



# Formation Java



# **1.Présentation, Principes**

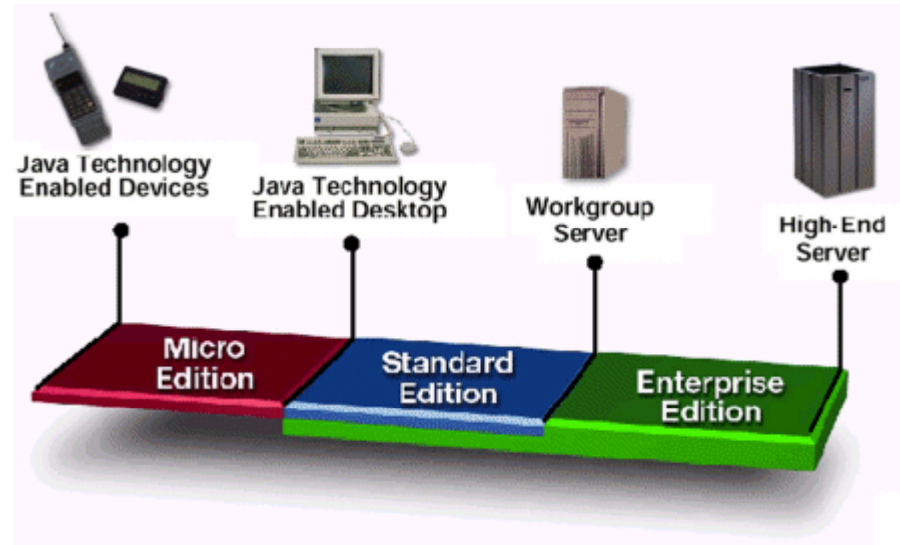
# Principes Java

- **Java** est le nom d'une technologie mise au point par Sun Microsystems (racheté depuis 2010 par Oracle)
- Permet de produire des logiciels indépendants de toute architecture matérielle (donc portables)
- le **langage Java** est un langage de programmation orienté objet
- Utilise des programmes générés dans un byteCode (fichier .class)
- Les programmes java s'exécutent dans une machine virtuelle



# Java - Présentation

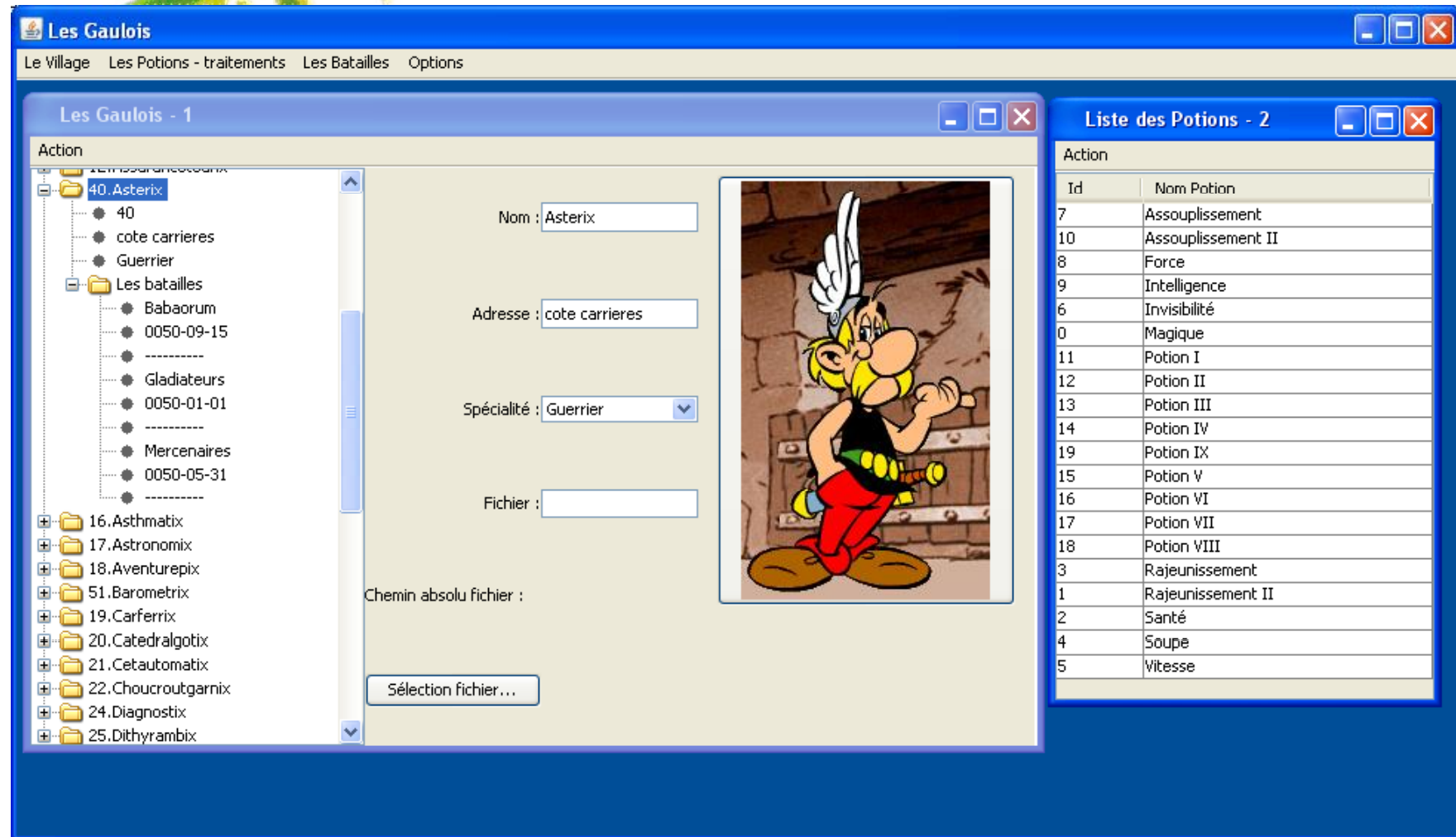
- La **plate-forme Java** correspond à la machine virtuelle Java plus des spécifications d'API :
  - **Java Platform, Standard Edition (Java SE)** contient les API de base et est destiné aux ordinateurs de bureau
  - **Java Platform, Enterprise Edition (Java EE)** contient, en plus du précédent, les API orientées entreprise et est destiné aux serveurs
  - **Java Platform, Micro Edition (Java ME)** est destiné aux appareils mobiles tels que assistants personnels ou Smartphones



- **Java Platform, Standard Edition (Java SE) :**
  - Création de **clients lourds ou riches** :
    - En installation locale (SWING / SWT)
    - En chargement depuis un serveur via internet (Java Web Start)
  - Création d'**applet**, composant chargeable et interprétable sur un poste via un navigateur

# Java J2SE – Exemple SWING

## ■ Client Riche :





# Java J2SE – Exemple d'applet

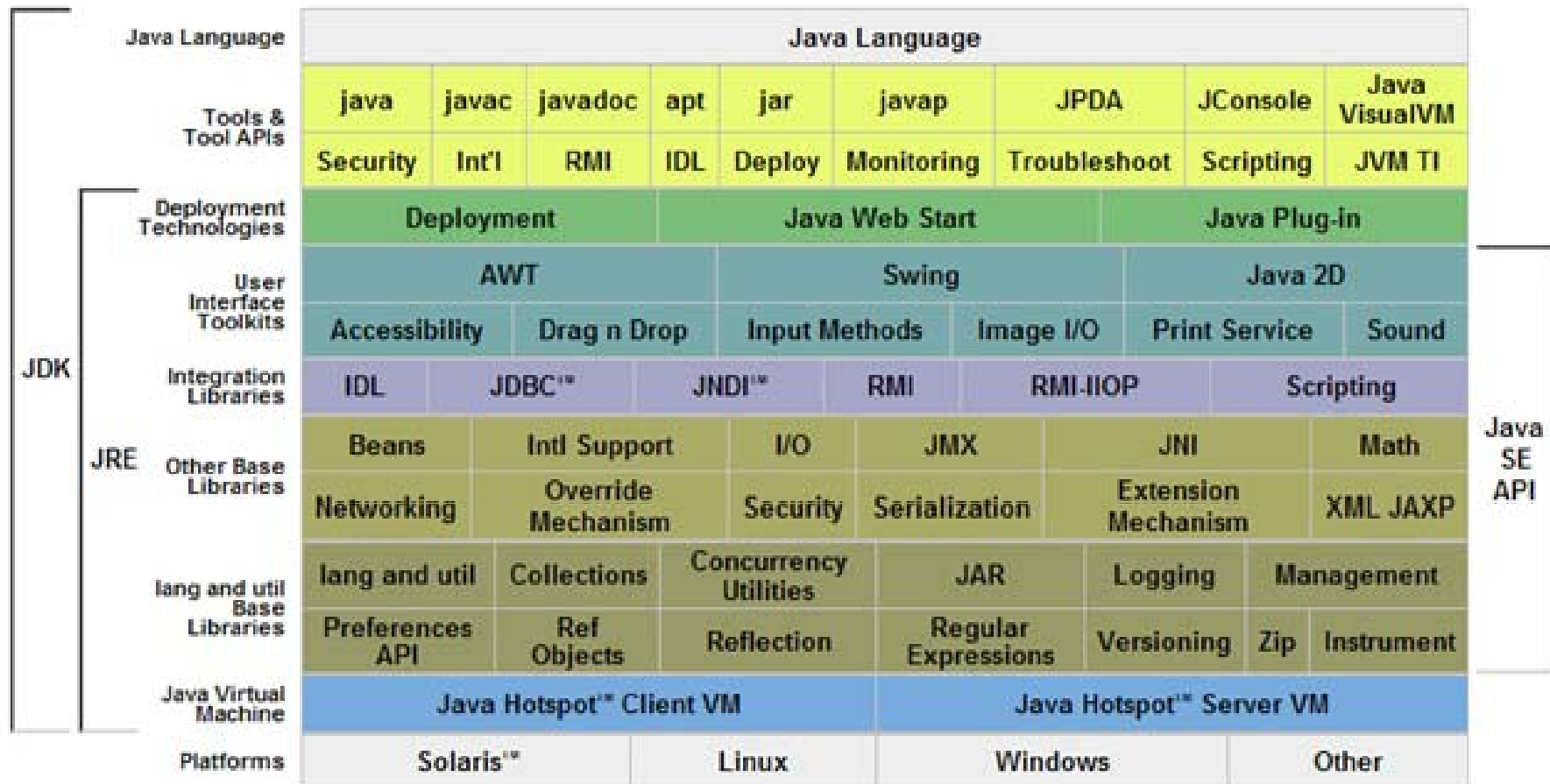
- Prendre : **Exemple Applet**
- Lancement : **`appletviewer AnimImage.class`**



