



# Programmation Orienté Objet Langage JAVA

**Licence professionnelle Génie Informatique  
Semestre 5**

Pr Y. ES-SAADY  
[y.essaady@uiz.ac.ma](mailto:y.essaady@uiz.ac.ma)

**2022/2023**

# Organisation du cours

---

## ■ Nous verrons

- **Rappel: POO de base**
  - Caractéristiques de Java et son environnement de développement
  - Structures fondamentales
  - La programmation par objets en Java
    - Héritage
    - Polymorphisme
- **Rappel: POO Avancée**
  - Les exceptions, les entrées / sorties en Java
  - La programmation des interfaces graphiques en Java
  - Les collections en Java
  - L'accès aux bases de données,.....
  - Les threads
  - Patrons de conception
- Mini projet

# Les outils

---

- La notation UML
- Langage de programmation : Java
- IDE : NetBeans ou autre (Eclipse, IntelliJ, etc.)
- Les tests : Junit
- Versioning : Git et GitHub
- GUI : JavaFX

# Evaluation

---

## ■ Contrôles Continus

- Des travaux à rendre sur classroom
- Projet par équipe de 3 à 4 étudiants
  - les notes seront individuelles
  - but : travailler en équipe,
  - période de réalisation :
  - soutenances le .....

## ■ Examen

- Durée de 1h30min
- questions de cours + exercices

# **Caractéristiques du langage Java**

# Caractéristiques du langage Java (1)

---

## ■ Simple

- Apprentissage facile
  - faible nombre de mots-clés
  - simplifications des fonctionnalités essentielles
- Développeurs opérationnels rapidement
- Pas de notion de pointeurs
- Ni la notion d'héritage multiple et la surcharge des opérateurs

## ■ Familiar

- Syntaxe proche de celle de C/C++

# Caractéristiques du langage Java (2)

---

## ■ Orienté objet

- Java ne permet d'utiliser que des objets (*hors les types de base*)
- Java est un *langage objet* de la famille des langages de *classe* comme C++
- Les grandes idées reprises sont : encapsulation, dualité classe /instance, attribut, méthode / message, visibilité, dualité interface/implémentation, héritage simple, redéfinition de méthodes, polymorphisme
- Chaque fichier source contient la définition d'une ou plusieurs classes qui sont utilisées les unes avec les autres pour former une application.

# Caractéristiques du langage Java (3)

---

## ■ Java est fortement typé

- Toutes les variables sont typées et il n'existe pas de conversion automatique qui risquerait une perte de données.

## ■ Fiable

- Gestion automatique de la mémoire
  - L'allocation de la mémoire pour un objet est **automatique** à sa création et Java récupère automatiquement la mémoire inutilisée suite à la destruction des objets
- Gestion des exceptions
- Sources d'erreurs limitées
  - typage fort, pas d'héritage multiple, pas de manipulations de pointeurs, etc.
- Vérifications faites par le compilateur facilitant une plus grande rigueur du code

# Caractéristiques du langage Java (4)

---

## ■ Java est indépendant de toute plate-forme

- Il est possible d'exécuter des programmes Java sur tous les environnements qui possèdent une Java Virtual Machine.

## ■ JAVA autorise le multitâche (multithreading)-

- Exécution de plusieurs processus effectuant chacun une tâche différente
- Mécanismes de synchronisation
- Fonctionnement sur des machines multiprocesseurs

# Caractéristiques du langage Java (5)

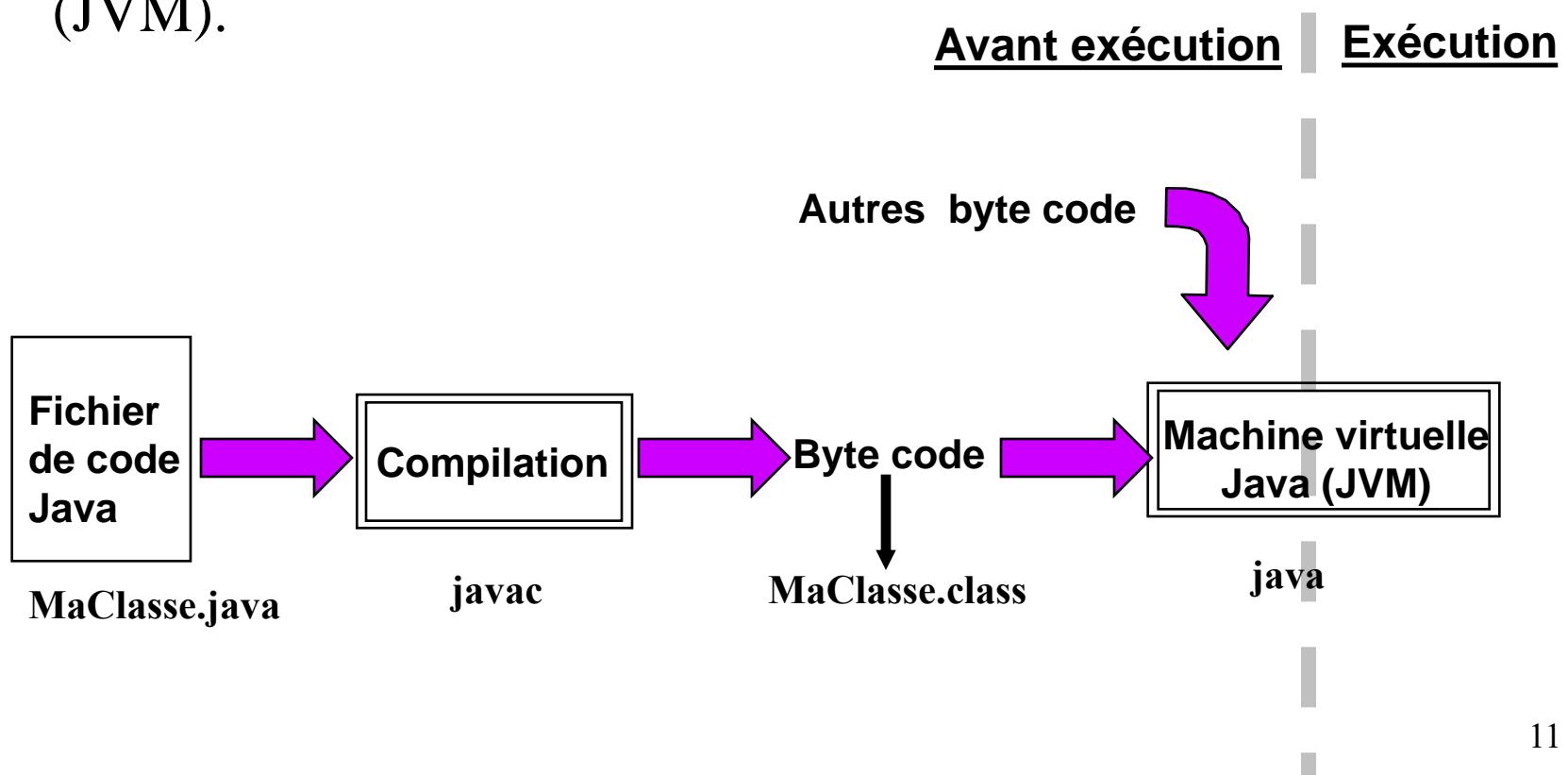
---

- **JAVA contient une très riche bibliothèque de classes (Packages) qui permettent de :**
  - Créer des interfaces graphiques
  - Utiliser les données multimédia
  - Communiquer à travers les réseaux
- **JAVA peut être utilisé sur INTERNET**
  - Les programmes Java s'exécutant sur les pages Web
- **JAVA est gratuit**

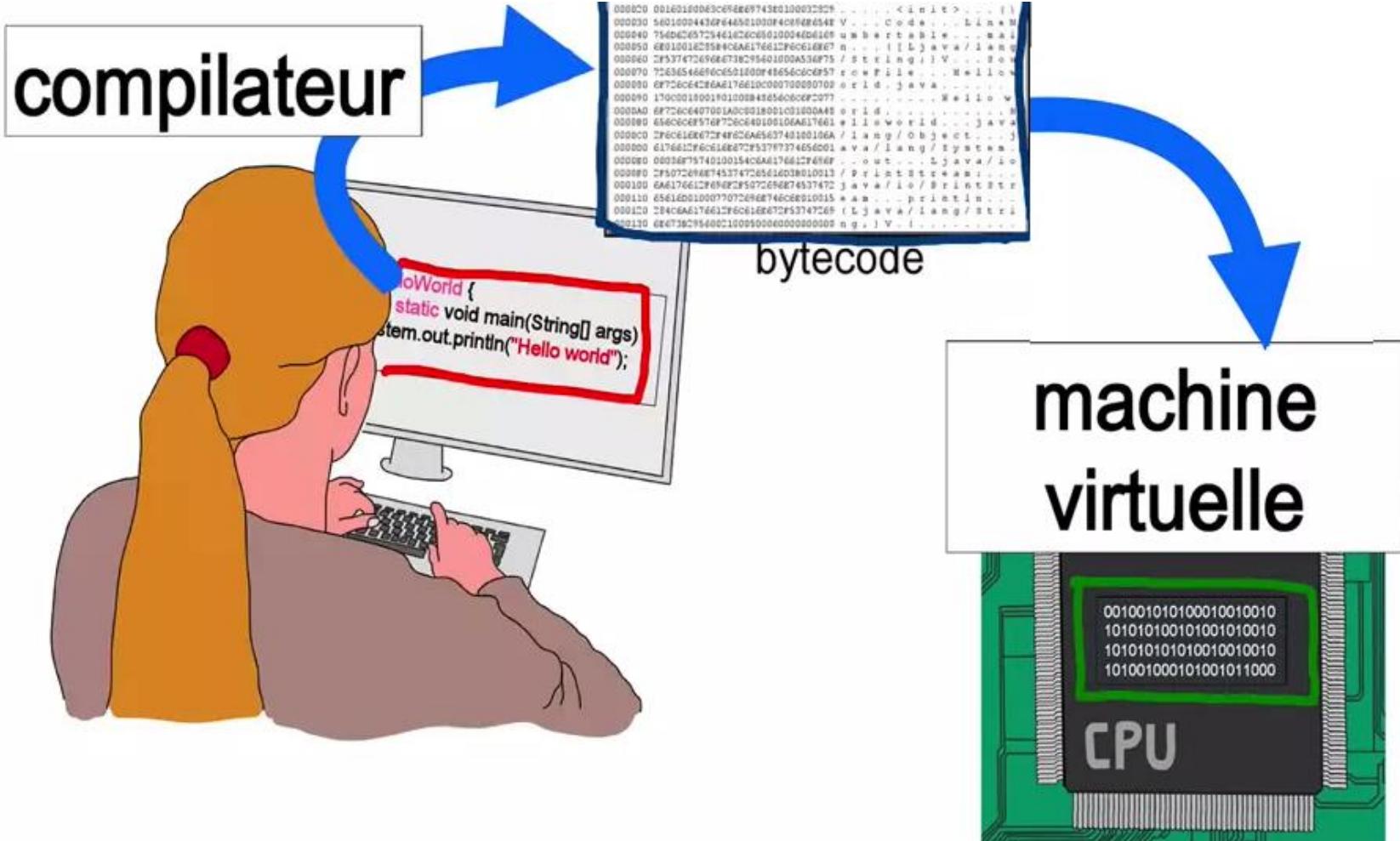
# Caractéristiques du langage Java (6)

## ■ Java est interprété

- Le code source est compilé en pseudo code ou byte code puis exécuté par un interpréteur Java : la Java Virtual Machine (JVM).



# Langage interprété



# L'API de Java

---

- **Java fournit de nombreuses librairies de classes remplissant des fonctionnalités très diverses : c'est l'API Java**
  - API (Application and Programming Interface /Interface pour la programmation d'applications) : Ensemble de bibliothèques permettant une programmation plus aisée car les fonctions deviennent indépendantes du matériel.
- **Ces classes sont regroupées, par catégories, en paquetages (ou "packages").**

# L'API de Java (2)

---

## ■ Les principaux paquetages

- **java.util** : structures de données classiques
- **java.io** : entrées / sorties
- **java.lang** : chaînes de caractères, interaction avec l'OS, threads
- **java.applet** : les applets sur le web
- **java.awt** : interfaces graphiques, images et dessins
- **javax.swing** : package récent proposant des composants « légers » pour la création d'interfaces graphiques
- **java.net** : sockets, URL
- **java.rmi** : Remote Method Invocation (pas abordé dans ce cours)
- **java.sql** : fournit le package JDBC

# L'API de Java (3)

---

- **La documentation de Java est standard, que ce soit pour les classes de l'API ou pour les classes utilisateur**
  - possibilité de génération automatique avec l'outil Javadoc.
- **Elle est au format HTML.**
  - intérêt de l'hypertexte pour naviguer dans la documentation

# L'API de Java (4)

---

- Pour chaque classe, il y a une page HTML contenant :

- la hiérarchie d'héritage de la classe,
- une description de la classe et son but général,
- la liste des attributs de la classe (locaux et hérités),
- la liste des constructeurs de la classe (locaux et hérités),
- la liste des méthodes de la classe (locaux et hérités),
- puis, chacune de ces trois dernières listes, avec la description détaillée de chaque élément.

# L'API de Java (5)

The screenshot shows a web browser displaying the Java Platform, Standard Edition 8 API Specification. The URL in the address bar is [docs.oracle.com/javase/8/docs/api/](https://docs.oracle.com/javase/8/docs/api/). The page has a dark blue header with the title "Java™ Platform Standard Ed. 8". Below the header is a navigation bar with links for OVERVIEW, PACKAGE, CLASS, USE, TREE, DEPRECATED, INDEX, and HELP. There are also links for PREV, NEXT, FRAMES, and NO FRAMES. The main content area features a large heading "Java™ Platform, Standard Edition 8 API Specification" and a sub-heading "This document is the API specification for the Java™ Platform, Standard Edition." Below this, there is a "See: Description" link. A sidebar on the left lists packages such as java.awt, java.awt.color, and java.awt.event. A table titled "Profiles" lists three profiles: compact1, compact2, and compact3. Another table titled "Packages" lists various Java packages with their descriptions.

Package	Description
<a href="#">java.applet</a>	Provides the classes necessary to create an applet and the classes an applet uses to communicate with its applet context.
<a href="#">java.awt</a>	Contains all of the classes for creating user interfaces and for painting graphics and images.
<a href="#">java.awt.color</a>	Provides classes for color spaces.
<a href="#">java.awt.datatransfer</a>	Provides interfaces and classes for transferring data between and within applications.
<a href="#">java.awt.dnd</a>	Drag and Drop is a direct manipulation gesture found in many Graphical User Interface systems that provides a mechanism to transfer information between two entities logically associated with presentation elements in the GUI.
<a href="#">java.awt.event</a>	Provides interfaces and classes for dealing with different types of events fired by AWT components.
<a href="#">java.awt.font</a>	Provides classes and interface relating to fonts.
<a href="#">java.awt.geom</a>	Provides the Java API classes for defining and performing operations on objects.

# Editions Java

---

- Java est disponible dans plusieurs éditions utilisées à des fins diverses.
  - **Java Standard Edition (Java SE)**, Contient les classes qui forment le cœur du langage Java
  - **Java Entreprise Edition (Java EE)** pour le développement d'applications d'entreprise. Elle contient les classes J SE plus d'autres classes pour le développement d'applications d'entreprise.
  - **Java Micro Edition (Java ME)**. pour les produits d'électronique grand public (téléphones portables,...) dans ce cas seule une petite partie du langage est utilisée.
- Chaque édition contient un **kit de développement Java (JDK)** pour développer des applications et un **environnement d'exécution Java (JRE)** pour les exécuter.

# Outil de développement : le JDK

---

- Environnement de développement fourni par Sun
- **JDK** signifie Java Development Kit (Kit de développement Java). Un ensemble de bibliothèques logicielles de base du langage de programmation Java
- Ce kit est librement téléchargeable sur le site web de Sun  
<http://www.oracle.com/technetwork/java/index.html>
- Il contient :
  - les classes de base de l'API java (plusieurs centaines),
  - la documentation au format HTML
  - le compilateur : javac
  - la JVM (machine virtuelle) : java
  - le générateur de documentation : javadoc
  - etc.

# Outil de développement : le JRE

---

- **Le JRE** (Java Runtime Environment) contient uniquement l'environnement d'exécution de programmes Java.
- Le JDK contient lui même le JRE.
- Le JRE seul doit être installé sur les machines où des applications Java doivent être exécutées.

# Editeur de code java: IDE (Integrated Development Environment)

---

- *Jedit* <http://www.jedit.org>



- *Eclipse* <http://www.eclipse.org>



- *NetBeans* <http://www.netbeans.org>



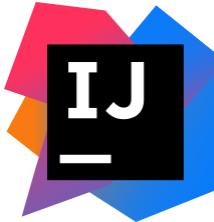
- *Jcreator* <http://www.jcreator.com/>



- *Android Studio*



- *IntelliJ*



# **Bases du langage Java**

# Les commentaires

---

## ■ /\* commentaire sur une ou plusieurs lignes \*/

- Identiques à ceux existant dans le langage C

## ■ // commentaire de fin de ligne

- Identiques à ceux existant en C++

## ■ /\*\* commentaire d'explication \*/

- Les commentaires d'explication se placent généralement juste avant une déclaration (d'attribut ou de méthode)
- Ils sont récupérés par l'utilitaire **javadoc** et inclus dans la documentation ainsi générée.

# Instructions, blocs et blancs

---

- Les instructions Java se terminent par un ;
- Les blocs sont délimités par :
  - { pour le début de bloc
  - } pour la fin du bloc
    - Un bloc permet de définir un regroupement d'instructions.
    - La définition d'une classe ou d'une méthode se fait dans un bloc.
- Les espaces, tabulations, sauts de ligne sont autorisés. Cela permet de présenter un code plus lisible.

# Point d'entrée d'un programme Java

---

- Pour pouvoir faire un programme exécutable il faut toujours une classe qui contienne une méthode particulière, la méthode « main »
  - c'est le point d'entrée dans le programme : le microprocesseur sait qu'il va commencer à exécuter les instructions à partir de cet endroit

```
public static void main(String arg[ ])
{
    ...
}
```

# Exemple (1)

## Fichier **Bonjour.java**

```
public class Bonjour
{
    //Accolade débutant la classe Bonjour

    public static void main(String args[])
    {
        //Accolade débutant la méthode main

        /* Pour l'instant juste une instruction */

        System.out.println("bonjour");

    } //Accolade fermant la méthode main

} //Accolade fermant la classe Bonjour
```

La classe est l'unité de base de nos programmes. Le mot clé en Java pour définir une classe est **class**

# Exemple (2)

## Fichier Bonjour.java

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

Accolades délimitant le début et la fin de la définition de la class Bonjour

Accolades délimitant le début et la fin de la méthode main

Les instructions se terminent par des ;

# Exemple (3)

---

## Fichier Bonjour.java

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

Une méthode peut recevoir des paramètres. Ici la méthode main reçoit le paramètre args qui est un tableau de chaîne de caractères.

# Compilation et exécution (1)

Fichier **Bonjour.java**

Compilation en bytecode java  
dans une console DOS:

La commande: **javac Bonjour.java**  
Génère un fichier **Bonjour.class**  
Exécution du programme  
(toujours depuis la console DOS)  
sur la JVM :

La commande : **java Bonjour**  
Affichage de « bonjour » dans la  
console

Le nom du fichier est nécessairement  
celui de la classe avec l'extension  
.java en plus Java est sensible à la  
casse des lettres.

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

# Compilation et exécution (2)

---

- Pour résumer, dans une console DOS, si j'ai un fichier Bonjour.java pour la classe Bonjour :
  - **javac Bonjour.java**
    - Compilation en bytecode java
    - Indication des erreurs de syntaxe éventuelles
    - Génération d'un fichier Bonjour.class si pas d'erreurs
  - **java Bonjour**
    - Java est la machine virtuelle
    - Exécution du bytecode
    - Nécessité de la méthode main, qui est le point d'entrée dans le programme

# Identificateurs (1)

---

- On a besoin de nommer les classes, les variables, les constantes, etc. ; on parle **d'identificateur**.
- Les identificateurs commencent par une **lettre**, **\_** ou **\$**  
*Attention : Java distingue les majuscules des minuscules*
- Conventions sur les identificateurs :
  - Si plusieurs mots sont accolés, mettre une majuscule à chacun des mots sauf le premier.
    - exemple : **uneVariableEntiere**
  - La première lettre est majuscule pour les classes et les interfaces
    - exemples : **MaClasse**, **UneJolieFenetre**

# Identificateurs (2)

---

## ■ Conventions sur les identificateurs :

- La première lettre est minuscule pour les méthodes, les attributs et les variables
  - exemples : **setLongueur**, **i**, **uneFenetre**
- Les constantes sont entièrement en majuscules
  - exemple : **LONGUEUR\_MAX**

# Les mots réservés de Java

---

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

# Les types de bases (1)

---

- En Java, tout est objet sauf les types de base.
- Il y a huit types de base :
  - un type booléen pour représenter les variables ne pouvant prendre que 2 valeurs (vrai et faux, 0 ou 1, etc.) : **boolean** avec les valeurs associées **true** et **false**
  - un type pour représenter les caractères : **char**
  - quatre types pour représenter les entiers de divers taille : **byte**, **short**, **int** et **long**
  - deux types pour représenter les réels : **float** et **double**
- La taille nécessaire au stockage de ces types est indépendante de la machine.

# Les types de bases (2) : les entiers

---

## ■ Les entiers (avec signe)

Type	Taille (octets)	Valeur minimale	Valeur maximale
byte	1	-128 (Byte.MIN_VALUE)	127 (Byte.MAX_VALUE)
short	2	-32 768 (Short.MIN_VALUE)	32 767 (Short.MAX_VALUE)
int	4	-2 147 483 648 (Integer.MIN_VALUE)	2 147 483 647 (Integer.MAX_VALUE)
long	8	-9 223 372 036 854 775 808 (Long.MIN_VALUE)	9 223 372 036 854 775 807 (Long.MAX_VALUE)

# Les types de bases (3) : les entiers

---

## ■ Notation

- 2: entier normal en base décimal, on utilise les 10 chiffres (0 à 9)
- 2L: entier au format long en base décimal
- **0xF**: entier en valeur hexadécimale (base 16)

## ■ Opérations sur les entiers

- opérateurs arithmétiques +, -, \*
- / :division entière si les 2 arguments sont des entiers
- % : reste de la division entière

### ■ exemples :

- 15 / 4 donne 3
- 15 % 2 donne 1

# Les types de bases (4) : les entiers

## ■ Opérations sur les entiers (suite)

- les opérateurs d’incrémentation **++** et de décrémentation et **--**

- ajoute ou retranche 1 à une variable

int n = 12;

n++; //Maintenant n vaut 13

- n++; « équivalent à » n = n+1;  
n--; « équivalent à » n = n-1;
  - 8++; est une instruction illégale
  - peut s’utiliser de manière suffixée : ++n. La différence avec la version préfixée se voit quand on les utilisent dans les expressions.  
En version suffixée la (dé/inc)rémentation s’effectue en premier

```
int m=7; int n=7;
int a=2 * ++m;
int b=2 * n++;
```

// a =16, m = 8  
// b = 14, n = 8

# Les types de bases (5) : les réels

---

## ■ Les réels

- **float** : codé sur 32 bits, peuvent représenter des nombres allant de  $-10^{35}$  à  $+10^{35}$
- **double** : codé sur 64 bits, peuvent représenter des nombres allant de  $-10^{400}$  à  $+10^{400}$

## ■ Notation

- 4.55 ou 4.55**D** est un réel double précision
- 4.55**f** est un réel simple précision: le **f** signale que c'est un *float*
- 4.25**E4** qui est équivalent à **b=4.25e4** c'est la notation **exponentielle** ou **scientifique** (= 42500.0).

