Fiches Java

Alain Plantec
Université de Bretagne Occidentale
U.F.R. Sciences et Techniques
Département Informatique
mailto:alain.plantec@univ-brest.fr

2009-2012

Table des matières

Chapitre 1

Introduction

Dans ces fiches vous trouverez ce qu'il faut retenir du cours de programmation Java. Elles doivent être complétées avec les notes que vous prenez en cours mais aussi par les exemples et les développements des travaux dirigés et pratiques.

Pour aborder ce cours il est nécessaire d'avoir déjà une expérience minimale de programmation par exemple, en C, en C++, en Smalltalk, ...

Les fiches rappellent tout d'abord les principes de l'orienté objet avant de présenter les éléments importants du langage Java.

1.1 Quelques sources documentaires

On trouve de la documentation sur Java partout sur internet :

Ressources en français chez Eteks http://www.eteks.com

Autres documents pédagogiques en français : http://rmdiscala.developpez.com/cours/

Documentations chez Oracle

- Portail Java http://www.oracle.com/technetwork/java/javase/documentation/
- IHM avec Swing http://docs.oracle.com/javase/tutorial/uiswing/TOC.html
- Conventions http://www.oracle.com/technetwork/java/codeconv-138413.html

Portail *Tutorialspoints* http://www.tutorialspoint.com/java/

D'autres tutoriels en anglais http://www.javabeginner.com

Bonne intro du dessin 2D : http://duj.developpez.com/tutoriels/java/dessin/intro/

Programmer un compilateur avec java

Jlex/Cup de Princeton

- Analyseur syntaxique http://www.cs.princeton.edu/~appel/modern/java/CUP/
- Analyseur lexical http://www.cs.princeton.edu/~appel/modern/java/JLex/

JavaCC http://java.net/projects/javacc

1.2 Un rappel sur les objets

1.2.1 Objet, classe et instance

Un objet est l'instance d'une classe.
La classe sert de modèle (de "moule") pour la création de ses instances.
Les objets communiquent par envoi de messages :

- un message est envoyé à un receveur,
- son nom est celui de la méthode à exécuter,
- le nom est accompagné des paramètres nécessaires à son exécution.

La réception d'un message par un objet provoque une réaction. Ce que fait un objet en réaction à la réception d'un message est spécifié dans une une méthode ou fonction membre. Dans une fonction membre on trouve donc le code exécuté lors de la réception du message.
Une classe peut être définie à partir d'une autre classe par héritage. Une classe B qui hérite d'une classe A est une spécialisation de A. Une instance de B est donc aussi une instance de A.

1.2.1.1 Anatomie d'un objet

Un objet se constitue d'un ensemble de méthodes et d'un ensemble d'attributs :

- un d'attribut (ou variable d'instance, champ ou encore propriété) représente une propriété fondamentale ;
  - à un moment t, l'ensemble des valeurs des attributs d'un objet définissent l'état de l'objet.
- une méthode (opération, fonction membre ou fonction d'instance) spécifie un comportement lors de la réception d'un message particulier.

Tout objet possède une identité unique qui lui permet d'être distingué des autres objets.

1.2.1.2 Principe d'encapsulation

L'encapsulation est un principe fondamental :
Les propriétés d'un objet ne sont manipulées que par l'objet propriétaire ; Les autres objets n'ont de visibilité que sur ce que la classe de l'objet rend visible : en principe uniquement des méthodes publiques.

Conséquences :

- pour utiliser une classe, on se s'intéresse qu'à son interface et non à sa mise en œuvre ;
- un objet est autonome, les frontières sont clairement établies.

1.3 La classe

Une classe est un modèle utilisé pour la construction d'objets. Elle comprend une partie privée et une partie publique.
La partie privée comprend :

- les variable d'instance (ou encore attributs ou propriétés),
- le corps des méthodes.

La partie publique décrit l'interface que met en œuvre la classe (son interface) avec le nom des méthodes publiques et leurs paramètres.
Par instanciation, une classe permet la création d'objets conformes.

1.3.1 Instanciation

C'est la création d'un objet à partir d'une classe. On dit que l'objet est une instance de la classe. L'instanciation se déroule en 3 phases :

- création d'un espace mémoire permettant de stocker les valeurs des attributs ;
- initialisation de l'espace mémoire par exécution d'un constructeur ;
- attribution d'une référence unique à l'objet créé ; cette référence constitue l'identité l'objet ;

1.3.2 Héritage

Une classe peut être spécifiée par spécialisation ou raffinement d'une classe existante. On dit que la nouvelle classe hérite de la classe existante.

Les classes d'un système forment ainsi une ou des hiérarchies de classes.
Chaque sous-classe hérite des attributs et des méthodes de sa super-classe.
Une sous-classe peut ajouter de nouveaux attributs et de nouvelles méthodes.
Elle peut aussi redéfinir ou spécialiser le comportement (la mise en œuvre des méthodes) de sa super-classe.

1.4 Polymorphisme

Le polymorphisme est une propriété des objets qui est liée à l'héritage :

– si $inst$ est une instance de la classe C, alors $inst$ est aussi une instance de toutes les super-classes dont hérite la classe C.

Par exemple : la classe *Carre* hérite de *Quadrilataire* et *Quadrilataire* hérite de *Geometrique*. Une instance de Carre est aussi une instance de Quadrilataire et de Geometrique.
Une instance de *Geometrique* peut être créé par instanciation de l'une de ses sous-classes, directes ou indirectes.

1.5 La liaison dynamique

La liaison dynamique fait que c'est à l'exécution, lors de la réception du message, que le comportement (la méthode) à exécuter est choisie :

– c'est la méthode de la classe instanciée qui est exécutée (celle de l'objet qui reçoit le message) ;

– ce sont bien les objets qui répondent aux messages.

Dans un langage comme le C, la liaison est statique :

– le lien entre un nom de fonction et le code à exécuter est effectué par le compilateur, avant l'exécution.

Polymorphisme + liaison dynamique : ces deux caractéristiques fondamentales autorisent une très forte généricité des développements objets.

1.6 Résumé de ce que nous avons vu

– Un système à objets est constitué d'objets qui communiquent et interagissent par envoi de message ;

– Java est un langage permettant de modéliser des systèmes à objets ;

– Java est un langage Classe-Instance : un objet est une instance d'une classe ;

– Une classe sert de moule ou patron pour la création de ses instances ;

– Une classe encapsule la définition des données (champs) et celle du comportement (réponse aux messages) ;

– Les champs sont typés ;

– Le comportement est défini par les méthodes ou fonctions membres de la classe ;

– Le lien est dynamique, c'est à dire que le comportement est déterminé pendant l'exécution, la méthode à exécuter est recherchée à partir de la classe de l'objet qui reçoit le message.

Chapitre 2

Compiler et exécuter ses premières classes

2.1 Hello World en java

Le programme *Hello World* affiche simplement "Hello World !" sur la console.

2.1.1 La version classique

Voici le contenu du fichier HelloWorld.java. En Java, la seule façon de programmer du code exécutable est de déclarer une classe. Donc, même pour l'affichage d'un message sur la console, le programme comprend la définition d'une classe simple avec une méthode *main*.

```
1 /*
2  * La classe HelloWorld class met en oeuvre une application
3  * qui affiche "Hello World !"
4 */
5 class HelloWorld {
6    public static void main(String[] args) { // Attention a bien reproduire fidelement cette ligne
7       System.out.println("Hello World !"); // Affichage de "Hello World !" sur la sortie standard
8    }
9 }
```

Ce programme montre :

- 1 à 4 : des commentaires entre /* et */ comme en C ;
- 5 à 9 : la déclaration d'une classe dont le nom est *HelloWorld* avec
  - 6 à 8 : la définition de la fonction principale *main* ; cette méthode est obligatoirement déclarée *static* et doit avoir un seul argument de type tableau de *String* (on expliquera tout ça plus tard) ;
  - 7 : l'envoi du message *println* à l'objet référencé par la variable *System.out* ;

Pour pouvoir exécuter votre programme vous devez d'abord compiler votre classe. Pour cela, saisissez votre code dans un fichier texte (à l'aide de votre éditeur préféré). Ensuite, vous sauvegardez votre code dans un fichier nommé *HelloWorld.java*.

Respectez scrupuleusement cette règle de nommage : le nom du fichier doit être construit avec le nom de votre classe (qui définit *main*), le préfixe est le nom de la classe et le suffixe ou l'extension et ".java". Cela donne le nom de fichier *HelloWorld.java*.

Pour compiler, vous ouvrez une console dans le répertoire dans lequel se trouve le fichier *HelloWorld.java* et vous saisissez la ligne de commande :

```
1 javac HelloWorld.java
```

Attention, *javac* est le nom du compilateur et on compile un fichier qui contient la définition d'une classe. On obtient le fichier *HelloWorld.class*.

Pour exécuter, toujours dans votre console :

```
1 java HelloWorldApp
```

Attention, ici, *java* est le nom de la machine virtuelle java, l'argument passé est le nom d'une classe dans laquelle se trouve la fonction $main$.

2.1.2 La version applet

Applet Un applet est une application exécutable depuis votre navigateur web. Il nécessite la définition d'une classe java et doit être "exécuté" depuis un navigateur web. On charge donc une page HTML sur son navigateur et la machine virtuelle java est exécutée par le navigateur.

La classe Java de l'applet. Voici le contenu du fichier HelloWorldApp.java. On note qu'il n'y a pas de méthode *main* mais une redéfinition de la méthode *paint*. Attention, il s'agit d'une méthode d'instance. Cette méthode doit prendre un seul argument de type *Graphics* :

```
1 import java.applet.Applet;
2 import java.awt.Graphics;
3
4 public class HelloWorldApp extends Applet {
5   public void paint(Graphics g) { // Attention fonction membre non static
6     g.drawString("Hello world!", 50, 25); // Affichage de la chaine a la position 50@25
7   }
8 }
```

Ce programme montre :

- 5 et 6 : une nouvelle forme de commentaire, de // à la fin de la ligne comme en C++ ;
- 1 à 2 : des imports pour réutiliser des classes pour construire un Applet ; il y a beaucoup, énormément de classes à réutiliser en Java ; il suffit de les importer et on peut les réutiliser dans nos propres programmes ;
- 4 à 8 : la déclaration de la classe *HelloWorldApp* qui hérite de *Applet* ;
- 5 à 7 : la fonction membre (ou méthode) qui décrit ce que doit faire une instance de *HelloWorldApp* quand elle reçoit le message *paint* avec la référence d'une instance de *Graphics* (nommée *g*) passée en argument ;
- 6 : envoi du message *drawString* à l'objet référencé par *g* avec une chaine de caractères et deux entiers pour les coordonnées ;

Pour compiler on utilise le compilateur *javac* :

```
1 javac HelloWorldApp.java
```

Le code HTML pour l'applet. Le code HTML suivant est contenu dans le fichier *hello.html*. Il permet de charger l'application depuis un navigateur.

```
1 <HTML>
2   <HEAD> <TITLE> A Hello World applet </TITLE> </HEAD>
3   <BODY>
4     <APPLET CODE="HelloWorld.class" WIDTH=150 HEIGHT=25>
5     </APPLET>
6   </BODY>
7 </HTML>
```

Pour exécuter l'applet, il suffit de charger le fichier *hello.html* dans votre navigateur web. Vous n'avez donc pas à programmer de fonction *main* comme pour la version simple. C'est déjà fait pour un applet :

- dans le code HTML, vous avez indiqué le nom de la classe à utiliser (ligne 4) ;
- le navigateur va construire une instance de cette classe et lui envoyer le message *paint* en lui passant en argument la référence de l'objet qui représente une partie de la fenêtre.

