POO JAVA

ENSEIGNANT :
AMEVOR Kossi A.

Table des matières

Chapitre : 1

RAPPELS SUR LES NOTIONS DE CLASSE

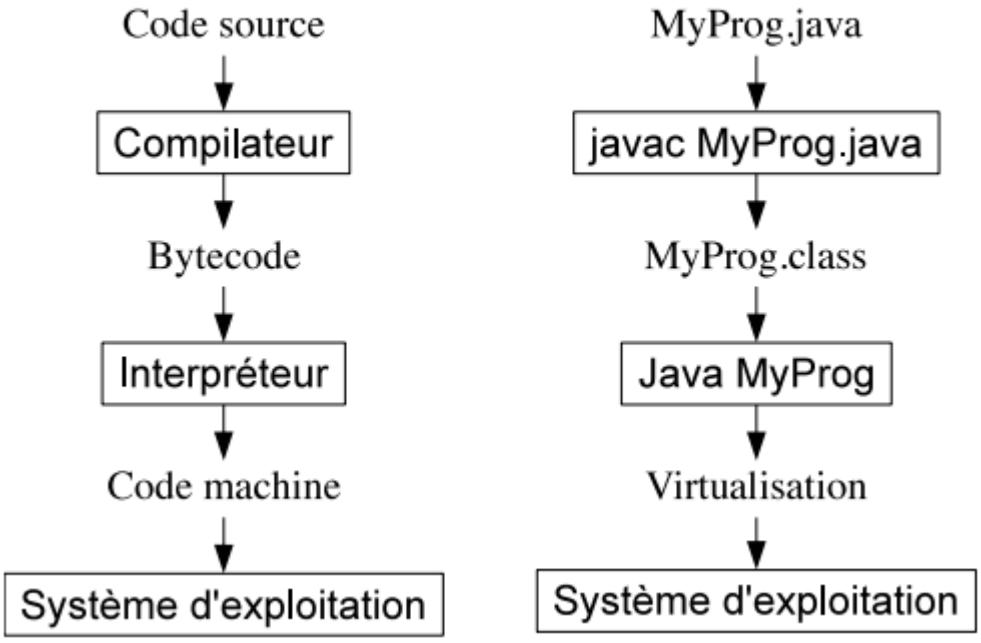
Introduction au langage Java

Le langage Java est un langage généraliste de programmation synthétisant les principaux langages existants lors de sa création en 1995 par *Sun Microsystems*. Il permet une programmation orientée-objet (à l'instar de SmallTalk et, dans une moindre mesure, C++), modulaire (langage ADA) et reprend une syntaxe très proche de celle du langage C. Outre son orientation objet, le langage Java a l'avantage d'être **modulaire** (on peut écrire desportions de code génériques, c-à-d utilisables par plusieurs applications), **rigoureux** (la plupart des erreurs se produisent à la compilation et non à l'exécution) et **portable** (un même programme compilé peut s'exécuter sur différents environnements). En contre-partie, les applications Java ont le défaut d'être plus lentes à l'exécution que des applications programmées en C par exemple.

1.1 Environnement java

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

Exemple :



Un programmeur Java écrit son code source, sous la forme de classes, dans des fichiers dont l'extension est .java. Ce code source est alors compilé par le compilateur javac en un langage appelé ***bytecode*** et enregistre le résultat dans un fichier dont l'extension est .class. Le ***bytecode*** ainsi obtenu n'est pas directement utilisable. Il doit être interprété par la machine virtuelle de Java qui transforme alors le code compilé en code machine compréhensible par le système d'exploitation. C'est la raison pour laquelle Java est un langage portable : le bytecode reste le même quelque soit l'environnement d'exécution.

En 2009, Sun Microsystems est racheté par Oracle Corporation qui fournit dorénavant les outils de développement Java SE (Standard Edition) contenus dans le Java Development Kit (JDK).

1.2 Programmation orientée-objet

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

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

1.2.1 Classe

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

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

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

```java
class Rectangle {

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

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

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

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

➢ Encapsulation

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

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

➢ Méthode constructeur

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

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

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

1.2.2 Objet

- **Instanciation**

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

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

L'usage de parenthèses à l'initialisation du vecteur, montre qu'une méthode est appelée pour l'instanciation. Cette méthode est un constructeur de la classe Cercle. Si le constructeur appelé nécessite des paramètres d'entrée, ceux-ci doivent être précisés entre ces parenthèses (comme lors d'un appel classique de méthode). L'instanciation

d'un objet de la classe Rectangle faisant appel au constructeur donné en exemple ci-dessous pourra s'écrire :

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

➢ Accès aux variables et aux méthodes

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

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

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

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

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

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

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

```java
class Carre {

    int cote;
    int origine_x;
    int origine_y;

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

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

Chapitre : 2 ELEMENTS DE PROGRAMMATION JAVA

2.1 Premiers pas

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

2.1.1 Classe HelloWorld

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

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

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

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

2.1.2 Packages

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

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

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

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

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

- une seule classe du package est importée :

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

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

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

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

```
import java.util.Date;

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

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

```
package fr.emse;

import java.util.Date;