# Les types de bases (6) : les réels

---

## ■ Les opérateurs

- opérateurs classiques +, -, \*, /
- attention pour la division :
  - $15 / 4$  donne 3   ***division entière***
  - $15 \% 2$  donne 1
  - $11.0 / 4$  donne 2.75  
(si l'un des termes de la division est un réel, la division retournera un réel).
- **puissance** : utilisation de la méthode **pow** de la classe Math.
  - **double y = Math.pow(x, a)** équivalent à **x<sup>a</sup>**, x et a étant de type double

# Les types de bases (7) : les booléens

---

## ■ Les booléens

- **boolean**

contient soit vrai (**true**) soit faux (**false**)

## ■ Les opérateurs logiques de comparaisons

- Egalité : opérateur **==**

- Différence : opérateur **!=**

- supérieur et inférieur strictement à : opérateurs **>** et **<**

- supérieur et inférieur ou égal : opérateurs **>=** et **<=**

# Les types de bases (8) : les booléens

---

## ■ Notation

**boolean x;**

**x= true;**

**x= false;**

**x= (5==5);** // l'expression (5==5) est évaluée puis la valeur de l'expression sera affectée à x qui vaut alors vrai

**x= (5!=4);** // x vaut vrai, ici on obtient vrai car 5 est différent de 4

**x= (5>5);** // x vaut faux, 5 n'est pas supérieur strictement à 5

**x= (5<=5);** // x vaut vrai, 5 est bien inférieur ou égal à 5

# Les types de bases (9) : les booléens

---

## ■ Les autres opérateurs logiques

- et logique : **&&**
- ou logique : **||**
- non logique : **!**
- **Exemples** : si a et b sont 2 variables booléennes

```
boolean a,b, c;  
a= true;  
b= false;  
c= (a && b);    // c vaut false  
c= (a || b);     // c vaut true  
c= !(a && b);   // c vaut true  
c=!a;            // c vaut false
```

# Les types de bases (10) : les caractères

---

## ■ Les caractères

- **char** : contient une seule lettre
- le type char désigne des caractères en représentation Unicode
  - Codage sur 2 octets contrairement à ASCII/ANSI codé sur 1 octet. Le codage ASCII/ANSI est un sous-ensemble d'Unicode
  - Notation hexadécimale des caractères Unicode de ‘ \u0000’ à ‘ \uFFFF’.
  - Plus d’information sur Unicode à : [www.unicode.org](http://www.unicode.org)

# Les types de bases (11) : les caractères

---

## ■ Notation

Caractères spéciaux	Affichage
\'	Apostrophe
\"	Guillemet
\	anti slash
\t	Tabulation
\b	retour arrière (backspace)
\r	retour chariot
\f	saut de page (form feed)
\n	saut de ligne (newline)

# Les types de bases (12)

## exemple et remarque

---

```
int x = 0, y = 0;  
float z = 3.1415F;  
double w = 3.1415;  
long t = 99L;  
boolean test = true;  
char c = 'a';
```

### ■ Remarque importante :

En Java, une variable n'ayant pas encore reçu de valeur ne peut pas être utilisée, //erreur de compilation.

### Exemple

```
int n ;  
System.out.println ("n " + n ) ; // erreur de compilation : la valeur de n  
// n'est pas définie ici
```

# Conversion des types primitifs (1)

---

- Il est possible de convertir une valeur codée dans un type donné vers un type de plus grande capacité et ceci sans perte de précision. On peut imaginer la conversion comme ceci:  
**byte -> short -> int -> long -> float -> double**
- La conversion de type de données de la gauche vers la droite (le sens de l'élargissement) s'effectue automatiquement, tandis que dans le sens inverse (ou sens de la restriction), il est nécessaire de passer par une conversion explicite.
- **// Conversion automatique**

```
float x = 10 ;
```

```
double y;
```

```
y=x ;
```

# Conversion des types primitifs (2)

---

## // Conversion explicite

```
float x ;  
double y=10.5;  
x=y ; //erreur  
x=(float)y; // conversion (cast)
```

- Il est également possible de placer des caractères spéciaux (**l, f, et d**) immédiatement après le littéral numérique afin d'effectuer une opération de Conversion.

- **// Conversion d'une valeur numérique en type long**

```
long a = 19999999999999999999L;
```

- **// Conversion d'une valeur numérique en type float**

```
float b = 3.0F;
```

```
float x = b + 24F;
```

# Constantes

---

- Java permet de déclarer que la valeur d'une variable ne doit pas être modifiée pendant l'exécution du programme.
- Les constantes peuvent être déclarées dans les classes ou dans les méthodes en utilisant le mot-clé *final*.

**final int CODE = 1537503;**

**final double PI = 3.14;**

**Final String COULEUR\_NOIRE = "#000000";**

La tentative de modifier une constante dans le programme entraînera une erreur lors de la compilation.

# Les structures de contrôles (1)

---

■ Les structures de contrôle classiques existent en Java :

- **if, else**
- **switch, case, default, break**
- **for**
- **while**
- **do, while**

# Les structures de contrôles (2) : if / else

---

## ■ Instructions conditionnelles

- Effectuer une ou plusieurs instructions seulement si une certaine condition est vraie

**if** (*condition*) *instruction*;

et plus généralement : **if** (*condition*)  
*{ bloc d'instructions }*

*condition doit être un booléen ou renvoyer une valeur booléenne*

- Effectuer une ou plusieurs instructions si une certaine condition est vérifiée sinon effectuer d'autres instructions

**if** (*condition*) *instruction1*; **else** *instruction2*;

et plus généralement **if** (*condition*) *{ 1<sup>er</sup> bloc d'instructions }*  
**else** *{ 2<sup>ème</sup> bloc d'instruction }*

# Les structures de contrôles (3) : if / else

---

Max.java

```
import java.io.*;  
  
public class Max  
{  
    public static void main(String args[])  
    {  
        Console console = System.console();  
        int nb1 = Integer.parseInt(console.readLine("Entrer un entier:"));  
        int nb2 = Integer.parseInt(console.readLine("Entrer un autre entier:"));  
        if (nb1 > nb2)  
            System.out.println("l'entier le plus grand est "+ nb1);  
        else  
            System.out.println("l'entier le plus grand est "+ nb2);  
    }  
}
```

# Les structures de contrôles (4) : while

---

## ■ Boucles indéterminées

- On veut répéter une ou plusieurs instructions un nombre indéterminés de fois : on répète l'instruction ou le bloc d'instruction tant que une certaine condition reste vraie
- nous avons en Java une première boucle while (tant que)
  - **while (*condition*) {*bloc d'instructions*}**
  - les instructions dans le bloc sont répétées tant que la condition reste vraie.
  - On ne rentre jamais dans la boucle si la condition est fausse dès le départ

# Les structures de contrôles (5) : while

---

## ■ Boucles indéterminées

- un autre type de boucle avec le while:

- **do {*bloc d'instructions*} while (*condition*)**
- les instructions dans le bloc sont répétées tant que la condition reste vraie.
- On rentre toujours au moins une fois dans la boucle : la condition est testée en fin de boucle.

# Les structures de contrôles (6) : while

---

Facto1.java

```
import java.io.*;  
  
public class Facto1  
{  
    public static void main(String args[])  
    {  
        int n, result,i;  
        n = Integer.parseInt(System.console().readLine("Entrer une valeur pour n:"));  
        result = 1; i = n;  
        while (i > 1)  
        {  
            result = result * i;  
            i--;  
        }  
        System.out.println("la factorielle de "+n+" vaut "+result);  
    }  
}
```

# Les structures de contrôles (7) : for

---

## ■ Boucles déterminées

- On veut répéter une ou plusieurs instructions un nombre déterminés de fois : on répète l'instruction ou le bloc d'instructions pour un certain nombre de pas.

- **La boucle for**

`for (int i = 1; i <= 10; i++)`

`System.out.println(i); //affichage des nombres de 1 à 10`

- une boucle for est en fait équivalente à une boucle while

`for (instruction1; expression1; expression2) {bloc}`

... est équivalent à ...

`instruction 1;`

`while (expression1)`

`{bloc; expression2}`

# Les structures de contrôles (8) : for

Facto2.java

```
import java.io.*;  
  
public class Facto2  
{  
    public static void main(String args[])  
    {  
        int n, result,i;  
        n = Integer.parseInt(System.console().readLine("Entrer une valeur pour n:"));  
        result = 1;  
        for(i =n; i > 1; i--)  
        {  
            result = result * i;  
        }  
        System.out.println("la factorielle de "+n+" vaut "+result);  
    }  
}
```

# Les structures de contrôles (9) : switch

---

## ■ Sélection multiples

- l'utilisation de if / else peut s'avérer lourde quand on doit traiter plusieurs sélections et de multiples alternatives
- pour cela existe en Java le **switch** / **case** assez identique à celui de C/C++
- La valeur sur laquelle on teste doit être un char ou un entier
- L'exécution des instructions correspondant à une alternative commence au niveau du **case** correspondant et se termine à la rencontre d'une instruction **break** ou arrivée à la fin du **switch**

# Les structures de contrôles (10) : switch

Alternative.java

```
import java.io.*;  
  
public class Alternative  
{  
    public static void main(String args[])  
    {  
        int nb = Integer.parseInt(System.console().readLine("Entrer une valeur pour n:"));  
        switch(nb)  
        {  
            case 1:  
                System.out.println("Un"); break;  
            case 2:  
                System.out.println("Deux"); break;  
            default:  
                System.out.println("Autre nombre"); break;  
        }  
    }  
}
```

Variable contenant la valeur que l'on veut tester.

Première alternative : on affiche *Un* et on sort du bloc du switch au break;

Deuxième alternative : on affiche *Deux* et on sort du bloc du switch au break;

Alternative par défaut: on réalise une action par défaut.

# Les tableaux (1)

---

- **Les tableaux permettent de stocker plusieurs valeurs de même type dans une variable.**
  - Les valeurs contenues dans la variable sont repérées par un indice
  - En langage java, les tableaux sont des objets
- **Déclaration**
  - `int tab [ ];`
  - `String chaines[ ];`
- **Création d'un tableau**
  - `tab = new int [20]; // tableau de 20 int`
  - `chaines = new String [100]; // tableau de 100 chaine`

# Les tableaux (2)

---

- **Le nombre d'éléments du tableau est mémorisé.**
- **Java peut ainsi détecter à l'exécution le dépassement d'indice et générer une exception. Mot clé length**
  - Il est récupérable par `nomTableau.length`

```
int tab = new int [20]; // tableau de 20 int
int taille = tab.length; //taille vaut 20
```
- **Comme en C/C++, les indices d'un tableau commencent à ‘0’.**
- **Donc un tableau de taille 100 aura ses indices qui iront de 0 à 99.**

# Les tableaux (3)

---

## ■ Initialisation

**tab[0]=1;**

**tab[1]=2; //etc.**

**noms[0] = new String( "Ahmed");**

**noms[1] = new String( "Ali");**

## ■ Création et initialisation simultanées

**String noms [ ] = {"Ahmed", " Ali"};**

# Les tableaux (4)

Tab1.java

```
public class Tab1
{
    public static void main (String args[])
    {
        int tab[ ] ;
        tab = new int[4];
        tab[0]=5;
        tab[1]=3;
        tab[2]=7;
        tab[3]=tab[0]+tab[1];
    }
}
```

Les indices vont toujours de 0 à (taille-1)

Pour déclarer une variable tableau on indique le *type* des éléments du tableau et le *nom de la variable tableau* suivi de [ ]

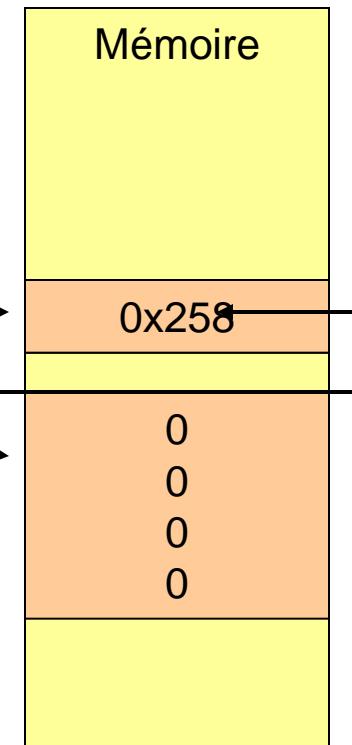
on utilise new <type> [taille]; pour initialiser le tableau

On peut ensuite affecter des valeurs au différentes cases du tableau : <nom\_tableau>[indice]

# Les tableaux (5)

Tab1.java

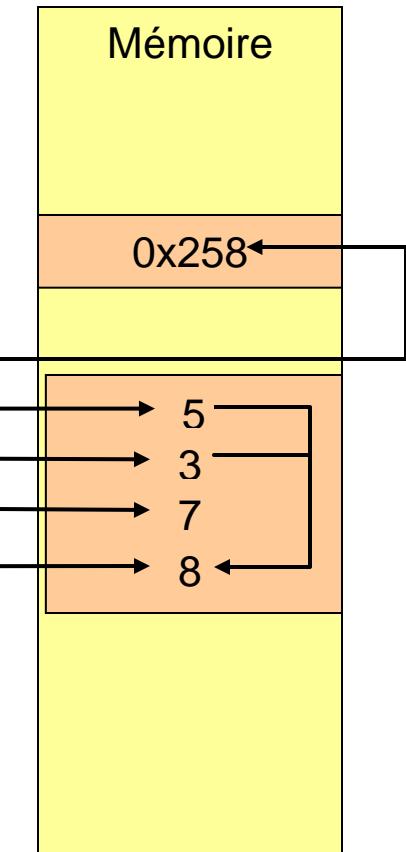
```
public class Tab1
{
    public static void main (String args[])
    {
        int tab[ ] ; _____
        tab = new int[4]; _____
        tab[0]=5;
        tab[1]=3;
        tab[2]=7;
        tab[3]=tab[0]+tab[1];
    }
}
```



# Les tableaux (6)

Tab1.java

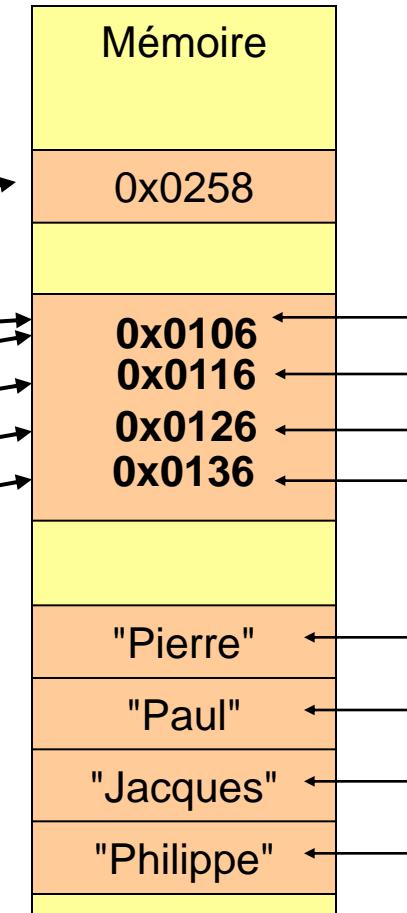
```
public class Tab1
{
    public static void main (String args[])
    {
        int tab[ ] ;
        tab = new int[4];
        tab[0]=5;
        tab[1]=3;
        tab[2]=7;
        tab[3]=tab[0]+tab[1];
    }
}
```



# Les tableaux (7)

Tab2.java

```
public class Tab2
{
    public static void main (String args[])
    {
        String tab[ ] ;
        tab = new String[4];
        tab[0]=new String("Pierre");
        tab[1]=new String("Paul");
        tab[2]=new String("Jacques");
        tab[3]=new String("Philippe");
    }
}
```



# La classe String (1)

---

- Attention ce n'est pas un type de base. Il s'agit d'une classe défini dans l'API Java (Dans le package `java.lang`)

`String s="aaa"; // s contient la chaîne "aaa"`

`String s=new String("aaa"); // identique à la ligne précédente`

- La concaténation

– l'opérateur `+` entre 2 String les concatène :

`String str1 = "Bonjour ! ";`

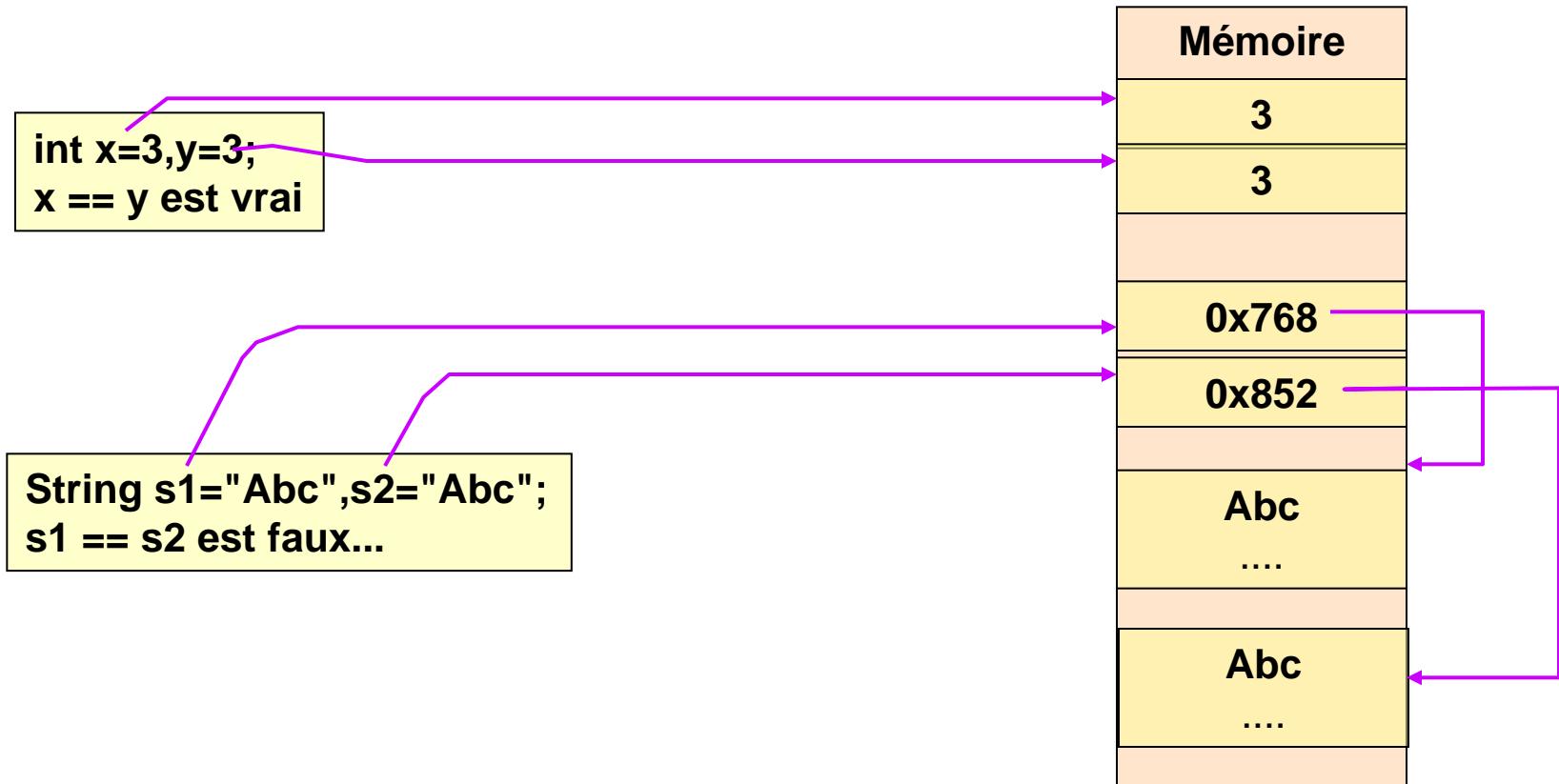
`String str2 = null;`

`str2 = "Comment vas-tu ?";`

`String str3 = str1 + str2; /* Concaténation de chaînes : str3 contient " Bonjour ! Comment vas-tu ?"`

# Différences entre objets et types de base

---



# La classe String (2)

---

## ■ Longueur d'un objet String :

- méthode **int length()** : renvoie la longueur de la chaîne

```
String str1 = "bonjour";
```

```
int n = str1.length(); // n vaut 7
```

## ■ Sous-chaînes

- méthode **String substring(int debut, int fin)**

- extraction de la sous-chaine depuis la position **debut** jusqu'à la position **fin** non-comprise.

```
String str2 = str1.substring(0,3); // str2 contient la valeur "bon"
```

- le premier caractère d'une chaîne occupe la position 0
- le deuxième paramètre de **substring** indique la position du premier caractère que l'on ne souhaite pas copier

# La classe String (3)

---

## ■ Récupération d'un caractère dans une chaîne

- méthode **char charAt(int pos)** : renvoie le caractère situé à la position pos dans la chaîne de caractère à laquelle on envoie ce message

```
String str1 = "bonjour";
```

```
char unJ = str1.charAt(3); // unJ contient le caractère 'j'
```

## ■ Modification des objets String

- Les String sont inaltérables en Java : on ne peut modifier individuellement les caractères d'une chaîne.
- Par contre il est possible de modifier le contenu de la variable contenant la chaîne (la variable ne référence plus la même chaîne).

```
str1 = str1.substring(0,3) + " soir"; /* str1 contient  
maintenant la chaîne "bonsoir" */
```

# La classe String (4)

---

## ■ Les chaînes de caractères sont des objets :

- pour tester si 2 chaînes sont égales il faut utiliser la méthode **boolean equals(String str)** et non **==**
- pour tester si 2 chaînes sont égales à la casse près il faut utiliser la méthode **boolean equalsIgnoreCase(String str)**

```
String str1 = "BonJour";  
String str2 = "bonjour"; String str3 = "bonjour";  
boolean a, b, c, d;  
a = str1.equals("BonJour"); //a contient la valeur true  
b = (str2 == str3); //b contient la valeur false  
c = str1.equalsIgnoreCase(str2); //c contient la valeur true  
d = "bonjour".equals(str2); //d contient la valeur true
```

# La classe String (5)

---

## ■ Quelques autres méthodes utiles

- boolean **startsWith(String str)** : pour tester si une chaîne de caractère commence par la chaîne de caractère str
- boolean **endsWith(String str)** : pour tester si une chaîne de caractère se termine par la chaîne de caractère str

```
String str1 = "bonjour ";
boolean a = str1.startsWith("bon"); //a vaut true
boolean b = str1.endsWith("jour"); //b vaut true
```

# La méthode static : String.valueOf

La méthode **String.valueOf** est une méthode statique permettant de convertir tous les types de données en String.

```
/public static String valueOf( int i );
/public static String valueOf( byte b );
/public static String valueOf( short s );
/public static String valueOf( boolean b );
/public static String valueOf( long l );
/public static String valueOf( float f );
/public static String valueOf( double d );
/public static String valueOf( char c );
/public static String valueOf( char[] cTab);
/public static String valueOf( Object obj );
```

```
int n = 427;
String ch = String.valueOf(n);
// fourni une chaîne obtenue par formatage de
// la valeur contenue dans n, soit ici "427"
ch = "" + n;
// utilisation artificielle d'une chaîne vide
// pour pouvoir recourir à l'opérateur "+"
```

# Chaînes de caractères

## Comparaison

---

### Chaine1.compareTo(chaine2)

- Permet de savoir si la première chaîne est avant ou après la deuxième au sens lexicographique.
- Le résultat de la comparaison est:
  - ✓ Une valeur **négative** si la chaîne1 est avant la chaîne2;
  - ✓ Une valeur **positive** si la chaîne1 est après la chaîne2;
  - ✓ **0** si les deux chaînes sont égales

# Comparaison de chaînes (suite)

---

```
public int compareTo(String s);
```

```
String s0 = new String( "Bénin" );
String s1 = new String( "Québec" );
String s2 = new String( "Zimbabwe" );
```

```
System.out.println( s0.compareTo(s1) );
System.out.println(s1.compareTo("Québec"));
System.out.println( s2.compareTo(s1) );
```



-15 < 0
0
9 > 0

# Chaînes de caractères

## Conversion d'une chaîne en un type primitif

---

- Pour convertir une chaîne en un entier de type **int**, on utilisera la méthode statique **parseInt** de la classe enveloppe **Integer**, comme ceci

```
String ch = "3587"  
int n = Integer.parseInt(ch);
```

**Byte.parseInt**,  
**Short.parseInt**,  
**Integer.parseInt**,  
**Long.parseLong**,  
**Float.parseFloat**,  
**Double.parseDouble**

# Méthodes diverses

---

```
public String trim();
```

```
String s0 = "_____ chaines avec des espaces _____";
```

```
System.out.println( "_" +s0+ "_" );
```

```
System.out.println( "_" +s0.trim()+"_" );
```



```
| chaines avec des espaces |  
|chaines avec des espaces!|
```

# La classe Math

---

- Les fonctions mathématiques les plus connues sont regroupées dans la classe Math qui appartient au package `java.lang`
  - les fonctions trigonométriques
  - les fonctions d'arrondi, de valeur absolue, ...
  - la racine carrée, la puissance, l'exponentiel, le logarithme, etc.