- Le JDK de Sun comporte des outils de compilation et de test des applets et des applications Java.
- Quelques plates-formes de développement Java :
  - **WELOGIC (BEA)**
  - **JBUILDER**
  - **ECLIPSE (Open Source projet Eclipse)**
  - **NetBeans**
  - ...



# JDK - JRE



- Concepts de Java **hérités de C++**, mais aussi d'autres langages orientés objet.
- Comme la plupart de ces langages, Java possède :
  - **bibliothèques de classes (packages)**, qui fournissent les types de données de base
  - possibilités d'entrées/sorties du système
  - d'autres fonctions utilitaires.
- Ces bibliothèques de base font partie de **JDK (Java Developer Kit)**, qui comprend également des classes pour la gestion de réseau, des protocoles Internet communs et des outils d'interface utilisateur.
- Ces bibliothèques sont écrites en langage Java, et sont donc **portables sur toutes les plates-formes**.

- **L'écriture d'un programme Java :**

- ➔ Conception et construction de **plusieurs classes**.

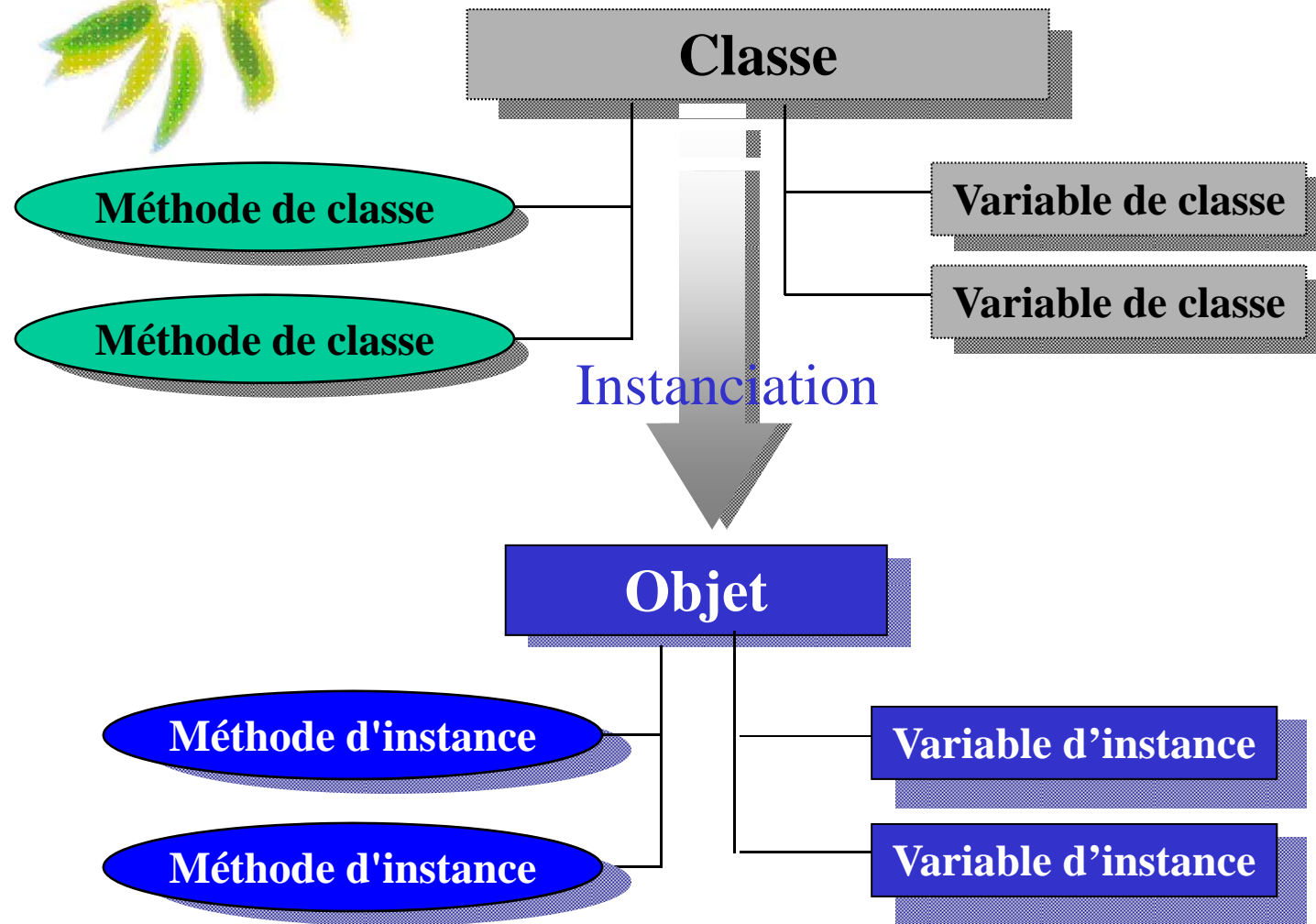
- Ainsi, les programmes s'exécutent, des instances de ces classes se créent et s'effacent au fur et à mesure des besoins.

- **Travail programmeur Java :**

- ➔ Utiliser, créer le bon ensemble de classes pour que les programmes puissent fonctionner correctement.

- L'environnement Java offre un ensemble standard de classes (une bibliothèque de classes) couvrant un grand nombre de comportements de base.

# Attribut et comportement





## **2. Classes et méthodes**



# Création d'une classe

- La création de la classe Moto montre comment définir les méthodes et les variables d'instance dans une classe.
- Exemple d'une création de classe :

```
class Moto  
{  
    //...  
}
```

- C'est une classe Java dans sa version la plus simple.





## Variables d'instance

- Création des variables d'instance pour cette classe :

```
String marque;  
String couleur;  
boolean moteur_en_marche;
```

- Parmi les trois variables d'instance de l'exemple, marque et couleur contiennent **des objets String**. (Chaîne de caractères ; String s'écrit avec un S majuscule et fait partie de la bibliothèque de classes standard).
- La troisième, moteur\_en\_marche, est une **variable booléenne** (type de données primaire)

- Les comportements (méthodes) de la classe Moto peuvent être nombreux, mais l'exemple se limite à un seul : une méthode qui démarre le moteur.

```
void demarrer_moteur()  
{  
    if (moteur_en_marche == true)  
        System.out.println("Le moteur tourne deja");  
    else  
    {  
        moteur_en_marche = true;  
        System.out.println("Le moteur est maintenant en  
        marche.");  
    }  
}
```

