithmétiques. La seule exception à cette règle est le cas de la concaténation de deux chaînes de caractères :

```
String s1 = "POO : ";
String s2 = s1 + "UML et Java";
```

JAVA construit un nouvel objet String à partir de la concaténation des chaînes de caractères et le fournit comme résultat de l'expression (ce nouvel objet String sera dans l'exemple référencé par s2).

Les objets de la classe **String** sont **immuables** : ils ne peuvent être modifiés. Lorsqu'on effectue une concaténation par l'opérateur “+”, le compilateur crée de nouveaux objets intermédiaires (dont certains sont éventuellement inutiles au programmeur) et crée en particulier un objet de type StringBuffer.

Exemple :

```
String s = "UML" + " et " + "Java";
```

`java.lang.String`

```
String()
String(Byte[ ] bytes)
char charAt(int index)
int compareTo(String anotherString) // teste l'ordre lexicogra-
phique d'après l'encodage unicode[7].
boolean endsWith(String suffix)
boolean equals(Object obj) // généralement obj est de type String !
boolean equalsIgnoreCase(String anotherString)
int indexOf(int ch)
int indexOf(int ch, int fromIndex)
int indexOf(String str)
int indexOf(String str, int fromIndex)
int lastIndexOf(String str)
int length()
boolean matches(String pattern) // pattern est le motif d'une expres-
sion régulière
String replaceAll(String regex, String replacement)
String replaceFirst(String regex, String replacement)
String[] split(String regex)
```

7. Dans le cas d'une internationalisation (caractères diacritiques, diphtongues, ...), on peut utiliser java.text.Collator.compare( ).

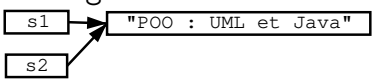
```
boolean startsWith(String prefix)
String subString(int beginIndex)
String subString(int beginIndex, int endIndex)
char[ ] toCharArray()
String toLowerCase()
String toUpperCase()
String trim()
static String valueOf(type t) // cf. 5.4.1
```

Comparaison de chaînes de caractères

 N.B. : La méthode "**equals( )**", héritée de la classe Object et redéfinie dans la classe String, permet de comparer deux chaînes de caractères. Il est important de noter que l'égalité ne peut être testée par le signe "==" car les chaînes de caractères sont des objets (et non des types de base), on vérifierait alors l'égalité de référence et non l'égalité de contenu ! !

Important : Cette affirmation peut s'avérer fausse dans le cas particulier de déclarations sans allocation dynamique de mémoire :

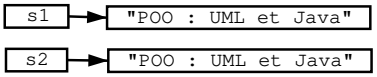
```
String s1 = "POO : UML et Java";
String s2 = "POO : UML et Java";
```



Dans ce cas, s1 == s2 est vrai [8] car le compilateur cherche à optimiser la mémoire nécessaire au stockage des chaînes de caractères [9].

Remarque :

```
String s1 = new String("POO : UML et Java");
String s2 = new String("POO : UML et Java");
```



Dans ce cas on peut être sûr que s1 == s2 est faux car s1 et s2 référencent deux objets distincts, par contre s1.equals(s2) est vrai.

Conversion d'un type de base en une chaîne de caractères

La classe String possède un ensemble de méthodes *static* permettant la conversion depuis un type de base (type primitif) vers une chaîne de caractères [10] : les méthode valueOf( ) :

```
static String valueOf(char c)
static String valueOf(boolean b)
static String valueOf(byte b)
static String valueOf(short s)
static String valueOf(int i)
```

8. Ce comportement particulier peut constituer une source d'erreur difficilement décelable.
9. En réalité, tout dépendra des règles d'optimisation du compilateur, il est donc indispensable de toujours comparer des chaînes de caractères par la méthode **equals( )** : s1.equals(s2). (sauf bien sûr si l'on souhaite faire une comparaison de référence et non de contenu).
10. Cet ensemble de méthodes de même nom mais de signatures différentes est un bon exemple du mécanisme de surdéfinition (surcharge) de méthode (différenciation sur le type du paramètre passé).

```
static String valueOf(long l)
static String valueOf(float f)
static String valueOf(double d)
```

Conversion d'un objet quelconque en une chaîne de caractères : la méthode toString( ) :

De très nombreuses classes JAVA redéfinissent la méthode toString( ) [11] héritée de la classe Object (qui par défaut affiche la valeur de la référence) afin de transformer un objet quelconque en une chaîne de caractères "équivalente" le représentant.

5.4.2 La classe StringBuffer

La classe StringBuffer permet de manipuler des chaînes de caractères variables. Les objets de la classe **StringBuffer** sont **modifiables** : le contenu et la longueur d'un objet de type StringBuffer peuvent être changés.

`java.lang.StringBuffer`

```
StringBuffer(String str)
StringBuffer append(String str) // méthode surchargée (10 x)
StringBuffer insert(int offset, String str) // méthode surchargée (10 x)
StringBuffer replace(int start, int end, String str)
String substring(int start, int end)
String toString() // redéfinition de la méthode héritée de Object
```

5.4.3 La classe StringTokenizer

La classe StringTokenizer permet de découper une chaîne de caractères en plusieurs morceaux en fonction d'un ou plusieurs séparateurs (*tokens*). On peut réaliser une analyse syntaxique d'une chaîne en "unités lexicales" séparées par des délimiteurs.

`java.lang.StringTokenizer`

```
StringTokenizer(String text, String tokens)
int countTokens()
boolean hasMoreTokens()
String nextToken()
```

Remarque : l'API [12] java.util.regex apparue avec le JDK 1.4 offre des solutions puissantes de manipulations de chaînes de caractères par les expressions régulières (*regular expression*), voir également String.split( ).

5.5 La classe Object

En JAVA, toutes les classes dérivent implicitement de la classe Object, une hiérarchie de classes quelconques peut donc toujours être représentée par un arbre d'héritage dont la racine est la classe Object.

11. La méthode toString( ) est un bon exemple d'un cas de redéfinition de méthode dans des classes dérivées.
12. API : *Application Programming Interface* : couche logicielle de programmation offrant un ensemble services sous la forme d'une interface. En JAVA, ensemble de classes fournissant ce service.

`java.lang.Object`

```
Object()
protected Object clone()
boolean equals(Object obj)
protected void finalize()
int hashcode() // utilisée pour les classes de dictionnaire
Class getClass() [13]
Object newInstance() // instanciation de la classe chargée par forName()
String toString()
```

La redéfinition de la méthode `toString()`

La plupart des classes fournies avec le JDK redéfinissent la méthode toString( ) de la classe Object afin de standardiser les mécanismes d'affichage en mode console et de permettre une conversion de tout objet en une chaîne de caractères.

La redéfinition de la méthode `equals(Object obj)`

Attention : lors de la redéfinition, si l'argument n'est pas de type Object dans la signature de la méthode equals( ) de la classe dérivée alors il s'agit d'une surcharge et non d'une redéfinition de la méthode equals( ).

La redéfinition de la méthode `clone()`

Pour pouvoir utiliser cette méthode, la classe considérée doit implémenter l'interface de marquage java.lang.Cloneable et rendre public la méthode clone() lors de la redéfinition [14].

Exemple : `Point p2 = (Point)(p1.clone());`

Remarque : la méthode clone( ) peut être appliquée à un tableau.

5.5.1 Comparaison d'objets

 N.B. : Il est important de ne pas confondre la comparaison d'objets et la comparaison de référence.

Exemple : soit deux points de même coordonnées :

```
Point p1 = new Point(1, 2);
Point p2 = new Point(1, 2);
```

La comparaison (p1 == p2) donne comme résultat **false** puisque p1 et p2 référencent des objets distincts.

Pour pouvoir comparer ces deux points, il faut **redéfinir** la méthode **equals( )** héritée de la classe Object (cf. § 5.5) dans la classe Point [15] :

13. Il est également possible d'obtenir la classe d'appartenance d'une classe C quelconque par C.class; (qui renvoie une référence sur un objet de type "class").
14. N.B. : pour les attributs de type référence, il n'y a par défaut qu'une copie superficielle. si une copie en profondeur est nécessaire, il faut écrire explicitement le code correspondant.
15. Par défaut la méthode equals( ) de la classe Object compare les références.

```
public boolean equals(Object obj) {
    boolean egalite = false;
    if (obj instanceof Point) {
        egalite = ((this.abscisse == ((Point)obj).abscisse) &&
(this.ordonnee == ((Point)obj).ordonnee));
    }
    return egalite;
}
```

on peut alors tester indifféremment p1.equals(p2) ou p2.equals(p1).

5.5.2 La classe Class et la méthode getClass( )

`java.lang.Class`

classe particulière permettant l'introspection c'est à dire la capacité auto-descriptive de certains objets[16].

```
String getName()
Object newInstance() // ne peut appeler que le constructeur par défaut
Class Class.forName(String className) // charge dynamique-
ment dans la JVM une nouvelle classe dont le nom est passé en para-
mètre[17].
Constructor[] getConstructors() // constructeurs public de la classe
Constructor[] getDeclaredConstructors() // tous les construc-
teurs de la classe.
Field[] getDeclaredFields() // tous les attributs décla-
rés dans la classe
Method[] getDeclaredMethods() // toutes les méthodes décla-
rées dans la classe
Field[] getFields() // tous les attributs public y compris les attri-
buts hérités.
Method[] getMethods() // toutes les méthodes public, y com-
pris celles héritées.
URL getRessource(String filename)
```

Remarque : l'utilisation des méthodes de réflexion de la classe Class peut être limitée par le gestionnaire de sécurité.

5.6 Erreurs et exceptions

Définition : Une exception est un objet particulier de la classe **Exception** ou d'une de ses sous-classes, qui est créé au moment où une situation anormale est détectée. Une fois créée, l'exception

16. La réflexion est notamment utilisée par les javabeans et la sérialisation : c'est la possibilité pour une classe ou un objet de s'examiner. L'API de réflexion contient les classes :

```
java.lang.reflect.Field
java.lang.reflect.Method
java.lang.reflect.Constructor
```

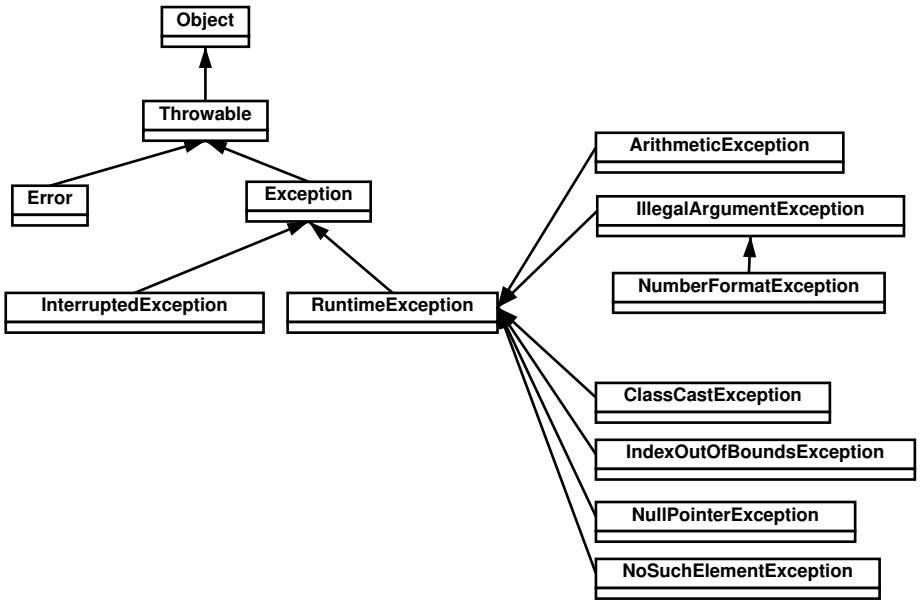
17. cf. RMI pour un exemple d'utilisation.

est lancée (ou levée[18]) à l'aide de l'instruction **throw**, ce qui a pour effet d'interrompre l'exécution normale du programme et de propager l'exception jusqu'au premier bloc d'instructions capable de la capter et de la traiter (**catch**). Si aucun *catch* ne convient, l'exception est propagée jusqu'à la JVM qui arrête alors le programme (ex. : NullPointerException).

Il existe deux catégories d'exception : les exceptions **contrôlées** (*checked*) et les exceptions **non contrôlées** (*unchecked*). Les exceptions qui sont des classes filles de la classe java.lang.RuntimeException ou de la classe java.lang.Error ne sont pas contrôlées : contrairement aux autres classes (et sous-classes) d'exception qui doivent être prises en charge par le programme (classe Exception ou classes dérivée de java.lang.Exception), le compilateur ne requiert pas que les exceptions de type Error ou RuntimeException soient gérées. Cela signifie qu'il n'est pas obligatoire de les intercepter (à l'aide d'un bloc try / catch). Dans ce cas (exception non crontôlée), la méthode peut lever une exception sans que sa signature ne le mentionne (omission de la clause *throws*), par exemple pour les exceptions de type NoSuchElementException, IllegalArgumentException, ...

Remarque : le bloc d'instruction situé dans la clause **finally** sera exécuté systématiquement (qu'une exception ait été levée ou non), la clause *finally* est optionnelle.

La hiérarchie des classes d'exceptions :



Les exception de type IndexOutOfBoundsException ou NullPointerException par exemple n'ont donc pas à être traitées explicitement dans le programme.

Le traitement d'une exception est réalisé en quatre étapes :

Étape 1 : Indiquer qu'une méthode est susceptible de lever une exception : **throws** (attention au "s" final) :

```
public void methodeTestException(int arg) throws NoSuchElementExcep-
tion { ... }
```

Étape 2 : créer et lancer l'exception dans la méthode concernée : **throw**

```
if(...) {
    throw new NoSuchElementException();
}
```

18. Les expressions "levée", "lancée" ou "déclenchée" sont équivalente.

Étape 3 : tester un bloc de code utilisant cette méthode : **try**

```
try {
    methodeTestException(12);
}
```

Étape 4 : attraper cette exception et la traiter : **catch**

```
catch (NoSuchElementException nsee) {
    nsee.printStackTrace();
}
```

Remarque : les instructions catch peuvent être multiples, dans ce cas elles sont écrites dans un ordre précis de priorité (la plus spécifique d'abord, la plus générale à la fin).

