

Langage Java OO



Pr. Chebihi Fayçal
F.chebihi@gmail.com

Qu'est ce que java?

Langage de **programmation orienté objet** (Classe, Objet, Héritage, Encapsulation et Polymorphisme)

Avec java on peut créer des application **multiplateformes**. Les applications java sont **portables**. C'est-à-dire, on peut créer une application java dans une plateforme donnée et on peut l'exécuter sur n'importe quelle autre plateforme.

Le principe de java est : **Write Once Run Every Where**

Open source: On peut récupérer le code source de java. Ce qui permet aux développeurs, en cas de besoin, de développer ou modifier des fonctionnalités de java.

Qu'est ce que java?

Java est utilisé pour créer :

Des applications Desktop

Des applets java (applications java destinées à s'exécuter dans une page web)

Des applications pour les smart phones

Des applications embarquées dans des cartes à puces

Des application JEE (Java Entreprise Edition)

Pour créer une application java, il faut installer un kit de développement java

JSDK : Java Standard Developpement Kit, pour développer les application DeskTop

JME : Java Mobile Edition, pour développer les applications pour les téléphones portables

JEE : Java Entreprise Edition, pour développer les applications qui vont s'exécuter dans un serveur d'application JEE (Web Sphere Web Logic, JBoss).

JCA : Java Card Editon, pour développer les applications qui vont s'exécuter dans des cartes à puces.

Différents modes de compilation

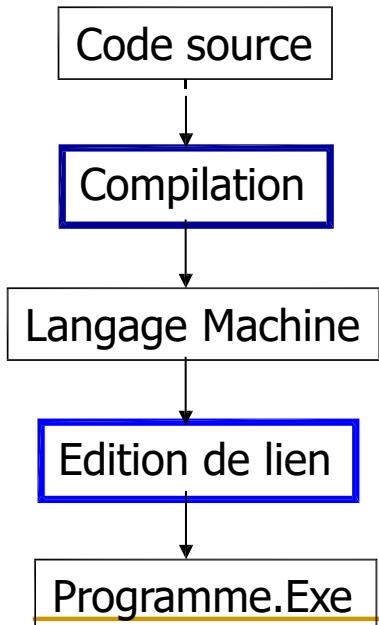
Java est un langage compilé et interprété

Compilation en mode natif

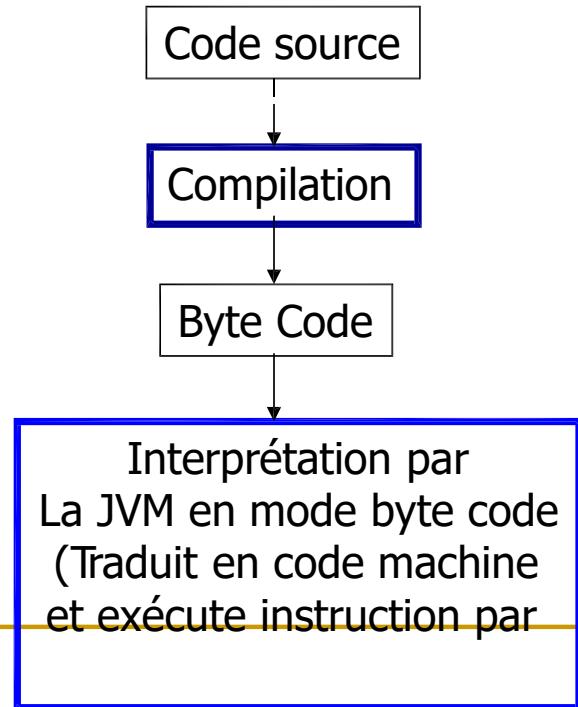
Compilation Byte Code

Compilation en mode JIT(Just In Time)

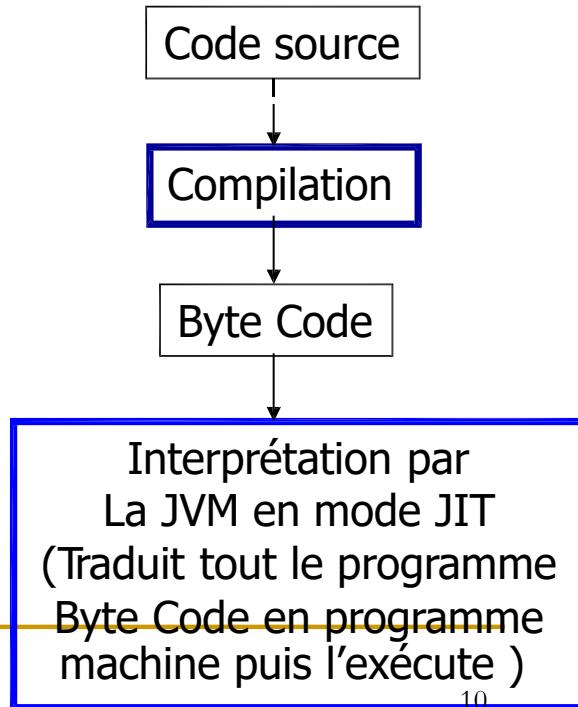
Natif



Byte Code



JIT



Installation de java

Le Kit de développement java JDK peut être téléchargé gratuitement à partir du site de Sun Microsystem son éditeur principal (www.java.sun.com).

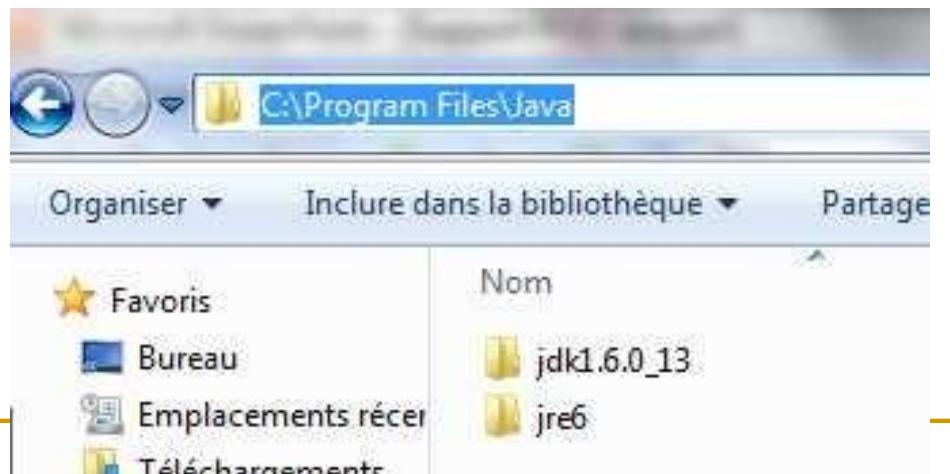
Le JDK contient 3 trois pacquages :

J2Sdk1.6.exe : Kit de développement proprement dit

Jre1.6.exe : Machine virtuelle java

jdk15-doc.zip : Documentation java

Exécuter **jdk-6u13-windows-i586-p.exe**. Le JDK sera installé dans le répertoire c:\program files\java et installe également jre1.6 dans le même dossier.



Ce que contient le JDK

C:\Program Files\Java\jdk1.6.0_13\bin			
Nom	Modifié le	Type	Taille
appletviewer.exe	18/12/2010 09:36	Application	27 Ko
apt.exe	18/12/2010 09:36	Application	27 Ko
beanreg.dll	18/12/2010 09:36	Extension de l'app...	29 Ko
extcheck.exe	18/12/2010 09:36	Application	27 Ko
HtmlConverter.exe	18/12/2010 09:36	Application	48 Ko
idlj.exe	18/12/2010 09:36	Application	27 Ko
jar.exe	18/12/2010 09:36	Application	27 Ko
jarsigner.exe	18/12/2010 09:36	Application	27 Ko
java.exe	18/12/2010 09:36	Application	136 Ko
javac.exe	18/12/2010 09:36	Application	27 Ko
javadoc.exe	18/12/2010 09:36	Application	27 Ko
javah.exe	18/12/2010 09:36	Application	27 Ko
javap.exe	18/12/2010 09:36	Application	27 Ko
java-rmi.exe	18/12/2010 09:36	Application	27 Ko
javaw.exe	18/12/2010 09:36	Application	136 Ko
javaws.exe	18/12/2010 09:36	Application	140 Ko
jconsole.exe	18/12/2010 09:36	Application	28 Ko

Kit de développement java

Les programmes nécessaires au développement java sont placés dans le répertoire c:\jdk1.5\bin à savoir:

javac.exe : Compilateur java.

java.exe : Interpréteur du bytecode java.

appletviewer.exe : Pour tester les applets java.

Jdb.exe : Débogueur java.

Javap.exe : désassembler du bytecode.

Javadoc.exe : Générer la documentation de vos programmes java.

Javah.exe : Permet de lier des programmes Java avec des méthodes natives, écrites dans un autre langage et dépendant du système.

jar.exe : Permet de compresser les classes Java ainsi que tous les fichiers nécessaires à l'exécution d'un programme (graphiques, sons, etc.). Il permet en particulier d'optimiser le chargement des applets sur Internet.

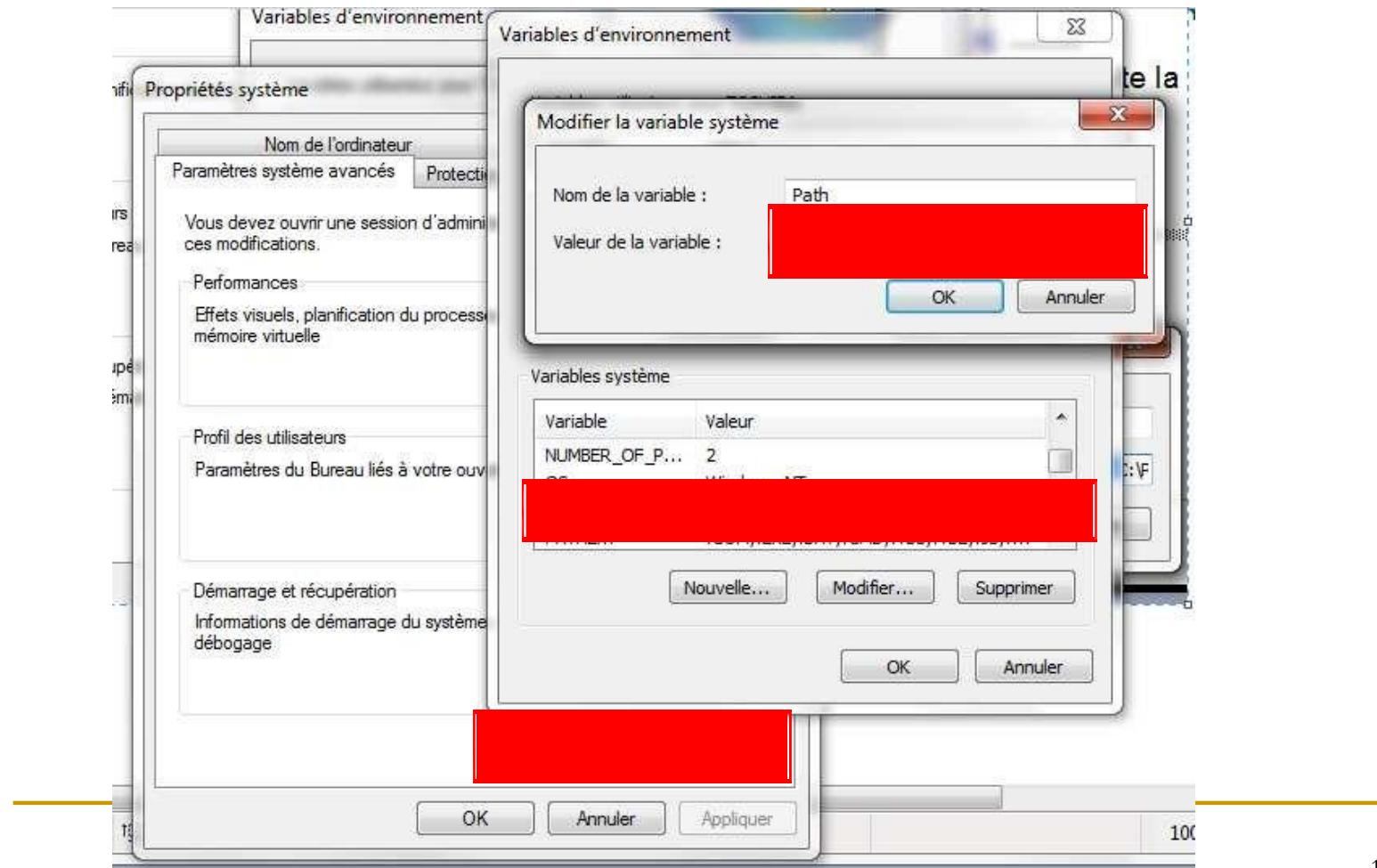
jarsigner.exe : Un utilitaire permettant de signer les fichiers archives produits par **jar.exe**.

Configuration de l'environnement

La configuration de l'environnement comporte deux aspects :

- Définir la variable d'environnement **path** qui indique le chemin d'accès aux programmes exécutables : Cette variable path devrait contenir le chemin du JDK utilisé:
 - **path= C:\Program Files\Java\jdk1.6.0_13\bin;**
- Quand elle exécute une application java, la JVM consulte la variable d'environnement **classpath** qui contient le chemin d'accès aux classes java utilisées par cette application.
 - **classpath= .; c:\monProjet\lib; c:\programmation**

Configurer la variable d'environnement path sous windows



Outils de développement java

Un Editeur de texte ASCII: on peut utiliser un simple éditeur comme notepad de windows mais il est préférable d 'utiliser un éditeur conçu pour la programmation java exemples: Ultraedit, JCreator,

Eclipse est l'environnement de développement java le plus préféré pour les développeur java. Il est gratuit et c'est un environnement ouvert.

Autres IDE java :

JDeveloper de Oracle.

JBuilder de Borland.

Premier programme java

Remarques:

Le nom du fichier java doit être le même que celui de la classe qui contient la fonction principale main.

Pour compiler le programme source, il faut faire appel au programme javac.exe qui se trouve dans le dossier c:\jdk1.2\bin.

Pour rendre accessible ce programme depuis n 'importe quel répertoire, il faut ajouter la commande : path c:\jdk1.2\bin dans le fichier autoexec.bat.

Après compilation du programme PremierProgramme.java, il y a génération du fichier PremierProgramme.class qui représente le ByteCode de programme.

Pour exécuter ce programme en byte code, il faut faire appel au programme java.exe qui représente l interpréter du bytecode.

Premier programme java

```
public class PremierProgramme {  
    public static void main(String[] args) {  
        System.out.println("First Test");  
    }  
}
```

Lancer un éditeur de texte ASCII et Ecrire le code source de ce programme.

Enregistrer ce fichier dans un nouveau répertoire c:\exojava sous le nom

PremierProgramme.java

Compiler ce programme sur ligne de commande Dos :

c:\exojava>javac PremierProgramme.java

Corriger les Erreurs de compilation

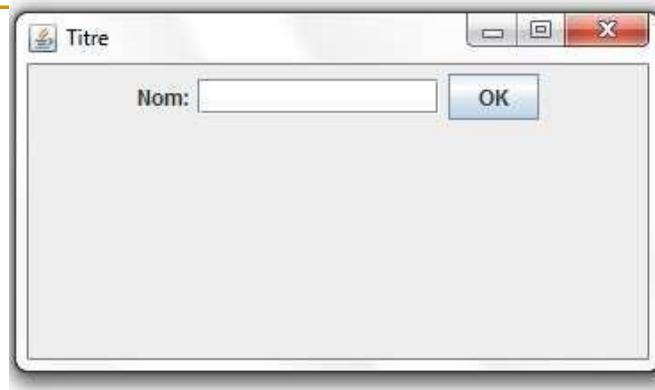
Exécuter le programme sur ligne de commande

c:\exojava>java PremierProgramme



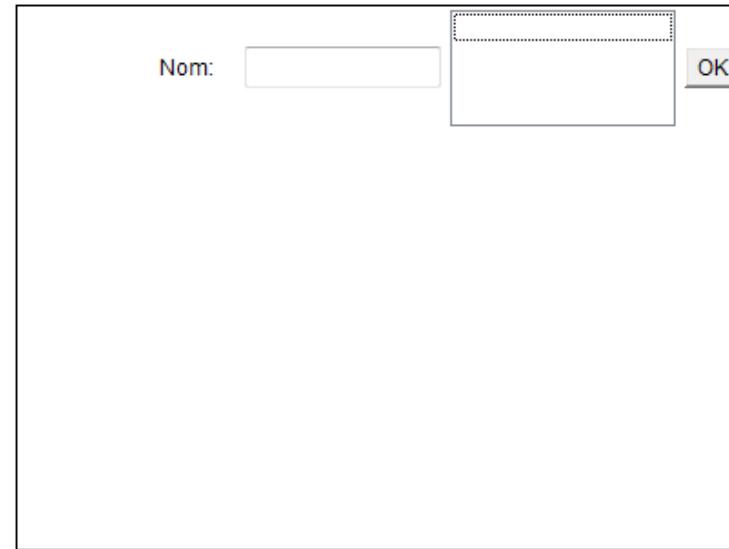
Première Application graphique

```
import javax.swing.*;
import java.awt.*;
public class FirstGraphicApp {
public static void main(String[] args) {
    // Créer une nouvelle fenêtre
    JFrame jf=new JFrame("Titre");
    //Créer les composants graphiques
    JLabel l=new JLabel("Nom:");
    JTextField t=new JTextField(12);
    JButton b=new JButton("OK");
    //Définir une technique de mise en page
    jf.setLayout(new FlowLayout());
    //Ajouter les composants à la fenêtre
    jf.add(l);jf.add(t);jf.add(b);
    //Définir les dimensions de la fenêtre
    jf.setBounds(10, 10, 400, 400);
    //Afficher la fenêtre
    jf.setVisible(true);
}
```



Première Applet

```
import java.applet.Applet;
import java.awt.*;
public class FirstApplet extends Applet{
public void init(){
    add(new Label("Nom:"));
    add(new TextField(12));
    add(new List());
    add(new Button("OK"));
}
public void paint(Graphics g) {
    g.drawRect(2, 2, 400, 300);
}
}
```



Rédiger le programme source.

Enregistrer le fichier sous le nom FirstApplet.java

Compiler le programme source et corriger les erreurs.

Créer un page HTML qui affiche l'applet sur un navigateur web:

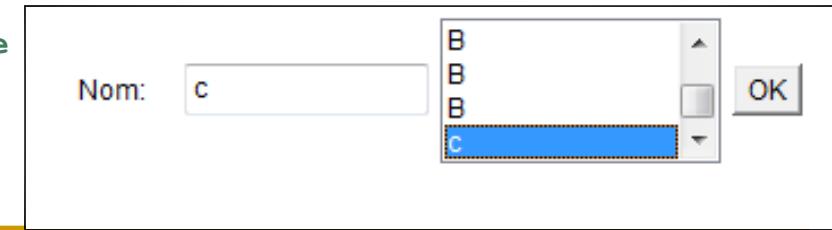
```
<html>
<body>
<applet code=<< FirstApplet.class" width="500" height="500"></applet>
</body>
</html>
```

[Vous pouvez également AppletViewer.exe pour tester l'applet :](#)

C:\exojava>appletviewer page.htm

Deuxième Applet avec Gestion des événements

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
public class DeuxiemeApplet extends Applet implements ActionListener {
    // Déclarer et créer les composants graphiques
    Label lNom=new Label("Nom:");
    TextField tNom=new TextField(12);
    List listNoms=new List();
    Button b=new Button("OK");
    // Initialisation de l'applet
    public void init() {
        // Ajouter les composants à l'applet
        add(lNom);add(tNom);add(listNoms);add(b);
        // En cliquant sur le bouton b le gestionnaire
        // des événements actionPerformed s'exécute
        b.addActionListener(this);
    }
    // Méthode qui permet de gérer les événements
    public void actionPerformed(ActionEvent e) {
        if(e.getSource()==b) {
            // Lire le contenu de la zone de texte
            String nom=tNom.getText();
            // Ajouter ce contenu dans la liste
            listNoms.add(nom);
        }
    }
}
```



Structures fondamentales du langage java

Structure du langage java

Au niveau syntaxe, Java est un langage de programmation qui ressemble beaucoup au langage C++

Toute fois quelques simplifications ont été apportées à java pour des raisons de sécurité et d'optimisation.

Dans cette partie nous ferons une présentation succincte des types primitifs, les enveloppeurs, déclarations des variables, le casting des primitives, les opérateurs arithmétiques et logiques, les structures de contrôle (if, switch, for et while)

Les primitives

Java dispose des primitives suivantes :

<u>Primitive</u>	<u>Étendue</u>	<u>Taille</u>
char	0 à 65 535	16 bits
byte	-128 à +127	8 bits
short	-32 768 à +32 767	16 bits
int	-2 147 483 648 à + 2 147 483 647	32 bits
long		
float		64 bits
double	de $\pm 1.4\text{E}-45$ à $\pm 3.40282347\text{E}38$	32 bits
boolean		64 bits
void	true ou false	1 bit
	-	0 bit

Utilisation des primitives

Les primitives sont utilisées de façon très simple. Elles doivent être déclarées, tout comme les handles d'objets, avec une syntaxe similaire, par exemple :

```
int i;  
char c;  
boolean fini;
```

Les primitives peuvent être initialisées en même temps que la déclaration.

```
int i = 12;  
char c = 'a';  
boolean fini = true;
```

Utilisation des primitives

Comment choisir le nom d'une variable:

Pour respecter la typologie de java, les nom des variables commencent toujours par un caractère en minuscule et pour indiquer un séparateur de mots, on utilise les majuscules. Exemples:

```
int nbPersonnes;  
String nomPersonne;
```

Valeurs par défaut des primitives:

Toutes les primitives de type numérique utilisées comme membres d'un objet sont initialisées à la valeur 0. Le type boolean est initialisé à la valeur **false**.

Casting des primitives

Le casting des primitives

Le *casting* (mot anglais qui signifie *moulage*), également appelé *cast* ou, parfois, *transtypage*, consiste à effectuer une conversion d'un type vers un autre type.

Le casting peut être effectué dans deux conditions différentes

Vers un type plus général. On parle alors de *sur-casting* ou de *sur-typage*.

Vers un type plus particulier. On parle alors de *sous-casting* ou de *sous-typage*.

Casting des primitives

Sur-casting : Le sur-casting peut se faire implicitement ou explicitement.

Exemples :

```
int a=6; // le type int est codé sur 32 bits
```

```
long b; // le type long est codé sur 64 bits
```

Casting implicite :

```
b=a;
```

Casting explicite

```
b=(long)a;
```

Sous-Casting : Le sous-casting ne peut se faire qu'explicitement.

```
1 : float a = (float)5.5;
```

```
2 : double c = (double)a;
```

```
4 : int d = 8;
```

```
5 : byte f = (byte)d;
```

Les enveloppeurs (wearpers)

Les primitives sont enveloppées dans des objets appelés enveloppeurs (Wearpers). Les enveloppeurs sont des classe

<u>Classe</u>	<u>Primitive</u>
Character	char
Byte	byte
Short	short
Integer	int
Long	long
Float	float
Double	double
Boolean	boolean
Void	-
BigInteger -	-
BigDecimal	-

Utilisation des primitives et enveloppeurs

Exemple:

```
double v1=5.5; // v1 est une primitive  
Double v2=new Double(5.6); // v2 est un objet  
long a=5; // a est une primitive  
Long b=new Long(5); // b est un objet  
Long c= 5L; // c est un objet  
System.out.println("a="+a);  
System.out.println("b="+b.longValue());  
System.out.println("c="+c.byteValue());  
System.out.println("v1="+v1);  
System.out.println("v2="+v2.intValue());
```

Résultat:

```
a=5  
b=5  
c=5  
V1=5.5  
V2=5
```

Opérateurs

Opérateur d 'affectation:

- **x=3;** // x reçoit 3
- **x=y=z=w+5;** // z reçoit w+5, y reçoit z et x reçoit y

Les opérateurs arithmétiques à deux opérandes:

- **+** : addition
- **-** : soustraction
- ***** : multiplication
- **/** : division
- **%** : modulo (reste de la division euclidienne)

Opérateurs

Les opérateurs arithmétiques à deux opérandes
(Les raccourcis)

`x = x + 4;` ou `x+=4;`
`z = z * y;` ou `Z*=y;`
`v = v % w;` ou `v%=w;`

Les opérateurs relationnels:

- . `==` : équivalent
- . `<` : plus petit que
- . `>` : plus grand que
- . `<=` : plus petit ou égal
- . `>=` : plus grand ou égal
- . `!=` : non équivalent

Les opérateurs d'incrémentations et de décrémentation:

`++` : Pour incrémenter (`i++` ou `++i`)
`--` : Pour décrémenter (`i--` ou `--i`)

Opérateurs

Les opérateurs logiques

&& Et (deux opérandes)

|| Ou (deux opérandes)

! Non (un seul opérande)

L'opérateur à trois opérandes ?:

condition ? expression_si_vrai : expression_si_faux

exemple : `x = (y < 5) ? 4 * y : 2 * y;`

Equivalent à :

`if (y < 5)`

`x = 4 * y;`

`else`

`x = 2 * y;`

Structures de contrôle

L'instruction conditionnelle if

La syntaxe de l'instruction **if** peut être décrite de la façon suivante:

```
if (expression) instruction;  
ou :  
if (expression) {  
    instruction1;  
    instruction2;  
}
```

L'instruction conditionnelle else

```
if (expression) {  
    instruction1;  
}  
else {  
    instruction2;  
}
```

Structures de contrôle

Les instructions conditionnelles imbriquées

Java permet d'écrire ce type de structure sous la forme :

```
if (expression1) {  
    bloc1;  
}  
  
else if (expression2) {  
    bloc2;  
}  
  
else if (expression3) {  
    bloc3;  
}  
  
else {  
    bloc5;  
}
```

Structures de contrôle: L'instruction switch

Syntaxe :

```
switch( variable) {  
    case valeur1: instr1;break;  
    case valeur2: instr2;break;  
    case valeurN: instrN;break;  
    default: instr;break;
```

Exemple:

```
import java.util.Scanner;  
  
public class Test {  
    public static void main(String[] args) {  
        System.out.print("Donner un nombre:");  
        Scanner clavier=new Scanner(System.in);  
        int nb=clavier.nextInt();  
        switch(nb){  
            case 1 : System.out.println("Lundi");break;  
            case 2 : System.out.println("Mardi");break;  
            case 3 : System.out.println("Mercredi");break;  
            default :System.out.println("Autrement");break;  
        }  
    }  
}
```

Structures de contrôle

La boucle for

La boucle **for** est une structure employée pour exécuter un bloc d'instructions un nombre de fois en principe connu à l'avance. Elle utilise la syntaxe suivante :

```
for (initialisation;test;incrémentation) {  
    instructions;  
}
```

Exemple :

```
for (int i = 2; i < 10;i++) {  
    System.out.println("I="+i);  
}
```

Structures de contrôle

Sortie d'une boucle par return

```
int[] tab=new int[]{4,6,5,8};  
for (int i = 0; i < tab.length; i++) {  
    if (tab[i] == 5) {  
        return i;  
    }  
}
```

Branchement au moyen des instructions break et continue

break:

```
int x = 10;  
for (int i = 0; i < 10; i++) {  
    x--;  
    if (x == 5) break; _____  
}  
System.out.println(x);            ←
```

continue:

```
for (int i = 0; i < 10; i++) {       ←  
    if (i == 5) continue; _____  
    System.out.println(i);  
}
```

Structures de contrôle

L 'instruction While

```
while (condition) {  
    BlocInstructions;  
}
```

L 'instruction do .. while

