

# Conception et Programmation Orientée Objet Java

# Introduction

Qualité logicielle

---

# Rappels :Qualité d'un Logiciel

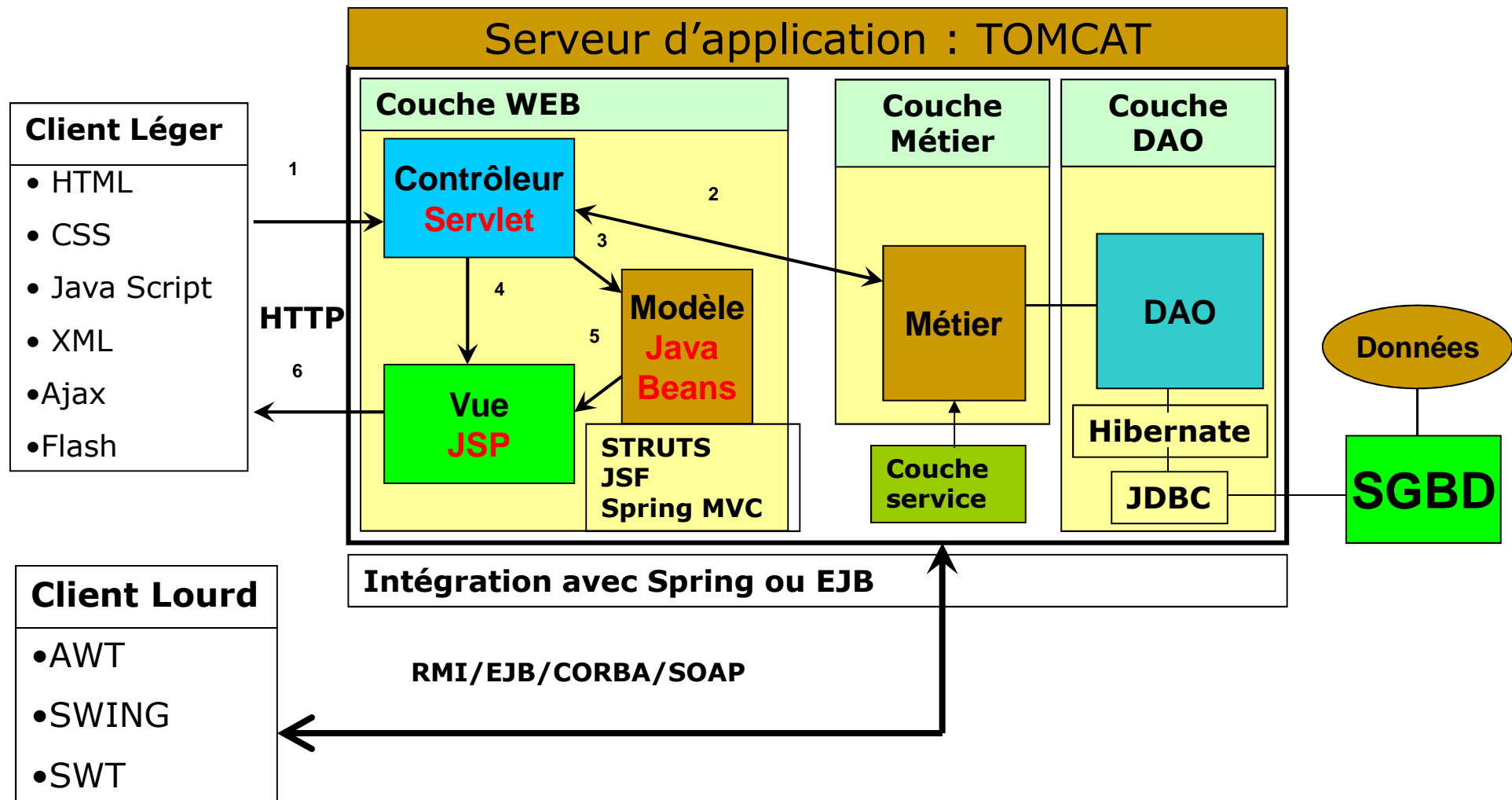
- La qualité d'un logiciel se mesure par rapport à plusieurs critères :
  - ❑ Répondre aux spécifications fonctionnelles :
    - Une application est créée pour répondre , tout d'abord, aux besoins fonctionnels des entreprises.
  - ❑ Les performances:
    - La rapidité d'exécution et Le temps de réponse
    - Doit être bâtie sur une architecture robuste.
    - Eviter le problème de montée en charge
  - ❑ La maintenance:
    - Une application doit évoluer dans le temps.
    - Doit être fermée à la modification et ouverte à l'extension
    - Une application qui n'évolue pas meurt.
    - Une application mal conçue est difficile à maintenir, par suite elle finit un jour à la poubelle.

---

# Qualité d'un Logiciel

- La qualité d'un logiciel se mesure par rapport à plusieurs critères : (Suite)
  - Sécurité
    - Garantir l'intégrité et la sécurité des données
  - Portabilité
    - Doit être capable de s'exécuter dans différentes plateformes.
  - Capacité de communiquer avec d'autres applications distantes.
  - Disponibilité et tolérance aux pannes
  - Capacité de fournir le service à différents type de clients :
    - Client lourd : Interfaces graphiques SWING
    - Interface Web : protocole Http
    - Téléphone : SMS
    - ....
  - Design des ses interfaces graphiques
    - Charte graphique et charte de navigation
    - Accès via différentes interfaces (Web, Téléphone, PDA, ,)
  - Coût du logiciel

# Architecture J2EE



# Programmation Orientée Objet Java

---

# Programme

- Java?
- Programmation orientée Objet Java
  - Objet et Classe
  - Héritage et accessibilité
  - Polymorphisme
  - Collections
- Notions de Design Patterns
- Exceptions et Entrées Sorties
- Interfaces graphique AWT et SWING
- Accès aux bases de données
- Threads et Sockets

---

# 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?

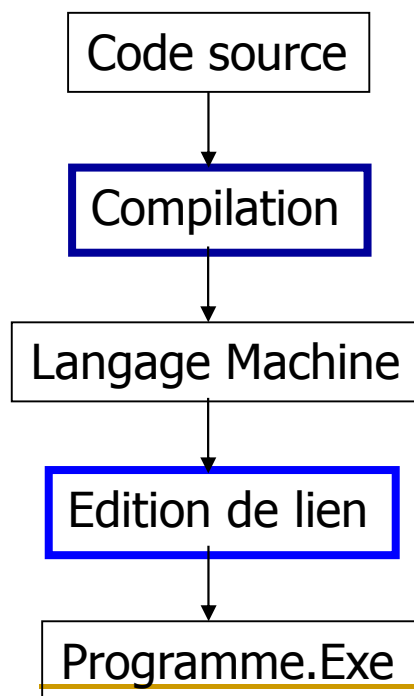
- Java est utilisé pour créer :
  - ❑ Des applications Desktop
  - ❑ Des applets java (applications java destinées à s'exécuter dans une page web)
  - ❑ Des applications pour les smart phones
  - ❑ Des applications embarquées dans des cartes à puces
  - ❑ Des application JEE (Java Entreprise Edition)
- Pour créer une application java, il faut installer un kit de développement java
  - ❑ JSDK : Java Standard Development 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

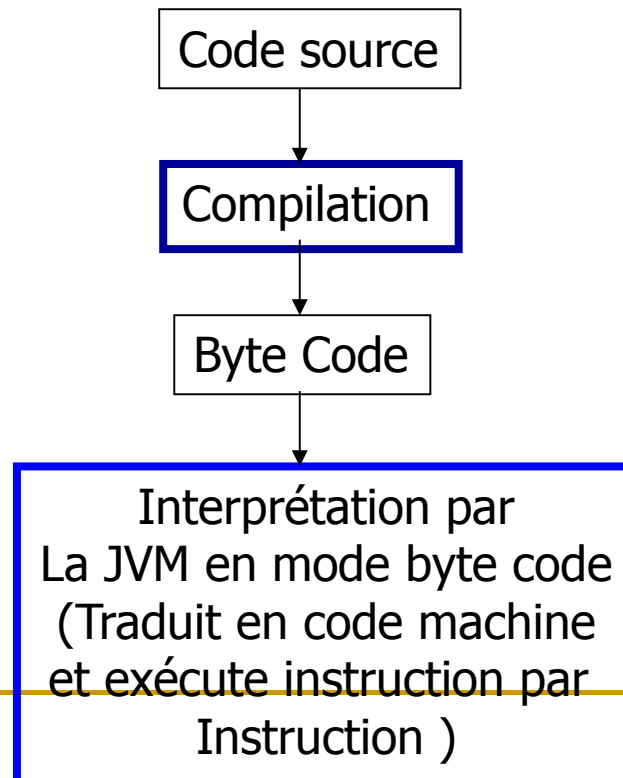
Java est un langage compilé et interprété