Quelques exemples d'exception :

`java.lang.NullPointerException` *: est levée lorsqu'on tente d'accéder à un attribut ou une méthode via une référence null*
`java.util.NoSuchElementException` *: est levée lorsqu'un élément n'appartient pas à une collection*
`java.lang.NumberFormatException` *: peut être lancée par la méthode Integer.parseInt()*
`java.lang.IOException, java.lang.FileNotFoundException` *: peuvent être lancées lors des accès fichiers*
`java.lang.ArrayIndexOutOfBoundsException` *: est levée lorsqu'on tente d'accéder à un élément en dehors des limites d'un tableau*

Remarque : Comme toute autre classe, il est possible de créer un type d'exception personnalisé et étendant la classe Exception ou une de ses sous classes.

`java.lang.Exception extends Throwable`

C'est la classe de base de l'ensemble de la hiérarchie des exceptions [19] (autres que les erreurs).

```
Exception()
Exception(String msg)
String getMessage()
printStackTrace()
String toString()
```

19. Remarque : depuis la version 1.4 du JDK, plusieurs méthodes supplémentaires sont apparues afin de permettre le chaînage des exceptions :

```
Exception(String msg, Exception e)
Exception getCause()
Stack getStackTrace()
initCause(Exception e)
```

Chapitre 6

Tableaux, collections et dictionnaires

6.1 Les Tableaux

En JAVA, les tableaux sont traités comme les classes, on crée des pseudo-objets, on les manipule donc avec des références (particulières) et **ils doivent être instanciés** (opérateur ***new***). Il est possible de construire des tableaux de l'un quelconque des types de base ainsi que des tableaux de références. Toutes les valeurs d'un tableau doivent être de même type[1]. Aucune méthode *public* n'est associé aux tableaux, seul le pseudo-attribut **length** est consultable (accessible en lecture seule), il contient le nombre d'éléments du tableau.

déclaration :

```
type[ ] nom; // ou type nom[ ];
```

Les deux formes précédentes sont équivalentes, néanmoins l'écriture de type `int[ ] tab` est recommandée et l'écriture `int tab[ ]` est prise en charge pour des raisons de compatibilité avec C et C++.

exemple

```
int[ ] tabEntiers;
int tabEntiers[ ];
```

Exemple d'un tableau de points nommé tabPoints (ces deux instructions sont rigoureusement équivalentes) :

```
Point[] tabPoints;
Point tabPoints[];
```


N.B : Lorsqu'on manipule des objets, les tableaux sont en réalité des tableaux de références.

Remarque : les variables tabEntiers et tabPoints sont elles-même des références.

définition

En JAVA, déclaration et définition sont deux étapes distinctes : la déclaration ne réserve pas de mémoire et ne crée qu'une référence (de type tableau). Lors de sa création (définition), chaque valeur du tableau est initialisée selon sa valeur par défaut cf. § 2.1 (sauf initialisation explicite cf. ci-après). Dans les exemples qui suivent, on crée un tableau de 5 entiers et un autre de 3 points.

1. S'il n'y a aucune ambiguïté pour les type primitif, il est possible dans le cas de références (d'une classe de base) d'y placer des références de d'un type dérivé : cf. § 3.7.

```
tabEntiers = new int[5];
tabPoints = new Point[3];
```

On pourra donc écrire la définition au niveau des constructeurs.

Remarque : on peut réaliser la déclaration et la définition simultanément :

```
int[] tabEntiers = new int[5];
Point[] tabPoints = new Point[3];
```

initialisation

Il est possible de décomposer chaque étapes de la construction et de l'initialisation d'un tableau :

```
Point p1 = new Point(1, 2);
Point p2 = new Point(3, 4);
Point p3 = new Point(5, 6);
Point[] tabPoints;
tabPoints = new Point[3];
tabPoints[0] = p1;
tabPoints[1] = p2;
tabPoints[2] = p3;
```

ou d'utiliser une écriture condensée (sans préciser la taille du tableau que le compilateur calculera automatiquement à partir des valeurs initiales) :

```
int[] tabEntiers = {1, 2, 3, 4, 5};
Point[] tabPoints = {p1, p2, p3};
String tabStrings[] = {"POO", "UML", "Java"};
```

l'attribut length

l'attribut (d'instance) **length**, de type int contient la taille du tableau, il n'est pas modifiable.

exemple :

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

Tableaux multidimensionnels

Les tableaux multidimensionnels sont vus comme des tableaux de tableaux et peuvent de ce fait être créés en une seule instruction ou en plusieurs :

```
int[][] tab1;
tab1 = new int[3][4]; // tableau à 2 dimensions de 3 lignes et 4 co-
lonnes
int[][] tab2 = new int[3][]; // tableau à 2 dimen-
sions dont seule la première est initialement précisée.
tab2[0] = new int[4];
tab2[1] = new int[4];
tab2[2] = new int[4];
```

Ces deux exemples sont équivalents.

Remarque : par ce mécanisme, il est possible de créer des tableaux à deux dimensions non rectangulaires (tableau triangulaire par exemple).

Remarque : S'il n'est pas possible (comme en C) de modifier la taille d'un tableau après sa création, il existe en revanche une méthode *static* de la classe *System* permettant de copier tout ou partie d'un tableau dans un autre tableau de taille différente :

```
java.lang.System.arraycopy(Object src, int src_position, Ob-
ject dst, int dst_position, int length)
```

Conversion

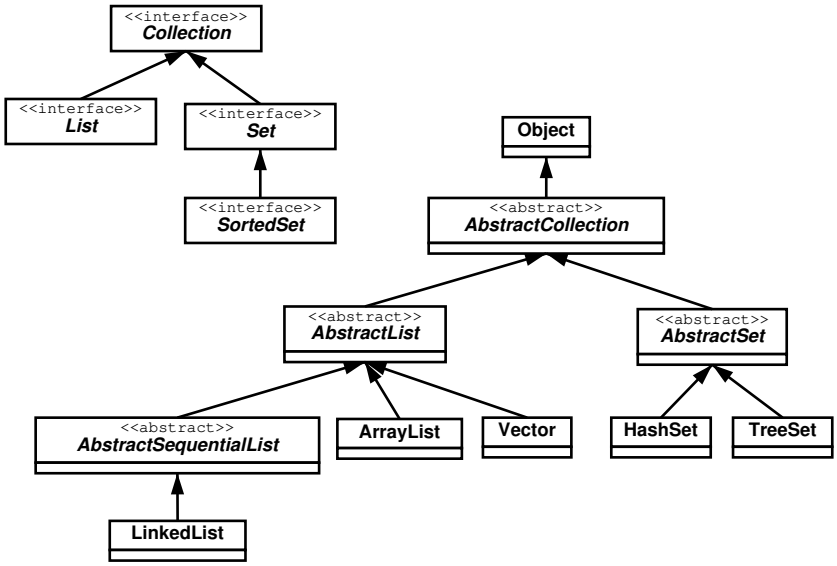
Remarque : il est possible de convertir un tableau en une collection et réciproquement :

java.util.Arrays

```
static List asList(Object[] tab)
```

6.2 Les collections

Une collection est un ensemble de taille dynamique, les éléments de l'ensemble sont nécessairement des références sur des objets. Depuis la version 1.5 du langage, les collections sont paramétriques.



- ▷ La classe TreeSet implémente l'interface SortedSet.
- ▷ La classe HashSet implémente l'interface Set.
- ▷ Les classes ArrayList, LinkedList et Vector implémentent l'interface List.
- ▷ Les classes HashMap, HashTable, LinkedHashMap, WeakHashMap et IdentityHashMap implémentent l'interface Map.
- ▷ La classe TreeMap implémente l'interface SortedMap.

6.2.1 L'interface java.util.Collection<E> [2]

De nombreuses classes de l'API des collections implémentent l'**interface Collection<E>** (ou une des ses sous-interfaces).

```
java.util.Collection<E>

    boolean add(E element);
    boolean remove(Object element);
    boolean contains(Object element);
    int size();
    boolean isEmpty();
    Iterator iterator();
    Object[] toArray();
```

Remarque : l'interface Set<E> (qui hérite de l'interface Collection<E>) assure que la duplication d'éléments n'est pas autorisée. La sous-interface SortedSet<E> ajoute les fonctionnalités d'ensemble trié : le critère de tri s'appuie sur un objet de type Comparator.

L'interface List<E> (qui hérite de l'interface Collection<E>) permet de manipuler les éléments comme dans un tableau :

```
java.util.List<E>

    void add(int index, E element);
    void remove(int index);
    Object get(int index);
    Object set(int index, E element);
```

L'**interface Map<K, V>** permet de manipuler des dictionnaires ou tableaux associatifs. Chaque élément de l'ensemble est constitué d'une paire clef / valeur (*key / value*) :

```
java.util.Map<K, V>

    V put(K key, V value);
    V get(Object key);
    V remove(Object key);
    int size();
    Set<K> keySet();
    Collection<V> values(); // la collection renvoyée peut contenir des élé-
    ments dupliqués.
```

Remarque : l'interface SortedMap<K, V> (héritée de Map<K, V>) gère les paires clef / valeur en ordre trié sur les clefs.

6.2.2 La classe ArrayList<E>

Depuis la version 1.2 du JDK, la classe ArrayList est à utiliser de préférence à la classes Vector.

2. A ne pas confondre avec la **classe** Collections (avec "s") qui propose un ensemble de méthodes *static* permettant d'implémenter une version synchronisée de n'importe quelle collection (dans le cas d'accès concurrents par plusieurs Threads). Elle permet également de créer des versions en lecture seule et un ensemble de manipulations (`sort( )`, `binarySearch( )`, `fill( )`, `max( )`, `min( )`, `reverse( )`, ...).

La classe **ArrayList<E>** permet d'implémenter des tableaux dynamiques d'objets (du type paramétrique E). Comme un tableau, un "ArrayList" contient des éléments auxquels on peut accéder en utilisant un index (de type int). La particularité d'un "ArrayList" est que sa taille peut augmenter ou diminuer en fonction des besoins en s'adaptant suivant les opérations d'ajout ou de suppression d'élément. Cette adaptation se fait de manière dynamique, après que le ArrayList ait été créé : par ré-allocation dynamique de mémoire.

Afin d'optimiser la gestion de la mémoire, tout "ArrayList" s'appuie sur deux paramètres : *capacity* et *capacityIncrement*. La capacité (*capacity*) est au moins aussi grande que la taille du ArrayLsit ; elle est généralement supérieure car lorsque des éléments sont ajoutés au ArrayList, la possibilité de stockage du vecteur augmente par blocs de la taille de la capacité d'incrément (*capacityIncrement*).

`java.util.ArrayList<E>`

```
// constructeurs
ArrayList<E>() // crée un tableau dynamique ArrayList d'une capacité ini-
tiale de 10 éléments
ArrayList<E>(Collection<E> c)
ArrayList<E>(int initialCapacity)
ArrayList<E>(int initialCapacity, int capacityIncrement)
// principales méthodes
boolean add(E element) // ajoute l'élément passé en para-
mètre à la fin de la liste
void add(int index, E element) // insère element à la position index
boolean addAll(Collection<E> c) // ajoute un ensemble d'éléments conte-
nus dans la collection c
boolean addAll(int index, Collection<E> c)
void clear() // supprime tous les éléments
Object clone() // héritée de la classe Object (cf. 5.5)
boolean contains(E element)
void ensureCapacity(int minCapacity)
E get(int index)
int indexOf(Object element)
boolean isEmpty()
int lastIndexOf(Object element)
E remove(int index) // supprime l'élément de la position index
boolean remove(Object element) // supprime la première occur-
rence de l'élément spécifié (méthode spécifiée dans l'interface List)
void removeRange(int fromIndex, int toIndex)
E set(int index, E element) // (re)définit l'élément de la position in-
dex
int size() // retourne le nombre d'éléments de la liste
List subList(int start, int end)
Object[] toArray()
void trimToSize()
```

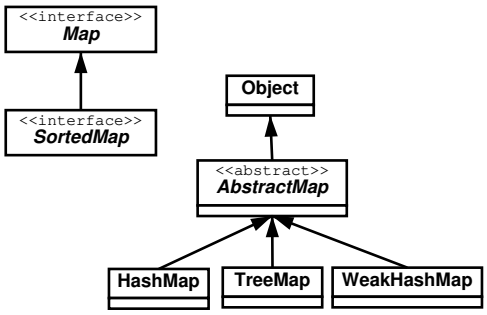
De même que pour un tableau, il est possible de parcourir un *ArrayList*. Il existe plusieurs alternatives pour parcourir un *ArrayLsit* : la technique la plus simple (spécifique à la collection *ArrayLsit*) est présentée ci-dessous et est très proche d'une technique de parcours d'un tableau. Deux autres techniques plus élaborées et communes à toutes les collections sont détaillées au § 6.3.2.

Exemple : Pour lister le contenu d'un ArrayLsit al de points (classe Point), il est possible d'écrire un code très proche de la syntaxe des tableaux.

```
for(int i = 0 ; i < al.size() ; i++) {
    Point p = al.get(i);
    p.deplacer(1, 1);
    System.out.println(p);
}
```

6.3 Les dictionnaires

Les classes HashMap et HashTable sont des dictionnaires (*Dictionary*). Les dictionnaires correspondent à des tableaux associatifs, ce sont des structures de données où l'on peut stocker des paires clef / valeur (*key / value*).



Remarque : Les algorithmes de *hashcode* (code de hachage) des classes HashTable et HashMap s'appuient sur la méthode equals( ) de l'objet constituant la clef. L'algorithme gère les cas (rares) de collisions entre deux objets différents ayant le même *hashcode*.