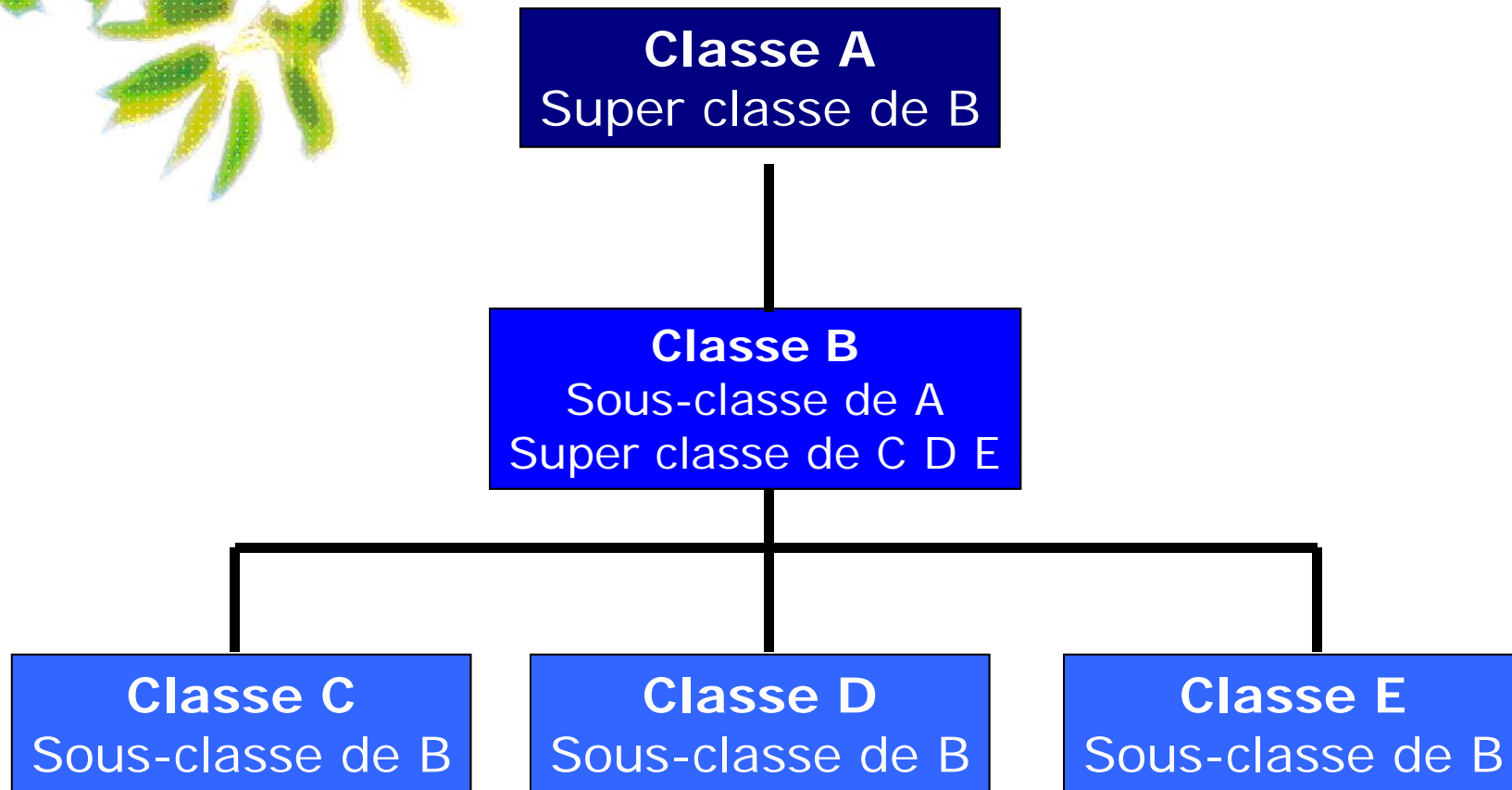
*NB : parenthèses vides à la suite du nom d'une méthode (convention pour indiquer qu'il s'agit d'une méthode et non d'une variable).*



## Exemple de classe

```
1: class Moto
2: {
3:   String marque;
4:   String couleur;
5:   boolean moteur_en_marche = false;
6:
7:   void demarrer_moteur()
8:   {
9:     if (moteur_en_marche == true)
10:      System.out.println("Le moteur tourne déjà.");
11:     else
12:     {
13:       moteur_en_marche = true;
14:       System.out.println("Le moteur est maintenant démarré");
15:     }
16:   }
17: }
```

- Essentiel pour la programmation orientée objet
- Influence directement la conception et l'écriture des classes Java.
- Mécanisme puissant qui facilite l'écriture d'une classe.
- Il donne un **accès automatique à toute l'information** contenue dans cette autre classe, il suffit de spécifier les différences de cette classe par rapport à une autre
- Avec l'héritage, toutes les classes (écrites ou issues d'autres bibliothèques de classes ainsi que les classes utilitaires standard) suivent une **hiérarchie stricte**.



- Chaque classe possède une et une seule super-classe, (située au-dessus dans la hiérarchie)
- Eventuellement une ou plusieurs sous-classes (les classes situées au-dessous dans la hiérarchie).
- **Les classes héritent de celles qui sont situées au-dessus d'elles dans la hiérarchie.**
- **Les sous-classes héritent de toutes les méthodes et variables de leurs super-classes.**
- Ainsi, si une super-classe définit un comportement utile à une classe,
  - la sous-classe récupère automatiquement le comportement à partir de sa super-classe,
  - et ainsi de suite jusqu'au sommet de la hiérarchie.



- Une classe devient alors une combinaison de toutes les caractéristiques de la hiérarchie de classes dont elle hérite.
- Les sous-classes héritent des attributs et des comportements de leur super-classe

**La classe object se trouve au sommet de la hiérarchie des classes Java.**

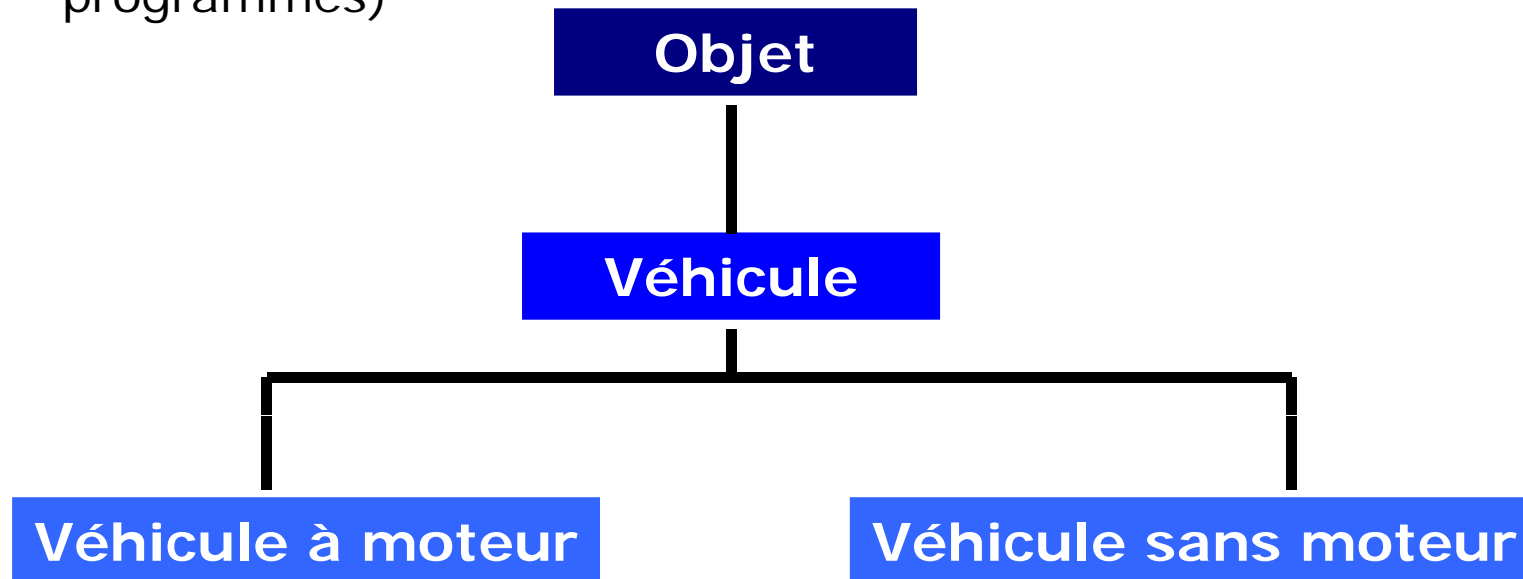
- Classe Object :
  - Toutes les classes héritent de cette super-classe
  - C'est la classe la plus générale de la hiérarchie.
  - Elle définit le comportement spécifique à tous les objets dans Java.

- Par exemple, vous pouvez vouloir créer une version de Moto possédant un carénage personnalisé.
- Pour récupérer toutes les informations Moto, il suffit de définir une classe qui en hérite.
- La classe récupère alors tous les comportements définis dans Moto (et dans les super-classes de Moto).
- Ensuite, il reste à gérer les différences entre cette classe et Moto.