```
do{  
    BlocInstructions;  
}  
while (condition);
```

Exemple :

```
int s=0;int i=0;  
while (i<10){  
    s+=i;  
    i++;  
}  
System.out.println("Somme="+s);
```

Exemple :

```
int s=0;int i=0;  
do{  
    s+=i;  
    i++;  
}while (i<10);  
System.out.println("Somme="+s);
```

Programmation orientée objet avec JAVA

Méthode orientée objet

La méthode orientée objet permet de concevoir une application sous la forme d'un ensemble d'objets reliés entre eux par des relations

Lorsque que l'on programme avec cette méthode, la première question que l'on se pose plus souvent est :

«qu'est-ce que je manipule ? »,

Au lieu de « qu'est-ce que je fait ? ».

L'une des caractéristiques de cette méthode permet de concevoir de nouveaux objets à partir d'objets existants.

On peut donc réutiliser les objets dans plusieurs applications.

La réutilisation du code fut un argument déterminant pour venter les avantages des langages à objets.

Pour faire la programmation orientée objet il faut maitriser les fondamentaux de l'orienté objet à savoir:

Objet et classe

Héritage

Encapsulation (Accessibilité)

Polymorphisme

Objet

Un objet est une structure informatique définie par un état et un comportement

Objet=état + comportement

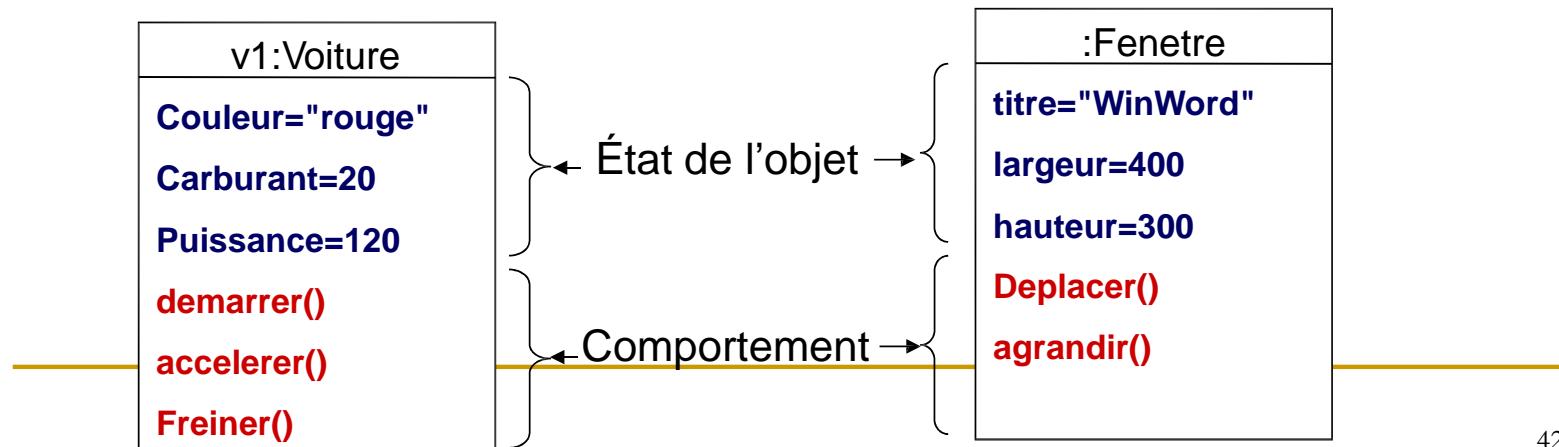
L'état regroupe les valeurs instantanées de tous les attributs de l'objet.

Le comportement regroupe toutes les compétences et décrit les actions et les réactions de l'objet. Autrement dit le comportement est défini par les opérations que l'objet peut effectuer.

L'état d'un objet peut changer dans le temps.

Généralement, c'est le comportement qui modifie l'état de l'objet

Exemples:



Identité d'un objet

En plus de son état, un objet possède une **identité** qui caractérise son existence propre.

Cette identité s'appelle également référence ou handle de l'objet

En terme informatique de bas niveau, l'identité d'un objet représente son adresse mémoire.

Deux objets ne peuvent pas avoir la même identité: c'est-à-dire que deux objet ne peuvent pas avoir le même emplacement mémoire.

Classes

Les objets qui ont des caractéristiques communes sont regroupés dans une entité appelé classe.

La classe décrit le domaine de définition d'un ensemble d'objets.

Chaque objet appartient à une classe

Les généralités sont contenues dans les classes et les particularités dans les objets.

Les objets informatique sont construits à partir de leur classe par un processus qui s'appelle l'instanciation.

Tout objet est une instance d'une classe.

Caractéristique d'une classe

Une classe est définie par:

- Les attributs
- Les méthodes

Les attributs permettent de décrire l'état de des objets de cette classe.

Chaque attribut est défini par:

- Son nom
- Son type
- Éventuellement sa valeur initiale

Les méthodes permettent de décrire le comportement des objets de cette classe.

Une méthode représente une procédure ou une fonction qui permet d'exécuter un certain nombre d'instructions.

Parmi les méthodes d'une classe, existe deux méthodes particulières:

Une méthode qui est appelée au moment de la création d'un objet de cette classe. Cette méthode est appelée **CONSTRUCTEUR**

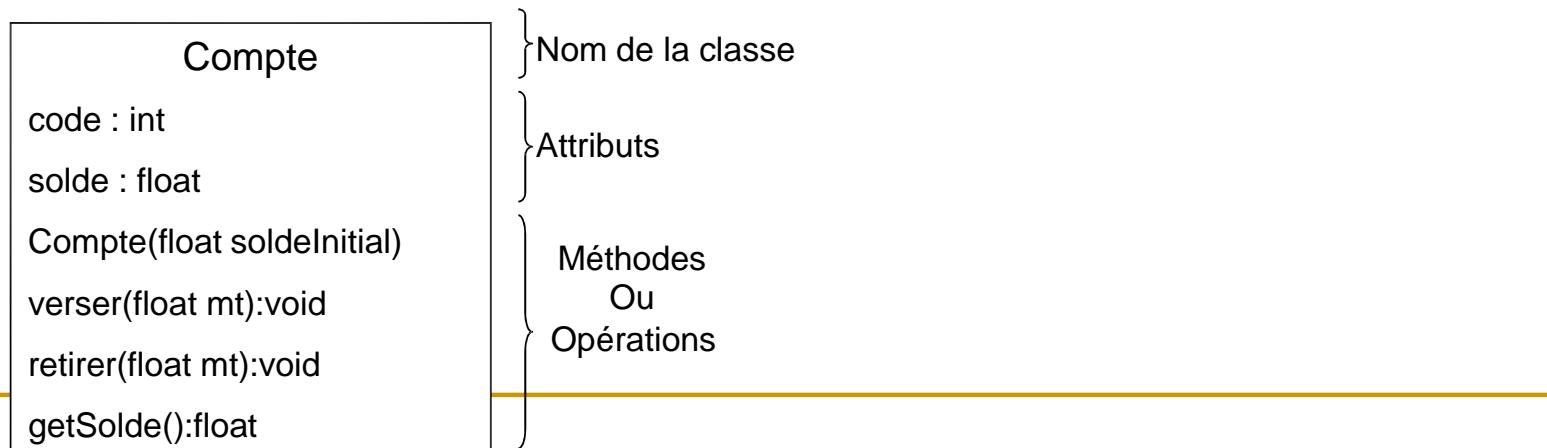
Une méthode qui est appelée au moment de la destruction d'un objet. Cette méthode s'appelle le **DESTRUCTEUR**

Représentation UML d'une classe

Une classe est représenté par un rectangle à 3 compartiments:

- Un compartiment qui contient le nom de la classe
- Un compartiment qui contient la déclaration des attributs
- Un compartiment qui contient les méthodes

Exemples:

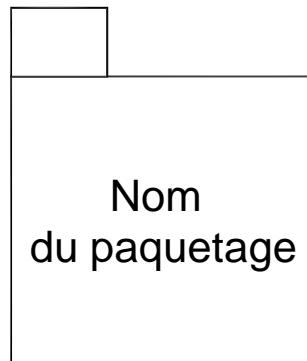


Les classes sont stockées dans des packages

Les packages offrent un mécanisme général pour la partition des modèles et le regroupement des éléments de la modélisation

Chaque package est représenté graphiquement par un dossier

Les packages divisent et organisent les modèles de la même manière que les dossier organisent le système de fichier



Accessibilité au membres d'une classe

Dans java, il existe 4 **niveaux de protection** :

private (-) : Un membre privé d'une classe n'est accessible qu'à l'intérieur de cette classe.

protected (#) : un membre protégé d'une classe est accessible à :

- L'intérieur de cette classe

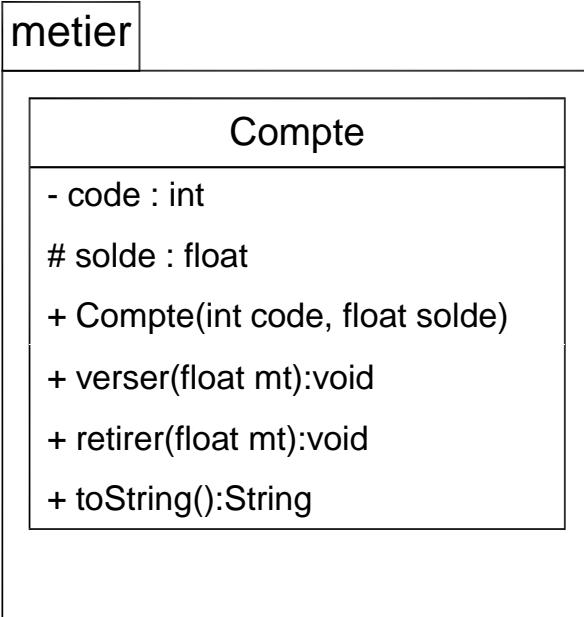
- Aux classes dérivées de cette classe.

- Aux classes du même package.

public (+) : accès à partir de toute entité interne ou externe à la classe

Autorisation par défaut : dans java, en l'absence des trois autorisations précédentes, l'autorisation par défaut est **package**. Cette autorisation indique que uniquement les classes du même package ont l'autorisation d'accès.

Exemple d'implémentation d'une classe avec Java



```
package metier;
public class Compte {
    // Attributs private
    int code; protected
    float solde;
    // Constructeur
    public Compte(int c, float s) {
        code=c;
        solde=s;
    }
    // Méthode pour verser un montant
    public void verser(float mt) {
        solde+=mt;
    }
    // Méthode pour retirer un montant
    public void retirer(float mt) {
        solde-=mt;
    }
    // Une méthode qui retourne l'état du compte
    public String toString() {
        return (" Code="+code+" Solde="+solde);
    }
}
```

Création des objets dans java

Dans java, pour créer un objet d'une classe , On utilise la commande new suivie du constructeur de la classe.

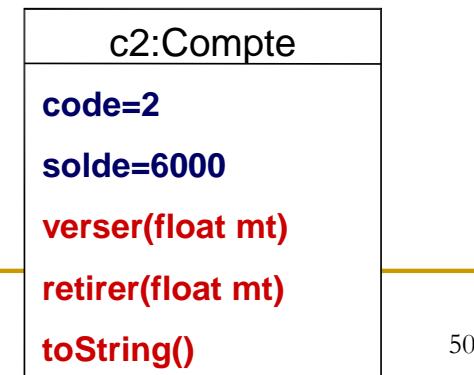
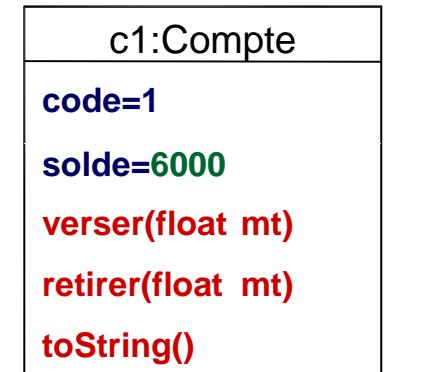
La commande new Crée un objet dans l'espace mémoire et retourne l'adresse mémoire de celui-ci.

Cette adresse mémoire devrait être affectée à une variable qui représente l'identité de l'objet. Cette référence est appelée handle.

```
package test;

import metier.Compte;

public class Application {
    public static void main(String[] args) {
        Compte c1=new Compte(1,5000);
        Compte c2=new Compte(2,6000);
        c1.verser(3000);
        c1.retirer(2000);
        System.out.println(c1.toString());
    }
}
```



Constructeur par défaut

Quand on ne définit aucun constructeur pour une classe, le compilateur crée le constructeur par défaut.

Le constructeur par défaut n'a aucun paramètre et ne fait aucune initialisation

Exemple de classe :

```
public class Personne {  
    // Les Attributs  
    private int code;  
    private String nom;  
    // Les Méthodes  
    public void setNom(String n) {  
        this.nom=n;  
    }  
    public String getNom() {  
        return nom;  
    }  
}
```

Instanciation en utilisant le constructeur par défaut :

```
Personne p=new Personne();  
p.setNom("AZER");  
System.out.println(p.getNom());
```

Getters et Setters

Les attributs privés d'une classe ne sont accessibles qu'à l'intérieur de la classe.

Pour donner la possibilité à d'autres classes d'accéder aux membres privés, il faut définir dans la classes des méthodes publiques qui permettent de :

lire la variables privés. Ce genre de méthodes s'appellent les **accesseurs** ou **Getters**
modifier les variables privés. Ce genre de méthodes s'appellent les **mutateurs** ou
Setters

Les getters sont des méthodes qui commencent toujours par le mot **get** et finissent par le nom de l'attribut en écrivant en majuscule la lettre qui vient juste après le get. Les getters retourne toujours le même type que l'attribut correspondant.

Par exemple, dans la classe CompteSimple, nous avons défini un attribut privé :
private String nom;

Le getter de cette variable est :

```
public String getNom(){  
    return nom;  
}
```

Les setters sont des méthodes qui commencent toujours par le mot set et finissent par le nom de l'attribut en écrivant en majuscule la lettre qui vient juste après le set. Les setters sont toujours de type void et reçoivent un paramètre qui est de même type que la variable:

Exemple:

```
public void setNom( String n ){  
    this.nom=n;  
}
```

Encapsulation

```
public class Application {  
    public static void main(String[] args) {  
        Personne p=new Personne();  
        p.setNom("AZER");  
        System.out.println(p.getNom());  
    }  
}
```

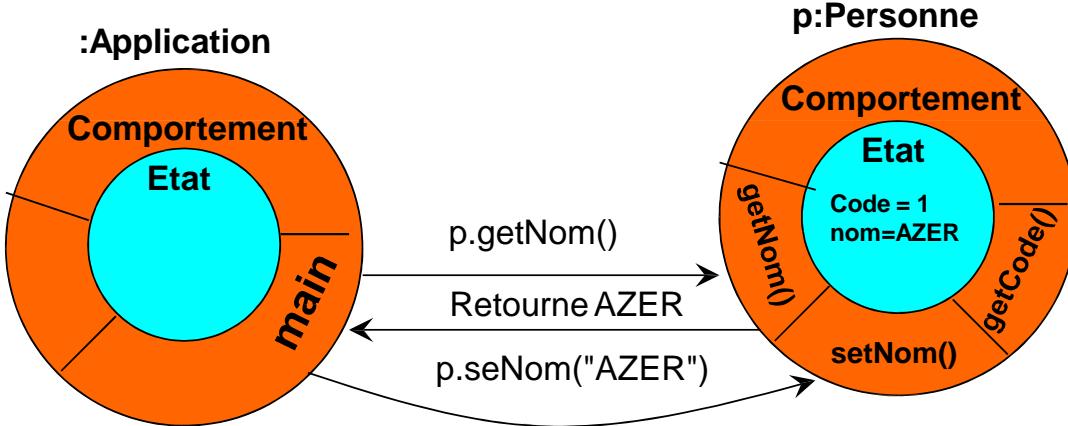
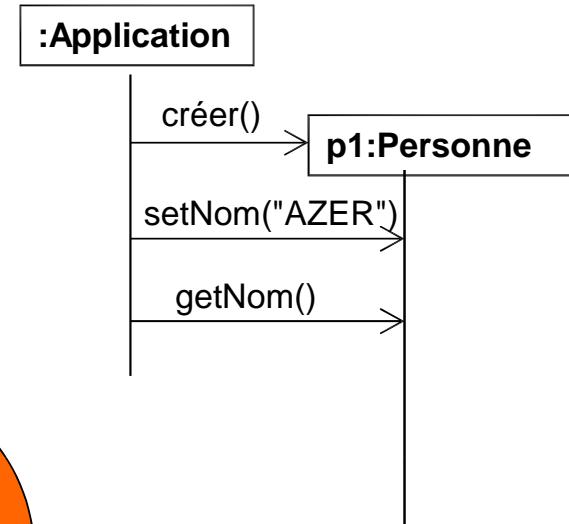


Diagramme de séquence :



- Généralement, l'état d'un objet est privé ou protégé et son comportement est public
- Quand l'état de l'objet est privé Seules les méthodes de ses qui ont le droit d'y accéder
- Quand l'état de l'objet est protégé, les méthodes des classes dérivées et les classes appartenant au même package

Membres statiques d'une classe.

Dans l'exemple de la classe Compte, chaque objet Compte possède ses propres variables code et solde. Les variables code et solde sont appelées variables d'instances.

Les objets d'une même classe peuvent partager des mêmes variables qui sont stockées au niveau de la classe. Ce genre de variables, s'appellent les variables statiques ou variables de classes.

Un attribut statique d'une classe est un attribut qui appartient à la classe et partagé par tous les objets de cette classe.

Comme un attribut une méthode peut être déclarée statique, ce qui signifie qu'elle appartient à la classe et partagée par toutes les instances de cette classe.

Dans la notation UML, les membres statiques d'une classe sont soulignés.

Exemple:

Supposant nous voulions ajouter à la classe Compte une variable qui permet de stocker le nombre de comptes créés.

Comme la valeur de variable nbComptes est la même pour tous les objets, celle-ci sera déclarée statique. Si non, elle sera dupliquée dans chaque nouveau objet créé.

La valeur de nbComptes est au départ initialisée à 0, et pendant la création d'une nouvelle instance (au niveau du constructeur), nbCompte est incrémentée et on profite de la valeur de nbComptes pour initialiser le code du compte.

Compte
- code : int
solde : float
- <u>nbComptes:int</u>
+ Compte(float solde)
+ verser(float mt):void
+ retirer(float mt):void
+ toString():String
+ <u>getNbComptes():int</u>

```
package metier;
public class Compte {
    // Variables d'instances
    private int code;
    private float solde;
    // Variable de classe ou statique
    private static int nbComptes;
    public Compte(float solde) {
        this.code=++nbComptes;
        this.solde=solde;
    }
    // Méthode pour verser un montant
    public void verser(float mt) {
        solde+=mt;
    }
    // Méthode pour retirer un montant
    public void retirer(float mt) {
        solde-=mt;
    }
    // retourne l'état du compte
    public String toString() {
        return(" Code="+code+" Solde="+solde);
    }
    // retourne la valeur de nbComptes
    public static int getNbComptes() {
        return(nbComptes);
    }
}
```

Application de test

```
package test;

import metier.Compte;
public class Application {
    public static void main(String[] args) {
        Compte c1=new Compte(5000);
        Compte c2=new Compte(6000);
        c1.verser(3000);
        c1.retirer(2000);
        System.out.println(c1.toString());
        System.out.println(Compte.nbComptes);
        System.out.println(c1.nbComptes)
    }
}
```

Classe Compte
<u>nbCompte=2</u>
<u>getNbComptes()</u>

c1:Compte	c2:Compte
code=1 solde=6000	code=2 solde=6000
verser(float mt) retirer(float mt) toString()	verser(float mt) retirer(float mt) toString()

Code=1 Solde= 6000
2
2

Destruction des objets : Garbage Collector

Dans certains langages de programmation, le programmeur doit s'occuper lui-même de détruire les objets inutilisables.

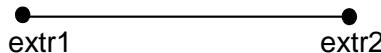
Java détruit automatiquement tous les objets inutilisables en utilisant ce qu'on appelle le **garbage collector (ramasseur d'ordures)**. Qui s'exécute automatiquement dès que la mémoire disponible est inférieure à un certain seuil.

Tous les objets qui ne sont pas retenus par des handles seront détruits.

Ce phénomène ralenti parfois le fonctionnement de java.

Pour signaler au garbage collector que vous voulez détruire un objet d'une classe, vous pouvez faire appel à la méthode `finalize()` redéfinie dans la classe.

Exercice 1 : Modélisation d'un segment



On souhaite créer une application qui permet de manipuler des segments. Un segment est défini par la valeur de ses deux extrémités extr1 et extr2. Pour créer un segment, il faut préciser les valeurs de extr1 et extr2. Les opérations que l'on souhaite exécuter sur le segment sont :

ordonne() : méthode qui permet d'ordonner extr1 et extr2 si extr1 est supérieur à extr2

getLongueur() : méthode qui retourne la longueur du segment.

appartient(int x) : retourne si x appartient au segment ou non.

toString() : retourne une chaîne de caractères de type SEGMENT[extr1,extr2]

Faire une représentation UML de la classe Segment.

Implémenter en java la classe Segment

Créer une application TestSegment qui permet de :

Créer objet de la classe Segment avec les valeurs extr1=24 et extr2=12.

Afficher l'état de cet objet en utilisant la méthode toString().

Afficher la longueur de ce segment.

Afficher si le point x=15, appartient à ce segment.

Changer les valeurs des deux extrémités de ce segment.

Afficher à nouveau la Longueur du segment

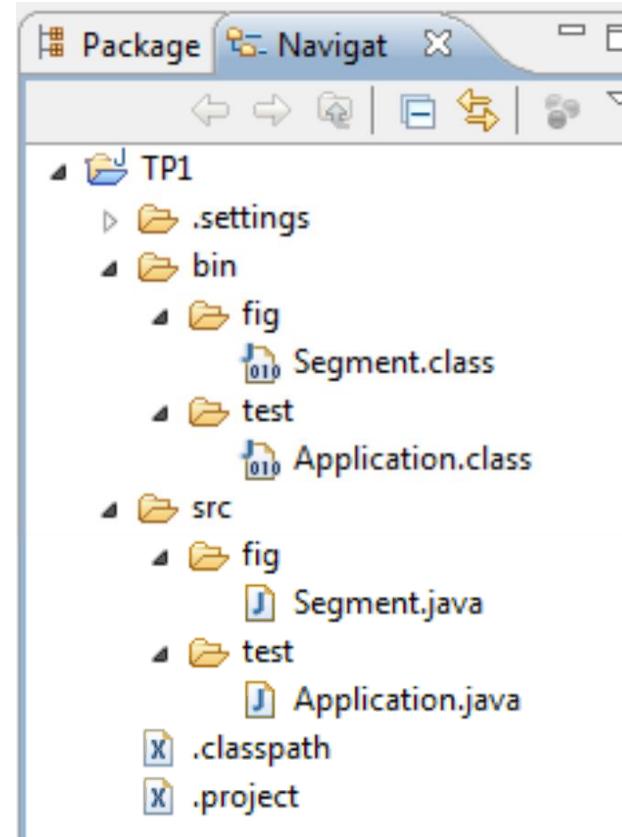
Diagramme de classes

Segment
+ extr1 : int
+ extr2 : int
+ Segment (int e1,int e2)
+ ordonne()
+ getLongueur() : int
+ appartient(int x) : boolean
+ toString() : String

TestSegment
<u>+ main(String[] args):void</u>

Solution : Segment.java

```
package fig;
public class Segment {
    public int extr1;
    public int extr2;
    // Constructeur
    public Segment(int a,int b){
        extr1=a;extr2=b;ordonne();
    }
    public void ordonne(){
        if(extr1>extr2){
            int z=extr1;
            extr1=extr2;
            extr2=z;
        }
    }
    public int getLongueur(){
        return(extr2-extr1);
    }
    public boolean appartient(int x){
        if((x>extr1)&&(x<extr2))
            return true;
        else return false;
    }
    public String toString(){
        return ("segment["+extr1+","+extr2+"]);"
    }
}
```



Application

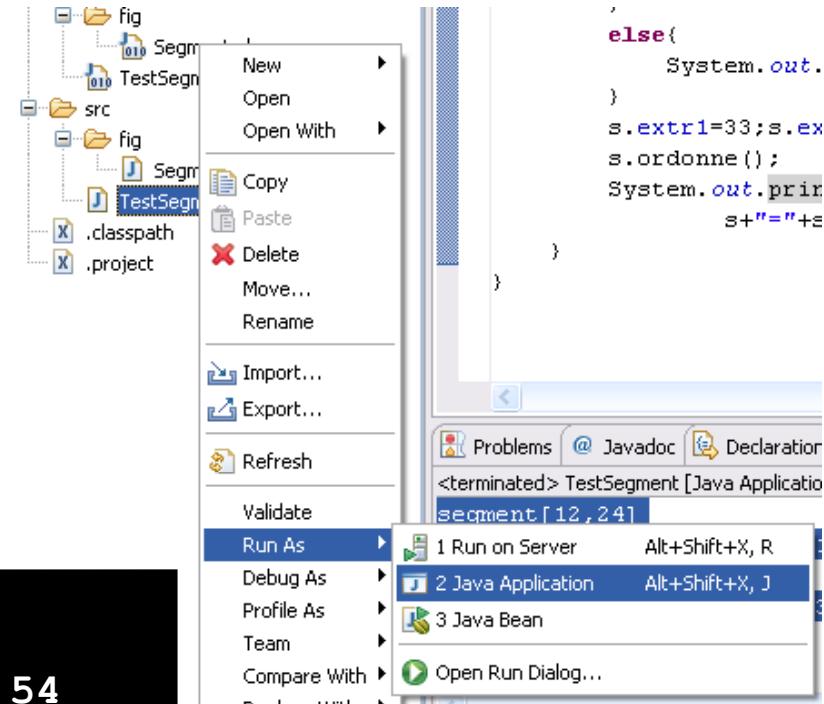
```
package test;
import java.util.Scanner;
import fig.Segment;
public class Application {
    public static void main(String[] args) {
        Scanner clavier=new Scanner(System.in);
        System.out.print("Donner Extr1:");
        int e1=clavier.nextInt();
        System.out.print("Donner Extr2:");
        int e2=clavier.nextInt();
        Segment s=new Segment(e1, e2);
        System.out.println("Longueur du"+s.toString()+" est :"+
            s.getLongueur());
        System.out.print("Donner X:");
        int x=clavier.nextInt();
        if(s.appartient(x)==true)
            System.out.println(x+" Appartient au "+s);
        else
            System.out.println(x+" N'appartient pas au "+s);
    }
}
```

```
Donner Extr1:67
Donner Extr2:13
Longueur du segment[13,67] est :54
Donner X:7
7 N'appartient pas au segment[13,67]
```

Exécution de la classe TestSegment

Pour exécuter une application (Classe qui contient la méthode main) avec Eclipse, on clique avec le bouton droit de la souris sur la classe, puis on choisit dans le menu contextuel, Run As > Java Application