- Ce sont des méthodes de classe (`static`)

double calcul = **Math.sqrt** (**Math.pow**(5,2) + **Math.pow**(7,2));

double **sqrt**(double x) : racine carrée de x

double **pow**(double x, double y) : x puissance y

# **CLASSES ET OBJETS**

# Programmation Orientée Objet (POO)

- Trouver une solution logicielle passe par la description du problème à l'aide d'un langage donné (analyse, décomposition, modélisation, ...):

- Assembleur,
  - Procédural (C ou Fortran),
  - Objet,

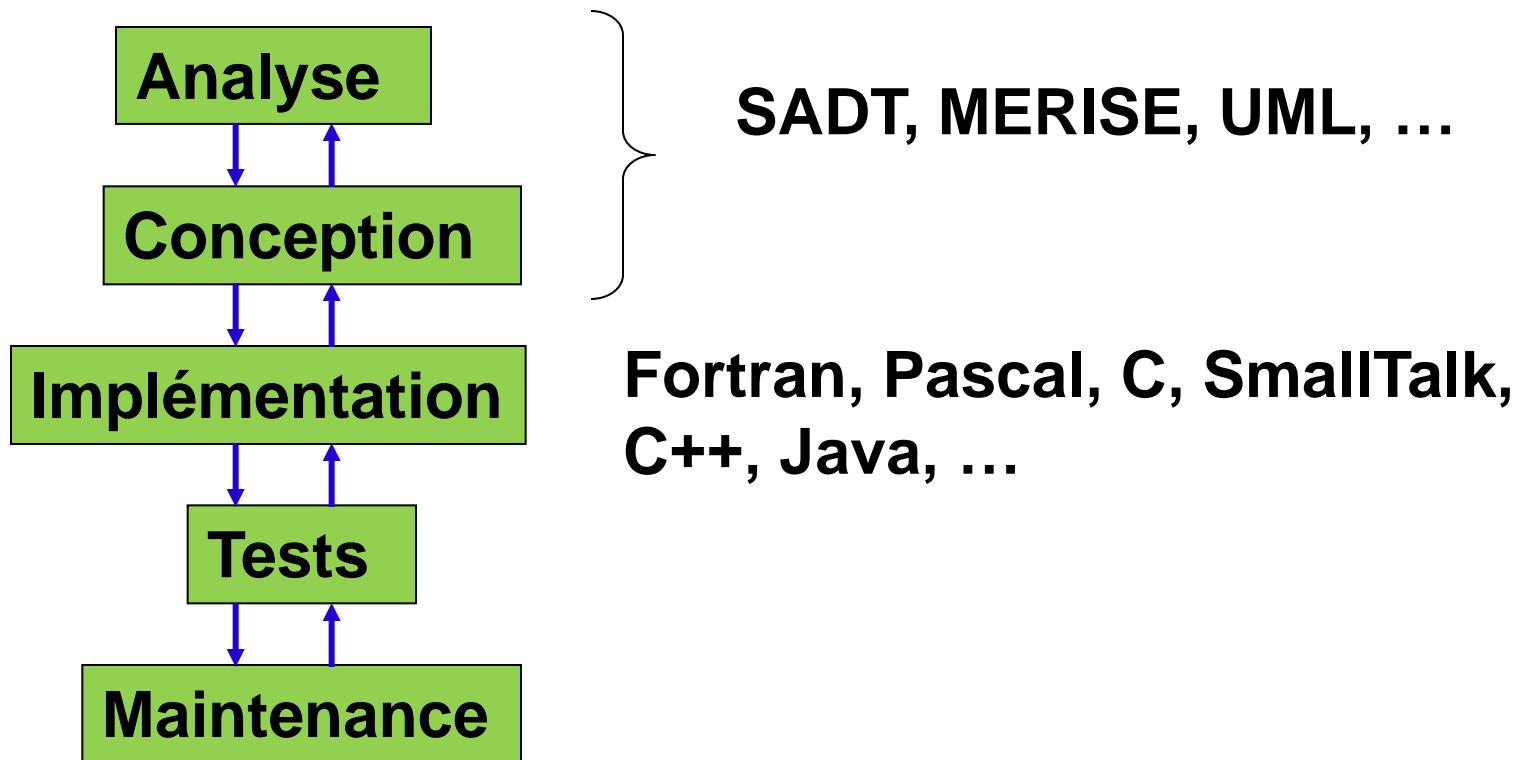
```
mov ax,500    ;  
mov bx,0x20   ;  
add ax,bx    ;  
  
int x = 500;  
int y = 32;  
x+=y;
```

```
licence.setNombreEtudiants(5  
00);  
licence.ajoutEtudiant(32);
```

- La progression de l'abstraction
- La POO permet de décrire le problème à l'aide de termes liés au problème et non à la machine.
- L'unité est l'objet.

# Modèle en cascade pour le développement logiciel

---



# Langages procéduraux et fonctionnels

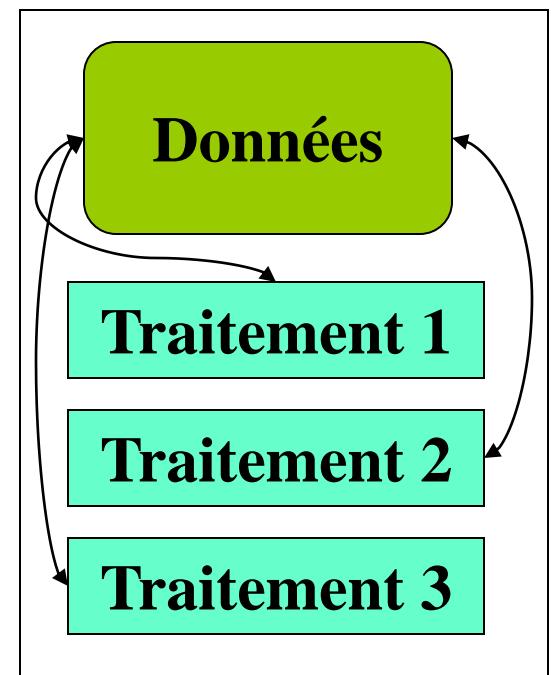
---

Un programme est composé de plusieurs procédures (ou fonctions) :

- qui effectuent un traitement sur des données (**procédure**)
- qui retournent une valeur après leur invocation (**fonction**)

Certains langages ne distinguent pas procédures et fonctions.

Exemples de langages procéduraux ou fonctionnels : Fortran, Lisp, C, ...



# Langages Orientés-Objet

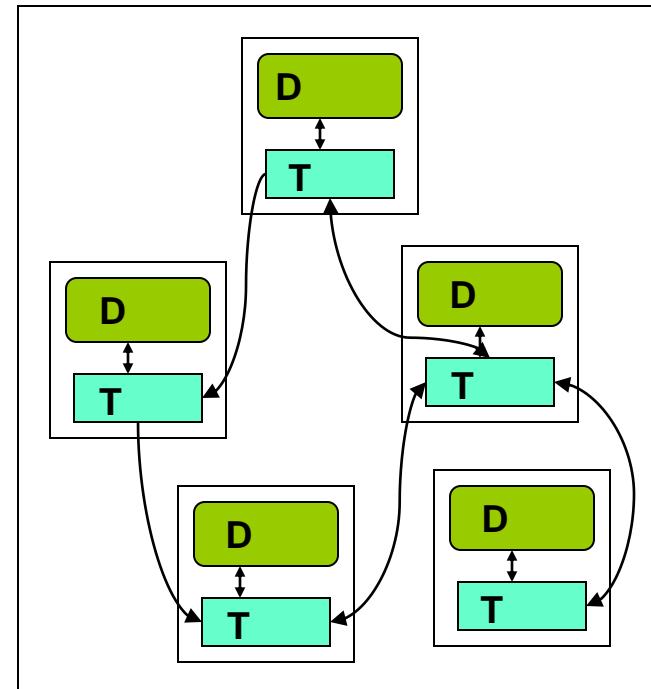
---

Un programme est composé de plusieurs objets qui contiennent :

- des données "internes"
- des traitements manipulant ces données internes ou d'autres données

Les données d'un objet sont appelés ses **attributs** et ses traitements sont ses **méthodes** (ou opérations).

Exemples de langages orientés-objet : SmallTalk, C++, Java, Python, C#, VB.Net, ...



# L'approche objet

---

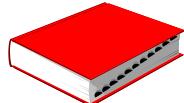
- Idée de base de l'Approche Orientée Objet (A.O.O.) repose sur **l'observation** de la façon dont nous procédons dans notre vie de tous les jours.
- **Qu'est-ce qu'un objet ?**
  - Le monde qui nous entoure est composé d'objets
  - Ces objets ont tous deux caractéristiques
    - un état
    - un comportement
- **Objet = Attributs + méthodes**
- **Ex. un objet "Télévision"**
  - **ses états** ={ allumé/éteint, chaîne courante, volume de son, marque },
  - **ses comportements** ={ allumer, changer de chaîne, changer de volume de son, éteindre }



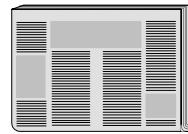
# Objet

- **Approche procédurale :**  
"Que doit faire mon programme ?"
- **Approche orientée-objet :**  
"De quoi doit être composé mon programme ?"
- **Cette composition est conséquence d'un choix de modélisation fait pendant la conception**

## Exemple: Gestion d'une bibliothèque



Germinal  
Émile Zola



Le Monde



Le seigneur des anneaux  
J.R.R.Tolkien



Le Matin



Alice Dupont  
Directrice



Michel Martin  
Bibliothécaire



Arsène Deschamps  
Lecteur



Anne Durand  
Lectrice

# Classe

Des objets similaires peuvent être informatiquement décrits par une même abstraction : **une classe**

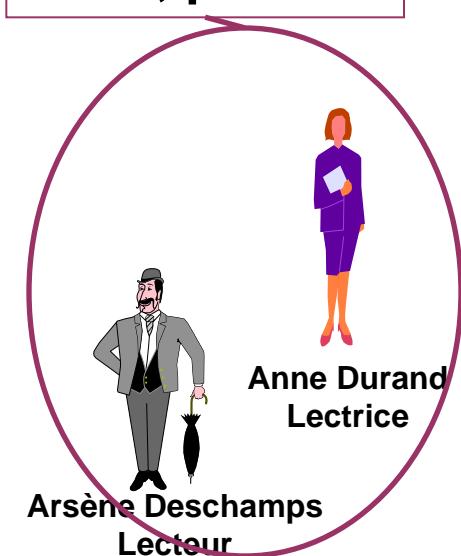
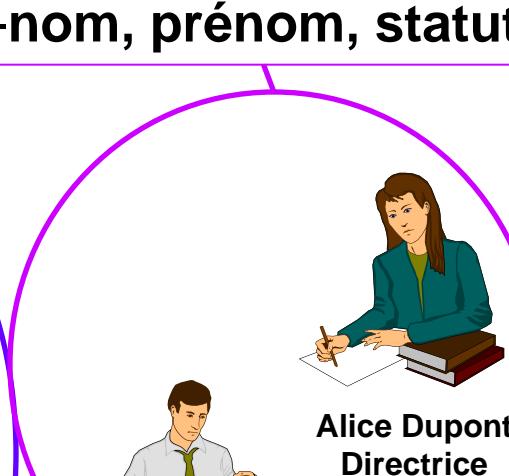
- même structure de données et méthodes de traitement
- valeurs différentes pour chaque objet

**Classe Livre**  
-titre, auteur

**Classe Journal**  
-titre

**Classe Employé**  
-nom, prénom, statut

**Classe Lecteur**  
-nom, prénom



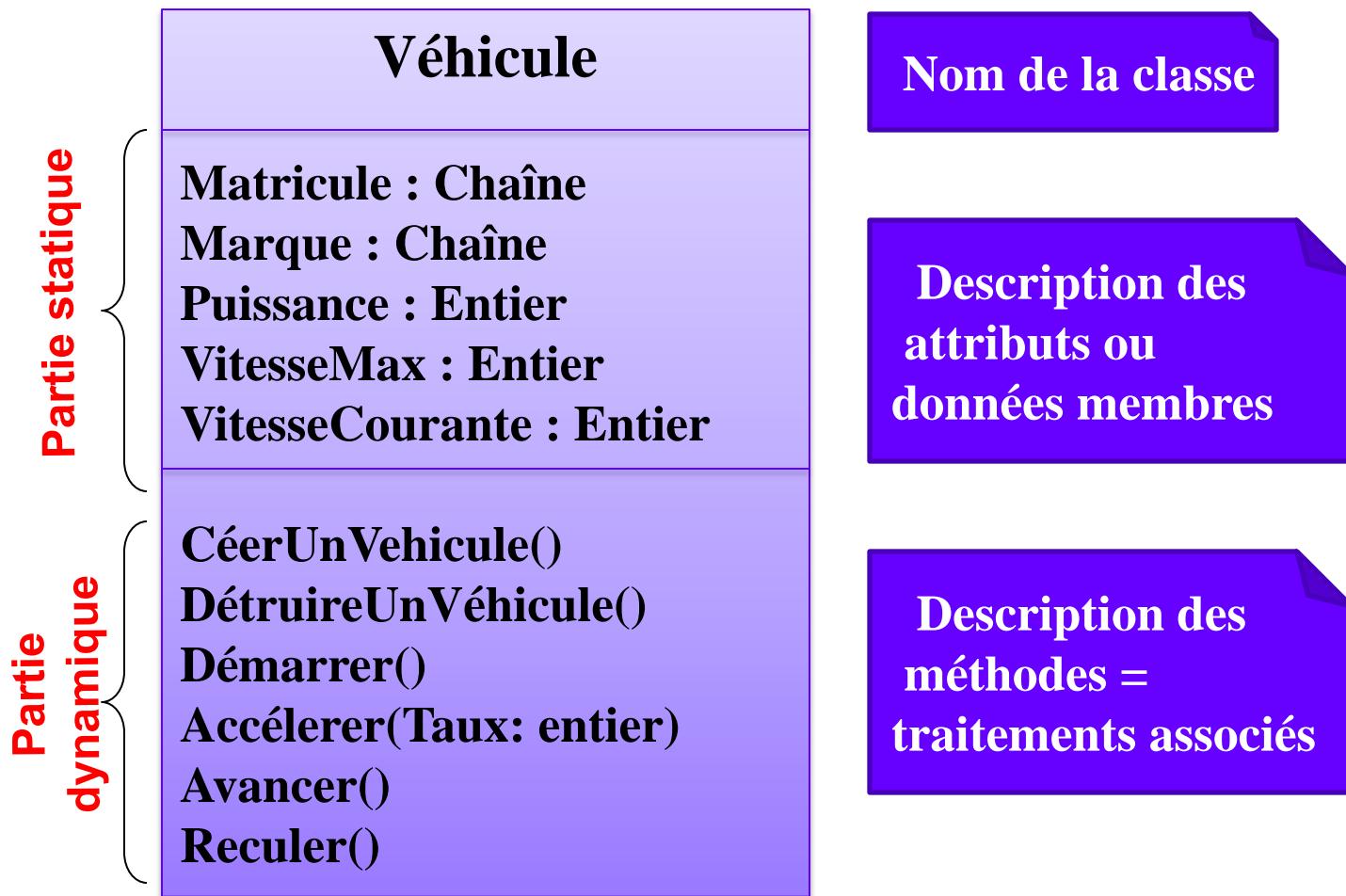
# Contenu d'une classe

Une classe est composée de plusieurs membres dont chacun est soit :

- un attribut : variable typée
- une méthode (ou opération) : ensemble d'instructions de traitement

```
class CompteBancaire {  
    String proprietaire;  
    double solde;  
  
    double getSolde() {  
        return solde;  
    }  
  
    void credite(double val) {  
        solde = solde + val;  
    }  
}
```

# Contenu d'une classe



# Contenu d'une classe

---

```
public class etudiant {  
    String nom;  
    int promo;  
    String traiterExam(String enonce)  
    {...}  
}
```

Attributs

```
public class prof {  
    String nom;  
    String faireSujetExam()  
    {...}  
    int corrigerExam(String copie)  
    {...}  
}
```

Méthodes

# Concept de classe

---

- En informatique, la **classe est un modèle** décrivant les caractéristiques communes et le comportement d'un ensemble d'objets : la classe est un **moule** et l'objet est ce qui est moulé à partir de cette classe
- Mais l'état de chaque objet est indépendant des autres
  - Les objets sont des représentations dynamiques (appelées instances) du modèle défini au travers de la classe
  - Une classe permet d'instancier plusieurs objets
  - Chaque objet est instance d'une seule classe

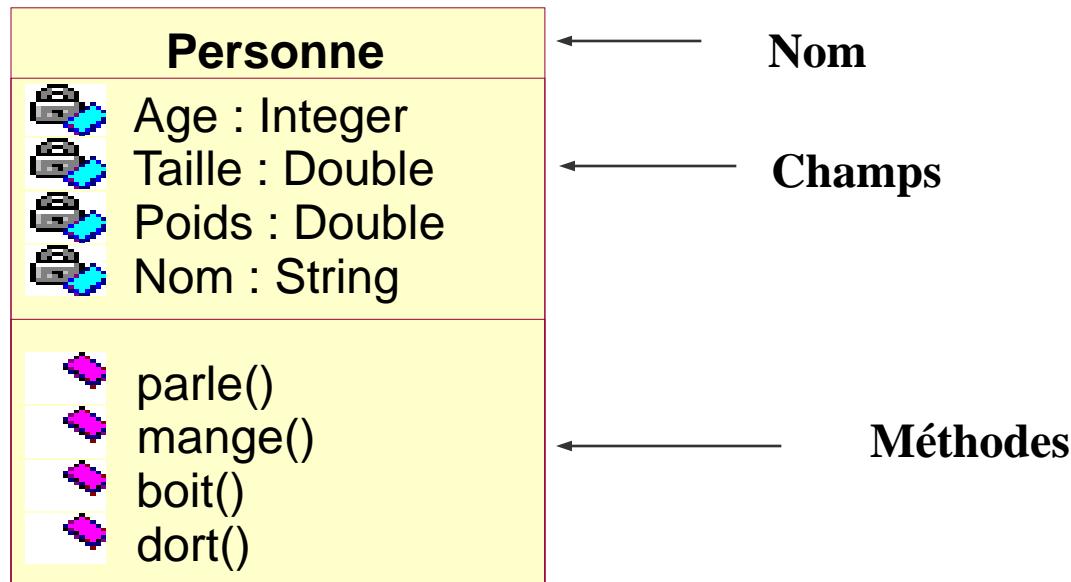
# La classe : définition

---

- **Classe : description d'une famille d'objets ayant une même structure et un même comportement. Elle est caractérisée par :**
  - Un nom
  - Une composante statique : des **champs** (ou **attributs**) nommés ayant une valeur. Ils caractérisent l'état des objets pendant l'exécution du programme
  - Une composante dynamique : des **méthodes** représentant le comportement des objets de cette classe. Elles manipulent les champs des objets et caractérisent les actions pouvant être effectuées par les objets.