**Le mécanisme, qui consiste à définir de nouvelles classes en fonction de leurs différences avec leurs super-classes, s'appelle sous-classement.**

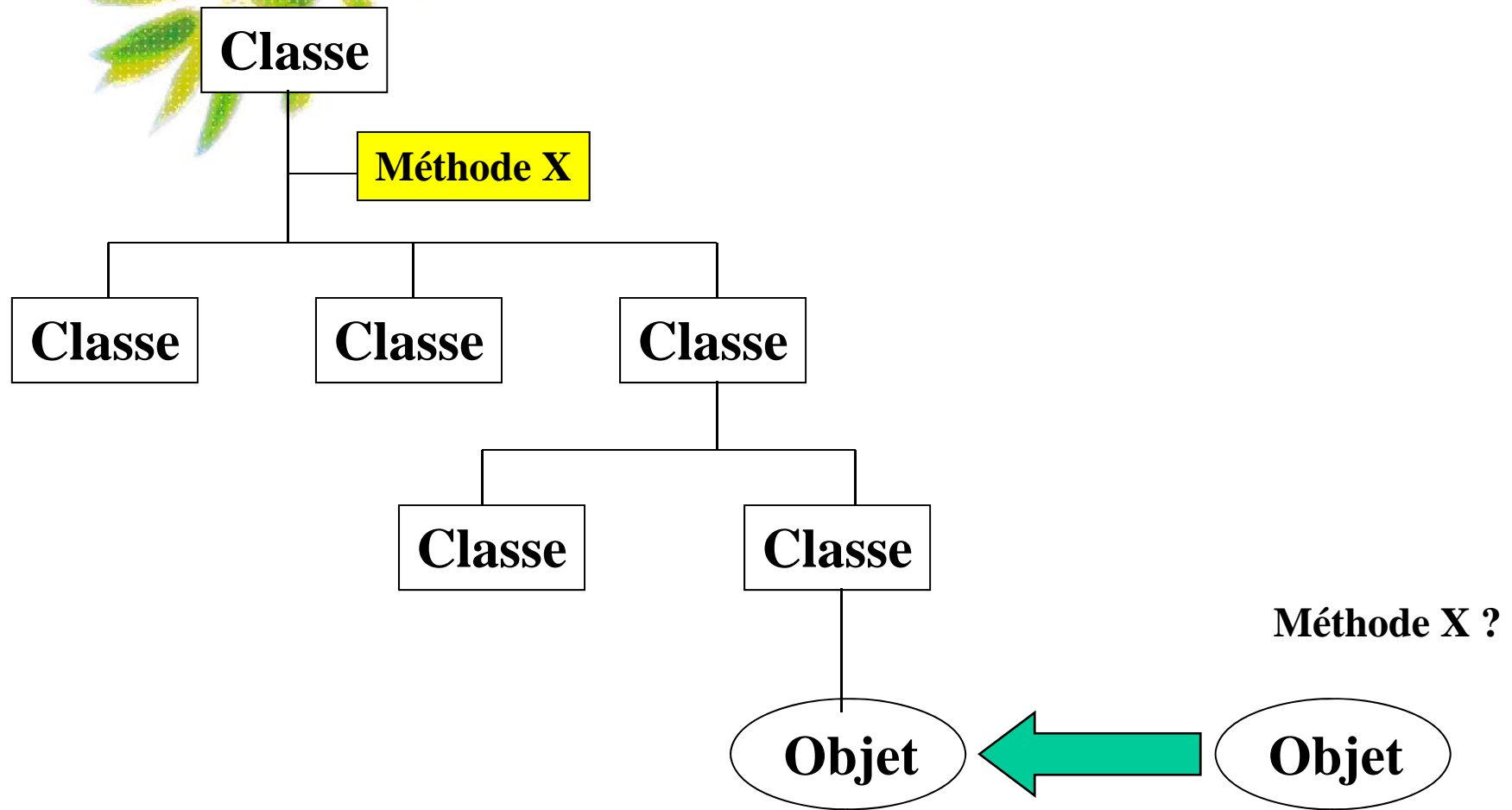
# Hiérarchie de classes

- Elle permet lors de la réalisation de programmes complexes de se servir des notions d'héritage pour optimiser l'utilisation des objets et la réutilisation de méthodes et de variables (allègement des programmes)