- Compilation en mode natif
- Compilation Byte Code
- Compilation en mode JIT(Just In Time)

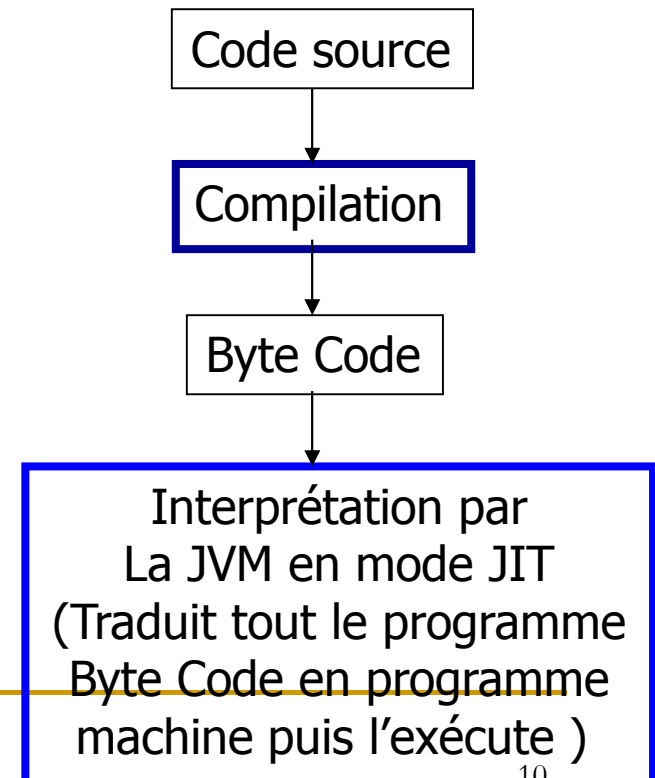
## Natif



## Byte Code

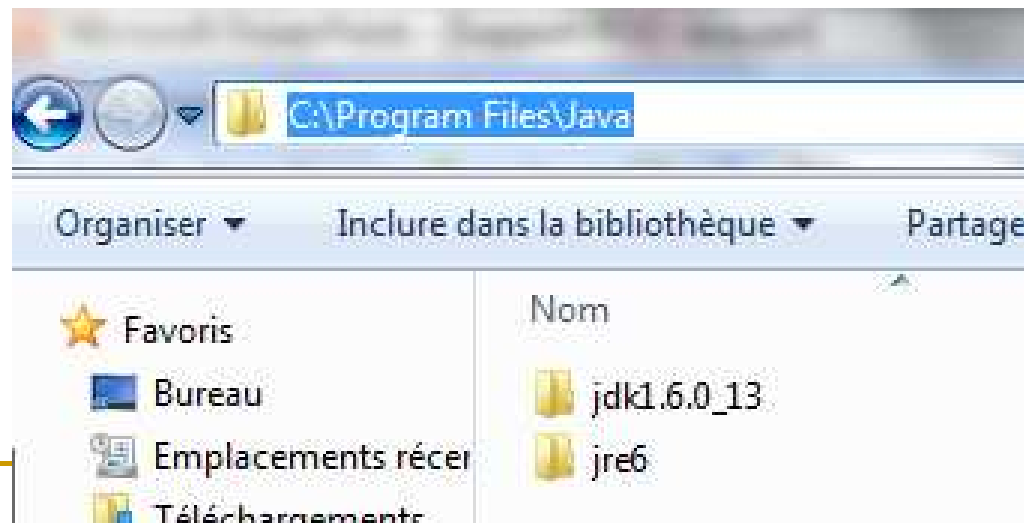


## JIT

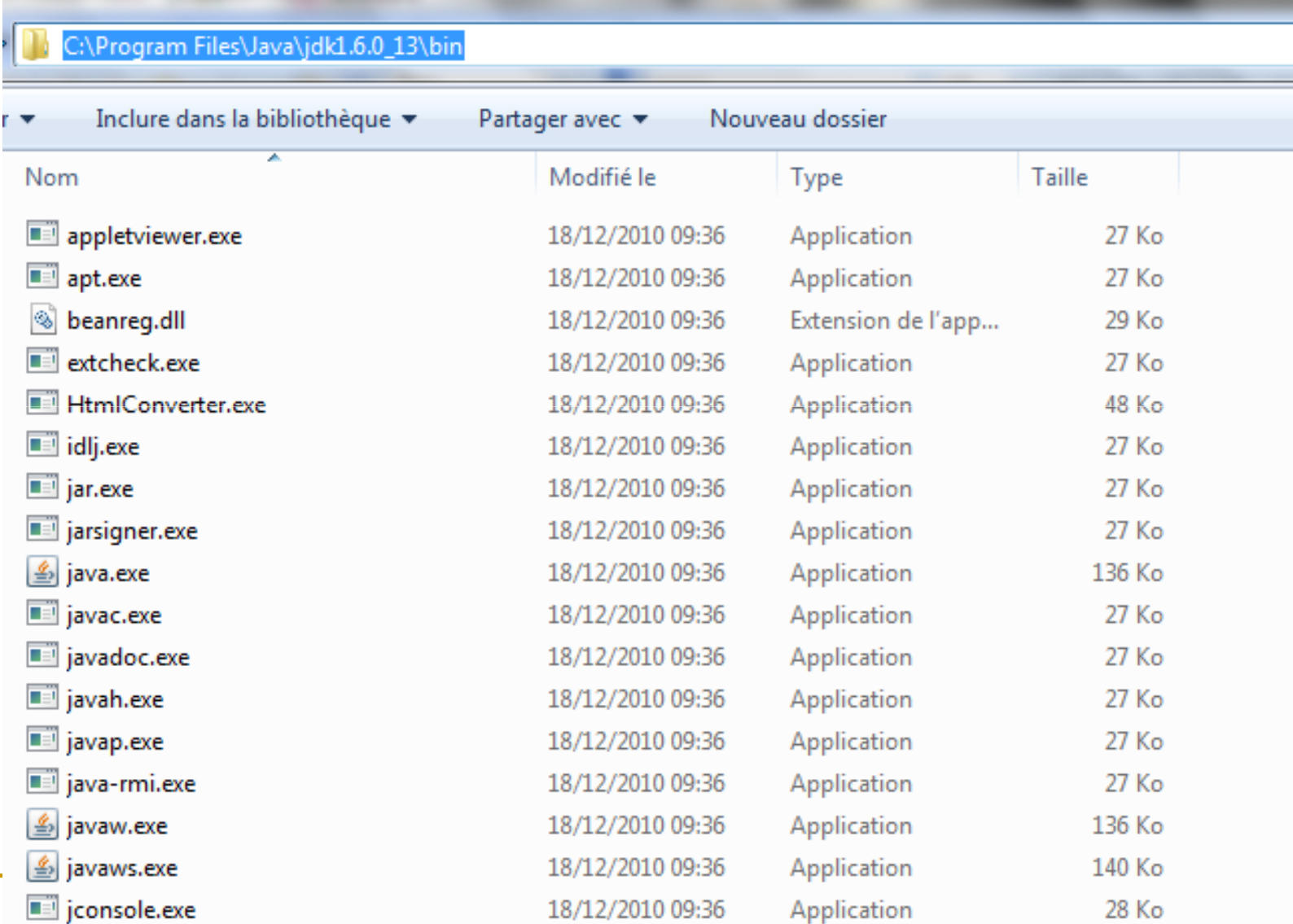


# Installation de java

- Le Kit de développement java JDK peut être téléchargé gratuitement à partir du site de Sun Microsystems son éditeur principal ([www.java.sun.com](http://www.java.sun.com)).
- Le JDK contient 3 trois paquets :
  - 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