# La classe : représentation graphique

---



Une classe représentée avec la notation UML  
(Unified Modeling Language)

# Syntaxe de définition d'une classe

## Exemple : Une classe définissant un point

Class Point

Nom de la Classe

```
{      float x; // abscisse du point
        float y; // ordonnée du point
        void initialiser(float X, float Y){
            x=X;
            y=Y;
        }
        void afficher(){
            System.out.println("x="+x+"\ty="+y);
        }
}
```

Attributs

Méthodes

# L'instanciation

---

## ■ Instance

- représentant physique d'une classe
- obtenu par moulage du dictionnaire des variables et détenant les valeurs de ces variables.
- Son comportement est défini par les méthodes de sa classe
- Par abus de langage « instance » = « objet »

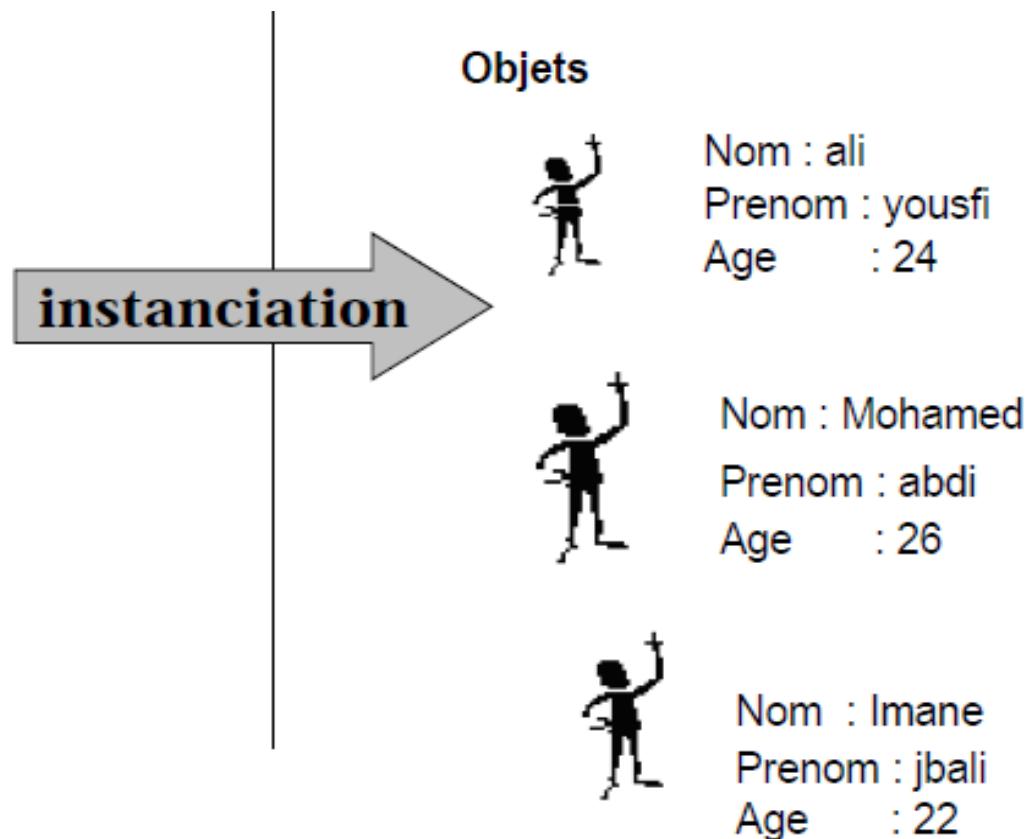
## ■ Exemple :

- si nous avons une classe voiture, alors votre voiture est une instance particulière de la classe voiture.
- Classe = concept, description
- Objet = représentant *concret* d'une classe

# L'instanciation : Exemple

- Une classe peut être réutilisée pour instancier plusieurs objets

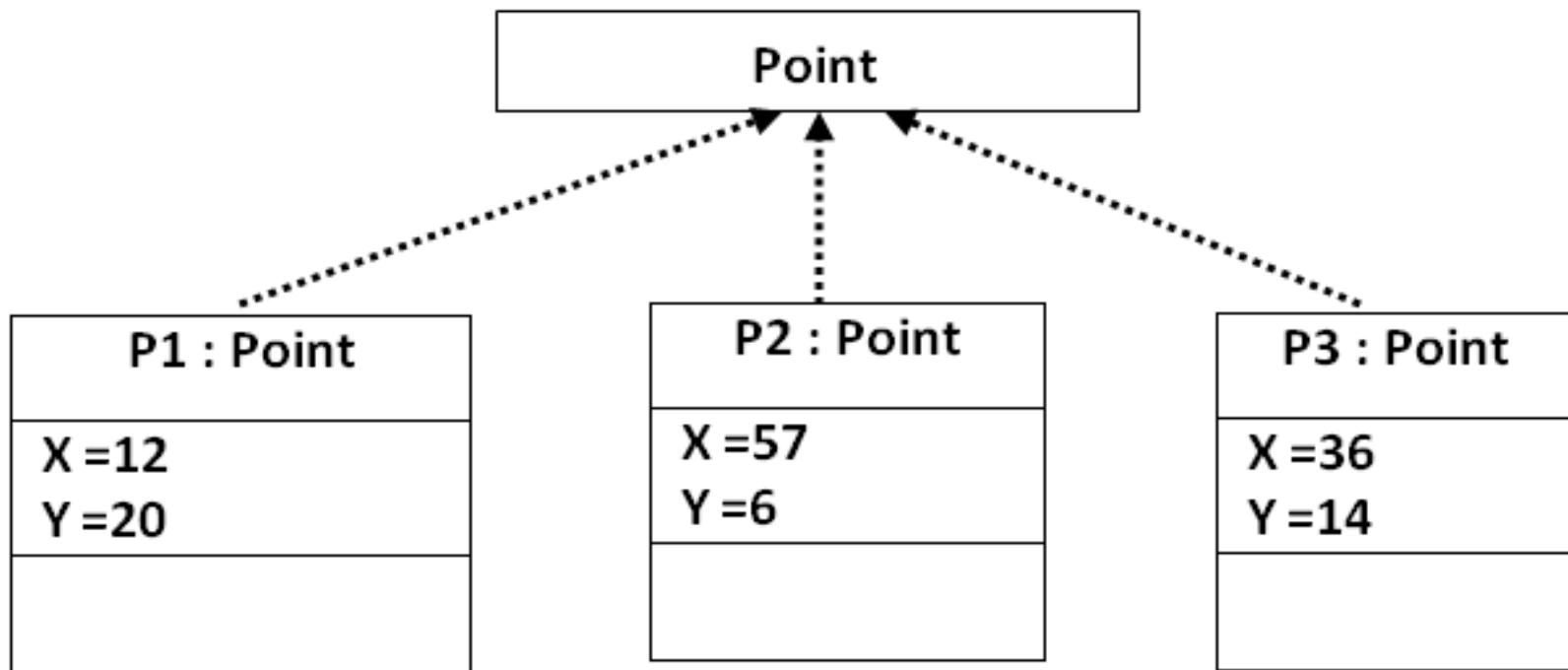
```
Class Personne {  
    String Nom;  
    String Prenom;  
    int age;  
    Void afficheNom()  
    {...};  
}
```



# L'instanciation : Exemple

---

- Chaque **Objet** (**p1, p2, p3**) instance de la **classe Point** possède sa propre valeur (son propre x et son propre y)



# Création d'objet

---

- La création d'un objet à partir d'une classe s'appelle : **instanciation**
- L'objet créé s'appel : **instance de la classe.**
- Pour créer un Objet :
  - Déclaration de l'objet
  - Instanciation et initialisation de l'objet en utilisant l'opérateur **new** suivi du **constructeur** de la classe.
- **Exemple :**

```
Point p ;           //Déclaration de l'objet
p=new Point() ;    //instanciation. (p est une instance de Point).
Point p2=new Point() ; // Déclaration et instanciation.
```

# Les constructeurs

---

- L'appel de **new** pour créer un nouvel objet déclenche, dans l'ordre :
  - L'allocation mémoire nécessaire au stockage de ce nouvel objet et l'initialisation par défaut de ces attributs,
  - L'initialisation explicite des attributs, s'il y a lieu,
  - L'exécution d'un constructeur.
- Un **constructeur** est une méthode d'initialisation.

```
public class Application
{
    public static void main(String args[])
    {
        Personne pr= new Personne()
        pr.setNom("Hamidi");
    }
}
```

Le constructeur est ici celui par défaut (pas de constructeur défini dans la classe Personne)

# Les constructeurs

---

- Lorsque l'initialisation explicite n'est pas possible (par exemple lorsque la valeur initiale d'un attribut est demandée dynamiquement à l'utilisateur), il est possible de réaliser l'initialisation au travers d'un constructeur.
- Le constructeur est une méthode :
  - de même nom que la classe,
  - sans type de retour.
- Toute classe possède au moins un constructeur. Si le programmeur ne l'écrit pas, il en existe un par défaut, sans paramètres, de code vide.

# Les constructeurs

## Personne.java

```
public class Personne
{
    public String nom;
    public String prenom;
    public int age;
    public Personne(String unNom,
                    String unPrenom,
                    int unAge)
    {
        nom=unNom;
        prenom=unPrenom;
        age = unAge;
    }
}
```

Définition d'un Constructeur. Le constructeur par défaut (Personne()) n'existe plus. Le code précédent donnera une erreur

```
public class Application
{
    public static void main(String args[])
    {
        Personne pr= new Personne();
        pr.setNom(" Hamidi");
    }
}
```

Va donner une erreur à la compilation

# Les constructeurs

---

- Pour une même classe, il peut y avoir plusieurs constructeurs, de signatures différentes (surcharge).
- L'appel de ces constructeurs est réalisé avec le **new** auquel on fait passer les paramètres.
  - `p1 = new Personne("Hamidi", "Ali", 56);`
- Déclenchement du "bon" constructeur
  - Il se fait en fonction des paramètres passés lors de l'appel (nombre et types).
- **Attention**
  - Si le programmeur crée un constructeur (même si c'est un constructeur avec paramètres), le constructeur par défaut n'est plus disponible.  
Attention aux erreurs de compilation !

# Les constructeurs

## Personne.java

```
public class Personne
{
    public String nom;
    public String prenom;
    public int age;
    public Personne()
    {
        nom=null; prenom=null;
        age = 0;
    }
    public Personne(String unNom,
                    String unPrenom, int unAge)
    {
        nom=unNom;
        prenom=unPrenom; age = unAge;
    }
}
```

Redéfinition d'un  
Constructeur sans  
paramètres

On définit plusieurs  
constructeurs  
qui se différencient  
uniquement  
par leurs paramètres (on  
parle  
de leur signature)

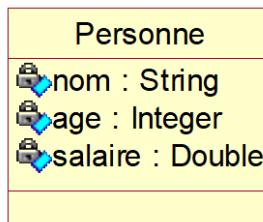
# Classe et objet en Java

---

Du modèle à ...

... la classe Java et  
de la classe à ...

... des instances  
de cette classe



```
class Personne
{
    String nom;
    int age;
    float salaire;
}
```

```
Personne pr1, pr2;  
pr1= new Personne ();  
pr2= new Personne ();
```

L'opérateur d'instanciation en Java est **new** :  
**MaClasse monObjet = new MaClasse();**

En fait, **new** va réserver l'espace mémoire nécessaire  
pour créer l'objet « **monObjet** » de la classe  
« **MaClasse** »

# Exemple

```
class Point
{
    //Attributs
    float x;
    float y;
    //Constructeur
    Point (float X, float Y){
        x=X;
        y=Y;
    }
    //Méthodes
    void afficher(){
        System.out.println("x="+x+"\ty="+y);
    }
    float distance() {
        float d;
        d=(float)Math.sqrt(x*x+y*y);
        return d;
    }
}
```

```
class TestPoint
{
    public static void main(String args[]) {
        //déclaration d'un objet de type Point
        Point p1;

        //Création d'un point p1
        P1=new Point(); //erreur le constructeur Point() n'est pas défini
        p1=new Point(10,20);

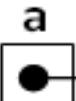
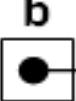
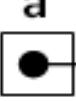
        p1.afficher(); //appel de la méthode afficher
        System.out.println("Distance entre O et p1:" + p1.distance());

        //déclaration d'un autre objet de type Point
        Point p2;

        //Création d'un point p2
        p2=new Point(30,40);
        p2.afficher();

        System.out.println("Distance entre O et p2:" + p2.distance());
    }
}
```

# Affectation d'objets :

Point a, b ;	<p>a  → null</p> <p>b  → null</p>
a=new Point(10,20) ; a.afficher(); → x=10 y=20	<p>a  → x y 10 20</p>
b=new Point(-2,5) ; b.afficher(); → x=-2 y=5	<p>b  → x y -2 5</p>
b=a ; b.afficher(); → x=10 y=20	<p>a  → x y 10 20</p> <p>b  → x y -2 5</p>

# Encapsulation des données

---

- L'**encapsulation** permet de protéger les données d'un objet.
- En effet, il n'est pas possible d'agir directement sur les données d'un objet. Il est nécessaire de passer par ses méthodes (**accesseurs et mutateurs**) qui jouent le rôle d'interface obligatoire.
- L'appel de méthode est en fait l'envoi d'un message à l'objet.
- Vu de l'extérieur, un objet se caractérise uniquement par les spécifications de ses méthodes.
- Le fonctionnement interne de l'objet est caché au monde extérieur.
- Cette démarche vise à atteindre deux objectifs :
  - Augmenter la sécurité des programmes
  - Faciliter la maintenance de l'objet.

# Encapsulation des données

---

- En java, il est possible d'agir sur la visibilité (accessibilité) des membres (attributs et méthodes) d'une classe vis-à-vis d'autres classes en précédant la déclaration de chaque attribut ou méthode par l'un des modificateurs (**public, private, protected**)
  - **private** : accessible que depuis l'intérieur de la classe où il est définie.
  - **public** : accessible depuis n'importe quelle classe.
  - **protected** : accessible depuis sa classe, depuis les classes du même package ou depuis les classes dérivées
- L'encapsulation consiste à protéger les attributs d'un objet, pour cela ces attributs doivent être déclarés **private** et on utilise des méthodes **public** pour y accéder (ces méthodes sont appelées **accesseurs** et **mutateurs**).

# Encapsulation

## ■ Exemples.

```
public class Personne
{
    public String nom;
    public String prenom;
    private int age;
    public Personne(String N,
                   String P, int A)
    {
        nom=N;
        prenom=P;
        age = A;
    }
}
```

```
class TestPersonne{
    public static void main(String[] args){
        Personne
        p= new Personne ("Hamidi", "Ali" , 20) ;

        System.out.println("Nom = " + p.nom);
        //Affiche: Nom= Hamidi

        System.out.println("age= " + p.age);
        //erreur : car age est privé
    }
}
```

# Accesseurs

---

- Un accesseur (**getter**) est une méthode permettant de récupérer le contenu d'un attribut protégée. Un accesseur, pour accomplir sa fonction :
  - doit avoir comme **type de retour** le type de la variable à renvoyer
  - ne doit pas nécessairement posséder d'arguments (paramètres)
- Une convention de nommage veut que l'on fasse commencer le nom de l'accesseur par le préfixe **get**.

# Accesseurs

```
public class Personne{  
    private String nom;  
    private int age;  
    public Personne(String N, int A) {  
        nom=N;  
        age = A;  
    }  
//Accesseur de nom.  
    public String getNom( ) {  
        return (nom);  
    }  
//Accesseur de age.  
    public int getAge( ) {  
        return (age);  
    }  
}
```

```
class TestPersonne{  
    public static void main(String[] args){  
        Personne p= new Personne ("Ali", 20) ;  
  
        System.out.println("age=" + p.age);  
        //erreur : car age est privé  
  
//solution  
        System.out.println("age="+  
            p.getAge());  
    }  
}
```

# Mutateurs

---

- Un **mutateur** (**setter**) est une méthode permettant de modifier le contenu d'un attribut protégée. Un mutateur, pour accomplir sa fonction :
  - doit avoir comme paramètre la valeur à assigner à la donnée membre. Le paramètre doit donc être du type de la donnée membre
  - ne doit pas nécessairement renvoyer de valeur (il possède dans sa plus simple expression le type **void**)
- Une convention de nommage veut que l'on fasse commencer le nom du mutateur par le préfix **set**.

# Mutateurs

```
public class Personne{  
    private String nom;  
    private int age;  
    public Personne(String N, int A){  
        nom=N;  
        age = A;  
    }  
//accesseur pour récupérer age  
    public int getAge(){  
        return age;  
    }  
//mutateur pour modifier age  
    public void setAge(int a){  
        age = a;  
    }  
}
```

```
class TestPersonne{  
    public static void main(String[] args){  
        Personne p= new Personne ("Ali", 20) ;  
        System.out.println("age=" + p.age);  
        //erreur : car age est privé  
        //solution  
        System.out.println("age="+ p.getAge());  
        p.age=35 ; //erreur age est privé  
        //solution  
        p.setAge(35) ;  
    }  
}
```

# L'opérateur **this**

---

- Le mot-clé **this** désigne, dans une classe, l'instance courante de la classe elle-même.
- Exemple 1 :
  - si le constructeur a comme argument un nom de variable identique à une variable de la classe et qu'on souhaite l'initialiser avec la variable passée en paramètre, alors on écrit :

```
constructeur(int maVar){  
    this.maVar=maVar  
}
```

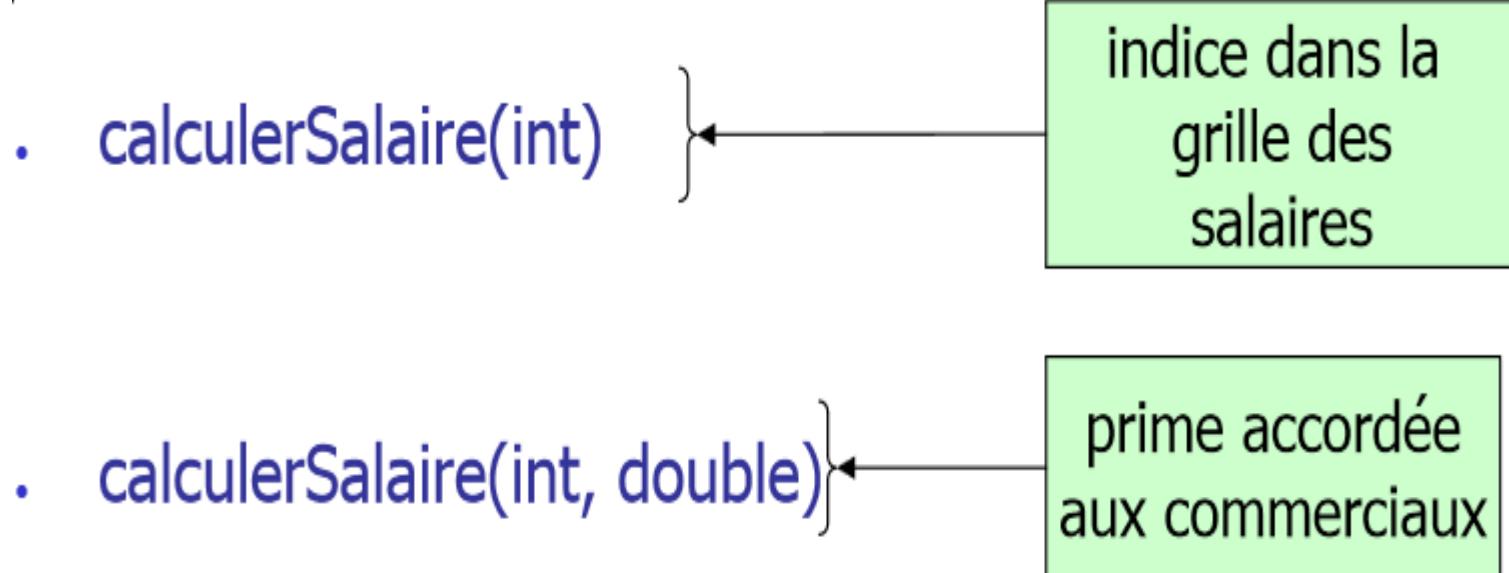
# L'opérateur this (Exemples)

```
public class Point {  
    private int x;  
    private int y;  
    public Point() {  
        x = 0;  
        y = 0;  
    }  
    public Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
}
```

```
public class Personne{  
    private String nom;  
    private int age;  
    public Personne(String nom, int age){  
        this.nom=nom;  
        this.age = age;  
    }  
    public int getAge(){  
        return age;  
    }  
    public void setAge(int age){  
        this.age = age;  
    }  
}
```

# Surcharge d'une méthode

- En Java, on peut surcharger une méthode, c'est-à-dire, ajouter une méthode qui a le même nom mais pas la même signature qu'une autre méthode :



# Surcharge d'une méthode (suite)

---

- En Java, il est interdit de surcharger une méthode en changeant le type de retour
- Autrement dit, on ne peut différencier deux méthodes par leur type de retour
- **Par exemple**
  - il est interdit d'avoir ces deux méthodes dans une classe
    - **int calculerSalaire(int)**
    - **double calculerSalaire(int)**

# Surcharge d'une méthode (Exemple)

```
public class Test {  
    public Test() {  
        MaMethode();  
        MaMethode(50);  
    }  
    public void MaMethode(int variable) {  
        System.out.println("Le nombre que vous avez passé en  
paramètre vaut : " + variable);  
    }  
    public void MaMethode() {  
        System.out.println("Vous avez appelé la méthode sans  
paramètre");  
    }  
}
```

# Accès aux attributs et méthodes d'un objet

---

```
public class Application
{
    public static void main(String args[])
    {
        Personne pr= new Personne()
        pr.nom = "Alami" ; // si l'attribut nom est public
        pr.setAge(25) ;
    }
}
```

## Remarque :

Contrairement aux variables, les attributs d'une classe, s'ils ne sont pas initialisés, se voient affecter automatiquement une valeur par défaut.

Cette valeur vaut : 0 pour les variables numériques, false pour les booléens, et null pour les références.

# Portée des variables (1)

- Les variables sont connues et ne sont connues qu'à l'intérieur du bloc dans lequel elles sont déclarées

```
public class test
{
    int i, k, j;
    public void fonct1(int z)
    {
        int r, j;
        r = z;
    }
    public void fonct2()
    {
        k = r;
    }
}
```

Ce sont 2 variables différentes

k est connu au niveau de la méthode fonct2() car déclaré dans le bloc de la classe. Par contre la variable r n'est pas défini pour fonct2(). On obtient une erreur à la compilation

# Portée des variables (2)

- En cas de conflit de nom entre des variables locales et des variables globales, c'est toujours la variable la plus locale qui est considérée comme étant la variable désignée par cette partie du programme
  - Attention par conséquent à ce type de conflit quand on manipule des variables *globales*.

C'est le **j** défini en local  
qui est utilisé dans  
la méthode fonct()

```
public class Test
{
    int i, k, j;
    public void fonct(int z)
    {
        int j,r;
        j = z;
    }
}
```

# Destruction d'objets (1)

---

- Java n'a pas prévu la notion de destructeur telle quelle en existe en C++.
- La destruction des références d'objets se fait automatiquement grâce à un processus appelé "ramasse miettes" (ou **garbage collector**) qui s'occupe de collecter les objets qui ne sont plus référenciés.
- La destruction des objets peut se faire manuellement par la méthode **System.gc()**. Cette méthode lance en tâche de fond le ramasse-miettes.

# Destruction d'objets (2)

---

- Il est possible au programmeur d'indiquer ce qu'il faut faire juste avant de détruire un objet.
- C'est le but de la méthode **finalize()** de l'objet.
- Cette méthode est utile, par exemple, pour :
  - fermer une base de données,
  - fermer un fichier,
  - couper une connexion réseau,
  - etc.

# Destruction d'objets (3)

## ■ Exemple.

```
public class MonMessage {  
    public void Afficher(String strMsg){  
        System.out.println(strMsg);  
    }  
    public void finalize( ) {  
        System.out.println("c'est la fin de  
l'objet");  
    }  
}
```

```
public class TestMonMessage {  
    public static void main(String[] args) {  
        MonMessage Msg=new MonMessage();  
        Msg.Afficher("Bonjour tout le monde");  
        Msg=null;  
        System.gc();  
    }  
}
```

Le programme affiche après l'exécution :

**Bonjour tout le monde**  
**c'est la fin de l'objet**

# Variables de classe (1)

---

- Il peut s'avérer nécessaire de définir un attribut dont la valeur soit partagée par toutes les instances d'une classe. On parle de **variable de classe**.
- Ces variables sont, de plus, stockées une seule fois, pour toutes les instances d'une classe.
- Donc tous les objets partagent la même valeur
- Les attributs statiques sont déclarés par le modificateur : **static**
- **Accès à l'attribut static :**
  - depuis une méthode de la classe comme pour tout autre attribut,
  - via du nom de la classe,      **NomClasse.nomAttributStatique**
  - à l'aide instance de la classe. **nomObjet.nomAttributStatique**

# Variables de classe (2)

## Exemple1

```
class Toto {  
    int a; /* attribut d'objet  
(non statique)*/  
  
    static int b; /*attribut de  
classe*/  
}
```

```
public class TesteToto {  
    public static void main (String[] arg) {  
        Toto.b=10;  
        Toto obj1= new Toto ();  
        obj1.a=20;  
        Toto obj2= new Toto ();  
        obj2.a=30;  
        obj2.b=15;//Pour tous les objets, b=15  
        System.out.println(Toto.b); // affiche 15  
        System.out.println(obj1.b); // affiche 15  
        System.out.println(obj1.a); //affiche 20  
        System.out.println(obj2.a); // affiche 30  
    }  
}
```

# Variables de classe (3)

## Exemple2

```
class Personne {  
    String nom ;  
    int age ;  
    public static int nb = 0;  
  
    public Personne (String  
        nom , int age )  {  
        this.nom=nom ;  
        this.age=age ;  
        nb++;  
    }  
}
```

```
public class TestePersonne{  
    public static void main( String args[])  
    {  
        int n= Personne.nb;  
        Personne p1, p2 ;  
        p1=new Personne("Alami",20) ;  
        p2= new Personne("Fatima",18) ;  
        System.out.println(Personne.nb) ; //affiche 2  
        System.out.println(p1.nb) ; //affiche 2  
        System.out.println(p2.nb) ; //affiche 2  
    }  
}
```

# Méthodes de classe (1)

---

- Il peut être nécessaire de disposer d'une méthode qui puisse être appelée sans instance de la classe. C'est une **méthode de classe**.
- On utilise là aussi le mot réservé **static**
- Puisqu'une méthode de classe peut être appelée sans même qu'il n'existe d'instance, une méthode de classe **ne peut pas accéder à des attributs non statiques**. Elle ne peut accéder qu'à ses propres variables et à des variables de classe.