```
Donner Extr1:67
Donner Extr2:13
Longueur dusegment[13,67] est :54
Donner X:7
7 N'appartient pas au segment[13,67]
```



Exercice 2

Une cercle est défini par :

Un point qui représente son centre : centre(x,y) et un rayon.

On peut créer un cercle de deux manières :

Soit en précisant son centre et un point du cercle.

Soit en précisant son centre et son rayon

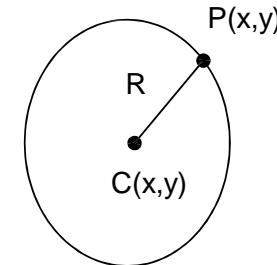
Les opérations que l'on souhaite exécuter sur un cercle sont :

getPerimetre() : retourne le périmètre du cercle

getSurface() : retourne la surface du cercle.

appartient(Point p) : retourne si le point p appartient ou non à l'intérieur du cercle.

toString() : retourne une chaîne de caractères de type CERCLE(x,y,R)



1. Etablir le diagramme de classes

2. Créer les classe Point définie par:

Les attributs x et y de type int

Un constructeur qui initialise les valeurs de x et y .

Une méthode `toString()`.

3. Créer la classe Cercle

4. Créer une application qui permet de :

a. Créer un cercle défini par le centre $c(100,100)$ et un point $p(200,200)$

b. Créer un cercle défini par le centre $c(130,100)$ et de rayon $r=40$

c. Afficher le périmètre et le rayon des deux cercles.

d. Afficher si le point $p(120,100)$ appartient à l'intersection des deux cercles ou non.

Héritage et accessibilité

Héritage

Dans la programmation orientée objet, l'héritage offre un moyen très efficace qui permet la réutilisation du code.

En effet une classe peut hériter d'une autre classe des attributs et des méthodes.

L'héritage, quand il peut être exploité, fait gagner beaucoup de temps en terme de développement et en terme de maintenance des applications.

La réutilisation du code fut un argument déterminant pour venter les méthodes orientées objets.

Exemple de problème

Supposons que nous souhaitions créer une application qui permet de manipuler différents types de comptes bancaires: les compte simple, les comptes épargnes et les comptes payants.

Tous les types de comptes sont caractériser par:

Un code et un solde

Lors de la création d'un compte, son code qui est défini automatiquement en fonction du nombre de comptes créés;

Un compte peut subir les opérations de versement et de retrait. Pour ces deux opérations, il faut connaître le montant de l'opération.

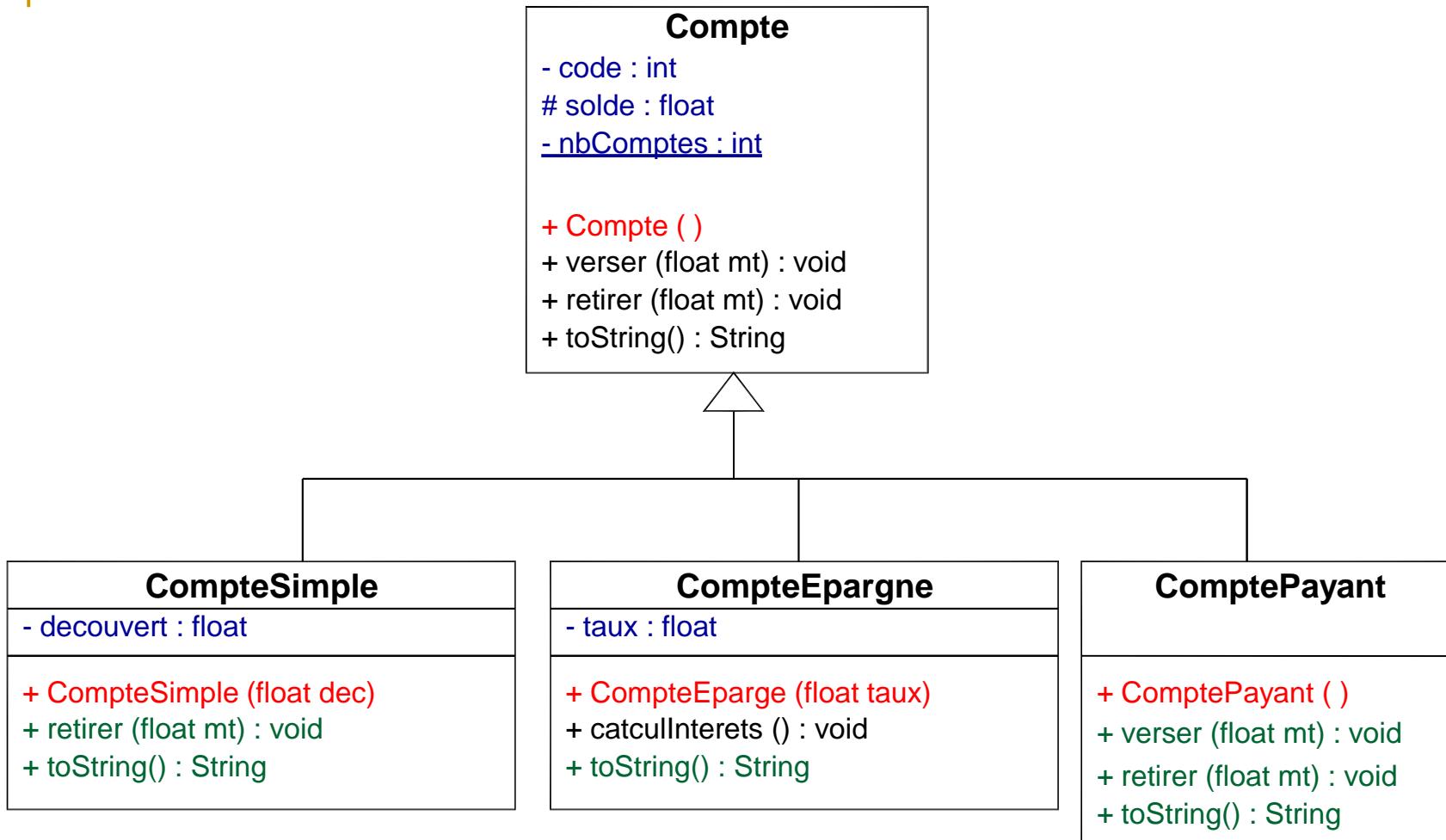
Pour consulter un compte on peut faire appel à sa méthode `toString()`

Un compte simple est un compte qui possède un découvert. Ce qui signifie que ce compte peut être débiteur jusqu'à la valeur du découvert.

Un compte Epargne est un compte bancaire qui possède en plus un champ «tauxIntérêt» et une méthode `calculIntérêt()` qui permet de mettre à jour le solde en tenant compte des intérêts.

Un ComptePayant est un compte bancaire pour lequel chaque opération de retrait et de versement est payante et vaut 5 % du montant de l'opération.

Diagramme de classes



Implémentation java de la classe Compte

```
public class Compte {  
    private int code;  
    protected float solde;  
    private static int nbComptes;  
  
    public Compte( ) {  
        ++nbComptes;  
        code=nbComptes;  
        this.solde=0;  
    }  
    public void verser(float mt) {  
        solde+=mt;  
    }  
    public void retirer(float mt) {  
        if (mt<solde) solde-=mt;  
    }  
    public String toString() {  
        return ("Code="+code+" Solde="+solde);  
    }  
}
```

Héritage : extends

La classe CompteSimple est une classe qui hérite de la classe Compte.

Pour désigner l'héritage dans java, on utilise le mot **extends**

```
public class CompteSimple extends Compte {  
}
```

La classe CompteSimple hérite de la classe CompteBancaire tout ses membres sauf le constructeur.

Dans java une classe hérite toujours d'une seule classe.

Si une classe n'hérite pas explicitement d'une autre classe, elle hérite implicitement de la classe Object.

La classe Compte hérite de la classe Object.

La classe CompteSimple hérite directement de la classe Compte et indirectement de la classe Object.

Définir les constructeur de la classe dérivée

Le constructeur de la classe dérivée peut faire appel au constructeur de la classe parente en utilisant le mot **super()** suivi de ses paramètres.

```
public class CompteSimple extends Compte {  
    private float découvert;  
    //constructeur  
    public CompteSimple(float découvert) {  
        super();  
        this.découvert=découvert;  
    }  
}
```

Redéfinition des méthodes

Quand une classe hérite d'une autre classe, elle peut redéfinir les méthodes héritées.

Dans notre cas la classe CompteSimple hérite de la classe Compte la méthode retirer(). nous avons besoin de redéfinir cette méthode pour prendre en considération la valeur du découvert.

```
public class CompteSimple extends Compte {  
    private float découvert;  
    // constructeur  
    public CompteSimple(float découvert) {  
        super();  
        this.découvert=découvert;  
    }  
    // Redéfinition de la méthode retirer  
    public void retirer(float mt) {  
        if(mt-découvert<=solde)  
            solde-=mt;  
    }  
}
```

Redéfinition des méthodes

Dans la méthode redéfinie de la nouvelle classe dérivée, on peut faire appel à la méthode de la classe parente en utilisant le mot **super** suivi d'un point et du nom de la méthode

Dans cette nouvelle classe dérivée, nous allons redéfinir également la méthode `toString()`.

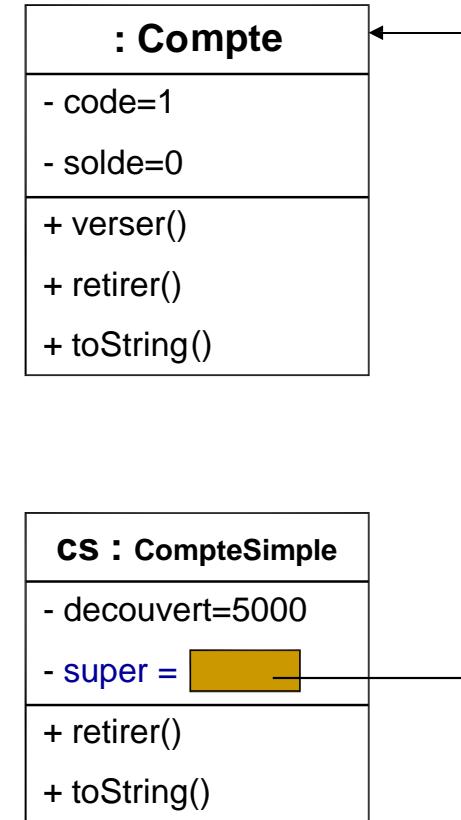
```
public class CompteSimple extends Compte {  
    private float découvert;  
  
    // constructeur  
    // Redéfinition de la méthode retirer  
    public void retirer(float mt) {  
        if (mt+découvert>solde)  
            solde-=mt;  
    }  
    // Redéfinition de la méthode toString  
    public String toString() {  
        return ("Compte Simple "+super.toString()+"  
Découvert="+découvert);  
    }  
}
```

Héritage à la loupe : Instanciation

Quand on crée une instance d'une classe, la classe parente est automatiquement instanciée et l'objet de la classe parente est associé à l'objet créé à travers la référence « **super** » injectée par le compilateur

```
CompteSimple cs=new CompteSimple(5000);
```

Lors de l'instanciation, l'héritage entre les classes est traduit par une composition entre un objet de la classe instanciée et d'un objet de la classe parente qui est créé implicitement.



Surcharge

Dans une classe, on peut définir plusieurs constructeurs. Chacun ayant une signature différentes (paramètres différents)

On dit que le constructeur est surchargé

On peut également surcharger une méthode. Cela peut dire qu'on peut définir, dans la même classe plusieurs méthodes qui ont le même nom et des signatures différentes;

La signature d'une méthode désigne la liste des arguments avec leurs types.

Dans la classe CompteSimple, par exemple, on peut ajouter un autre constructeur sans paramètre

Un constructeur peut appeler un autre constructeur de la même classe en utilisant le mot **this()** avec des paramètres éventuels

Surcharge de constructeurs

```
public class CompteSimple extends Compte {  
    private float découvert;  
    //Premier constructeur  
    public CompteSimple(float découvert) {  
        super();  
        this.découvert=découvert;  
    }  
    //Deuxième constructeur  
    public CompteSimple() {  
        this(0);  
    }  
}
```

On peut créer une instance de la classe **CompteSimple** en faisant appel à l'un des deux constructeur :

```
CompteSimple cs1=new CompteSimple(5000);  
CompteSimple cs2=new CompteSimple();
```

Accessibilité

Accessibilité

Les trois critères permettant d'utiliser une classe sont *Qui*, *Quoi*, *Où*. Il faut donc :

Que l'utilisateur soit autorisé (*Qui*).

Que le type d'utilisation souhaité soit autorisé (*Quoi*).

Que l'adresse de la classe soit connue (*Où*).

Pour utiliser donc une classe, il faut :

Connaitre le package où se trouve la classe (*Où*)

Importer la classe en spécifiant son package.

Qu'est ce qu'on peut faire avec cette classe:

Est-ce qu'on a le droit de l'instancier

Est-ce qu'on a le droit d'exploiter les membres de ses instances

Est-ce qu'on a le droit d'hériter de cette classe.

Est-ce qu'elle contient des membres statiques

Connaitre qui a le droit d'accéder aux membres de cette instance.

Les packages (Où)

Nous avons souvent utilisé la classe System pour afficher un message : **System.out.println()**,

En consultant la documentation de java, nous allons constater que le chemin d'accès complet à la classe System est java.lang.System.

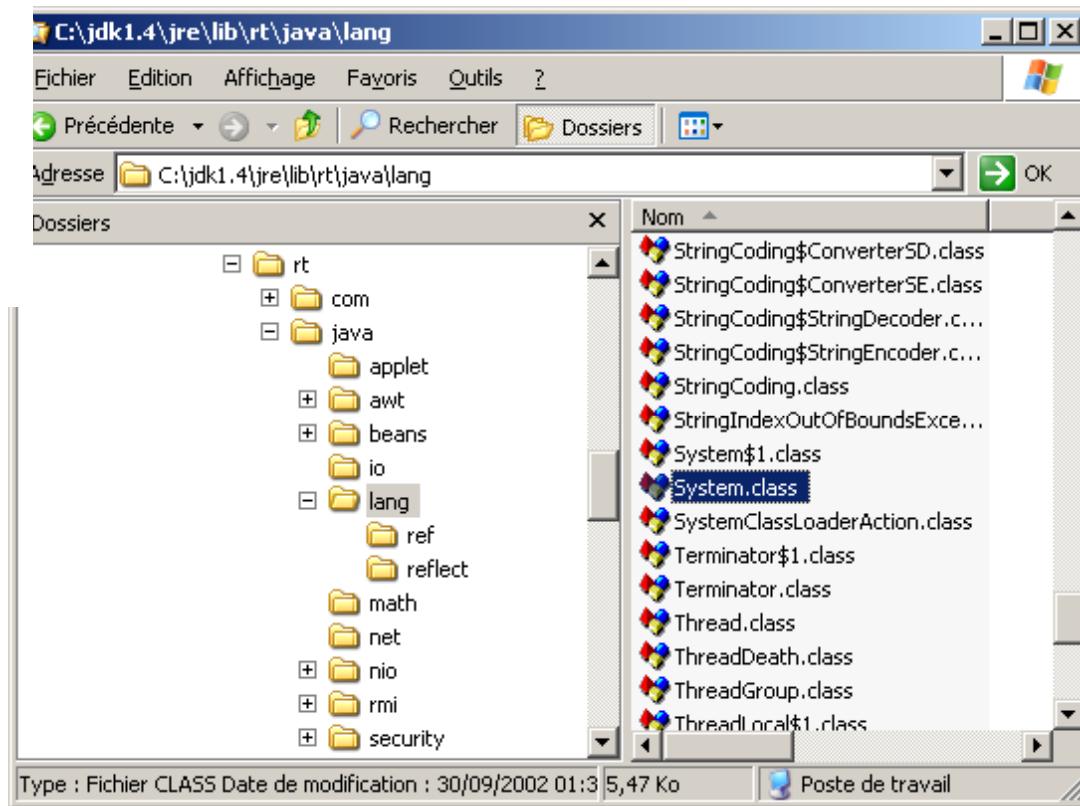
La classe System étant stockée dans le sous dossier lang du dossier java.

java.lang.System est le chemin d'accès qui présente la particularité d'utiliser un point « . » comme séparateur.

Java.lang qui contient la classe System est appelé « **package** »

Les packages (Où)

java.lang
Class System
java.lang.Object
└─java.lang.System



Les packages (Où)

Notion de package:

Java dispose d'un mécanisme pour la recherche des classes.

Au moment de l'exécution, La JVM recherche les classes en priorité :

Dans le répertoire courant, c'est-à-dire celui où se trouve la classe appelante, si la variable d'environnement **CLASSPATH** n'est pas définie ;

Dans les chemins spécifiés par la variable d'environnement **CLASSPATH** si celle-ci est définie.

L'instruction package:

Si vous souhaitez qu'une classe que vous avez créée appartienne à un package particulier, vous devez le spécifier explicitement au moyen de l'instruction **package**, suivie du nom du package.

Cette instruction doit être la première du fichier.

Elle concerne toutes les classes définies dans ce fichier.

L'instruction import

Pour utiliser une classe, il faut

Soit écrire le nom de la classe précédée par son package.

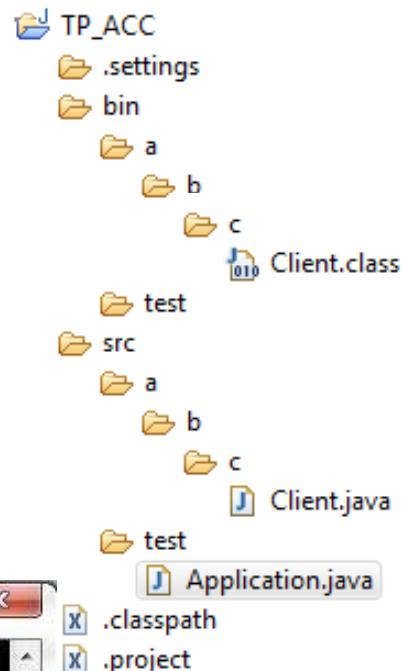
Soit importer cette classe en la déclarant dans la clause import. Et dans ce cas là, seul le nom de la classe suffit pour l'utiliser.

Les packages (Où)

Application:

```
package a.b.c;
public class Client {
    private int code;
    private String nom;
    public Client(int code, String nom) {
        this.code = code;
        this.nom = nom;
    }
    public String getNom() {
        return nom;
    }
    public int getCode() {
        return code;
    }
}
```

```
package test;
import a.b.c.Client;
public class Application {
    public static void main(String[] args) {
        Client c=new Client(2,"Salih");
        System.out.println("Nom="+c.getNom());
    }
}
```



Pour Compiler la classe **Client.java** sur ligne de commande:

javac -d cheminbin Client.java

cheminbin représente le dossier des fichiers .class.

```
C:\JA2\TP_ACC\src>javac -d ../bin a/b/c/Client.java
C:\JA2\TP_ACC\src>_
```

Les packages (Où)

Les fichiers .jar

- Les fichiers **.jar** sont des fichiers compressés comme les fichiers **.zip** selon un algorithme particulier devenu un standard.
- Ils sont parfois appelés *fichiers d'archives* ou, plus simplement, *archives*. Ces fichiers sont produits par des outils de compression tels que Pkzip (sous DOS) ou Winzip (sous Windows), ou encore par **jar.exe**.
- Les fichiers **.jar** peuvent contenir une multitude de fichiers compressés avec l'indication de leur chemin d'accès.
- Les packages standard de Java sont organisés de cette manière, dans un fichier nommé **rt.jar** placé dans le sous-répertoire **lib** du répertoire où est installé le JDK.
- Dans le cas d'une installation standard de Java 6 sur le disque C:, le chemin d'accès complet à la classe **System** est donc : c:\jdk1.6\jre\lib\rt.jar\java\lang\System

Création de vos propres fichiers .jar ou .zip

Vous pouvez utiliser le programme jar.exe du jdk pour créer les fichiers **.jar**

Syntaxe : jar [options] nomarchive.jar fichiers

Exemple qui permet d'archive le contenu du dossier a :

```
C:\AJ2\TP_ACC\bin> jar cf archive.jar a
```

Ce qui peut être fait(Quoi)

Nous avons maintenant fait le tour de la question *Où* ?

Pour qu'une classe puisse être utilisée (directement ou par l'intermédiaire d'un de ses membres), il faut non seulement être capable de la trouver, mais aussi qu'elle soit adaptée à l'usage que l'on veut en faire.

Une classe peut servir à plusieurs choses :

- Créer des objets, en étant **instanciée**.

- Créer de nouvelles classes, en étant **étendue**.

- On peut utiliser directement ses membres statiques (sans qu'elle soit instanciée.)

- On peut utiliser les membres de ses instances.

Les différents modificateurs qui permettent d'apporter des restrictions à l'utilisation d'une classe sont:

abstract, final, static, synchronized et native

Classe abstraite

Une classe abstraite est une classe qui ne peut pas être instanciée.

La classe Compte de notre modèle peut être déclarée abstract pour indiquer au compilateur que cette classe ne peut pas être instancié.

Une classe abstraite est généralement créée pour en faire dériver de nouvelle classe par héritage.

```
public abstract class Compte {  
    private int code;  
    protected float solde;  
    private static int nbComptes;  
    // Constructeurs  
    // Méthodes  
}
```

Les méthodes abstraites

Une méthode abstraite peut être déclarée à l'intérieur d'une classe abstraite.

Une méthode abstraite est une méthode qui n'a pas de définition.

Une méthode abstraite est une méthode qui doit être redéfinie dans les classes dérivées.

Exemple :

On peut ajouter à la classe Compte une méthode abstraite nommée afficher() pour indiquer que tous les comptes doivent redéfinir cette méthode.

Les méthodes abstraites

```
public abstract class Compte {  
    // Membres  
    ...  
    // Méthode abstraite  
    public abstract void afficher();  
  
}
```

```
public class CompteSimple extends Compte {  
    // Membres  
    ...  
    public void afficher(){  
        System.out.println("Solde="+solde+"  
Découvert="+decouvert);  
    }  
}
```

Interfaces

Une interface est une sorte de classe abstraite qui ne contient que des méthodes abstraites.

Dans java une classe hérite d'une seule classe et peut hériter en même temps de plusieurs interface.

On dit qu'une classe implémente une ou plusieurs interface.

Une interface peut hériter de plusieurs interfaces. Exemple d'interface:

```
public interface Solvable {  
    public void solver();  
    public double getSoile();  
}
```

Pour indiquer que la classe CompteSimple implémente cette interface on peut écrire:

```
public class CompteSimple extends Compte implements Solvable {  
    private float decouvert;  
    public void afficher() {  
        System.out.println("Solde="+solde+" Découvert="+decouvert);  
    }  
    public double getSoile() {  
        return solde;  
    }  
    public void solver() {  
        this.solde=0;  
    } }
```

Classe de type final

Une classe de type final est une classes qui ne peut pas être dérivée.

Autrement dit, on ne peut pas hériter d'une classe final.

La classe CompteSimple peut être déclarée final en écrivant:

```
public final class CompteSimple extends Compte {  
    private float découvert;  
    public void afficher() {  
        System.out.println("Solde="+solde+"  
Découvert="+découvert);  
    }  
}
```

Variables et méthodes final

Une variable final est une variable dont la valeur ne peut pas changer. Autrement dit, c'est une constante:

Exemple : `final double PI=3.14;`

Une méthode final est une méthode qui ne peut pas être redéfinie dans les classes dérivées.

Exemple : La méthode verser de la classe suivante ne peut pas être redéfinie dans les classes dérivées car elle est déclarée final

```
public class Compte {  
    private int code; protected float solde;  
    private static int nbComptes;  
  
    public final void verser(float mt) {  
        solde+=mt;  
    }  
    public void retirer(float mt) {  
        if(mt<solde) solde-=mt;  
    }  
}
```

Membres statiques d'une classe

Les membres (attributs ou méthodes) d'une classes sont des membres qui appartiennent à la classe et sont partagés par toutes les instances de cette classe.

Les membres statiques ne sont pas instanciés lors de l'instanciation de la classe

Les membres statiques sont accessible en utilisant directement le nom de la classe qui les contient.

Il n'est donc pas nécessaire de créer une instance d'une classe pour utiliser les membres statiques.

Les membre statiques sont également accessible via les instances de la classe qui les contient.

Exemple d'utilisation:

```
double d=Math.sqrt(9);
```

Ici nous avons fait appel à la méthode sqrt de la classe Math sans créer aucune instance. Ceci est possible car la méthode sqrt est statique.

Si cette méthode n'était pas statique, il faut tout d'abord créer un objet de la classe Math avant de faire appel à cette méthode:

```
Math m=new Math();
double d=m.sqrt(9);
```

Les seuls membres d'une classe, qui sont accessibles, sans instantiation sont les membres statiques.

Qui peut le faire (Qui):

private:

L'autorisation **private** est la plus restrictive. Elle s'applique aux membres d'une classe (variables, méthodes et classes internes).

Les éléments déclarés **private** ne sont accessibles que depuis la classe qui les contient.

Ce type d'autorisation est souvent employé pour les variables qui ne doivent être modifiées ou lues qu'à l'aide d'un *getter* ou d'un *setter*.

public:

Un membre public d'une classe peut être utilisé par n'importe quelle autre classe.

En UML le membres public sont indiqués par le signe +

protected:

Les membres d'une classe peuvent être déclarés **protected**.

Dans ce cas, l'accès en est réservé aux méthodes des classes appartenant au même package
aux classes dérivées de ces classes,
ainsi qu'aux classes appartenant aux mêmes packages que les classes dérivées.

Autorisation par défaut : **package**

L'autorisation par défaut, que nous appelons **package**, s'applique aux classes, interfaces, variables et méthodes.

Les éléments qui disposent de cette autorisation sont accessibles à toutes les méthodes des classes du même package.

Résumé: Héritage

Une classe peut hériter d'une autre classe en utilisant le mot **extends**.

Une classe Hérite d'une autre tout ses membres sauf le constructeur.

Il faut toujours définir le constructeur de la nouvelle classe dérivée.

Le constructeur de la classe dérivée peut appeler le constructeur de la classe parente en utilisant le mot **super()**, avec la liste des paramètres.

Quand une classe hérite d'une autre classe, elle a le droit de redéfinir les méthodes héritées.

Dans une méthode redéfinie, on peut appeler la méthode de la classe parente en écrivant le mot **super** suivi d'un point et du nom de la méthode parente. (**super.méthode()**).

Un constructeur peut appeler un autre constructeur de la même classe en utilisant le mot **this()** avec des paramètres du constructeur.

Résumé: Accessibilité

Pour utiliser une classe il faut connaître:

- Où trouver la classe (package)

- Quels sont les droits d'accès à cette classe (Quoi?)

- Quelles sont les classes qui ont le droit d'accéder aux membres de cette classe (Qui?)

Où?

- Pour utiliser une classe il faut importer son package en utilisant l'instruction **import**

- Pour déclarer le package d'appartenance d'une classe on utilise l'instruction **package**

- La variable d'environnement **classpath** permet de déclarer les chemins où la JVM trouvera les classes d'une application

Résumé: Accessibilité

Quoi?

abstract :

Une classe abstraite est une classe qui ne peut pas être instanciée.

Une méthode abstraite est une méthode qui peut être définie à l'intérieur d'une classe abstraite. C'est une méthode qui n'a pas de définition. Par conséquent, elle doit être redéfinie dans les classes dérivées.

Une interface est une sorte de classe abstraite qui ne contient que des méthodes abstraites.

Dans Java une classe hérite toujours d'une seule classe et peut implémenter plusieurs interfaces.

final:

Une classe final est une classe qui ne peut pas être dérivée.

Une méthode final est une méthode qui ne peut pas être redéfinie dans les classes dérivées.

Une variable final est une variable dont la valeur ne peut pas changer

On utilise final pour deux raisons: une raison de sécurité et une raison d'optimisation

static:

Les membres statiques d'une classe appartiennent à la classe et partagés par toutes ses objets

Les membres statiques sont accessibles en utilisant directement le nom de la classe

Les membres statiques sont accessibles sans avoir besoin de créer une instance de la classe qui les contient

Les membres statiques sont également accessibles via les instances de la classe qui les contient

Résumé: Accessibilité

Qui?

Java dispose de 4 niveaux d'autorisations:

private :

protected:

public:

package (Autorisation par défaut)

Travail à faire

Implémenter la classe Compte

Implémenter la classe CompteSimple

Implémenter la classe CompteEpargne

Implémenter la classe ComptePayant

Créer une application pour tester les différentes classes.

Compte.java