A screenshot of a Windows Explorer window showing the contents of the JDK bin directory. The address bar displays the path `C:\Program Files\Java\jdk1.6.0_13\bin`. The window has a menu bar with options: 'Inclure dans la bibliothèque', 'Partager avec', and 'Nouveau dossier'. The main area shows a list of files and folders with columns for 'Nom', 'Modifié le', 'Type', and 'Taille'.

| 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

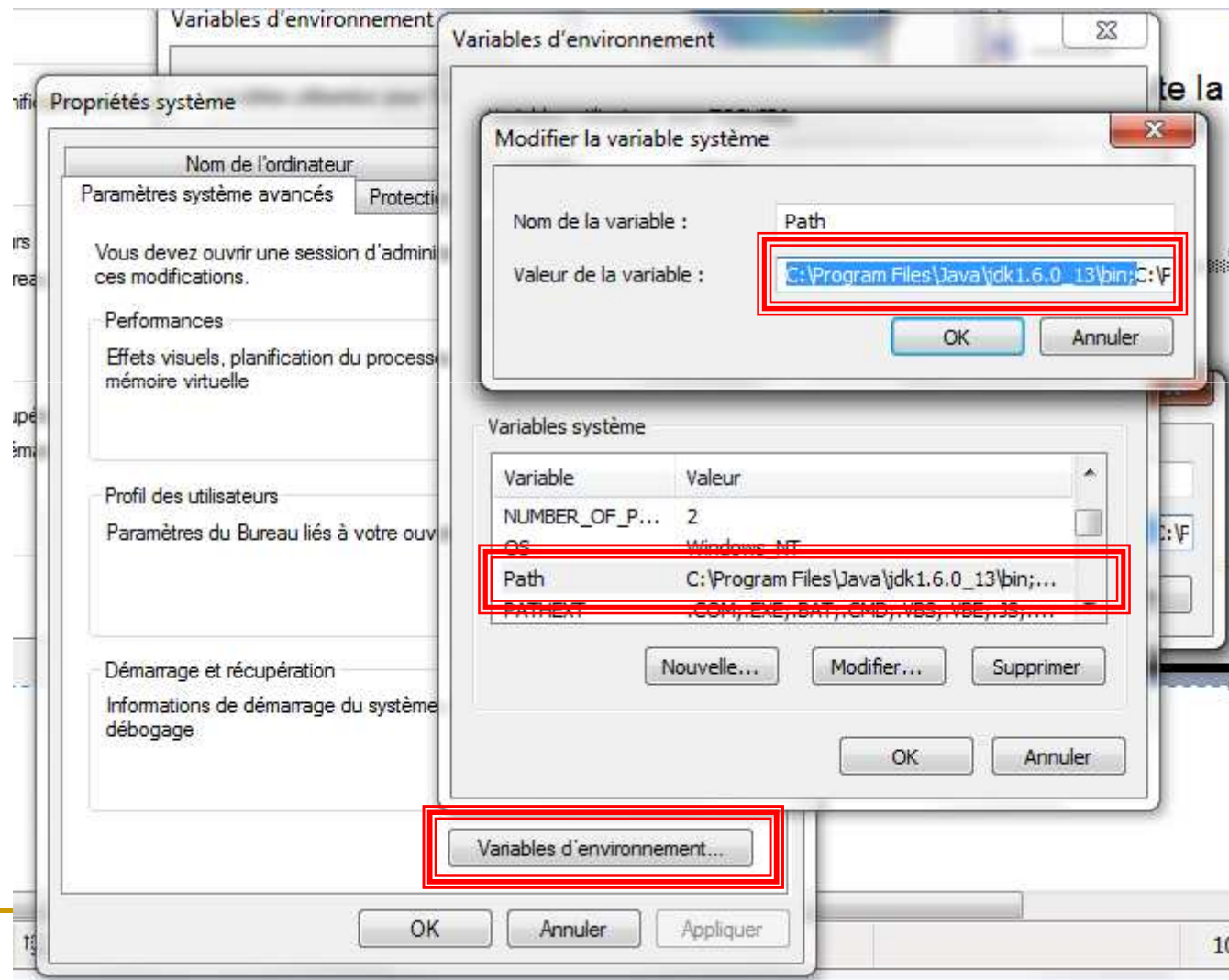
- Les programmes nécessaires au développement java sont placés dans le répertoire c:\jdk1.5\bin à savoir:
  - ❑ **javac.exe** : Compilateur java.
  - ❑ **java.exe** : Interpréteur du bytecode java.
  - ❑ **appletviewer.exe** : Pour tester les applets java.
  - ❑ **Jdb.exe** : Débogueur java.
  - ❑ **Javap.exe** : désassembleur 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 :
  - ❑ JDevlopper 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");  
    }  
}
```

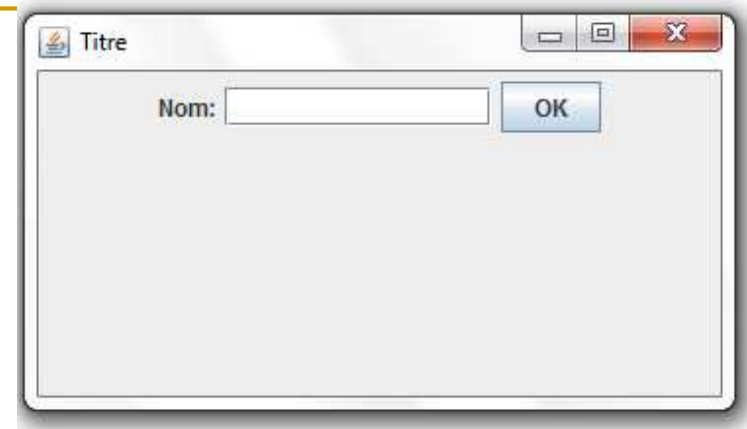
- Lancer un éditeur de texte ASCII et Ecrire le code source de ce programme.
- Enregistrer ce fichier dans un nouveau répertoire c:\exojava sous le nom **PremierProgramme.java**
- Compiler ce programme sur ligne de commande Dos :  
c:\exojava>javac PremierProgramme.java
- Corriger les Erreurs de compilation
- Exécuter le programme sur ligne de commande  
c:\exojava>java PremierProgramme



```
C:\Windows\system32\cmd.exe  
Microsoft Windows [version 6.1.7600]  
Copyright (c) 2009 Microsoft Corporation. Tous droits réservés.  
  
C:\Users\TOSHIBA>cd..  
C:\Users>cd..  
C:\>cd exojava  
C:\exojava>javac PremierProgramme.java  
C:\exojava>java PremierProgramme  
Ca marche !  
C:\exojava>_
```

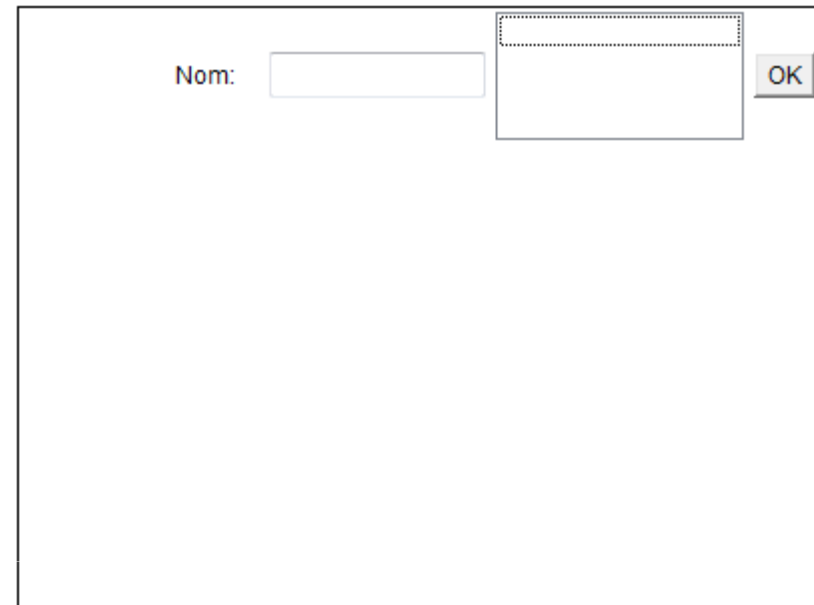
# Première Application graphique