6.3.1 la classe HashMap<K,V>

La classe HashMap est une classe de haut niveau d'abstraction permettant des manipulations aisées sur un dictionnaire, elle utilise une table de hachage et des algorithmes permettant d'assurer un temps d'accès constant à tout élément du dictionnaire. On accède a ces éléments par deux méthodes get( ) et put( ). Depuis la version 1.2 du JDK, il est préconisé d'employer la classe HashMap (qui remplace l'ancienne classe HashTable des versions 1.0 et 1.1).

`java.util.HashMap`

```
HashMap<K,V>()
V get(Object key)
V put(K key, V value)
Collection<V> values()
Set<K> keySet()
```

6.3.2 Parcourir une collection : l'interfaces Iterator<E>

Pour parcourir une collection "al" de type ArrayList, il est possible de travailler comme si cette collection était un tableau :

```
for ( int i = 0 ; i < al.size() ; i++ ) {
    System.out.println(al.get(i));
}
```

Les classes de l'API des collections implémentent les interfaces (Iterator, ListIterator et / ou Enumeration) permettant de les parcourir. L'interface Enumeration est la plus ancienne et a été remplacée depuis le JDK 1.2 par l'interface **Iterator** dans un soucis d'harmonisation des noms de méthodes. Elle offre de plus une méthode supplémentaire (de suppression).

L'interface `Iterator<E>` :

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove(); // nécessairement associé (consécutif) à next()
}
```

Exemple de parcours d'une collection :

```
for (Iterator<Point> it = al.iterator() ; it.hasNext() ; ) {
    System.out.println(it.next());
}
```

Remarque : depuis la version 1.5 du langage, il existe une structure supplémentaire plus compacte (de type for each) permettant de parcourir une collection (ou un tableau) :

```
for (Object o : collection) {
    System.out.println(o);
}
```

Chapitre 7

L'architecture MVC *(Model Vue Controller)*

L'architecture MVC (*Model Vue Controller*) consiste à séparer les responsabilités : le modèle contient les données et fournit les méthodes pour y accéder en consultation ou en modification (i.e. les accesseurs). La vue est responsable de la représentation visuelle d'une partie ou de la totalité des données. Le contrôleur se charge de gérer les événements en provenance de la vue : il est responsable de la cohérence entre la vue et le modèle.

Cette architecture permet une organisation en "couches" superposées et distinctes. Par ailleurs, il est possible d'avoir plusieurs vues sur le même modèle de données (on peut imaginer des vues partielles ou exhaustives selon le statut de l'utilisateur : consultation ou administration par exemple).

L'implémentation de MVC dans Swing diffère un peu de l'architecture décrite ci-dessus mais les principes restent identiques. Avec Swing (ou AWT), la vue et le contrôleur sont étroitement liés et ne constituent généralement qu'un seul composant logiciel. On obtient alors une architecture à deux couches : le modèle d'une part et l'ensemble vue / contrôleur d'autre part.

Une modification par un utilisateur d'une vue doit provoquer la mise à jour au sein du modèle et la répercussion sur l'ensemble des vues ; de même une modification interne du modèle doit permettre d'avertir la ou les vues (par une notification) que les données ont changé.

- La "couche M" : gère l'état du modèle.
- La "couche V" :
  - représente le modèle à l'écran
  - gère le déplacement et le redimensionnement de la vue
  - intercepte les événements utilisateur
- La "couche C" : synchronise les changements entre le modèle et ses vues.

Chapitre 8

Interface graphique Swing

8.1 Principes généraux

Les composants graphiques Swing [1] (appelés composants légers) ont pour objectif d'assurer un rendu fidèle quel que soit le système d'exploitation c'est-à-dire de permettre la réalisation d'interfaces graphiques utilisateur indépendantes de la plate-forme. Les composants Swing sont réunis dans le paquetage javax.swing.

Conteneurs et composants : Tous les composants graphiques visibles sur une interface utilisateur sont des composants, qui dérivent de la classe abstraite JComponent placés dans des conteneurs.

Les éléments graphiques développés à l'aide de la bibliothèque de classes **Swing** sont dessinés par Swing et la JVM : l'apparence visuelle des éléments graphiques n'est pas déléguée au S.E. [2]

Swing permet de réaliser des interfaces graphiques presque exclusivement constituées de composants légers (sans pair (*peer*)) indépendants de la plate-forme d'exécution. Seuls les conteneurs de très haut niveau (JWindow et JFrame) interagissent avec le système de fenêtrage sous-jacent.

Grâce au *"**Pluggable Look and Feel**"* (*plaf*) , le rendu graphique peut être paramétré puisqu'il est effectué par Swing [3]. De même, il est possible de concevoir des éléments graphiques qui n'aient pas d'équivalent au niveau du S.E. Il existe 3 *look and feel* de base livrés avec Swing :

- ▷ **metal**, c'est le *look and feel* standard
- ▷ **motif**
- ▷ **windows**

Les classes de la bibliothèque graphique Swing appartiennent au paquetage **javax.swing** [4]

8.2 La gestion du "*look and feel*"

`javax.swing.UIManager`

```
static void setLookAndFeel(String className) // exception à gérer
```

1. en remplacement des composants AWT depuis le JDK 1.2.
2. Certaines différences peuvent néanmoins persister, en particulier sur l'aspect visuel des fenêtres principales (JWindow et JFrame) d'une application car en dernier recours c'est quand même le gestionnaire de fenêtre du S.E. qui affiche l'application à l'écran.
3. On obtient donc une diversification de l'affichage en fonction du type de mise ne page (*look and feel*) choisi, indépendamment du système graphique.
4. Les composants Swing n'appartenaient pas aux premières versions du langage (ils sont apparus avec le JDK 1.2), historiquement seuls les composants AWT existaient. Ils sont donc associés à un nouveau paquetage javax.swing (x pour *extended*).

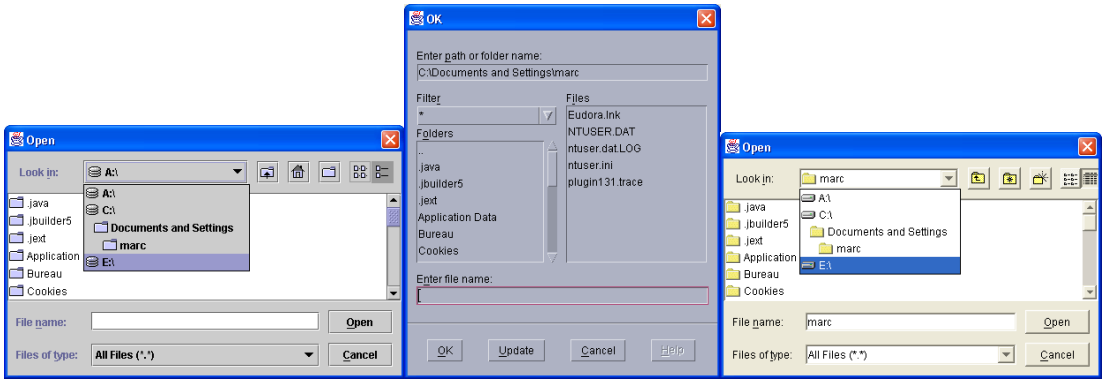


FIGURE 8.1 – les trois *look and feel* standards de Swing

```
static void setLookAndFeel(LookAndFeel looknfeel)
static String getSystemLookAndFeelClassName()
static String getCrossPlatformLookAndFeelClassName()
```

Remarque : le *look and feel* peut être défini dans le fichier swing.properties situé dans le répertoire JAVA_HOME/lib [5]

par la clef : swing.defaultlaf=... Si rien n'est spécifié, c'est le *look and feel* par défaut (metal) qui est chargé. Les trois classes de *look and feel* sont (cf. fig. 8.1) :

- ▷ String metal = "javax.swing.plaf.metal.MetalLookAndFeel" ;
- ▷ String motif = "com.sun.java.swing.plaf.motif.MotifLookAndFeel" ;
- ▷ String windows = "com.sun.java.swing.plaf.windows.WindowsLookAndFeel" ;

Remarque : pour modifier les éléments par défaut en anglais, la classe UIManager dispose d'une méthode static put( ). On peut par exemple changer le texte d'un bouton :

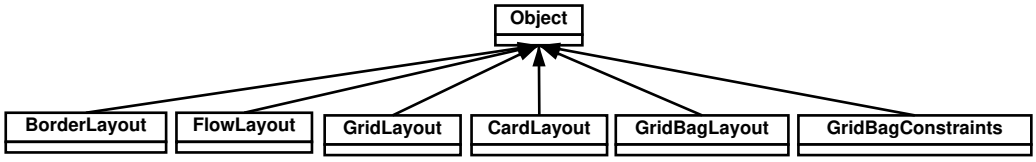
```
UIManager.put("JFileChooser.cancelButtonText", "Annuler");
```

Pour que les modifications de *Look and Feel* soient prises en compte, il faut demander la mise à jour des composants en appelant la méthode suivante sur le conteneur principal :

```
SwingUtilities.updateComponentTreeUI(Component comp)
```

8.3 Les gestionnaires de présentation : LayoutManager

AWT et Swing utilisent des gestionnaires de placement (*layout manager*) pour disposer des composants graphiques à l'intérieur de conteneurs. Les gestionnaires de placement définissent une stratégie de disposition des composants au lieu de spécifier des positions absolues.



L'association d'un gestionnaire de placement à un container (JFrame, JDialog, JPanel, JApplet ...) se fait par la méthode setLayout( ) de ce container :

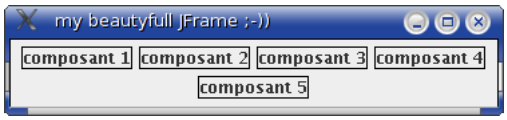
```
void setLayout(LayoutManager mgr)
```

Pour mélanger des gestionnaires de mise en page dans une même fenêtre graphique, il faut utiliser un sous conteneur de type JPanel (en positionnant un nouveau gestionnaire sur le JPanel par setLayout( )).

5. JAVA_HOME désignant le répertoire d'installation du JDK.

8.3.1 FlowLayout

Les composants sont placés “au mieux” dans la fenêtre :



constructeur(s) :

```
FlowLayout() // CENTER par défaut
FlowLayout(int alignement) // FlowLayout.LEFT, FlowLayout.CENTER, Flow-
Layout.RIGHT
FlowLayout(int alignement, int espacement_horizontal, int espace-
ment_vertical)
```

l’ajout d’un composant au conteneur associé se fait par :

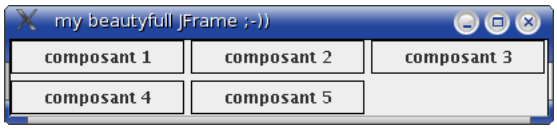
```
add(Component comp)
```

Remarque : l’ordre d’ajout des composants dans le conteneur est important.

Remarque : c’est le LayoutManager par défaut pour les JPanel et les JApplet

8.3.2 GridLayout

Les composants sont placés dans une grille invisible (ici de 2 lignes et 3 colonnes) :



constructeur(s) :

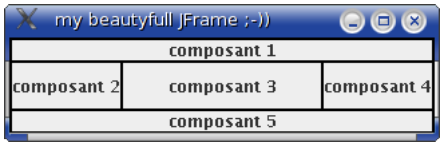
```
GridLayout(int lignes, int colonnes)
GridLayout(int rows, int cols, int hgap, int vgap) // nb lignes, nb co-
lonnes, espacement horizontal, espacement vertical
```

l’ajout d’un composant au conteneur associé se fait par :

```
add(Component comp) // l'ordre d'insertion des compo-
sants dans la grille se fait de gauche à droite et de haut en bas
add(Component comp, int index) // index = -1 pour un ajout à la fin
```

8.3.3 BorderLayout

Les composants sont placés dans 5 zones : nord, sud, est, ouest et centre :



constructeur(s) :

```
BorderLayout()
```

l'ajout d'un composant au conteneur associé se fait par :

```
void add(Component comp, Ob-
ject constraints) // constraints est un des attri-
buts de classe (de type String) : BorderLayout.NORTH, Border-
Layout.SOUTH, BorderLayout.EAST, BorderLayout.WEST, BorderLayout.CENTER
```

Remarque : c'est le LayoutManager par défaut pour les JFrame et les JWindow

8.3.4 CardLayout

Ce gestionnaire permet de réaliser des interfaces a onglets, il y a une superposition fonctionnelle des différents composants. En général, ce sont des regroupements de composants (cf. JPanel p.62) qui sont superposés. Les méthodes next( ), previous( ), first( ) et last( ) permettent de placer en avant-plan le composant souhaité de la pile de composants.

constructeur(s) :

```
CardLayout()
CardLayout(int hgap, int vgap)
```

8.3.5 GridBagLayout

Plus complet que GridLayout, ce gestionnaire permet en particulier de fusionner des cellules contigües de la grille. Il s'utilise conjointement à GridBagConstraints. Le placement des composants est plus précis et plus souple que le GridLayout mais la programmation est plus lourde.

constructeur(s) :

```
GridBagLayout()
```

8.3.6 BoxLayout (javax.swing.BoxLayout)

Ce gestionnaire de placement est apparu avec swing et permet d'aligner les composants horizontalement ou verticalement.