```
package metier;

public abstract class Compte {
    private int code;
    protected float solde;
    private static int nbComptes;

    public Compte(float s) {
        code=++nbComptes;
        this.solde=s;
    }
    public void retirer(float mt) {
        if(mt<solde) solde-=mt;
    }
    public void verser(float mt) {
        solde+=mt;
    }
    public String toString() {
        return("Code="+code+
            " Solde="+solde);
    }
}

// Getters et Setters
public int getCode() {
    return code;
}
public float getSolde() {
    return solde;
}
public static int getNbComptes() {
    return nbComptes;
}
```

CompteSimple.java

```
package metier;

public final class CompteSimple
    extends Compte {
private float decouvert;

// Constructeurs
public CompteSimple(float s, float d) {
    super(s);
    this.decouvert = d;
}
public CompteSimple() {
    super();
}
public void retirer(float mt) {
    if(solde+decouvert>mt) solde-=mt;
}
public String toString() {
    return "Compte Simple
        "+super.toString()+""
        Solde="+solde;
}

//Getters et Setters
public float getDecouvert() {
    return decouvert;
}
public void setDecouvert(float
    decouvert) {
    this.decouvert = decouvert;
}
```

CompteEpargne.java

```
package metier;

public class CompteEpargne extends
    Compte {
    private float taux;
    //Constructeurs

    public CompteEpargne() {
        this(0, 6);
    }

    public CompteEpargne(float
        solde, float taux) {
        super(solde);
        this.taux=taux;
    }
    public void calculInterets() {
        solde=solde*(1+taux/100);
    }
    public String toString() {
        return "Compte Epargne
            "+super.toString()+" Taux="+taux;
    }
}
```

```
// Getters et Setters
public float getTaux() {
    return taux;
}
public void setTaux(float taux) {
    this.taux = taux;
}
```

ComptePayant.java

```
package metier;

public class ComptePayant
    extends Compte {
    // Constructeur
    public ComptePayant(float
        solde) {
        super(solde);
    }
    public void verser(float mt) {
        super.verser(mt);
        super.retirer(mt*5/100);
    }
    public void retirer(float mt) {
        super.retirer(mt);
        super.retirer(mt*5/100);
    }
    public String toString() {
        return super.toString();
    }
}
```

Application TestCompte.java

```
package test;

import metier.*;
public class TestCompte {
public static void main(String[] args) {
// Tester la classe Compte Simple
CompteSimple c1=new CompteSimple(8000,4000);
System.out.println(c1.toString());
c1.verser(3000);
c1.retirer(5000);
c1.setDecouvert(5500);
System.out.println(c1.toString());

// Tester la classe Compte Epargne
CompteEpargne c2=new CompteEpargne(50000,5);
System.out.println(c2.toString());
c2.verser(30000);
c2.retirer(6000);
c2.calculInterets();
System.out.println(c2.toString());
c2.setTaux(6);
c2.calculInterets();
System.out.println(c2);
```

```
// Test de ComptePayant
ComptePayant c3=new
    ComptePayant(5000);
System.out.println(c3);
c3.verser(6000);
c3.retirer(4000);
System.out.println(c3);
}
```

Polymorphisme

Polymorphisme

Le polymorphisme offre aux objets la possibilité d'appartenir à plusieurs catégories à la fois.

En effet, nous avons certainement tous appris à l'école qu'il était impossible d'additionner des pommes et des oranges
Mais, on peut écrire l'expression suivante:
3 pommes + 5 oranges = 8 fruits

Polymorphisme

Le sur-casting des objets:

Une façon de décrire l'exemple consistant à additionner des pommes et des oranges serait d'imaginer que nous disons pommes et oranges mais que nous manipulons en fait des fruits. Nous pourrions écrire alors la formule correcte :

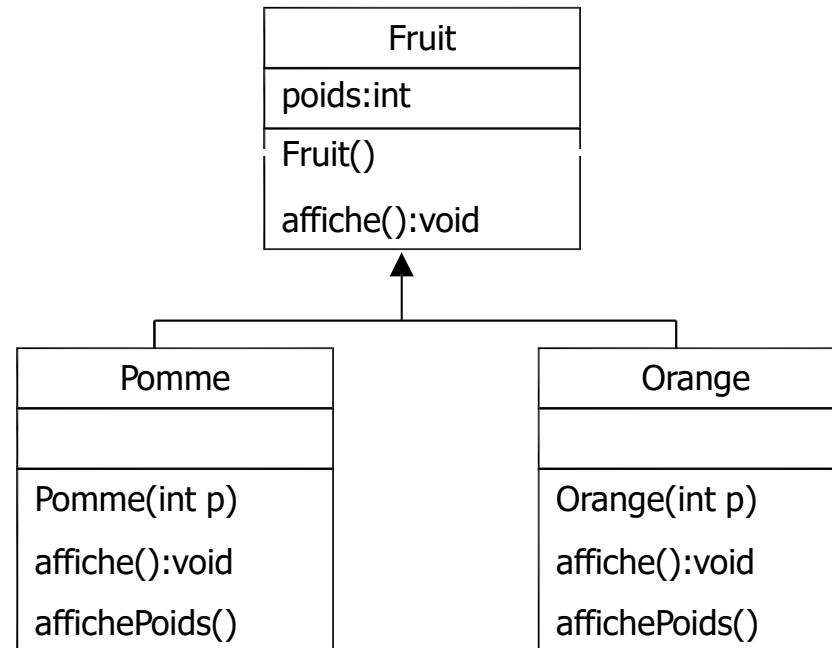
$$\begin{array}{r} 3 \text{ (fruits) pommes} \\ + 5 \text{ (fruits) oranges} \\ \hline \end{array}$$

$$= 8 \text{ fruits}$$

Cette façon de voir les choses implique que les pommes et les oranges soient "transformés" en fruits préalablement à l'établissement du problème. Cette transformation est appelée *sur-casting*

Instanciation et héritage

Considérons l'exemple suivant:



Instanciation et héritage

```
public abstract class Fruit{  
    int poids;  
    public Fruit(){  
        System.out.println("Création d'un  
fruit");  
    }  
    public void affiche(){  
        System.out.println("c'est un fruit");  
    }  
}
```

```
public class Pomme extends Fruit{  
    public Pomme(int p){  
        poids=p;  
        System.out.println("création d'une pomme  
de "+ poids+" grammes");  
    }  
    public void affiche(){  
        System.out.println("C'est une pomme");  
    }  
    public void affichePoids(){  
        System.out.println("le poids de la pomme  
est:"+poids+" grammes");  
    }  
}
```

```
public class Orange extends Fruit{  
    public Orange(int p){ poids=p;  
        System.out.println("création d'une  
Orange de "+ poids+" grammes");  
    }  
    public void affiche(){  
        System.out.println("C'est une  
Orange");  
    }  
    public void affichePoids(){  
  
        System.out.println("le poids de la  
Orange est:"+poids+" grammes");  
    }  
}
```

```
public class Polymorphisme{  
    public static void main(String[] args){  
        Pomme p=new Pomme(72);  
        Orange o=new Orange(80);  
    }  
}
```

Instanciation et héritage

Le résultat affiché par le programme est:

Création d'un fruit

Création d'une pomme de 72 grammes

Création d'un fruit

création d'une orange de 80 grammes

Nous constatons qu'avant de créer une Pomme, le programme crée un Fruit, comme le montre l'exécution du constructeur de cette classe. La même chose se passe lorsque nous créons une Orange

Sur-casting des objets

Considérons l'exemple suivant:

```
public class Polymorphisme2{  
    public static void main(String[] args){  
        // Sur-casting implicite  
        Fruit f1=new Orange(40);  
        // Sur-casting explicite  
        Fruit f2=(Fruit) new Pomme(60);  
        // Sur-casting implicite  
        f2=new Orange(40);  
    }  
}
```

Sur-casting des objets

Un objet de type Pomme peut être affecté à un handle de type fruit sans aucun problème :

```
Fruit f1;  
f1=new Pomme(60);
```

Dans ce cas l'objet Pomme est converti automatiquement en Fruit.

On dit que l'objet Pomme est sur-casté en Fruit.

Dans java, le sur-casting peut se faire implicitement.

Toutefois, on peut faire le sur-casting explicitement sans qu'il soit nécessaire.

La casting explicit se fait en précisant la classe vers laquelle on convertit l'objet entre parenthèse. Exemple :

```
f2=(Fruit)new Orange(40);
```

Sous-Casting des objets

Considérons l'exemple suivant:

```
public class Polymorphisme3{
    public static void main(String[] args) {
        Fruit f1;
        Fruit f2;
        f1=new Pomme(60);
        f2=new Orange(40);
f1.affichePoids(); _____→
        ((Pomme)f1).affichePoids();
    }
}
```

Erreur de compilation:

```
Polymorphisme3.java:5: cannot resolve symbol
  symbol  : method affichePoids ()
  location: class Fruit
  f1.affichePoids();
          ^
1 error
```

Solution : Sous-casting explicit

Sous-casting des objets

Ce message indique que l'objet f1 qui est de type Fruit ne possède pas la méthode affichePoids().

Cela est tout à fait vrai car cette méthode est définie dans les classes Pomme et Oranges et non dans la classe Fruit.

En fait, même si le handle f1 pointe un objet Pomme, le compilateur ne tient pas en considération cette affectation, et pour lui f1 est un Fruit.

Il faudra donc convertir explicitement l'objet f1 qui de type Fruit en Pomme.

Cette conversion s'appelle Sous-casting qui indique la conversion d'un objet d'une classe vers un autre objet d'une classe dérivée.

Dans ce cas de figure, le sous-casting doit se faire explicitement.

L'erreur de compilation peut être évité en écrivant la syntaxe suivante :

((Pomme)f1).affichePoids();

Cette instruction indique que l'objet f1 , de type Fruit, est converti en Pomme, ensuite la méthode affichePoids() de l'objet Pomme est appelé ce qui est correcte.

Late Binding

Dans la plupart des langages, lorsque le compilateur rencontre un appel de méthode, il doit être à même de savoir exactement de quelle méthode il s'agit.

Le lien entre l'appel et la méthode est alors établi au moment de la compilation. Cette technique est appelée *early binding*, que l'on pourrait traduire par *liaison précoce*.

Java utilise cette technique pour les appels de méthodes déclarées **final**.

Elle a l'avantage de permettre certaines optimisations.

En revanche, pour les méthodes qui ne sont pas **final**, Java utilise la technique du *late binding* (*liaison tardive*).

Dans ce cas, le compilateur n'établit le lien entre l'appel et la méthode qu'au moment de l'exécution du programme.

Ce lien est établi avec la version la plus spécifique de la méthode.

Dans notre cas, nous la méthode `affiche()` possède 3 versions définies dans les classes `Fruit`, `Pomme` et `Orange`.

Grâce au *late binding*, java est capable de déterminer, au moment de l'exécution, quelle version de méthode qui sera appelée ce que nous pouvons vérifier par le programme suivant :

Late Binding

```
public class Polymorphisme4{
    public static void main(String[] args) {
        Fruit f1;
        Fruit f2;
        f1=new Pomme(60); f2=new Orange(40);
        f1.affiche(); //((Pomme)f1).affiche();

        f2.affiche();
    }
}
```

L'exécution de ce programme donne :

Création d'un fruit

Création d'une pomme de 60 grammes

Création d'un fruit

Création d'une orange de 40 grammes

C'est une pomme

C'est une Orange

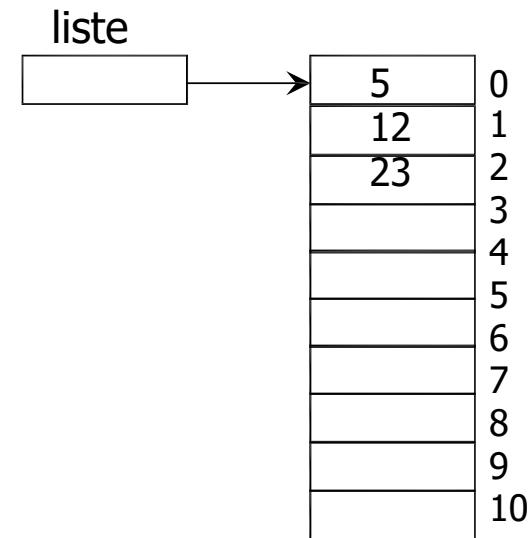
Polymorphisme

Pour résumer, un objet est une instance de :

- sa classe,
- toutes les classes parentes de sa classe,
- toutes les interfaces qu'il implémente,
- toutes les interfaces parentes des interfaces qu'il implémente,
- toutes les interfaces qu'implémentent les classes parentes de sa classe,
- toutes les interfaces parentes des précédentes.

Tableaux et Collections

Tableaux de primitives



Tableaux de primitives:

Déclaration :

Exemple : Tableau de nombres entiers

```
int[] liste;
```

liste est un handle destiné à pointer vers un tableau d'entier

Création du tableau

```
liste = new int[11];
```

Manipulation des éléments du tableau:

```
liste[0]=5; liste[1]=12; liste[3]=23;  
for(int i=0;i<liste.length;i++) {  
    System.out.println(liste[i]);  
}
```

Tableaux d'objets

Déclaration :

Exemple : Tableau d'objets Fruit

```
Fruit[] lesFruits;
```

Création du tableau

```
lesFruits = new Fruit[5];
```

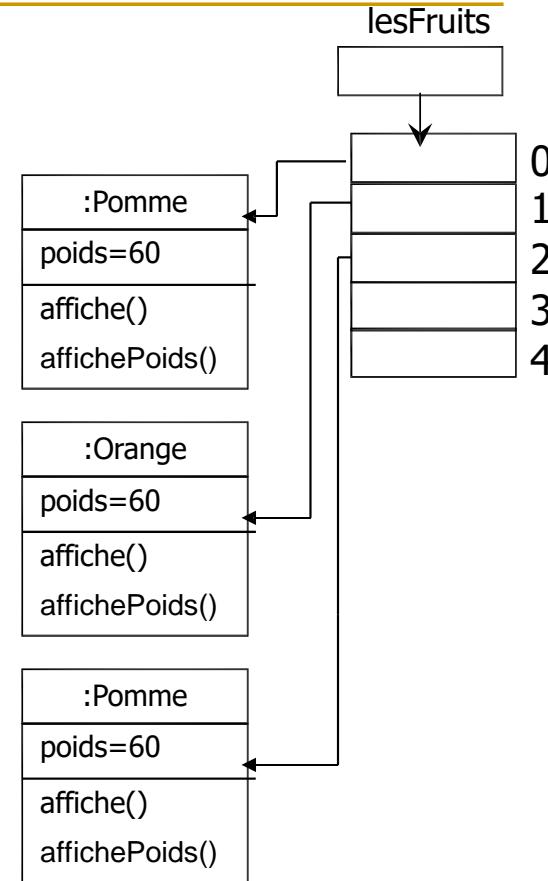
Création des objets:

```
lesFruits[0]=new Pomme(60);  
lesFruits[1]=new Orange(100);  
lesFruits[2]=new Pomme(55);
```

Manipulation des objets:

```
for(int i=0;i<lesFruits.length;i++) {  
    lesFruits[i].affiche();  
    if(lesFruits[i] instanceof Pomme)  
        ((Pomme)lesFruits[i]).affichePoids();  
    else  
        ((Orange)lesFruits[i]).affichePoids();  
}
```

Un tableau d'objets est un tableau de handles



Collections

Une collection est un tableau dynamique d'objets de type Object.

Une collection fournit un ensemble de méthodes qui permettent:

- D'ajouter un nouveau objet dans le tableau

- Supprimer un objet du tableau

- Rechercher des objets selon des critères

- Trier le tableau d'objets

- Contrôler les objets du tableau

- Etc...

Dans un problème, les tableaux peuvent être utilisés quand la dimension du tableau est fixe.

Dans le cas contraire, il faut utiliser les collections

Java fournit plusieurs types de collections:

- ArrayList

- Vector

- Iterator

- HashMap

- Etc...

Dans cette partie du cours, nous allons présenter uniquement comment utiliser les collections ArrayList, Vector, Iterator et HashMap

~~Vous aurez l'occasion de découvrir les autres collections dans les prochains cours~~

Collection ArrayList

ArrayList est une classe du package java.util, qui implémente l'interface List.
Déclaration d'une collection de type List qui devrait stocker des objets de type Fruit:

```
List<Fruit> fruits;
```

Création de la liste:

```
fruits=new ArrayList<Fruit>();
```

Ajouter deux objets de type Fruit à la liste:

```
fruits.add(new Pomme(30));
```

```
fruits.add(new Orange(25));
```

Faire appel à la méthode affiche() de tous les objets de la liste:

En utilisant la boucle classique for

```
for(int i=0;i<fruits.size();i++) {  
    fruits.get(i).affiche();  
}
```

En utilisant la boucle for each

```
for(Fruit f:fruits)  
    f.affiche();
```

Supprimer le deuxième Objet de la liste

```
fruits.remove(1);
```

Exemple d'utilisation de ArrayList

```
import java.util.ArrayList;import java.util.List;
public class Appl {
    public static void main(String[] args) {
        // Déclaration d'une liste de type Fruit
        List<Fruit> fruits;
        // Création de la liste
        fruits=new ArrayList<Fruit>();
        // Ajout de 3 objets Pomme, Orange et Pomme à la liste
        fruits.add(new Pomme(30));
        fruits.add(new Orange(25));
        fruits.add(new Pomme(60));
        // Parcourir tous les objets
        for(int i=0;i<fruits.size();i++) {
            // Faire appel à la méthode affiche() de chaque Fruit de la
            // liste
            fruits.get(i).affiche();
        }
        // Une autre manière plus simple pour parcourir une liste
        for(Fruit f:fruits) // Pour chaque Fruit de la liste
            f.affiche(); // Faire appel à la méthode affiche() du Fruit f
    }
}
```

Collection Vector

Vecor est une classe du package java.util qui fonctionne comme ArrayList

Déclaration d'un Vecteur qui devrait stocker des objets de type Fruit:

```
Vector<Fruit> fruits;
```

Création de la liste:

```
fruits=new Vector<Fruit>();
```

Ajouter deux objets de type Fruit à la liste:

```
fruits.add(new Pomme(30));
```

```
fruits.add(new Orange(25));
```

Faire appel à la méthode affiche() de tous les objets de la liste:

En utilisant la boucle classique for

```
for(int i=0;i<fruits.size();i++) {  
    fruits.get(i).affiche();  
}
```

En utilisant la boucle for each

```
for(Fruit f:fruits)  
    f.affiche();
```

Supprimer le deuxième Objet de la liste

```
fruits.remove(1);
```

Exemple d'utilisation de Vector

```
import java.util.Vector;
public class App2 {
    public static void main(String[] args) {
        // Déclaration d'un vecteur de type Fruit
        Vector<Fruit> fruits;
        // Création du vecteur
        fruits=new Vector<Fruit>();
        // Ajout de 3 objets Pomme, Orange et Pomme au vecteur
        fruits.add(new Pomme(30));
        fruits.add(new Orange(25));
        fruits.add(new Pomme(60));
        // Parcourir tous les objets
        for(int i=0;i<fruits.size();i++){
            // Faire appel à la méthode affiche() de chaque Fruit
            fruits.get(i).affiche();
        }
        // Une autre manière plus simple pour parcourir un vecteur
        for(Fruit f:fruits) // Pour chaque Fruit du vecteur
            f.affiche(); // Faire appel à la méthode affiche() du Fruit f
    }
}
```

Collection de type Iterator

La collection de type Iterator du package java.util est souvent utilisée pour afficher les objets d'une autre collection

En effet il est possible d'obtenir un iterator à partir de chaque collection.

Exemple :

Création d'un vecteur de Fruit.

```
Vector<Fruit> fruits=new Vector<Fruit>();
```

Ajouter des fruits aux vecteur

```
fruits.add(new Pomme(30));  
fruits.add(new Orange(25));  
fruits.add(new Pomme(60));
```

Création d'un Iterator à partir de ce vecteur

```
Iterator<Fruit> it=fruits.iterator();
```

Parcourir l'Iterator:

```
while(it.hasNext()) {  
    Fruit f=it.next();  
    f.affiche();  
}
```

Notez bien que, après avoir parcouru un iterator, il devient vide

Collection de type HashMap

La collection `HashMap` est une classe qui implémente l'interface `Map`. Cette collection permet de créer un tableau dynamique d'objet de type `Object` qui sont identifiés par une clé.

Déclaration et création d'une collection de type `HashMap` qui contient des fruits identifiés par une clé de type `String` :

```
Map<String, Fruit> fruits=new HashMap<String, Fruit>();
```

Ajouter deux objets de type `Fruit` à la collection

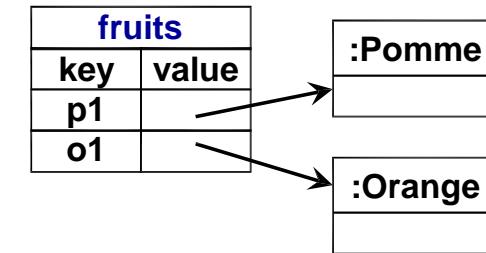
```
fruits.put("p1", new Pomme(40));  
fruits.put("o1", new Orange(60));
```

Récupérer un objet ayant pour clé "p1"

```
Fruit f=fruits.get("p1");  
f.affiche();
```

Parcourir toute la collection:

```
Iterator<String> it=fruits.keySet().iterator();  
while(it.hasNext()) { String  
key=it.next(); Fruit  
ff=fruits.get(key);  
System.out.println(key);  
ff.affiche();  
}
```

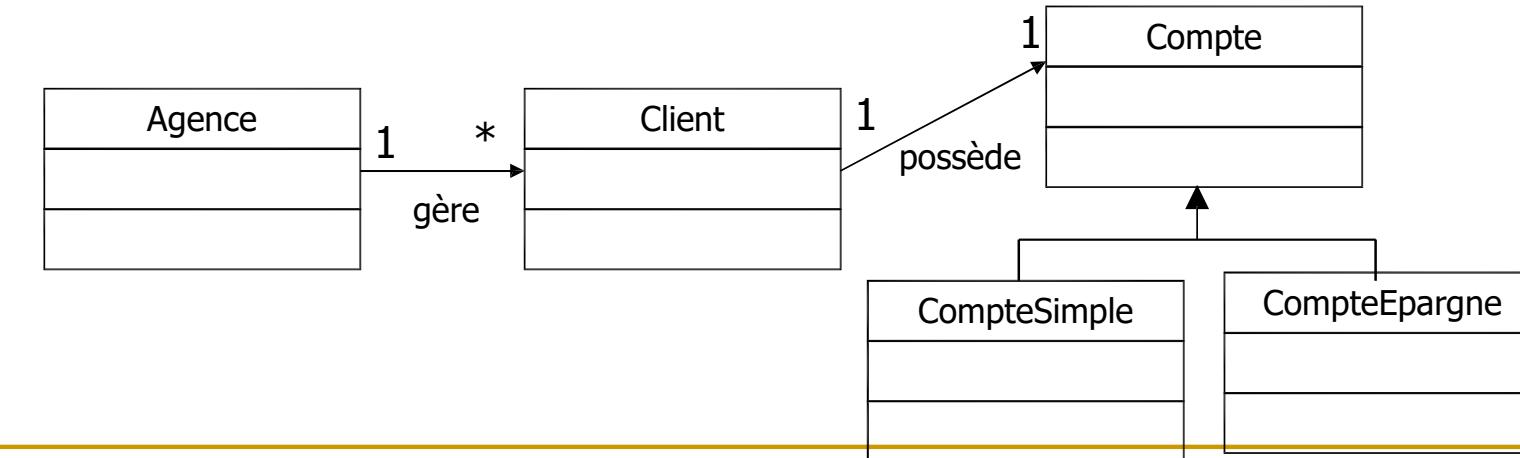


Associations entre classes

Une agence gère plusieurs clients.

Chaque client possède un seul compte

Le compte peut être soit un compteSimple ou un compteEpargne.



Traduction des association

La première association gère de type (1---*) entre Agence et Client se traduit par :

Création d'un tableau d'objets Client ou d'un vecteur comme attribut de la classe Agence.

Ce tableau ou ce vecteur sert à référencer les clients de l'agence.

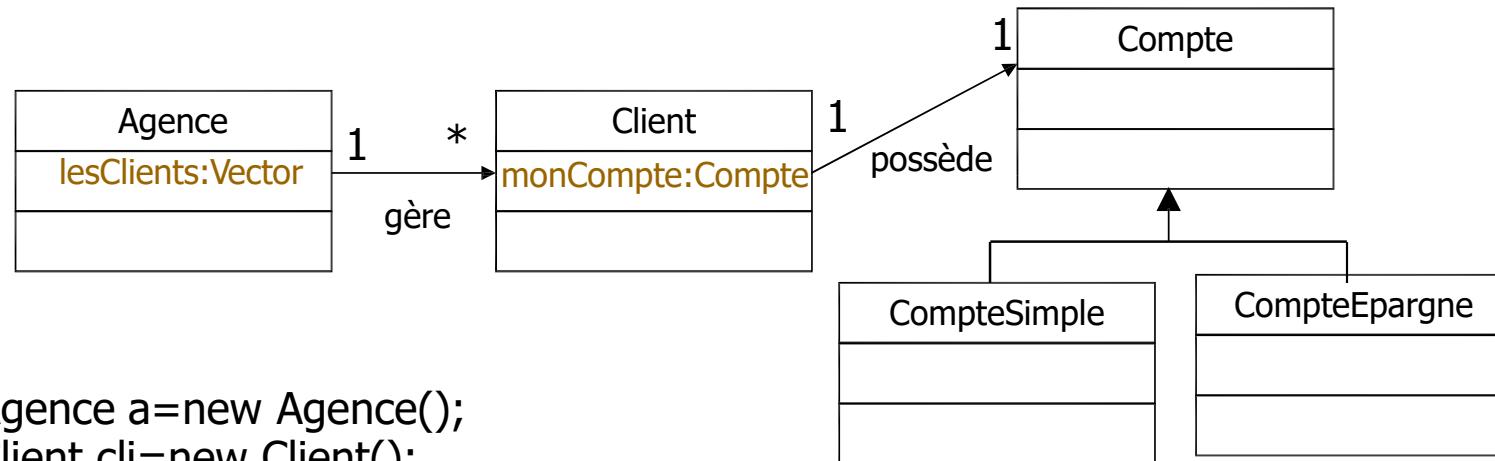
La deuxième association possède de type (1---1) se traduit par:

Création d'un handle de type Compte dans la classe Client.

Ce handle sert à référencer l'objet Compte de ce client.

La troisième règle de gestion est une relation d'héritage et non une association.

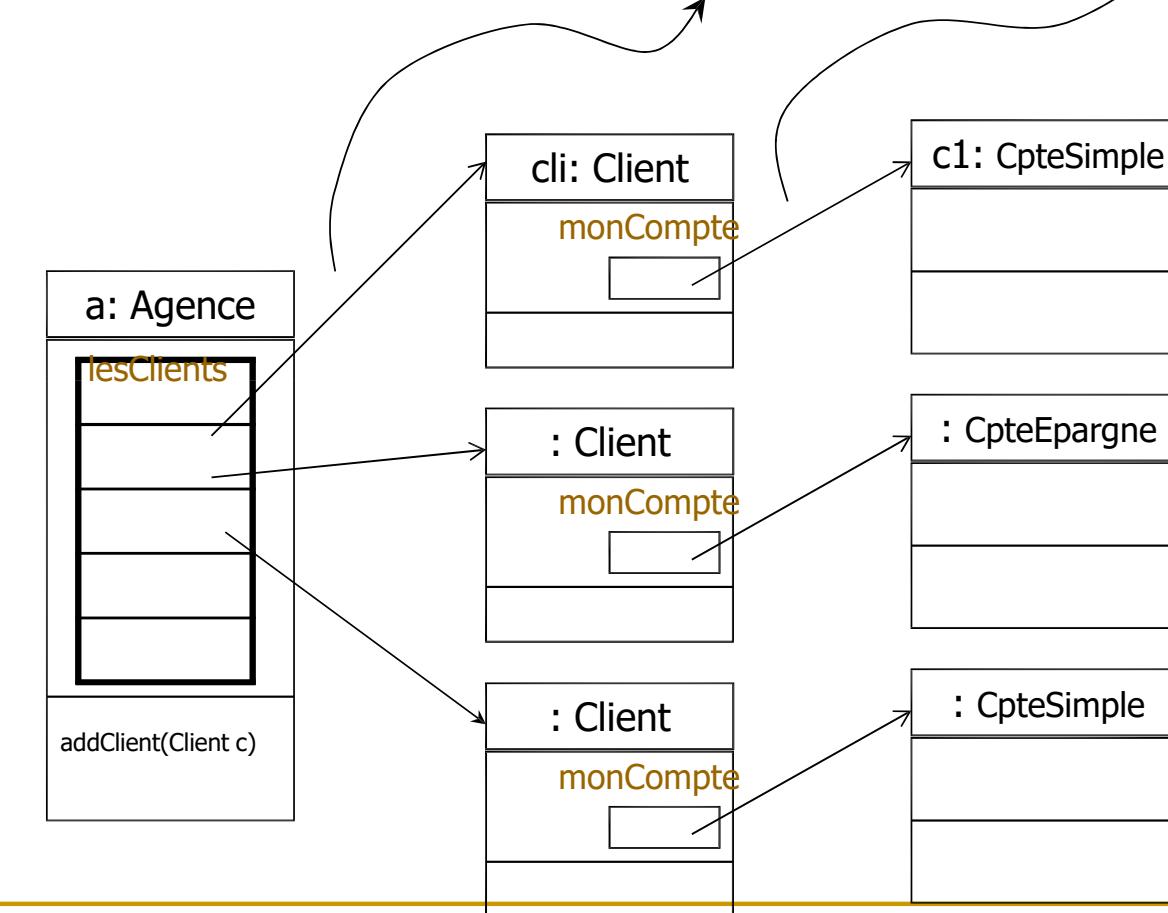
Traduction des association