Un applet un peu plus riche. Les *println* sont visibles sur la sortie standard avec l'outil *appletviewer*. La classe suivante permet de suivre ce qui se passe en redéfinissant les méthodes *init*, *start*, *stop* et *destroy*.

```
1 import java.applet.Applet;
2 import java.awt.Graphics;
3
4 public class HelloWorldApp2 extends Applet {
5   public void init() {
6     System.out.println("(1X) Chargement : init()");
7   }
```

```
8     public void start() {
9        System.out.println("(nX) Page ouverte : start()");
10    }
11    public void stop() {
12       System.out.println("(nX) Page quittee : stop()");
13    }
14    public void destroy() {
15       System.out.println("(1X) Applet detruite : destroy()");
16    }
17    public void paint(java.awt.Graphics g) {
18       g.drawString("Hello world !", 50, 25);
19    }
20 }
```

Ce programme montre :

- 1 à 2 : comme dans la version précédente, des imports pour réutiliser des classes pour construire un Applet ;
- 4 à 20 : la déclaration de la classe *HelloWorldApp2* qui hérite de *Applet* ;
- 5 à 16 : les méthodes *init*, *start*, *stop* et *destroy* ; en fait ces méthodes sont déjà définies dans la classe *Applet* ; il s'agit donc de redéfinitions pour changer ou compléter ou *spécialiser* le comportement de ces méthodes ;

2.2 Dessiner un polygone

Vous pouvez vous-même créer une fenêtre et dessiner dans cette fenêtre. Pour cela, on a besoin d'un objet pour la fenêtre et d'un objet placé dans la fenêtre pour dessiner. On va bien sur réutiliser des classes existantes pour faire tout ça. C'est normal, en programmation objet, on passe beaucoup de temps à découvrir ce qui existe et à le réutiliser. Par exemple pour dessiner un polygone :

```
1  // On reutilise les packages existants pour dessiner
2  import java.awt.Graphics;
3  import javax.swing.JComponent;
4  import javax.swing.JFrame;
5
6  // On a besoin d'une classe pour dessiner dans la fenetre
7  class MyCanvas extends JComponent {
8     public void paint(Graphics g) { // Fonction appelee pour dessiner
9        int[] x = {10,120,120,10}; // Tableau pour les x
10       int[] y = {40,40,80,80}; // Tableau pour les y
11       g.drawPolygon (x, y, x.length); // Dessin du polygone
12    }
13 }
14
15 public class DrawPolygon {
16    public static void main(String[] args) { // Notre fonction main
```

```
17        JFrame window = new JFrame() ; // On cree une fenetre
18        window.setBounds(30, 30, 300, 300) ; // On indique la taille de la fenetre
19        window.getContentPane().add(new MyCanvas()) ; // On ajoute un composant pour dessiner
20        window.setVisible(true) ; // Et hop, on demande àla fenetre de s'ouvrir
21    }
22 }
```

Ce programme montre :

- 1 : un commentaire ;
- 2 à 4 : des imports pour réutiliser des classes pour construire la fenêtre ;
- 7 à 13 : la déclaration de votre classe *MyCanvas* pour dessiner ; elle réutilise ce qui est déjà fait dans la classe *JComponent* ;
- 8 à 12 : la redéfinition de la méthode *paint*, qui est déjà définie dans *JComponent* ; c'est dans cette méthode que vous dessinez :
  - la méthode *paint* prend en argument la référence sur un objet de la classe *Graphic* ; cet objet comprend le message *drawPolygon* avec des coordonnées pour les points passées en argument ;
  - 9 à 10 : on prépare une série de coordonnées pour les x et les y ;
  - 11 : on envoie le message *drawPolygon* en passant en argument les coordonnées en x, en y et le nombre de coordonnées ;
- 15 à 22 : votre classe *DrawPolygon* qui contient la fonction principale *main* :
  - 17 : la référence *window* est initialisée avec une instance de *JFrame* ;
  - 18 : le message *setBound* est envoyé à l'objet référencé par *window* pour indiquer une taille à la fenêtre ;
  - 19 : une instance de note composant MyCanvas est créeet ajoutée à la fenêtre ;
  - 20 : le message *setVisible* est envoyé à l'instance de *JFrame* référencée par window pour ouvrir la fenêtre.

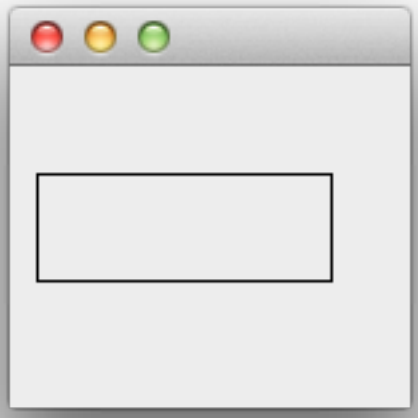
Pour compiler on utilise à nouveau le compilateur *javac*. Ouvrez une console dans le répertoire où vous avez sauvegardé votre fichier *DrawPolygon.java* et saisissez la commande suivante :

```
1 javac DrawPolygon.java
```

Pour exécuter :

```
1 java DrawPolygon
```

Vous devez voir cette fenêtre s'ouvrir avec votre polygone.



2.3 Conventions d'écriture

Il est très important de respecter des convention pour l'écriture de vos programme. Les conventions peuvent être particulières suivant le contexte, le projet ou votre entreprise. Oracle édite ses propres conventions (voir http://www.oracle.com/technetwork/java/codeconv-138413.html).

Voici quelques points très importants :

- *Sensibilité à la casse* : Java différencie les minuscules des majuscule pour le noms et les symboles prédéfinis (*Aaa* et différent de *aaa*) ;
- *Nom des classes* : Nommez les avec une première lettre en majuscule ; capitalisez vos noms comme par exemple *CarLight* au lieu de *Carlight* ou de *carlight* ;
- *Nom des méthodes* : Ils commencent par une minuscule et doivent être capitalisé ;
- *Nom du fichier* : Il est construit par le nom de la classe en préfixe et le suffixe '.java' (voir les explications dans le chapitre 2.1.1) ;
- *Méthode principale main* : doit être exactement : public static void main(String args[])"

2.4 Résumé de ce que nous avons vu

- Les exemples donnés dans cette partie montrent que toutes les actions d'un programme sont écrites dans des méthodes ;
- Java met en œuvre deux sortes de méthodes : (1) les méthode d'instance et (2) les méthodes de classe (ces dernières sont déclarées à l'aide du mot clé *static* ;

- pour envoyer un message à un objet, on utilise une notation ”pointée” (ex. g.drawPolygon (x, y, x.length) avec *g* comme receveur du message *drawPolygon*) ;
- Pour une application ”classique” la méthode de classe *main* déclare les envois de messages effectués au lancement du programme ; pour un applet, on déclare les envois de message effectués au démarrage dans la méthode *paint*.
- On a pu très facilement dessiner dans une fenêtre en réutilisant des classes ; il faut donc passer du temps pour découvrir ce qui existe déjà et qu’on peut réutiliser ;
- Pour bien écrire vos programmes, il faut respecter des conventions ; choisissez de préférences des conventions standards.

Chapitre 3

Déclarer et utiliser des variables

On a trois sortes de variables :

– les champs dans le classes (variables d'instance ou de classe),

– les variable locales aux traitements,

– les paramètres formels pour les fonctions d'instance ou de classe.

Les paramètres ne sont pas vraiment des variables. Ils se déclarent de la même façon. Par contre, ils ne sont pas affectés de la même manière. Les champs et les variables locales sont affectés à l'aide de l'opérateur d'affectation alors que les paramètres formels sont affectés par le passage d'une valeur ou d'une référence lors de l'envoi d'un message.

3.1 Forme générale d'une déclaration

Java est un langage typé statiquement. En conséquence, toute variable est typée lors de sa déclaration. Nous avons déjà vu des déclarations. Par exemple dans la classe DrawPolygone, la variable locale *window* de la fonction *main* déclarée de la façon suivante : `JFrame window = new JFrame();` (voir le chapitre 2.2 page 8).

Voici d'autres exemples dans la classe *Point* :

```
1 public class Point {
2    int x;
3    int y;
4    publix moveTo (int xcoord, int ycoord) {
5       // ...
6    }
7    // ...
8 }
```

Dans la classe *Point*, nous voyons la déclaration des variables d'instance *x* et *y* et la déclaration des paramètres formels *xcoord* et *ycoord* de la fonction membre *moveTo*. On observe qu'une déclaration se constitue du type suivi du nom de la variable.

Le choix du nom est libre. La seule contrainte est qu'il doit obligatoirement commencer par une lettre. Il est cependant conseillé de respecter des règles utiles notamment pour la lisibilité et l'intelligibilité du code. Vous pouvez par exemple, vous reporter aux conventions de codage d'Oracle (voir http://www.oracle.com/technetwork/java/codeconv-138413.html).

On dispose de trois catégories de types en Java :

- les types primitifs (ou de base) pour les caractères, les entiers, les flottants,...
- les types composés : les tableaux et les types référence (lorsque le nom du type est celui d'une classe) ;
- les interfaces. (lorsque le nom du type est celui d'une interface)

Dans ce chapitre nous nous intéressons plus particulièrement aux types primitifs et aux tableaux. Nous abordons les types référence sur une classe ou une interface en fin de chapitre. Nous verrons plus tard plus précisément la déclaration et l'utilisation des classes et des interfaces.

3.2 Types primitifs

On dispose d'un ensemble fini de types primitifs. Ils sont donnés dans le tableau suivant :

Type	Signification	Taille	Plage de valeurs possibles
char	Caractère Unicode	2	'\u0000' ... '\uffff' (0 à 65535)
byte	Entier très court	1	-128 ... +127
short	Entier court	2	-32 768 ... +32 767
int	Entier	4	$-231 \approx -2,147 \times 109$ ... $+231 - 1 \approx 2,147 \times 109$
long	Entier long	8	$-263 \approx -9,223 \times 1018$ ...$+263 - 1 \approx 9,223 \times 1018$
float	Réel simple	4	$\pm 2 - 149 \approx 1.4 \times 10 - 45$ ... $\pm 2128 - 2104 \approx 3.4 \times 1038$
double	Réel double	8	$\pm 2 - 1074 \approx 4,9 \times 10 - 324$ ... $\pm 21024 - 2971 \approx 1,8 \times 10308$
boolean	Booléen	1	true ou false

- le type *long* occupe 64 bits (contrairement à 32 bits en C) ;
- le type *char* occupe 16 bits en Java (contrairement à 8 bits en C) ;
- le type *byte* occupe 8 bits ;

3.2.1 Les littéraux

Pour affecter une valeur à une variable de type primitif, il suffit de saisir la valeur. Un littéral est une représentation directe dans le code source d'une valeur fixe. Il s'agit d'une valeur dite littérale qui est reconnue par le compilateur (les types primitifs sont aussi appelés types littéraux). Le compilateur est capable d'analyser les caractères que vous avez saisi et de construire en interne une valeur constante correspondante. Cette valeur pourra être affectée à une variable de type primitif.

3.2.1.1 Les littéraux entiers

Ce sont les plus utilisés, il représentent les entiers. Tous les types entiers sont signés.

Si on veut préciser qu'une valeur est de type *long* alors on peut ajouter le caractère *L* à la fin de l'entier.

Pour les types *byte*, *short* et *int* on peut tout simplement saisir la valeur décimale entière (en base 10). Il est aussi possible d'utiliser des hexadécimaux ou des valeurs binaires :

```
47 ; // Decimal
456456654999L ; // Decimal long
0xA12399FF ; // Hexadecimal
0b11111000011 ; // Binaire
```

3.2.1.2 Les flottants

Si on veut utiliser un type *float* alors on termine le nombre par le caractère *F* ou *f* sinon, le compilateur crée systématiquement un *double*. Ajouter un *D* ou un *d* à la fin d'un flottant pour indiquer explicitement qu'il s'agit d'un littéral de type *double* est aussi possible. On peut utiliser la notation scientifique (avec le caractère *e* ou le caractère *E*).

Voici quelques exemples :

```
987.6 ; // Flottant double
987.6D ; // Meme Flottant double
987.6d ; // Meme Flottant double

987.6F ; // Flottant float
987.6f ; // Meme Flottant float

9.867E2 ; // Flottant double en notation scientifique
9.867e2 ; // Meme Flottant double en notation scientifique
```

3.2.1.3 Les caractères

Les variables de type *char* peuvent contenir un caractère Unicode, encodé en UTF-16. Ces caractères peuvent être directement saisis (si votre clavier le permet ou bien encodé à l'aide d'une syntaxe *escape*. Le caractère est dans tous les cas compris entre deux quotes simples :

```
'A';
'z';
'\u0108'; // C en unicode
'\u00ED' // S en unicode
```

les valeurs littérales des caractères spéciaux sont les mêmes qu'en C : '\b' pour *backspace*, '\f' pour un *form feed*, '\n' pour un saut de ligne, '\t' pour une tabulation, '\" pour le caractère ', '\"' pour le caractère ", '\\' pour le caractère \,...

3.2.2 Les booléens

On dispose des deux symboles réservés *true* et *false* pour représenter respectivement la valeur *vrais* et *faux*.

```
boolean done = false;
// du code ...
done = true;
```

3.2.3 Les chaînes de caractères

Une chaine de caractères est encodées par une suite de caractères entre deux double quote :

```
"AAAA"; Chaine de caracteres avec 4 caracteres 'A'
'AB\u0108'; // Chaine ABC avec le C en unicode
'Hello\tWorld !\n' // Chaine de caracteres avec une tabulation '\t' et un retour a la ligne '\n'
```

Lorsque le compilateur analyse une chaine de caractères entre double quote, il construit automatiquement une instance de la classe *String* :

```
"AAAA".equals(new String("AAAA")); // La valeur de cette expression est true
```

Le message *equals* permet de tester l'égalité entre deux objets. Pour une chaine de caractères, l'envoi de ce message retourne *true* si les caractères du receveur sont les même et dans le même ordre que ceux de l'argument. La classe *String* est donc particulière. Nous la verrons plus en détail plus tard.