public class DateMain {
   ...
}
```

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

2.2 Variables et méthodes

2.2.1 Visibilité des champs

Dans les exemples précédents, le mot-clé public apparaît parfois au début d'une déclaration de classe ou de méthode sans qu'il ait été expliqué jusqu'ici. Ce mot-clé

autorise n'importe quel objet à utiliser la classe ou la méthode déclarée comme publique. La portée de cette autorisation dépend de l'élément à laquelle elle s'applique.

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

Le mode public n'est, bien sûr, pas le seul type d'accès disponible en Java. Deux autres motclés peuvent être utilisés en plus du type d'accès par défaut : protected et private. Le tableau 3.2 récapitule ces différents types d'accès.

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

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

2.2 Variables et méthodes de classe

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

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

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

La classe Math fournit un ensemble d'outils (variables et méthodes) très utiles pour des programmes devant effectuer des opération mathématiques complexes. Dans la portion de classe reproduite ci-dessus, on peut notamment y trouver une approximation de la valeur de π et une méthode convertissant la mesure d'un angle d'une valeur en degrés en une valeur en radians. Dans le cas de cette classe, il est tout à fait inutile de créer et

d'instancier un objet à partir de la classe Math. En effet, la valeur de π ou la conversion de degrés en radians ne vont pas varier suivant l'objet auquel elles sont rattachées. Ce sont des variables et des méthodes de classe qui peuvent être invoquées à partir de toute autre classe (car elles sont déclarées en accès public) de la manière suivante :

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

Chapitre : 3

HERITAGE EN JAVA

Dans certaines applications, les classes utilisées ont en commun certaines variables, méthodes de traitement ou même des signatures de méthode. Avec un langage de programmation orienté- objet, on peut définir une classe à différent niveaux d'abstraction permettant ainsi de factoriser certains attributs communs à plusieurs classes. Une classe générale définit alors un ensemble d'attributs qui sont partagés par d'autres classes, dont on dira qu'elles *héritent* de cette classe générale. Par exemple, les classes Carre et Rectangle peuvent partager une méthode surface() renvoyant le résultat du calcul de la surface de la figure. Plutôt que d'écrire deux fois cette méthode, on peut définir une relation d'héritage entre les classes Carre et Rectangle. Dans ce cas, seule la classe Rectangle contient le code de la méthode surface() mais celle-ci est également utilisable sur les objets de la classe Carre si elle hérite de Rectangle.

3.1 Principe de l'héritage

L'idée principale de l'héritage est d'organiser les classes de manière hiérarchique. La relation d'héritage est unidirectionnelle et, si une classe B hérite d'une classe A, on dira que B est une sousclasse de A. Cette notion de sous-classe signifie que la classe B est un cas particulier de la classe A et donc que les objets instanciant la classe B instancient également la classe A. Prenons comme exemple des classes Carre, Rectangle et Cercle. La figure 3.1 propose une organisation hiérarchique de ces classes telle que Carre hérite de Rectangle qui hérite, ainsi que Cercle, d'une classe Forme.

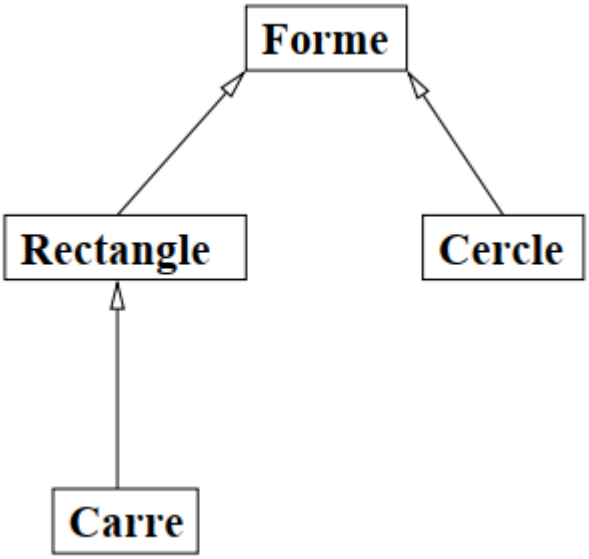


Figure 3.1 Exemple de relations d'héritage

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

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

```
public class Rectangle extends Forme {

   private int largeur;
   private int longueur;

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

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

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

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

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

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

3.1.1 Redéfinition

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

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

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

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

Remarques :

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

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

```
public class Carre extends Rectangle {

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

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

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

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

3.1.2 Polymorphisme

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

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

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

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

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

element[0] est une forme

element[0] est un rectangle

element[1] est une forme

element[1] est un cercle

element[2] est une forme

element[2] est un rectangle

element[3] est une forme

element[3] est un rectangle

element[3] est un carré

L'ensemble des classes Java, y compris celles écrites en dehors de l'API, forme une hiérarchie avec une racine unique. Cette racine est la classe Object dont hérite toute autre classe. En effet, si vous ne précisez pas explicitement une relation d'héritage lors

de l'écriture d'une classe, celle-ci hérite par défaut de la classe Object. Grâce à cette propriété, des classes génériques de création et de gestion d'un ensemble, plus élaborées que les tableaux, regroupent des objets appartenant à la classe Object (donc de n'importe quelle classe).

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

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

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

Résultat :
rectangle 10x20
cercle 15
rectangle 5x30
carré 10

Dans l'état actuel de nos classes, ce code ne pourra cependant pas être compilé. En effet, la fonction affiche() est appelée sur des objets dont la classe déclarée est Forme mais celle-ci ne contient aucune fonction appelée affiche() (elle est seulement définie

dans ses sous-classes). Pour compiler ce programme, il faut transformer la classe Forme en une interface ou une classe abstraite tel que cela est fait dans les sections suivantes.

3.2 Interfaces

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

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

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

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

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

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

Et

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

En déclarant un tableau constitué d'objets implémentant l'interface Forme, on peut appeler la méthode affiche() qui existe et est implémentée par chaque objet.

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

3.3 Classes abstraites

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

Le mot-clé abstract est utilisé devant le mot-clé class pour déclarer une classe

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

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

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

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

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

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

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

   public abstract int surface();

   public abstract void affiche();
}
```