```
Agence a=new Agence();
Client cli=new Client();
CompteSimple c1=new CompteSimple();
cli.monCompte=c1;
a.addClient(cli);
```

Exemple de diagramme d'objets

```
Agence a=new Agence();
Client cli=new Client();
CompteSimple c1=new CompteSimple();
cli.monCompte=c1; ←
a.lesClients.addElement(cli);
```



Exceptions et Entrées sorties

Exceptions

Souvent, un programme doit traiter des situations exceptionnelles qui n'ont pas un rapport direct avec sa tâche principale.

Ceci oblige le programmeur à réaliser de nombreux tests avant d'écrire les instructions utiles du programme. Cette situation a deux inconvénients majeurs :

Le programmeur peut omettre de tester une condition ;

Le code devient vite illisible car la partie utile est masquée par les tests.

Java remédie à cela en introduisant un *Mécanisme de gestion des exceptions*.

Grâce à ce mécanisme, on peut améliorer grandement la lisibilité du code

En séparant

le code utile (Code métier)

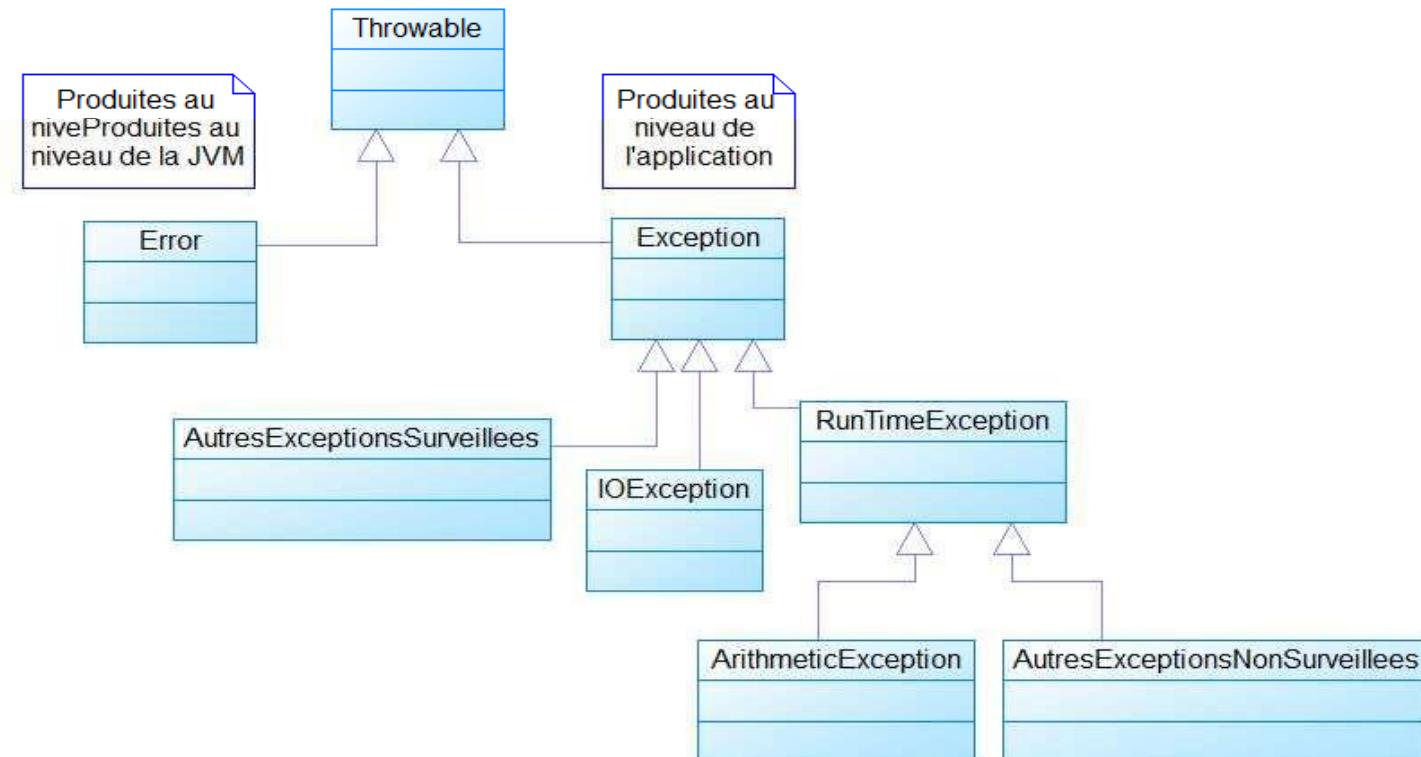
de celui qui traite des situations exceptionnelles,

Hiérarchie des exceptions

Java peut générer deux types d'erreurs au moment de l'exécution:

Des erreurs produites par l'application dans des cas exceptionnels que le programmeur devrait prévoir et traiter dans son application. Ce genre d'erreur sont de type Exception

Des erreurs qui peuvent être générées au niveau de la JVM et que le programmeur ne peut prévoir dans son application. Ce type d'erreurs sont de type Error.



Les Exceptions

En Java, on peut classer les exceptions en deux catégories :

Les exceptions surveillées,

Les exceptions non surveillées.

Java oblige le programmeur à traiter les erreurs surveillées. Elles sont signalées par le compilateur

Les erreurs non surveillées peuvent être traitées ou non. Et ne sont pas signalées par le compilateur

Un premier exemple:

Considérons une application qui permet de :

Saisir au clavier deux entiers a et b

Faire appel à une fonction qui permet de calculer et de retourner
a divisé par b.

Affiche le résultat

```
import java.util.Scanner;
public class App1 {
    public static int calcul(int a,int b){
        int c=a/b;
        return c;
    }
    public static void main(String[] args) { Scanner
        clavier=new Scanner(System.in);
        System.out.print("Donner a:");int a=clavier.nextInt();
        System.out.print("Donner b:");int b=clavier.nextInt();
        int resultat=calcul(a, b);
        System.out.println("Resultat="+resultat);
    }
}
```

Exécution

Nous constatons que le compilateur ne signale pas le cas où b est égal à zero.

Ce qui constitue un cas fatal pour l'application.

Voyons ce qui se passe au moment de l'exécution

Scénario 1 : Le cas normal

```
Donner a:12  
Donner b:6  
Resultat=2
```

Scénario 2: cas où b=0

```
Donner a:12  
Donner b:0  
Exception in thread "main" java.lang.ArithmetricException: / by zero  
at App1.calcul(App1.java:4)  
at App1.main(App1.java:11)
```

Un bug dans l'application

Le cas du scénario 2 indique que qu'une erreur fatale s'est produite dans l'application au moment de l'exécution.

Cette exception est de type `ArithmeticException`. Elle concerne une division par zero

L'origine de cette exception étant la méthode `calcul` dans la ligne numéro 4.

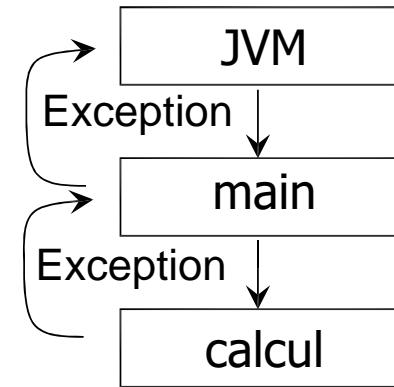
Cette exception n'a pas été traité dans `calcul`.

Elle remonte ensuite vers `main` à la ligne numéro 11 dont elle n'a pas été traitée.

Après l'exception est signalée à la JVM.

Quand une exception arrive à la JVM, cette dernière arrête l'exécution de l'application, ce qui constitue un bug fatale.

Le fait que le message « `Resultat=` » n'a pas été affiché, montre que l'application ne continue pas son exécution normal après la division par zero



Traiter l'exception

Dans java, pour traiter les exceptions, on doit utiliser le bloc try catch de la manière suivante:

```
import java.util.Scanner;
public class App1 {
    public static int calcul(int a,int b){
        int c=a/b;
        return c;
    }
    public static void main(String[] args) {
        Scanner clavier=new Scanner(System.in);
        System.out.print("Donner a:");int a=clavier.nextInt();
        System.out.print("Donner b:");int b=clavier.nextInt();
        int resultat=0;
        try{
            resultat=calcul(a, b);
        }
        catch (ArithmException e) {
            System.out.println("Divisio par zero");
        }
        System.out.println("Resultat="+resultat);
    }
}
```

Scénario 1

```
Donner a:12
Donner b:6
Resultat=2
```

Scénario 2

```
Donner a:12
Donner b:0
Divisio par zero
Resultat=0
```

Principale Méthodes d'une Exception

Tous les types d'exceptions possèdent les méthodes suivantes :

getMessage() : retourne le message de l'exception

```
System.out.println(e.getMessage());
```

Réstat affiché : / by zero

toString() : retourne une chaîne qui contient le type de l'exception et le message de l'exception.

```
System.out.println(e.toString());
```

Réstat affiché : java.lang.ArithmetricException: / by zero

printStackTrace: affiche la trace de l'exception

```
e.printStackTrace();
```

Réstat affiché :

java.lang.ArithmetricException: / by zero

at App1.calcul(App1.java:4)

at App1.main(App1.java:13)

Exemple : Générer, Relancer ou Jeter une exception surveillée de type Exception

Considérons le cas d'un compte qui est défini par un code et un solde et sur lequel, on peut verser un montant, retirer un montant et consulter le solde.

```
package metier;
public class Compte {
    private int code;
    private float solde;
    public void verser(float mt){
        solde=solde+mt;
    }
    public void retirer(float mt) throws Exception{
        if(mt>solde) throw new Exception("Solde Insuffisant");
        solde=solde-mt;
    }
    public float getSolde(){
        return solde;
    }
}
```

Utilisation de la classe Compte

En faisant appel à la méthode retirer, le compilateur signale que cette dernière peut générer une exception de type Exception

C'est une Exception surveillée. Elle doit être traitée par le programmeur

```
package pres;
import java.util.Scanner;
import metier.Compte;
public class Application {
public static void main(String[] args) {
    Compte cp=new Compte();
    Scanner clavier=new Scanner(System.in);
    System.out.print("Montant à verser:");
    float mt1=clavier.nextFloat();
    cp.verser(mt1);
    System.out.println("Solde Actuel:"+cp.getSolde());
    System.out.print("Montant à retirer:");
    float mt2=clavier.nextFloat();
    cp.retirer(mt2); // Le compilateur signe l'Exception
}
}
```

Traiter l'exception

Deux solutions :

Soit utiliser le bloc try catch

```
try {  
    cp.retirer(mt2);  
} catch (Exception e) {  
    System.out.println(e.getMessage());  
}
```

Ou déclarer que cette exception est ignorée dans la méthode main et dans ce cas là, elle remonte vers le niveau supérieur. Dans notre cas la JVM.

```
public static void main(String[] args) throws Exception {
```

Exécution de l'exemple

```
package pres;
import java.util.Scanner;
import metier.Compte;
public class Application {
public static void main(String[] args) {
    Compte cp=new Compte();
    Scanner clavier=new Scanner(System.in);
    System.out.print("Montant à verser:");
    float mt1=clavier.nextFloat();
    cp.verser(mt1);
    System.out.println("Solde Actuel:"+cp.getSolde());
    System.out.print("Montant à retirer:");
    float mt2=clavier.nextFloat();
    try {
        cp.retirer(mt2);
    } catch (Exception e) {
        System.out.println(e.getMessage());
    }
    System.out.println("Solde Final="+cp.getSolde());
}
}
```

Scénario 1

Montant à verser:5000
Solde Actuel:5000.0
Montant à retirer:2000
Solde Final=3000.0

Scénario 2

Montant à verser:5000
Solde Actuel:5000.0
Montant à retirer:7000
Solde Insuffisant
Solde Final=5000.0

Exemple : Générer une exception non surveillée de type RuntimeException

```
package metier;
public class Compte {
    private int code;
    private float solde;
    public void verser(float mt){
        solde=solde+mt;
    }
    public void retirer(float mt){
        if(mt>solde) throw new RuntimeException("Solde Insuffisant");
        solde=solde-mt;
    }
    public float getSolde(){
        return solde;
    }
}
```

Utilisation de la classe Compte

En faisant appel à la méthode retirer, le compilateur ne signale rien

C'est une Exception non surveillée. On est pas obligé de la traiter pour que le programme soit compilé

```
package pres;
import java.util.Scanner;
import metier.Compte;
public class Application {
public static void main(String[] args) {
    Compte cp=new Compte();
    Scanner clavier=new Scanner(System.in);
    System.out.print("Montant à verser:");
    float mt1=clavier.nextFloat();
    cp.verser(mt1);
    System.out.println("Solde Actuel:"+cp.getSolde());
    System.out.print("Montant à retirer:");
    float mt2=clavier.nextFloat();
    cp.retirer(mt2); // Le compilateur ne signale rien
}}
```

Personnaliser les exception métier.

L'exception générée dans la méthode retirer, dans le cas où le solde est insuffisant est une exception métier.

Il est plus professionnel de créer une nouvelle Exception nommée SoldeInsuffisantException de la manière suivante :

```
package metier;  
public class SoldeInsuffisantException extends Exception {  
    public SoldeInsuffisantException(String message) {  
        super(message);  
    }  
}
```

En héritant de la classe Exception, nous créons une exception surveillée.

Pour créer une exception non surveillée, vous pouvez hériter de la classe RuntimeException

Utiliser cette nouvelle exception métier

```
package metier;
public class Compte {
    private int code;
    private float solde;
    public void verser(float mt){
        solde=solde+mt;
    }
    public void retirer(float mt) throws SoldeInsuffisantException{
        if(mt>solde) throw new SoldeInsuffisantException("Solde Insuffisant");
        solde=solde-mt;
    }
    public float getSolde(){
        return solde;
    }
}
```

Application

```
package pres;
import java.util.Scanner;
import metier.Compte;
import metier.SoldeInsuffisantException;
public class Application {
    public static void main(String[] args) {
        Compte cp=new Compte();
        Scanner clavier=new Scanner(System.in);
        System.out.print("Montant à verser:");
        float mt1=clavier.nextFloat();
        cp.verser(mt1);
        System.out.println("Solde Actuel:"+cp.getSolde());
        System.out.print("Montant à retirer:");
        float mt2=clavier.nextFloat();
        try {
            cp.retirer(mt2);
        } catch (SoldeInsuffisantException e) {
            System.out.println(e.getMessage());
        }
        System.out.println("Solde Final="+cp.getSolde());
    }
}
```

Scénario 1

```
Montant à verser:5000
Solde Actuel:5000.0
Montant à retirer:2000
Solde Final=3000.0
```

Scénario 2

```
Montant à verser:5000
Solde Actuel:5000.0
Montant à retirer:7000
Solde Insuffisant
Solde Final=5000.0
```

Scénario 3

```
Montant à verser:azerty
Exception in thread "main"
java.util.InputMismatchException
at java.util.Scanner.throwFor(Scanner.java:840)
at java.util.Scanner.next(Scanner.java:1461) at
java.util.Scanner.nextFloat(Scanner.java:2319)
at pres.Application.main(Application.java:9)
```

Améliorer l'application

Dans le scénario 3, nous découvrons qu'une autre exception non surveillée est générée dans le cas où on saisie une chaîne de caractères et non pas un nombre.

Cette exception est de type `InputMismatchException`, générée par la méthode `nextFloat()` de la classe `Scanner`.

Nous devrions donc faire plusieurs `catch` dans la méthode `main`

Application

```
package pres;  
import java.util.InputMismatchException;import java.util.Scanner;  
import metier.Compte;import metier.SoldeInsuffisantException;  
public class Application {  
    public static void main(String[] args) {  
        Compte cp=new Compte();  
        try {  
            Scanner clavier=new Scanner(System.in);  
            System.out.print("Montant à verser:");  
            float mt1=clavier.nextFloat();  
            cp.verser(mt1);  
            System.out.println("Solde Actuel:"+cp.getSolde());  
            System.out.print("Montant à retirer:");  
            float mt2=clavier.nextFloat();  
            cp.retirer(mt2);  
        }  
        catch (SoldeInsuffisantException e) {  
            System.out.println(e.getMessage());  
        }  
        catch (InputMismatchException e) {  
            System.out.println("Problème de saisie");  
        }  
        System.out.println("Solde  
    }
```

Un autre cas exceptionnel dans la méthode retirer

Supposons que l'on ne doit pas accepter un montant négatif dans la méthode retirer.

On devrait générer une exception de type MontantNegatifException.

Il faut d'abord créer cette nouvelle exception métier.

Le cas MontantNegatifException

- L'exception métier MontantNegatifException

```
package metier;  
public class MontantNegatifException extends Exception {  
    public MontantNegatifException(String message) {  
        super(message);  
    }  
}
```

- Le méthode retirer de la classe Compte

```
public void retirer(float mt) throws  
SoldeInsuffisantException, MontantNegatifException{  
    if(mt<0) throw new MontantNegatifException("Montant "+mt+"  
négatif");  
    if(mt>solde) throw new SoldeInsuffisantException("Solde  
Insuffisant");  
    solde=solde-mt;  
}
```

Application : Contenu de la méthode main

```
Compte cp=new Compte();
try {
    Scanner clavier=new Scanner(System.in);
    System.out.print("Montant à verser:");
    float mt1=clavier.nextFloat();
    cp.verser(mt1);
    System.out.println("Solde Actuel:"+cp.getSolde());
    System.out.print("Montant à retirer:");
    float mt2=clavier.nextFloat();
    cp.retirer(mt2);
}
catch (SoldeInsuffisanteException e) {
    System.out.println(e.getMessage());
}
catch (InputMismatchException e) {
    System.out.println("Problème de saisie");
}
catch (MontantNegatifException e) {
    System.out.println(e.getMessage());
}
System.out.println("Solde
```

Scénario 1

Montant à verser:5000
Solde Actuel:5000.0
Montant à retirer:2000
Solde Final=3000.0

Scénario 2

Montant à verser:5000
Solde Actuel:5000.0
Montant à retirer:7000
Solde Insuffisant
Solde Final=5000.0

Scénario 3

Montant à verser:5000
Solde Actuel:5000.0
Montant à retirer:-2000
Montant -2000.0 négatif
Solde Final=5000.0

Scénario 4

Montant à verser:azerty
Problème de saisie
Solde Final=0.0

Le bloc finally

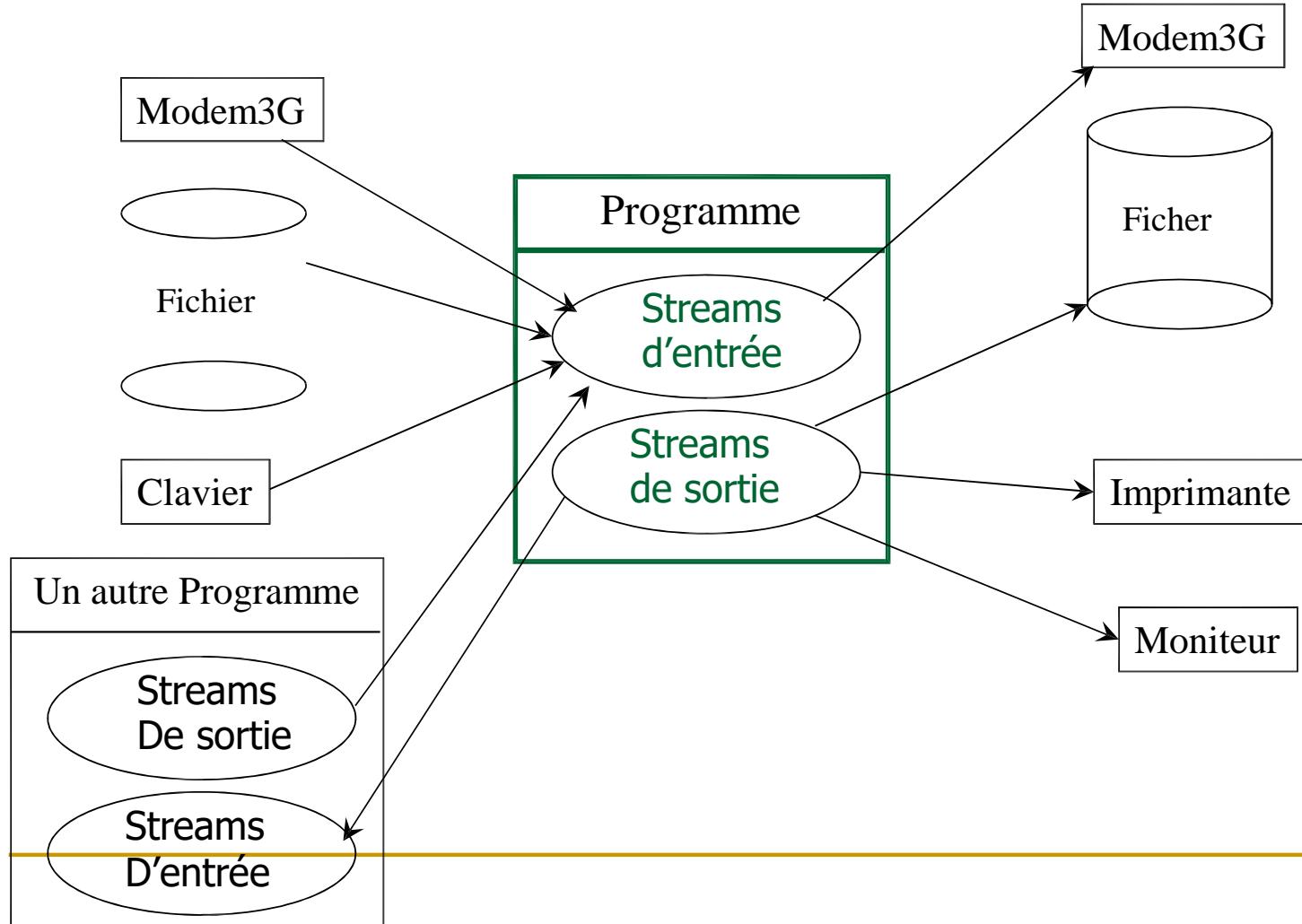
La syntaxe complète du bloc try est la suivante :

```
try {
    System.out.println("Traitement Normale");
}
catch (SoldeInsuffisantException e) {
    System.out.println("Premier cas Exceptionnel");
}
catch (NegativeArraySizeException e) {
    System.out.println("dexuième cas Exceptionnel");
}
finally{
    System.out.println("Traitement par défaut!");
}
System.out.println("Suite du programme!");
```

Le bloc **finally** s'exécute quelque soit les différents scénarios.

Entrés Sorties

Entrées Sorties



Principe des entrées/sorties

Pour effectuer une entrée ou une sortie de données en Java, le principe est simple et se résume aux opérations suivantes :

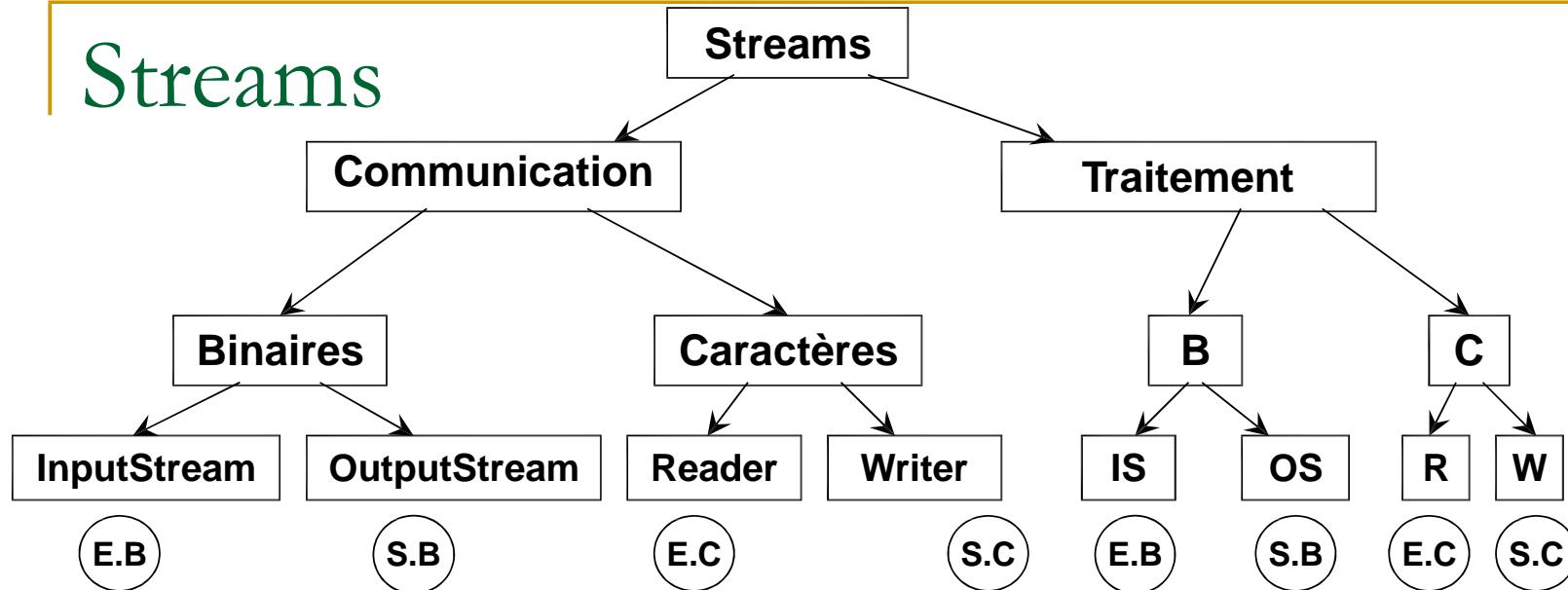
Ouverture d'un moyen de communication.

Écriture ou lecture des données.

Fermeture du moyen de communication.

En Java, les moyens de communication sont représentés par des objets particuliers appelés (en anglais) **stream**. Ce mot, qui signifie **courant**, ou **flux**

Streams



Il existe deux types de streams:

Streams de Communication: établit une liaison entre le programme et une destination.

Streams binaires : Exemple **FileInputStream**, **FileOutputStream**

Streams de caractères : Exemple **FileReader**, **FileWriter**

Streams de Traitement : Permet de traiter les information des streams de communication.

Streams binaires : ex **BufferedInputStream**, **ZipInputStream**, **ZipOutputStream**,...

Streams de caractères : **BufferedReader**, **BufferedWriter**, ...

La classe File

La classe File permet de donner des informations sur un fichier ou un répertoire

La création d'un objet de la classe File peut se faire de différentes manières :