```
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);
}
}
```

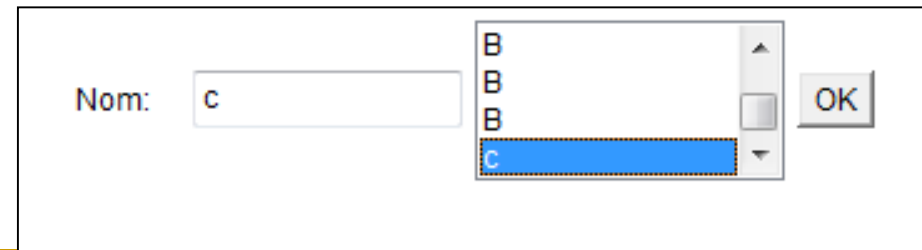


- Rédiger le programme source.
- Enregistrer le fichier sous le nom FirstApplet.java
- Compiler le programme source et corriger les erreurs.
- Créer un page HTML qui affiche l'applet sur un navigateur web:

```
<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, swich, for et while)

# Les primitives

Java dispose des primitives suivantes :

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

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

# 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 décrémentation (`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

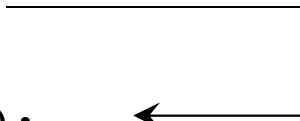
- **Sortie d'une boucle par *return***

```
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



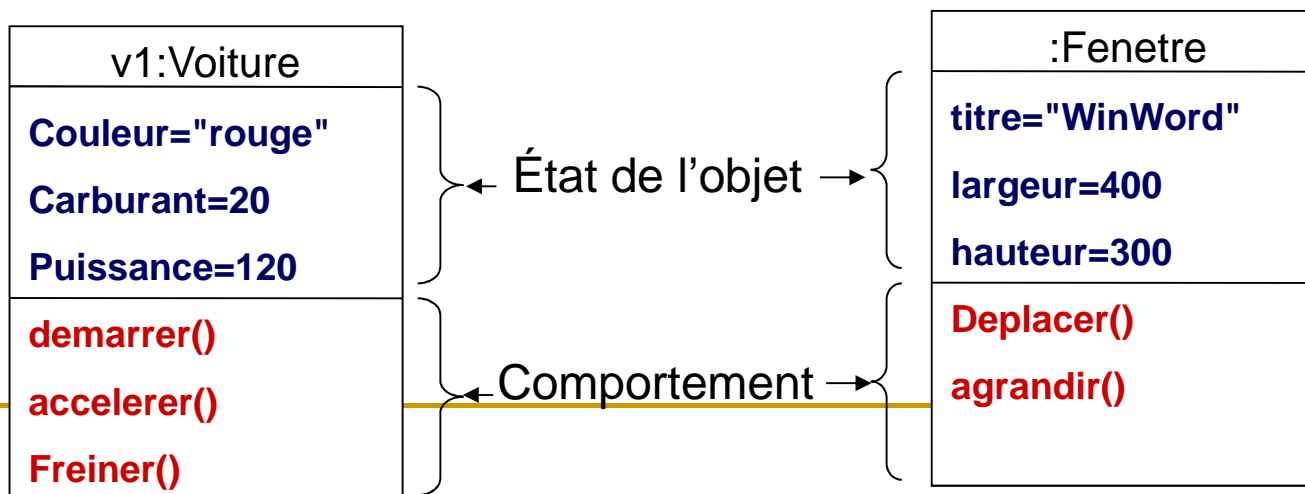
---

# 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 maîtriser 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

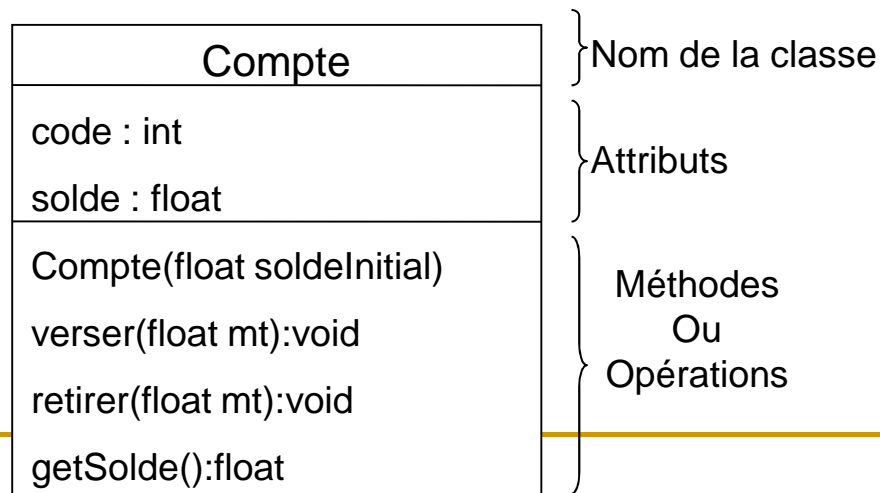
- Les objets qui ont des caractéristiques communes sont regroupés dans une entité appelé classe.
- La classe décrit le domaine de définition d'un ensemble d'objets.
- Chaque objet appartient à une classe
- Les généralités sont contenues dans les classe et les particularités dans les objets.
- Les objets 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 défini 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 méthode 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

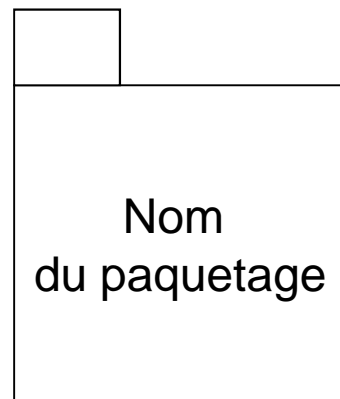
- Une classe est représenté par un rectangle à 3 compartiments:
  - Un compartiment qui contient le nom de la classe
  - Un compartiment qui contient la déclaration des attributs
  - Un compartiment qui contient les méthodes
- Exemples:



---

# 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



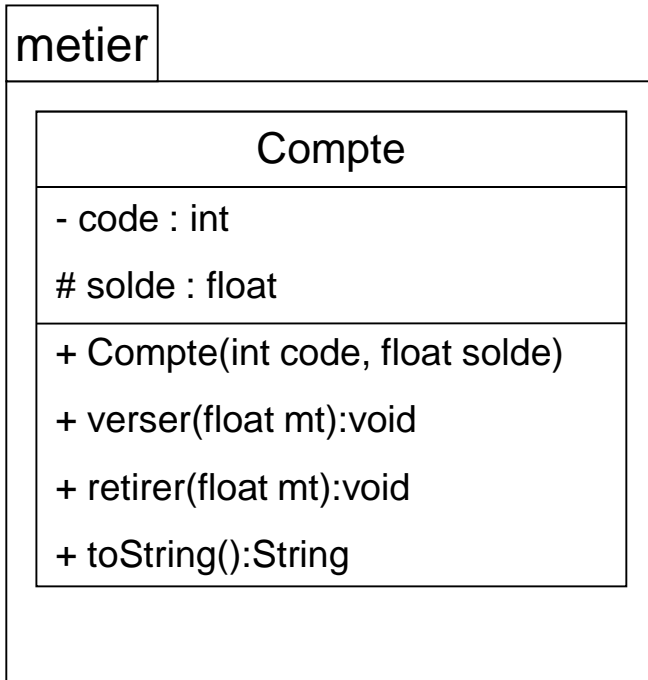
---

# 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());
    }
}
```