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

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

Et

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

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

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

3.4 Classes et méthodes génériques

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

```
package java.util ;

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

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

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

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

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

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

Chapitre : 4

GESTION DES EXCEPTIONS

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

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

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

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

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

4.1 Déclaration

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

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

Cette méthode convertit une chaîne de caractères, qui doit contenir uniquement des chiffres, en un entier. Une erreur peut se produire si cette chaîne de caractères ne contient pas que des chiffres. Dans ce cas une exception de la classe NumberFormatException est émise. Une exception peut être émise dans une méthode de deux manières : (i) par une autre mé- thode appelée dans le corps de la première méthode ; (ii) par la création d'un objet instanciant la classe Exception (ou la classe Throwable) et la levée explicite de l'exception en utilisant le mot-clé throw.

L'exemple ci-dessous illustre ce second cas :

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

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

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

4.2 Interception et traitement

Avant de coder le traitement de certaines exceptions, il faut préciser l'endroit où elles vont être interceptées. Si une méthode A appelle une méthode B qui appelle une méthode C qui appelle une méthode D, et que cette méthode D lève une exception, celle-ci est d'abord transmise à C qui peut l'intercepter ou la transmettre à B, qui peut aussi l'intercepter ou la transmettre à A. L'interception d'exceptions se fait par une sorte de "mise sur écoute" d'une portion de code. Pour cela on utilise le mot-clé try suivi du bloc à surveiller. Si aucune exception ne se produit dans le bloc correspondant, le programme se déroule normalement comme si l'instruction try était absente. Par contre, si une exception est levée, le traitement de l'exception est exécuté puis l'exécution du programme reprend son cours après le bloc testé. Il est également nécessaire de préciser quelles classes d'exception doivent être interceptées dans le bloc testé. L'interception d'une classe d'exception s'écrit grâce au mot-clé catch suivi de la classe concerné, d'un nom de variable correspondant à l'objet exception, puis du traitement. Si une exception est levée sans qu'aucune interception ne soit prévue pour sa classe, celle-ci est propagée à la méthode précédente.

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

```
public class ExempleTraitementException {

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

Trois classes d'exception sont ici traitées :

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

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

4.3 Classes d'exception

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

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

Tableau 4.1 – Classes d'exception fréquentes

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

4.4 Classification des erreurs en Java

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

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

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

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

Exception vérifiée. Alors que notre programme est en cours d'exécution, une trace de la pile des exceptions est affichée, pointant vers une partie de notre code avec gestion d'exception. Visiblement, nous avons produit du code qui est capable de lever une exception vérifiée (comme FileNotFoundException) mais les données passées à notre

programme ne valide pas ces exceptions (par exemple, lorsque l'on essaie d'ouvrir un fichier qui n'existe pas). Il faut alors revoir les données passées en paramètre du programme. Notre code a bien détecté les problèmes qu'il fallait détecter. Le chapitre suivant sur les entrées/sorties présentent de nombreux exemples relatifs à ce cas.

Chapitre : 5

LE DEVELOPPEMNT D'INTERFACE GRAPHIQUE AVEC SWING

5.1 Présentation de Swing

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

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

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

5.2 Swing contient plusieurs packages :

javax.swing	package principal : il contient les interfaces, les principaux composants, les modèles par défaut
javax.swing.border	Classes représentant les bordures
javax.swing.colorchooser	Classes définissant un composant pour la sélection de couleurs
javax.swing.event	Classes et interfaces pour les événements spécifiques à Swing. Les autres événements sont ceux d'AWT (java.awt.event)
javax.swing.filechooser	Classes définissant un composant pour la sélection de fichiers
javax.swing.plaf	Classes et interfaces génériques pour gérer l'apparence
javax.swing.plaf.basic	Classes et interfaces de base pour gérer l'apparence
javax.swing.plaf.metal	Classes et interfaces pour définir l'apparence Metal qui est l'apparence par défaut
javax.swing.table	Classes définissant un composant pour la présentation de données sous forme de tableau
javax.swing.text	Classes et interfaces de bases pour les composants manipulant du texte
javax.swing.text.html	Classes permettant le support du format HTML
javax.swing.text.html.parser	Classes permettant d'analyser des données au format

	HTML
javax.swing.text.rtf	Classes permettant le support du format RTF
javax.swing.tree	Classes définissant un composant pour la présentation de données sous forme d'arbre
javax.swing.undo	Classes permettant d'implémenter les fonctions annuler/refaire

5.3 La classe JFrame

JFrame est l'équivalent de la classe Frame de l'AWT : les principales différences sont l'utilisation du double buffering qui améliore les rafraichissements et l'utilisation d'un panneau de contenu (contentPane) pour insérer des composants (ils ne sont plus insérés sans le JFrame mais dans l'objet contentPane qui lui est associé). Elle représente une fenêtre principale qui possède un titre, une taille modifiable et éventuellement un menu.

La classe possède plusieurs constructeurs :

Constructeur	Rôle
JFrame()	
JFrame(String)	Création d'une instance en précisant le titre

5.3.1 Les étiquettes

- **La classe JLabel**

Le composant JLabel propose les mêmes fonctionnalités que les intitulés AWT mais ils peuvent en plus contenir des icônes.

Cette classe possède plusieurs constructeurs :

Constructeurs	Rôle
JLabel()	Création d'une instance sans texte ni image
JLabel(Icon)	Création d'une instance en précisant l'image
JLabel(Icon, int)	Création d'une instance en précisant l'image et l'alignement horizontal
JLabel(String)	Création d'une instance en précisant le texte
JLabel(String, Icon, int)	Création d'une instance en précisant le texte, l'image et l'alignement horizontal
JLabel(String, int)	Création d'une instance en précisant le texte et l'alignement horizontal

Le composant JLabel permet d'afficher un texte et/ou une icône en précisant leur alignement. L'icône doit être au format GIF et peut être une animation dans ce format.

Exemple (code Java 1.1) :

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

public class TestJLabel1 {

  public static void main(String argv[]) {

    JFrame f = new JFrame("ma fenetre");
    f.setSize(100,200);

    JPanel pannel = new JPanel();
    JLabel jLabel1 =new JLabel("Mon texte dans JLabel");
    pannel.add(jLabel1);

    ImageIcon icone = new ImageIcon("book.gif");
    JLabel jLabel2 =new JLabel(icone);
    pannel.add(jLabel2);