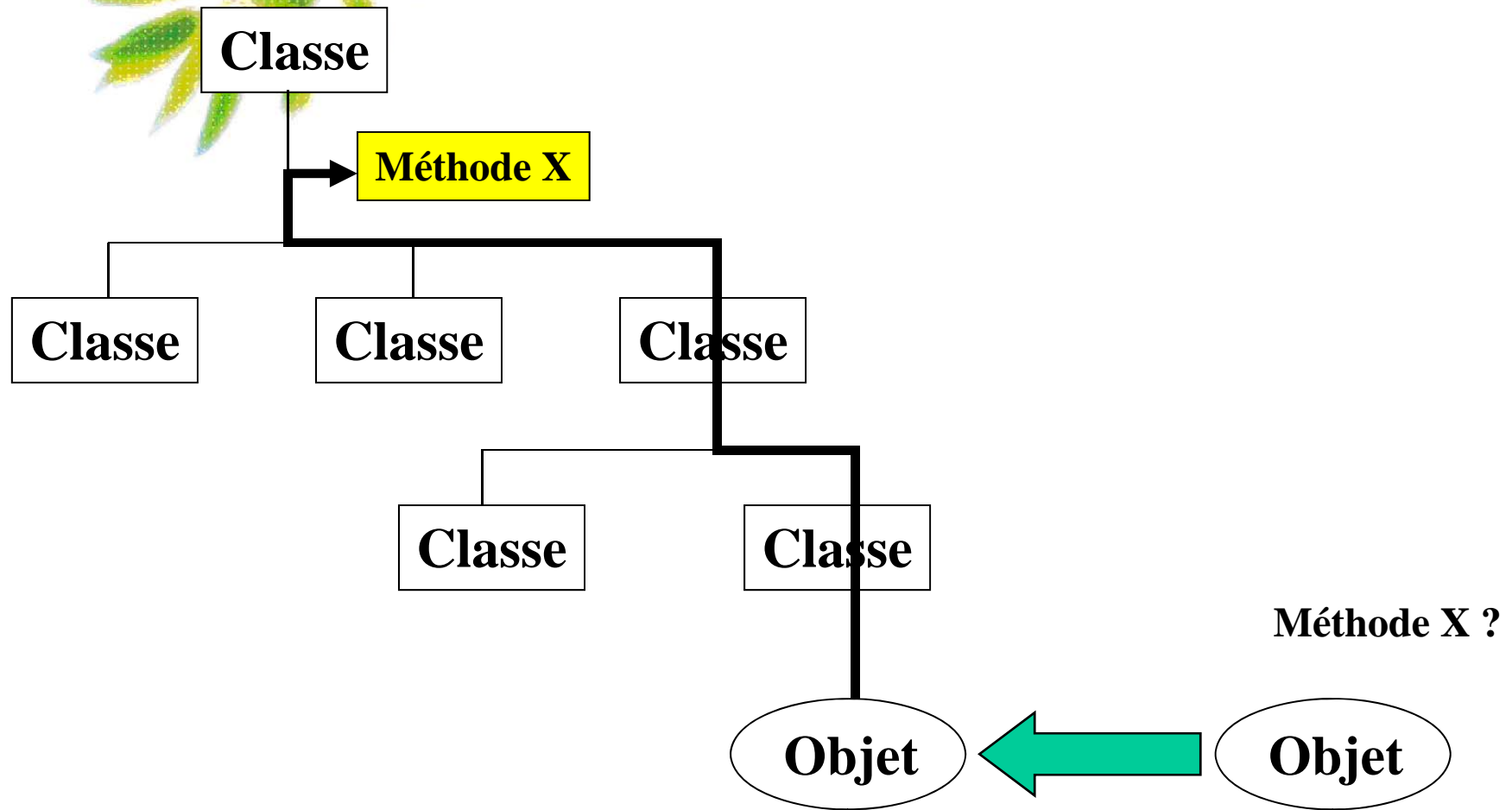


**Hiérarchie de base des véhicules.**

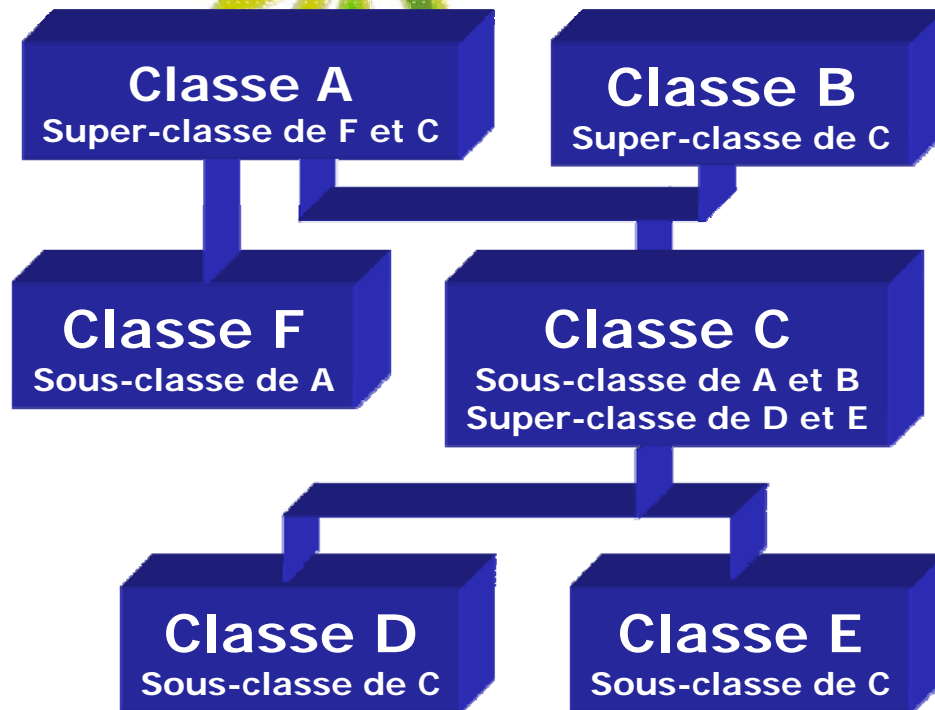
# Héritage - fonctionnement



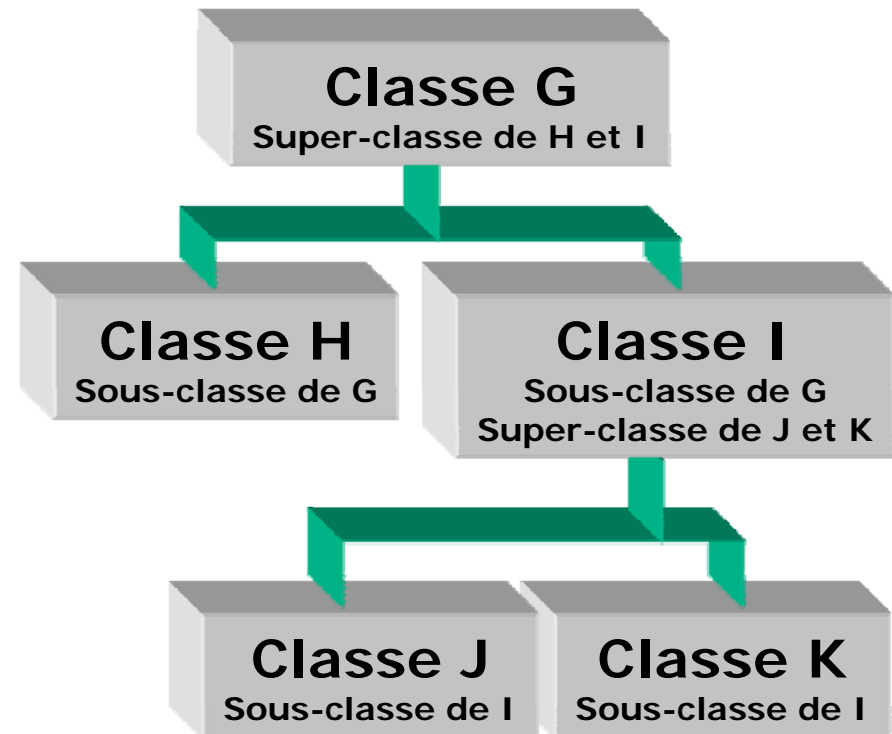
# Héritage - fonctionnement



# Héritage simple et multiple



Multiple Exemple : C++



Simple Exemple :  
JAVA – PHP5





# Interface et package

- Permettent la conception et la mise en oeuvre de **groupes de classes** et du **comportement de ces classes**
- L'héritage simple restreint la mise en œuvre de comportements complexes issus de branches de classes différentes
- Les interfaces permettent cette résolution.

**Une interface permet de rassembler un ensemble de noms de méthodes en un seul endroit, et d'ajouter ces méthodes en bloc à des classes qui en ont besoin**

- Les interfaces contiennent :
  - Les noms de méthodes
  - Les indications d'interface (par exemple les arguments)
  - Mais pas les définitions des méthodes concernées



# Interface et package

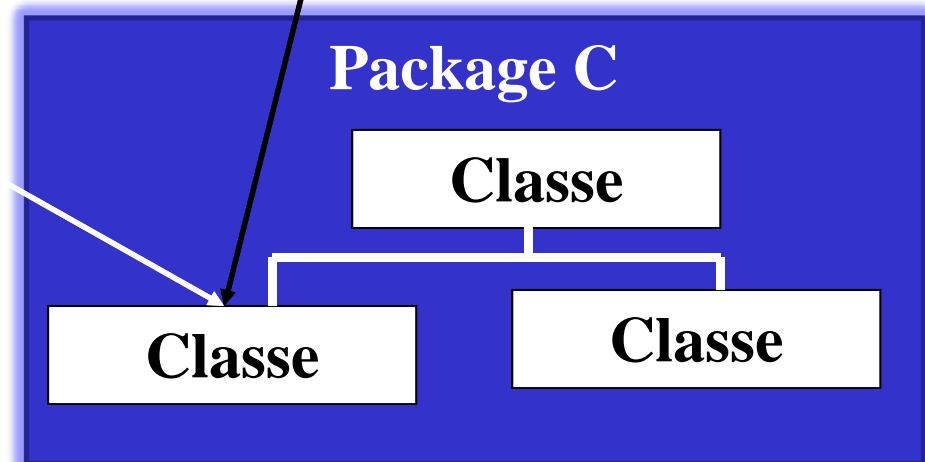
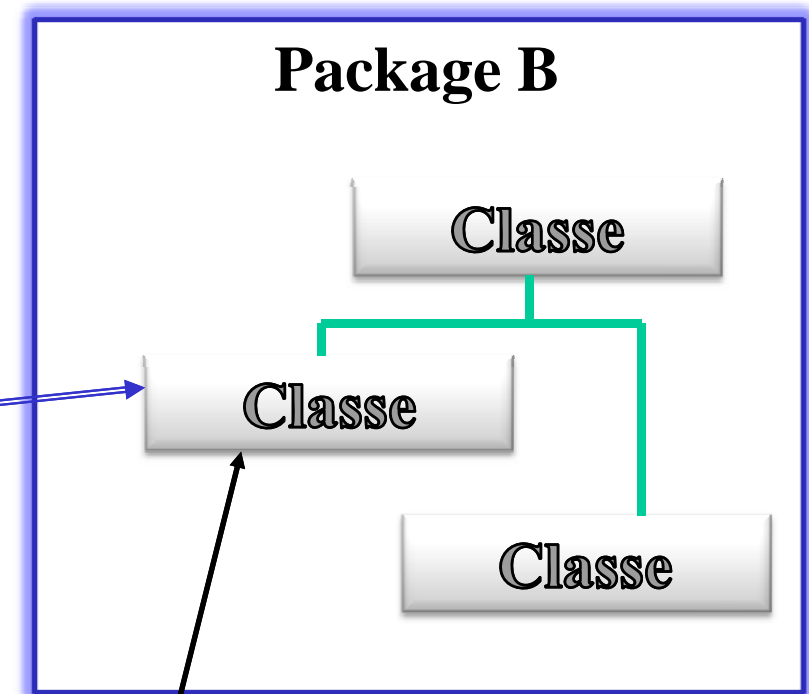
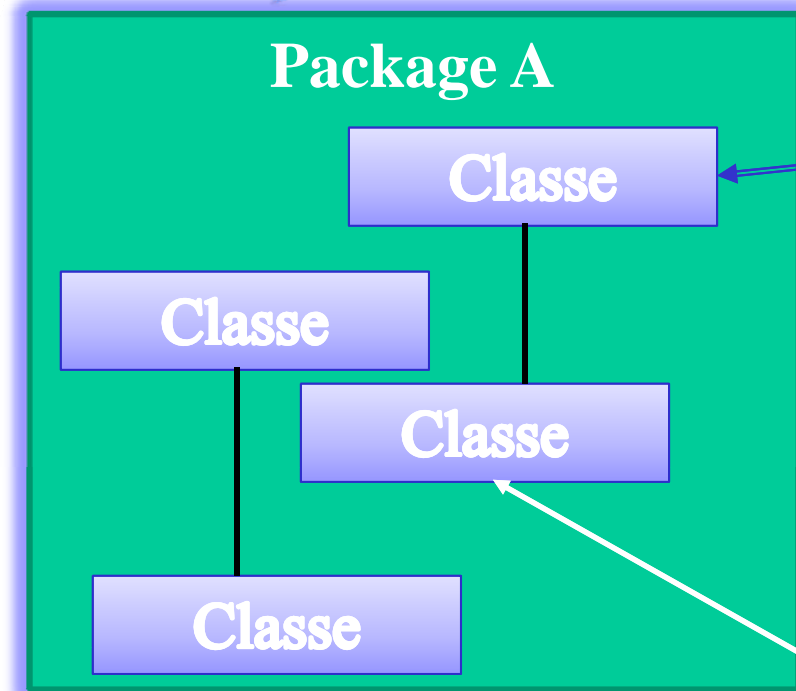
- Une classe Java :
  - possède une super-classe et une seule
  - peut implémenter un nombre quelconque d'interfaces.
- L'implémentation d'une interface consiste à citer et définir des méthodes dont les noms figurent dans l'interface.
- Ainsi, deux classes très différentes peuvent répondre aux mêmes appels de méthodes mentionnées dans l'interface implémentée dans les deux classes



# Interface et package

- **Les packages regroupent des classes et des interfaces ayant un rapport entre elles, dans une seule collection ou bibliothèque.**
- Grâce à eux :
  - Les groupes de classes sont disponibles uniquement s'ils sont nécessaires
  - Cela permet d'éviter des conflits entre noms de classes de différents groupes.
- Les bibliothèques de classes du JDK sont dans le package java
- Par défaut, les classes accèdent seulement à java.lang (le package du langage de base fourni avec le package Java).
- Pour utiliser celles d'un autre package, il faut les référencer explicitement avec le nom du package ou les importer dans le fichier source.

# Interface et package





## **3. Bases du langage**



# Instructions, expressions

- Une instruction représente une opération.