```
File f1=new File("c:/projet/fichier.ext");
File f2=new File("c:/projet", "fichier.ext");
File f3=new File("c:/projet");
```

Principales méthodes de la classe File

<code>String getName();</code>	Retourne le nom du fichier.
<code>String getPath();</code>	Retourne la localisation du fichier en relatif.
<code>String getAbsolutePath();</code>	Idem mais en absolu.
<code>String getParent();</code>	Retourne le nom du répertoire parent.
<code>boolean renameTo(File newFile);</code>	Permet de renommer un fichier.
<code>boolean exists();</code>	Est-ce que le fichier existe ?
<code>boolean canRead();</code>	Le fichier est t-il lisible ?
<code>boolean canWrite();</code>	Le fichier est t-il modifiable ?
<code>boolean isDirectory();</code>	Permet de savoir si c'est un répertoire.
<code>boolean isFile();</code>	Permet de savoir si c'est un fichier.
<code>long length();</code>	Quelle est sa longueur (en octets) ?
<code>boolean delete();</code>	Permet d'effacer le fichier.
<code>boolean mkdir();</code>	Permet de créer un répertoire.
<code>String[] list();</code>	On demande la liste des fichiers localisés dans le répertoire.

Premier Exemple d'utilisation de la classe File

Afficher le contenu d'un répertoire en affichant si les éléments de ce répertoire sont des fichier ou des répertoires.

Dans le cas où il s'agit d'un fichier afficher la capacité physique du fichier.

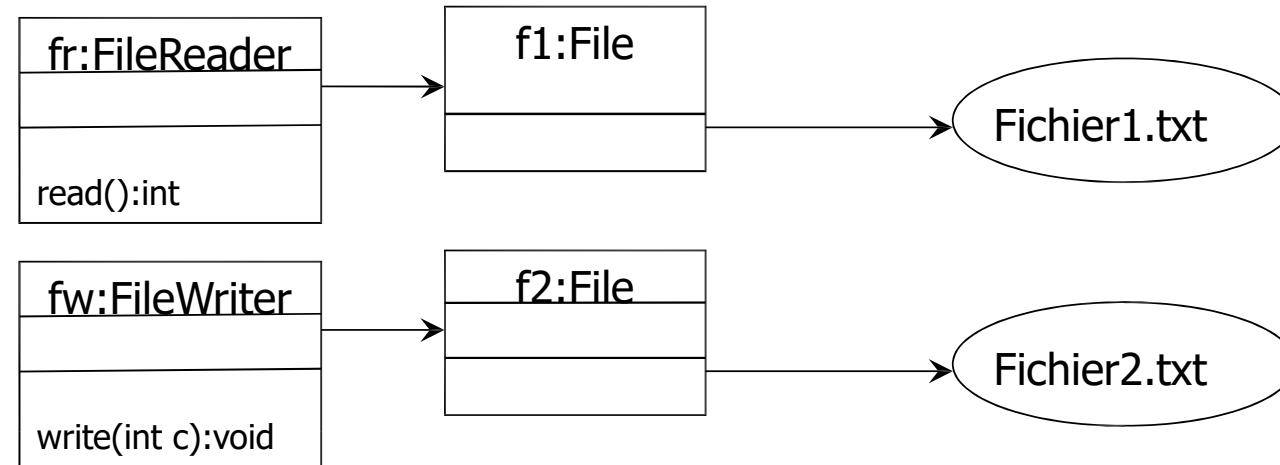
```
import java.io.File;
public class Application1 {
public static void main(String[] args) {
String rep="c:/"; File f=new File(rep);
if(f.exists()){
String[] contenu=f.list();
for(int i=0;i<contenu.length;i++){
File f2=new File(rep,contenu[i]);
if(f2.isDirectory())
System.out.println("REP:"+contenu[i]);
else
System.out.println("Fichier :"+contenu[i]+"
Size:"+contenu[i].length());
}
}
else{
System.out.println(rep+" n'existe pas");
}
}}
```

```
Fichier :aff2 (2).asm Size:12
Fichier :aff2.asm Size:8
REP:And
REP:androideProjets
REP:AP
REP:APP
Fichier :aqua_bitmap.cpp
Size:15
Fichier :autoexec.bat Size:12
```

Exercice 1

Ecrire une application java qui permet d'afficher le contenu d'un répertoire y compris le contenu de ses sous répertoires

Lire et Écrire sur un fichier texte



```
File f1=new File("c:/Fichier1.txt");
FileReader fr=new FileReader(f1);
File f2=new File("c:/Fichier2.txt");
FileWriter fw=new FileWriter(f2);
int c;
while((c=fr.read())!=-1){
    fw.write(c);
}
fr.close();fw.close();
```

Exemple 1 : Crypter un fichier Texte

Considérons un fichier texte nommé « operations.txt » qui contient les opérations sur un compte.

Chaque ligne de ce fichier représente une opération qui est définie par :

- le numéro de l'opération,
- le numéro de compte,
- la date d'opération,
- le type d'opération (Versement ou Retrait)
- et le montant de l'opération.

Voici un exemple de fichier :

```
321;CC1;2011-01-11;V;4500
512;CC1;2011-01-11;V;26000
623;CC1;2011-01-11;R;9000
815;CC1;2011-01-11;R;2500
```

Exemple 1 : Crypter un fichier Texte

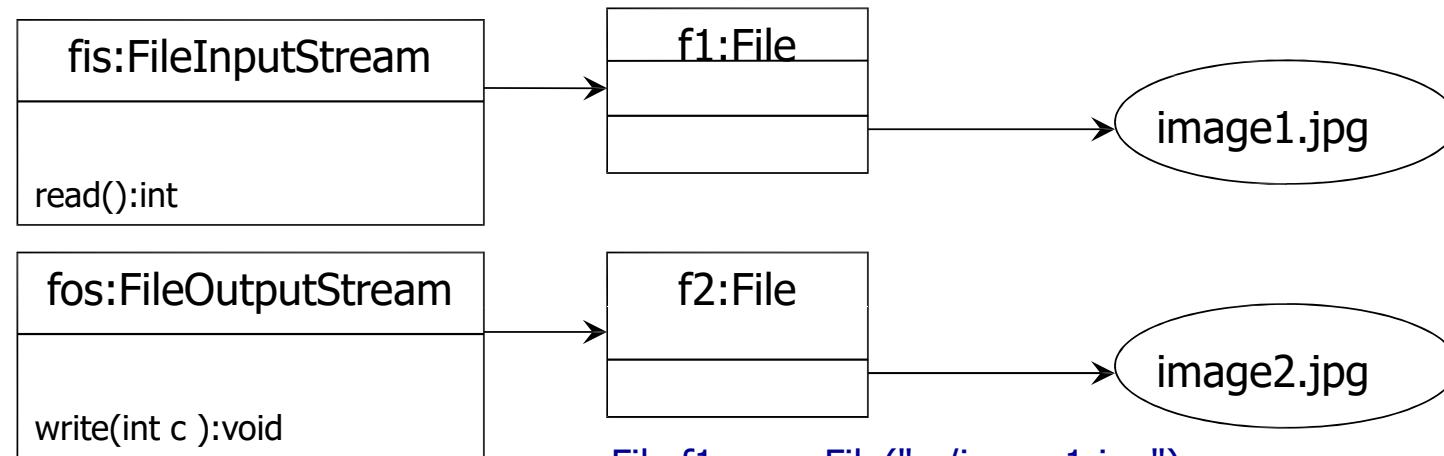
```
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
public class App2 {
    public static void main(String[] args) throws Exception {
        File f1=new File("operations.txt");
        FileReader fr=new FileReader(f1);
        File f2=new File("operationsCryptes.txt");          operations.txt
        FileWriter fw=new FileWriter(f2);
        int c;
        while((c=fr.read())!=-1){
            fw.write(c+1);
        }
        fr.close();
        fw.close();
    }
}
```

321;CC1;2011-01-11;V;4500
512;CC1;2011-01-11;V;26000
623;CC1;2011-01-11;R;9000
815;CC1;2011-01-11;R;2500

operationsCryptes.txt

432<DD2<3122.12.22<W<5611623<DD2<3122.12.22<W<37111734<DD2<
3122.12.22<S:<111926<DD2<3122.12.22<S<3611

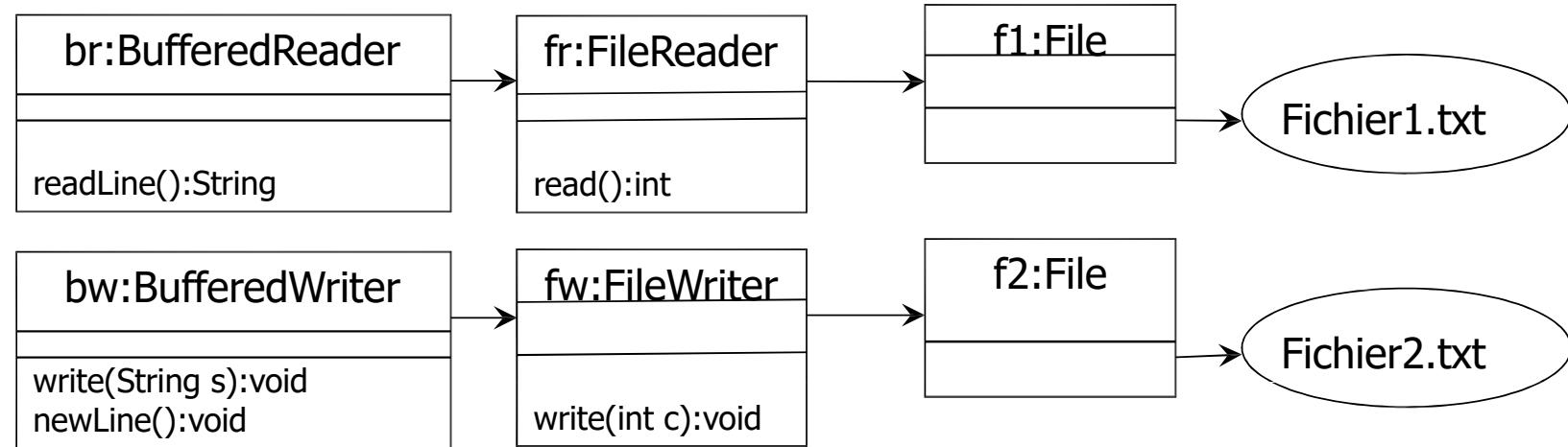
Lire et Écrire sur un fichier binaire



```
File f1=new File("c:/image1.jpg");
FileInputStream fis=new FileInputStream(f1); File
f2=new File("c:/image2.jpg"); FileOutputStream
fos=new FileOutputStream(f2); int c;
while((c=fis.read())!=-1){
    fos.write(c);
}

fis.close();fos.close();
```

Lire et Écrire sur un fichier texte ligne par ligne : Streams de traitement : BufferedReader et BufferedWriter

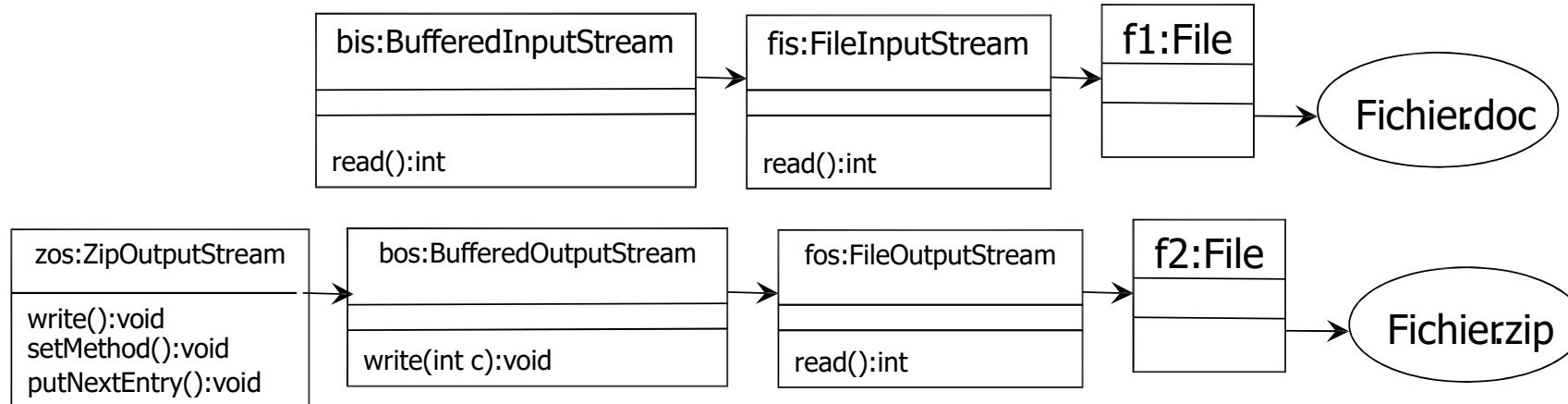


```
File f1=new File("c:/Fichier1.txt");
FileReader fr=new FileReader(f1);
BufferedReader br=new BufferedReader(fr);
File f2=new File("c:/Fichier2.txt");
FileWriter fw=new FileWriter(f2);
BufferedWriter bw=new BufferedWriter(fw);
String s;
while((s=br.readLine())!=null){
    bw.write(s);bw.newLine();
}
br.close();bw.close();
```

Exemple 3 : Afficher le total des opérations de versements et de retraits

```
import java.io.*;
public class App3 {
    public static void main(String[] args) throws Exception {
        File f=new File("operations.txt");
        FileReader fr=new FileReader(f);
        BufferedReader br=new BufferedReader(fr);
        String op;double totalVersements=0;double totalRetraits=0;
        while((op=br.readLine())!=null){
            String[] tabOp=op.split(";");
            String typeOp=tabOp[3];
            double montant=Double.parseDouble(tabOp[4]);
            if(typeOp.equals("V"))
                totalVersements+=montant;
            else
                totalRetraits+=montant;
        }
        System.out.println("Total Versement:"+totalVersements);
        System.out.println("Total Retrait:"+totalRetraits);
    }
}
```

Compression ZIP : Stream de traitement : ZipOutputStream

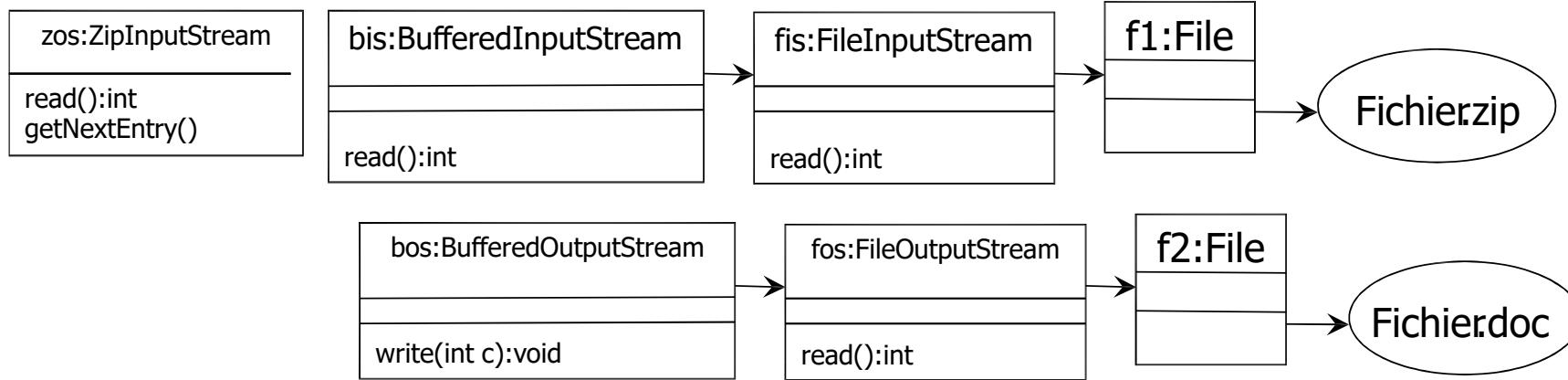


```
File f1=new File("c:/","Fichierdoc");
FileInputStream fis=new FileInputStream(f1);
BufferedInputStream bis=new BufferedInputStream(fis);
File f2=new File("Fichierzip");
FileOutputStream fos=new FileOutputStream(f2);
BufferedOutputStream bos=new BufferedOutputStream(fos);
ZipOutputStream zos=new ZipOutputStream(bos);
zos.setMethod(ZipOutputStream.DEFLATED);
zos.putNextEntry(new ZipEntry(f1.getName()));
int c;
while ((c=bis.read())!=-1){
    zos.write(c);
}
zos.close();bis.close();
```

Exemple 4 : Compression d'un fichier

```
import java.io.*;import java.util.zip.*;
public class App4 {
    public static void main(String[] args) throws Exception {
        File f1=new File("fichier.doc");
        FileInputStream fis=new FileInputStream(f1);
        BufferedInputStream bis=new BufferedInputStream(fis);
        File f2=new File("fichier.zip");
        FileOutputStream fos=new FileOutputStream(f2);
        BufferedOutputStream bos=new BufferedOutputStream(fos);
        ZipOutputStream zos=new ZipOutputStream(bos);
        zos.setMethod(ZipOutputStream.DEFLATED);
        zos.putNextEntry(new ZipEntry(f1.getName()));
        int c;
        while((c=bis.read())!=-1){
            zos.write(c);
        }
        bis.close(); zos.close();
        System.out.println("Capacite de "+f1.getName()+" est : "+f1.length());
        System.out.println("Capacite de "+f2.getName()+" est : "+f2.length());
    }
}
```

Décompression ZIP : Stream de traitement : ZipInputStream



```
File f1=new File("Fichierzip");
FileInputStream fis=new FileInputStream(f1);
BufferedInputStream bis=new BufferedInputStream(fis);
ZipInputStream zis=new ZipInputStream(bis);
ZipEntry ze=zis.getNextEntry();
File f2=new File(ze.getName());
FileOutputStream fos=new FileOutputStream(f2);
BufferedOutputStream bos=new BufferedOutputStream(fos);
int c;
while ((c=zis.read())!=-1){
    bos.write(c);
}
zis.close();bos.close();
```

Exemple 5 : Décompression d'un fichier zip

```
import java.io.*;import java.util.zip.*;
public class App5 {
public static void main(String[] args) throws Exception {
    File f1=new File("fichier.zip");
    FileInputStream fis=new FileInputStream(f1);
    BufferedInputStream bis=new BufferedInputStream(fis);
    ZipInputStream zis=new ZipInputStream(bis);
    ZipEntry ze=zis.getNextEntry();
    File f2=new File(ze.getName());
    FileOutputStream fos=new FileOutputStream(f2);
    BufferedOutputStream bos=new BufferedOutputStream(fos);
    int c;
    while((c=zis.read())!=-1){
        bos.write(c);
    }
    zis.close();  bos.close();
    System.out.println("Capacité de "+f1.getName()+" est : "+f1.length());
    System.out.println("Capacité de "+f2.getName()+" est : "+f2.length());
}
}
```

Exercice 2

Créer une application java qui permet de compresser le contenu d'un répertoire y compris le contenu de ses sous répertoires

Créer une autre application java qui permet de décompresser un fichier ZIP

La sérialisation

La sérialisation est une opérations qui permet d'envoyer un objet sous forme d'une tableau d'octets dans une sortie quelconque (Fichier, réseau, port série etc..)

Les applications distribuées utilisent beaucoup ce concept pour échanger les objets java entre les applications via le réseau.

Pour sérialiser un objet, on utiliser la méthode **writeObject()** de la classe **ObjectOutputStream**

La désérialisation est une opération qui permet de reconstruire l'objet à partir d'une série d'octets récupérés à partir d'une entrée quelconque.

Pour dé sérialiser un objet, on utilise la méthode **readObject()** de la classe **ObjectInputStream**.

Pour pouvoir sérialiser un objet, sa classe doit implémenter l'interface **Serializable**

Pour designer les attributs d'un objet qui ne doivent pas être sérialisés, on doit les déclarer **transient**

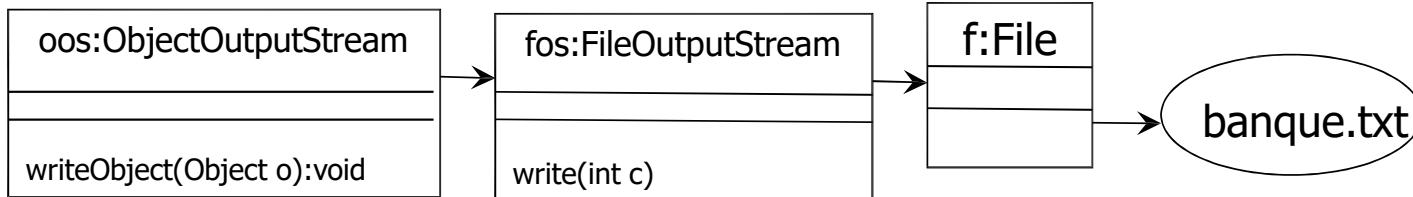
Exemple d'une classe Serializable

```
package metier;
import java.io.Serializable;
import java.util.Date;
public class Operation implements Serializable {
    private int numeroOperation;
    private transient Date dateOperation;
    private String numeroCompte;
    private String typeOperation;
    private double montant;

    public Operation() { }

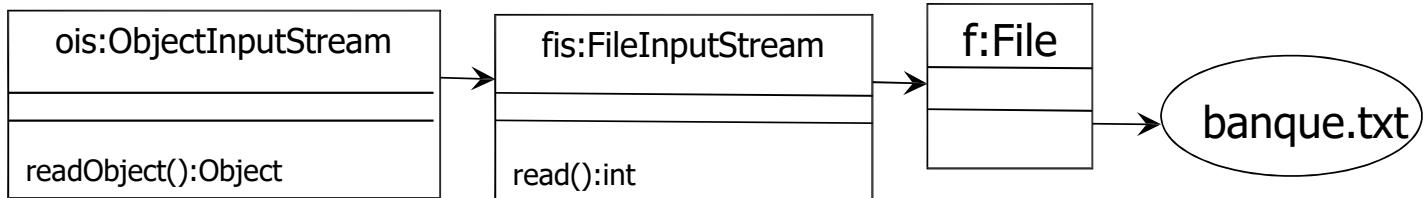
    public Operation(int numOp, Date dateOp, String numC, String to, double mt) {
        this.numeroOperation = numOp;this.dateOperation = dateOp;
        this.numeroCompte = numC;this.typeOperation = to;
        this.montant = mt;
    }
    // Getters et Setters
}
```

Sérialisation



```
import java.io.*;import java.util.Date;  
import metier.Operation;  
public class Serialisation {  
    public static void main(String[] args) throws Exception {  
        Operation op1=new Operation(1,new Date(), "CC1", "V", 40000);  
        Operation op2=new Operation(2,new Date(), "CC1", "R", 6000);  
        File f=new File("banque.txt"); FileOutputStream  
        fos=new FileOutputStream(f); ObjectOutputStream  
        oos=new ObjectOutputStream(fos);  
        oos.writeObject(op1);  
        oos.writeObject(op2);  
        oos.close();  
    }  
}
```

DéSérialisation



```
import java.io.*;
import metier.Operation;
public class Deserialisation {
    public static void main(String[] args) throws Exception {
        File f=new File("banque.txt");
        FileInputStream fis=new FileInputStream(f);
        ObjectInputStream ois=new ObjectInputStream(fis);
        Operation op1=(Operation) ois.readObject();
        Operation op2=(Operation) ois.readObject();
        System.out.println("Num:"+op1.getNumeroOperation());
        System.out.println("Date:"+op1.getDateOperation());
        System.out.println("Compte:"+op1.getNumeroCompte());
        System.out.println("Type:"+op1.getTypeOperation());
        System.out.println("Montant:"+op1.getMontant());
    }
}
```

Exercice 3

Créer une application java qui permet de sérialiser les objets de la classe opérations dans un fichier compressé au format zip.

Créer une application qui permet de dé sérialiser ces objets du fichier compressé.

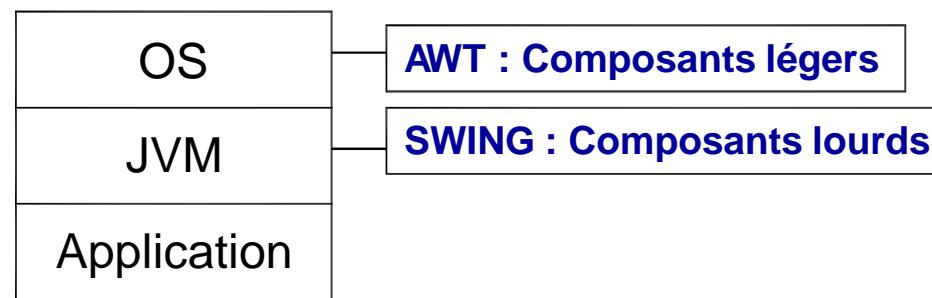
Interfaces graphique dans java

Interfaces graphiques

En java, il existe deux types de composants graphiques:

Composants **AWT** (Abstract Window ToolKit): Composants qui font appel aux composants graphiques de l'OS

Composants **SWING** : Composants écrit complètement avec java et sont indépendant de l'OS



Composants AWT

Nous avons à notre disposition principalement trois types d'objets :

Les **Components** qui sont des composants graphiques. Exemple (Button, Label, TextField...)

Les **Containers** qui contiennent les Components.
Exemple (Frame, Pannel,)

Les **Layouts** qui sont en fait des stratégies de placement de Components pour les Containers.
Exemple (FlowLayout, BorderLayout, GridLayout...)

Premier Exemple AWT et SWING

```
import java.awt.*;
public class AppAWT {
    public static void main(String[] args){
        // Créer une fenêtre
        Frame f=new Frame("Titre");
        // Créer une zone de texte
        TextField t=new TextField(12);
        // Créer un bouton OK
        Button b=new Button("OK");
        // Définir une technique de placement
        f.setLayout(new FlowLayout());
        //ajouter la zone de texte t à frame f
        f.add(t);
        // ajouter le bouton b à la frame f
        f.add(b);
        // Définir les dimensions de la frame
        f.setBounds(10,10,500,500);
        // Afficher la frame
        f.setVisible(true);
    }
}
```