    JLabel jLabel3 =new JLabel("Mon texte",icone,SwingConstants.LEFT);
    pannel.add(jLabel3);

    f.getContentPane().add(pannel);
    f.setVisible(true);
  }
}
```

La classe JLabel définit plusieurs méthodes pour modifier l'apparence du composant :

Méthodes	Rôle
setText()	Permet d'initialiser ou de modifier le texte affiché
setOpaque()	Indique si le composant est transparent (paramètre false) ou opaque (true)
setBackground()	Indique la couleur de fond du composant (setOpaque doit être à true)
setFont()	Permet de préciser la police du texte
setForeGround()	Permet de préciser la couleur du texte
setHorizontalAlignment()	Permet de modifier l'alignement horizontal du texte et de l'icône
setVerticalAlignment()	Permet de modifier l'alignement vertical du texte et de l'icône

setHorizontalTextAlignment()	Permet de modifier l'alignement horizontal du texte uniquement
setVerticalTextAlignment()	Permet de modifier l'alignement vertical du texte uniquement. Exemple : jLabel.setVerticalTextPosition(SwingConstants.TOP);
setIcon()	Permet d'assigner une icône
setDisabledIcon()	Permet de définir l'icône associée au JLabel lorsqu'il est désactivé

L'alignement vertical par défaut d'un JLabel est centré. L'alignement horizontal par défaut est soit à droite s'il ne contient que du texte, soit centré s'il contient une image avec ou sans texte. Pour modifier cet alignement, il suffit d'utiliser les méthodes ci-dessus en utilisant des constantes en paramètres : SwingConstants.LEFT, SwingConstants.CENTER, SwingConstants.RIGHT, SwingConstants.TOP, SwingConstants.BOTTOM

Par défaut, un JLabel est transparent : son fond n'est pas dessiné. Pour le dessiner, il faut utiliser la méthode setOpaque() :

Exemple (code Java 1.1) :

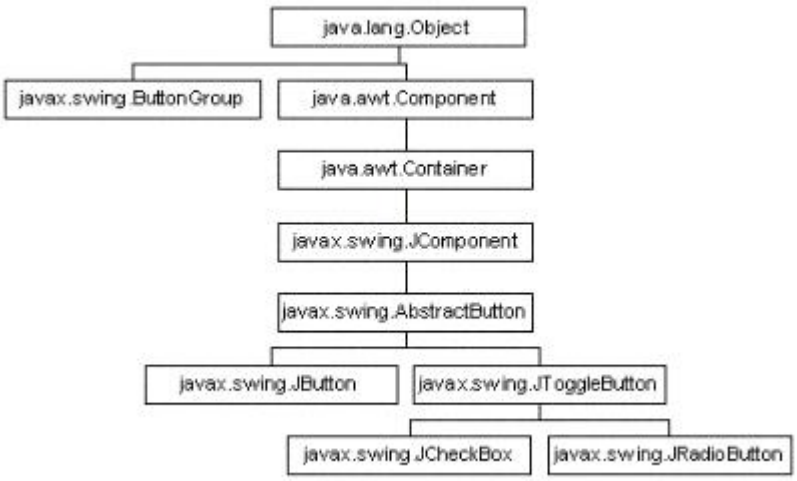
```java
import javax.swing.*;
import java.awt.*;

public class TestJLabel2 {

  public static void main(String argv[]) {

    JFrame f = new JFrame("ma fenetre");

    f.setSize(100,200);
    JPanel pannel = new JPanel();

    JLabel jLabel1 =new JLabel("Mon texte dans JLabel 1");
    jLabel1.setBackground(Color.red);
    pannel.add(jLabel1);

    JLabel jLabel2 =new JLabel("Mon texte dans JLabel 2");
    jLabel2.setBackground(Color.red);
    jLabel2.setOpaque(true);
    pannel.add(jLabel2);

    f.getContentPane().add(pannel);
    f.setVisible(true);
  }
}
```

➢ Les panneaux : la classe JPannel

La classe JPanel est un conteneur utilisé pour regrouper et organiser des composants grâce à un gestionnaire de présentation (layout manager). Le gestionnaire par défaut d'un JPanel est un objet de la classe FlowLayout.

5.3.2 Les bouton

Il existe plusieurs boutons définis par Swing.



➢ La classe AbstractButton

C'est une classe abstraite dont héritent les boutons Swing JButton, JMenuItem et JToggleButton. Cette classe définit de nombreuses méthodes dont les principales sont :

Méthode	Rôle
AddActionListener	Associer un écouteur sur un événement de type ActionEvent
AddChangeListener	Associer un écouteur sur un événement de type ChangeEvent
AddItemListener	Associer un écouteur sur un événement de type ItemEvent
doClick()	Déclencher un clic par programmation
getText()	Obtenir le texte affiché par le composant
setDisabledIcon()	Associer une icône affichée lorsque le composant a l'état désélectionné
setDisabledSelectedIcon()	Associer une icône affichée lors du passage de la souris sur le composant à l'état désélectionné
setEnabled()	Activer/désactiver le composant
setMnemonic()	Associer un raccourci clavier
setPressedIcon()	Associer une icône affichée lorsque le composant est cliqué
setRolloverIcon()	Associer une icône affichée lors du passage de la souris sur le composant

setRolloverSelectedIcon()	Associer une icône affichée lors du passage de la souris sur le composant à l'état sélectionné
setSelectedIcon()	Associer une icône affichée lorsque le composant a l'état sélectionné
setText()	Mettre à jour le texte du composant
isSelected()	Indiquer si le composant est dans l'état sélectionné
setSelected()	Définir l'état du composant (sélectionné ou non selon la valeur fournie en paramètre

Tous les boutons peuvent afficher du texte et/ou une image.

Il est possible de préciser une image différente lors du passage de la souris sur le composant et lors de l'enfoncement du bouton : dans ce cas, il faut créer trois images pour chacun des états (normal, enfoncé et survolé). L'image normale est associée au bouton grâce au constructeur, l'image enfoncée grâce à la méthode setPressedIcon() et l'image lors d'un survol grâce à la méthode setRolloverIcon(). Il suffit enfin d'appeler la méthode setRolloverEnable() avec en paramètre la valeur true.

Exemple (code Java 1.1) :

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

public class swing4 extends JFrame {

   public swing4() {
     super("titre de l'application");

     WindowListener l = new WindowAdapter() {
       public void windowClosing(WindowEvent e){
         System.exit(0);
       }
     };
     addWindowListener(l);

     ImageIcon imageNormale = new ImageIcon("arrow.gif");
     ImageIcon imagePassage =  new ImageIcon("arrowr.gif");
     ImageIcon imageEnfoncee = new ImageIcon("arrowy.gif");

     JButton bouton = new JButton("Mon bouton",imageNormale);
     bouton.setPressedIcon(imageEnfoncee);
     bouton.setRolloverIcon(imagePassage);
     bouton.setRolloverEnabled(true);
     getContentPane().add(bouton, "Center");

     JPanel panneau = new JPanel();
     panneau.add(bouton);
     setContentPane(panneau);
     setSize(200,100);
     setVisible(true);
   }

   public static void main(String [] args){
      JFrame frame = new swing4();
   }
}
```