# Méthodes de classe (2)

## Exemple1

```
class UneClasse {  
    public int x;  
    public static int y ;  
  
    public static void maMéthode() {  
        int v ;  
        x = 5; // Erreur de compilation  
        y=7 ; //juste  
        v=10 ; //juste  
    }  
}
```

```
class TesterUneClasse  
{ public static void main(String  
args[]) {  
    UneClasse.maMéthode() ;  
}  
}
```

**La méthode maMéthode() est une méthode de classe (static) donc elle ne peut accéder à un attribut non lui-même attribut de classe (non statique)**

**Appel de maMéthode() via le nom de la classe sans même qu'il n'existe d'instance (objet).**

# Méthodes de classe (3)

---

## ■ Autres exemples de méthodes de classe courantes : (Math, Integer et Float sont des noms de classe)

- Math.sin(x);
- String chaine = String.valueOf (i); //convertir le nombre i en chaine.
- int n = Integer.parseInt(ch) ; //Convertir la chaine ch en un int.
- float a= Float.parseFloat(ch) ; // Convertir la chaine ch en un float.

---

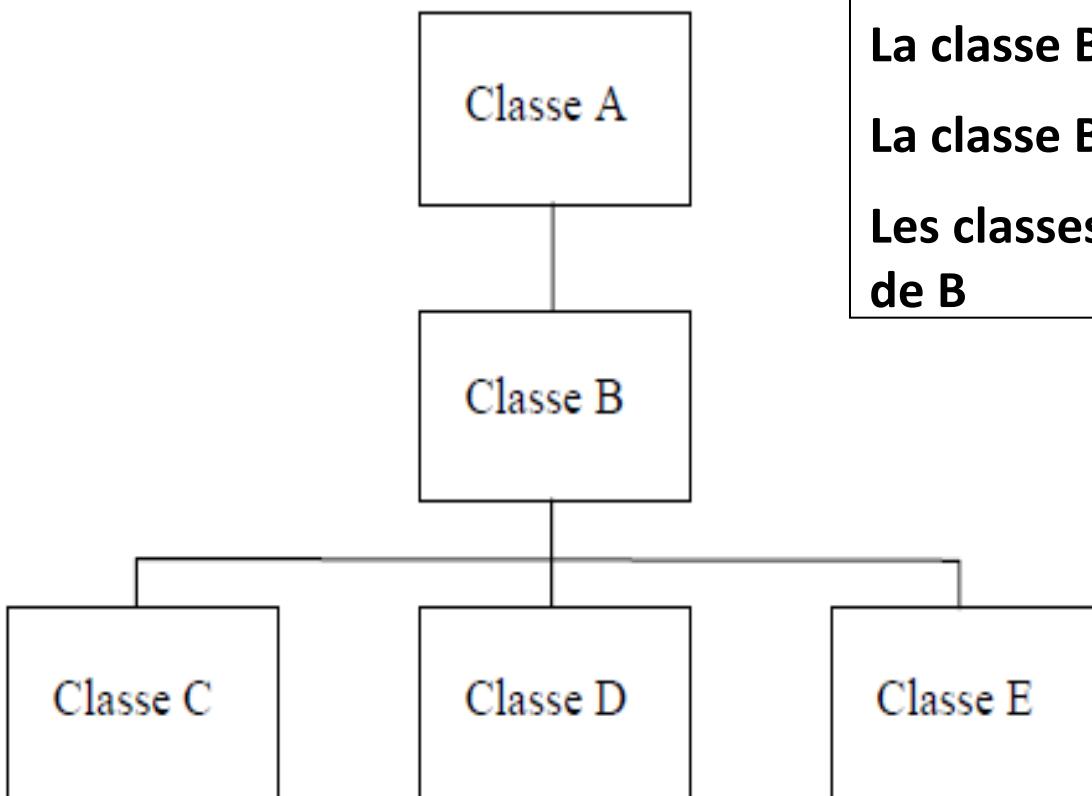
# HÉRITAGE ET POLYMORPHISME

# L'héritage (1)

---

- **L'héritage** est un mécanisme qui permet à une classe d'hériter de l'ensemble du comportement et des attributs d'une autre classe.
- Grâce à l'héritage, une classe peut disposer immédiatement de toutes les fonctionnalités d'une classe existante. De ce fait, pour créer la nouvelle classe, il suffit d'indiquer dans quelle mesure elle diffère de la classe existante.
- Une classe qui hérite d'une autre classe est appelée **sous-classe** ou **classe dérivée**, et la classe qui offre son héritage à une autre est appelée **super-classe**, ou **classe de base**.

# L'héritage (2)



**La classe A est la super-classe de B**

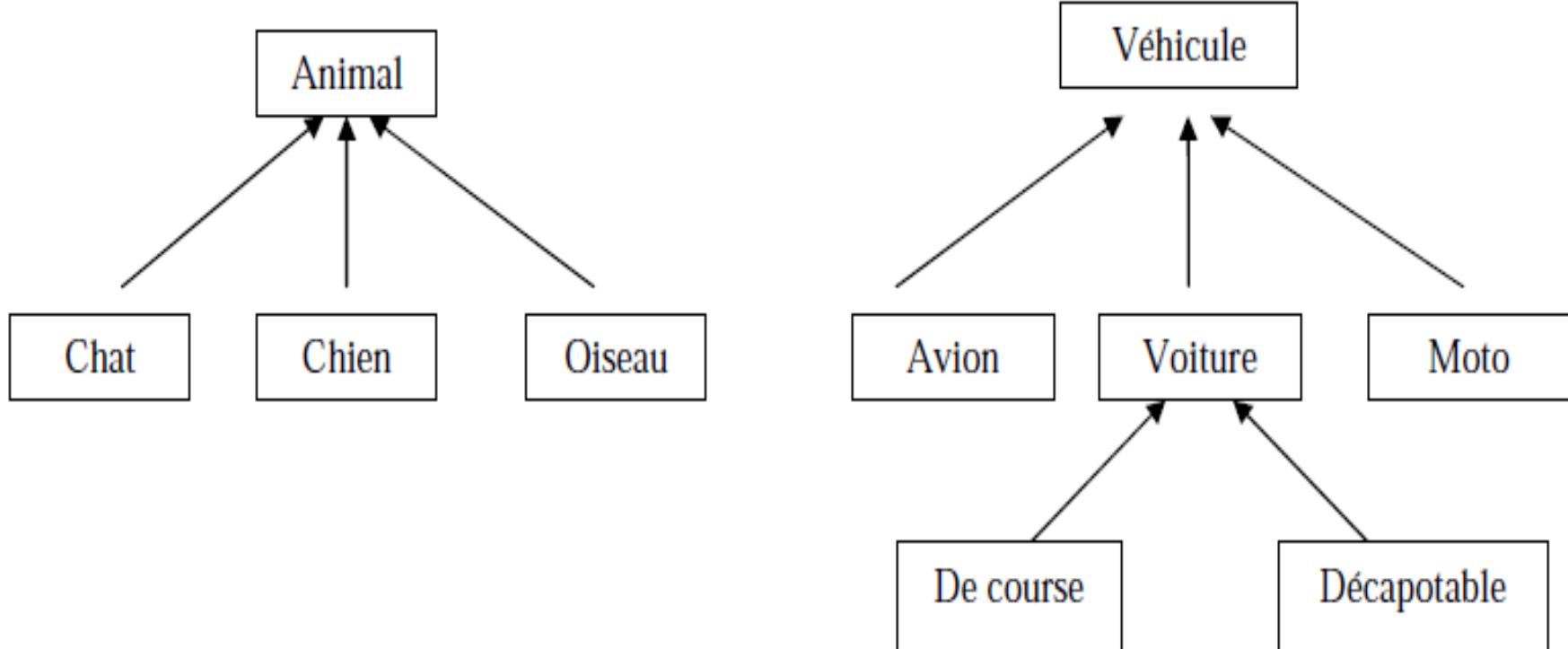
**La classe B est une sous-classe de A**

**La classe B est la super-classe de C, D et E**

**Les classes C, D et E sont des sous-classes de B**

# L'héritage (3) : exemple de graphe de l'héritage

---



**L'héritage représente la relation: EST-UN  
Exemple:**

Un chat est un animal

Une moto est un véhicule

Une voiture de course est une voiture

# L'héritage (4) : définition et principe

---

## ■ Héritage :

- mécanisme permettant le partage et la réutilisation de propriétés entre les objets. La relation d'héritage est une relation de généralisation / spécialisation.

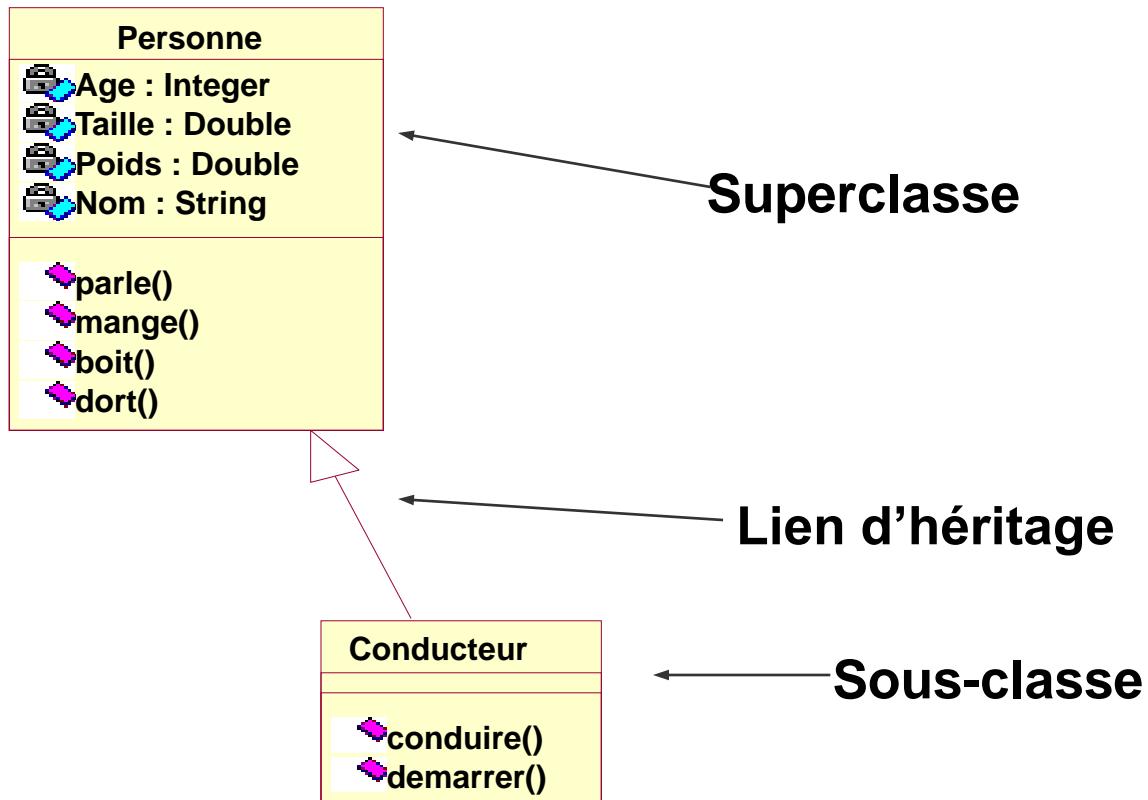
## ■ Hiérarchisation

- La classe dont on dérive est dite **CLASSE DE BASE** : Animal est la classe de base (classe supérieure),
- Les classes obtenues par dérivation sont dites **CLASSES DÉRIVÉES** : Chat, Chien et Oiseau sont des classes dérivées (sous-classes).

## ■ Principe de l'héritage

- **Besoins** : éviter de dupliquer du code (attributs et méthodes) dans différentes classes qui partagent des caractéristiques communes
  - facilite les modifications futures => elles n'ont besoin d'être faites qu'à un seul endroit
  - représentation explicite d'une logique d'héritage des concepts du domaine (relation "est-un")

# L'héritage (5) : représentation graphique



Représentation UML d'un héritage (simple)

# L'héritage en Java (1)

---

- Java implémente le mécanisme d'héritage simple qui permet de "factoriser" de l'information grâce à une relation de généralisation / spécialisation entre deux classes.
- **L'héritage multiple n'existe pas en Java.**
  - L'implémentation d'interface permet de compenser cette limitation
- Pour le programmeur, il s'agit d'indiquer, dans la sous-classe, le nom de la superclasse dont elle hérite.
- Pour signifier qu'une classe fille hérite d'une classe mère, on utilise le mot clé **extends**

**class fille extends mere {**

.....

**}**

# L'héritage en Java (Exemple)

```
class Personne
```

```
{
```

```
    private String nom;
```

```
    private Date date_naissance;
```

```
// ...
```

```
}
```

```
class Employe extends Personne
```

```
{
```

```
    private float salaire;
```

```
// ...
```

```
}
```

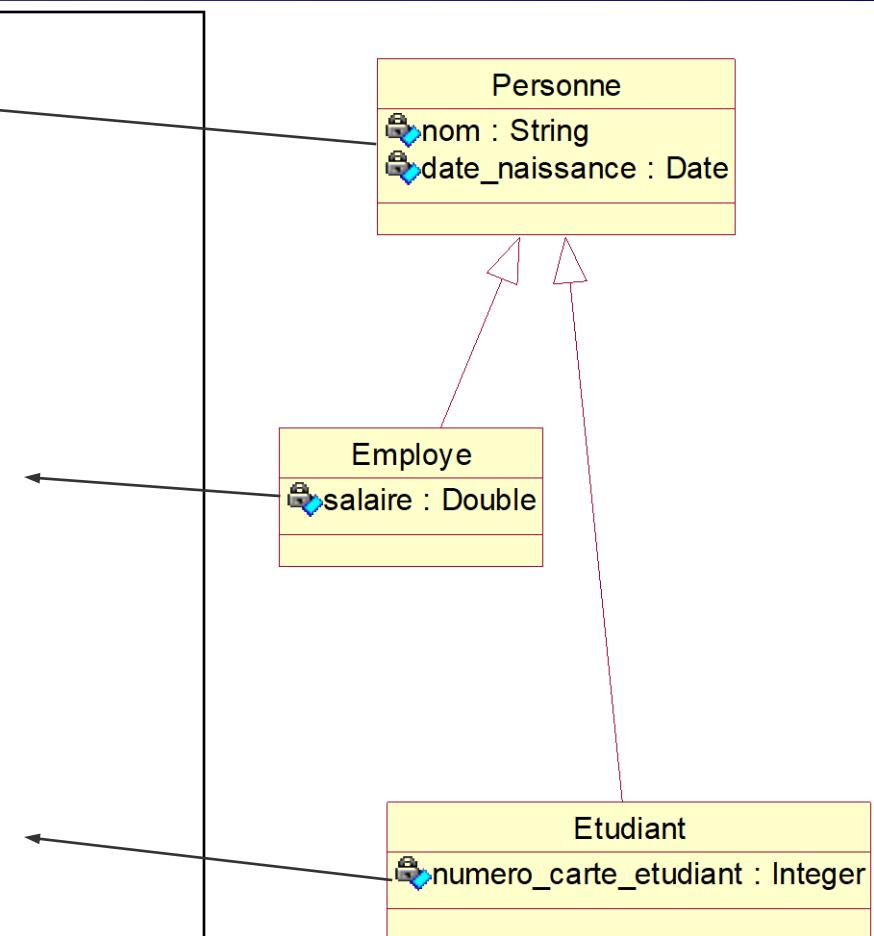
```
class Etudiant extends Personne
```

```
{
```

```
    private int numero_carte_etudiant;
```

```
// ...
```

```
}
```



# L'héritage en Java (2)

---

## ■ En java, toutes les classes sont dérivées de la classe **Object**.

- La classe Object est la classe de base de toutes les autres. C'est la seule classe de Java qui ne possède pas de classe mère.
- Tous les objets en Java, quelle que soit leur classe, sont du type Object. Cela implique que tous les objets possèdent déjà à leur naissance un certain nombre d'attributs et de méthodes dérivées d'Object.
- Dans la déclaration d'une classe, si la clause **extends** n'est pas présente, la surclasse immédiatement supérieure est donc **Object**.
- Une méthode telle que **toString()** est définie dans Object, ce qui explique que tout objet Java possède obligatoirement une méthode **toString()**.
- Il est possible (et même souhaitable) de redéfinir la méthode **toString()** dans les classes.

# L'héritage en Java (3)

---

## ■ Constructeurs et héritage

- Si une classe **A** hérite d'une classe **B** alors, la construction d'une instance de **A** commence toujours par l'appel d'un constructeur de la super-classe **B**.
- Pour forcer l'appel d'un constructeur précis, on utilisera le mot réservé **super**. Cet appel devra être la **première instruction** du constructeur.
- Si aucun appel explicite à un des constructeurs de la classe mère n'est fait, le compilateur rajoute l'instruction **super()** à tous les constructeurs de la classe fille.

# L'héritage en Java (4)

```
public class Personne
{
    private String nom, prenom;
    private int anNaissance;
    public Personne()
    {
        nom=" "; prenom=" ";
    }
    public Personne(String nom,
                    String prenom, int anNaissance)
    {
        this.nom=nom;
        this.prenom=prenom;
        this.anNaissance=anNaissance;
    }
}
```

```
public class Employe extends Personne
{
    private float salaire;
    public Employe () {}
    public Employe (String nom,
                   String prenom, int annee, float
sal)
    {
        super(nom, prenom, annee);
        salaire=sal;
    }
}
```

Appel explicite à ce constructeur  
avec le mot clé super

# Les membres protégés (1)

- Les sous-classes n'ont pas accès aux membres private de leur classe mère.

```
public class Personne
{
    private String nom, prenom;
    private int anNaissance;
    public Personne(String nom,
                    String prenom, int anNaissance)
    {
        this.nom=nom;
        this.prenom=prenom;
        this.anNaissance=anNaissance
    }
}
```

```
public class Employe extends Personne
{
    public Employe () {}

    public void affiche() {
        System.out.println(" nom :" + nom);
    }
}
```

**Erreur!**  
nom est un attribut  
private de Personne et n'est  
donc pas accessible dans  
la classe Employe

# Les membres protégés (2)

---

- Si on veut qu'un attribut ou une méthode soit encapsulé pour l'extérieur mais qu'il soit accessible par les sous-classes, il faut le déclarer **protected** à la place de **private**.

```
public class Personne
{
    protected String nom, prenom;
    public int anNaissance;
    public Personne(String nom,
                    String prenom, int anNaissance)
    {
        this.nom=nom;
        this.prenom=prenom;
        this.anNaissance=anNaissance;
    }
}
```

```
public class Employe extends Personne
{
    public Employe () {}

    public void affiche() {
        System.out.println(" nom :" + nom);
    }
}
```

# Accessibilité des membres (résumé)

---

Modificateur d'accès	Accessible à la classe	Accessible à une classe du même package	Accessible à une classe fille	Accessible à tous
public	Oui	Oui	Oui	Oui
protected	Oui	Oui	Oui	Non
par défaut	Oui	Oui	Non	Non
private	Oui	Non	Non	Non

# Redéfinition de méthodes

- Une sous-classe peut redéfinir des méthodes existant dans une de ses superclasses (directe ou indirectes), à des fins de spécialisation.
  - La méthode redéfinie **doit avoir la même signature**.

## Exemple

```
class Employe extends Personne
{
    private float salaire;
    public calculePrime( )
    {
        // ...
    }
}
```

redéfinition

```
class Cadre extends
Employe
{
    public calculePrime()
    {
        // ...
    }
}
```

# Recherche dynamique des méthodes (1)

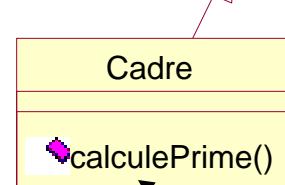
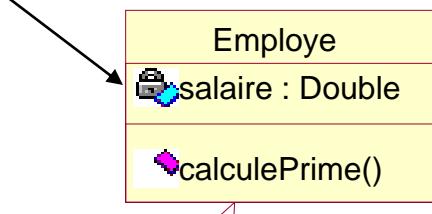
---

## ■ le mécanisme de "lookup" dynamique :

- déclenchement de la méthode la plus spécifique d'un objet, c'est-à-dire celle correspondant au type réel de l'objet, déterminé à l'exécution uniquement (et non le type de la référence, seul type connu à la compilation, qui peut être plus générique).
- Cette dynamicité permet d'écrire du code plus générique.

# Recherche dynamique des méthodes (2)

```
Employe pr= new Employe();  
pr.calculerPrime();
```



```
Employe pr= new Cadre();  
pr.calculerPrime();
```

# Surdéfinition de méthodes (1)

---

- Dans une même classe, plusieurs méthodes peuvent posséder le même nom, pourvu qu'elles diffèrent en nombre et/ou type de paramètres.
  - On parle de **surdéfinition** ou **surcharge**
  - Le choix de la méthode à utiliser est fonction des paramètres passés à l'appel.
    - Ce choix est réalisé de façon statique (c'est-à-dire à la compilation).
  - Très souvent les constructeurs sont surchargés (plusieurs constructeurs prenant des paramètres différents et initialisant de manières différentes les objets)

# Surdéfinition de méthodes (2)

---

## ■ Quelques précisions cependant :

- le type de retour seul, ne suffit pas à distinguer deux méthodes de signatures identiques par ailleurs,
- les types des paramètres doivent être "suffisamment" différents pour qu'il n'y ait pas d'ambiguïtés, en particulier avec les conversions de types automatiques (exemple : promotion d'un float en double).

## ■ Ne pas confondre surcharge et redéfinition (même si très proche)

- On redéfinit dans une classe des méthodes héritées de sa superclasse.
  - même nom, même signature, même type de retour.
- On surcharge quand plusieurs méthodes dans une même classe ont le même nom mais des paramètres différents.
  - même nom, **mais** signature différente et peu importe le type de retour.

# Mot-clé final

---

- **variable ou paramètre** : sa valeur ne plus être modifiée  
**final float pi=3.14f;**

L'affectation d'un attribut final peut être faite lors de l'instanciation.

- **méthode** : interdire une éventuelle redéfinition d'une méthode  
**public final void f() {...}**

- **classe finale** : la classe ne peut plus être étendue.