Code=1 Solde= 6000

| c1:Compte         |
|-------------------|
| code=1            |
| solde=6000        |
| verser(float mt)  |
| retirer(float mt) |
| toString()        |

| c2:Compte         |
|-------------------|
| code=2            |
| solde=6000        |
| verser(float mt)  |
| retirer(float mt) |
| 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());
```

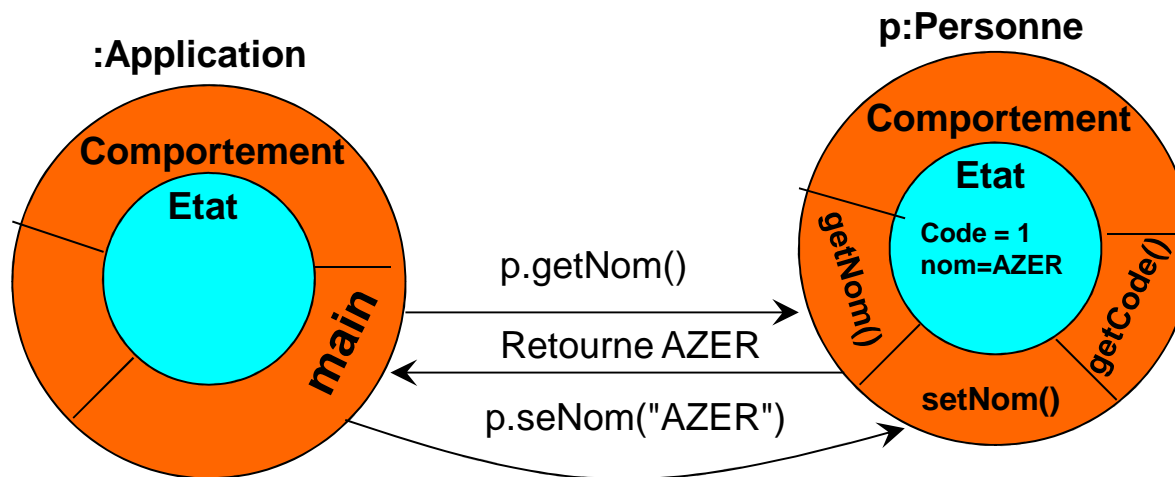
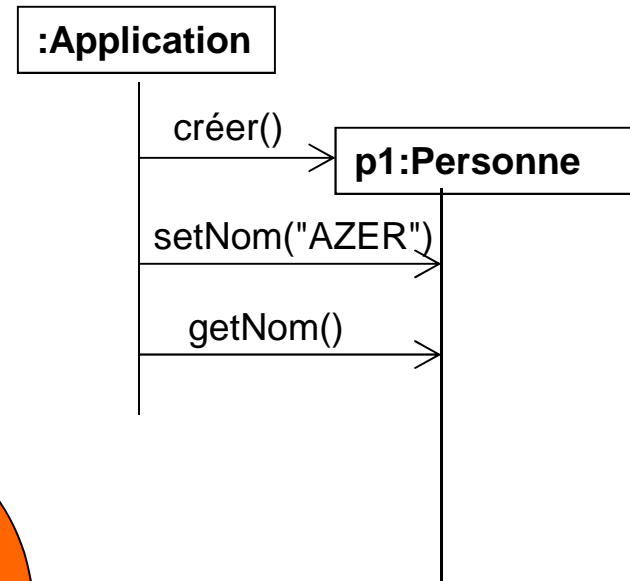
# Getters et Setters

- 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 meme 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 publique
- Quand l'état de l'objet est privé Seules les méthode 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 peuvent également y accéder

---

# 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 le 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<br># solde : float<br>- <u>nbComptes:int</u>  |
| + Compte(float solde)<br>+ verser(float mt):void<br>+ retirer(float mt):void<br>+ toString():String<br>+ <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

nbCompte=2

getNbComptes()

c1:Compte

code=1

solde=6000

verser(float mt)

retirer(float mt)

toString()

c2:Compte

code=2

solde=6000

verser(float mt)

retirer(float mt)

toString()

Code=1 Solde= 6000

2

2

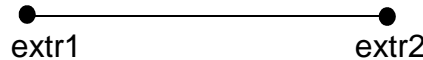


---

# 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 ralentit 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

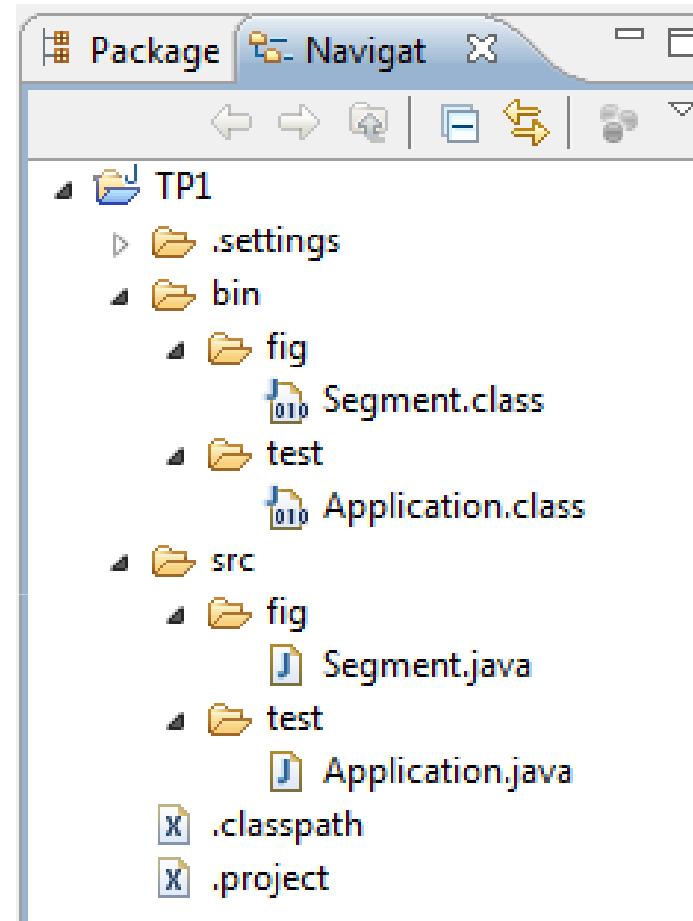
# Diagramme de classes

| 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+"]");
    }
}
```



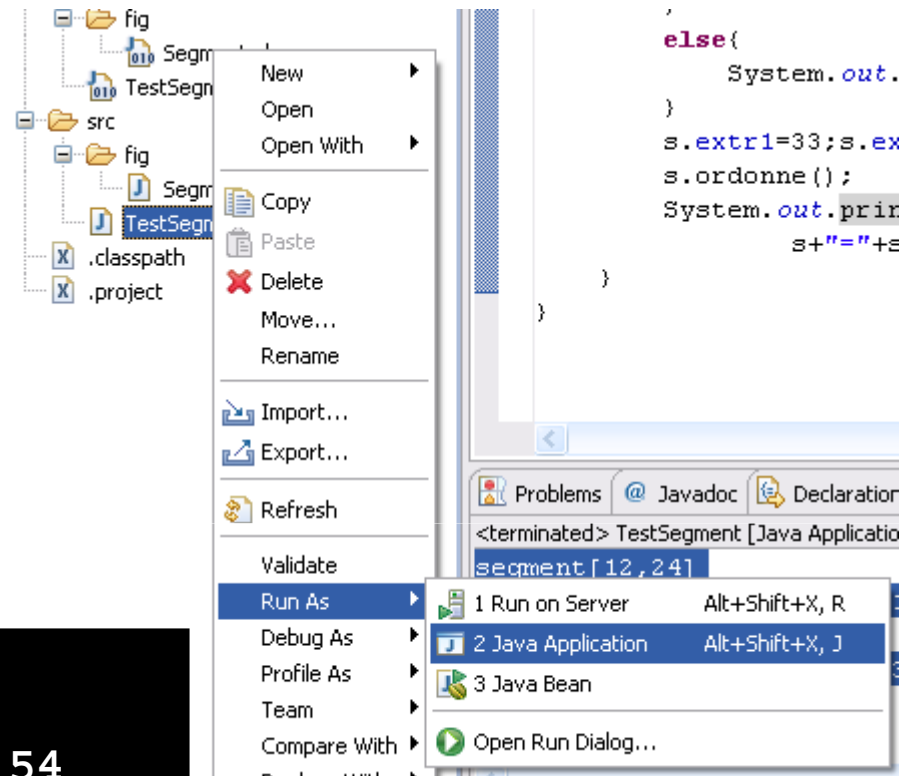
# Application

```
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'appartien pas au "+s);
    }
}
```