3.2.4 Création et affectation d'un type primitif

On peut directement affecter sa valeur à la variable dans la déclaration ou par une affectation :

```
int x = 10;
int y;
y = 5;
origine.moveTo(0,0);
corner.moveTo(x,y);
```

Ce code montre :

– 1 : la déclaration de la variable *x* de type *int* et son initialisation directe avec la valeur 10 ;
– 2 : la déclaration simple de la variable y ;
– 3 : l'affectation de la valeur 5 à la variable y ;
– 4 et 5 : le passage de valeurs pour les paramètres formels de type *int xcoord* et *ycoord* de la fonction *moveTo* de la classe *Point* (voir sa déclaration dans le chapitre 3.1).

3.2.5 Valeur par défaut

3.2.5.1 Cas des champs dans une classe

Un champ (variable d'instance ou de classe) déclaré mais non initialisé est affecté d'une valeur par défaut par le compilateur. De manière schématique, cette valeur est 0 ou bien la référence *null* suivant le type du champ :

Type	Valeur par défaut
byte	0
short	0
int	0
long	0
float	0.0f
double	0.0d
char	'\u0000'
Type référence	null
boolean	false

3.2.5.2 Cas des paramètres formels

Les paramètres formels n'ont pas de valeur par défaut. Il est obligatoire de passer une valeur pour tous les paramètres formels lors d'un envoi de message.

3.2.5.3 Cas des variables locales

Pour les variables locales, aucune valeur par défaut n'est affectée par le compilateur. Une valeur doit obligatoirement être explicitement affectée avant toute utilisation d'une variable locale. Si une variable locale est utilisée avant d'avoir été initialisée, alors le compilateur renvoie une erreur.

3.3 Type tableau

Un tableau comprend une suite d'éléments de types compatibles et accessibles par leur position (index entier). Dans un tableau, on peut stocker des éléments de n'importe quel type (primitif, tableau, type référence). Un tableau n'est pas un type primitif, c'est une instance dont la classe hérite de Objet. Une variable tableau stocke ainsi une référence vers une instance particulière.

Un tableau se déclare, se crée à l'aide de l'opérateur *new* et s'utilise en accédant aux éléments à l'aide des crochets :

```
class DemoTableauFlash {
   public static void main(String[] args) {
      // Deux possibilites pour la declaration
      int [] tab1 ; // declaration d'un tableau d'entiers
      int tab2 [] ; // declaration d'un tableau d'entiers
      Object tab3 [] = { new Object (), new String("coucou") } ; // Tableau cree litteralement

      tab1 = new int[3] ; // Creation d'un tableau d'entier (taille 3) et affectation a tab1
      tab1[0] = 10 ; // Affectation de 10 dans la premiere case du tableau
      tab1[1] = 11 ; // Affectation de 11 dans la deuxieme case du tableau
      tab1[2] = 12 ; // Affectation de 12 dans la troisieme case du tableau (la derniere)

      // Affichage de la taille du tableau tab1 (c'est 3) et de la valeur du premier element (c'est 10)
      System.out.println("Taille de tab1 : " + tab1.length) ;
      System.out.println("Entier dans tab1 a l'indice 0 :" + tab1[0]) ;
      System.out.println(tab1.toString()) ;
      System.out.println(tab1.clone().toString()) ;
   }
}
```

3.3.1 Déclaration de tableau

On utilise un couple de crochets ouvrant et fermant ([ ]) par dimension. On n'indique pas les bornes, elles sont fixées lors de la création du tableau.

Les crochets peuvent être avant ou après le nom de la variable tableau :

```
int [] tableau_d_entiers ; // Tableau d'entiers a une dimension
int tableau_d_entiers_aussi [] ; // Tableau d'entiers a une dimension (2de syntaxe)
char [][] matrice_de_caracteres ; //Tableau de caracteres a deux dimensions
```

3.3.2 Création d'un tableau

Il y a différentes possibilités suivant qu'il s'agisse d'un tableau à une ou plusieurs dimensions.

3.3.2.1 Tableau à une dimension

Un tableau est créé avec une certaine taille. La taille d'un tableau ne peut pas être modifiée. Elle est accessible à l'aide de l'attribut *length*.

Une variable qui référence un tableau est initialisés par une affectation ou directement lors de leur déclaration. Pour un tableau à plusieurs dimensions, seule la dernière dimension

n'est pas obligatoirement initialisée.

La création peut s'effectuer de deux façons :

– à l'aide d'un tableau littéral (donné par une suite d'éléments entre parenthèses, les éléments sont séparés par une virgule) ;

– à l'aide de l'opérateur *new*.

Voici quelques exemples de créations et d'initialisation de tableaux :

```
int [] arrayOfInts = { 32, 87,3, 589, 12, 1076, 2000, 8};
char [] arrofchar = new char [10];

int [] tabofint;
tabofint = new int [100];
```

La taille de création ne peut pas être négative par contre il est possible de créer un tableau vide :

```
(new int [0]).length() == 0; // expression vrais
```

3.3.2.2 Tableau à plusieurs dimensions

Pour la déclaration, on indique autant de [] que de dimensions. Pour la création, on dispose de deux possibilités :

– toutes les dimensions sont de taille fixe ;

– ou seule la dernière dimensions est de taille inconnue.

Si la dernière dimension est laissée non créée, alors les éléments de la dernière dimension doivent être posés dans le tableau avant leur utilisation. Dans ce cas, la taille de la dernière dimension peut varier.

```
char [][] matofchar = new char [10][10]; // Les deux dimensions sont fixes
// On peut directement utiliser les éélments du tableau
matofchar[0][0] = 'A';
matofchar[0][1] = 'B';

char [][] matofchar2 = new char [2][]; // La derniere dimension est laissee non initialisee
// Alors, elle doit l'etre avant utilisation des cases
matofchar2[0] = new char [30];
matofchar2[1] = new char [10];
matofchar2[0][0] = 'A';
matofchar2[0][1] = 'B';
```

3.3.3 Manipulation d'un tableau

Les indices commencent à 0. L'attribut *length* de type entier donne la dimension du tableau fixée lors de sa création.

```
MaClasse tab[] = new MaClasse[N];
for (int i = 0; i < tab.length; i++)
   tab[i] = new MaClasse();

int mat[][] = new int[10][10];
for (int i = 0; i < tab.length; i++)
   for (j = 0; j < tab[i].length; j++)
      tab[i][j] = i * j;
```

Lors de l'accès à un élément du tableau, la validité de l'indice est vérifié à l'exécution. En cas d'erreur, l'exception *ArrayIndexOutOfBoundsException* est levée.

Exemple :

```
int [] ti = new int [10];
ti[10] = 3;
```

à l'exécution cela donne sur la console :

```
java.lang.ArrayIndexOutOfBoundsException : 1
            at HelloWorld.main(HelloWorld.java :35)
```

Le type des éléments stockés dans un tableau doit être compatible avec le type déclaré du tableau. Si ce n'est pas le cas, alors l'exception *ArrayStoreException* est lancée par le système.

3.4 Type référence de classe ou d'interface

Si le type d'une variable est le nom d'une classe ou d'une interface, alors le contenu de la variable peut être une référence sur une instance d'une classe ou d'une interface compatible.

```
class Person implements Affichable {
   String nom;
   public void affiche () { System.out.print("Person("+this.nom+")"); }
}

interface Affichable {
   void affiche();
}
```

Ce code montre :

- 1 à 3 : la déclaration de la classe *Person* qui met en œuvre l'interface *Affichable* (ne vous inquietez pas si vous ne comprenez pas tout, nous expliquerons tout ça plus tard) ;
- 2 : la déclaration du champ nom de type *String* (type référence sur une instance de la classe *String*) ;
- 3 : la déclaration de la méthode *affiche* ;

– 6 à 8 : la déclaration de l'*interface Affichable* ;

On dispose du symbole particulier *null* pour initialiser une variable de type référence sur une classe ou sur une interface pour indiquer que la variable ne référence pas d'instance valide. Attention, si vous envoyez un message à un objet par le biais d'une variable référence contenant *null*, alors c'est une erreur qui est traitée à l'exécution.

Voyons maintenant la déclaration de variables référence pour *Person* et *Affichable* :

```
1 Person p1 = new Person();
2 p1.affiche();
3 Affichable p2 = new Person();
4 p2.affiche();
5 p1 = null; // On ne peut plus envoyer de message àl'objet reference par p1
```

Ce code montre deux déclarations de variable dont le type est une référence :

– 1 : la variable *p1* est de type référence sur une instance de *Person* (directement initialisé ici avec une nouvelle instance) ;

– 2 : envoi du message *affiche* à l'objet référencé par *p1* ;

– 3 : la variable *p2* est de type référence sur une instance d'une classe qui met en œuvre l'interface *Affichable* ; comme *Person* met en œuvre l'interface *Affichable*, alors, on peut l'initialiser avec une instance de *Person* :

– 4 : envoi du message *affiche* à l'objet référencé par *p2* ;

– 5 : affectation de la valeur particulière *null* à *p1* ;

3.5 Résumé de ce que nous avons vu

– Nous avons vu qu'il y a trois sortes de variables, les champs dans les classes, les variables locales et les paramètres formels :

– Nous avons décrit la forme générale des déclarations et comment on initialise et manipule des variables de type primitif et tableau ;

– Nous avons introduits les notions de type référence sur une instance ;

Chapitre 4

Définition des classes

La classe est la brique de base de tout langage objet. Ce chapitre décrit les aspects syntaxiques pour la déclaration d'une classe sans trop entrer dans les détails. L'objectif est de pouvoir très rapidement passer aux TP et TD.

4.1 La classe

4.1.1 Les différentes formes de déclaration d'une classe

Déclaration d'une classe simple
On utilise le mot clé *class*.

```
1 class NomDeClasse {
2     // Declaration des champs, des methodes, des constructeurs, ...
3 }
```

Une classe simple dérive implicitement de la classe *Object*.
this est la référence de l'objet courant qui exécute la méthode ou l'expression d'initialisation. *this* contient la référence du receveur.

Exemple : la classe *Point* suivante modélise un point dans un espace à deux dimensions. Elle comprend deux champs x et y : La classe *Magnitude* décrit des objets qui peuvent être comparés suivant une dimension linéaire (on parle aussi d'ampleur ou de grandeur). Les nombres, les dates et les caractères sont des magnitudes. Voici la définition de la classe *Magnitude* :

```
1 class Magnitude {
2 }
```

Déclaration d'une sous-classe
On utilise le mot clé *extends*.

```
1 class NomClasseDerivee extends NomSuperClasse {
2 }
```

Une classe ne peut hériter que d'une seule super classe.

Exemple : la classe *Number* décrit le concept de nombre. Un nombre est une magnitude, on peut définir la classe *Number* comme une sous-classe de *Magnitude* :

```
1 class Number extends Magnitude {
2 }
```

Déclaration d'une classe qui met en œuvre une ou plusieurs interfaces
On utilise le mot clé *implements*

```
1 class NomDeClasse implements NomInterface //, NomInterface2,...
2 {
3 // ...
4 }
```

Une interface déclare un ensemble de méthodes. Une classe peut mettre en œuvre plusieurs interfaces (on expliquera plus tard plus en détail ce qu'est une interface, pourquoi et comment on s'en sert).

Par exemple, la classe Magnitude peut être déclarée comme mettant en œuvre l'interface *Serializable* :

```
1 class Magnitude implements Serializable{
2 }
```

Déclaration d'une sous-classe qui met en œuvre une ou plusieurs interfaces

```
class NomDeClasse extends NomDeSuperClasse implements NomInterface //, NomInterface2, ...
{
// ...
}
```

Par exemple, la classe Double est une spécialisation de Number qui est déclarée comme mettant en œuvre les interfaces *Serializable* et *Comparable*.

```
class Double extends Number implements Serializable, Comparable {
}
```

4.1.2 Variantes

Par défaut, une classe simplement déclarée avec le mot clé *class* n'est visible, utilisable seulement par les autres classes du même *package* dans laquelle elle est déclarée et il est possible d'en déclarer des sous-classes. Des précisions peuvent être indiquées concernant la visibilité d'une classe, la possibilité de déclarer des sous-classes ou d'instancier la classe. La définition d'une classe peut être éventuellement complétée d'un symbol avant le mot clé *class* qui peut prendre une ou plusieurs des valeurs suivantes :

- *public* : une seule classe ou interface *public* par fichier source ; une classe *public* peut être utilisée à l'extérieur du *package*.
- *final* : une classe ne peut pas hériter d'une classe déclarée avec *final* ;
- *abstract* : pour une classe abstraite (non instanciable) ; peut comporter une ou plusieurs méthodes *abstract* ; une classe qui comporte une méthode abstract, doit être déclarée abstract.

Une classe *abstract* ne peut pas être aussi déclarée *final*.
Si une méthode est déclarée *abstract* alots la classe doit obligatoirement déclarée *abstract*.

Exemples, la classe *GraphicPoint* suivante ne peut pas avoir de sous-classe et la classe Person est abstraite (elle ne peut pas être instanciée) :

```
final class GraphicPoint {
// ...
}
abstract public class Person {
// ...
}
```

- On ne peut pas déclarer de sous-classe à *GraphicPoint* ;
- La classe *Person* avoir des sous-classes dans d'autres packages que dans celui ou Person est déclarée ;
- La classe *Person* ne peut pas être directement instanciée (on est obligé de déclarer et d'instancier des sous-classes concrètes) ;

4.2 Les champs d'une classe

Les champs sont les variables membres d'une classe.