➢ La classe Jbutton

JButton est un composant qui représente un bouton : il peut contenir un texte et/ou une icône. Les constructeurs sont :

Constructeur	Rôle
JButton()	
JButton(String)	préciser le texte du bouton
JButton(Icon)	préciser une icône
JButton(String, Icon)	préciser un texte et une icône

Il ne gère pas d'état. Toutes les indications concernant le contenu du composant JLabel sont valables pour le composant JButton.

Exemple (code Java 1.1) : un bouton avec une image

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

public class swing3 extends JFrame {

   public swing3() {

      super("titre de l'application");

      WindowListener l = new WindowAdapter() {
         public void windowClosing(WindowEvent e){
            System.exit(0);
         }
      };
      addWindowListener(l);

      ImageIcon img = new ImageIcon("tips.gif");
      JButton bouton = new JButton("Mon bouton",img);

      JPanel panneau = new JPanel();
      panneau.add(bouton);
      setContentPane(panneau);
      setSize(200,100);
      setVisible(true);
  }

  public static void main(String [] args){
      JFrame frame = new swing3();
  }

}
```

➢ La classe JToggleButton

Cette classe définit un bouton à deux états : c'est la classe mère des composants JCheckBox et JRadioButton.

La méthode setSelected() héritée de AbstractButton permet de mettre à jour l'état du bouton. La méthode isSelected() permet de connaître cet état.

➢ La classe ButtonGroup

La classe ButtonGroup permet de gérer un ensemble de boutons en garantissant qu'un seul bouton du groupe sera sélectionné.

Pour utiliser la classe ButtonGroup, il suffit d'instancier un objet et d'ajouter des boutons (objets héritant de la classe AbstractButton) grâce à la méthode add(). Il est préférable d'utiliser des objets de la classe JToggleButton ou d'une de ses classes filles car elles sont capables de gérer leurs états.

Exemple (code Java 1.1) :

```java
import javax.swing.*;

public class TestGroupButton1 {

  public static void main(String argv[]) {

    JFrame f = new JFrame("ma fenetre");
    f.setSize(300,100);
    JPanel pannel = new JPanel();

    ButtonGroup groupe = new ButtonGroup();
    JRadioButton bouton1 = new JRadioButton("Bouton 1");
    groupe.add(bouton1);
    pannel.add(bouton1);
    JRadioButton bouton2 = new JRadioButton("Bouton 2");
    groupe.add(bouton2);
    pannel.add(bouton2);
    JRadioButton bouton3 = new JRadioButton("Bouton 3");
    groupe.add(bouton3);
    pannel.add(bouton3);

    f.getContentPane().add(pannel);
    f.setVisible(true);
  }
}
```

➢ Les cases à cocher : la classe JCheckBox

Les constructeurs sont les suivants :

Constructeur	Rôle
JCheckBox(String)	précise l'intitulé
JCheckBox(String, boolean)	précise l'intitulé et l'état
JCheckBox(Icon)	spécifie l'icône utilisée
JCheckBox(Icon, boolean)	précise l'intitulé et l'état du bouton
JCheckBox(String, Icon)	précise l'intitulé et l'icône
JCheckBox(String, Icon, boolean)	précise l'intitulé, une icône et l'état

Un groupe de cases à cocher peut-être défini avec la classe ButtonGroup. Dans ce cas, un seul composant du groupe peut être sélectionné. Pour l'utiliser, il faut créer un objet

de la classe ButtonGroup et utiliser la méthode add() pour ajouter un composant au groupe.

Exemple (code Java 1.1) :

```java
import javax.swing.*;

public class TestJCheckBox1 {

  public static void main(String argv[]) {

    JFrame f = new JFrame("ma fenetre");
    f.setSize(300,100);
    JPanel pannel = new JPanel();

    JCheckBox bouton1 = new JCheckBox("Bouton 1");
    pannel.add(bouton1);
    JCheckBox bouton2 = new JCheckBox("Bouton 2");
    pannel.add(bouton2);
    JCheckBox bouton3 = new JCheckBox("Bouton 3");
    pannel.add(bouton3);

    f.getContentPane().add(pannel);
    f.setVisible(true);
  }
}
```

➢ Les boutons radio : La classe JRadioButton

Un objet de type JRadioButton représente un bouton radio d'un groupe de boutons . A un instant donné, un seul des boutons radio associés à un même groupe peut être sélectionné. La classe JRadioButton hérite de la classe AbstractButton.

Un bouton radio possède un libellé et éventuellement une icône qui peut être précisée, pour chacun des états du bouton, en utilisant les méthodes setIcon(), setSelectedIcon() et setPressedIcon().

Exemple (code Java 1.1) :