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

# Exécution de la classe TestSegment

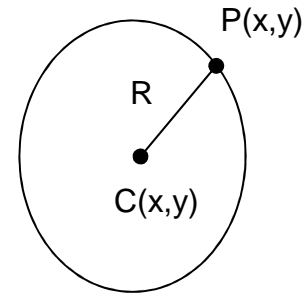
- Pour exécuter une application (Classe qui contient la méthode main) avec Eclipse, on clique avec le bouton droit de la souris sur la classe, puis on choisit dans le menu contextuel, Run As > Java Application



```
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

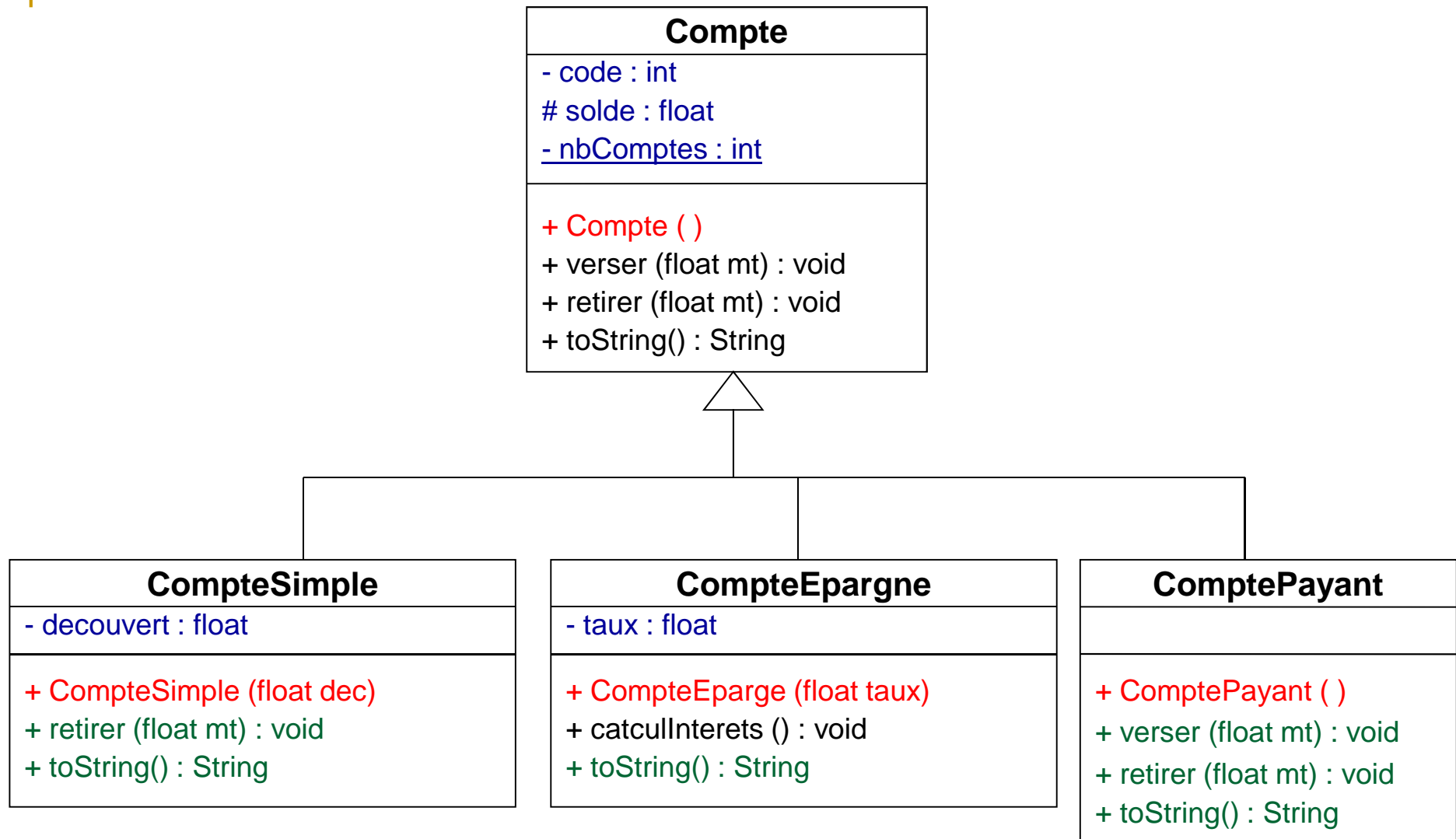
- Dans la programmation orientée objet, l'héritage offre un moyen très efficace qui permet la réutilisation du code.
- En effet une classe peut hériter d'une autre classe des attributs et des méthodes.
- L'héritage, quand il peut être exploité, fait gagner beaucoup de temps en terme de développement 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 souhaitons 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 «tauxInterêt» 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 decouvert;  
    //constructeur  
    public CompteSimple(float decouvert){  
        super();  
        this.decouvert=decouvert;  
    }  
}
```

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

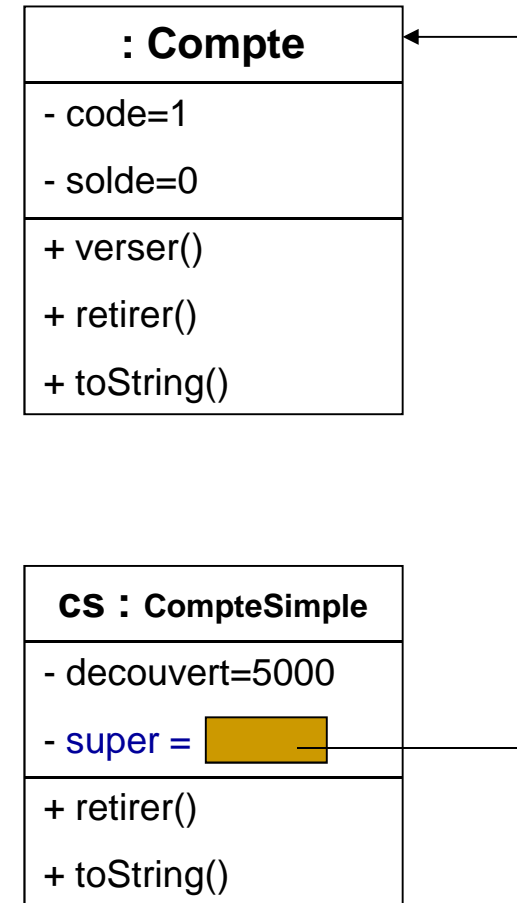


# Héritage à la loupe : Instanciation

- 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 decouvert;  
    //Premier constructeur  
    public CompteSimple(float decouvert){  
        super();  
        this.decouvert=decouvert;  
    }  
    //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 ou 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** »