- Exemples :

```
int i = 1;  
import java.awt.Font;  
System.out.println("Cette moto est une "+  
marque + " " + couleur);
```

- Les instructions retournent parfois des valeurs (addition, test ...).
    - Les espaces placés dans les instructions Java sont ignorés.
    - Une instruction peut figurer sur une seule ligne ou sur plusieurs.
    - Toutes les instructions Java se terminent par un point-virgule.
- Java possède également des instructions composées, ou blocs, utilisées comme une instruction simple :
  - Elles commencent et finissent par des accolades ({ }).





# Types de données, variables

- Les variables sont des emplacements de mémoire dans lesquels on peut stocker des valeurs.
- Toute variable a un nom, un type et une valeur.
- Avant d'utiliser une variable, il faut la déclarer.
- On peut ensuite lui attribuer une valeur.
- **Java possède trois sortes de variables :**
  - les variables d'instance
  - les variables de classe
  - les variables locales.



# Types de données, variables

- Les **variables d'instance** déterminent les attributs ou l'état d'un objet donné.
- Les **variables de classe** ont le même type de fonction, mais leurs valeurs s'appliquent à tous les objets de la classe (et à la classe elle-même) et non à un seul objet.
- Les **variables locales** sont déclarées et utilisées dans les définitions de méthodes.
  - Elles servent :
    - de compteur d'index dans une boucle,
    - de variable temporaire
    - elles contiennent des valeurs utiles à la définition d'une méthode.



# Déclaration de variables

- Une **variable** doit être **déclarée** avant d'être utilisée dans un programme.

- Sa déclaration se compose d'un **type** et d'un **nom de variable** :

```
int mon_age;  
String mon_nom;  
boolean est_fatigue;
```

- Les définitions de variables se placent n'importe où dans la définition de la méthode, **par principe, elles sont déclarées en tête** :

```
public static void main (String args[])  
{  
    int compteur;  
    String titre;  
    boolean test;  
    ...  
}
```



# Déclaration de variables

- Toutes les variables du **même type peuvent être déclarées ensemble**

```
int x, y, z;  
String prenom, nom;
```

- L'**initialisation** d'une variable est possible dans la déclaration :

```
int mon_age, ma_taille, peinture = 43;  
String mon_nom = "Jean-Marie";  
boolean test = true;  
int a = 4, b = 5, c = 6;
```

- Si une ligne contient plusieurs variables mais une valeur d'initialisation unique (voir première ligne de l'exemple), seule la dernière variable est initialisée.
- Pour initialiser plusieurs variables sur une seule ligne, il suffit de les séparer avec une virgule, comme dans la dernière ligne de l'exemple ci-avant.



# Déclaration de variables

**Avant d'utiliser une variable locale, il faut lui attribuer une valeur.**

■ Pour cette raison, il est préférable d'initialiser les variables locales dans leurs déclarations. Cette **contrainte n'existe pas pour les variables d'instance ou de classe**, qui ont une valeur par défaut en fonction de leur type :

null	pour les instances de classe
0	pour les variables numériques
'\0'	pour les caractères
false	pour les booléens.



## Noms de variables

- Les **noms de variables** du langage Java **commencent par une lettre**, le **signe de soulignement** (`_`) ou le **signe dollar** (`$`).
- Ils **ne peuvent pas commencer par un chiffre**.
- Les **caractères suivants sont des chiffres ou des lettres**.
- Les **symboles comme % , \* , @ ou / doivent être évités** car ils sont souvent réservés aux opérateurs de Java.
- Enfin, **Java différencie les majuscules et les minuscules**. Aussi, la variable `X` est-elle différente de la variable `x`.
- Un autre exemple : `rose` n'est ni une `Rose` ni une `ROSE`.



# Type de variables

- Une déclaration de variable se compose d'un nom mais aussi d'un type, qui détermine les valeurs possibles de cette variable.
- Un type de variable :
  - un des huit types de données primaires,
  - un nom de classe,
  - un tableau
  - un vecteur
  - une collection
  - ...





# Type de données primaires

- Les **huit types de données primaires** regroupent :
  - les **types entiers usuels**
  - les **nombre à virgule flottante**
  - les **caractères**
  - les **valeurs booléennes**.
- Il sont appelés primaires car ils sont construits dans le système et ne sont pas des objets.
- Il existe **quatre types d'entiers** dans Java, correspondant chacun à un intervalle de valeurs.
- Ils sont **tous signés**, et acceptent donc les nombres négatifs ou positifs.
- Le **choix du type** dépend des **valeurs maximales et minimales** que peut prendre la variable.



## Type de données primaires

- Si une valeur stockée devient trop grande pour le type de la variable, elle est tronquée, sans autre indication.

Type	Taille	Intervalle de valeurs
byte	8 bits	-128 à 127
short	16 bits	-32 768 à 32 767
int	32 bits	-2 147 483 648 à 2 147 483 647
long	64 bits	-9 223 372 036 854 775 808 à 9 223 372 036 854 775 807

- Les nombres à **virgule flottante** sont utilisés pour les nombres comportant **une partie décimale**.

- Java contient deux types de nombres à virgule flottante : **float (32 bits, simple précision)** et **double (64 bits, double précision)**.



# Type de données primaires

- Le type **char** définit les **caractères**.
- Enfin, le type **boolean** accepte deux valeurs : **true** ou **false**.
- Notez que **tous les types de données primaires sont indiqués en minuscules**.
- Ce point est important, car **il existe aussi des classes de mêmes noms**, mais commençant par une majuscule, et dont le comportement est différent.
- Par exemple :
  - Le type primaire **boolean** est différent de la classe **Boolean**.

- Les **variables** de Java peuvent aussi être déclarées pour **recevoir un objet d'une classe** particulière :

```
String nom;  
Font police;  
OvalShape oval;
```

- Ces variables reçoivent uniquement les objets d'une classe donnée ou de l'une de ses sous-classes.
- Cette seconde variante est utile lorsqu'une variable doit pouvoir contenir des instances de diverses classes ayant une même super-classe.

- Par exemple, dans le cas d'un ensemble de classes de fruits (Pomme, Poire, Fraise, etc.), toutes héritières de la classe générale Fruit
- Si vous déclarez une variable de type Fruit, cette variable peut contenir des instances de n'importe laquelle des sous-classes de Fruit.
- **Si l'on déclare une variable de type object, cette variable peut contenir n'importe quel objet.**
- **Le sur-Classement perd en précision**
- **Le sous-classement spécialise les objets ou les classes**

- Java possède trois sortes de commentaires, deux pour les commentaires ordinaires du code source, et une troisième pour le système particulier de documentation javadoc.

**Les symboles `/*` et `*/` entourent des commentaires multilignes.** Le texte compris entre ces deux bornes est ignoré :

```
/*          1° ligne de commentaire
            2° ligne de commentaire
            3° ligne de commentaire
            4° ligne de commentaire */
```

- Les commentaires ne s'imbriquent pas ; autrement dit, vous ne pouvez pas placer un commentaire à l'intérieur d'un commentaire.

- Le double slash (//) précède les commentaires dans une ligne. Tout le texte situé entre le double slash et la fin de ligne est ignoré.

```
int vices =7;           // Ceci est un commentaire
```

- Enfin, les commentaires spéciaux destinés au système javadoc commencent avec `/**` et se terminent par `*/`.



# Constante nombre

- entiers :
  - tous signés : -, exemple : **-45**
  - exprimés en décimal, octal ou hexadécimal :
    - Octal commence par 0
      - ex : **0777 ou 0444**
    - Hexadécimal commence par 0x (ou 0X)
      - ex : **0xFF ou 0XAFA5**
- constantes à virgule flottante, deux parties : la part entière et décimale
  - ex : **5.677777**
- la lettre f (ou F) à un nombre force le type float
  - ex : **2.56F**
- notation scientifique : lettres e ou E suivies de l'exposant
  - ex : **10e45** ou **.36E-2**
- booléen : **true ou false**

# Constante caractère

- caractère : 'a' caractère ayant la valeur a.
- se composent d'une lettre entourée de guillemets simples : 'a', '#', '3' ...
- stockés sous forme de codes Unicode 16 bits
- caractères spéciaux :

Séquence	Signification
\n	Retour à la ligne
\t	Tabulation
\b	Retour arrière
\r	Retour chariot
\f	Saut de page
\\	Slash inversé
\'	Guillemet simple
\"	Guillemet double
\ddd	Octal
\xdd	Hexadécimal
\udddd	Caractère Unicode



## Constante chaîne de caractères

- Une chaîne est une combinaison de caractères.
- **En langage Java, les chaînes sont des objets de la classe String, et non de simples tableaux de caractères.**
- De plus, elles empruntent certaines de leurs caractéristiques aux tableaux (la possibilité de tester leur longueur, d'ajouter ou d'effacer certains caractères).
- Comme ce sont des objets à part entière pour Java, elles possèdent des méthodes permettant de les combiner, de les tester ou de les modifier très facilement.