```
BoxLayout(Container contain, int align) // contain est une réfé-
rence sur le container, align peut prendre l'une des deux valeurs Box-
Layout.X_AXIS ou BoxLayout.Y_AXIS
```

Comme pour les autres gestionnaires de placement, on utilise la méthode setLayout( ).

Remarque : La méthode statique createHorizontalStrut(int i) de la classe Box peut être mise en oeuvre avec ce nouveau gestionnaire pour une mise en page plus sophistiquée.

8.3.7 Aucun gestionnaire

Il est possible de réaliser une gestion statique et absolue des composants dans un conteneur par :

```
setLayout(null)
```

Cette possibilité va à l'encontre de la philosophie de Java et ne doit être utilisée qu'en dernier recours.

N.B. : La valeur "null" n'est pas équivalente au gestionnaire de placement par défaut (`FlowLayout` pour un `JPanel` ...).

8.4 Java et la programmation événementielle

Événements et écouteurs :

Java et la programmation événementielle : Les événements sont envoyés par un objet source à un ou plusieurs écouteurs ou *Listener*. Un *listener* implémente des méthodes de gestion d'événements prescrites. Il s'enregistre auprès d'une source de ce type d'événements[6].

événement	Déclenché par	Interface *Listener*	méthodes gestionnaire

`java.awt.event.`
`ComponentEvent`
	Tous les composants	`ComponentListener`	`componentResized()`
			`componentMoved()`
			`componentShown()`
			`componentHidden()`

`java.awt.event.`
 	
`FocusEvent`
	Tous les composants	`FocusListener`	`focusGained()`
			`focusLost()`

`java.awt.event.`
`KeyEvent`
	Tous les composants	`KeyListener`	`keyTipped()`
			`keyPressed()`
			`keyReleased()`

`java.awt.event.`
`MouseEvent`
	Tous les composants	`MouseListener`	`mouseClicked()`
			`mousePressed()`
			`mouseReleased()`
			`mouseEntered()`
			`mouseExited()`

		`MouseMotionListener`	`mouseDragged()` `mouseMoved()`

`java.awt.event.`
 	
`ActionEvent`
	`JButton`
	`JCheckBoxMenuItem`
	`JComboBox`
	`JFileChooser`
	`JList`
	`JRadioButtonMenuItem`
	`JTextField`
	`JToggleButton`
		`ActionListener`	`actionPerformed()`
			`getActionCommand()`
			`getSource()`

`java.awt.event.`
`WindowEvent`
	`JFrame`
	`JDialog`
	`JWindow`
		`WindowListener`	`windowOpened()`
			`windowClosing()`
			`windowClosed()`
			`windowIconified()`
			`windowDeiconified()`
			`windowActivated()`
			`windowDeactivated()`

TABLE 8.1 – Quelques exemples d'événements de composants et conteneurs (AWT et Swing)

Les événements traités lors de la programmation des interfaces graphiques sont principalement issus du clavier et de la souris. La gestion des événements en JAVA est basé sur le principe de la délégation. Chaque événement est représenté par un objet qui constitue la "mémoire" de l'événement. Cet objet contient des informations générales telles que la source de l'événement ou le moment auquel il a eu lieu et des informations spécifiques dépendant du type d'événement.

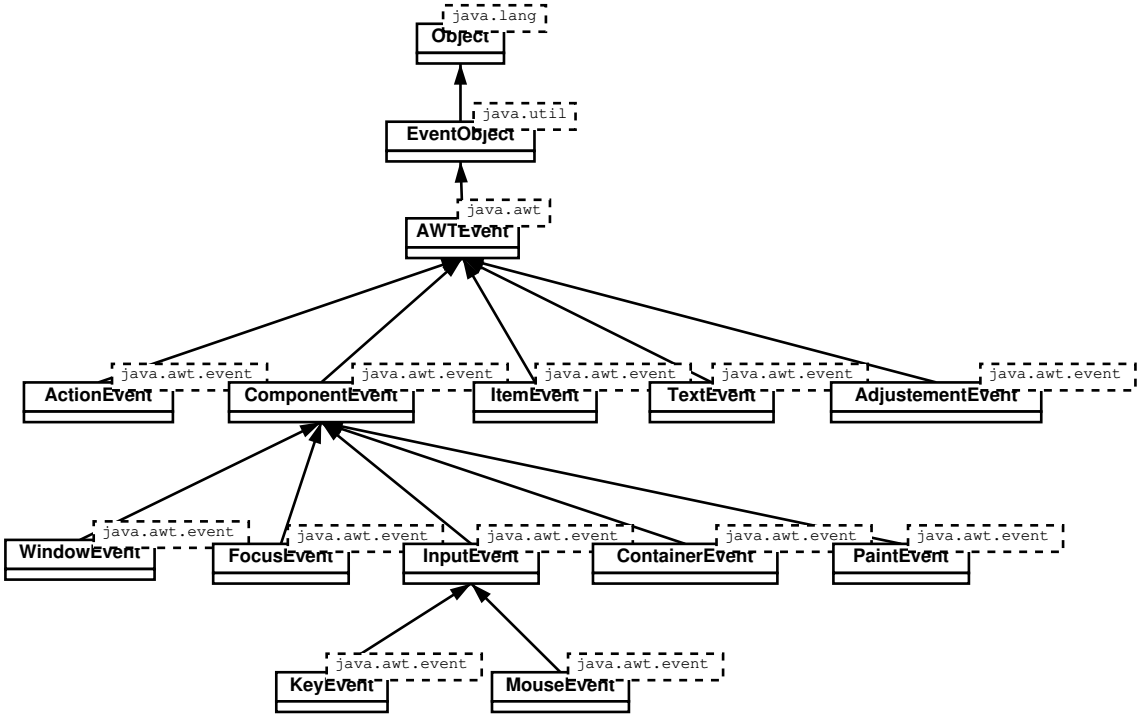
Tout objet peut être source d'événement, il doit alors :

6. La file d'attente des événements est gérée par un unique *thread* système.

▷ posséder une méthode d'inscription (d'abonnement) et de désinscription d'écouteurs.
▷ transmettre les événements en appelant les méthodes requises des écouteurs.

La gestion des événements en Java s'effectue en deux temps :

1. Étape préalable : abonnement du listener (écouteur) auprès du composant graphique (addXxx-Listener(XxxListener listener)).
2. Lors d'une action (un clic par exemple) : notification du listener par le composant graphique par appel de la méthode associée (ex. : actionPerformed(ActionEvent e)).

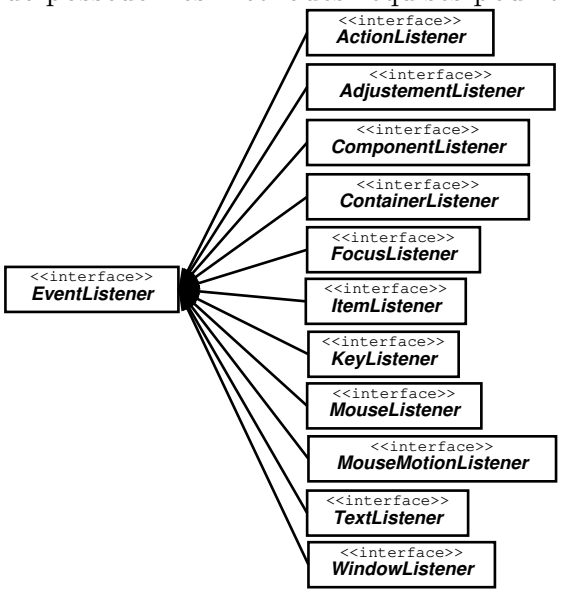


Remarque : une source peut avoir plusieurs écouteurs et un écouteur peut être inscrit auprès de plusieurs sources différentes.

8.4.1 Les interfaces d'écouteurs (package java.awt.event)

N'importe quel objet peut recevoir des événements de son choix, il lui suffit pour cela :

▷ de s'inscrire comme écouteur auprès d'un objet source d'événements.
▷ de posséder les méthodes requises pour traiter ce type d'événements.



```
ActionListener[7] // validation d'un bouton ou d'un menu via la souris ou le clavier
AdjustmentListener // ex. : barre de défilement
FocusListener // basculement du focus par tabulation
ItemListener // ex. : case à cocher
KeyListener // clavier
MouseListener // souris : ex. : clic : méthodes mousePressed(MouseEvent me) et mouseReleased(MouseEvent me)
MouseMotionListener // déplacement de la souris
WindowListener // pour tous les événement de type : WINDOW_DESTROY, WINDOW_EXPOSE (basculement au premier plan), WINDOW_ICONIFY, WINDOW_DEICONIFY, WINDOW_MOVED
```

8.4.2 Associer un écouteur à un composant

```
addActionListener() // clic sur un bouton, sélection dans un menu
addAdjustmentListener()
addFocusListener()
addItemListener()
addListSelectionListener() // pour les JList
addKeyListener()
addMouseListener() // pour les boutons de la souris
addMouseMotionListener() // pour les déplacements de la souris
addPropertyChangeListener()
addWindowListener()
```

8.4.3 Les événements

Les classes d'événement appartiennent principalement au paquetage `java.awt.event` ; elles définissent au minimum les méthodes `getSource()`, `getX()` et `getY()`.

```
ActionEvent
ChangeEvent // paquetage javax.swing.event
DocumentEvent
FocusEvent
KeyEvent
ListSelectionEvent // cf. JList
MouseEvent // méthode boolean isPopupTrigger()
PropertyChangeEvent // paquetage java.beans
WindowEvent
```

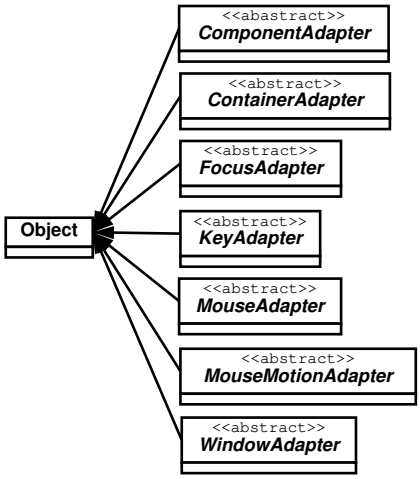
exemple avec `java.awt.event.ActionEvent` **:**

```
Object getSource() // méthode héritée de EventObject
String getActionCommand()
getModifiers()
paramString()
```

7. cette interface ne possède qu'une seule méthode : public void actionPerformed(ActionEvent event);

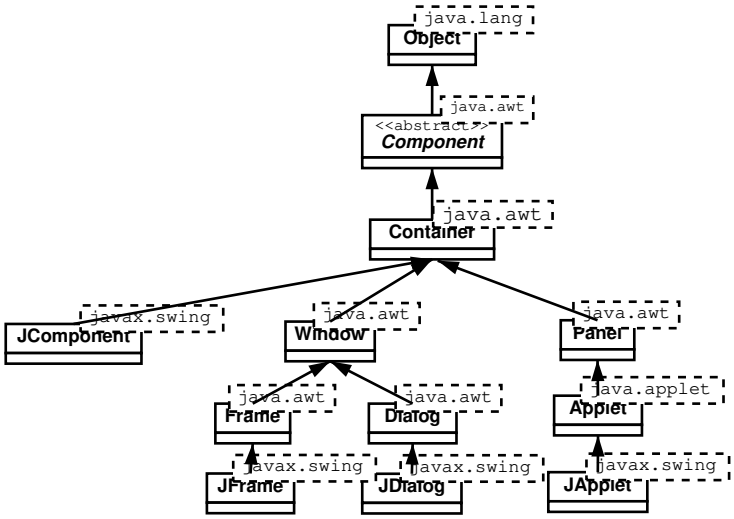
8.4.4 Les Adapter

Les "Adapter" sont des classes particulières qui implémentent les interfaces écouteurs avec un comportement par défaut (i.e. les méthodes ne font rien), ces classes permettent dans certains cas de remplacer une implémentation d'interface (avec définitions obligatoire de toutes les méthodes prévues) par un héritage en ne redéfinissant que la ou les méthodes souhaitées.



Exemple : la classe WindowAdapter implémente les 7 méthodes de l'interface WindowListener avec des corps de méthodes vides.

8.5 Les principales classes de conteneurs Swing

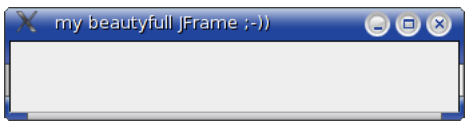


`javax.swing.JWindow`

Fenêtre sans bouton système ni titre, non déplaçable et non redimensionnable par l'utilisateur. Doit donc être réservé au "*splash screen*" de lancement d'une application.

`javax.swing.JFrame`

Fenêtre principale d'une application



```
JFrame()
JFrame(String title)
void addWindowListener(WindowListener wl)
JMenuBar getJMenuBar()
void dispose() // destruction de la fenêtre (héritée de Window)
Container getContentPane()
void hide() // héritée de Window, deprecated
void pack() // dimensionne la fenêtre de manière optimale en fonc-
tion des composants qu'elle contient (héritée de Window)
void setBounds(int x, int y, int width, int height)
void setContentPane(Container ctr)
void setDefaultCloseOperation(int opera-
tion) // JFrame.EXIT_ON_CLOSE, WindowCons-
tants.DO_NOTHING_ON_CLOSE, JFrame.DISPOSE_ON_CLOSE
void setJMenuBar(JMenuBar menuBar)
void setResizable(boolean resize)
void setSize(int width, int height)
void setTitle(String title)
void setVisible(boolean visible) // héritée de Component
void show() // hérité de Window, deprecated
void toBack() // arrière plan
void toFront() // avant plan (premier plan)
```

 N.B. : la méthode `add()` permettant d'ajouter les composants dans la fenêtre s'applique au **Container** associé à la fenêtre (récupérable par la méthode `getContentPane()`) et non directement à la `JFrame` (`jframe.getContentPane().add(...)`).

Les différents conteneurs d'une `JFrame` sont organisés selon un modèle en couches :

`RootPane` : conteneur racine qui contient tous les autres (en particulier JMenuBar) sur le principe du "*Design Patern Composite*" (modèle de conception basé sur une arborescence de composants imbriqués).

`ContentPane` : conteneur des composants graphiques autres que le menu (c'est le JPanel de haut niveau d'une JFrame).

`GlassPane` : conteneur "transparent" situé devant le ContentPane.

`javax.swing.JPanel`

Panneau

```
JPanel()
add(Component comp) // héritée de java.awt.Container
setLayout(LayoutManager mgr)
```

`javax.swing.JTabbedPane`

panneau à onglets

```
JTabbedPane()
addTab(String title, JPanel pane)
setToolTipTextAt(int index, String text)
```

`javax.swing.JScrollPane`

conteneur avec barres de défilement (ascenseurs) n'acceptant qu'un seul composant (pouvant être lui-même un sous-conteneur). Le composant contenu dans le JScrollPane doit implémenter l'interface Scrollable.