**final class A { ...}**

**Cela signifie qu'aucune autre classe ne peut en hériter**

# L'autoréférence : this (1)

---

- Le mot réservé **this**, utilisé dans une méthode, désigne la référence de l'instance à laquelle le message a été envoyé (donc celle sur laquelle la méthode est « exécutée »).
- Il est utilisé principalement :
  - lorsqu'une référence à l'instance courante doit être passée en paramètre à une méthode,
  - pour lever une ambiguïté,
  - dans un constructeur, pour appeler un autre constructeur de la même classe.

# L'autoréférence : this (2)

```
class Personne
{
    public String nom;
    Personne (String nom)
    {
        this.nom=nom;
    }
}
```

Pour lever l'ambiguïté sur le mot « nom » et déterminer si c'est le nom du paramètre ou de l'attribut

```
public MaClasse(int a, int b) {...}
```

```
public MaClasse (int c)
{
    this(c,0);
}
```

```
public MaClasse ()
{
    this(10);
}
```

Appelle le constructeur  
MaClasse(int a, int b)

Appelle le constructeur  
MaClasse(int c)

# Référence à la superclasse

---

- Le mot réservé **super** permet de faire référence au constructeur de la superclasse directe mais aussi à d'autres informations provenant de cette superclasse.

```
class Employe extends Personne
{
    private float salaire;
    public float calculePrime()
    {
        return (salaire * 0.05);
    }
    // ...
}
```

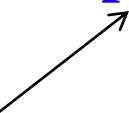
Appel à la méthode  
**calculPrime()** de la  
superclasse de Cadre

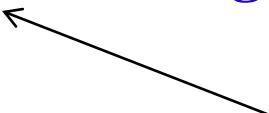
```
class Cadre extends Employe
{
    public float calculPrime()
    {
        return (super.calculPrime() / 2);
    }
    // ...
}
```

# Transtypage et héritage : surclassement

---

- Le transtypage consiste à convertir un objet d'une classe en objet d'une autre classe
- Vers une «super-class» (**upcast** ou **surclassement**), c'est toujours possible : on peut toujours transtyper vers une super-classe, plus pauvre en informations.
- **Rappel :** le **cast** est le fait de forcer le compilateur à considérer un objet comme étant d'un type qui n'est pas le type déclaré ou le type réel de l'objet.
- En Java, les seuls casts autorisés entre classes sont les casts entre classes mères et classes filles.
- **Notions de :** **UpCasting** et **DownCasting**.

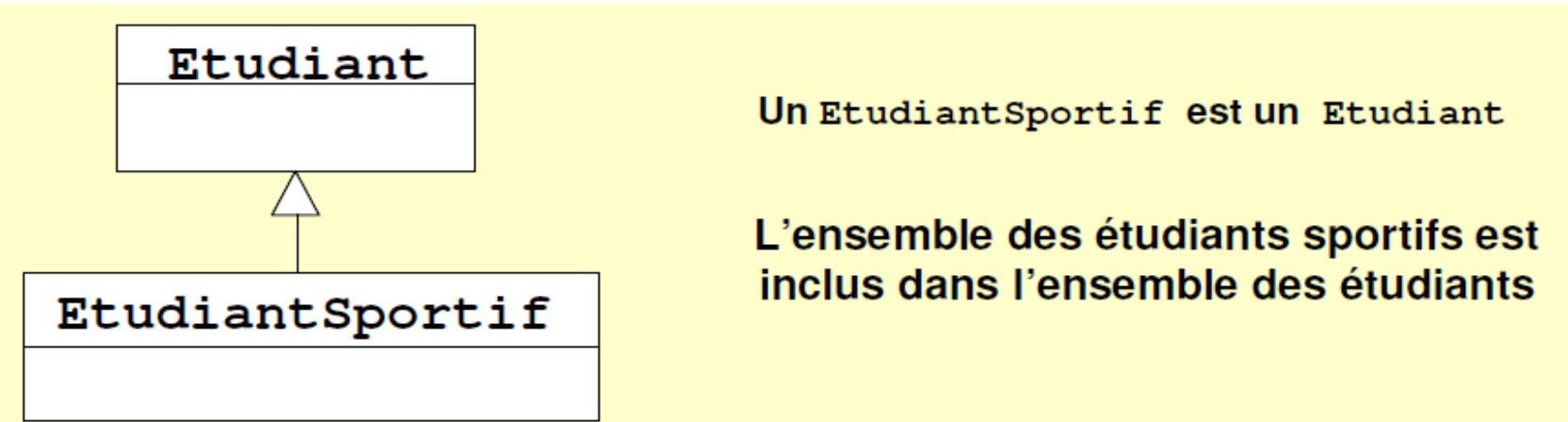
 **classe fille → classe mère**

 **classe mère → classe fille**

# UpCasting (surclassement)

---

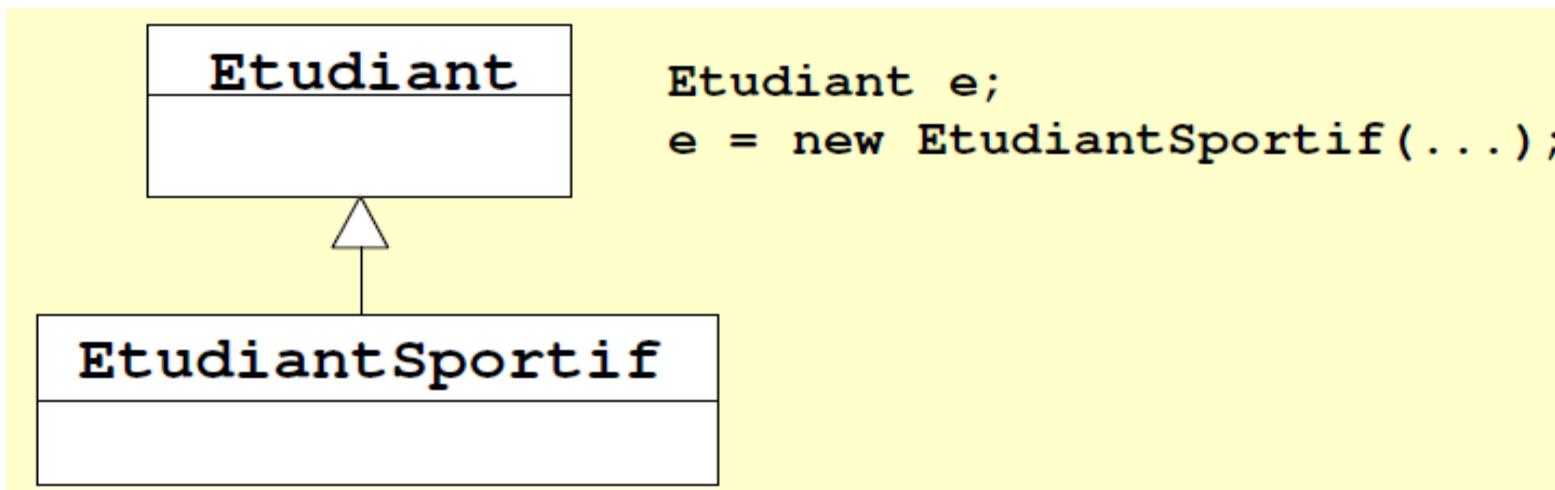
*Une classe B qui hérite de la classe A peut être vue comme un sous-type (sous ensemble) du type défini par la classe A.*



# UpCasting (surclassement)

---

- Tout objet instance de la classe B peut être aussi vu comme une instance de la classe A.
- Cette relation est directement supportée par le langage JAVA :
  - à une référence déclarée de type A il est possible d'affecter une valeur qui est une référence vers un objet de type B (**surclassement ou upcasting**)



# UpCasting (surclassement)

- Lorsqu'un objet est "sur-classé" il est vu comme un objet du type de la référence utilisée pour le désigner
  - Ses fonctionnalités sont alors restreintes à celles proposées par la classe du type de la référence

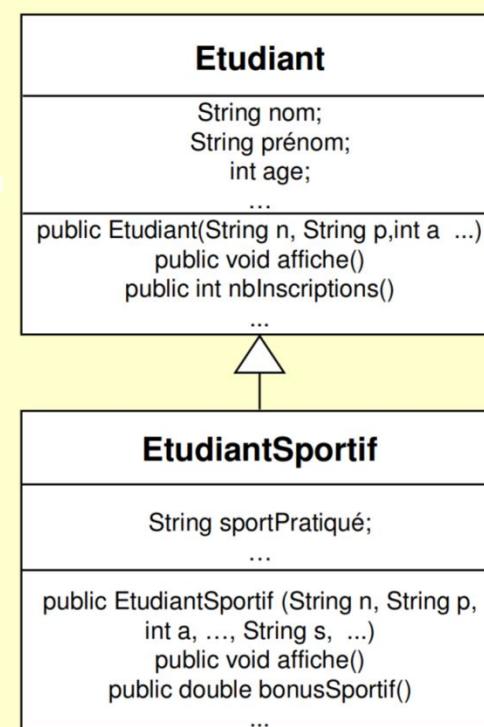
```
EtudiantSportif es;
es = new EtudiantSportif("DUPONT", "Jean",
                        25, ..., ...);

Etudiant e;
e = es; // upcasting

e.affiche();
es.affiche();

e.nbInscriptions();
es.nbInscriptions();

es.bonusSportif();
e.bonusSportif();
```



# Downcasting

---

- Pour que la conversion fonctionne, il faut qu'à l'exécution le type effectif de la référence à convertir soit une des classes ascendantes !

```
ClasseA objA = new ClasseB(); // surclassement, upcasting  
ClasseB objB = (ClasseB) objA; // downcasting
```

- Le downcasting permet de forcer un type à la compilation
  - C'est une promesse que l'on fait au moment de la compilation.
- Pour que le transtypage soit valide, il faut qu'à l'exécution le type effectif de **objA** soit compatible avec le type ClasseB
  - **Compatible** : la même classe ou n'importe quelle sous classe de ClasseB
- Si la promesse n'est pas tenue une erreur d'exécution se produit

# Cast : conversion de classes (Exemple)

```
class Personne
```

```
{  
    private String nom;  
    private Date date_naissance;  
    // ...  
}
```

```
class Employe extends Personne
```

```
{  
    public float salaire;  
    // ...  
}
```

```
Personne pr= new Employe ();  
float s = pr.salaire; // Erreur de compilation  
float sal = ( (Employe) pr).salaire; // OK
```

A ce niveau pour le compilateur dans la variable « pr » c'est un objet de la classe Personne, donc qui n'a pas d'attribut « salaire »

On « force » le type de la variable « pr » pour pouvoir accéder à l'attribut « salaire ». On peut le faire car c'est bien un objet Employe qui est dans cette variable

# Opérateur instanceof (1)

---

- L'opérateur **instanceof** confère aux instances une capacité d'introspection : il permet de savoir si un objet est instance d'une classe donnée.

**obj instanceof A** : retourne true si **obj** est une instance de **A** et false sinon.

```
if ( ... )
    Personne pr= new Etudiant();
else
    Personne pr= new Employe();
//...
if (pr instanceof Employe)
    // discuter affaires
else
    // proposer un stage
```

# Opérateur instanceof (2)

```
class Test{  
    public static void main(String args[]) {  
        Personne ahmed = new Employe("Ahmed", "1/25/1990",7500);  
        Personne anasse = new Etudiant("anasse", "1/25/1994",120);  
  
        System.out.println(ahmed instanceof Personne);  
        System.out.println(ahmed instanceof Employe);  
        System.out.println(anasse instanceof Personne);  
        System.out.println(anasse instanceof Etudiant);  
        System.out.println(anasse instanceof Employe);  
        System.out.println(ahmed instanceof Etudiant);  
    }  
}
```

true

true

true

true

false

false

# Polymorphisme

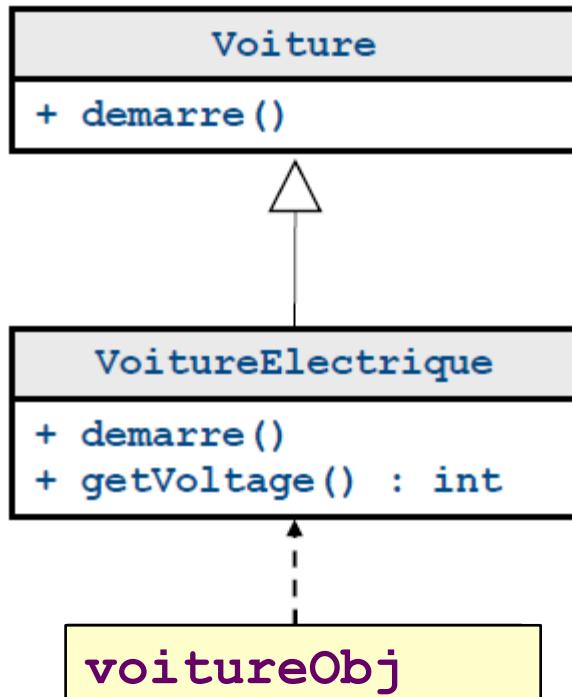
---

- **Le terme polymorphisme** décrit la caractéristique d'un élément qui peut prendre plusieurs formes, comme l'eau qui se trouve à l'état solide, liquide ou gazeux.
- **En P.O.O, on appelle polymorphisme :**
  - le fait qu'un objet d'une classe puisse être manipulé comme s'il appartenait à une autre classe.
  - le fait que la même opération puisse se comporter différemment sur différentes classes de la hiérarchie.
- **Le polymorphisme** constitue la troisième caractéristique essentielle d'un langage orienté objet après l'abstraction des données (**encapsulation**) et **l'héritage**

# Définition du polymorphisme

## ■ Définition :

- Un langage orienté objet est dit polymorphique, s'il offre la possibilité de pouvoir percevoir un objet en tant qu'instance de classes variées, selon les besoins.
- Une classe **B** qui hérite de la classe **A** peut être vue comme un sous-type du type défini par la classe **A**



- **Rappel**
  - **voitureObj** est une instance de la classe **VoitureElectrique**
- **Mais aussi**
  - **voitureObj** est une instance de la classe **Voiture**

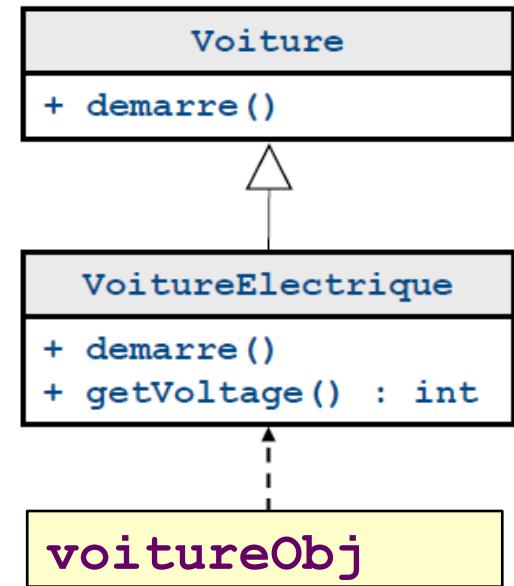
# Polymorphisme et Java : surclassement

## ■ Java est polymorphique

- A une référence de la classe **Voiture**, possible d'affecter une valeur qui est une référence vers un objet de la classe **VoitureElectrique**
- On parle de **surclassement** ou *upcasting*
- A une référence d'un type donné, soit A, il est possible d'affecter une valeur qui correspond à une référence vers un objet dont le type effectif est n'importe quelle sous classe directe ou indirecte de A

Objet de type sous-classe directe de Voiture

```
public class Test {  
    public static void main (String[] argv) {  
        Voiture voitureObj = new VoitureElectrique(...);  
    }  
}
```



# Polymorphisme et Java : surclassement

## ■ A la compilation

- Lorsqu'un objet est « surklassé », il est vu par le compilateur comme un objet du type de la référence utilisée pour le désigner
- Ses fonctionnalités sont alors restreintes à celles proposées par la classe du type de la référence

```
public class Test {  
    public static void main (String[] argv) {  
        // Déclaration et création d'un objet Voiture  
        Voiture voitureObj = new VoitureElectrique(...);  
        // Utilisation d'une méthode de la classe Voiture  
        voitureObj.demarre();  
        // Utilisation d'une méthode de la classe VoitureElectrique  
        System.out.println(voitureObj.getVoltage()); // Erreur  
    }  
}
```



Examiner le type de la référence

La méthode *getVoltage()* n'est pas disponible dans la classe Voiture!!!

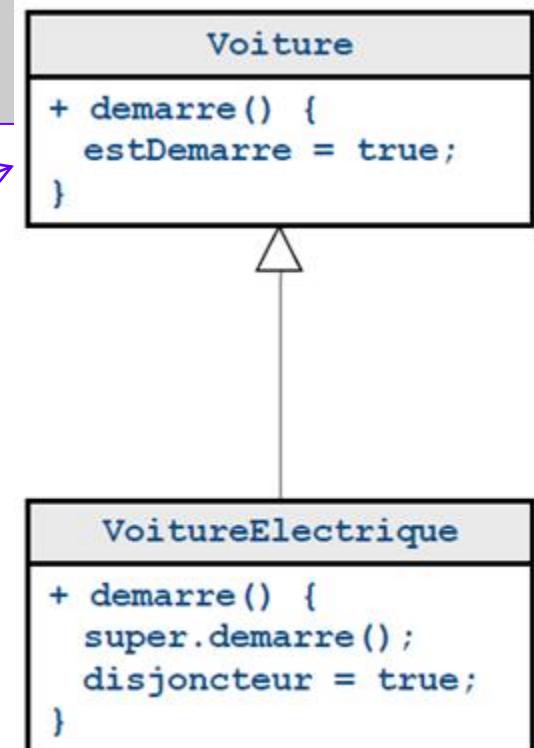
# Polymorphisme et Java : lien dynamique

```
public class Test {  
    public static void main (String[] argv) {  
        Voiture voitureObj= new VoitureElectrique(...);  
        voitureObj.demarre();  
    }  
}
```

L'objet voitureObj initialise les attributs de la classe VoitureElectrique

voitureObj.demarre();

**Constat** : C'est la méthode *demarre()* de *VoitureElectrique* qui est appelée.  
Puis elle appelle (par super...) la méthode de la super-classe



# Polymorphisme et Java : lien dynamique

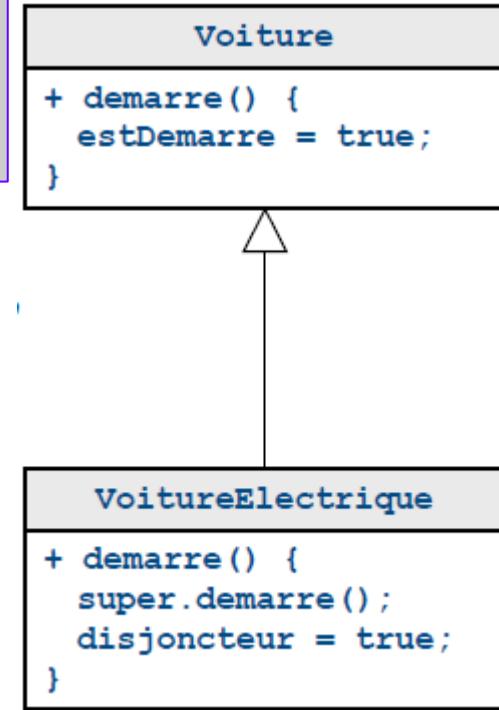
---

## ■ A l'exécution

- Lorsqu'une méthode d'un objet est accédée au travers d'une référence « surclassée », c'est la méthode telle qu'elle est définie au niveau de la classe effective de l'objet qui est invoquée et exécutée
- La méthode à **exécuter** est déterminée à l'exécution et non pas à la **compilation**
- On parle de **liaison tardive**, **lien dynamique**, **dynamic binding**, **latebinding** ou **run-time binding**

# Polymorphisme et Java : bilan

```
public class Test {  
    public static void main (String[] argv) {  
        Voiture maVoit = new VoitureElectrique(...);  
        maVoit.demarre();  
    }  
}
```



## ■ Surclassement (compilation)

- Une variable **maVoit** est déclarée comme étant une référence vers un objet de la classe **Voiture**
- Un objet de la classe **VoitureElectrique** est créé
- Pour le compilateur **maVoit** reste une référence d'un objet de la classe **Voiture**, et il empêche d'accéder aux méthodes spécifiques à **VoitureElectrique**

## ■ Liaison dynamique (exécution)

- Une variable **maVoit** est bien une référence vers un objet de la classe **VoitureElectrique**

# Polymorphisme et Java : bilan

**ClasseA a;**

## Surclassement

la référence peut désigner des objets de classe différente (n'importe quelle sous classe de ClasseA)

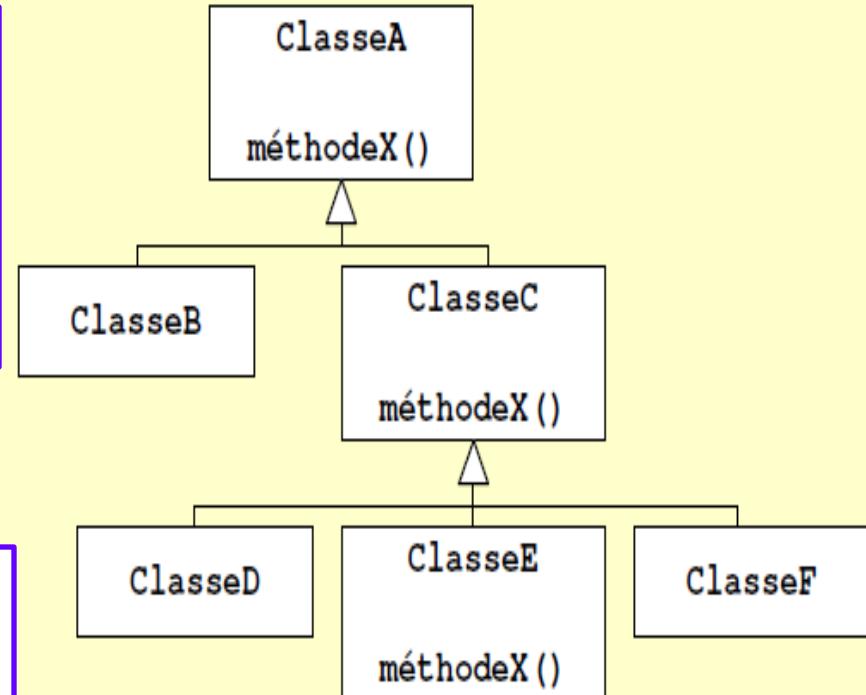
**a= new ClasseE();**

+

**a.méthodeX();**

## Lien dynamique

Le comportement est différent selon la classe effective de l'objet

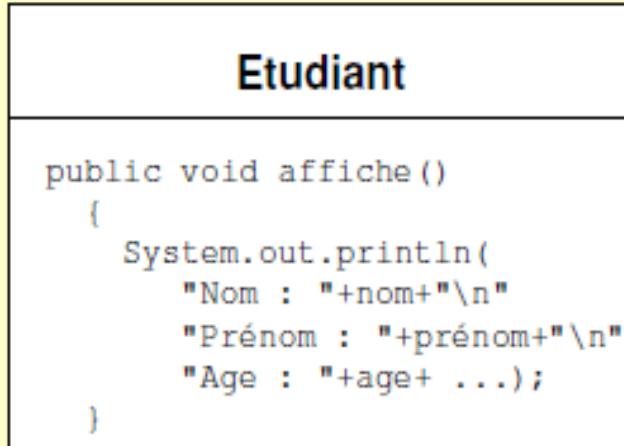


Manipulation uniforme des objets de plusieurs classes par l'intermédiaire d'une classe de base commune