## Constante chaîne de caractères

- Les chaînes sont représentées par une série de caractères entourée de guillemets doubles :

```
"Je suis une chaîne de caractères"
```

```
" " //une chaîne vide
```

- Les chaînes peuvent contenir des caractères de contrôle (retour à la ligne, tabulation ou caractères Unicode) :

```
"Une chaîne avec \t tabulation"
```

```
"Une chaîne \" entre guillemet" double"
```

```
"Une chaîne écrite par Java\u2122"
```



# Opérateurs arithmétiques

- Java possède cinq opérateurs arithmétiques de base

Opérateur	Signification	Exemple
+	<b>Addition</b>	<b>3 + 4</b>
-	<b>Soustraction</b>	<b>5 - 7</b>
*	<b>Multiplication</b>	<b>5 * 5</b>
/	<b>Division</b>	<b>14 / 7</b>
%	<b>Modulo</b>	<b>20 % 7</b>

- Tous ces opérateurs utilisent deux opérandes et sont placés entre ces derniers.
- L'opérateur de soustraction (-) peut aussi servir à inverser le signe d'un opérande.

- Les applications sont des **programmes Java autonomes**.
- Les applications sont distinctes des applets, dont la visualisation exige l'utilisation d'un navigateur compatible Java.
- Une application Java se compose d'une ou de plusieurs classes de dimensions quelconques.
- La seule chose indispensable pour faire tourner une application Java, c'est une classe servant de base de lancement pour le reste du programme (**méthode main()**).



## Méthode main()

- La signature de la méthode main ( ) est toujours conforme à la syntaxe :

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

- **public** indique que cette méthode est accessible à d'autres classes et à d'autres objets. Une méthode main ( ) doit être déclarée public.
- **static** signifie qu'il s'agit d'une méthode de classe.
- **void** indique que la méthode main ( ) ne retourne rien.
- **main( )** prend un argument, qui est un tableau de chaînes. Il s'agit des arguments (paramètres) en ligne de commande





# Opérateurs arithmétiques

## 3.1 ArithmeticTest

- Deux particularités de Java :
  - La première est la classe Math, qui fournit de nombreuses opérations mathématiques courantes comme la racine carrée, la valeur absolue, le cosinus...
  - La seconde particularité est propre à Java depuis 1.3/1.4 : les classes BigDecimal et BigInteger.
  - Ces classes fournissent des mécanismes pour stocker et traiter des nombres extrêmement grands, appelés "bignums", qui peuvent l'être trop pour les types primaires

- L'affectation des variables est une forme d'expression qui retourne une valeur.
  - Aussi est-il possible de chaîner des variables :  
$$\mathbf{x = v = z = 0}$$
  - Dans cet exemple, les trois variables prennent la valeur 0.
- La partie droite de l'expression d'affectation est toujours évaluée avant l'opération d'affectation.
- Une expression comme  $\mathbf{x = x + 2}$  est donc traitée ainsi :
  - 2 est ajouté à la valeur de x,
  - puis le résultat est affecté à x.



## Opérations abrégées

■ Ce type d'opération est tellement courant que Java en possède une version abrégée empruntée à C et C++.

### Expression

$x += y$

$x -= y$

$x *= y$

$x /= y$

### Signification

$x = x + y$

$x = x - y$

$x = x * y$

$x = x / y$



# Incrémentation, décrémentation

- Les opérateurs `++` et `--` permettent d'incrémenter ou de décrémentation de 1 la valeur d'une variable.
  - Par exemple, `x++` ajoute 1 à la valeur de `x` comme le ferait l'expression `x = x + 1`.
  - De la même façon, `x--` diminue de 1 la valeur de `x`.
- Ces deux opérateurs se placent devant ou derrière la valeur à incrémenter ou à décrémentation.



# Incrémentation, décrémentation

- Soit les deux expressions suivantes :

**$y = x++;$**        **$\Rightarrow$**        **$y = x;$**   
    **$x = x+1;$**

**$y = ++x;$**        **$\Rightarrow$**        **$x = x+1;$**   
    **$y = x;$**

- Elles donnent des résultats différents :
  - Quand les opérateurs sont placés après la valeur ( $x++$  ou  $x--$ ),  $y$  prend la valeur de  $x$  avant incrémentation.
  - Avec la notation en préfixe, la valeur de  $x$  est attribuée à  $y$  après incrémentation.



# Opérateurs de comparaison

- Java possède plusieurs expressions permettant d'effectuer des comparaisons.
- Chacune d'elles retourne une valeur booléenne (true ou false).

Opérateur	Signification	Exemple
<code>==</code>	Egal	<code>x == 3</code>
<code>!=</code>	Différent	<code>x != 3</code>
<code>&lt;</code>	Inférieur à	<code>x &lt; 3</code>
<code>&gt;</code>	Supérieur à	<code>x &gt; 3</code>
<code>&lt;=</code>	Inférieur ou égal à	<code>x &lt;= 3</code>
<code>&gt;=</code>	Supérieur ou égal à	<code>x &gt;= 3</code>



# Opérateurs logiques

- **ET** logique représenté par les opérateurs **&** ou **&&**
  - L'expression est vraie si les deux opérandes sont également vrais.
- **OU** est symbolisé par **|** ou **||**
  - Les expressions OU sont vraies si l'un des deux opérandes est vrai.
- **OU EXCLUSIF** est symbolisé par **^** ou **^^**
  - Il retourne la valeur true (vrai) si les deux opérandes sont différents (l'un vrai et l'autre faux ou vice versa). Dans le cas contraire, il retourne la valeur false (faux).
- Les opérateurs **&**, **|** et **^** sont plutôt utilisés pour des opérations logiques au niveau bit.
- Le **NON** logique est représenté par l'opérateur **!**
  - Si **x** est vrai, **!x** est faux.





# Hiérarchie des opérateurs

- La hiérarchie des opérateurs indique l'ordre d'évaluation des expressions.
- Soit l'expression suivante :  $y = 6 + 4 / 2$ 
  - La valeur de  $y$  est 5 ou 8, selon la partie de l'expression qui est évaluée en premier (  $6 + 4$  ou  $4 / 2$  ).
  - La hiérarchie des opérateurs impose un ordre de traitement des opérations.



# Arithmétique sur les chaînes

- Une particularité de Java est l'utilisation de l'opérateur d'addition (+) pour créer **et concaténer des chaînes**.

- Exemple :

```
System.out.println(nom + " est un " + couleur );
```

- Un objet ou un type peut être converti en chaîne grâce à la méthode **toString ( )**.
  - Tout objet possède une représentation chaîne par défaut (le nom de la classe suivi d'accolades, toString ( )
  - L'opérateur +=, étudié précédemment, est aussi utilisé dans une chaîne.
  - L'instruction : **mon\_nom += " JM "**; est équivalente à : **mon\_nom = mon\_nom + " JM "**; exactement comme pour les nombres.



## **4.Travailler avec des objets**



# Création de nouveaux objets

- L'écriture d'un programme Java = définition de classes (modèles pour objets).

**Pour créer leurs objets, il faut impérativement utiliser l'opérateur new.**

- A l'inverse, les types de données primaires définissent des nombres et des caractères ou des booléens.



# Opérateur new

- Les objets sont créés au moyen de l'opérateur new :

```
Nom_classe nom_objet = new Nom_classe();
```

- Exemple :

```
String str = new String();  
Random r = new Random();  
moto m2 = new moto();
```

- Si les parenthèses sont vides, comme dans l'exemple ci-avant, un objet basique est créé.

- Si elles contiennent des arguments, ceux-ci déterminent les valeurs initiales des variables d'instance.

- Exemple :

```
Date dt = new Date(90, 4, 1, 4, 30);  
Point pt = new Point(0,0);
```



## Opérateur new

- Le nombre et le type des arguments à l'intérieur des parenthèses sont définis par la classe elle-même, au moyen d'une méthode spéciale appelée constructeur.

**Si vous essayez de créer une nouvelle instance d'une classe avec un mauvais nombre ou un mauvais type d'arguments ==> Erreur**

```
Nom_classe nom_objet = new Nom_classe(arguments)
```



## Opérateur new, exemples

- Trois façons différentes de créer un objet Date avec l'opérateur new.

```
Date d1, d2, d3;
```

```
// Création de d1 à la date du jour
```

```
d1 = new Date();
```

```
// Création de d2 au 01 Août 1971 à 07:30
```

```
d2 = new Date(71, 7, 1, 7, 30);
```

```
// Création de d3 au 03 Avril 1993 à 03:24 PM
```

```
d3 = new Date("April 3 1993 3:24 PM");
```

**Java numérote les mois en commençant par 0. Donc le 7<sup>o</sup> mois est le mois d'août.**





## Opérations réalisées par new

1°) Création d'un nouvel objet d'une classe donnée

2°) Allocation de la mémoire :

- Appel **constructeur** (méthode spéciale, définie dans les classes, servant à créer et à initialiser de nouvelles instances), défini dans la classe concernée

3°) Initialisation de l' objet et de ses variables

4°) Création des autres objets dont cet objet peut lui-même avoir besoin

■ Il peut y avoir différentes définitions de constructeurs dans une classe , il peuvent avoir différents nombres et types d'arguments.

■ Ainsi, le constructeur appelé correspond aux arguments passés à new.



# Variables d'instances et de classes, accès

- **Accès aux valeurs :**

- Utilisation de la **notation point**

- La référence à une variable d'instance est composée de deux parties :

- l'objet, à gauche du point, et la variable, à droite.