```
JScrollPane()
JScrollPane(JComponent comp) // ex. JScrollPane(JList list) ou JScroll-
Pane(JTree tree)
void setViewPort(JComponent comp)
ViewPort getViewPort() // cf. classe ViewPort
```

`javax.swing.JSplitPane`

Conteneur permettant de séparer son contenu en deux zones distinctes (verticalement ou horizontalement) et redimensionnables dynamiquement. En général, les composants sont eux-même des conteneurs de type JPanel.

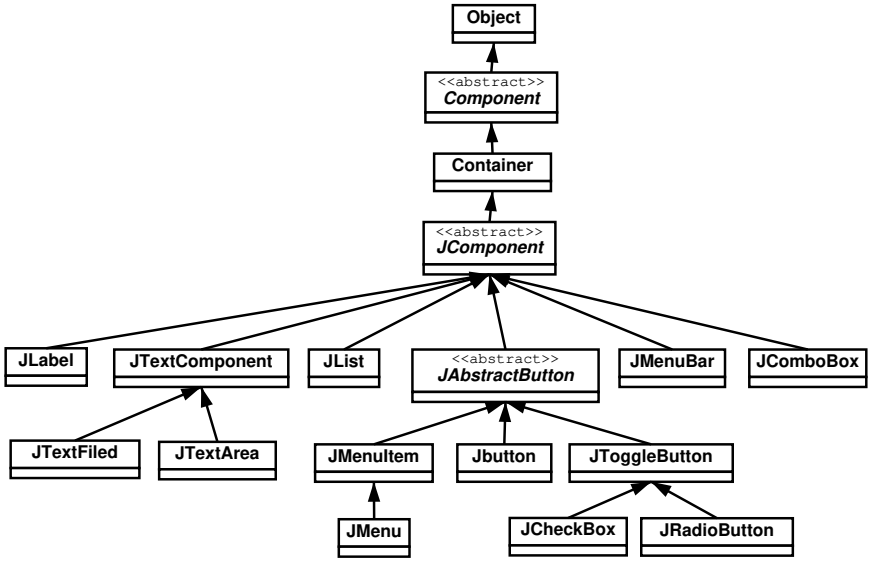
```
void setLeftComponent(JComponent comp) // séparation verticale
void setRightComponent(JComponent comp) // séparation verticale
void setTopComponent(JComponent comp) // séparation horizontale
void setBottomComponent(JComponent comp) // séparation horizontale
void setContinuousLayout(boolean b) // modification dyna-
mique de la taille des zones
void setOneTouchExpandable(boolean b) // masquable en un clic
```

`javax.swing.JDesktopPane`

Conteneur de haut niveau permettant d'obtenir un effet semblable au bureau multi-fenêtré. Il contient des fenêtres internes (JInternalFrame).

8.6 Les principales classes de composants Swing

Il existe une hiérarchie de classes swing, dont la classe de base est la classe JComponent.



Remarque : la classe Object appartient au paquetage java.lang et les classes Component et Container appartiennent au paquetage java.awt, toutes les autres classes font partie du paquetage javax.swing

`javax.swing.JComponent`

```
Dimension getPreferredSize()
boolean isEnabled()
boolean isVisible()
void setBackground(Color c)
void setBorder(Border border)
void setForground(Color c)
void setEnable(boolean enable)
void setPreferredSize(Dimension preferredSize)
void setToolTipText(String text) // info-
bulle, l'argument text peut être au format HTML.
void setVisible(boolean visible)
```

Remarque : on peut paramétrer les délais associés aux info-bulles :

```
ToolTipManager.sharedInstance().setInitialDelay(int delay); // en ms
ToolTipManager.sharedInstance().setDismissDelay(int delay); // en ms
```

Remarque : la méthode `setBorder()` accepte en paramètre un objet de type javax.swing.border.Border qui est une interface. on peut récupérer des objets de type Border par les méthodes de la classe `javax.swing.BorderFactory` :

```
Border createEmptyBorder()
Border createEtchedBorder()
Border createLoweredBevelBorder()
Border createRaisedBevelBorder()
Border createTitledBorder(String title)
```

8.6.1 Les composants visuels graphiques (cf. fig. 8.2 page 67).

`javax.swing.JLabel`

Affichage d'un texte statique éventuellement accompagné d'une icône :


```
JLabel()
JLabel(String text) // rem. : le texte peut être au format HTML
JLabel(String text, Icon image)
JLabel(Icon image, int horizontalAlignment) // SwingConstants.LEFT, CEN-
TER, RIGHT
JLabel(String text, Icon image, int horizontalAlignment)
String getText()
void setIcon(Icon image)
void setOpaque(boolean opaque)
void setText(String text)
```

`javax.swing.JButton`

Bouton constitué d'un texte et / ou d'une icône.


```
JButton()
JButton(String text)
JButton(String text, Icon image)
JButton(Icon image)
void setMnemonic(char c) // combinaison clavier <Alt + c>
void setPressedIcon(Icon image)
void setRolloverIcon(Icon image)
```

`javax.swing.JToggleButton`

Bouton à deux états généralement utilisé dans les barres d'outils

```
JToggleButton(Icon image)
boolean isSelected()
void setIcon(Icon image)
void setSelectedIcon(Icon image)
void setSelected(boolean select)
```

`javax.swing.JTextField`

Champs de saisie de texte.


```
JTextField()
JTextField(String text)
JTextField(int length) // taille en caractères
JTextField(String text, int length)
String getText()
void setColumns(int length)
setEditable(boolean editable) // true par défaut
void setFont(Font font)
void setText(String text)
```

Remarque : des méthodes `copy()`, `cut()` et `paste()` permettent de réaliser des échanges avec le presse-papiers.

`javax.swing.JPasswordField`

Variante de `JTextField`, champs de saisie masquée de texte.

```
JPasswordField(String text)
JPasswordField(String text, int length)
char[] getPassword() // correspond au getText() d'un JText-
Field mais est moins vulnérable aux attaques car ne renvoie pas un ob-
jet de type String.
void setEchoChar(char mask)
```

`javax.swing.JTextArea`

Zone de saisie de texte multi-lignes.

```
JTextArea()
JTextArea(in rows, int columns)
JTextArea(String text)
void append(String text)
```

`javax.swing.JCheckBox`

Case à cocher

```
JCheckBox(String label)
JCheckBox(String label, boolean checked)
void setSelected(boolean checked)
boolean isSelected()
```

`javax.swing.JList`

```
JList(Object[] listData) // type String
String getSelectedValue()
int getSelectedIndex()
```

`javax.swing.JComboBox`

Liste déroulante

```
JComboBox(String[] list)
JComboBox(Vector v)
void setEditable(boolean editable)
void setMaximumRowCount(int nbrows)
```

`javax.swing.JRadioButton`

Bouton radio : Les boutons radio sont ajoutés à un objet de type javax.swing.ButtonGroup permettant de réaliser l'exclusion mutuelle des boutons radio **et** au conteneur.

```
JRadioButton(string text)
```

8.7 Un exemple simple

Les interfaces graphiques remplacent avantageusement les programmes de test précédents en ligne de commande.

Exemple de la classe IHMPoint

```
// ================================================================
/**
* Description: exemple simple de mise en oeuvre d'une interface gra-
phique swing <br />
```

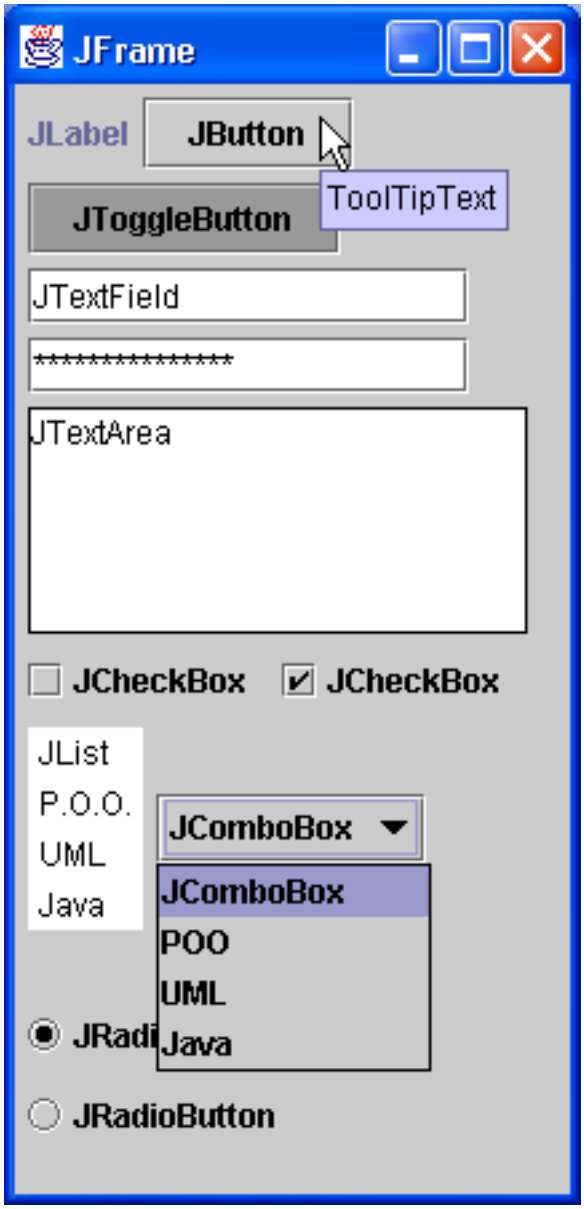


FIGURE 8.2 – Les composants Swing

```
* version 1.1
*/
package ihm;
import java.awt.FlowLayout;
import javax.swing.*; // JFrame, JButton, JTextField, JLabel
import java.awt.event.*; // ActionListener
import geometrie.Point;
// ======================================================================
// Rappel : un fichier source Java ne peut conte-
nir qu'une seule classe publique
public class IHMPoint extends JFrame {
    // attributs
    protected Point p; // le modèle
    protected JLabel jlTexte;
    protected JTextField jtfAbs, jtfOrd;
    protected JButton jbDeplacer, jbQuitter;
    public IHMPoint(String titre) {
        // appel du constructeur de la classe JFrame
```

```
        super(titre);
        p = new Point();
        jlTexte = new JLabel(" IHM classe Point avec swing ");
        jtfAbs = new JTextField(5);
        jtfOrd = new JTextField(5);
        setLayout(new FlowLayout()); // gestionnaire de présentation
        // gestion de la "X" de la fenêtre
        // peut être remplacée par un écouteur de fenêtre (WindowListe-
ner ou
        // WindowAdapter) dans le cas d'une gestion spécifique
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jbDeplacer = new JButton("Deplacer");
        jbDeplacer.addActionListener(new ActionDeplacer());
        jbDeplacer.setToolTipText("translation : x+5 , y+5");
        jbQuitter = new JButton("Quitter");
        jbQuitter.addActionListener(new ActionQuitter());
        getContentPane().add(jlTexte);
        getContentPane().add(jtfAbs);
        getContentPane().add(jtfOrd);
        getContentPane().add(jbDeplacer);
        getContentPane().add(jbQuitter);
        rafraichir();
        pack(); // pour obtenir une fenêtre de taille optimum
        setVisible(true);
    }

    // méthode de rafraichissement des champs de saisie
    // Remarque : cette méthode n'est pas public car elle est unique-
ment destinée à un usage interne
    protected void rafraichir() {
            jtfAbs.setText(String.valueOf(p.getAbscisse()));
            jtfOrd.setText(String.valueOf(p.getOrdonnee()));
        }
    }
    // inner class permettant de gérer les événements sur les élé-
ments graphiques
    class ActionDeplacer implements ActionListener {
        public void actionPerformed(ActionEvent ae) {
            p.deplacer(5, 5); // translation
            rafraichir();
        }
    }

    // inner class permettant de gérer les événements sur les élé-
ments graphiques
    class ActionQuitter implements ActionListener {
        public void actionPerformed(ActionEvent ae) {
            System.exit(0);
        }
    }
    // remarque : toute classe peut implementer une methode "main"
    public static void main (String args[]) {
```

```
            new IHMPoint("Fenêtre Point");
        }
    }
    // fin de la classe "IHMPoint" version 1.1
    // ======================== fin du fichier IHMPoint.java ============
```

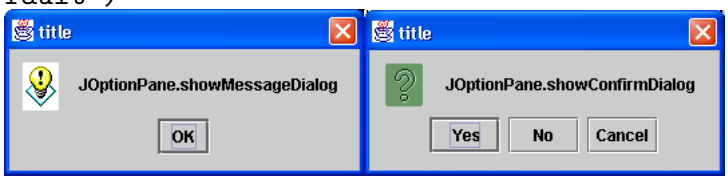
L'exécution de ce programme crée la fenêtre suivante à l'écran :

8.8 Les boîtes de dialogue et les boîtes de message

`javax.swing.JOptionPane`

La classe JOptionPane définit plusieurs méthodes statiques permettant de réaliser des boîtes de dialogue prédéfinies (de type Ok / Cancel ou Yes / No / Cancel ...).