4.2.1 Forme générale d'une déclaration

Une classe *Classe1* peut déclarer un champ qui est une référence à une classe *Classe2* déclarée après *Classe1*.

```
class Classe1
{
 UnType champ1;
 UnType champ2, champ3;
 UnType champ5 = valeurOuExpression; // Initialisation d'un champ
 // Creation d'une reference sur un tableau
 UnType tableau1 [];
 // Allocation d'un tableau de taille n initialise avec les n valeurs
 UnType tableau2 [] = {valeur1, valeur2, /*..., */ valeurn};
}
```

Exemple, la classe *Fraction* spécifie des nombres avec un numérateur et un dénominateur. La classe *Fraction* hérite donc de *Number* et déclare deux champs de type entier.

```
class Fraction extends Number {
   int numerator;
   int denominator;
}
```

Il est possible d'initialiser directement le numérateur et le dénominateur avec des valeurs par défaut :

```
class Fraction extends Number {
   int numerator = 0;
   int denominator = 1;
}
```

4.2.2 Variantes

Les champs peuvent être de portée et de nature différentes. La portée et la nature d'un champ peuvent être précisés avant le nom du type de champ par un symbol. Les mots-clés suivants peuvent être utilisés :

- pour la portée, *public*, *protected* ou *private* : un champ *public* est accessible partout où est accessible sa classe. Un champ *protected* est accessible par les autres classes du même package que sa classe, et par les classes dérivées de sa classe. Un champ *private* n'est accessible qu'à l'intérieur du corps de sa classe ;
- *static* : si un champ est *static*, il est créé en un unique exemplaire quelque soit le nombre d'instances de sa classe ;

- *final* : un champ *final* n'est pas modifiable une fois initialisé : c'est une constante qui peut prendre une valeur initiale différente d'un objet à l'autre ;
- *transient* : un champ *transient* sert à indiquer aux méthodes gérant la persistance que le champ n'est pas persistant (sert à éviter de sauvegarder des champs ne servant qu'à des calculs intermédiaires) ;
- *volatile* : Un champ *volatile* permet d'être sûr que deux threads auront accès de manière ordonnée à ce champ.

Les champs d'une classe *Classe1* dont le contrôle d'accès est *protected* sont accessibles par les classes dérivées de *Classe1*, mais aussi par les classes du même package que *Classe1*.

Par défaut, les champs d'une classe *Classe1* sont accessibles uniquement par les autres classes du même package que *Classe1*.

4.2.3 Initialisation des champs

Si le champ est déclaré comme *static*, alors l'initialisation est effectuée au chargement de la classe, sinon l'initialisation est effectuée lors de la création de l'instance, avant l'exécution du constructeur.

Les champs non initialisés prennent obligatoirement la valeur par défaut de leur type (voir la chapitre 3.2.5).

Les champs d'instance ou *static* peuvent être aussi initialisés à leur déclaration ou bien initialisés dans un bloc d'initialisation respectivement d'instance ou *static*.

4.2.3.1 Bloc d'initialisation statique

Le corps d'une classe peut comporter un ou plusieurs blocs d'initialisation *static*.

Ces blocs sont exécutés au chargement d'une classe, et permettent d'exécuter des opérations sur les champs *static*. Ils sont exécutés dans l'ordre de déclaration et ne peuvent manipuler que les champs *static* déclarés avant le bloc.

```
1  class Classe1
2  {
3     // Declaration de champs static
4     static int champ1 = 10;
5     static int champ2;
6     static // un bloc d'initialisation static
7     {
8        champ2 = champ1 * 2;
9     }
10 }
```

4.2.3.2 Bloc d'initialisation d'instance

Le corps d'une classe peut comporter un ou plusieurs blocs d'initialisation d'instance.

Comme pour les constructeurs, ces blocs sont exécutés à la création d'un nouvel objet dans l'ordre de déclaration.

Peut être utile pour factoriser du code commun à tous les constructeurs.

```
1  class Classe1
2  {
3     // Declaration d'un champ comptant le nombre d'instances creees
4     static int nombreInstances = 0 ;
5     int champ1 ; // champ d'instance
6     { // Bloc d'initialisation d'instance
7        champ1 = 10 ;
8        nombreInstances++ ;
9     }
10 }
```

4.3 Déclaration des méthodes

4.3.1 Forme générale d'une déclaration

Les formes possibles pour la déclaration d'une méthode sont les suivantes :

```
1  class Classe1 {
2      TypeOuVoid methode1 (TypeP1 param1Name /*,... , TypePN paramNName */) {
3        // code de methode1
4      }
5      TypeOuVoid methode2 (/*...*/) throws TypeThrowable /*, TypeThrowable2 */ {
6            // code de methode2
7      }
8      abstract TypeOuVoid methode3 (/* ... */) ; // Pas de code
9      native TypeOuVoid methode3 (/* ... */) ; // Pas de code
10 }
```

TypeOuVoid peut être :

- soit un type (primitif, tableau ou référence) si la fonction retourne un résultat, soit *void* si aucun résultat n'est retrourné ;
- *TypeThrowable* doit être une classe dérivée de la classe *Throwable* ; ils servent à indiquer quelles exceptions sont susceptibles d'être lancées et non interceptées dans le code de la méthode.

4.3.2 Variantes

Comme pour les champs, la portée et la nature de la méthode peuvent être indiqués avant le type retourné ou le *void*. Le symbol indiquant la portée ou la nature d'une méthode peut être un ou plusieurs des mots-clés suivants :

- *public*, *protected* ou *private* : une méthode *public* est accessible partout où est accessible sa classe ; une méthode *protected* est accessible par les autres classes du même *package* que *Classe1*, et par les classes dérivées de *Classe1* ; une méthode *private* n'est accessible qu'à l'intérieur du corps de *Classe1*.
- *static* : si une méthode est *static* alors, c'est une méthode de classe qui ne peut utiliser que les champs et les méthodes de *Classe1*, qui sont aussi déclarés *static* ;
- *final* : une méthode *final* ne peut pas être surchargée par les classes dérivant de sa classe.
- *abstract* : une méthode *abstract* permet de déclarer une méthode d'instance sans en donner le code de mise en œuvre, et ne peut apparaître qu'au sein d'une classe *abstract* ; toute classe non *abstract* dérivée de cette classe, doit obligatoirement mettre œuvre cette méthode, en la surchargeant.
- *native* : une méthode native est implémentée dans une bibliothèque annexe propre à la plateforme de développement, en C ou en C++ par exemple.
- *synchronized* : une méthode *synchronized* permet d'obtenir un verrou sur l'objet sur lequel elle est appelée (ou sur la classe si la méthode est aussi *static*) ; ainsi, la méthode ne peut être appelée que par un seul à la fois.

4.3.3 Passage des arguments

Java ne met en œuvre que le passage des arguments par copie. Une méthode reçoit la valeur de chacun des paramètres qui lui sont passés, et ces paramètres se comportent comme des variables locales :

- si un paramètre est d'un type base, la modification de sa valeur n'a de portée qu'à l'intérieur du corps de la méthode ;
- si un paramètre est une référence sur un objet, alors vous pouvez modifier le contenu de l'objet référencé en lui envoyant des messages ou en modifiant les champs public ;
- pour protéger un objet passé en argument à une méthode, vous pouvez utiliser la méthode *clone* pour créer une copie d'un objet et passer la copie en argument.

```
1  public class TestReference {
2     static void modifieStringPointe(StringBuffer s) {
3        s.append("&" + s.toString());
4     }
5     static void essaiDeModifierReference (String s) {
6        s = new String(s + "&" + s);
7     }
8     public static void main (String argv[]) {
9        StringBuffer sb = new StringBuffer("Coucou");
10       modifieStringPointe(sb);
```

```
11        System.out.println(sb); // "Coucou&Coucou"
12        String s = new String("Coucou");
13        essaiDeModifierReference(s);
14        System.out.println(s); // "Coucou"
15    }
16 }
```

Ce code montre :

- 2 : la méthode *modifieStringPointe* montre qu'il est possible de modifie l'état d'un objet dont la référence est passée en argument, en lui envoyant des messages ; c'est le cas de l'argument *s*, qui référence un *StringBuffer* et qui reçoit le message *append* dans le corps de la méthode ;
- 5 : la méthode *essaiDeModifierReference* affecte une nouvelle référence à *s* ;
- 9-14 : dans la méthode *main*, on montre que le *StringBuffer* passé en argument à *modifieString* est bien modifié par l'exécution de la méthode *modifieStringPointe* ; on montre aussi que la référence de type String passée en argument à *essaiDeModifierReference* n'est pas modifié par l'exécution de la méthode.

4.3.4 Tableau de références

Pour un tableau de références, chaque élément du tableau doit être initialisé en créant un objet et en le stockant dans une case du tableau. Par exemple :

```
1  public class TestTableau {
2     public static void main (String argv[]) {
3        String [] tab; // Declaration du tableau
4        tab = new String[2]; // Creation du tableau
5        tab[0] = new String("AA"); // Affectation d'une instance de String dans la èpremire case
6        tab[1] = new String("BB");// Affectation d'une instance de String dans la seconde case
7     }
8  }
```

4.3.5 Surcharge des méthodes

Une méthode est surchargée quand elle est déclarée plusieurs fois dans une même classe ou ses classes dérivées, avec le même nom mais des paramètres de types différents :

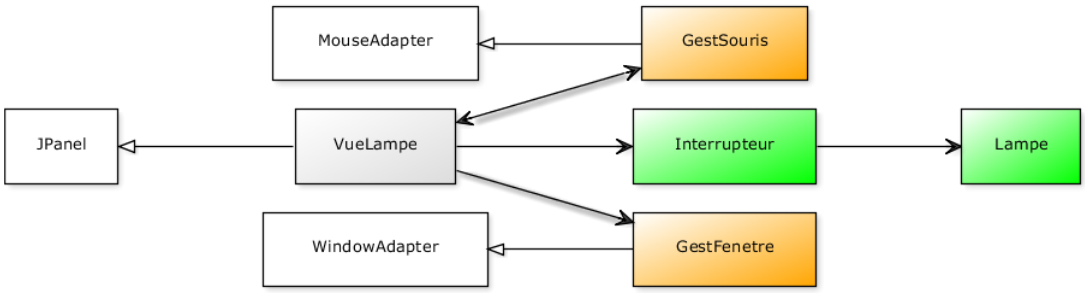
```
1  class Classe1 {
2     void methodeSurchargee (int entier) {
3     }
4     void methodeSurchargee (float nombre) {
5     }
6  }
7
8  class Classe2 extends Classe1 {
9     void methodeSurchargee (float nombre, short param) {
10    }
11 }
```

Il est interdit de créer deux méthodes avec les mêmes paramètres et un type de valeur de retour différent (par exemple, int methode () et float methode ()).

4.4 Un exemple : animation d'un modèle de lampe

Nous pouvons déjà illustrer les éléments syntaxiques présentés dans ce chapitre par une petite animation 2D. Nous montrons ici un exemple d'animation d'une lampe : on allume ou on éteint une lampe en cliquant sur un interrupteur. La lampe est représentée par un cercle de couleur dans une fenêtre. L'interrupteur est représenté par un rectangle. Quand on clique Le clic de la souris sera exploité pour simuler le changement d'état de l'interrupteur (ouvert ou fermé) et en conséquence, celui de la lampe (allumée ou éteinte).

Cet exemple comprend donc tout d'abord les classes *Lampe* et *Interrupteur*. Ces classes décrivent notre modèle. Nous devons aussi mettre en œuvre une classe pour la vue et un classe pour la gestion de la souris dans la vue.



4.4.1 Le modèle

Voici les classes *Lampe* et *Interrupteur* constituant notre modèle.