```java
import javax.swing.*;

public class TestJRadioButton1 {

  public static void main(String argv[]) {

    JFrame f = new JFrame("ma fenetre");
    f.setSize(300,100);
    JPanel pannel = new JPanel();
    JRadioButton bouton1 = new JRadioButton("Bouton 1");
    pannel.add(bouton1);
    JRadioButton bouton2 = new JRadioButton("Bouton 2");
    pannel.add(bouton2);
    JRadioButton bouton3 = new JRadioButton("Bouton 3");
    pannel.add(bouton3);

    f.getContentPane().add(pannel);
    f.setVisible(true);
  }
}
```

La méthode isSelected() permet de savoir si le bouton est sélectionné ou non.

La classe JRadioButton possède plusieurs constructeurs :

Constructeur	Rôle
JRadioButton()	Créer un bouton non sélectionné sans libellé
JRadioButton(Icon)	Créer un bouton non sélectionné sans libellé avec l'icône fournie en paramètre
JRadioButton(Icon, boolean)	Créer un bouton sans libellé avec l'icône et l'état fournis en paramètres
JRadioButton(String)	Créer un bouton non sélectionné avec le libellé fourni en paramètre
JRadioButton(String, boolean)	Créer un bouton avec le libellé et l'état fournis en paramètres
JRadioButton(String, Icon)	Créer un bouton non sélectionné avec le libellé et l'icône fournis en paramètres
JRadioButton(String, Icon, boolean)	Créer un bouton avec le libellé, l'icône et l'état fournis en paramètres

Un groupe de boutons radio est encapsulé dans un objet de type ButtonGroup.

Il faut ajouter tous les JRadioButton du groupe en utilisant la méthode add() de la classe ButtonGroup. Lors de la sélection d'un bouton, c'est l'objet de type ButtonGroup qui se charge de déselectionner le bouton précédemment sélectionné dans le groupe.

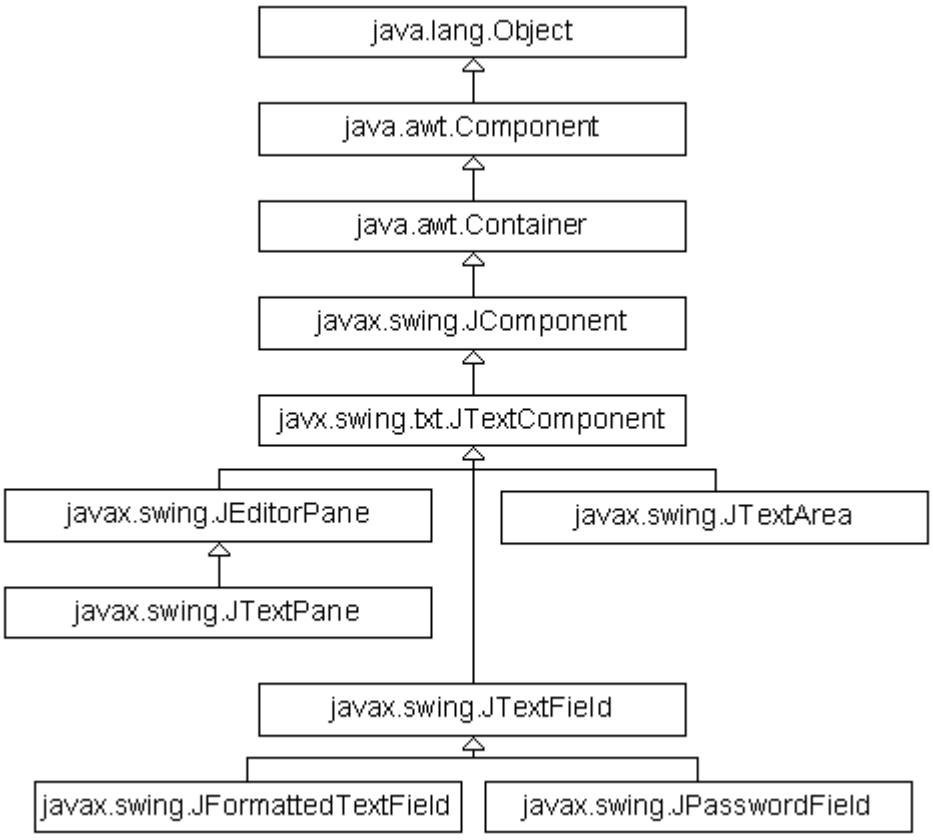
Un groupe n'a pas l'obligation d'avoir un bouton sélectionné.

Exemple :

```
import java.awt.BorderLayout;
import java.awt.Container;
import java.awt.GridLayout;
import javax.swing.BorderFactory;
import javax.swing.ButtonGroup;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JRadioButton;
import javax.swing.border.Border;

public class TestJRadioButton extends JFrame {
  public static void main(String args[]) {
    TestJRadioButton app = new TestJRadioButton();
    app.init();
  }

  public void init() {
    this.setTitle("Test radio boutons");

    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    JPanel panel = new JPanel(new GridLayout(0,1));
    Border border = BorderFactory.createTitledBorder("Sélection");
    panel.setBorder(border);
    ButtonGroup group = new ButtonGroup();
    JRadioButton radio1 = new JRadioButton("Choix 1", true);
    JRadioButton radio2 = new JRadioButton("Choix 2");
    JRadioButton radio3 = new JRadioButton("Choix 3");
    group.add(radio1);
    panel.add(radio1);
    group.add(radio2);
    panel.add(radio2);
    group.add(radio3);
    panel.add(radio3);
    Container contentPane = this.getContentPane();
    contentPane.add(panel, BorderLayout.CENTER);
    this.setSize(300, 150);
    this.setVisible(true);
  }
}
```

5.3.3 Les composants de saisie de texte

Swing possède plusieurs composants pour permettre la saisie de texte.



5.3.3.1 La classe JTextComponent

La classe abstraite JTextComponent est la classe mère de tous les composants permettant la saisie de texte.

Les données du composant (le modèle dans le motif de conception MVC) sont encapsulées dans un objet qui implémente l'interface Document. Deux classes implémentant cette interface sont fournies en standard : PlainDocument pour du texte simple et StyledDocument pour du texte riche pouvant contenir entre autres plusieurs polices de caractères, des couleurs, des images, ...

La classe JTextComponent possède de nombreuses méthodes dont les principales sont :

Méthode	Rôle
void copy()	Copier le contenu du texte et le mettre dans le presse papier système
void cut()	Couper le contenu du texte et le mettre dans le presse papier système
Document getDocument()	Renvoyer l'objet de type Document qui encapsule le texte saisi
String getSelectectedText()	Renvoyer le texte sélectionné dans le composant
int getSelectionEnd()	Renvoyer la position de la fin de la sélection
int getSelectionStart()	Renvoyer la position du début de la sélection
String getText()	Renvoyer le texte saisi

String getText(int, int)	Renvoyer une portion du texte débutant à partir de la position donnée par le premier paramètre et la longueur donnée dans le second paramètre
bool isEditable()	Renvoyer un booléen qui précise si le texte est éditable ou non
void paste()	Coller le contenu du presse papier système dans le composant
void select(int,int)	Sélectionner une portion du texte dont les positions de début et de fin sont fournies en paramètres
void setCaretPosition(int)	Déplacer le curseur dans le texte à la position précisé en paramètre
void setEditable(boolean)	Permet de préciser si les données du composant sont éditables ou non
void setSelectionEnd(int)	Modifier la position de la fin de la sélection
void setSelectionStart(int)	Modifier la position du début de la sélection
void setText(String)	Modifier le contenu du texte

Toutes ces méthodes sont donc accessibles grâce à l'héritage pour tous les composants de saisie de texte proposés par Swing.

➢ La classe JTextField

La classe javax.Swing.JTextField est un composant qui permet la saisie d'une seule ligne de texte simple. Son modèle utilise un objet de type PlainDocument.

Exemple (code Java 1.1) :