```
import javax.swing.*;
import java.awt.FlowLayout;
public class AppSWING {
    public static void main(String[] args){
        JFrame f=new JFrame("Titre");
        JTextField t=new JTextField(12);
        JButton b=new JButton("OK");
        f.setLayout(new FlowLayout());
        f.add(t); f.add(b);
        f.setBounds(10,10,500,500);
        f.setVisible(true);
    }
}
```



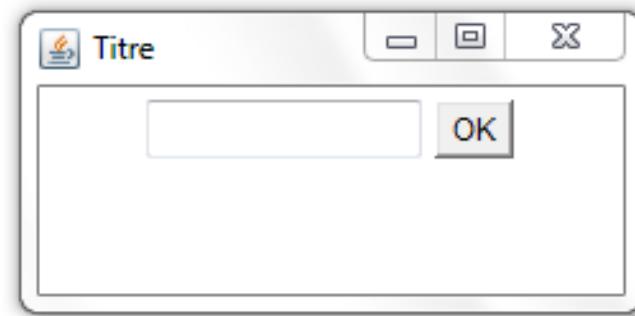
AWT



SWING

Une autre manière plus pratique : Hériter de la classe Frame

```
import java.awt.*;
public class MaFenetreAWT extends Frame{
    // Créer une zone de texte
    TextField t=new TextField(12);
    // Créer un bouton OK
    Button b=new Button("OK");
    public MaFenetreAWT() {
        // Définir une technique de placement
        this.setLayout(new FlowLayout());
        //ajouter la zone de texte t à la frame this
        this.add(t);
        // ajouter le bouton b à la frame this
        this.add(b);
        // Définir les dimensions de la frame
        this.setBounds(10,10,500,500);
        // Afficher la frame
        this.setVisible(true);
    }
    public static void main(String[] args) {
        MaFenetreAWT f=new MaFenetreAWT();
    }
}
```



Gestion des événements

Dans java, pour q'un objet puisse répondre à un événement, il faut lui attacher un écouteur (Listener).

Il existe différents types de Listeners:

WindowsListener : pour gérer les événement sur la fenêtre

ActionListener :pour gérer les événements produits sur les composants graphiques.

KeyListener : pour gérer les événements du clavier

MouseListener : pour gérer les événements de la souris.

....

Gestionnaire d'événement ActionListener

ActionListener est une interface qui définit une seule méthode:

```
public void actionPerformed(ActionEvent e);
```

L'événement actionPerformed est produit quand on valide une action par un clique ou par la touche de validation du clavier.

Pour gérer cet événement dans une application, il faut créer un classe implémentant l'interface ActionListener et redéfinir la réponse aux événements utilisateur, produits dans l'interface, dans la méthode actionPerformed.

La classe implémentant cette interface s'appelle un listener (écouteur) ou un gestionnaire d'événements.

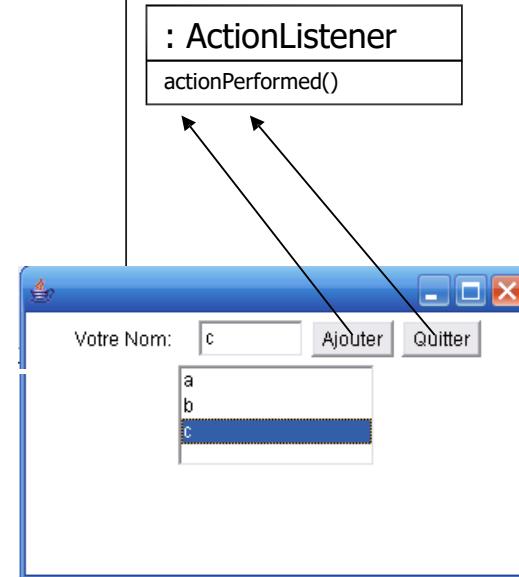
Quand on clique, par exemple, sur un bouton qui est branché sur cet écouteur, la méthode actionPerformed de l'écouteur s'exécute

```

import java.awt.*;import java.awt.event.*;
public class TestEvent extends Frame implements ActionListener
{
    Label l=new Label("Votre Nom:");
    TextField t=new TextField(12);
    Button b=new Button("Ajouter"); List liste=new List();
    Button b2=new Button("Quitter");
    public TestEvent() {
        this.setLayout(new FlowLayout());
        this.add(l);this.add(t);this.add(b);
        this.add(b2);this.add(liste);
        this.setBounds(10,10, 400, 400);
        b.addActionListener(this);
        b2.addActionListener(this);
        this.setVisible(true);
    }
    // Gestionnaire des événements
    public void actionPerformed(ActionEvent e) {
        // Si la source de l'événement est le bouton b
        if(e.getSource()==b){
            // Lire le contenu de la zone de texte
            String s=t.getText();
            // Ajouter la valeur de s à la liste
            liste.add(s);
        }
        // Si la source de l'événement est b2
        else if(e.getSource()==b2){
            // Quitter l'application
            System.exit(0);
        }
    }
    public static void main(String[] args) {
        new TestEvent();
    }
}

```

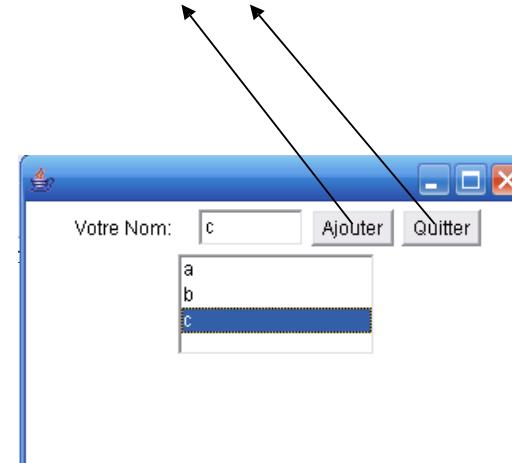
Gestion des événements



En utilisant une classe interne

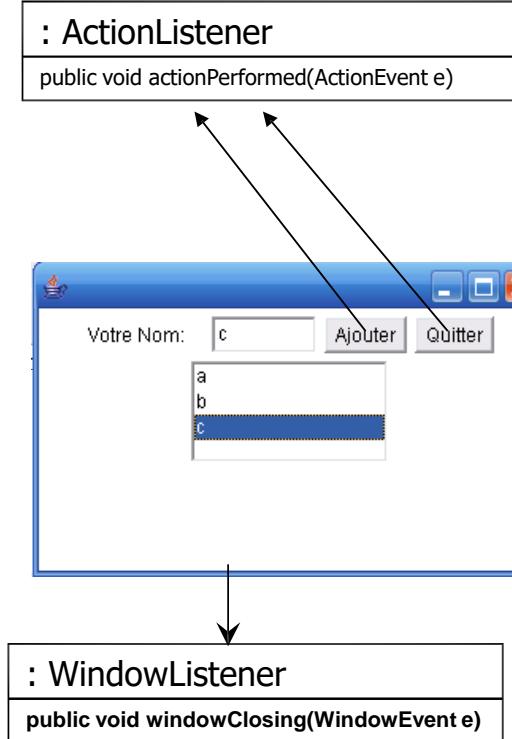
```
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
public class TestEvent2 extends Frame{
    Label l=new Label("Votre Nom:");
    TextField t=new TextField("*****");
    Button b=new Button("Ajouter");
    List liste=new List();
    Button b2=new Button("Quitter");
    class Handler implements ActionListener{
        public void actionPerformed(ActionEvent e) {
            if(e.getSource()==b){
                String s=t.getText();
                liste.add(s);
            }
            else if(e.getSource()==b2){
                System.exit(0);
            }
        }
    }
    public TestEvent2() {
        this.setLayout(new FlowLayout());
        this.add(l);this.add(t);this.add(b);
        this.add(b2);this.add(liste);
        this.setBounds(10,10, 400, 400);
        Handler h=new Handler();
        b.addActionListener(h);
        b2.addActionListener(h);
        this.setVisible(true);
    }
    public static void main(String[] args)
    {
        new TestEvent2();
    }
}
```

h: ActionListener
public void actionPerformed(ActionEvent e)



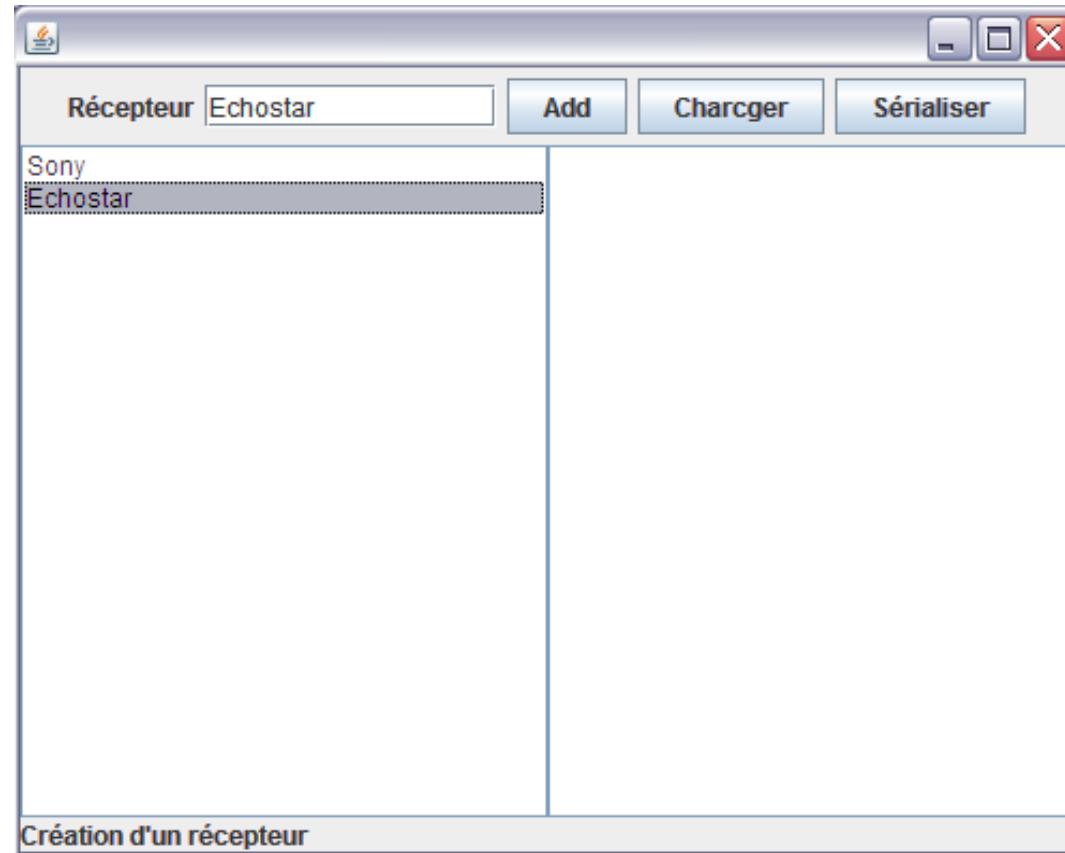
En créant directement un objet de ActionListener

```
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
public class TestEvent extends Frame{
    Label l=new Label("Votre Nom:");
    TextField t=new TextField(12);
    Button b=new Button("Ajouter");
    List liste=new List();
    Button b2=new Button("Quitter");
    public TestEvent() {
        this.setLayout(new FlowLayout());
        this.add(l);this.add(t);this.add(b);
        this.add(b2);this.add(liste);
        this.setBounds(10,10, 400, 400);
        // Si on clique le bouton b1
        b.addActionListener(
    new ActionListener(){
        public void actionPerformed(ActionEvent e) {
            String s=t.getText();
            liste.add(s);
        }
    });
    // Si on clique le bouton b2
    b2.addActionListener(
    new ActionListener(){
        public void actionPerformed(ActionEvent e) {
            System.exit(0);
        }
    });
    // Si on clique le bouton fermer de la fenêtre
    this.addWindowListener(new WindowAdapter(){
        public void windowClosing(WindowEvent e) {
            System.exit(0);
        }
    });
    this.setVisible(true);
}
public static void main(String[] args) {
new TestEvent();
}}
```



Exemple d'application SWING

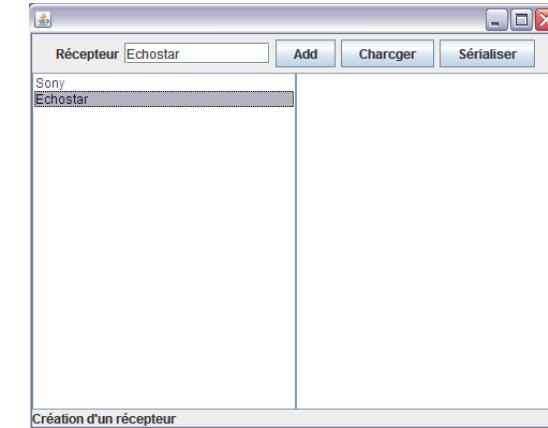
Interface concernant le TP entrée sorties (Récepteur)



Code de l'application : Création des composants

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

import javax.swing.*;
public class RecepteurSWING extends JFrame implements ActionListener
{
    private JLabel jl=new JLabel("Récepteur");
    private JTextField jtf1=new JTextField(12);
    private JButton jb1=new JButton("Add");
    private JButton jb2=new JButton("Chargger");
    private JButton jb3=new JButton("Sérialiser");
    private List liste1=new List();
    private List liste2=new List();
    private JLabel jlErr=new JLabel("OK");
```



Code de l'application : Constructeur

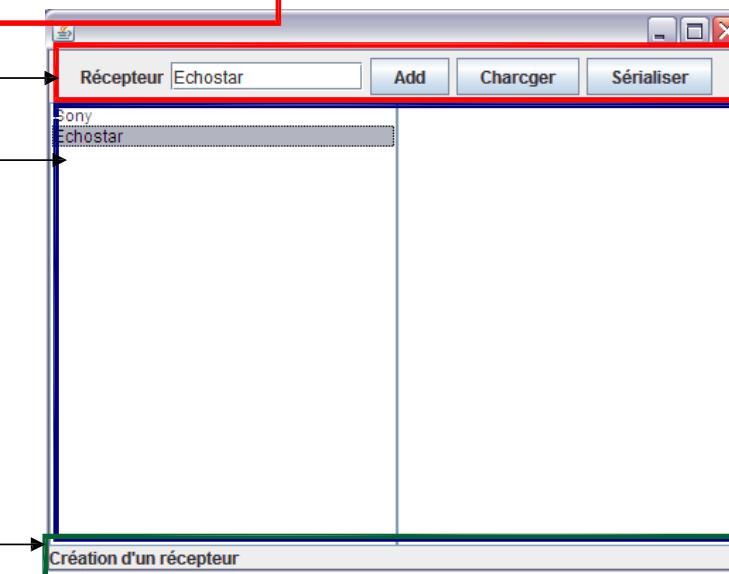
```
public RecepteurSWING() {
    this.setDefaultCloseOperation(EXIT_ON_CLOSE);
    this.setLayout(new BorderLayout());

    JPanel jp1=new JPanel();
    jp1.setLayout(new FlowLayout());
    jp1.add(jl);jp1.add(jtf1);jp1.add(jb1);jp1.add(jb2);
    jp1.add(jb3);
    this.add(jp1,BorderLayout.NORTH);

    JPanel jp2=new JPanel();
    jp2.setLayout(new GridLayout());
    jp2.add(liste1);
    jp2.add(liste2);
    this.add(jp2,BorderLayout.CENTER);

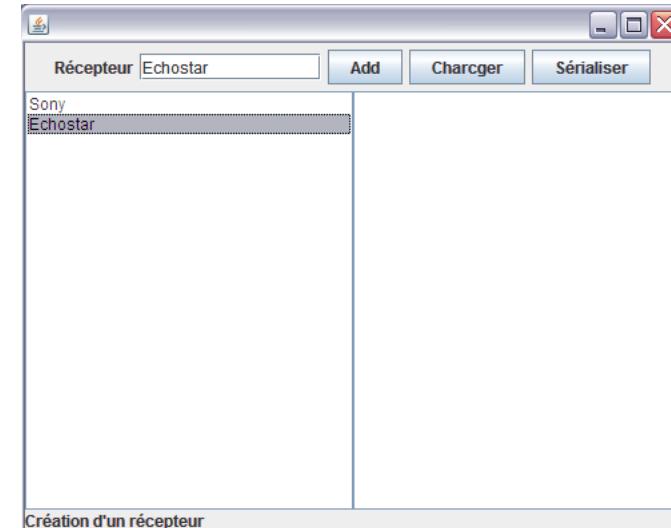
    this.add(jlErr,BorderLayout.SOUTH);

    jb1.addActionListener(this);
    jb2.addActionListener(this);
    jb3.addActionListener(this);
    this.setBounds(10, 10, 500, 400);
    this.setVisible(true);
}
```



Code de l'application :Réponse aux événements

```
public void actionPerformed(ActionEvent e) {  
    if(e.getSource()==jb1){  
        String n=jtf1.getText();  
        listel.add(n);  
        jlErr.setText("Création d'un récepteur");  
        /* A compléter */  
    }  
    else if(e.getSource()==jb2){  
        jlErr.setText("Chargement des chaines");  
        /* A compléter */  
    }  
    else if(e.getSource()==jb3){  
        /* A compléter */  
        jlErr.setText("Sérialisation");  
    }  
}  
  
public static void main(String[] args) {  
    new RecepteurSWING();  
}
```



Application

On souhaite créer une application java qui permet gérer une société de transport de cargaisons transportant des marchandises. La société gère un ensemble de cargaisons. Chaque cargaison contient plusieurs marchandises. Chaque marchandise est définie par son numéro, son poids et son volume.

Il existe deux types de cargaisons : Routière et Aérienne. Chaque cargaison est définie par sa référence et sa distance de parcours. Le cout de transport d'une cargaison est calculé en fonction du type de la cargaison.

Une cargaison aérienne est une cargaison dont le cout est calculé selon la formule suivante :

$\text{cout}=10 \times \text{distance} \times \text{poids total des marchandises}$ si le volume total est inférieur à 80000

$\text{cout}=12 \times \text{distance} \times \text{poids total des marchandises}$ si le volume total est supérieur ou égal à 80000

Une cargaison routière est une cargaison dont le cout est calculé selon la formule suivante :

$\text{cout}=4 \times \text{distance} \times \text{poids total}$ si le volume total est inférieur à 380000

$\text{cout}=6 \times \text{distance} \times \text{poids total}$ si le volume total est supérieur ou égale à 380000

Application

Pour chaque cargaison, on souhaite ajouter une marchandise, supprimer une marchandise, consulter une marchandise sachant son numéro, consulter toutes les marchandises de la cargaison, consulter le poids total de la cargaison, consulter le volume total de la cargaison et consulter le cout de la cargaison.

Cette application peut être utilisée par les clients et les administrateurs.

Le client peut effectuer les opérations suivantes :

- Consulter une cargaison sachant sa référence.
- Consulter une marchandise sachant son numéro.
- Lire le fichier Cargaisons.
- Consulter toutes les cargaisons.

L'administrateur peut effectuer toutes les opérations effectuées par le client. En plus, il peut :

- Ajouter une nouvelle cargaison.
- Ajouter une marchandise à une cargaison.
- Supprimer une cargaison
- Enregistrer les cargaisons dans un fichier.

Toutes les opérations nécessitent une authentification

Questions :

1. Etablir un diagramme Use case UML.
2. Etablir le diagramme de classes en prenant en considération les critères suivants.

La classe SocieteTransport devrait implémenter les deux interfaces IClientTransport et IAdminTransport déclarant, respectivement les opérations relatives aux rôles Client et Admin.

Dans une première implémentation de SocieteTransport, on suppose que les cargaisons sont stockées dans une liste de type HashMap de la classe SocieteTransport.

Dans une deuxième implémentation, nous supposerons que les cargaisons et les marchandises sont stockées dans une base de données relationnelle.

L'association entre cargaison et Marchandise est bidirectionnelle.

3. Ecrire le code java des classes entités Marchandise, Cargaison, CargaisonRoutière et CargaisonAérienne
4. Ecrire le code java des deux interfaces IClientTransport et IAdminTransport
5. Créer une première implémentation java de ces deux interfaces

Questions :

7. Créer une application qui permet de :

Créer une instance de SocieteTransport pour un administrateur.

Ajouter trois cargaisons routières et une cargaison aérienne à société de transport : « CR1 », « CA1 » et « CR2 »

Ajouter 3 marchandises à la cargaison dont la référence est CR1 (Numéros 1, 2, 3)

Ajouter 2 marchandises à la cargaison dont la référence est CA1 (Numéros 4,5)

Afficher toutes les informations concernant la cargaison CR1

Afficher toutes les informations concernant la marchandise 3.

Sérialiser les données dans le fichier « transport1.data»

8. Créer une autre application qui permet de :

Créer une instance de la classe SocieteTransport pour un client.

Charger les données à partir du fichier « transport1.data »

Afficher toutes les informations concernant la cargaison CA1

9. Créer une application SWING qui permet la saisie, l'ajout, la suppression, la consultation des données de l'application.

Accès aux bases de données via JDBC

M.Youssfi

Pilotes JDBC

Pour qu'une application java puisse communiquer avec un serveur de bases de données, elle a besoin d'utiliser les pilotes JDBC (Java Data Base Connectivity)

Les Pilotes JDBC est une bibliothèque de classes java qui permet, à une application java, de communiquer avec un SGBD via le réseau en utilisant le protocole TCP/IP

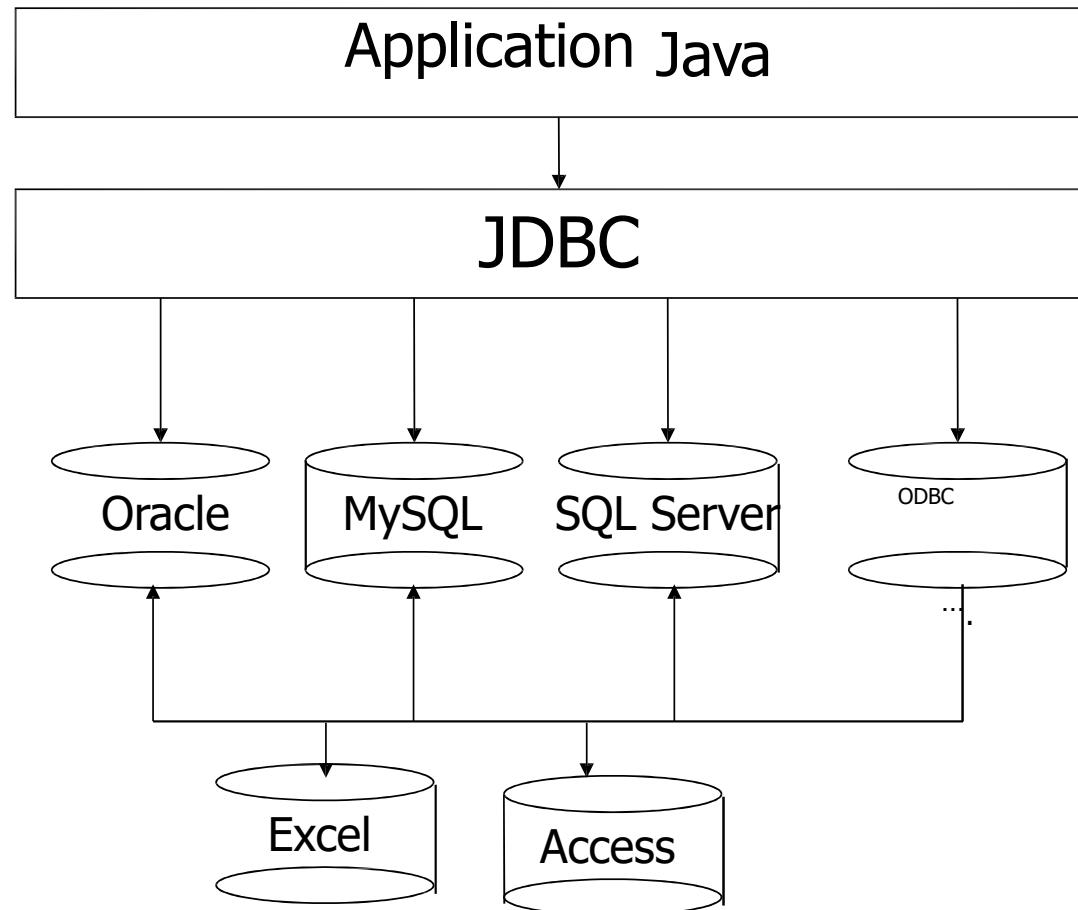
Chaque SGBD possède ses propres pilotes JDBC.

Il existe un pilote particulier « JdbcOdbcDriver » qui permet à une application java communiquer avec n'importe quelle source de données via les pilotes ODBC (Open Data Base Connectivity)

Les pilotes ODBC permettent à une application Windows de communiquer une base de données quelconque (Access, Excel, MySQL, Oracle, SQL SERVER etc...)

La bibliothèque JDBC a été conçu comme interface pour l'exécution de requêtes SQL.Une application JDBC est isolée des caractéristiques particulières du système de base de données utilisé.

Java et JDBC



Créer une application JDBC

Pour créer une application élémentaire de manipulation d'une base de données il faut suivre les étapes suivantes :

Chargement du Pilote JDBC ;

Identification de la source de données ;

Allocation d'un objet **Connection**

Allocation d'un objet Instruction **Statement** (ou **PreparedStatement**) ;

Exécution d'une requête à l'aide de l'objet Statement ;

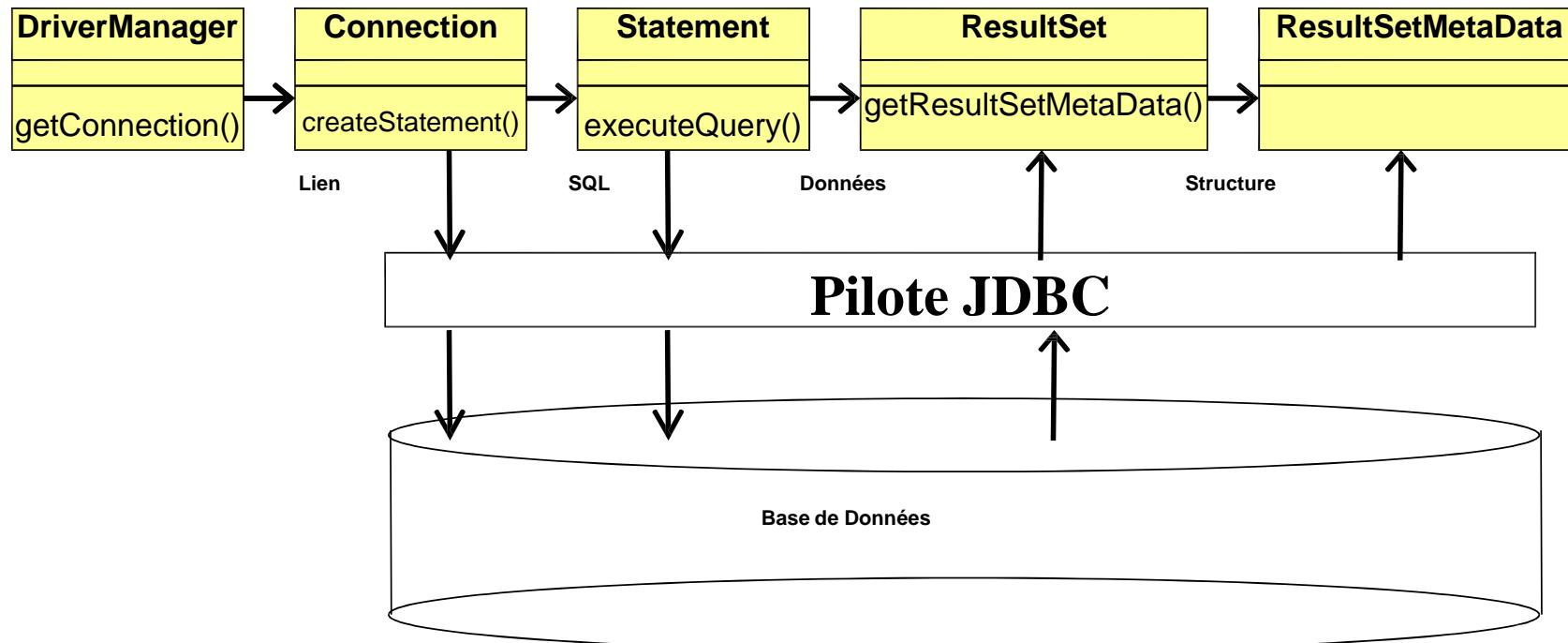
Récupération de données à partir de l'objet renvoyé **ResultSet** ;

Fermeture de l'objet ResultSet ;

Fermeture de l'objet Statement ;

Fermeture de l'objet Connection.