4.4.1.1 La classe *Lampe*

Une lampe peut être allumée ou éteinte. Notre modèle de lampe gère l'état *lampe allumée* ou *éteinte*. Pour conserver cet état, un champ *boolean* est utilisé.

```
1  public class Lampe {
2    boolean allumee = false;
3
4    public void allumer() {
5      allumee = true;
6    }
7
8    public void eteindre() {
9      allumee = false;
10   }
11
12   public boolean estAllumee() {
13     return allumee;
14   }
15
16 }
```

Ce code montre :

- 1 à 23 : la déclaration de la classe *Lampe* ;
- 2 : la déclaration du champ booléen pour l'état de la lampe : allumée ou éteinte ;
- 4 à 10 : deux méthodes pour changer l'état de la lampe ;
- 12 : une méthode pour questionner une lampe sur son état ;

4.4.1.2 La classe *Interrupteur*

Un interrupteur peut être ouvert ou fermé : lorsqu'il est fermé, le courant passe. Cet état est géré à l'aide d'un champ booléen. Un interrupteur est relié à une lampe. Lorsque l'interrupteur change d'état, alors la lampe doit aussi changer d'état : si l'interrupteur est ouvert, alors la lampe est éteinte, sinon, la lampe est allumée. Un champ référence sur un objet de type Lampe est ajouté à l'interrupteur. Ce champ permet à l'interrupteur de directement accéder à la lampe pour gérer son état.

```
1  public class Interrupteur {
2    boolean ferme = false;
3    Lampe lampe = null;
4
5    private void miseAJourLampe() {
6      if (lampe != null) {
7        if (this.estFerme()) {
8          lampe.allumer();
9        } else {
10         lampe.eteindre();
11       }
```

```
12        }
13    }
14
15    public void connecterAvec(Lampe l) {
16        lampe = l;
17        miseAJourLampe();
18    }
19
20    public void fermer() {
21        ferme = true;
22        miseAJourLampe();
23    }
24
25    public void ouvrir() {
26        ferme = false;
27        miseAJourLampe();
28    }
29
30    public boolean estFerme() {
31        return ferme;
32    }
33
34    public void ouvrirOuFermer() {
35        if (this.estFerme()) {
36            this.ouvrir();
37        } else {
38            this.fermer();
39        }
40    }
41
42 }
```

Ce code montre :

- 1 à 42 : la déclaration de la classe *Interrupteur* ;
- 2 : la déclaration du champ booléen pour l'état fermé ou ouvert ;
- 3 : la déclaration du champ référence pour la lampe connectée à l'interrupteur ;
- 5 : la méthode *miseAJourLampe* permet de gérer l'état de la lampe en fonctiuon de celui de l'interrupteur :
  - 6 : on s'assure bien que la lampe est reliée car sinon, il n'est pas possible de lui envoyer de message ;
  - 7 à 11 : si l'interrupteur est fermé, alors le message *allumer* est envoyé à la lampe, sinon, le message *eteindre* est envoyé à la lampe ;

  on remarque que cette méthode est déclarée pour usage interne uniquement (mot clé *private* devant le type de retour de la fonction) ;

- 15 : la méthode qui permet d'associer une lampe à un interrupteur ; dès qu'une lampe est associée, alors son état est positionné en cohérence avec celui de l'interrupteur pas l'envoi du message *miseAJourLampe* ;
- 20 à 28 : deux méthodes pour changer l'état ouvert ou fermé de l'interrupteur ; on remarque que chacune d'elle modifie l'état de la lampe par l'envoi du message *miseAJourLampe* ;
- 30 : un accesseur pour l'état de l'interrupteur ;
- 34 : une méthode pour inverser l'état de l'interrupteur et par conséquence ce de la lampe.

4.4.2 Animation de notre *modèle*

Jusqu'à présent, on ne s'est pas du tout préoccupé de la façon dont sont représentés l'interrupteur et la lampe et non plus de la façon dont l'utilisateur interagit avec le modèle. Ces rôles sont ceux de la vue et du contrôleur. Voici les classes permettant d'animer notre modèle avec une représentation en 2D. Visuellement, cette vue est simplement constituée d'une fenêtre avec un cercle pour la lampe et un rectangle pour l'interrupteur. Le cercle qui représente la lampe est jaune quand la lampe est allumée, il est gris lorsque la lampe est éteinte. Le rectangle qui représente l'interrupteur est noir. Les chaines "OUVERT" ou "FERME" sont affichées dans le rectangle de l'interrupteur suivant l'état de ce dernier. Du point de vue contrôle, l'utilisateur peut inverser l'interrupteur en cliquant dessus avec la souris.

4.4.2.1 La programmation de la vue

Voici le code complet de la classe *VueLampe* :

```
1  public class VueLampe extends JPanel {
2     Lampe lampe;
3     Interrupteur interrupteur;
4     Rectangle intRect = new Rectangle(0, 0, 60, 25);
5
6     private Color getCouleurLampe() {
7        if (lampe.estAllumee())
8           return Color.YELLOW;
9        else
10          return Color.GRAY;
11    }
12
13    private String getInterrupteurLibelle() {
14       if (interrupteur.estFerme())
15          return "FERME";
16       else
17          return "OUVERT";
18    }
19
20    public void paint(Graphics g) {
```

```
21        super.paint(g);
22        Color c = g.getColor();
23        g.setColor(this.getCouleurLampe());
24        g.fillOval(50, 25, 100, 100);
25        g.setColor(Color.BLACK);
26        g.fillRect((int) intRect.getX(), (int) intRect.getY(),
27              (int) intRect.getWidth(), (int) intRect.getHeight());
28        g.setColor(Color.WHITE);
29        g.drawString(this.getInterrupteurLibelle(), (int) intRect.getX(),
30              (int) intRect.getY() + 15);
31        g.setColor(c);
32     }
33
34     public void ouvrirFenetre() {
35        JFrame window = new JFrame();
36        window.getContentPane().add(this);
37        window.pack();
38        window.setBounds(new Rectangle(0, 0, 200, 200));
39        window.setVisible(true);
40        lampe = new Lampe();
41        interrupteur = new Interrupteur();
42        interrupteur.connecterAvec(lampe);
43        GestSouris gs = new GestSouris();
44        gs.gereSourisPour(this);
45        this.addMouseListener(gs);
46        window.addWindowListener(new GestFenetre());
47     }
48
49     public void boutonGaucheDeSouris(Point p) {
50        if (intRect.contains(p)) {
51           interrupteur.ouvrirOuFermer();
52           this.repaint();
53        }
54     }
55
56     public static void main(String[] args) throws InterruptedException {
57        VueLampe vl = new VueLampe();
58        vl.ouvrirFenetre();
59     }
60
61  }
```

Voyons chaque partie de cette classe séparément. Tout d'abord les éléments en interaction, la vue comprend une référence sur une lampe et sur l'interrupteur.

```
1  public class VueLampe extends JPanel {
2     private static final long serialVersionUID = 1L;
3     Lampe lampe;
```

```
4   Interrupteur interrupteur;
5   Rectangle intRect = new Rectangle(0, 0, 60, 25);
6   ...
```

Ce code montre :

- 1 : la vue est programmée à l'aide d'un *JPanel*. Cette classe déjà existante nous permet de dessiner dans le panneau d'une fenêtre ; la classe *VueLampe* est déclarée simplement comme une sous-classe de *JPanel* par l'utilisation du mot clé *extends* ;
- 2 : déclaration d'un identifiant pour notre panneau, il n'est pas utilisé dans notre mise en œuvre, mais est proposé par *eclipse* (notre environnement de développement) pour éviter un message d'avertissement lors de la compilation ;
- 3 et 4 : la déclaration des références vers la lampe et vers l'interrupteur ; leur valeur d'initialisation par défaut est *null* ;
- 5 : le rectangle qui représente les bornes de notre interrupteur, nous en aurons besoin pour la mlise en place de son contrôle par la souris ;

Les deux méthodes *getCouleurLampe* et *getInterrupteurLibelle* sont définies pour calculer la couleur de la lampe et le libelle à afficher pour l'interrupteur. Ces deux méthodes interrogent respectivement la lampe et l'interrupteur pour calculer l'expression à retourner. Elles sont à usage privé par les autres méthodes de la classe *VueLampe*, elles sont donc déclarées *private*.

```
1   private Color getCouleurLampe() {
2     if (lampe.estAllumee())
3       return Color.YELLOW;
4     else
5       return Color.GRAY;
6   }
7
8   private String getInterrupteurLibelle() {
9     if (interrupteur.estFerme())
10      return "FERME";
11    else
12      return "OUVERT";
13  }
```

La méthode *paint* est définie par défaut dans la classe *JPanel* pour dessiner le fond du panneau. Ici, nous redéfinissons cette méthode pour en plus dessiner notre lampe et notre interrupteur.

```
1   public void paint(Graphics g) {
2     super.paint(g);
3     Color c = g.getColor();
4     g.setColor(this.getCouleurLampe());
5     g.fillOval(50, 25, 100, 100);
6     g.setColor(Color.BLACK);
7     g.fillRect((int) intRect.getX(), (int) intRect.getY(),
8           (int) intRect.getWidth(), (int) intRect.getHeight());
```

```
9       g.setColor(Color.WHITE);
10      g.drawString(this.getInterrupteurLibelle(), (int) intRect.getX(),
11              (int) intRect.getY() + 15);
12      g.setColor(c);
13  }
```

Ce code montre :

- 1 : la signature de la fonction qui doit être exactement la même que celle de la super-classe (*JPanel*) ; cette méthode s'appelle donc forcement *paint*, elle prend donc forcément une référence d'un objet de type *Graphics* en argument, elle ne retourne aucun résultat et est déclarée publique ;
- 2 : le code de la méthode *paint* tel qu'il est déclaré dans la super-classe doit d'abord être exécuté ; c'est ce que nous déclarons par l'instruction super.paint(g); ; nous verrons ce point plus précisément plus tard ;
- 3 : la couleur du panneau est mise de coté dans une variable locale pour qu'elle puisse être restituée à la fin de l'exécution de *paint* dans *VueLampe* ;
- 4-5 : le dessin du cercle pour la lampe ; la couleur de la lampe est récupérée en résultat de l'envoi du message *getCouleurLampe* ;
- 6-10 : le dessin du rectangle noir pour l'interrupteur ; le libelle à afficher est récupéré en résultat de l'envoi du message *getInterrupteurLibelle* ;
- 12 : la couleur originale est replacée en utilisant celle référencée par la variable locale *c* récupérée en début de fonction *paint*.

Le message *paint* n'est pas envoyé par vous même. La méthode *paint* que vous avez programmée est exécutée lorsque le panneau est dans une fenêtre et que la fenêtre est ouverte. Voici la méthode *ouvrirFenetre* que vous pouvez écrire pour créer et ouvrir une fenêtre :

```
1   public void ouvrirFenetre() {
2       JFrame window = new JFrame();
3       window.getContentPane().add(this);
4       window.pack();
5       window.setBounds(new Rectangle(0, 0, 200, 200));
6       window.setVisible(true);
7       lampe = new Lampe();
8       interrupteur = new Interrupteur();
9       interrupteur.connecterAvec(lampe);
10      GestSouris gs = new GestSouris();
11      gs.gereSourisPour(this);
12      this.addMouseListener(gs);
13      window.addWindowListener(new GestFenetre());
14  }
```

Ce code montre :

- 2-6 : création et ouverture d'une fenêtre (instance de la classe *JFrame*) ;
- 7-9 : création du modèle avec la lampe et l'interrupteur, et association de la lampe et de l'interrupteur ;

– 10-13 : partie concernant le contrôle : création et association des différents gestionnaires d'évènements, le plus important ici est celui de la souris (*GestSouris*) que nous verrons dans la suite.

La méthode *boutonGaucheDeSouris* appelée lorsqu'un clic avec le bouton gauche de la souris est détecté dans la fenêtre.

```
1  public void boutonGaucheDeSouris(Point p) {
2    if (intRect.contains(p)) {
3      interrupteur.ouvrirOuFermer();
4      this.repaint();
5    }
6  }
```

Ce code montre :

– 2 : contrôle de la position du clic : il doit se situer dans le rectangle représenté par l'interrupteur ;

– 3-4 : lorsque le clic est dans le rectangle de l'interrupteur, son état est inversé par l'envoi du message *ouvrirOuFermer* à l'interrupteur et la vue est mise à jour par l'envoi du message *repaint* au panneau ; la méthode *repaint* n'est pas définie par vous même, elle est déjà mise en œuvre par la classe *JPanel*.

La fonction *main* crée une instance de VueLampe et lui envoie le message *ouvrirFenêtre* :

```
1  public static void main(String[] args) throws InterruptedException {
2    VueLampe vl = new VueLampe();
3    vl.ouvrirFenetre();
4  }
```

4.4.2.2 La programmation du contrôle

Le contrôle est nécessaire à deux niveaux. Tout d'abord pour réagir à la fermeture de la fenêtre et ensuite pour réagir au clic de souris dans la fenêtre.

Ces deux contrôles sont mis en œuvre par les deux classes *GestFenetre* et *GestSouris* :

```
1  class GestFenetre extends WindowAdapter {
2    public void windowClosing(WindowEvent e) {
3      System.exit(0);
4    }
5  }
6
7  class GestSouris extends MouseAdapter {
8    private VueLampe vl;
9
10   public void gereSourisPour(VueLampe vue) {
11     this.vl = vue;
12   }
13
14   public void mouseClicked(MouseEvent evt) {
```

```
15          if (SwingUtilities.isLeftMouseButton(evt)) {
16              vl.boutonGaucheDeSouris(evt.getPoint());
17          }
18      }
19  }
```

Le contrôle consiste à redéfinir des méthodes prédéfinies pour la réponse aux évènements. Les classes qui le mettent en oeuvre héritent de *WindowAdapter* et de *MouseAdapter* qui mettent fournissent un contrôle par défaut qui ne fait rien. Le développeur redéfinit les méthodes utiles pour son application, par exemple, *windowClosing* pour la fermeture de la fenêtre et *mouseClicked* pour la gestion du clic de souris :

- 3 : la fermeture de la fenêtre provoque la sortie de l'application ;
- 16 : le clic de souris provoque l'envoi du message *boutonGaucheDeSouris* à la vue ;

4.5 Résumé de ce que nous avons vu

- nous avons vu les différentes façons de déclarer une classe avec ou sans héritage notamment ;
- nous avons vu comme déclarer et initialiser des champs (variables d'instance ou attributs) dans nos classes ;
- nous avons vu comment déclarer des méthodes, nous avons discuté des arguments de méthode et de la notion de surcharge ;
- nous avons illustré ce que nous avons appris à l'aide d'un exemple d'animation graphique d'une lampe reliée à un interrupteur et allumée ou éteinte à l'aide de la souris.