```
static void showMessageDialog(JFrame parent, String message)
    // parent peut être null
static void showMessageDialog(JFrame parent, String mes-
sage, String title, int type)
    // type = JOptionPane.ERROR_MESSAGE, JOption-
Pane.INFORMATION_MESSAGE, JOptionPane.WARNING_MESSAGE, JOption-
Pane.QUESTION_MESSAGE, JOptionPane.PLAIN_MESSAGE
static void showMessageDialog(JFrame parent, String mes-
sage, String title, int type, Icon icon)
static int showConfirmDialog(JFrame parent, String message)
    // la réponse peut être : JOptionPane.YES_OPTION, JOption-
Pane.NO_OPTION, JOptionPane.CANCEL_OPTION, JOptionPane.CLOSED_OPTION
static int showConfirmDialog(JFrame parent, String mes-
sage, String title, int type)
    // type peut être : JOptionPane.DEFAULT_OPTION, JOption-
Pane.YES_NO_OPTION,  JOptionPane.YES_NO_CANCEL_OPTION, JOption-
Pane.OK_CANCEL_OPTION
static int showInputDialog(JFrame parent, String message)
static int showInputDialog(JFrame parent, String mes-
sage, String title, int type)
static int showInputDialog(JFrame parent, String mes-
sage, String title, int type, Icon icon, String[] options, String de-
fault )
```



javax.swing.JDialog

Boîte de dialogue (fenêtre secondaire), peut être modale (*modal*) c'est à dire bloquante par rapport à la fenêtre (principale) associée ou non modale (*modless*). La classe `JDialog` possède de nombreuses méthodes communes avec `JFrame`.

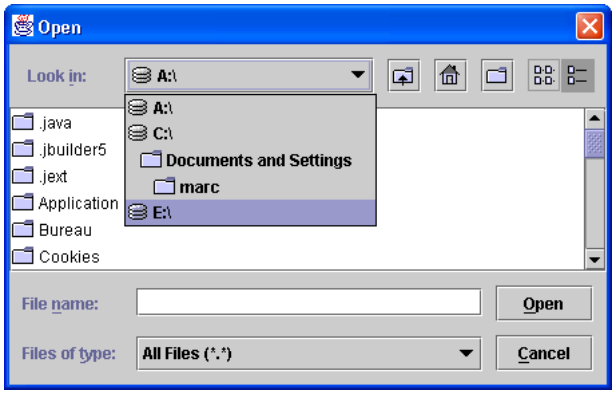
```
JDialog(JFrame parent)
JDialog(JFrame parent, String title, boolean modal)
    // si modal = true, la boîte de dialogue est bloquante
void dispose() // destruction
void hide() // cf. setVisible()
void setDefaultCloseOperation(int operation) // WindowCons-
tants.DO_NOTHING_ON_CLOSE, JFrame.EXIT_ON_CLOSE
void setSize(int width, int height)
void setResizable(boolean resize)
void setTitle(String title)
void setVisible(boolean visible)
void show() // cf. setVisible()
void toBack() // arrière plan
void toFront() // avant plan
```

javax.swing.JFileChooser

Boîte de dialogue prédéfinie pour l'ouverture ou la sauvegarde de fichiers.

```
JFileChooser()
File getSelectedFile()
void setFileFilter(FileFilter filter)
int showOpenDialog(JFrame parent)
int showSaveDialog(JFrame parent)
void setApproveButtonText(String text) // texte du bouton OK
void setCurrentDirectory(String dir) // répertoire HOME de l'utilisa-
teur par défaut
```

Remarque : cette classe définit également des constantes : `JFileChooser.APPROVE_OPTION` ...



8.9 Les menus et barres d'outils

`javax.swing.JMenuBar`

Barre de menu.

```
JMenu add(JMenu menu)
```

`javax.swing.JMenu`

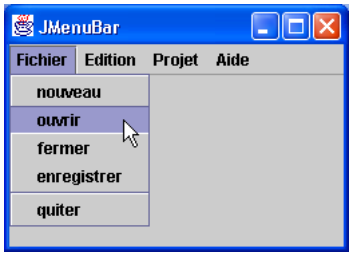
```
JMenu(String text)
JMenuItem add(JMenuItem menuItem)
void addSeparator()
void setEnabled(boolean enable)
void setMnemonic(int key) // key représente une touche du cla-
vier du type KeyEvent.VK_XXX (cf. tab. 8.2), l'option du menu est acces-
sible par Alt+key quand le menu (ou menu_item est visible.
```

`javax.swing.JMenuItem`

```
JMenuItem(String text)
void setAccelerator(KeyStroke ks) // raccourci clavier, ks est ob-
tenu par : KeyStroke.getKeyStroke(KeyEvent.VK_X, Inpu-
tEvent.CTRL_MASK); - le menu_item est alors acces-
sible par Ctrl+VK_X que le menu_item soit visible ou non.
void setEnabled(boolean enable)
```

Les classes `javax.swing.JCheckBoxMenuItem` et `javax.swing.JRadioButtonMenuItem` (à regrouper dans un ButtonGroup) sont des variantes :

```
JCheckBoxMenuItem(String text)
boolean getState()
void setState(boolean state)
```



`javax.swing.JPopupMenu`

Menu contextuel "surgissant" : il faut implémenter un `MouseListener` (ou étendre un MouseAdapter) en traitant les trois méthodes `mousePressed( )`, `mouseClicked( )` et `mouseReleased( )` en appelant la méthode `isPopupTrigger( )` sur l'objet `MouseEvent`.

```
add(JMenu menu)
show(JComponent comp, int x, int y)
```

`javax.swing.JToolBar`

Barre d'outils

```
void add(JComponent comp)
void addSeparator() // barre de séparation verticale
void setVisible(boolean visible)
void setFloatable(boolean floatable) // barre d'outils flottante
```

8.10 Autres éléments des interfaces graphiques

Les touches du clavier

L'ensemble des constantes liées au clavier est défini dans la classe KeyEvent.

constantes (*Virtual Key*)	constantes (*Virtual Key*)
VK_0 à VK_9	VK_HOME
VK_NUMPAD0 à VK_NUMPAD9	VK_INSERT
VK_A à VK_Z	VK_LEFT
VK_F1 à VK_F24	VK_NUM_LOCK
VK_ALT	VK_PAGE_DOWN
VK_ALT_GRAPH	VK_PAGE_UP
VK_CAPS_LOCK	VK_PRINTSCREEN
VK_CONTROL	VK_RIGHT
VK_DELETE	VK_SCROLL_LOCK
VK_DOWN	VK_SHIFT
VK_END	VK_SPACE
VK_ENTER	VK_TAB
VK_ESCAPE	

TABLE 8.2 – Les principales identifications des touches du clavier

Java 2D

Pour les problèmes de design plus élaborés, il existe la classe Graphics2D (et sa méthode setTransform( )) associée en particulier à la classe AfineTransform qui permet de réaliser des homothéties et des translations.

Chapitre 9

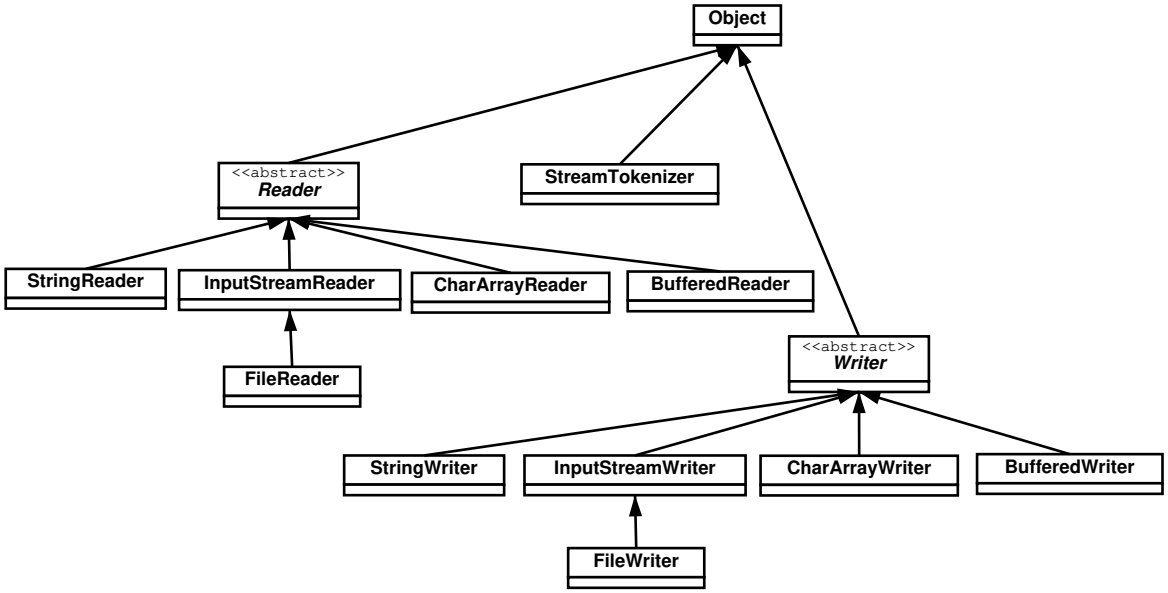
Les fichiers

Ce chapitre présente les manipulations de sauvegarde et restauration sur fichiers. L'ensemble des classes permettant les manipulations de flux constituent l'API java.io (input / output).

Les lectures et écritures sur fichiers en Java sont traitées comme des flux de caractères ou d'octets. Il existe deux familles de classes en fonction du type de flux. Les classes InputStreamReader et OutputStreamWriter fournissent un "pont" entre les deux types de flux et permettent de réaliser des conversions caractères <—> octets.

9.1 Lecture et écriture sur fichiers textes

Les classes Reader et Writer et leurs classes dérivées permettent de manipuler des caractères :



9.1.1 Manipulation de caractères

Il existe deux types de classes manipulant des caractères : les accès caractère par caractère (en lecture et en écriture) : FileReader et FileWriter et les accès par chaînes de caractères : BufferedReader et BufferedWriter.

java.io.FileReader extends Reader

permet une lecture caractère par caractère

```
FileReader(File file)
FileReader(string fileName)
close()
```

java.io.FileWriter extends Writer

permet une écriture caractère par caractère

```
FileWriter(String fileName)
close()
```

Remarques :

- ▷ Si le fichier existe déjà, son contenu est effacé.
- ▷ Si le fichier n'existe pas et si le chemin d'accès est valide (existe), le fichier est créé.
- ▷ Si le chemin n'est pas valide, l'exception FileNotFoundException est propagée.

java.io.BufferedReader extends Reader

permet une lecture ligne par ligne

```
BufferedReader(FileReader reader)
String readLine()
```

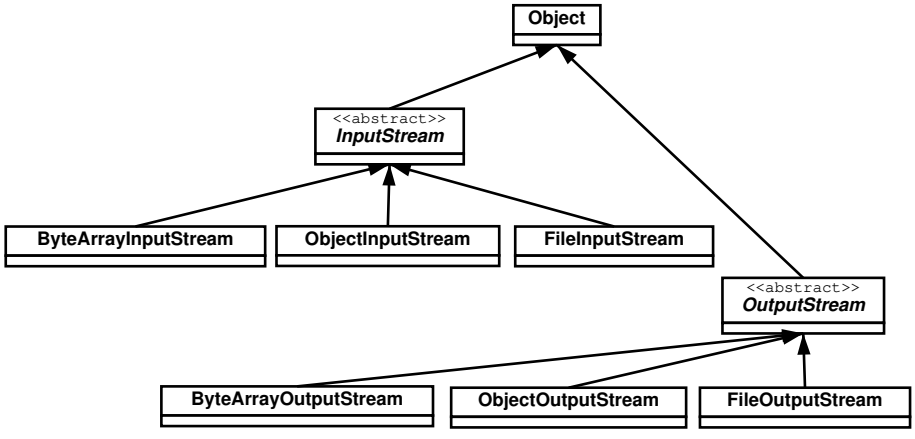
java.io.BufferedWriter extends Writer

pour une écriture ligne par ligne

```
BufferedWriter(FileWriter writer)
write(String str)
newLine() // pour écrire un retour à la ligne dans le fichier
```

9.2 Lecture et écriture sur fichiers binaires

Les classes InputStream et OutputStream et leurs classes dérivées permettent de manipuler des octets :



La plupart des classes manipulant des fichiers sont susceptibles de lancer des exceptions de type java.io.IOException ou d'une de ses sous-classes.

9.2.1 Manipulation d'octets

java.io.FileInputStream extends InputStream

pour une lecture octet par octet.

```
FileInputStream(String fileName)
FileInputStream(File name)
```

java.io.FileOutputStream extends OutputStream

permet une écriture octet par octet.

```
FileOutputStream(String fileName)
FileOutputStream(File name)
```

9.2.2 Le mécanisme de la sérialisation

Java propose un mécanisme puissant de sauvegarde d'objet sur fichier. Pour qu'un objet soit serialisable, il faut que sa classe d'appartenance implémente l'interface java.io.Serializable[1] et que l'ensemble de ses attributs soient eux-mêmes sérialisables[2].

java.io.ObjectInputStream

```
Object readObject()
```

java.io.ObjectOutputStream

```
void writeObject(Object obj)
```

les qualificateurs transient et volatile

Le qualificateur **transient** indique que l'attribut ne doit pas être sauvegardé lors de la sérialisation de l'objet.

9.3 Autres manipulations sur les fichiers

9.3.1 Les fichiers et répertoires

java.io.File

```
String getAbsolutePath()
String getName()
File getParent() // répertoire dans lequel se trouve le fichier[3]
String getPath()
boolean isDirectory()
```

1. Cette interface particulière ne déclare aucune méthode, qu'il faudrait sinon définir mais constitue un indicateur (drapeau ou flag) pour la machine virtuelle qui autorise le processus de sérialisation (mise à plat d'un objet).
2. La plupart des classes standard JAVA sont sérialisables.
3. getParent( ).getName( ) renvoie un objet String contenant le nom du fichier sans son répertoire.