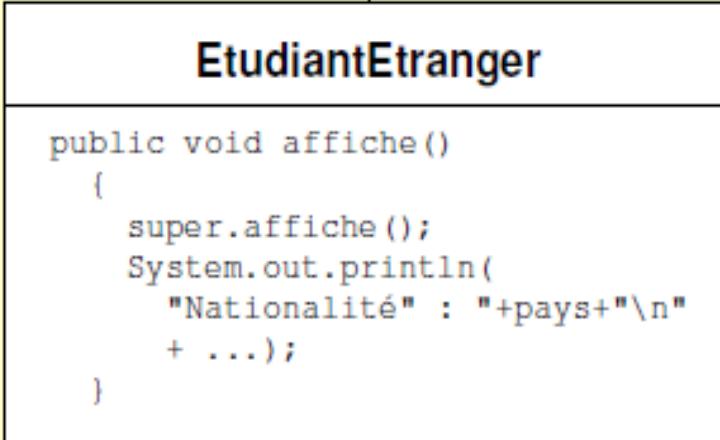
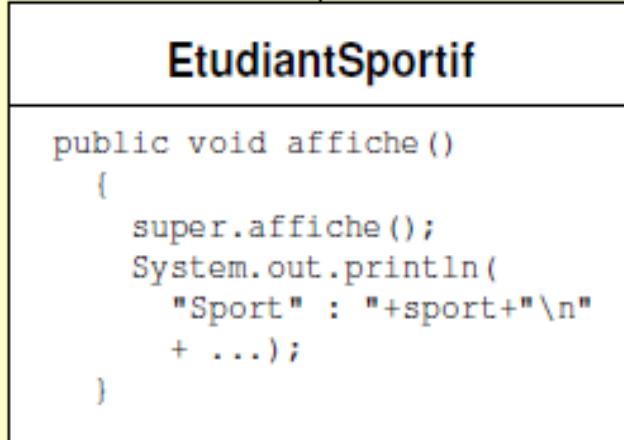
# Polymorphisme (Exemple)

liste peut contenir des étudiants de n'importe quel type

```
GroupeTD td1 = new GroupeTD();
td1.ajouter(new Etudiant("Ahmed", ...));
td1.ajouter(new EtudiantSportif("Ali",
    ... , "football"));
```



```
public class GroupeTD{
    Etudiant[] liste = new Etudiant[30];
    int nbEtudiants = 0;
    ...
    public void ajouter(Etudiant e)
    {
        if (nbEtudiants < liste.length)
            liste[nbEtudiants++] = e;
    }
    public void afficherListe()
    {
        for (int i=0;i<nbEtudiants; i++)
            liste[i].affiche();
    }
}
```



Si un nouveau type d'étudiant est défini, le code de GroupeTD reste inchangé

# Polymorphisme : ok, mais pourquoi faire ?

---

## ■ En utilisant le polymorphisme en association à la liaison dynamique

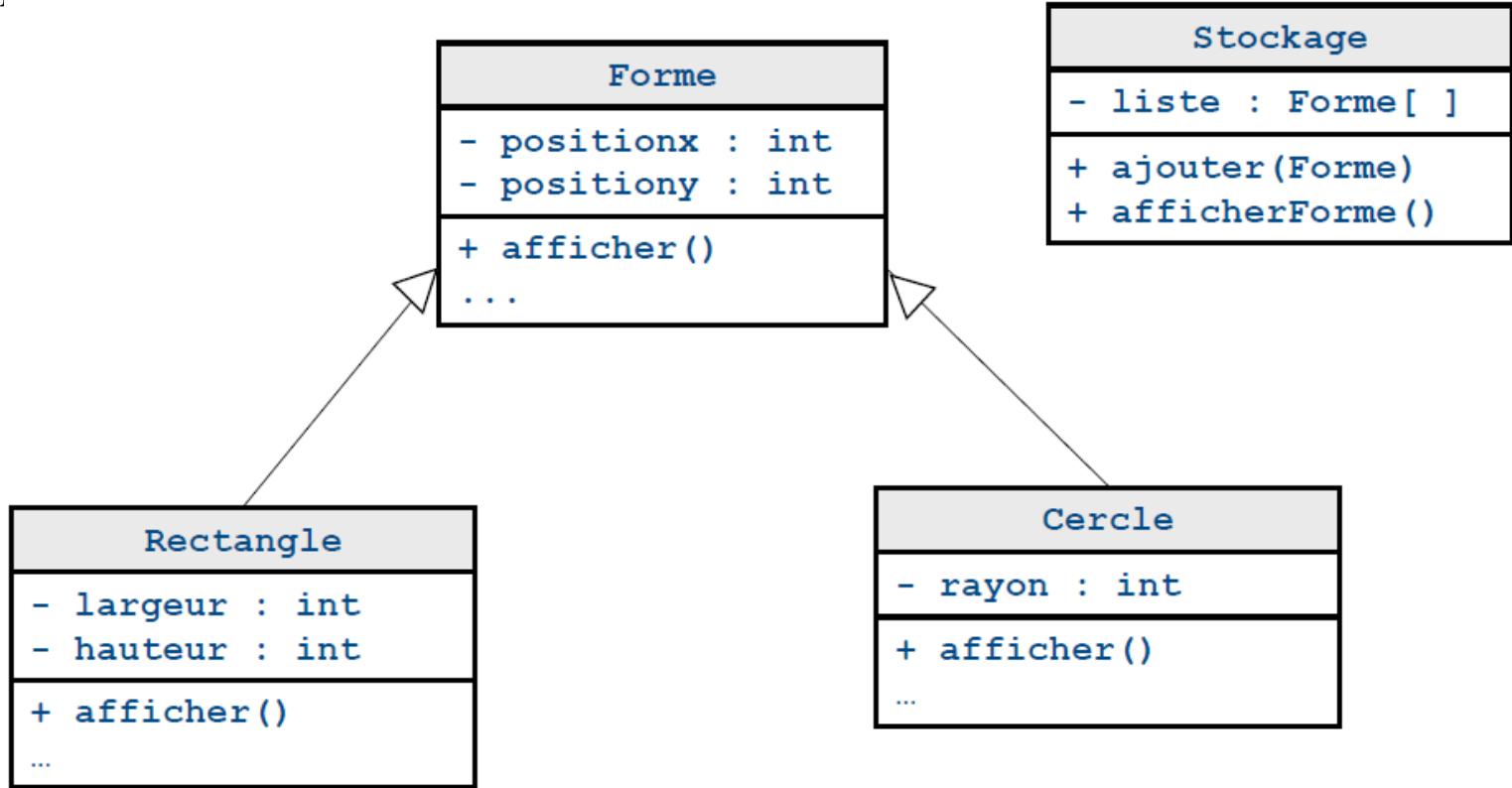
- Plus besoin de distinguer différents cas en fonction de la classe des objets
- Le polymorphisme constitue la troisième caractéristique essentielle d'un langage orienté objet après l'abstraction des données (encapsulation) et l'héritage
- Une plus grande facilité d'évolution du code. Possibilité de définir de nouvelles fonctionnalités en héritant de nouveaux types de données à partir d'une classe de base commune sans avoir besoin de modifier le code qui manipule la classe de base
- Développement **plus rapide**
- Plus grande **simplicité** et **meilleure organisation** du code
- Programmes plus facilement **extensibles**
- Maintenance du code **plus aisée**

# Polymorphisme : un exemple typique

---

## ■ Exemple : la géométrie

- Stocker des objets Forme de n'importe quel type (Rectangle ou Cercle) puis les afficher



# Polymorphisme : un exemple typique

## ■ Exemple (suite) : la géométrie

```
public class Stockage {  
    private Forme[] liste;  
    private int taille;  
    private int i;  
    public Stockage(int taille) {  
        this.taille = taille;  
        liste = new Forme[this.taille];  
        i = 0;  
    }  
    public void ajouter(Forme f) {  
        if (i < taille) {  
            liste[i] = f;      i++;  
        }  
    }  
    public void afficherForme() {  
        for (int i = 0; i < taille; i++)  
            liste[i].afficher();  
    }  
}
```

Si un nouveau type de Forme est défini, le code de la classe Stockage n'est pas modifié



```
public class Test {  
    public static void main (String[] argv) {  
        Stockage monStock = new Stockage(10);  
        monStock.ajouter(new Cercle(...));  
        monStock.ajouter(new Rectangle(...));  
        Rectangle monRect = new Rectangle(...);  
        Forme tonRect = new Rectangle(...);  
        monStock.ajouter(monRect);  
        monStock.ajouter(tonRect);  
    }  
}
```

---

# **CLASSES ABSTRAITES & INTERFACES**

# Classes abstraites (1)

---

- Une classe **abstraite** est une classe qui ne permet pas d'instancier des objets. Elle ne peut servir que de classe de base pour une dérivation. Elle se déclare ainsi

**abstract class A {...}**

- Dans une classe abstraite, on peut trouver classiquement des méthodes et des champs. Mais on peut aussi trouver des **méthodes dites abstraites**, c'est-à dire dont on ne fournit que la signature et le type de la valeur de retour.
- **Par exemple**

```
abstract class A {  
    public void f() { ... } // f est définie dans A  
    public abstract void g(int n); // g n'est pas définie  
    ... // dans A ; on n'écrit que l'entête  
    ... // ATTENTION : n'oubliez pas le point virgule  
}
```

# Classes abstraites (2)

---

## Quelques règles

- Dès qu'une classe comporte une ou plusieurs méthodes abstraites, elle est abstraite, et ce même si l'on n'indique pas le mot clé **abstract** devant sa déclaration (ce qui reste quand même vivement conseillé).
- Ceci est correct      

```
class A {  
    public abstract void f(); // OK  
    ...  
}
```
- Une méthode abstraite doit obligatoirement être déclarée **public**, ce qui est logique puisque sa vocation est d'être redéfinie dans une classe dérivée.

# Classes abstraites (3)

---

- Dans l'en-tête d'une méthode déclarée abstraite, les noms d'arguments muets doivent figurer (bien qu'ils ne servent à rien) :

```
abstract class A {  
    public abstract void g(int);  
    // erreur : nom d'argument (fictif) obligatoire  
}
```

- Une classe dérivée d'une classe non abstraite peut être déclarée abstraite et/ou contenir des méthodes abstraites

# Classes abstraites (Exemple)

```
abstract class Affichable {  
    public abstract void afficher();  
}  
  
class Entier extends Affichable {  
    private int valeur;  
    public Entier (int n) { valeur = n; }  
    public void afficher () {  
        System.out.println("Je suis un Entier de valeur  
:" + valeur);  
    }  
}  
  
class Reel extends Affichable {  
    private float valeur;  
    public Reel (float n) { valeur = n; }  
    public void afficher () {  
        System.out.println("Je suis un Reel de valeur :" + valeur);  
    }  
}
```

```
public class tableauheterogene {  
    public static void main (String [] args){  
        Affichable [] tabhet;  
        tabhet = new Affichable [3];  
        tabhet[0]=new Entier(5);  
        tabhet[1]=new Reel(12.34f);  
        //tabhet[2]=new Affichable(); /*  
        erreur: (classe abstraite)*/  
        tabhet[2]=new Entier(10);  
        for (int i=0; i<3;i++)  
            tabhet[i].afficher();  
    }  
}
```

Je suis un Entier de valeur :5  
Je suis un Reel de valeur :12.34  
Je suis un Entier de valeur :10

# Classes abstraites (3)

---

## ■ Remarques:

- Si on veut empêcher la création d'instances d'une classe on peut la déclarer abstraite même si aucune de ses méthodes n'est abstraite.
- Une méthode static ne peut être abstraite (car on ne peut redéfinir une méthode static)
- Les méthodes abstraites sont particulièrement utiles pour mettre en œuvre le polymorphisme.

# Les interfaces : définition

---

- Quand toutes les méthodes d'une classe sont abstraites et qu'il n'y a aucun attribut, on aboutit à la notion **d'interface**.
- Une interface est un prototype de classe.
- Elle définit la signature des méthodes qui doivent être implémentées dans les classes construites à partir de ce prototype.
- Une **interface** est une “classe” purement abstraite dont toutes les méthodes sont abstraites et publiques (les mots-clés abstract et public sont optionnels).

**interface NomInterface { ... }**

# Les interfaces

---

- Garantir aux utilisateurs d'une classe que ses instances peuvent assurer certains services, ou qu'elles possèdent certaines propriétés (par exemple, être comparable à d'autres instances).
- Pour préciser qu'une class implémente une interface, on utilise le mot-clé **implements** :

**class C implements Interf { ... }**

- Attention : la classe doit implémenter toutes les méthodes de l'interface, sinon elle doit être déclarée abstract :

**Abstract class C implements Interf { ... }**

- Une classe peut implémenter plusieurs interfaces :

**class C implements Interf1, inetrif2,...{ ... }**

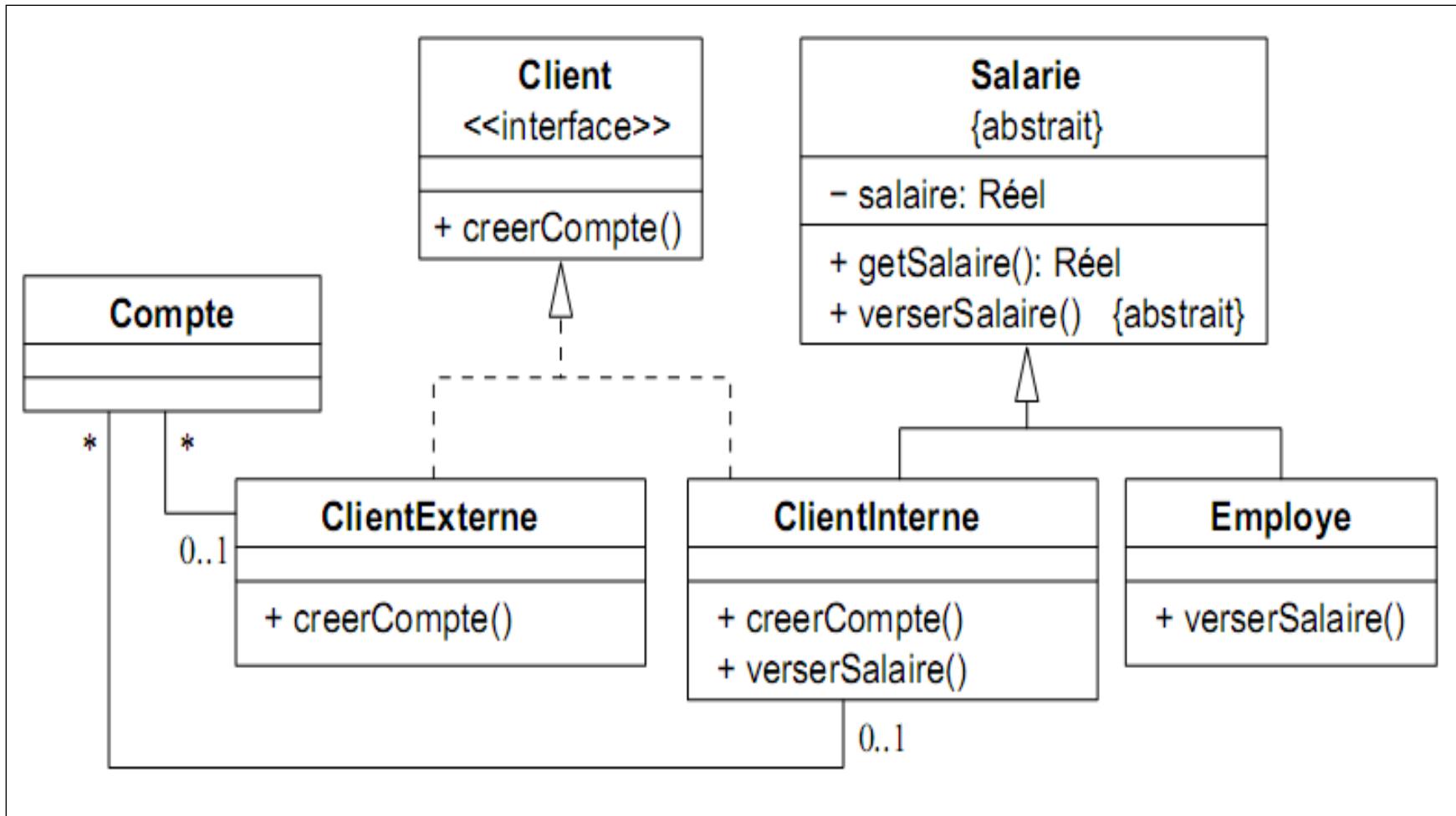
- Une classe peut hériter d'une autre classe et implémenter une ou plusieurs interfaces : **class C extends B implements A{ ... }**

# Les interfaces : implémentation

---

- Une interface ne possède pas de d'attribut. **Les interfaces ne sont pas instanciables** (comme les classes abstraites).
- Une interface n'a pas de constructeurs.
- Une interface peut définir des constantes, c'est-à-dire définir des variables déclarées publiques, statiques et finale et ayant une valeur constante d'affectation.
- On ne donne que la signature des méthodes qui sont nécessairement des méthodes d'instance publiques et abstraites.

# Les interfaces : Exemple



# Les interfaces : Exemple

---

```
public interface  
Client  
{  
    public void  
    creerCompte();  
}
```

```
public class ClientInterne extends Salarie  
implements Client  
{  
    private Compte compte;  
    public void creerCompte() {  
        compte = new Compte();  
    }  
    public void verserSalaire() {  
        compte.credite(salaire);  
    }  
}
```

---

# **LES COLLECTIONS**

**Tableaux, Listes, Ensembles, Listes associatives**

# Les collections: définition (1)

---

## ■ Structures de données

- C'est l'organisation efficace d'un ensemble de données, sous la forme de tableaux, de listes, de piles etc.
- Cette efficacité réside dans la quantité mémoire utilisée pour stocker les données, et le temps nécessaire pour réaliser des opérations sur ces données.

## ■ Collection

- Une collection est un objet qui regroupe plusieurs objets.
- Une collection permet de stocker, récupérer, manipuler et communiquer un ensemble de données

## ■ Objectifs

- adapter la structure collective aux besoins de la collection
- ne pas re-programmer les traitements répétitifs classiques (affichage, saisie, recherche d'éléments, ...)

# Les collections: définition (2)

---

- **Exemples:**
  - collection de joueurs
  - collection d'objets d'art
  - collection de voitures
- Les tableaux **Array** sont des exemples d'implémentation d'une collection
- La Classe **Vector** en est une autre
- Le package **java.util** contient un ensemble de classes et interfaces permettant de créer des collections.
- **Exemples:**
  - **ArrayList**, **Vector**, **HashSet**, **LinkedList**, **TreeSet**, **Arrays**,...
  - **List**, **Map**, **Set**, **SortedMap**, **SortedSet**

# Collections & Java

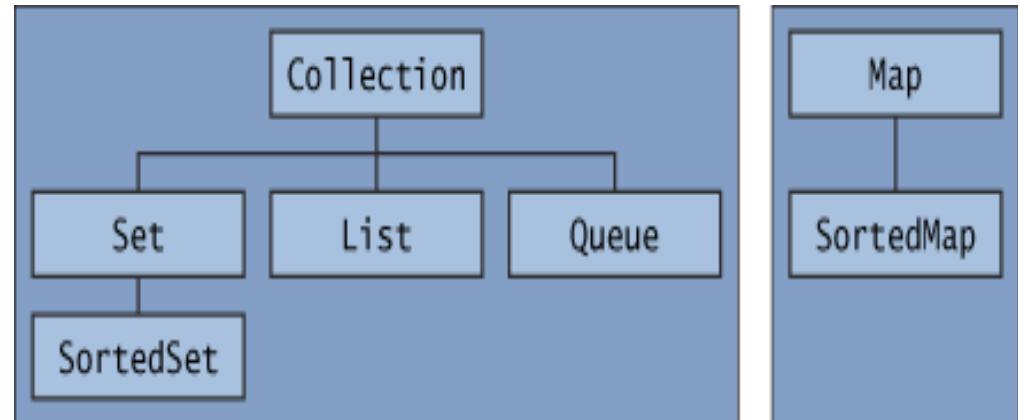
---

- Une **collection** gère un groupe d'un ensemble d'objets d'un type donné ; ou bien c'est un objet qui sert à stocker d'autres objets.
- Dans les premières versions de Java, les collections étaient représentées par les "**Array**", "**Vector**", "**Stack**" etc.
- Puis avec Java 1.2 (Java 2), est apparu le **framWork** de collections qui tout en gardant les principes de bases, il a apporté des modifications dans la manière avec laquelle ces collections ont été réalisées et hiérarchisées.
- Tout en collaborant entre elles, ces collections permettent de réaliser dans des catégories de logiciels des conceptions réutilisables.

# Différents types de collections(1)

2 hiérarchies principales :

- **Map** : collections indexées par des clés
- **Collection**



- **Collection**: un groupe d'objets où la duplication peut-être autorisée.
- **Map**: est un groupe de paires contenant une **clé** et une **valeur** associée à cette clé.
  - Cette interface n'hérite ni de Set ni de Collection.
  - La raison est que Collection traite des données simples alors que **Map** traite des données composées (clé,valeur). **SortedMap** est un Map trié.

# Différents types de collections(2)

---

- **Set**: est ensemble ne contenant que des valeurs et ces valeurs ne sont pas dupliquées.
  - Par exemple l'ensemble **A = {1,2,4,8}**. Set hérite donc de Collection, mais n'autorise pas la duplication. **SortedSet** est un Set trié.
- **List**: hérite aussi de collection, mais autorise la duplication. Dans cette interface, un système d'indexation a été introduit pour permettre l'accès (rapide) aux éléments de la liste.
  - **Classes implémentées** : **ArrayList** (tableau à taille variable), **LinkedList** (liste chaînée)
  - En règle générale, on utilise **ArrayList** mais **LinkedList** est utile si il y a beaucoup d'opérations 'insertions/suppressions (évite les décalages)

# Différents types de collections(2)

---

## ■ Collection

- **List:** structure séquentielle, l'utilisateur contrôle l'ordre des éléments.
  - **ArrayList:** implémentation dans tableau de taille variable, accès par indice
  - **Vector:** toujours implémentation dans tableau ... (plus vieux)
  - **LinkedList:** liste chaînée : implémentation par double chaînage et pointeurs
- **Queue:** FIFO, file à priorité
- **Set :** ensemble (pas deux éléments identiques)
  - **HashSet:** ensemble non ordonné, implémenté par table de hachage
  - **TreeSet:** ensemble ordonné implémenté par arbre binaire de recherche équilibré

## ■ Map<K, V> ensemble associatif

- **HashMap<K,V>** : les clés sont un ensemble non ordonné
- **TreeMap<K,V>** : item les clés sont un ensemble ordonné

# Les collections : les méthodes générales

---

## ■ Méthodes de la classe collection

- **boolean isEmpty()** : test si le conteneur est vide
- **int size()** : renvoi le nombre d'éléments du conteneur
- **boolean add(Object)** : ajoute un élément au conteneur
- **boolean addAll(Collection)** : ajoute tous les éléments d'une collection au conteneur
- **boolean remove(Object)** : supprime un élément au conteneur
- **boolean removeAll(Collection)** : supprime tous les éléments d'une collection du conteneur
- **void clear()** : supprimer tous les éléments du conteneur
- **boolean contains(Object)** : appartenance d'un élément au conteneur
- **Object [] toArray()** : transforme une collection en tableau

## ■ Méthodes statiques : algorithmes génériques

- **void sort(List)**
- **Object max(Collections), Object min(Collections)**
- **void reverse(list)**
- **void rotate(collection, distance)**

# Opération sur les ensembles [Set]

---

- **s1.containsAll(s2):** teste l'inclusion de s2 dans s1
  - **s1.addAll(s2) :**union de s1 et s2
  - **s1.retainAll(s2) :**intersection de s1 et s2
  - **s1.removeAll(s2) :**différence entre s1 et s2
- !\!** ne donne pas le même résultat que **s2.removeAll(s1)**

# Exemple d'ArrayList

---

```
import java.util.ArrayList;
public class TestPoint {
    public static void main(String args[]) {
        ArrayList liste = new ArrayList();
        liste.add(new Point(12,13));
        // création d'autres instances de Point
        for (int cpt=0; cpt<liste.size();cpt++)
            System.out.println(((Point)liste.get(cpt)).toString());
    }
}
```

# Interface Iterator

---

- L'interface collection est dotée d'une instance d'une classe qui implante l'interface **Iterator**.
- C'est l'outil utilisé pour parcourir une collection. L'interface **Iterator** contient ce qui suit:

```
public interface Iterator {  
    boolean hasNext();  
    Object next();  
    void remove(); // Optional  
}
```

- **hasNext** permet de vérifier s'il y a un élément qui suit.
- **next** permet de pointer l'élément suivant.
- **remove** permet de retirer l'élément courant.