Chapitre 5

Plus loin avec les classes

Ce chapitre précise comment Java permet de déclarer des classes, de créer des objets et d'exploiter l'héritage.

5.1 Notion de constructeur

Si vous voulez initialiser certains champs avec une valeur donnée à la création d'un nouvel objet, il vous faut déclarer une méthode spéciale appelée un constructeur.

5.1.1 Déclaration d'un constructeur

Un constructeur est une méthode qui a le même nom que la classe et pour laquelle aucun retour n'est précisé. Les instructions du corps d'un constructeur sont généralement destinées à initialiser les champs de l'objet nouvellement créé. Plusieurs constructeurs peuvent être définis dans une même classe avec des paramètres formels différents : on surcharge le constructeur.

```
class Classe1 {
    public Classe1 () {
    }
    Classe1 (TypeP1 param1Name /* ... */) {
    }
    Classe1 (/* ... */) throws TypeThrowable /*, TypeThrowable2 */ {
    }
}
```

La définition d'un constructeur n'est pas obligatoire il est même conseillé d'en limiter le nombre pour faciliter la compréhension et la simplicité du code. Dans notre exemple de la lampe, nous n'avons défini aucun constructeur. Cependant, un constructeur peut être intéressant pour simplifier l'écriture des programmes, simplifier la gestion de son état, rendre les classes plus réutilisables et limiter le nombre de fonctions publiques (mot clé *public* utilisé pour qualifier une fonction).

Reprenons notre exemple de la lampe. :

- pour la classe *VueLampe*, les modèle de lampe et d'interrupteur sont créés par la vue (fonction *ouvrirFenêtre*) ; cela qui fige la conception ; le développeur ne peut pas spécialiser la classe *Lampe* et utiliser la même classe de vue ;
- pour la classe *GestSouris*, on est obligé de spécifier la fonction publique *gereSourisPour* de plus, la vue dont *GestSouris* a obligatoirement besoin peut ne pas être associée ;
- pour la classe *Interrupteur* ; une instance de cette classe devrait être obligatoirement associée à une instance de Lampe or ce n'est pas le cas et on est obligé de gérer la possibilité qu'il n'y ait pas de lampe pour un interrupteur ; de plus, la fonction *connecterAvecLampe* a du être spécifiée.

Voici la nouvelle version avec l'utilisation des constructeurs, le code ajouté, simplifié et qui ne change pas est explicitement indiqué par des commentaires :

```
public class VueLampe extends JPanel {
  private static final long serialVersionUID = 1L;
  Lampe lampe;
  Interrupteur interrupteur;
  Rectangle intRect = new Rectangle(0, 0, 60, 25);

  // Ajout du constructeur
  public VueLampe (Interrupteur interrupteur) {
    this.interrupteur = interrupteur;
    lampe = this.interrupteur.lampe;
  }
```

```
12
13      private Color getCouleurLampe() { /* idem */ }
14      private String getInterrupteurLibelle() { /* idem */ }
15      public void paint(Graphics g) { /* idem */ }
16
17      public void ouvrirFenetre() {
18          JFrame window = new JFrame();
19          window.getContentPane().add(this);
20          window.pack();
21          window.setBounds(new Rectangle(0, 0, 200, 200));
22          window.setVisible(true);
23          // Simplification àpartir d'ici
24          new GestSouris(this);
25          new GestFenetre(window);
26      }
27
28      public void boutonGaucheDeSouris(Point p) { /* idem */ }
29
30      public static void main(String[] args) throws InterruptedException {
31          // Passage d'un interrupteur en argument au constructeur
32          VueLampe vl = new VueLampe(new Interrupteur(new Lampe()));
33          vl.ouvrirFenetre();
34      }
35
36  }
37
38  class Lampe { /* idem */ }
39
40  class Interrupteur {
41      boolean ferme = false;
42      Lampe lampe = null;
43
44      // Ajout du constructeur
45      public Interrupteur(Lampe lampe) {
46          this.lampe = lampe;
47          miseAJourLampe();
48      }
49
50      private void miseAJourLampe() { // Simplification ici
51          if (this.estFerme()) {
52              lampe.allumer();
53          } else {
54              lampe.eteindre();
55          }
56      }
57
58      public void fermer() { /* idem */ }
```

```
59    public void ouvrir() { /* idem */ }
60    public boolean estFerme() { /* idem */ }
61    public void ouvrirOuFermer() { /* idem */ }
62 }
63
64 class GestFenetre extends WindowAdapter {
65    // Ajout du constructeur
66    public GestFenetre(JFrame window) {
67       window.addWindowListener(this);
68    }
69
70    public void windowClosing(WindowEvent e) { /* idem */ }
71 }
72
73 class GestSouris extends MouseAdapter {
74    private VueLampe vl;
75    // Ajout du constructeur
76    public GestSouris(VueLampe vue) {
77       vl = vue;
78       vl.addMouseListener(this);
79    }
80
81    public void mouseClicked(MouseEvent evt) { /* idem */ }
82 }
```

5.1.2 A propos des arguments

5.1.2.1 Nom des arguments

Très souvent, les arguments d'un constructeur sont utilisés pour donner une valeur initiale à des variables d'instance. Dans ce cas, il est très lisible d'utiliser le même nom pour un argument et pour la variable d'instance initialisée. Mais attention, dans le corps d'une fonction, un nom désigne prioritairement un argument :

```
1 class Classe1 {
2    int var;
3    Classe1 (int var) {
4       this.var = var;
5    }
6 }
```

Dan le code du constructeur *var2* tout seul désigne l'argument. Pour désigner la variable d'instance, il faut utiliser le préfixe *this* comme dans l'affectation de la ligne 4. On peut aussi utiliser le préfixe *super* pour désigner éventuellement une variable de la super classe.

Cependant, cette pratique est à déconseiller : chaque classe doit avoir la responsabilité de gérer ses propres variables d'instance et non les variables d'instance héritées.

5.1.2.2 Nature des arguments

Certaines variables d'instance ne doivent pas être initialisées directement par un argument. Il est ainsi souvent problématique de définir systématiquement des constructeurs avec une liste d'arguments correspondant à toutes les variables d'instance.

Prenons l'exemple d'une université. Une université gère un ensemble d'étudiants. Une classe *Universite* peut être déclarée de la façon suivante :

```
class Universite {
   String nom = null;
   ArrayList<Etudiant> etudiants = null;
   ...
}
```

Une université a un nom et gère une liste d'étudiants. Le réflexe fréquent consiste à créer un constructeur avec un argument pour le nom mais aussi un argument pour la liste d'étudiants. Cela donne le code suivant :

```
public Universite(String nom, ArrayList<Etudiant> etudiants) {
   this.nom = nom;
   this.etudiants = etudiants;
}
```

C'est souvent une erreur : a t'on besoin de passer la liste d'étudiants ? En fait, ce constructeur est très souvent utilisé de la façon suivante :

```
   ArrayList<Etudiant> etudiants = new ArrayList<Etudiant>();
   etudiants.add(new Etudiant("Dupond", "Pierre", 12345));
   etudiants.add(new Etudiant("Dupond", "Pierre", 12345)); ...
   Universite u = new Universite("UBO", etudiants)) ;
```

Il est souvent préférable de laisser gérer sa liste d'étudiant et de masquer la façon dont sont stockés les étudiants dans l'instance d'*Universite*. De plus, si la collection *etudiants* est modifiés directement, cela risque d'altérer l'état de l'instance d'*Universite* (l'encapsulation est violée). Il est donc préférable de ne pas passer la liste d'étudiants en argument mais plutôt d'utiliser une méthode *ajouterEtudiant* de la façon suivante :

```
   Universite u = new Universite("UBO")) ;
   u.addEtudiant(new Etudiant("Dupond", "Pierre", 12345));
   u.addEtudiant(new Etudiant("Dupond", "Pierre", 12345)); ...
```

Le code du constructeur pour *Universite* serait le suivant :

```
public Universite(String nom) {
   this.nom = nom;
   this.etudiants = new ArrayList<Etudiant>();
}
```

Le code de *addEtudiant* serait le suivant :

```
public void addEtudiant(Etudiant etudiant) {
   etudiants. add(etudiant);
```

```
3     // avec en plus eventuellement d'autres actions comme par exemple
4     // l'emission d'un evenement, la modification de l'instance d'etudiant ajoute ...
5  }
```

5.1.3 Utilisation des mots clé *this* et *super* pour exécuter un constructeur

Le corps d'un constructeur peut éventuellement commencer par une des deux instructions suivantes :

```
1  this (argument1 /*, argument2, ...*/);
2  super (argument1 /*, argument2, ...*/);
```

Ces deux instruction sont en fait des appels explicites à un constructeur particulier. Il permettent donc d'invoquer un autre constructeur de la même classe ou d'une super-classe en lui passant des valeurs nécessaires à l'initialisation de l'objet.

Java impose que tout constructeur appelle directement ou indirectement un constructeur de sa super-classe : si *super* ou *this* n'est pas utilisé explicitement en première instruction d'un constructeur, alors, c'est *super* sans argument qui est implicitement exécutée. Dans ce cas, il faut que la super-classe ait un constructeur sans argument.

```
1  class Classe1 {
2     int var;
3     Classe1 (int var) {
4        this.var = var;
5     }
6  }
7
8  class Classe2 extends Classe1 {
9     int var2;
10    Classe2 (int var2) {
11       super (3); // Appel du constructeur de Classe1 avec la valeur 3
12       this.var2 = var2;
13    }
14    Classe2 () {
15       this (2); // Appel du premier constructeur avec la valeur 2
16    }
17 }
```

Ce code montre :

- 1 : la classe *Classe1* qui déclare un constructeur avec un argument ; cette classe n'a donc pas de constructeur sans argument implicite puisqu'un constructeur avec argument est spécifié ; on ne peut pas écrire new Classe1() ;
- 8 : la classe *Classe2* qui hérite de *Classe1* ; *Classe2* doit obligatoirement définir un constructeur puisque sa super-classe n'a pas de constructeur sans argument ;
- 11 : exécution explicite du constructeur avec argument de la super-classe *Classe1* ;
- 14 : constructeur sans argument de *Classe2* ;

– 15 : exécution explicite du constructeur avec argument de la classe *Classe2* ; ce code permet d'avoir une valeur par défaut pour le champ *var2* de la classe *Classe2* et pour le champ *var* de la classe *Classe1*.

Voici un autre exemple avec une erreur fréquente :

```
1  class Personne {
2     protected String nom ;
3     protected String prenom ;
4     public Personne (String nom, String prenom) {
5        this.nom = nom ;
6        this.prenom = prenom ;
7     }
8  }
9  class Etudiant extends Personne {
10    protected int num ;
11    public Etudiant (String nom, String prenom, int num) {
12       this.nom = nom ;
13       this.prenom = prenom ;
14       this.num = num ;
15    }
16 }
```

Si vous compilez ce code, voici le message d'erreur obtenu sur la sortie standard :

```
1  No constructor matching Personne() found in class Etudiant.
2        public Etudiant (String nom, String prenom, int num) {
3                         ^
```

Dans le constructeur de *Etudiant*, aucun appel explicite de constructeur n'est programmé avec *this* ou *super*. Alors, le constructeur de *Etudiant* exécute *super()* implicitement. L'erreur vient du fait que la super-classe ne comporte pas de constructeur sans argument. Pour corriger ce code, on a deux solutions, mais une bonne et une mauvaise :

– mauvaise solution : définir un constructeur sans argument dans la classe *Personne* pour ne plus avoir de message d'erreur :

```
1  class Personne {
2     ...
3     public Personne () {
4        this.nom = null ;
5        this.prenom = null ;
6     }
7  }
```

C'est une mauvaise solution car :

– une instance de *Personne* sans nom et sans prénom n'a pas de sens ; il est ainsi possible de créer des instances de Personne ayant *null* pour nom et prénom (état instable possible des instances de *Personne*) ;

– on accède directement à des variable d'instance de *Personne* depuis la sous-classe *Etudiant* ;

  - on ajoute du code inutile sans trop réfléchir.