```
boolean isFile()
int length() // taille du fichier en octets
String[] list() // liste des fichiers d'un répertoire
File[] listFiles() // liste des fichiers d'un répertoire
static File[] listRoots() // ensemble des répertoires racine
```

9.3.2 Autres classes de manipulations des fichiers

Remarque : la classe java.io.RandomAccessFile permet de lire et d'écrire des données à un endroit précis d'un fichier (cf. méthode seek( )) c'est à dire de manière "aléatoire" contrairement aux autres classes qui travaillent de manière séquentielle.

Remarque : les classes GZIPInputStream / GZIPOutputStream et ZIPInputStream / ZIPOutputStream du paquetage java.util.zip permettent de gérer directement des fichiers compressés.

9.3.3 java.nio

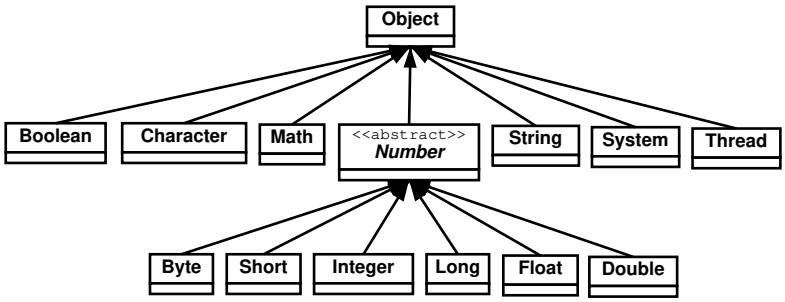
L'API java.nio (*new input - output*) apparu avec le JDK 1.4 offre un ensemble de classes permettant de réaliser des entrées / sorties (E/S) non bloquantes et sélectionnables avec des mécanismes de canaux (*channel*) et de tampon (*buffer*). Elle apporte par ailleurs des améliorations notables en terme de performances et de robustesse.

Chapitre 10

Compléments sur le langage

10.1 les classes qui encapsulent les types de base

Dans le paquetage java.lang, il existe un ensemble de classes qui encapsulent les types de base (cf. tableau 10.1) [1], en effet de nombreuses classes, en particulier les collections (cf. § 6.2 page 48) n'acceptent de manipuler que des objets. Le seul but de ces classes enveloppes est donc d'encapsuler un type simple dans un objet.



type	classe associée
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

TABLE 10.1 – les classes qui encapsulent les types de base

Certaines des méthodes de ces classes sont "*static*" et permettent de passer d'un type de base à un objet et réciproquement sans nécessairement instancier un objet de la classe associée.

Conversion d'un objet d'une classe enveloppe en un type de base

Chaque classe enveloppe fournit une méthode pour extraire la valeur simple de l'objet enveloppant :

```
char charValue() // classe Character
boolean booleanValue() // classe Boolean
```

1. De la même manière le type "void" est encapsulé par la classe "Void".

```
byte byteValue() // classe Byte
short shortValue() // classe Short
int integerValue() // classe Integer
long longValue() // Classe Long
float floatValue() // classe Float
double doubleValue() // classe Double
```

Conversion d'une chaîne de caractères en un type de base (type primitif) :

Les classes qui encapsulent les types de base possèdent des méthodes *static* permettant de convertir une chaîne de caractères en un type de base. Toutes ces méthodes renvoient un résultat dans un des types de base à partir de la chaîne de caractères passée en paramètre. Elle produisent une erreur (elles émettent une exception de type java.lang.NumberFormatException) en cas d'échec, ce sont les méthodes parseXxx(String s) :

```
static boolean getBoolean(String s) // classe Boo-
lean ("true" ou "false")
static byte parseByte(String s) // méthode de la classe Byte
static short parseShort(String s) // méthode de la classe Short
static int parseInt(String s) // méthode de la classe Integer
static long parseLong(String s) // méthode de la classe Long
static float parseFloat(String s) // méthode de la classe Float
static double parseDouble(string s) // méthode de la classe Double
```

Une variante de ces méthodes permet de travailler dans différentes bases pour les nombres entiers :

```
static byte parseByte(string s, int radix) // classe Byte
static short parseShort(String s, int radix) // classe Short
static int parseInt(String s, int radix) // classe Integer
static long parseLong(String s, int radix) // classe Long
```

Conversion d'un objet d'une classe enveloppe en une chaîne de caractères

```
String toString() // toutes les classes
```

Conversion d'un type de base en une chaîne de caractère

La classe String fournit un ensemble de méthodes permettant la conversion d'un des types de base en une chaîne de caractères (cf § 5.4.1 page 40). Par ailleurs les classes Integer et Long offrent un ensemble de méthodes static permettant des conversions dans différentes bases en une chaîne de caractères [2] :

```
static String toBinaryString(int i) // classe Integer
static string toBinaryString(long l) // classe Long
static String toHexString(int i) // classe Integer
static String toHexString(long l) // classe Long
static String toOctalString(int i)
static string toString(int i, int base)
```

2. Pour l'internationalisation (point décimal, valeur monétaire, ...), on peut utiliser les méthodes de la classe java.text.NumberFormat.

Limites des types

Chacune de ces classes définit deux constantes MIN_VALUE et MAX_VALUE.

Quelques autres méthodes

classe Character :

```
static boolean isWhiteSpace(char c)
static boolean isLetter(char c)
static char toUpperCase(char c)
```

10.2 le "documenteur" javadoc

Java offre un utilitaire permettant de générer automatiquement la documentation associée à une classe au format HTML. Le documenteur Javadoc s'appuie sur une variante des commentaires multilignes et utilise tous les commentaires commençant par "/**" :

```
/**
 *
 *
 */
```

Ces commentaires peuvent s'appliquer à une classe, une interface, un attribut, une méthode et sont placés juste avant la définition de l'élément

mot-clef	signification	s'applique à
@author	l'auteur du fichier	classe, interface
@deprecated	code appelé à disparaître	classe, attribut, méthode
@exception ou @throws	code susceptible de lever une exception	méthode ou constructeur
@param	paramètre de méthode	méthode ou constructeur
@return	valeur de retour	méthode
@see[3]	lien vers un autre code	tout élément
@serial	documente la sérialisation	i.e tous les éléments autres que ceux déclarés transient
@since	ancienneté du code	classe, attribut, méthode
@tag ou @link	liaison vers une autre classe, interface, méthode ou attribut	tout élément
@version	version du code ou du document	classe, interface

Table 10.2 – les mots-clef Javadoc

Quelques options de l'utilitaire javadoc :
- ▷ -author : prise en compte des balises de commentaire @author
- ▷ -version : prise en compte des balises de commentaire @version

10.2.1 Mémento HTML

```
<h1> ... </h1> à <h6> ... </h6> : (header) titres
<p> ... </p> : (paragraph) paragraphe
<br /> : (break) retour à la ligne explicite
```

<hr /> : (horizontal row) : ligne de séparation
 ... : (ordoned list) : liste numérotée (cf.)
 ... : (unordoned list) : liste à puce (cf.)
 ... : (list item) : élément d'une liste (cf. ou <ul>)
<dl> ... </dl> : (definition list) : liste de définitions (cf. <dt> et <dd>)
<dt> ... </dt> : (definition term) : élément à définir (cf. <dl>)
<dd> ... </dd> : (definition description) : définition d'un terme (cf. <dl>)
 ... : (anchor) : lien hypertexte
 : image (fichier externe et texte de substitution)

10.2.2 Javadoc et HTML

<h1> à <h6> ainsi que <hr /> ne doivent pas être utilisés dans les commentaires de documentation.

 peut être utilisé : l'image doit se trouver dans un sous répertoire "doc-files" dans le répertoire du fichier source (java).

10.3 la variable CLASSPATH

Les classes de base (paquetages java.lang, java.util, javax.swing ...) de l'environnement de développement JAVA n'ont pas à être renseignées dans la variable d'environnement CLASSPATH.

10.4 passage d'arguments à main( )

Comme dans la majorité des langages de programmation, il est possible de passer des arguments au programme.

Exemple :

```
public class TestArgs {
    public static void main(String[] args) {
        for ( i=0 ; i < args.length ; i++ ) {
            System.out.println(" argument[" + i + "] = " + args[i]);
        }
    }
}
```

Remarque : Contrairement au langage C, l'élément 0 de la liste des arguments est effectivement le premier argument et non le nom du programme.

10.5 Classes internes (inner-class) et classes anonymes

10.5.1 Classes internes

Les classes internes ou ***inner-class*** sont des classes imbriquées à l'intérieur d'une autre classe. Elles peuvent être assimilées à une extension de la notion de méthode. Leur intérêt principal réside dans

le fait que les méthodes d'une classe interne peuvent accéder aux attributs (d'instance) de leur objet englobant. Elle sont particulièrement utilisées lors de la réalisation d'interfaces graphiques.

Remarque : au même titre que les méthodes, les classes internes peuvent être déclarée "public".

10.5.2 Classes anonymes

Les classes anonymes sont des classes internes non nommées aux mécanismes de construction limités (héritage, constructeurs...). Elle peuvent être utilisée lors de la gestion des événements.

10.6 L'opérateur instanceof

L'opérateur ***instanceof*** renvoie un booléen : *true* si la référence pointe sur une instance de la classe précisée ou d'une de ses sous-classes.

Exemple :

```
String s1 = new String("Azertyuiop");
if(s1 instanceof String) // renvoie true
if(s1 instanceof Object) // renvoie true également
```

10.7 Quelques éléments pour les calculs mathématiques

L'ensemble des méthodes de la classe Math est *static*, cette classe n'est ni instanciable ni dérivable.

java.lang.Math

```
static double PI // constante π (PI)
static int abs(int i) // valeur absolue[4]
static double pow(double a, double b) // a puissance b
static double random() // générateur de nombre pseudo-
aléatoire de type double s'appuyant sur java.util.Random
static long round(double d) // arrondi à l'entier le plus proche
static double sqrt(double d) // racine carrée
```

Remarque : les classes BigDecimal et BigInteger du paquetage java.math permettent de manipuler des nombres de précision quelconque.

java.util.Random

Générateur de nombres pseudo-aléatoires : le générateur peut être initialisé avec une valeur, il permet de générer des nombres dans tous les types de base de Java.

```
Random(long seed) // seed est la graine du générateur de nombres pseudo-
aléatoires
int nextInt()
int nextInt(int max)[5] // permet de générer des nombres com-
```

4. il existe également :
   ```
   static double abs(double d)
   static float abs(float f)
   static long abs(long l)
   ```

5. de même, il existe `nextDouble( )`, `nextFloat( )`, `nextLong( )`, `nextByte( )`

```
pris entre 0 (inclus) et max (exclus)
```

10.8 Les manipulations de dates

JAVA fournit deux classes principales permettant de manipuler les dates : la classe java.util.Date qui encapsule un instant de temps (le nombre de millisecondes depuis le 1er janvier 1970 [6]) et la classe java.util.GregorianCalendar qui permet la conversion entre un instant de temps et le jour, le mois, l'année, ...

java.util.Date

```
Date()
```

java.util.GregorianCalendar extends Calendar

```
GregorianCalendar() // utilise l'horloge du système
GregorianCalendar(int year, int month, int day)
int get(int field) // field peut être Calendar.HOUR_OF_DAY, Calen-
dar.HOUR, Calendar.MINUTE, Calendar.SECOND, Calendar.MILLISECOND, Calen-
dar.DAY_OF_YEAR, Calendar.DAY_OF_WEEK, Calendar.DAY_OF_MONTH
static Calendar getInstance()
Date getTime()
void set(int year, int month, int day)
void set(int field, int value) // méthode héritée de Calen-
dar field peut être Calendar.YEAR, Calendar.MONTH, Calendar.DATE, Calen-
dar.DAY_OF_WEEK [7] void setTimeZone(TimeZone value)
```

Remarque : la classe java.text.DateFormat permet la manipulation des dates et heures sous la forme de chaînes de caractères.

10.9 Les entrées-sorties clavier et écran

Ce chapitre présente rapidement les entrées-sorties clavier et écran via une console (un terminal). L'interaction avec l'utilisateur se faisant en mode ligne de commande, cette approche est à réserver à des situations ne nécessitant pas d'interface graphique.

La classe **System,** et plus exactement les méthodes des attributs de type *stream,* permet de réaliser ces manipulations : elle possède trois attributs importants :

```
public static InputStream in;
public static PrintStream out;
public static PrintStream err;
```

6. La méthode statique System.currentTimeMillis( ) permet de récupérer la même information.
7. exemples :

```
set(Calendar.YEAR, 2005);
set(Calendar.MONTH, Calendar.JULY);
set(Calendar.DATE, 27);
set(Calendar.DAY_OF_WEEK, Calendar.WEDNESDAY)
```

L'ensemble des affichages à l'écran peut être assuré par les différentes variantes des méthodes print( ) et println( ) de la classe PrintStream.

Exemples :

```
System.out.print();
System.out.println()[8];
```

Les opérations de saisie au clavier peuvent s'avérer délicates, la classe java.util.Scanner apparue avec la version 5 du langage permet de réaliser les opérations les plus communes.

Exemple :

```
Scanner sc = new Scanner(System.in);
int i = sc.nextInt();
```

8. Remarque concernant print( ) et println( ) : afin d'assurer une portabilité maximale des programmes (en particulier entre les environnements Windows et Linux), il est recommandé de ne pas utiliser le caractère d'échappement '\n' pour les retour à la ligne car son implémentation diffère selon les systèmes. Si on souhaite effectuer un retour à la ligne il suffit d'utiliser println( ) au lieu de print( ).

Chapitre 11

Interface graphique AWT

11.1 Présentation

L'API AWT[1] permettait dans les premières versions du langage de réaliser des interfaces graphiques utilisateur (GUI) constituées de composants "lourds" liés à la couche graphique du S.E. : le rendu était dépendant de la plate-forme.

La bibliothèque de classes AWT utilise les ressources système qu'elle encapsule par des abstractions. L'apparence des composants graphiques AWT est régie par les ressources graphiques du système d'exploitation. Les interfaces AWT ont donc des apparences différentes en fonction du S.E. sur lequel s'exécute l'application.

 N.B. : L'aspect visuel d'une interface AWT étant dépendant de la plate-forme sur laquelle s'exécute l'application (contrairement à Swing), il est recommandé de ne pas mélanger des composants Swing avec des composants AWT. Néanmoins, dans certains cas les classes AWT n'ont pas été réécrites car elles n'interfèrent pas directement avec le rendu visuel.

11.1.1 Quelques classes AWT encore utiles

java.awt.Graphics (public abstract class Graphics extends Object)

Classe abstraite permettant de gérer un contexte graphique.