```java
import javax.swing.*;

public class JTextField1 {

  public static void main(String argv[]) {

    JFrame f = new JFrame("ma fenetre");
    f.setSize(300, 100);
    JPanel pannel = new JPanel();

    JTextField  testField1 = new JTextField ("mon texte");

    pannel.add(testField1);
    f.getContentPane().add(pannel);
    f.setVisible(true);
  }
}
```

La propriété horizontalAligment permet de préciser l'alignement du texte dans le composant en utilisant les valeurs JTextField.LEFT , JTextField.CENTER ou JTextField.RIGHT.

➢ La classe JPasswordField

La classe JPasswordField permet la saisie d'un texte dont tous les caractères saisis seront affichés sous la forme d'un caractère particulier ('*' par défaut). Cette classe hérite de la classe JTextField.

Exemple (code Java 1.1) :

```java
import java.awt.Dimension;

import javax.swing.*;

public class JPasswordField1 {

  public static void main(String argv[]) {

    JFrame f = new JFrame("ma fenetre");
    f.setSize(300, 100);
    JPanel pannel = new JPanel();

    JPasswordField  passwordField1 = new JPasswordField ("");
    passwordField1.setPreferredSize(new Dimension(100,20 ));

    pannel.add(passwordField1);
    f.getContentPane().add(pannel);
    f.setVisible(true);
  }
}
```

➢ La classe JTextArea

La classe JTextArea est un composant qui permet la saisie de texte simple en mode multiligne. Le modèle utilisé par ce composant est le PlainDocument : il ne peut donc contenir que du texte brut sans éléments multiples de formatage.

JTexteArea propose plusieurs méthodes pour ajouter du texte dans son modèle :

- soit fournir le texte en paramètre du constructeur utilisé
- soit utiliser la méthode setText() qui permet d'initialiser le texte du composant
- soit utiliser la méthode append() qui permet d'ajouter du texte à la fin de celui contenu dans le composant
- soit utiliser la méthode insert() qui permet d'insérer du texte dans le composant à une position données en caractères

La méthode replaceRange() permet de remplacer la partie de texte occupant les index donnés en paramètres par la chaîne fournie.

La propriété rows permet de définir le nombre de lignes affichées par le composant : cette propriété peut donc être modifiée lors d'un redimensionnement du composant. La propriété lineCount en lecture seule permet de savoir le nombre de lignes qui composent le texte. Il ne faut pas confondre ces deux propriétés.

Exemple (code Java 1.1) :

```java
import javax.swing.*;

public class JTextArea1 {

  public static void main(String argv[]) {

    JFrame f = new JFrame("ma fenetre");
    f.setSize(300, 100);
    JPanel pannel = new JPanel();

    JTextArea  textArea1 = new JTextArea ("mon texte");

    pannel.add(textArea1);
    f.getContentPane().add(pannel);
    f.setVisible(true);
  }
}
```

Par défaut, la taille du composant augmente au fur et à mesure de l'augmentation de la taille du texte qu'il contient. Pour éviter cet effet, il faut encapsuler le JTextArea dans un JScrollPane.

5.3.4 Les onglets

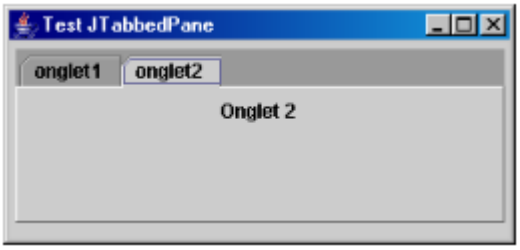
La classe javax.swing.JTabbedPane encapsule un ensemble d'onglets. Chaque onglet est constitué d'un titre, d'un composant et éventuellement d'une image.

Pour utiliser ce composant, il faut :

- instancier un objet de type JTabbedPane
- créer le composant de chaque onglet
- ajouter chaque onglet à l'objet JTabbedPane en utilisant la méthode addTab()

Exemple (code Java 1.1) :

```java
import java.awt.Dimension;
import java.awt.event.KeyEvent;

import javax.swing.*;

public class TestJTabbedPane1 {

  public static void main(String[] args) {
    JFrame f = new JFrame("Test JTabbedPane");
    f.setSize(320, 150);
    JPanel pannel = new JPanel();

    JTabbedPane onglets = new JTabbedPane(SwingConstants.TOP);

    JPanel onglet1 = new JPanel();
    JLabel titreOnglet1 = new JLabel("Onglet 1");
    onglet1.add(titreOnglet1);
    onglet1.setPreferredSize(new Dimension(300, 80));
    onglets.addTab("onglet1", onglet1);

    JPanel onglet2 = new JPanel();
    JLabel titreOnglet2 = new JLabel("Onglet 2");
    onglet2.add(titreOnglet2);
    onglets.addTab("onglet2", onglet2);

    onglets.setOpaque(true);
    pannel.add(onglets);
    f.getContentPane().add(pannel);
    f.setVisible(true);