# Exemple d'ArrayList avec Iterator

---

```
import java.util.ArrayList;
import java.util.Iterator;
public class TestPoint {
    public static void main(String args[])  {
        ArrayList<Point> liste = new ArrayList<Point>();
        liste.add(new Point(12,13));
        // création d'autres instances de Point
        Iterator<Point> it = liste.iterator();
        while(it.hasNext()) {
            Point tmp = it.next();
            System.out.println(tmp.toString());
        }
    }
}
```

# Exemple de LinkedList

```
import java.util.LinkedList;
public class TestPoint {
    public static void main(String args[])  {
        LinkedList maListe=new LinkedList() ; // on crée notre liste chaînée
        maListe.add(new Integer(1)) ; // on ajoute l'entier 1 à la liste
        maListe.add(new Float(2.15)) ; // on ajoute le flottant 2.15 à la liste
        /* On remarque ici que l'entier 1 est la tête de la liste, et que le flottant
           est la queue de la liste. */
        Integer monEntier=(Integer)maListe.getFirst() ;
        Float monFloat=(Float)maListe.getLast();
        System.out.println(maListe); //affiche [1, 2.15]
        maListe.remove(0) ; // on retire l'entier , le flottant devient la tête
        System.out.println(maListe.getFirst()); //affiche 2.15
    }
}
```

# Exemple de Set

---

```
import java.util.*;  
public class Test {  
    public static void main(String args[]) {  
        Set set = new HashSet(); // Une table de Hachage  
        set.add("Hamid");  
        set.add("Ali");  
        set.add("Brahim");  
        set.add("Hicham");  
        set.add("Fatima");  
        System.out.println(set); // Affiche: [Hamid, Brahim, Ali, Hicham, Fatima]  
        Set SetTrie = new TreeSet(set); // Un Set trié  
        System.out.println(SetTrie); // Affiche [Ali, Brahim, Fatima, Hamid, Hicham]  
    }  
}
```

# Exemple de Map

---

```
import java.util.Map;
import java.util.HashMap;
public class TestPoint {
    public static void main(String args[])  {
        Map map = new HashMap() ;
        map.put(1,"Janvier");
        map.put(2,"Fevrier");
        map.put(3,"Mars");
        map.put(4,"Avril");
        map.put(5,"Mai");
        map.put(6,"Juin");
        map.put(7,"Juillet");
        System.out.println(map); // {1=Janvier, 2=Fevrier, 3=Mars, 4=Avril, 5=Mai,
                                // 6=Juin, 7=Juillet,}
        System.out.println(map.get(5)); // affiche Mai
    }
}
```

## Exemple

Soit la classe Etudiant suivante:

Etudiant	
mat	int
nom	String
moy	float
 Etudiant() Etudiant(int,String,float) Afficher() String toString()	

- Ecrire la classe Etudiant.
- Ecrire la classe TestEtudiant qui contient un main permettant :
  - o Créer une collection ArrayList
  - o Ajouter quelque Etudiants
  - o Remplacer un étudiant
  - o Supprimer un étudiant
  - o Afficher la liste des étudiants

---

### TestEtudiant.java

```
import java.util.ArrayList;
public class TestEtudiant {

    static ArrayList L;

    public static void main(String args[])
    {
        //Remplir la liste (ArrayList)
        L=new ArrayList();

        Etudiant e;
        e=new Etudiant(100,"Alami",14.25f);
        L.add(e);
        L.add(new Etudiant(110,"Fikri",16));
        e=new Etudiant (130,"Chaouki",15.5f);
        L.add(e);

        //Afficher les éléments de la liste
        System.out.println("Liste des étudiants:");
        Afficheliste();

        //remplacer le 3ime étudiant par un autre
        e=new Etudiant(150,"Salimi",9);
        L.set(2,e);
        System.out.println("Liste des étudiants après modification:");
        Afficheliste();

        System.out.println("Liste des étudiants après suppression:");
        e=(Etudiant)L.remove(1);
        Afficheliste();

        System.out.println("L'étudiant supprimé est :" + e.toString());
    }

    static void Afficheliste()
    {
        for (int i=0;i<L.size();i++)
            ((Etudiant)L.get(i)).afficher();
    }
}
```

---

# **LES EXCEPTIONS**

**Traitement des erreurs  
en Java**

# C'est quoi une exception?

---

- Une **exception** est un événement (une **erreur**) qui se produit lors de l'exécution d'un programme, et qui va provoquer un fonctionnement anormal (par exemple l'arrêt du programme) de ce dernier.
- Peut intervenir à la demande du programmeur ou automatiquement quand des événements se produisent, par exemple
  - Division par zéro
  - des erreurs d'entrée-sortie (I/O fichiers)
  - des erreurs de saisie de données par l'utilisateur
  - des erreurs matérielles (crash du disque, ...)
  - des erreurs de programmation (indice d'un tableau hors limites, ...)
  - des erreurs liées à l'environnement d'exécution (mémoire insuffisante, ...)

# C'est quoi une exception?

```
public class Test{  
    static void main (String [] args) {  
        int c=1,a=1,b=0;  
        c= a/b;  
        System.out.println("res: " + c);  
        System.exit(0);  
    }  
}
```

Le système affiche l'erreur suivante:

Exception in thread "main" java.lang.ArithmetricException: / by zero  
at Test.main(Test.java:4)

# Gestion des exceptions

---

- La gestion des exceptions se substitue en quelque sorte à l'algorithmique permettant la gestion des erreurs.
- Dans l'exemple précédent, si nous avons voulu anticiper sur la gestion de l'erreur, il fallait prévoir un traitement adéquat pour contrer la division par zéro:

```
if (b!=0)
    c=a/b;
else
    // traitement de l'erreur.
```

- Le traitement de l'erreur pourrait consister à retourner une valeur:  
-1 pour une division par zéro, -2 pour un index qui déborde etc.
- Ce traitement devient fastidieux à la longue!

# Gestion des exceptions en java

---

- Le langage Java offre un mécanisme très souple pour la gestion des erreurs.
- Ce mécanisme permet d'isoler d'une part la partie du code générant l'erreur du reste du programme, et d'autre part de dissocier les opérations de détection et de traitement de cette erreur.
- Par ailleurs, le langage Java utilise des objets pour représenter les erreurs (**exceptions**) et l'héritage pour hiérarchiser les différents types d'exception.
- La gestion des erreurs consiste donc à définir le bloc pouvant provoquer l'erreur (le bloc **try**), et attraper (**catch**) les "objets" représentant les exceptions générées.

# Gestion des exceptions en java

```
public class Test {  
    public static void main (String [] args) {  
        int c=0,a=1,b=0;  
        try {  
            c= a/b;  
        }catch (ArithmetricException e) {  
            System.out.println("Erreur a été capturée");  
        }  
        System.out.println("res: " + c);  
        System.exit(0);  
    }  
}
```

Le programme affiche:

Erreur a été capturée  
res: 0

# Mécanisme du traitement d'exception

---

- Lorsqu'une exception est **levée** dans une méthode donnée, les instructions qui suivent le lancement de l'exception, se trouvant dans cette méthode, sont ignorées.
- L'exception peut-être **attrapée** par un bloc **catch** (s'il existe un **try**) se trouvant dans cette méthode.

# Traiter une exception

---

- Les sections **try** et **catch** servent à capturer une exception dans une méthode
- Exemple :

```
public void fonction(.....) {  
    try{  
        .....  
    }  
    catch {  
        .....  
        .....  
    }  
}
```



On tente de récupérer là.

# Traitement des exceptions

---

- Le bloc **try** est exécuté jusqu'à ce qu'il se termine avec succès ou bien qu'une exception soit levée.
- Dans ce dernier cas, les clauses **catch** sont examinées l'une après l'autre dans le but d'en trouver une qui traite cette classe d'exceptions (ou une superclasse).
- Les clauses **catch** doivent donc traiter les exceptions de la plus spécifique à la plus générale.
- Si une clause **catch** convenant à cette exception a été trouvée et le bloc exécuté, l'exécution du programme reprend son cours.

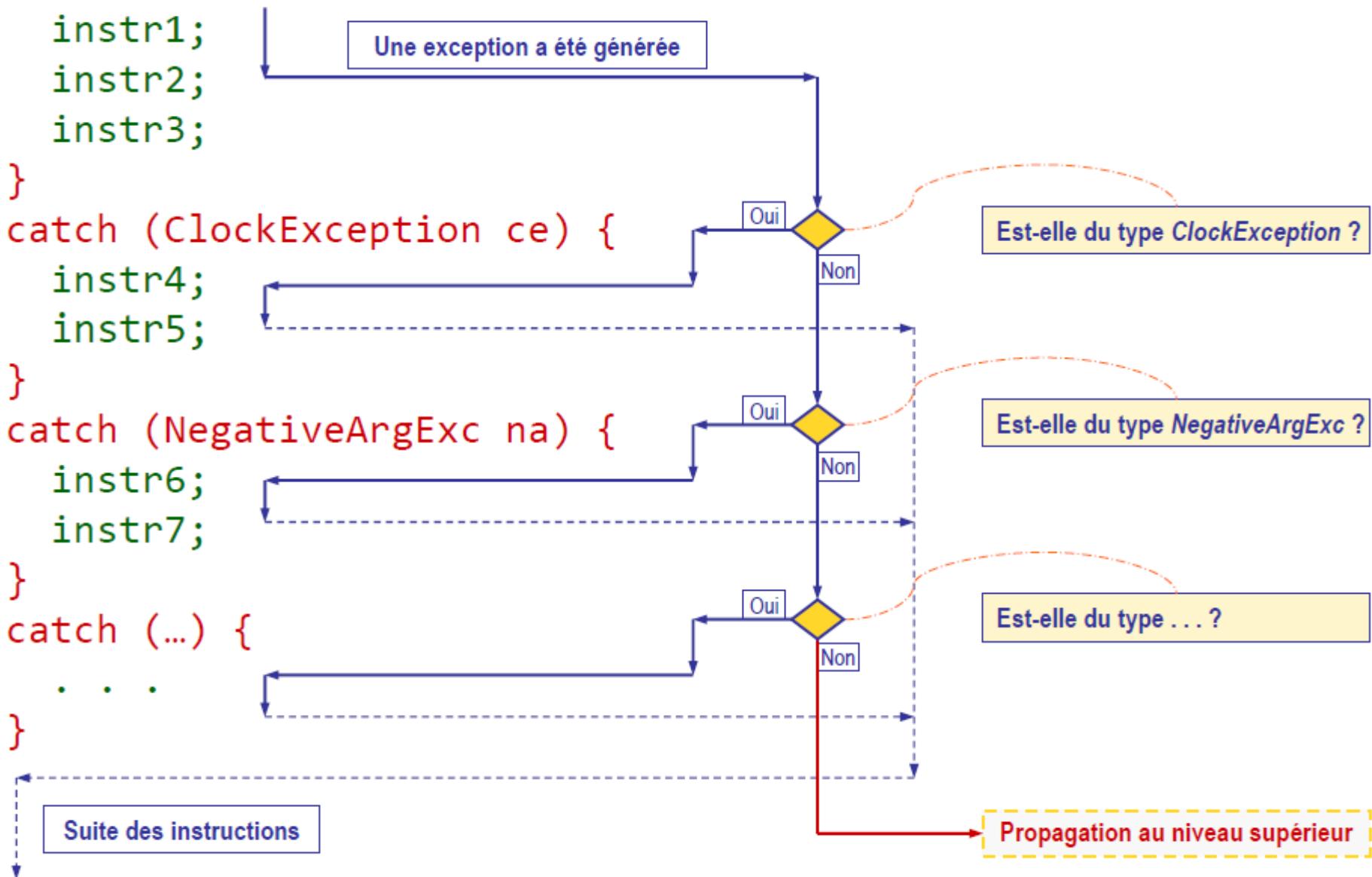
# Traitement des exceptions

---

- Si elles ne sont pas immédiatement capturées par un bloc **catch**, les exceptions se propagent en remontant la pile d'appels des méthodes, jusqu'à être traitées.
- Si une exception n'est jamais capturée, elle se propage jusqu'à la méthode **main()**, ce qui pousse l'interpréteur Java à afficher un message d'erreur et à s'arrêter.
- L'interpréteur Java affiche un message identifiant :
  - l'exception,
  - la méthode qui l'a causée,
  - la ligne correspondante dans le fichier.

# Traiter une exception (Résumé)

```
try {  
    instr1;  
    instr2;  
    instr3;  
}  
catch (ClockException ce) {  
    instr4;  
    instr5;  
}  
catch (NegativeArgExc na) {  
    instr6;  
    instr7;  
}  
catch (...) {  
    ...  
}
```



# try / catch / finally

---

```
try  {  
    ...  
}  
catch (<une-exception>)  {  
    ...  
}  
catch (<une_autre_exception>)  {  
    ...  
}  
...  
}
```

```
finally  {  
    ...  
}
```

- ⇒ Autant de blocs **catch** que l'on veut.
- ⇒ Bloc **finally** facultatif.

# Bloc **finally**

---

- Un bloc **finally** est généralement utilisé pour effectuer des opérations de conclusion (fermeture de fichiers, de connexion réseau, de base de données, etc.) qui devraient être effectuées dans tous les cas de figure.
- Le bloc **finally** évite donc de devoir placer ces instructions de conclusion dans le bloc **try** et dans tous les blocs **catch**.
- Le bloc **finally** est optionnel mais un bloc **try** doit obligatoirement être accompagné d'au moins un bloc **catch** ou d'un bloc **finally** (ou naturellement des deux).

# Générer (lever) une exception (1)

---

- Les exceptions peuvent être générées soit :
  - par le **système**, lors de l'exécution de certaines instructions
  - par l'exécution de l'instruction **throw** (dans les classes pré-définies de la plate-forme Java [librairies] ou dans le code que l'on écrit soi-même)
- Parmi les instructions qui génèrent des exceptions, on peut citer :
  - Division par zéro pour les entiers : **ArithmeticException**
  - Référence nulle : **NullPointerException**
  - Tentative de forçage de type illégale : **ClassCastException**
  - Tentative de création d'un tableau de taille négative :  
**NegativeArraySizeException**
  - Dépassemement de limite d'un tableau : **ArrayIndexOutOfBoundsException**

# Générer (lever) une exception (2)

---

- Pour générer explicitement une exception, on utilise l'instruction **throw** :  
**throw exception\_object ;**
- L'instruction **throw** sert à générer une exception (on dit également **lever** une exception, **lancer** une exception, ...).
- L'expression qui suit l'instruction **throw** doit être un objet qui représente une exception (un objet de type **Throwable**).
- Lors de la création de l'objet exception, on peut généralement lui associer un **message** (**String**) qui décrit l'événement.
- Le mot-clé **throws** (à ne pas confondre avec **throw**) est utilisé dans la déclaration de méthode (**signature**) pour annoncer la liste des exceptions que la méthode peut générer. L'utilisateur de la méthode est ainsi informé des exceptions qui peuvent survenir lors de son invocation et peut prendre les mesures nécessaires pour gérer ces événements exceptionnels

# Générer (lever) une exception (3)

```
public class TestDiv {  
    public int divint (int x, int y) throws ArithmeticException {  
        if (y==0) throw new ArithmeticException("y doit être différent de zéro");  
        return (x/y);  
    }  
    public static void main (String [] args) {  
        int c=0,a=1,b=0;  
        try {  
            c= divint(a,b);  
        } catch (ArithmeticException e) {  
            System.out.println("Erreur a été capturée");  
            System.out.println(e.getMessage());  
        }  
        System.out.println("res: " + c);  
    }  
}
```

Le programme affiche:

Erreur a été capturée  
y doit être différent de zéro  
res: 0

# Générer (lever) une exception (4)

---

- Le programmeur peut lever ses propres exceptions à l'aide du mot réservé **throw**.
- **throw** prend en paramètre un objet instance de **Throwable** ou d'une de ses sous-classes.
- Les objets exception sont souvent instanciés dans l'instruction même qui assure leur lancement.

```
throw new MonException("Mon exception s'est produite !!!");
```

# Générer (lever) une exception (5)

---

- L'exception elle-même est levée par l'instruction **throw**.
- Une méthode susceptible de lever une exception est identifiée par le mot-clé **throws** suivi du type de l'exception

Exemple :

```
public void ouvrirFichier(String name) throws MonException  
{if (name==null) throw new MonException();  
 else  
 {...}  
 }
```

# throws (1)

---

- Pour "laisser remonter" à la méthode appelante une exception qu'il ne veut pas traiter, le programmeur rajoute le mot réservé **throws** à la déclaration de la méthode dans laquelle l'exception est susceptible de se manifester.

```
public void uneMethode() throws IOException
{
    // ne traite pas l'exception IOException
    // mais est susceptible de la générer
}
```

# **throws**

---

- Les programmeurs qui utilisent une méthode connaissent ainsi les exceptions qu'elle peut lever.
- La classe de l'exception indiquée peut tout à fait être une super-classe de l'exception effectivement générée.
- Une même méthode peut tout à fait "laisser remonter" plusieurs types d'exceptions (séparés par des ,).

# Les objets Exception

---

- La classe **Exception** hérite de La classe **Throwable**.
- La classe **Throwable** définit un message de type **String** qui est hérité par toutes les classes d'exception.
- Ce champ est utilisé pour stocker le message décrivant l'exception.
- Il est positionné en passant un argument au constructeur.
- Ce message peut être récupéré par la méthode **getMessage()**.

# Hiérarchie des exceptions

**Throwable** est la classe de base, à partir de laquelle vont dériver toutes les exceptions.

**Error**: Elle gère les erreurs liées à la machine virtuelle (LinkageError, ThreadDeath etc.)

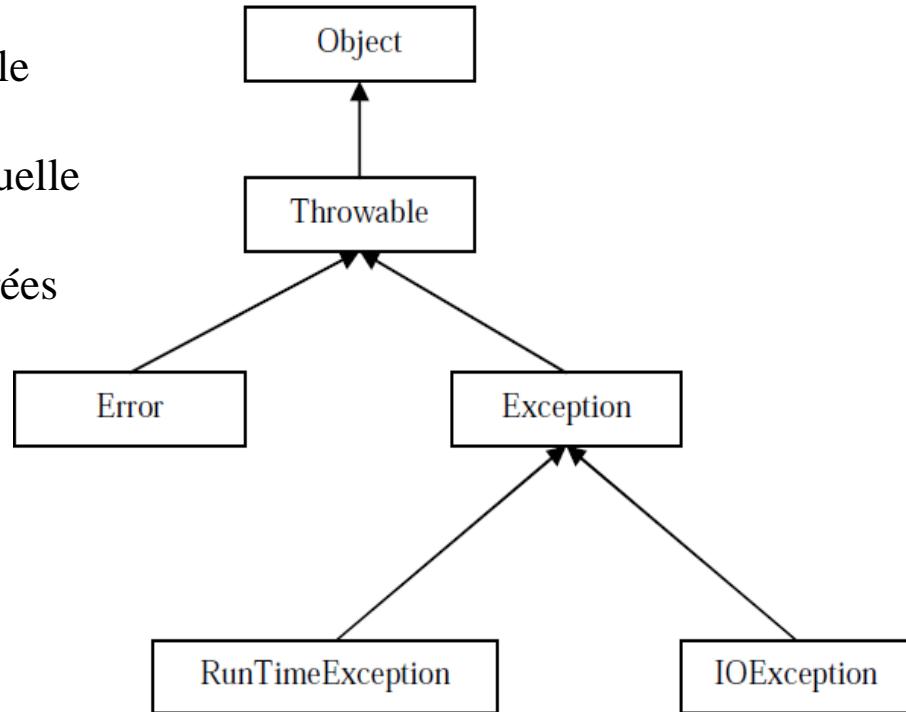
**Exception**: contient l'ensemble des exceptions gérées par le programmeur (ArithmeticException etc.).

**RunTimeException**: regroupe les erreurs de base (ArithmeticException etc.)

**IOException**: regroupe les erreurs entrée/sortie.

**Error** et **RunTimeException** appartiennent à la catégorie "unchecked" donc "throws" n'est pas nécessaire à coté de la méthode qui lance une exception de cette catégorie là.

Toutes les autres exceptions (y compris donc celles créées par le programmeur) appartiennent à la catégorie des "checked" où l'utilisation de "throws" est exigée.



# Exemple

---

```
public class MonException extends Exception
{
    public MonException()
    {
        super();
    }
    public MonException(String s)
    {
        super(s);
    }
}
```

# Exemple

```
public class MonException extends  
Exception {  
    public MonException() {  
        super();  
    }  
    public MonException(String s) {  
        super(s);  
    }  
    public void message(){  
        System.out.println("Erreur a été  
capturée dans MonException");  
    }  
}
```

```
public class Test {  
    public static int divint (int x, int y) throws  
MonException {  
    if (y==0) throw new MonException("y doit etre  
différent de zero");  
    return (x/y);  
}  
    public static void main (String [] args) {  
        int c=0,a=1,b=0;  
        try {    c= divint(a,b);  
        }  
        catch (MonException e) {  
            System.out.println("Erreur a été capturée");  
            System.out.println(e.getMessage());  
            e.message();  
        }  
    }  
}
```

Le programme affiche:

Erreur a été capturée  
y doit etre différent de zero  
Erreur a été capturée dans MonException  
res: 0

# Conclusion

---

- Grâce aux exceptions, Java possède un mécanisme sophistiqué de gestion des erreurs permettant d'écrire du code « robuste »
- Le programme peut déclencher des exceptions au moment opportun.
- Le programme peut capturer et traiter les exceptions grâce au bloc d'instruction **catch** ... **try** ... **finally**
- L'instruction **throw** sert à générer une exception
- Le programmeur peut définir ses propres classes d'exceptions