- meilleure solution : modifier le code du constructeur de *Etudiant* en exécutant explicitement le constructeur voulu de la super-classe *Personne* :

```
1 class Etudiant extends Personne {
2   protected int num;
3   public Etudiant (String nom, String prenom, int num) {
4     super(nom, prenom);
5     this.num = num;
6   }
7 }
```

  C'est une bien meilleure solution car :

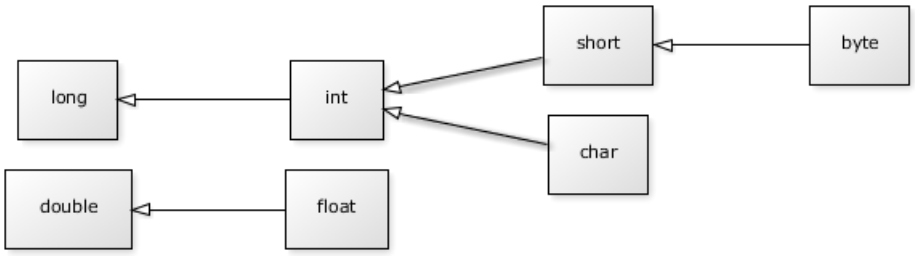
  - on ne modifie que le code de la sous-classe et on a pas de constructeur qui n'a pas de sens ;
  - on réutilise le code de la super-classe et on simplifie celui de la sous-classe ;
  - la responsabilité de modifier les variables d'instance est toujours bien localisée : les classes *Personne* et *Etudiant* gèrent leurs propres champs ;
  - on ajoute pas de code inutile.

5.2 Compatibilité de types

Pour affecter une valeur ou la référence d'un objet à une variable ou dans un élément de tableau, pour le passage d'un argument à une fonction ou pour le retour de fonction, des contrôles de types sont effectués par le compilateur et dynamiquement par la machine virtuelle. Le contrôle effectué consiste à vérifier que la valeur affectée, passée en argument ou retourné par une fonction est compatible avec le type déclaré pour la variable, le paramètre formel ou le retour de fonction.

5.2.1 Compatibilité des types primitifs

Pour les types primitifs, une compatibilité par défaut est assurée pour les types entiers et les types flottants. Le schéma suivant montre cette compatibilité par défaut : la flèche peut être lue comme "peut être affecté dans".



Ainsi, une valeur de type int peut être affectée dans une variable de type *int* ou de type *long*. Une valeur de type *byte* peut être affectée dans une variable de type *byte*, *short*, *int* ou

long, ... La compatibilité est donc toujours du type codé sur le moins grand nombre de bits vers le type codé sur le plus grand nombre de bits.

Cependant, on peut toujours par exemple, affecter un entier dans une variable de type float et l'inverse. Si les types ne sont pas compatibles par défaut, alors, il est possible de forcer le compilateur à accepter l'affectation en indiquant explicitement le type destination entre parenthèses. On parle de transtypage (on emploie aussi souvent le terme anglais cast) :

```
1    int i1 = 12;
2    int i2 = (int) 1.0;
3    float i3 = (float) 0;
4    float i4 = 1.0f;
```

- 1 : on affecte l'entier 12 à la variable entière i1 ; les types sont identiques donc l'affectation est directement possible ;
- 2 : le flottant 1.0 est affecté à i2, une variable entière ; cette affectation n'est pas autorisée car les types sont incompatibles ; le type est forcé ce qui a pour conséquence la conversion du flottant en entiier ;
- 3 : même cas que pour la ligne 2 ; l'entier 0 est converti en flottant pour être affecté à i3 ;
- 4 : on peut directement affecter 1.0f puisque le f après le 0 indique explicitement qu'il s'agit d'un type float.

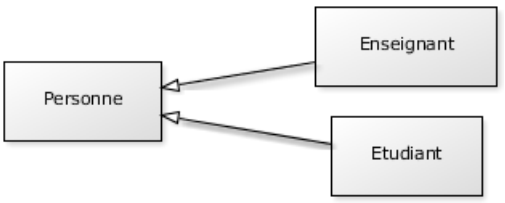
Tous les types primitifs peuvent être transtypés entre eux sauf le type *boolean*.

5.2.2 Compatibilité des types référence

Une classe spécifie un type. Nous avons vu que le type d'une variable référence peut être le nom d'une classe. La compatibilité se définit alors par rapport à l'héritage. Une référence vers un sous-type est compatible avec une référence sur un super-type. Par contre, pour le cas contraire, il est obligatoire d'utiliser le transtypage. Le transtypage peut être valide entre objets d'une même hiérarchie de classes.

La vérification de la validité d'un transtypage est effectuée par le compilateur mais aussi dynamiquement, par la machine virtuelle. En effet, lors de l'exécution d'une affectation, du passage d'un argument ou lors du retour d'une fonction, la machine virtuelle peut lancer une exception *ClassCastException* si les types sont incompatibles.

Les exemples donnés dans la suite sont étudiés avec la hiérarchie de classes suivante : les classes *Etudiant* et *Enseignant* héritent de la même super-classe *Personne*.




```
1 class Personne { ... }
2 class Etudiant extends Personne { ... }
3 class Enseignant extends Personne { ... }
```

5.2.2.1 Erreurs détectées à la compilation

Voici des exemples d'affectations dont certaines donnerons des erreurs de compilation :

```
1  class TranstypageEvalCompilation {
2    public static void main(String args[]) {
3      Personne personne = new Personne();
4      Etudiant etudiant = new Etudiant();
5      Enseignant enseignant = new Enseignant();
6
7      Etudiant etudiant2 = new Enseignant(); // ** Erreur **
8      Etudiant etudiant3 = new Personne(); // ** Erreur **
9      Enseignant enseignant2 = new Personne(); // ** Erreur **
10
11     Personne personne2 = etudiant;
12     Personne personne3 = enseignant;
13
14     Etudiant etudiant4 = (Etudiant) enseignant; // ** Erreur **
15     Enseignant enseignant3 = (Enseignant) etudiant; // ** Erreur **
16   }
17 }
```

- 3-5 : pour ces trois premières affectations il n'y a aucun problème puisque le type de la variable est identiques à celui de l'objet dont la référence est affectée ;
- 7 : erreur de compilation car un type *Enseignant* est incompatible avec un type *Etudiant*, ces deux types ne sont pas liés par une relation d'héritage ;
- 8 : erreur de compilation, le type *Personne* est super-classe de *Etudiant* ;
- 9 : erreur de compilation, le type *Personne* est super-classe de *Enseignant* ;
- 11-12 : pas de problème, le type de *etudiant* et celui de *enseignant* sont des sous-classes de *Personne* ;
- 14-15 : erreur de compilation, les types sont incompatibles et l'erreur est détectée malgré le transtypage.

5.2.2.2 Erreurs détectées à l'exécution

Voici des exemples d'incompatibilité qui ne seront pas détectées lors de la compilation mais qui produirons des erreurs lors de l'exécution (lancement de l'exception *ClassCastException*) :

```
1  class TranstypageEvalDynamique {
2    public static void main(String args[]) {
3      Personne personne = new Personne();
4      Etudiant etudiant = new Etudiant();
5      Enseignant enseignant = new Enseignant();
6
7      Etudiant etudiant2 = (Etudiant) personne; // ** Erreur **
8      Enseignant enseignant2 = (Enseignant) personne; // ** Erreur **
```

```
9
10      Object objet1 = enseignant;
11      Object objet2 = (Etudiant) objet1; // ** Erreur **
12    }
13  }
```

- 7 : *personne* est de type *Personne* qui est incompatible avec *Etudiant* mais le transtypage force le compilateur à accepter l'affectation ; l'erreur est par contre détectée à l'exécution ;
- 8 : idem que pour 7 ;
- 11 : l'objet référencé par *objet1* est une instance d'*Enseignant*, le transtypage en *Etudiant* provoque une erreur à l'exécution ;

5.3 Redéfinition d'une méthode

Dans un sous-classe, on peut redéfinir une méthode de la super-classe. La signature doit alors identique à la définition dans le super-classe. On peut utiliser la redéfinition pour étendre le comportement de la super classe dans la sous-classe. Pour cela, on peut exécuter le code de la méthode dans le super classe à l'aide du mot clé *super*. Contrairement au constructeur, l'utilisation de *super* n'est pas contraint à être en première instruction.

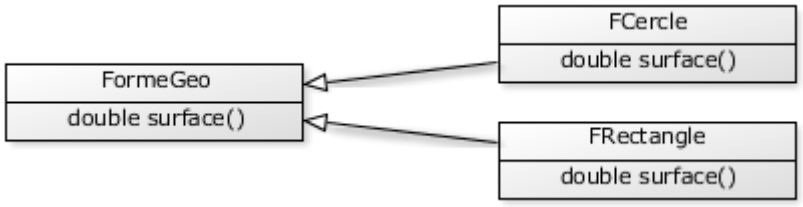
```
1  class Personne {
2    protected String nom;
3    protected String prenom;
4
5    public Personne (String nom, String prenom) {
6      this.nom = nom;
7      this.prenom = prenom;
8    }
9
10   String toString() {
11     return "Personne (" + nom + ", " + prenom +")";
12   }
13
14   void printOn(StringBuffer buf) {
15     buf.append("Nom : " + nom);
16     buf.append("Prenom : " + prenom);
17   }
18 }
19
20 class Etudiant extends Personne {
21   protected int num;
22
23   public Etudiant (String nom, String prenom, int num) {
24     super(nom, prenom);
25     this.num = num;
```

```
26    }
27
28    String toString() {
29       return "Etudiant (" + nom + ", " + prenom +", " + num + ")";
30    }
31
32    void printOn(StringBuffer buf) {
33       super.printOn(buf);
34       buf.append("Num : " + num);
35    }
36 }
```

5.4 Lien dynamique

En Java, c'est toujours le type réel de l'objet référencé qui compte pour le choix de la méthode à exécuter et pour les contrôles de type effectués à l'exécution. Le lien est donc toujours dynamique.

Dans la suite, nous illustrons le lien dynamique sur un exemple de calcul de surface pour des formes géométriques : *FRectangle* et *FCercle* sont des sous-classes de *FormeGeo*. Chacune de ces classes met en œuvre la méthode *surface*.




```
1  class FormeGeo { ...
2
3     double surface () {
4        return 0.0d;
5     }
6  }
7
8  class FRectangle extends FormeGeo {
9     int largeur;
10    int longueur; ...
11
12    double surface () {
13       return (double) largeur * longeur;
14    }
15 }
16
17 class FCercle extends FormeGeo {
18    int rayon; ...
```

```
19
20    double surface () {
21        Math.PI * Math.pow((double)rayon, 2);
22    }
23 }
```

Le message *surface* envoyé à un objet provoque l'exécution de la méthode de l'objet réellement référencé : soit celle de la classe *FRectangle*, soit celle de la classe *FCercle*. Ce n'est donc pas le type déclaré de la référence qui compte mais bien le type de l'objet référencé. La méthode réellement exécutée n'est donc connue qu'à l'exécution c'est pour cette raison qu'on parle de lien dynamique. Dans notre exemple, la version de *FormeGeo* (le type de la référence *f*) n'est jamais exécutée.

```
1 FormeGeom f = new FRectangle(4,6);
2 f.surface();
3 f = new FCercle(3.5);
4 f.surface();
```

- 1 : le type de *f* est *FormeGeo* ; une instance de *FRectangle* lui est affecté ;
- 2 : la méthode *surface* exécutée est celle de la classe *FRectangle* car l'objet référencé à l'exécution est un *FRectangle* ;
- 3 : la référence d'une instance de *FCercle* est maintenant affectée à *f* ;
- 4 : la méthode *surface* exécutée est celle de la classe *FCercle* ;

5.5 Méthode et classe abstraite

Nous avons vu qu'une méthode ou qu'une classe peut être qualifiée comme abstraite à l'aide du mot clé *abstract*. La notion d'abstraction est très importante, elle permet de définir des concepts ou du comportement générique sans en préciser le code. C'est un point clé pour la construction de composants génériques et réutilisables.

5.5.1 Méthode abstraite

Une méthode peut être déclarée abstraite. En conséquence, la méthode n'a pas de code, elle ne spécifie donc aucun comportement et doit être obligatoirement redéfinie dans une sous-classe.

L'intérêt est de définir un élément d'un interface commune à toutes les sous-classes d'une super-classe. En d'autres termes, on déclare un élément de contrat qui garantie qu'un comportement est disponible.

Pratiquement, une méthode abstraite est la signature d'une méthode. L'exemple suivant, illustre la notion de méthode abstraite avec la classe *Morph* qui représente le concept d'objet qui peut être dessiné en 2D. Les spécialisations pourront être des classes pour représenter une ligne, un rectangle, un cercle, ... C'est un exemple typique de polymorphisme.

Nous avons vu qu'on peut dessiner à l'aide des primitives mise en œuvre par la classe *Graphics*. Par contre, on ne sait pas a priori comment un *Morph* peut être dessiné. L'algorithme de dessin est en effet dépendant de la nature du *Morph* : si c'est un cercle, un rectangle, une ligne, ...