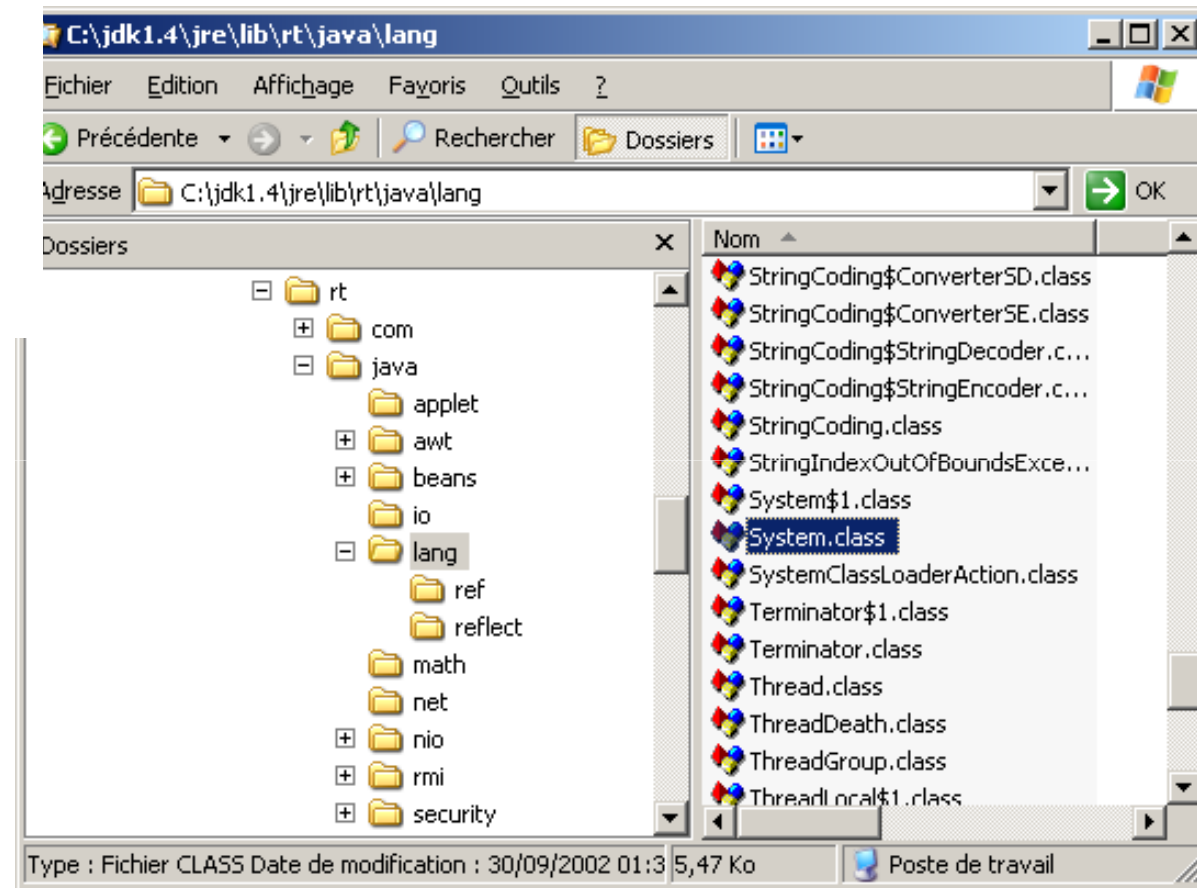
# Les packages (Où)

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:

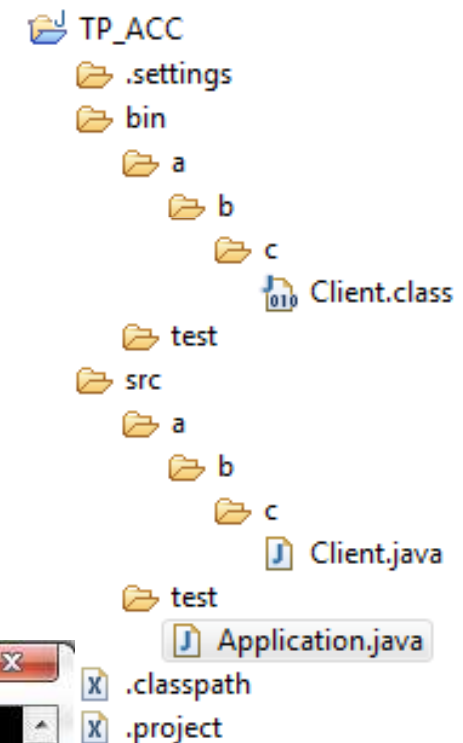
```
package a.b.c;
public class Client {
    private int code;
    private String nom;
    public Client(int code, String nom) {
        this.code = code;
        this.nom = nom;
    }
    public String getNom(){
        return(nom);
    }
    public int getCode(){
        return code;
    }
}
```

```
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:\Windows\system32\cmd.exe
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 getSolde();  
}
```

- 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 getSolde() {  
        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 decouvert;  
    public void afficher() {  
        System.out.println("Solde="+solde+"  
        Découvert="+decouvert);  
    }  
}
```



# 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 instanciation 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 les 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 :

```
3 (fruits) pommes  
+ 5 (fruits) oranges
```

-----

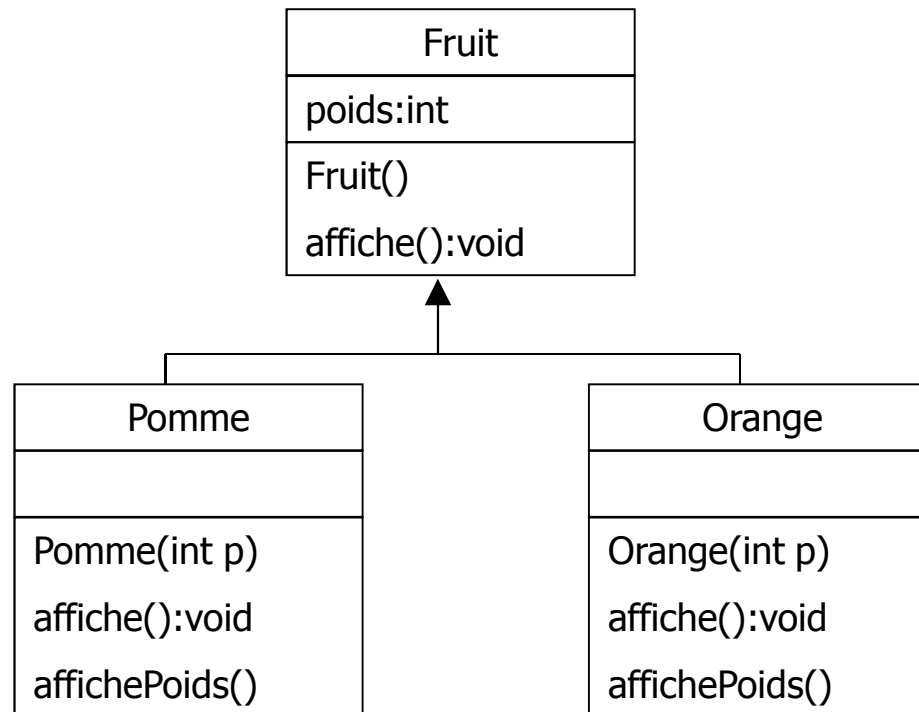
```
= 8 fruits
```

- 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:



# Instanciación 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);
    }
}
```

---

# Instanciación 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 explicite

---

# 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 :

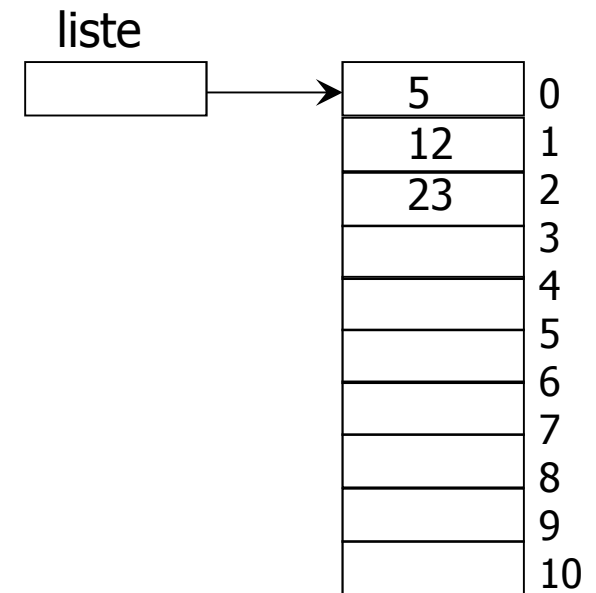
- sa classe,
- toutes les classes parentes de sa classe,
- toutes les interfaces qu'il implémente,
- toutes les interfaces parentes des interfaces qu'il implémente,
- toutes les interfaces qu'implémentent les classes parentes de sa classe,
- toutes les interfaces parentes des précédentes.