```
void drawString(String text)
void drawLine()
void drawRect()
void fillRect()
void drawRoundRect()
void fillRoundRect()
void drawPolygon()
void drawOval()
void fillOval()
void drawArc()
void fillArc()
void copyArea()
void clearRect()
void setFont(Font fnt)
```

1. *Abstract Windowing Toolkit*

```
void setColor(Color clr)
Color getColor()
```

java.awt.Font (public class Font extends Object)

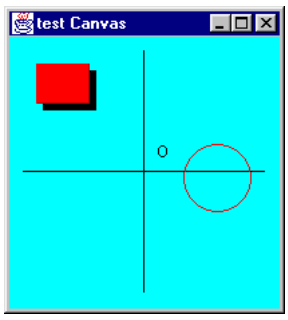
Permet de définir une police de caractères (par ex. : serif, sans-serif, monospace...) et s'utilise avec la méthode setFont(Font f) de la classe JComponent.

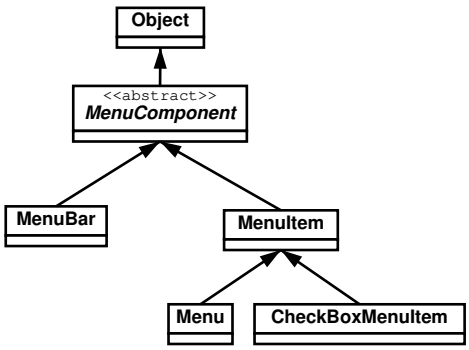

```
Font(String fontName, int style, int size) // le para-
mètre style peut être : Font.PLAIN (normal), Font.BOLD ou Font.ITALIC
deriveFont(float size)
deriveFont(int style)
String getFamily()
```

Remarque : Font[ ] fonts = GraphicsEnvironment.getLocalGraphicsEnvironment( ).getAllFonts( ) ; permet de récupérer l'ensemble des polices disponibles sur la plate-forme.

java.awt.Canvas (public class Canvas extends Component)

Zone rectangulaire permettant de dessiner.





java.awt.Color

java.awt.Toolkit

Classe utilitaire

```
static Toolkit getDefaultToolkit()
Dimension getScreenSize()
```

java.awt.Point

Pour information : classe Point du paquetage java.awt :

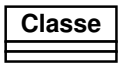
```
boolean equals(Object obj) // méthode héritée de Object
double getX()
double getY()
void setLocation(int x, int y)
void translate(int dx, int dy)
String toString() // redéfinition de la méthode toString() héri-
tée de Object
```

Chapitre 12

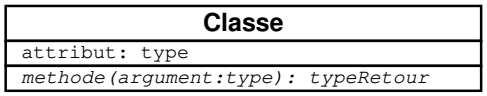
Mémento de la notation UML

12.1 éléments constitutifs des diagrammes de classe

classe

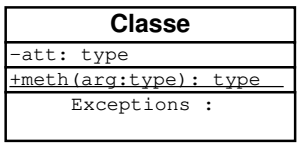


attributs et méthodes

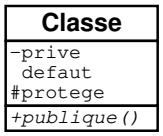


Remarque : Un attribut ou une méthode sera souligné si la portée est celle de la classe (static en Java) et non celle d'une instance.

Remarque : on peut ajouter, si nécessaire, un compartiment supplémentaire qui sera alors nommé (Ex. : Exceptions).



les niveaux de visibilité (- , , # , +)



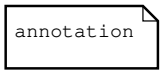
Remarque : le niveau par défaut correspond au niveau "package" aussi appelé niveau "implementation".

classe abstraite (ayant un stéréotype)

interface (ayant un stéréotype)

paquetages

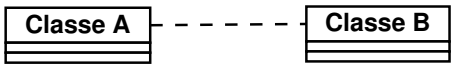
annotation



usage : note explicative ou commentaire.

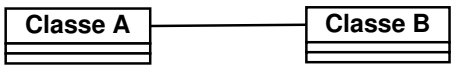
12.2 relations entres classes

dépendance



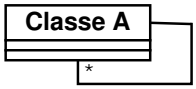
La dépendance correspond à la présence de la classe ClasseB comme argument dans la signature d'une méthode de ClasseA (ex. classe Window et classe Event) ou lorsqu'une variable locale d'une méthode de ClasseA est de type ClasseB. On peut préciser la direction de la dépendance [1].

association



Remarque : En général, on indiquera la multiplicité (cardinalité) de la relation : par exemple : 1 (exactement 1), n (exactement n), 0..n, 1..n, n..m (toute valeur comprise entre les deux bornes), *, 0..*, 1..* (toute valeur ou toute valeur supérieure ou égale à un minimum).

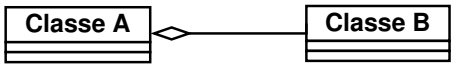
Remarque : une association (ou agrégation) réflexive (*"reflexive association"*) est possible :



N.B. : Une association est unidirectionnelle (on peut préciser son sens par une flèche) : un attribut de type ClasseB est présent dans ClasseA. (les associations bidirectionnelles affectent la maintenabilité du code).

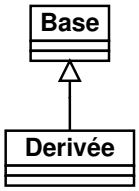
1. *Remarque* : on peut avoir une dépendance entre paquetages.

agrégation ou composition



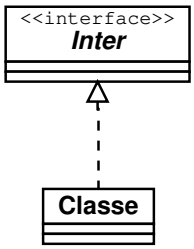
Remarque : l'agrégation est un type particulier d'association : la classe associée est alors une partie d'un tout (la cardinalité est précisée du côté de la classe B).

héritage (*"inheritance"*) ou généralisation



(mot-clef JAVA "extends").

réalisation : implémentation d'une interface en Java



(mot-clef JAVA "implements").

Chapitre 13

Installation du jdk

13.1 Sous Windows

Quelques conseils pour l'installation du jdk sous Windows

Remarque : pour l'édition des fichiers .java, vous pouvez utiliser un éditeur quelconque à partir du moment ou il fournit une reconnaissance syntaxique. Certains outils offre des possibilités de compilation ou d'exécution en restant dans la même interface (cf. Geany) et il est également possible d'utiliser les IDE de type NetBeans ou Eclipse.

Pour pouvoir utiliser Java sous Windows, vous devez disposer des éléments suivants :

- ▷ le JDK version 8 ou supérieure (actuellement la version 12).
- ▷ la documentation du langage au format HTML (recommandé).
- ▷ la variable CLASSPATH correctement renseignée.

Sous Windows, on peut travailler en ligne de commande dans une fenêtre terminal ("command") ou utilser un des outils cités précédemment.

Voici les différentes étapes de la procédure d'installation

1. Rassembler les fichiers nécessaires : j2sdk-8_1-win.exe et j2sdk-8_1-doc.zip (disponibles en téléchargement sur Internet : http ://www.oracle.com/technetwork/java/javase/downloads/index.html).
2. Installer le JDK de Sun (version 7 ou supérieure) : j2sdk-8_1-win.exe
3. Installer la documentation au format HTML : j2sdk-8_1-doc.zip (Remarque : la documentation complète est accessible directement en ligne : http ://download.oracle.com/javase/8/docs/api/index.ht
4. Créer l'arborescence des répertoires de travail (par exemple c :\java , c :\java\lib, c :\java\src et c :\java\javadoc)
5. Dans le répertoire c :\java, créer les fichiers ci-dessous : jdk.bat, javac.bat et javadoc.bat à l'aide (par exemple) du bloc-notes de Windows (Notepad).
6. Ouvrir une fenêtre Terminal (fenêtre de commandes).
7. se placer dans le répertoire de travail (cd c :\java)
8. Lancer jdk.bat en ligne de commande : taper jdk puis appuyer sur la touche [Enter]
9. Tester la compilation (javac Essai.java), l'exécution (java test.Essai) et la génération de la documentation de votre programme (javadoc Essai.java) en ligne de commande depuis le répertoire contenant vos sources java (c :\java\src).

Les 3 fichiers à créer :

a) jdk.bat // c'est le fichier qui devra être lancé au démarrage dans le terminal.

```
  echo pour le jdk
  doskey
  REM il faut que le repertoire contenant les fi-
chiers .bat soit places avant le "jdk" dans la variable PATH
  path=%PATH%;c:\java;c:\jdk8\bin
  REM ne pas oublier d'ajouter le repertoire de tra-
vail (c:\java\lib) dans la variable CLASSPATH
  set CLASSPATH=c:\jdk8\lib;c:\java\lib
  cd c:\java\src
```

b) javac.bat // ce fichier permettra la compilation des fichiers .java en fichiers .class (byte-code).

```
REM l'option -d indique le repertoire ou seront crees les fichiers .class
  c:\jdk8\bin\javac -classpath %CLASSPATH% -
d c:\java\lib %1 %2 %3 %4 %5 %6 %7
```

c) javadoc.bat // pour générer la documentation

```
  c:\jdk8\bin\javadoc -version -author -noindex -notree -
d c:\java\javadoc %1 %2 %3 %4 %5 %6 %7
```

Remarque : si la variable CLASSPATH est correctement renseignée, il n'est pas utile d'écrire un fichier java.bat pour lancer la machine virtuelle (JVM), celle-ci se lancera par défaut avec l'option -classpath %CLASSPATH%.

13.2 Sous Linux

Pour paramétrer le JDK sous Linux, le plus simple est de modifier un des fichiers script de démarrage : la configuration présentée ici utilise le fichier .bashrc (shell bash).

```
# pour le jdk
if [! -d $HOME/lib ]
then
   mkdir $HOME/lib
fi
if [! -d $HOME/doc ]
then
   mkdir $HOME/doc
fi
export CLASSPATH="$HOME/lib"
alias javac="javac -classpath $CLASSPATH -d $HOME/lib "
alias java="java -classpath $CLASSPATH "
alias javadoc="javadoc -version -author -noindex -notree -d $HOME/doc "
```

Chapitre 14

Bibliographie et références Internet

Ouvrages

[Ber01] - ***Swing la synthèse, développement des interfaces graphiques en Java*** - V. Berthie, J-B. Braud - Dunod, 2001

[Bon99] - ***Java, de l'esprit à la méthode,*** *2e édition* - M. Bonjour, G. Falquet, J. Guyot, A. Le Grand - Vuibert, 1999

[Boo00] - ***Le guide de l'utilisateur UML*** - G. Booch, J. Rumbaugh, I. Jacobson - Eyrolles, 2000

[Cla98] - **Java *la synthèse,*** *2e édition* - G. Clavel, N. Mirouze, S. Munerot, E. Pichon, M. Soukal - InterEditions, 1998

[Cla03] - **Java *la synthèse, concepts, architectures, framworks,*** *4e édition* - G. Clavel, N. Mirouze, S. Munerot, E. Pichon, M. Soukal - Dunod, 2003

[Eck98] - ***Thinking in* Java** - B. Eckel - Prentice-Hall, 1998

[Fla 02] - **Java *in a Nutshell,*** *Fourth Edition* - D. Flanagan - O'Reilly, 2002 - | - **Java *in a Nutshell, manuel de référence pour Java 2 (SDK 1.4)*** *4e édition* - D. Flanagan - O'Reilly, 2002

[Jac99] - **Java *by example*** - J. R. Jackson, A. L. McClellan - The Sun Microsystems Press - Java Series, 1999

[Lar00] - ***Éléments sur* Java** - P. Laroque - 2000

[Nie02] - ***Learning* Java,** *Second Edition* - P. Niemeyer, J. Knudsen - O'Reilly, 2002 - | - ***Introduction à* Java,** *2e édition* - P. Niemeyer, J. Knudsen - O'Reilly, 2002

[San99] - ***Bibliothèque de classes Java 2*** - K. Sankar - CampusPress, 1999

Sites Web sur Java

http ://www.oracle.com/technetwork/java/index.html - *le site de référence*

http ://download.oracle.com/javase/tutorial/ - *les tutoriels de* Sun

http ://download.oracle.com/javase/tutorial/collections/index.html - *les collections et dictionnaires*

http ://java.sun.com/docs/books/jls/ - *the Java Language Specification (jls)*

http ://www.jmdoudoux.fr/java/dej/chap-presentation.htm *- un site en français très bien fait*

http ://download.oracle.com/javase/7/docs/api/ *- la documentation officielle*

http ://download.oracle.com/javase/tutorial/ui/features/components.html *- retrouver d'un coup d'oeil l'ensemble des composants swing*

http ://fr.wikipedia.org/wiki/Java_(langage) *- présentation générale, historique et liens intéressants*

http ://java.developpez.com/cours/ *- de nombreuses ressources en français*

http ://download.oracle.com/javase/tutorial/uiswing/TOC.html *- pour la conception d'interfaces graphiques Swing.*

Chapitre 15

Glossaire

API : *Application Programming Interface*

GUI : *Graphical User Interface*

HTML : *HyperText Markup Language*

IEEE : *Institute of Electrical and Electronics Engineers*

IHM : Interface Homme-Machine

JEE : *Java Entreprise Edition*

JME : *Java Micro Edition*

JSE : *Java Standard Edition*

JDK : *Java Development Kit*

JRE : *Java Runtime Environment*

JSP : *Java Server Page*

JSF : *Java Server Faces*

JVM : *Java Virtual Machine*

O.S. : *Operating System*

plaf : *pluggable look and feel*

RMI : *Remote Method Invocation*

SDK : *Software Development Kit*

S.E. : Système d'Exploitation

UML : *Unified Modeling Language*

URL : *Uniform Ressource Locator*

Index