Créer une application JDBC



Démarche JDBC

Charger les pilotes JDBC :

Utiliser la méthode `forName` de la classe `Class`, en précisant le nom de la classe pilote.

Exemples:

Pour charger le pilote JdbcOdbcDriver:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver") ;
```

Pour charger le pilote jdbc de MySQL:

```
Class.forName("com.mysql.jdbc.Driver") ;
```

Créer une connexion

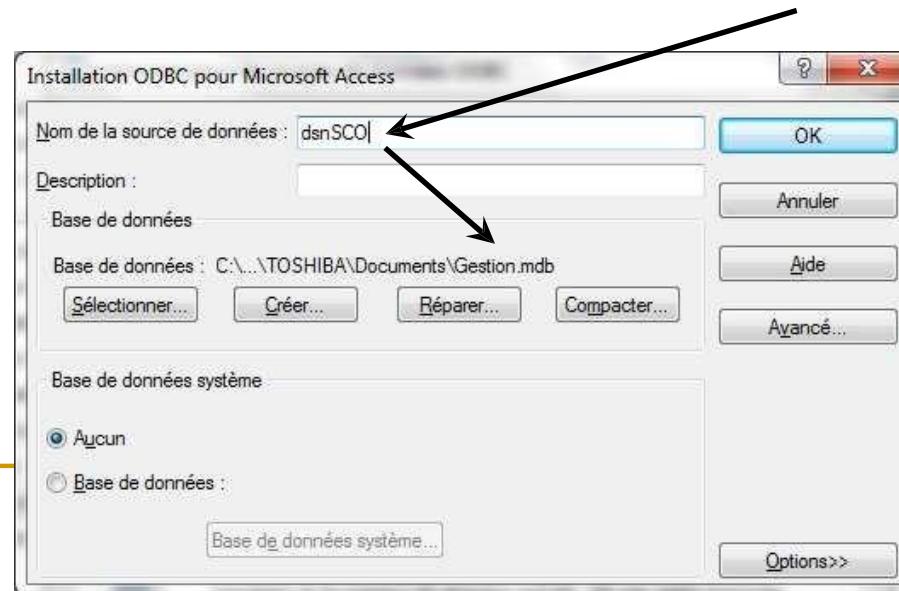
Pour créer une connexion à une base de données, il faut utiliser la méthode statique `getConnection()` de la classe `DriverManager`. Cette méthode fait appeler aux pilotes JDBC pour établir une connexion avec le SGBDR, en utilisant les sockets.

Pour un pilote `com.mysql.jdbc.Driver` :

```
Connection conn=DriverManager.getConnection("jdbc:mysql://localhost:3306/DB", "user", "pass");
```

Pour un pilote `sun.jdbc.odbc.JdbcOdbcDriver` :

```
Connection conn=DriverManager.getConnection("jdbc:odbc:dsnSCO", "user", "pass");
```



Objets Statement, ResultSet et ResultSetMetaData

Pour exécuter une requête SQL, on peut créer l'objet Statement en utilisant la méthode `createStatement()` de l'objet Connection.

Syntaxe de création de l'objet Statement

```
Statement st=conn.createStatement();
```

Exécution d'une requête SQL avec l'objet Statement :

Pour exécuter une requête SQL de type select, on peut utiliser la méthode `executeQuery()` de l'objet Statement. Cette méthode exécute la requête et stocke le résultat de la requête dans l'objet ResultSet:

```
ResultSet rs=st.executeQuery("select * from PRODUITS");
```

Pour exécuter une requête SQL de type insert, update et delete on peut utiliser la méthode `executeUpdate()` de l'objet Statement :

```
st.executeUpdate("insert into PRODUITS (...) values(...)");
```

Pour récupérer la structure d'une table, il faut créer l'objet ResultSetMetaData en utilisant la méthode `getMetaData()` de l'objet ResultSet.

```
ResultSetMetaData rsmd = rs.getMetaData();
```

Objet PreparedStatement

Pour exécuter une requête SQL, on peut également créer l'objet PreparedStatement en utilisant la méthode `prepareStatement()` de l'objet Connection.

Syntaxe de création de l'objet PreparedStatement

```
PreparedStatement ps=conn.prepareStatement("select *  
from PRODUITS where NOM_PROD like ? AND PRIX<?");
```

Définir les valeurs des paramètres de la requête:

```
ps.setString(1,"%"+motCle+"%");  
ps.setString(2, p);
```

Exécution d'une requête SQL avec l'objet PreparedStatement :

Pour exécuter une requête SQL de type select, on peut utiliser la méthode `executeQuery()` de l'objet Statement. Cette méthode exécute la requête et stocke le résultat de la requête dans l'objet ResultSet:

```
ResultSet rs=ps.executeQuery();
```

Pour exécuter une requête SQL de type insert, update et delete on peut utiliser la méthode `executeUpdate()` de l'objet Statement :

```
ps.executeUpdate();
```

Récupérer les données d'un ResultSet

Pour parcourir un ResultSet, on utilise sa méthode next() qui permet de passer d'une ligne à l'autre. Si la ligne suivante existe, la méthode next() retourne true. Si non elle retourne false.

Pour récupérer la valeur d'une colonne de la ligne courante du ResultSet, on peut utiliser les méthodes getInt(colonne), getString(colonne), getFloat(colonne), getDouble(colonne), getDate(colonne), etc... colonne représente le numéro ou le nom de la colonne de la ligne courante.

Syntaxe:

```
while(rs.next()) {  
    System.out.println(rs.getInt(1));  
    System.out.println(rs.getString("NOM_PROD"));  
    System.out.println(rs.getDouble("PRIX"));  
}
```

Exploitation de l'objet ResultSetMetaData

L'objet ResultSetMetaData est très utilisé quand on ne connaît pas la structure d'un ResultSet. Avec L'objet ResultSetMetaData, on peut connaître le nombre de colonnes du ResultSet, le nom, le type et la taille de chaque colonne.

Pour afficher, par exemple, le nom, le type et la taille de toutes les colonnes d'un ResultSet rs, on peut écrire le code suivant:

```
ResultSetMetaData rsmd=rs.getMetaData();
// Parcourir toutes les colonnes
for(int i=0;i<rsmd.getColumnCount();i++) {
    // afficher le nom de la colonne numéro i
    System.out.println(rsmd.getColumnName(i));
    // afficher le type de la colonne numéro i
    System.out.println(rsmd.getColumnTypeName(i));
    // afficher la taille de la colonne numéro i
    System.out.println(rsmd getColumnDisplaySize(i));
}
// Afficher tous les enregistrements du ResultSet rs
while (rs.next()){
for(int i=0;i<rsmd.getColumnCount();i++) {
System.out.println(rs.getString(i));
}
}
```

Mapping objet relationnel

Dans la pratique, on cherche toujours à séparer la logique de métier de la logique de présentation.

On peut dire qu'on peut diviser une application en 3 couches:

La couche d'accès aux données: DAO

Partie de l'application qui permet d'accéder aux données de l'applications . Ces données sont souvent stockées dans des bases de données relationnelles .

La couche Métier:

Regroupe l'ensemble des traitements que l'application doit effectuer.

La couche présentation:

S'occupe de la saisie des données et de l'affichage des résultats;

Architecture d'une application

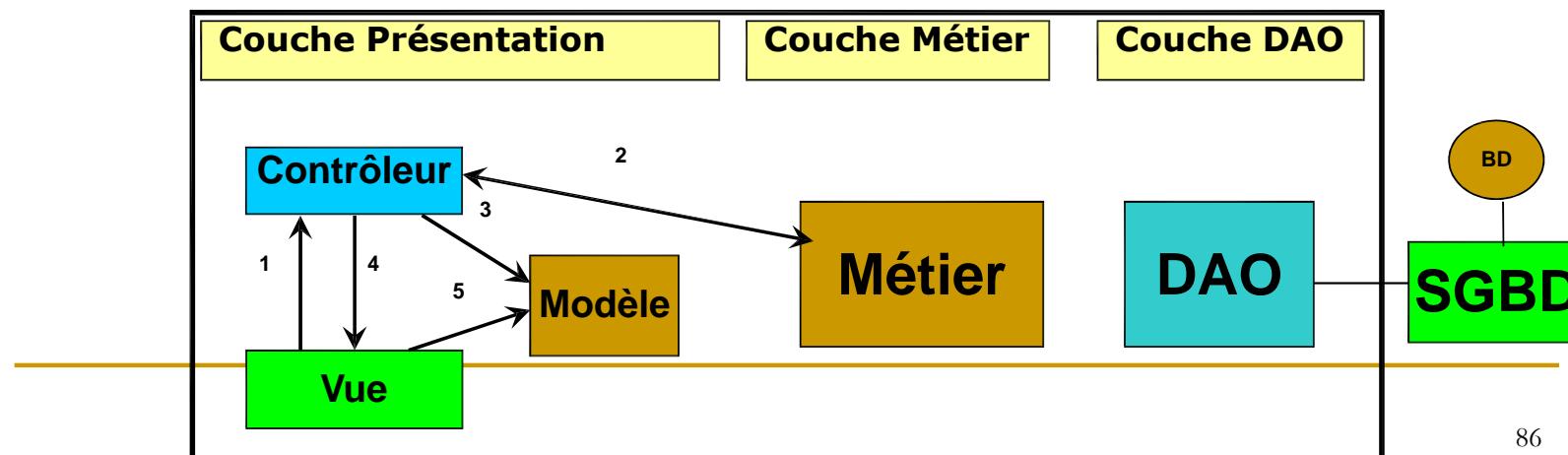
Une application se compose de plusieurs couches:

La couche DAO qui s'occupe de l'accès aux bases de données.

La couche métier qui s'occupe des traitements.

La couche présentation qui s'occupe de la saisie, le contrôle et l'affichage des résultats. Généralement la couche présentation respecte le pattern MVC qui fonctionne comme suit:

1. La vue permet de saisir les données, envoie ces données au contrôleur
2. Le contrôleur récupère les données saisies. Après la validation de ces données, il fait appel à la couche métier pour exécuter des traitements.
3. Le contrôleur stocke le résultat de le modèle.
4. Le contrôleur fait appel à la vue pour afficher les résultats.
5. La vue récupère les résultats à partir du modèle et les affiche.



Mapping objet relationnel

D'une manière générale les applications sont orientée objet :

- Manipulation des objet et des classes

- Utilisation de l'héritage et de l'encapsulation

- Utilisation du polymorphisme

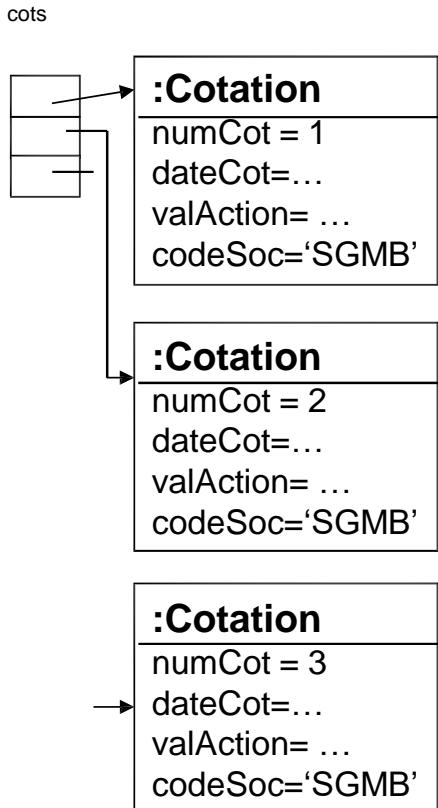
D'autre part les données persistantes sont souvent stockées dans des bases de données relationnelles.

Le mapping Objet relationnel consiste à faire correspondre un enregistrement d'une table de la base de données à un objet d'une classe correspondante.

Dans ce cas on parle d'une classe persistante.

Une classe persistante est une classe dont l'état de ses objets sont stockés dans une unité de sauvegarde (Base de données, Fichier, etc..)

Couche Métier : Mapping objet relationnel



```
public List<Cotation> getCotations(String codeSoc){
    List<Cotation> cotations=new ArrayList<Cotation>();
    try {
        Class.forName("com.mysql.jdbc.Driver");
        Connection conn=DriverManager.getConnection
        ("jdbc:mysql://localhost:3306/bourse_ws","root","");
        PreparedStatement ps=conn.prepareStatement
        ("select * from cotations where CODE_SOCIETE=?");
        ps.setString(1, codeSoc);
        ResultSet rs=ps.executeQuery();
        while(rs.next()){
            Cotation cot=new Cotation();
            cot.setNumCotation(rs.getLong("NUM_COTATION"));
            cot.setDateCotation(rs.getDate("DATE_COTATION"));
            cot.setCodeSociete(rs.getString("CODE_SOCIETE"));
            cot.setValAction(rs.getDouble("VAL_ACTION"));
            cotations.add(cot);
        }
    } catch (Exception e) { e.printStackTrace(); }
    return(cotations);
}
```

NUM_COTATION	DATE_COTATION	VAL_ACTION	CODE_SOCIETE
1	2008-08-30 15:57:50	2093.17199826538	SGMB
2	2008-08-30 15:57:52	258.769396752267	SGMB
3	2008-08-30 15:57:52	1050.71222698514	SGMB

Application Orientée objet

Base de données relationnelle

Application

On considère une base de données qui contient une table ETUDIANTS qui permet de stocker les étudiants d'une école. La structure de cette table est la suivante :

Champ	Type	Interclassement	Attributs	Null	Défaut	Extra
ID_ET	int(11)			Non	Aucun	auto_increment
NOM	varchar(25)	latin1_swedish_ci		Non	Aucun	
PRENOM	varchar(25)	latin1_swedish_ci		Non	Aucun	
EMAIL	varchar(25)	latin1_swedish_ci		Non	Aucun	
VILLE	varchar(200)	latin1_swedish_ci		Non	Aucun	

ID_ET	NOM	PRENOM	EMAIL	VILLE
1	A	PA	A@YAHOO.FR	casa
2	B	PB	B@YAHOO.FR	rabat
3	C	PC	C@YAHOO.FR	casa
4	BBCAAC	BBCAAC	ab@yahoo.fr	casa

Nous souhaitons créer une application java qui permet de saisir au clavier un motclé et d'afficher tous les étudiants dont le nom contient ce mot clé.

Dans cette application devons séparer la couche métier de la couche présentation.

Application

Pour cela, la couche métier est représentée par un modèle qui se compose de deux classes :

La classe Etudiant.java : c'est une classe persistante c'est-à-dire que chaque objet de cette classe correspond à un enregistrement de la table ETUDIANTS. Elle se compose des :

- champs privés idEtudiant, nom, prenom, email et ville,
- d'un constructeur par défaut,
- des getters et setters.

Ce genre de classe c'est ce qu'on appelle un java bean.

La classe Scolarite.java :

c'est une classe non persistante dont laquelle, on implémente les différentes méthodes métiers.

Dans cette classe, on fait le mapping objet relationnel qui consiste à convertir un enregistrement d'une table en objet correspondant.

Dans notre cas, une seule méthode nommée getEtudiants(String mc) qui permet de retourner une Liste qui contient tous les objets Etudiant dont le nom contient le mot Clé «mc»

Application

Travail à faire :

Couche données :

Créer la base de données « SCOLARITE » de type MSAccess ou MySQL

Pour la base de données Access, créer une source de données système nommée « dsnScolarite », associée à cette base de données.

Saisir quelques enregistrements de test

Couche métier. (package metier) :

Créer la classe persistante Etudiant.java

Créer la classe des business méthodes Scolarite.java

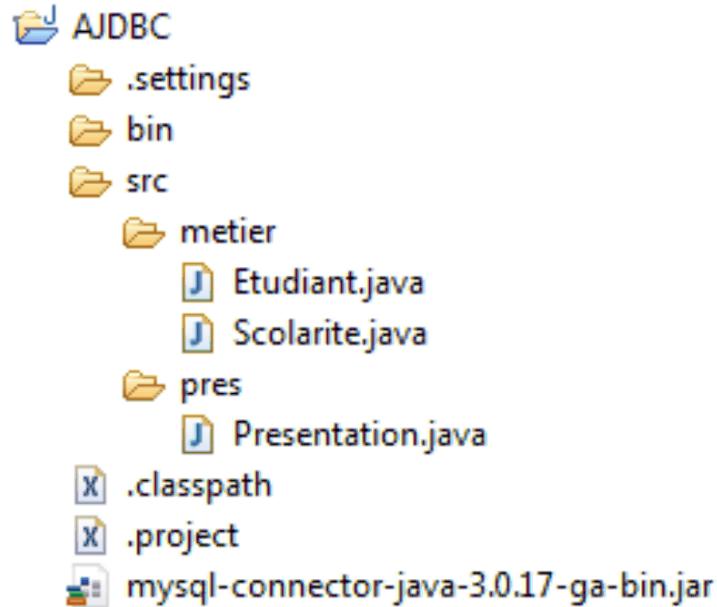
Couche présentation (package pres):

Créer une application de test qui permet de saisir au clavier le mot clé et qui affiche les étudiants dont le nom contient ce mot clé.

Couche métier : la classe persistante Etudiant.java

```
package metier;

public class Etudiant {
    private Long idEtudiant;
    private String nom,prenom,email;
    // Getters et Setters
}
```



Couche métier : la classe Scolarite.java

```
package metier;
import java.sql.*; import java.util.*;
public class Scolarite {
    public List<Etudiant> getEtudiantParMC(String mc){
        List<Etudiant> etds=new ArrayList<Etudiant>();
        try {
            Class.forName("com.mysql.jdbc.Driver");
            Connection conn=
                DriverManager.getConnection("jdbc:mysql://localhost:3306/DB_SCO","root","");
            PreparedStatement ps=conn.prepareStatement("select * from ETUDIANTS where NOM
                like ?");
            ps.setString(1,"%"+mc+"%");
            ResultSet rs=ps.executeQuery();
            while(rs.next()){
                Etudiant et=new Etudiant();
                et.setIdEtudiant(rs.getLong("ID_ET"));et.setNom(rs.getString("NOM"));
                et.setPrenom(rs.getString("PRENOM"));et.setEmail(rs.getString("EMAIL"));
                etds.add(et);
            }
        } catch (Exception e) { e.printStackTrace(); }
        return etds;
    }
}
```

Couche Présentation : Applications Simple

```
package pres;
import java.util.List;import java.util.Scanner;
import metier.Etudiant;import metier.Scolarite;
public class Presentation {
    public static void main(String[] args) {

        Scanner clavier=new Scanner(System.in);

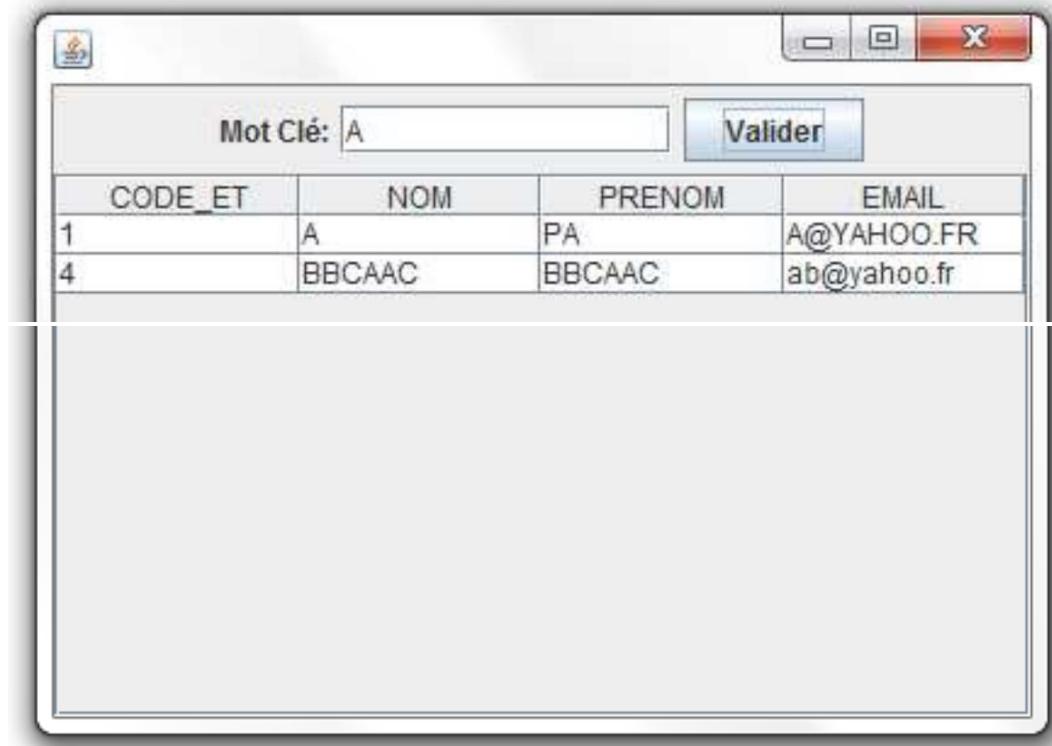
        System.out.print("Mot Clé:");
        String mc=clavier.next();

        Scolarite metier=new Scolarite();

        List<Etudiant> etds=metier.getEtudiantParMC(mc);

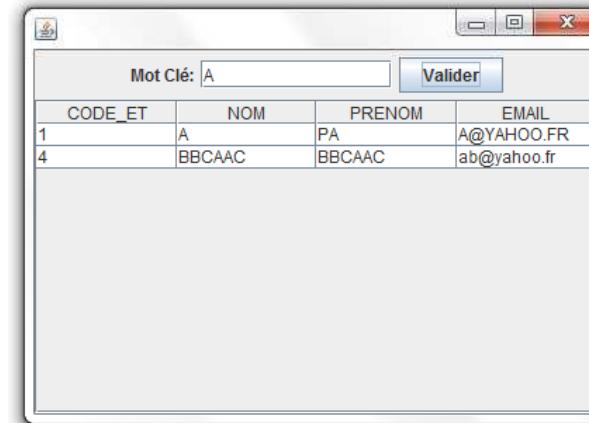
        for(Etudiant et:etds)
            System.out.println(et.getNom()+"\t"+et.getEmail());
    }
}
```

Couche Présentation : Application SWING



Le modèle de données pour JTable

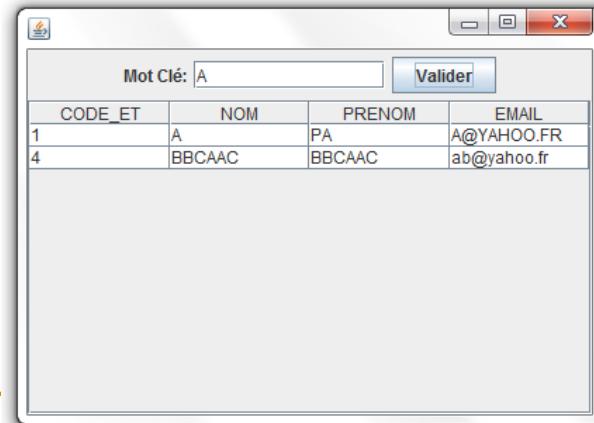
```
package pres;
import java.util.List;import java.util.Vector;
import javax.swing.table.AbstractTableModel;
import metier.Etudiant;
public class EtudiantModele extends AbstractTableModel{
    private String[] tabelColumnNames=new
    String[]{"CODE_ET","NOM","PRENOM","EMAIL"};
    private Vector<String[]> tableValues=new Vector<String[]>();
@Override
public int getRowCount() {
    return tableValues.size();
}
@Override
public int getColumnCount() {
    return tabelColumnNames.length;
}
@Override
public Object getValueAt(int rowIndex, int columnIndex) {
    return tableValues.get(rowIndex) [columnIndex];
}
```



Le modèle de données pour JTable (Suite)

```
@Override
public String getColumnName(int column) {
    return tabelColumnNames[column];
}

public void setData(List<Etudiant> etudiants) {
    tableValues=new Vector<String[]>();
    for(Etudiant et:etudiants){
        tableValues.add(new String[]{
            String.valueOf(et.getIdEtudiant()),et.getNom(),et.getPrenom(),
            et.getEmail()});
    }
    fireTableChanged(null);
}
}
```



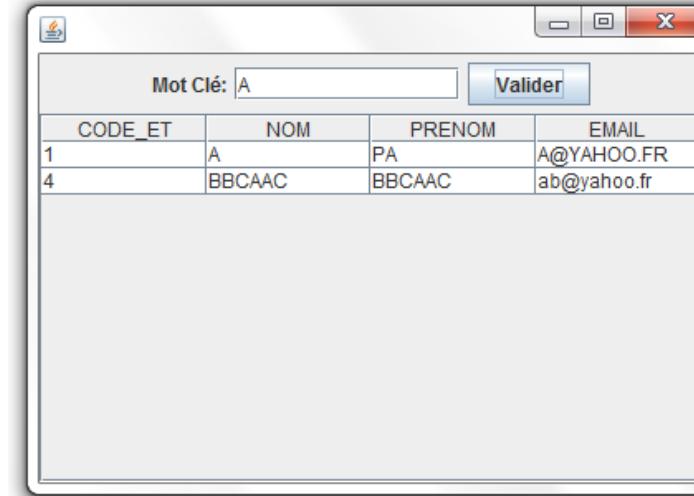
L'application SWING

```
package pres;
import java.awt.*;import java.awt.event.*;import javax.swing.*;
import metier.*;
public class EtudiantFrame extends JFrame {
    private JScrollPane jsp;

    private JLabel jLMC=new JLabel("Mot Clé:");
    private JTextField jTFMC=new JTextField(12);
    private JButton jBValidator=new JButton("Valider");

    private JTable jTableEtudiants;
    private JPanel jpN=new JPanel();

    private Scolarite metier
=new Scolarite();
    private EtudiantModele model;
```



L'application SWING (Suite)

```
public EtudiantFrame() {  
    this.setDefaultCloseOperation(DISPOSE_ON_CLOSE);  
    this.setLayout(new BorderLayout());  
    jTFMC.setToolTipText("Mot Clé:");  
    jpN.setLayout(new FlowLayout());  
    jpN.add(jLMC); jpN.add(jTFMC);  
    jpN.add(jBValider);  
    this.add(jpN, BorderLayout.NORTH);  
    model=new EtudiantModele();  
    jTableEtudiants=new JTable(model);  
    jsp=new JScrollPane(jTableEtudiants);  
    this.add(jsp, BorderLayout.CENTER);  
    this.setBounds(10,10,500,500);  
    this.setVisible(true);  
    jBValider.addActionListener(new ActionListener(){  
        public void actionPerformed(ActionEvent e) {  
            String mc=jTFMC.getText();  
            model.setData(metier.getEtudiantParMC(mc));  
        }  
    });  
}  
  
public static void main(String[] args) {  
    new EtudiantFrame();  
}
```