---

# Tableaux et Collections

---

# Tableaux de primitives

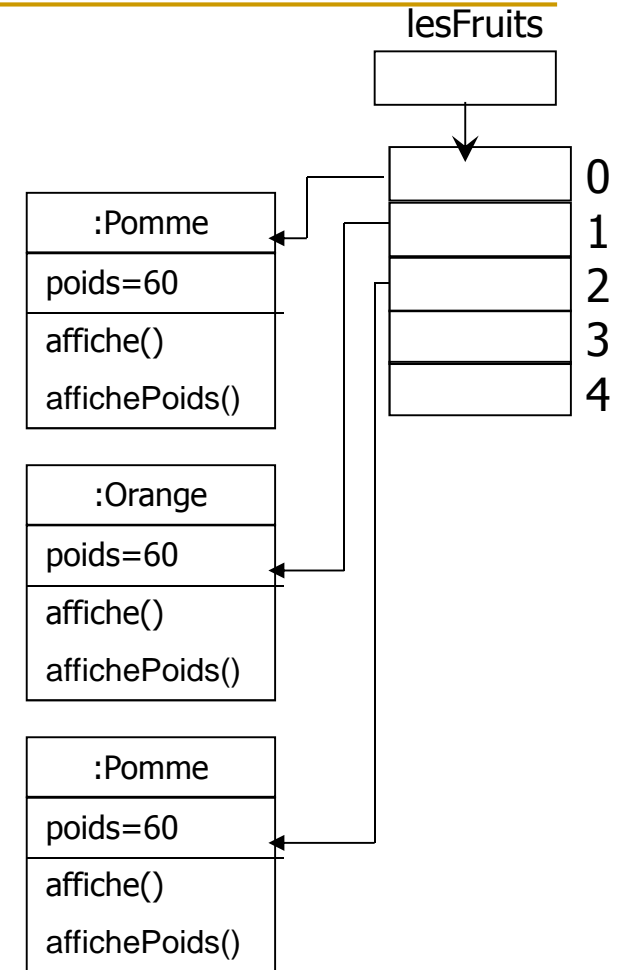


- Tableaux de primitives:
- Déclaration :
  - Exemple : Tableau de nombres entiers
    - `int[] liste;`
  - liste est un handle destiné à pointer vers un tableau d'entier
- Création du tableau
  - `liste = new int[11];`
- Manipulation des éléments du tableau:
  - `liste[0]=5; liste[1]=12; liste[3]=23;`
  - `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 App1 {
    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

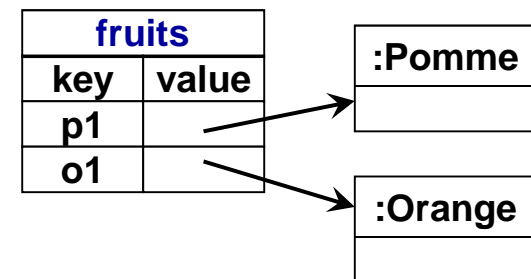
- ❑ `fruits.put("p1", new Pomme(40));`
  - ❑ `fruits.put("o1", new Orange(60));`

- Récupérer un objet ayant pour clé "p1"

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

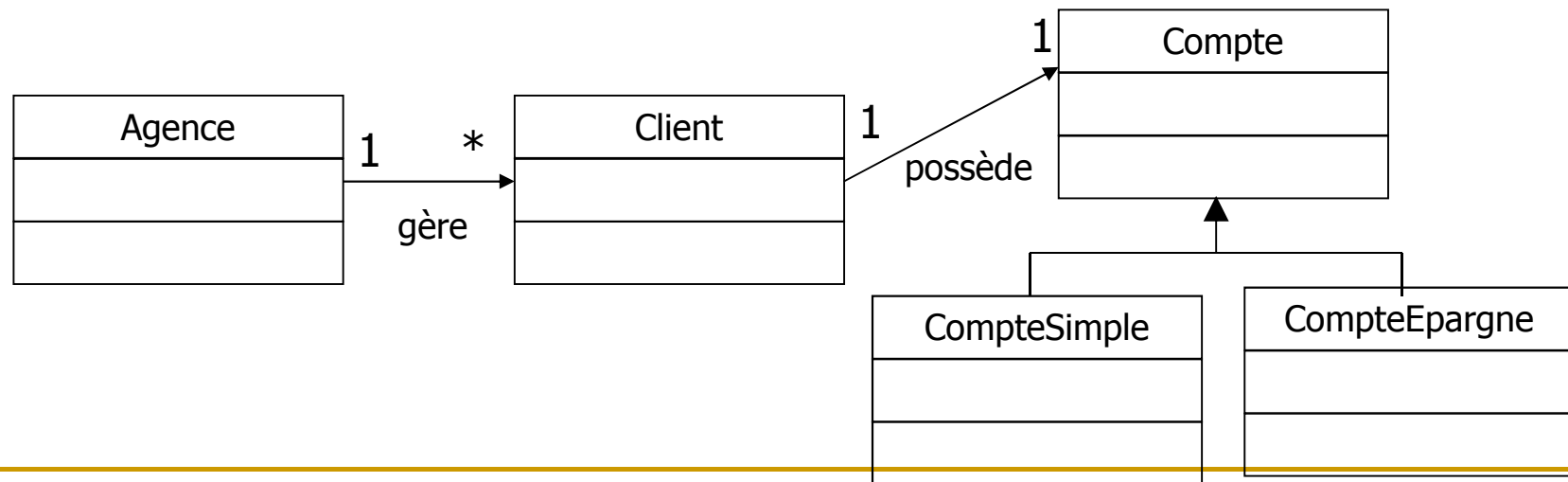
- Parcourir toute la collection:

```
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.

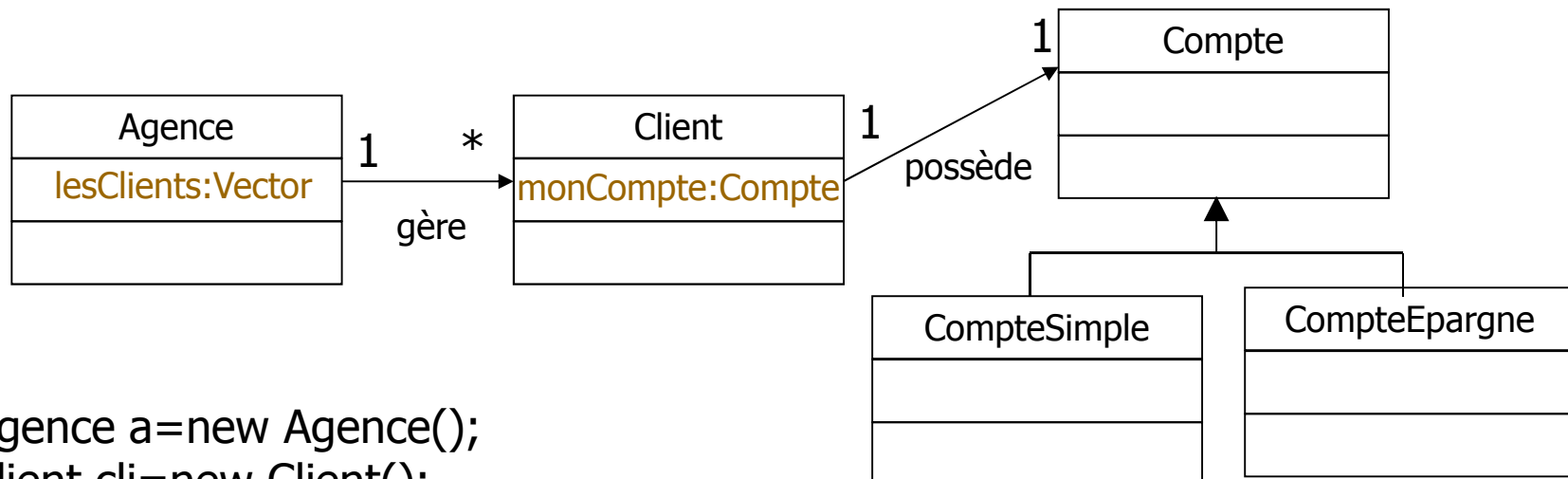


---

# 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);
```