Nous voulons pouvoir programmer des algorithmes génériques qui utilisent des instances de *Morph* sans faire d'hypothèse sur le type concret des instances de *Morph*. Nous voulons développer des algorithmes qui seront correctes quelque-soit les instances de *Morph* utilisées. Par exemple, si nous voulons modéliser un espace 2D qui peut être dessiné. Un tel espace comprend des objets *Morph*. On peut déclarer le classe *Espace2D* qui hérite de *JPanel* et qui met en œuvre sa méthode *paint* de la façon suivante :

```
1  public class Espace2D extends JPanel {
2    private List<Morph> submorphs = new ArrayList<Morph>();
3      public void paint(Graphics g) {
4            super.paint(g);
5            for (Iterator<Morph> itor = submorphs.iterator(); itor.hasNext();) {
6                  itor.next().draw(g);
7            }
8      }
9    ...
10 }
```

Ce code montre la déclaration d'une classe *Espace2D* :

- 2 : la collection *submorphs* destinée à contenir les instances de Morph contenues dans un espace 2D ;
- 3 : la méthode générique paint qui dessine l'espace 2D et tous les objets qu'il contient ; cette méthode fait l'hypothèse que les instances de *Morph* comprennent le message *draw* de sorte que la méthode *paint* soit correcte quelque-soit le contenu de l'espace 2D ;

La méthode *draw* doit donc être déclarée dans la classe *Morph* sans qu'on puisse indiquer précisément le code qu'elle exécute précisément. Nous allons donc déclarer cette méthode comme abstraite dans la classe *Morph* :

```
1  public abstract class Morph {
2    protected Rectangle bounds;
3    protected Color color;
4    public abstract void draw(Graphics g)  ;
5    ...
6  }
```

En conséquence, toute sous-classe concrète de *Morph* doit redéfinir la méthode *draw* avec le code qu'elle exécute comme par exemple, la classe *Oval* :

```
1  public class Oval extends Morph{
2    public void draw(Graphics g) {
3      Color c = g.getColor();
4      g.setColor(color);
5      g.fillOval(bounds.x,bounds.y,bounds.height,bounds.width);
6      g.setColor(c);
7    }
8    ...
9  }
```

5.5.2 Classe abstraite

Une classe abstraite est spécifiée avec le mot clé *abstract*.

```
1  public abstract class Morph {
2     protected Rectangle bounds;
3     protected Color color;
4     public abstract void draw(Graphics g) ;
5     public void setColor(Color c) { color = c; }
6     ...
7  }
```

Ce code montre :

- la définition de la classe abstraite *Morph* ;
- la classe Morph comprend les champs *bounds* et *color* ;
- elle déclare la méthode abstraite *draw* ;
- elle déclare aussi la méthode *setColor* qui elle n'est pas abstraite.

Toute sous classe d'une classe abstraite destinée à être instanciée doit obligatoirement définir le comportement pour toutes les méthodes déclarées abstraites de toutes ces super-classes.

Voici l'exemple de la sous-classe concrète *Oval* qui hérite de la classe abstraite *Morph* :

```
1  public class Oval extends Morph {
2     public void draw(Graphics g) {
3        Color c = g.getColor();
4        g.setColor(color);
5        g.fillOval(bounds.x,bounds.y,bounds.height,bounds.width);
6        g.setColor(c);
7     }
8     ...
9  }
```

Ce code montre :

- la méthode *draw* redéfinie de façon à expliciter comment une instance de *Oval* est dessinée.

Un espace 2D contient une liste d'objet qui fournissent le service déclaré par la classe abstraite *Morph* :

```
1  public class Espace2D extends JPanel {
2     private List<Morph> submorphs = new ArrayList<Morph>();
3        public void paint(Graphics g) {
4              super.paint(g);
5              for (Iterator<Morph> itor = submorphs.iterator(); itor.hasNext();) {
6                    itor.next().draw(g);
7              }
8        }
9     ...
10 }
```

– la méthode *draw* redéfinie de façon à expliciter concrètement comment une instance de *Oval* est dessinée.

5.6 Les interfaces

5.6.1 Définition et justification

Java n'autorise que l'héritage simple. Par contre, il dispose de la notion d'interface : qui permet de spécifier des comportement abstraits ; et ainsi de structurer les comportements.

On dit qu'une classe met en œuvre une ou plusieurs interfaces ; Le fait qu'une classe puisse mettre en œuvre plusieurs interfaces nous rapproche quand même de l'héritage multiple.

Un exemple d'interface :

```
1 interface StepIO { // Lecture/Ecriture au format STEP
2       boolean stepRead(InputStream in) ;
3       boolean stepWrite(OutputStream out) ;
4 }
```

5.6.2 Interface versus classe abstraite

Une interface se rapproche beaucoup d'une classe abstraite particulière pour laquelle toutes les méthodes sont abstraites. Une super-classe définit une interface commune pour toutes ses sous-classes actuelles ou futures. On se sert ainsi des super-classes pour déclarer un ensemble de services (on parle aussi d'interface) commun à toutes les instances des sous-classes. La super-classe abstraite permet de déclarer cet ensemble de services à haut niveau d'abstraction (sans indiquer le code des services).

Une interface définit aussi une interface mais une interface ne peut pas contenir de code et une classe peut mettre en œuvre plusieurs interfaces.

On ne peut pas instancier une interface. Une classe peut mettre en œuvre plusieurs interfaces et s'engage donc à mettre en œuvre toutes les méthodes abstraites de ces interfaces. Une interface peut hériter d'une autre interface ; on a donc deux hiérarchies parallèles, une pour les classes et l'autre pour les interfaces. Pratiquement, une interface peut aussi contenir des champs constants static final.

On peut déclarer (typer) une variable avec le nom d'une interface et lui affecter toute référence d'objet dont la classe met en œuvre l'interface. On peut aussi utiliser l'opérateur *instanceof* pour tester si la classe d'un objet déclare mettre en œuvre une interface.

Une interface, tout comme une classe abstraite peut être utilisée pour typer des champs ou des paramètres. Le champs peut alors référencer une instance de n'importe quelle classe concrète qui hérite de la classe abstraite ou une référence d'une instance de n'importe quelle classe concrète qui hérite de la classe abstraite peut être passé en argument ou enfin une références d'une classe qui met en œuvre l'interface.

```
1 class Rectangle extends FormeGeo implements StepIO { ...
2       boolean stepRead(InputStream in) {
3     // code de la fonction pour un Rectangle
4       }
```

```
5       boolean stepWrite(OutputStream out) {
6     // code de la fonction pour un Rectangle
7       }
8   }
9
10  class Etudiant implements StepIO { ...
11      boolean stepRead(InputStream in) { }
12      boolean stepWrite(OutputStream out) { }
13  }
```

On peut manipuler des objets STEP par le biais de l'interface *StepIO* : toutes les classes des objets STEP mettent en œuvre l'interface *StepIO* ; ces classes peuvent être de nature complètement différentes.

```
1  StepIO desObjetsSTEP[] = new StepIO[100];
2
3  desObjetsSTEP[0] = new Rectangle(...);
4  desObjetsSTEP[1] = new Carre(...);
5  desObjetsSTEP[2] = new Etudiant(...);
6
7  for (int i = 0; i < desObjetsSTEP.length(); i++) {
8     if (desObjetsSTEP[i] != null) {
9        desObjetsSTEP[i].stepWrite(outStream);
10    }
11 }
```

5.7 Destruction des objets

La machine virtuelle Java gère la mémoire occupée par les objets. La mémoire nécessaire à la mémorisation de tout nouvel objet est allouée dynamiquement à sa création, et la mémoire qu'il occupe est automatiquement libérée quand celui-ci n'est plus référencé par aucune variable du programme. Cette libération est réalisée grâce au ramasse miette.

La classe *Object* fournit la méthode *finalize*() qui est invoquée juste avant que le ramasse miette ne récupère la mémoire occupée par l'objet.

La fonction *finalize* peut être redéfinie pour permettre la libération de ressources qui ne seraient pas automatiquement libérées par le ramasse miette (contextes graphiques, ressources ou mémoire alloués par des méthodes native écrites en C ou C++). On peut explicitement indiquer à la machine virtuelle qu'une référence n'est plus utile en lui affectant la valeur *null*.

5.8 Notion de package

Un *package* est un regroupement de classes, de types interfaces ou d'énumérations qui constitue un espace de nommage. Concrètement, un package se constitue d'une arborescence de répertoires. Les répertoires feuilles contiennent les classes.

Si un package *nom1.nom2* contient la classe *NomClasse*, alors, la classe doit se trouver dans le fichier *quelquepart/nom1/nom2/NomClasse.class*.

Par exemple, si le package est *java.util*, alors le compilateur effectue sa recherche dans le répertoire *java/util*. Le compilateur Java utilise la variable d'environnement *CLASSPATH* contenant l'ensemble des chemins prédéfinis à utiliser avec un package pour construire le chemin complet d'accès aux classes. On peut utiliser l'option *-classpath* de *javac* et de l'interpréteur java.

Le mot clé *import* est employé pour indiquer explicitement l'utilisation d'un *package* déjà définit. Le mot clé *package* suivi d'un nom et utilisé pour construire un nouveau *package*.

5.8.1 Importer un package

Pour utiliser un package, il faut utiliser la directive import :

```
1  import NomClasse; // Importer une classe sans package
2  import nomPackage.NomClasse; // Importer une classe d'un package
3  import nomPackage.*; // Importer toutes les classes d'un package
```

L'astérisque permet d'importer les classes d'un package à la demande : le compilateur recherche une classe inconnue parmis les packages suivis d'un astérisque.

La classe *NomClasse* peut correspondre soit à un fichier source *nomClasse.java*, soit à un fichier compilé *nomClasse.class*, dans lequel est définie la classe public à importer.

import est optionnel pour les cas suivants :

- les classes appartiennent au même package ;
- la clause import *java.lang.*;* est implicite à chaque compilation ;
- si on utilise le nom complet d'une classe : *java.util.Date* au lieu de *Date*.

5.8.2 Déclarer un package

On peut aussi créer d'autres bibliothèques que celles fournies par le système de base. Ceci se fait grâce à la clause *package*, en première ligne du fichier source, avant les *imports*, de la façon suivante : *package nomPackage ;*.

5.8.3 Structure d'un package

Exemple du fichier *MaClasse.java* qui appartient au package *nomPackage* :

```
1  package nomPackage;
2
3  import nomClasse; // Importer une classe sans package
4  import nomPackage.nomClasse; // Importer une classe d'un package
5  import nomPackage.*; // Importer toutes les classes d'un package
6
7  // Une seule classe ou interface declaree public (meme nom que le fichier)
8  public class MaClasse {
9     // Corps de MaClasse
10 }
11
```

```
12  class MaClasse2 {
13      // Corps de MaClasse2
14  }
15
16  interface MonInterface {
17      // Corps de MonInterface
18  }
```

Les classes d'un même package peuvent s'utiliser mutuellement sans utiliser import (import implicite entre les classes d'un même package).

5.8.4 Organisation arborescente des packages

Le nom de package doit correspondre au chemin d'accès du code compilé à la classe. Imaginons une société *ISOStep* spécialisée dans la programation et la commercialisation d'outils logiciels pour la norme *STEP*. Cette société utiliserait *ISOStep* comme base pour le nom des packages des produits Java qu'elle livre. Par exemple *ISOStep.ExpressParser* pour le produit *ExpressParser*.

Le code compilé des classes de ce package seront enregistrées dans le sous-répertoire *ISOStep/ExpressParser*. Si dans ce sous-répertoire, on a le code compilé d'une classe public *Parser*, pour utiliser la classe *Parser*, on doit inclure la clause *import ISOStep.ExpressParser.Parser ;* et le fichier *Parser.java* doit définir la clause *package ISOStep.ExpressParser.Parser ;*

Quand une classe est placée dans un package :

- le nom du package fait partie du nom complet de la classe ;
- le nom du package doit correspondre à l'organisation arborescente des répertoires dans lesquels est stocké le code compilé des classes.

5.8.5 Gérer une arborescence de packages

La première règle est d'éditer le code source des classes, des interfaces et des énumérations dans des fichiers textes dont le nom est le nom du type et dont l'extension est *'.java'*.

```
1  // Fichier  : Lampe.java
2
3  package lampe ;
4
5  public class Lampe {
6      // Class implementation.
7  }
```

Ce fichier source doit être placé dans un répertoire dont le nom est cohérent par rapport au package de la classe : *<Racine>/lampe/Lampe.java*.

Souvent, les sociétés qui vendent des packages utilisent leut nom de domaine inversé pour nommer leurs packages. Par exemple, pour un domaine (fictif) *java.formation.com*, tous les packages sont nommés par *com/formation/java*. Tous les composants distribués se

trouvent alors sous se répertoire. Par exemple, notre exemple de lampe pourrait se situer sous *com/formation/java/animation2D/*.

Quand le code contenu dans un fichier est compilé, le compilateur crée un fichier résultat avec le code compilé pour chaque classe, chaque interface ou chaque type énuméré. Le nom du fichier est le nom du type classe, interface ou énuméré avec le suffix *'.class'*.

Index