  }
}
```



La classe JTabbedPane possède plusieurs méthodes qui permettent de définir le contenu de l'onglet :

Méthodes	Rôles
addTab(String, Component)	Permet d'ajouter un nouvel onglet dont le titre et le composant sont fournis en paramètres. Cette méthode possède plusieurs surcharges qui permettent de préciser une icône et une bulle d'aide
insertTab(String, Icon, Component, String, index)	Permet d'insérer un onglet dont la position est précisée dans le dernier paramètre

remove(int)	Permet de supprimer l'onglet dont l'index est fourni en paramètre
setTabPlacement	Permet de préciser le positionnement des onglets dans le composant JTabbedPane. Les valeurs possibles sont les constantes TOP, BOTTOM, LEFT et RIGHT définies dans la classe JTabbedPane.

La méthode getSelectedIndex() permet d'obtenir l'index de l'onglet courant. La méthode setSelectedIndex() permet de définir l'onglet courant.

5.3.5 Les menus

Les menus de Swing proposent certaines caractéristiques intéressantes en plus de celles proposées par un menu standard :

- les éléments de menu peuvent contenir une icône
- les éléments de menu peuvent être de type bouton radio ou case à cocher
- les éléments de menu peuvent avoir des raccourcis clavier (accelerators)

Les menus sont mis en oeuvre dans Swing avec un ensemble de classe :

- JMenuBar : encapsule une barre de menus
- JMenu : encapsule un menu
- JMenuItem : encapsule un élément d'un menu
- JCheckBoxMenuItem : encapsule un élément d'un menu sous la forme d'une case à cocher
- JRadioButtonMenuItem : encapsule un élément d'un menu sous la forme d'un bouton radio
- JSeparator : encapsule un élément d'un menu sous la forme d'un séparateur

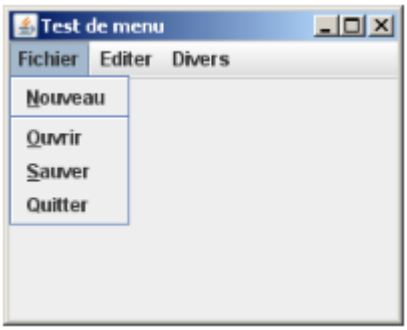
- JPopupMenu : encapsule un menu contextuel

Toutes ces classes héritent de façon directe ou indirecte de la classe JComponent.

Les éléments de menus cliquables héritent de la classe JAbstractButton.

JMenu hérite de la classe JMenuItem et non pas l'inverse car chaque JMenu contient un JMenuItem implicite qui encapsule le titre du menu.

La plupart des classes utilisées pour les menus implémentent l'interface MenuElement. Cette interface définit des méthodes pour la gestion des actions standards de l'utilisateur. Ces actions sont gérées par la classe MenuSelectionManager.



La classe JMenuBar encapsule une barre de menus qui contient zéro ou plusieurs menus. La classe JMenuBar utilise la classe DefaultSingleSelectionModel comme modèle de données : un seul de ces menus peut être activé à un instant T.

Pour ajouter des menus à la barre de menus, il faut utiliser la méthode add() de la classe JMenuBar qui attend en paramètre l'instance du menu.

Pour ajouter la barre de menus à une fenêtre, il faut utiliser la méthode setJMenuBar() d'une instance des classes JFrame, JInternalFrame, JDialog ou JApplet.

Comme la classe JMenuBar hérite de la classe JComponent, il est aussi possible d'instancier plusieurs JMenuBar et de les insérer dans un gestionnaire de positionnement comme n'importe quel composant. Ceci permet aussi de placer le menu à sa guise.

Exemple :

```
...
  public static void main(String s[]) {
    JFrame frame = new JFrame("Test de menu");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    TestMenuSwing1 menu = new TestMenuSwing1();
    frame.getContentPane().add(menu, BorderLayout.SOUTH);
    frame.setMinimumSize(new Dimension(250, 200));
    frame.pack();
    frame.setVisible(true);
  }
...
```

Les principales méthodes de la classe JMenuBar sont :

Méthodes	Rôle
JMenu add(JMenu)	Ajouter un menu à la barre de menus
JMenu getMenu(int)	Obtenir le menu dont l'index est fourni en paramètre
int getMenuCount()	Obtenir le nombre de menus de la barre de menus
MenuElement[] getSubElements()	Obtenir un tableau de tous les menus
boolean isSelected()	Retourner true si un composant du menu est sélectionné
void setMenuHelp (JMenu)	Cette méthode n'est pas implémentée et lève systématiquement une exception

➢ La classe JPopupMenu

La classe JPopupMenu encapsule un menu flottant qui n'est pas rattaché à une barre de menus mais à un composant.

La création d'un JPopMenu est similaire à la création d'un JMenu.

Il est préférable d'ajouter un élément de type JMenuItem grâce à la méthode add() de la classe JPopupMenu mais on peut aussi ajouter n'importe quel élément qui hérite de la classe Component en utilisant une surcharge de la méthode add().

Il est possible d'ajouter un élément à un index précis en utilisant la méthode insert().

La méthode addSeparator() permet d'ajouter un élément séparateur.

Pour afficher un menu flottant, il faut ajouter un listener sur l'événement déclenchant et utiliser la méthode show() de la classe JPopupMenu.

5.3.6 Affichage d'une image dans une application

Pour afficher une image dans une fenêtre, il y a plusieurs solutions.

La plus simple consiste à utiliser le composant JLabel qui est capable d'afficher du texte mais aussi une image

Exemple :

```
1  package com.jmdoudoux.test;
2
3  import java.awt.BorderLayout;
4  import javax.swing.ImageIcon;
5  import javax.swing.JFrame;
6  import javax.swing.JLabel;
7
8  public class MonApp  extends JFrame {
9
10   private static final long serialVersionUID = 1L;
11
12   public MonApp(String titre) {
13     super(titre);
14     this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
15     init();
16   }
17
18   private void init()
19   {
20     JLabel label = new JLabel(new ImageIcon("Duke.gif") );
21     this.add(label, BorderLayout.CENTER);
22     this.pack();
23   }
24
25   public static void main(String[] args) {
26     MonApp app = new MonApp("Afficher image");
27     app.setVisible(true);
28   }
29 }
```