- Exemple :

Objet :	<code>mon_objet.</code>
variable d'instance :	<code>var</code>
notation :	<b><code>mon_objet.var;</code></b>

- Si par exemple `var` est un objet contenant sa propre variable d'instance appelée `etat`, on notera :

**`mon_objet.var.etat;`**

# Variables de classe

- Les variables de classe sont définies et stockées dans la classe elle-même.
- Leurs valeurs s'appliquent à la classe et à tous ses objets.
- Les variables de classe sont définies avec le mot clé static placé devant la variable.

■ Exemple :

```
class MembreFamille
{
    static String nom = "Johnson";
    String prenom;
    int age;
    ...
}
```



## Variables de classe

```
class MembreFamille {  
    static String nom = "Johnson";    // var. classe  
    String prenom;                    // var. instance  
    int age;...  
}
```

- Les instances de la classe MembreFamille ont chacune leurs propres valeurs pour prenom et age.
- A l'inverse, la variable de classe nom a une seule valeur pour tous les membres de la famille.
- Si la valeur de nom est modifiée, toutes les instances de MembreFamille le sont aussi.



# Variables de classe

## 4.1 MembreFamille

- L'accès aux variables de classe et d'instance utilise la même notation.
- Pour accéder ou modifier la valeur d'une variable de classe, l'objet ou le nom de la classe doit précéder le point.
- Les deux lignes affichées par le code suivant sont identiques :

```
MembreFamille pere = new MembreFamille();
```

```
System.out.println("nom de famille = " + pere.nom);
```

```
System.out.println("nom de famille = " + MembreFamille.nom);
```

- Avec la possibilité de modifier la valeur d'une variable de classe à l'aide d'une instance, il est facile de se perdre entre les variables de classe et l'origine de leurs valeurs.
- C'est pourquoi il vaut mieux référencer une variable de classe à l'aide du nom de la classe.



## Appel des méthodes

- L'appel d'une méthode est similaire à la référence aux variables d'instance d'un objet (utilisation de la notation point).
- L'objet est placé à gauche du point, la méthode et ses arguments à droite, exemple :

**mon\_objet.methode\_1 (arg1, arg2, arg3);**

- Toutes les méthodes doivent être suivies de parenthèses, même si elles n'ont pas d'arguments :

**mon\_objet.methode\_sans\_argument ();**



# Appel des méthodes

- Si la méthode appelée est un objet qui contient aussi des méthodes, ces dernières s'imbriquent, comme pour les variables.
- Exemple :

```
mon_objet.getClass().getName();
```

- Les appels imbriqués de méthodes peuvent se combiner avec des références à des variables d'instance également imbriquées.
- Exemple :

```
•mon_objet.var.method_2(arg1, arg2);
```





# Méthodes de classe

- Comme les variables de classe, les **méthodes de classe** s'appliquent à la **classe tout entière**.
- Elles sont souvent utilisées pour des **méthodes à vocation générale**.
- Les méthodes de classe servent aussi à grouper au même endroit, dans une classe, des méthodes à caractère général.
- Exemple :
  - La **classe Math** définie dans le package `java.lang` contient de nombreuses méthodes de classe.
  - Ce sont des opérations mathématiques.
  - Il n'existe **pas d'instance de la classe Math**, mais on peut utiliser ses méthodes avec des arguments numériques ou booléens.



## Méthodes de classe, exemple

- Exemple :

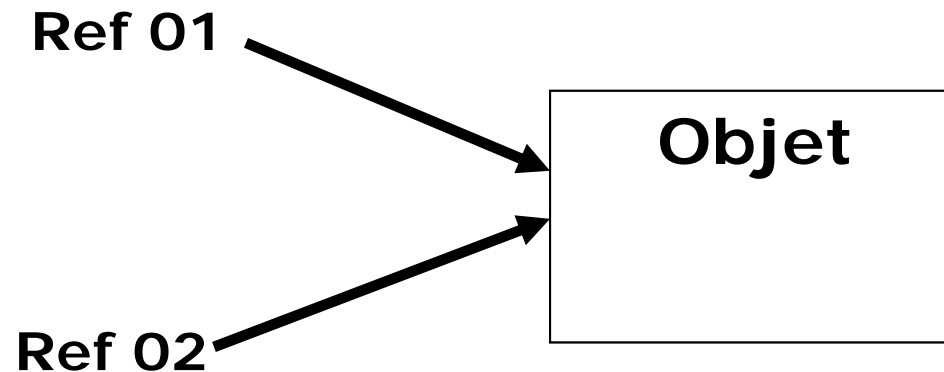
- La méthode de classe **Math.max ( )** prend deux arguments et retourne le plus grand des deux.

- Il n'est pas nécessaire de créer une nouvelle instance de Math ; il suffit d'appeler la méthode là où on en a besoin, comme suit :

```
int le_plus_grand = Math.max(x,y);
```

# Références aux objets

- Même si elles sont occultées, les références aux objets manipulés sont importantes.
- Lorsque des **objets sont attribués à des variables** ou passés en arguments à une méthode, ce sont leurs **références** qui sont transmises.
- En effet, ni les objets eux-mêmes, ni leur copie ne sont passés.





# Références aux objets, exemple

## 4.2 ReferenceObjet

```
Point pt1, pt2;  
pt1 = new Point(100, 180);  
pt2 = pt1;  
pt1.x = 200;  
pt1.y = 200;
```

### Résultat :

Point 1: 200, 200

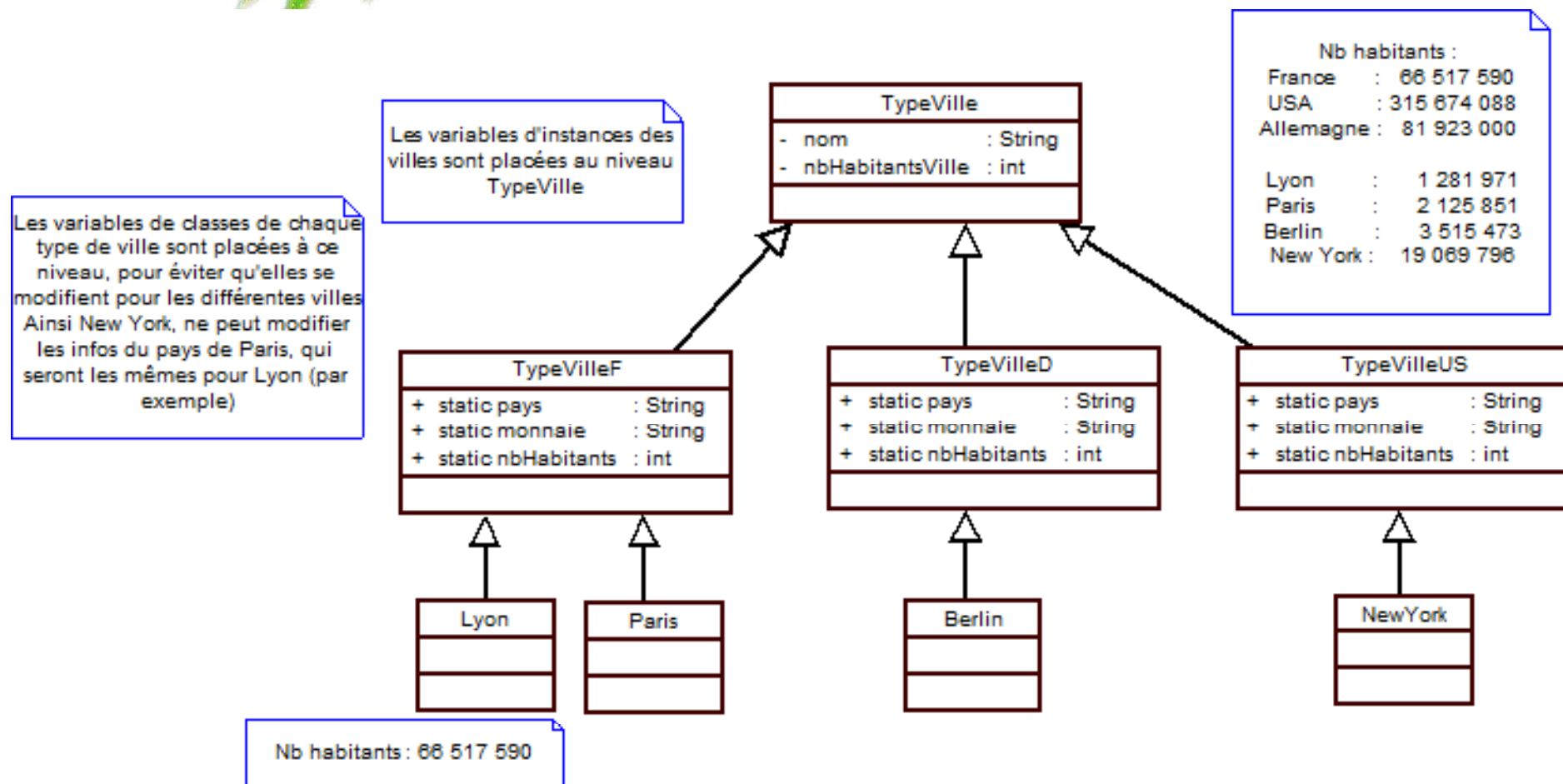
Point 2: 200, 200

- La question est : après la mise à jour des variables d'instance x et y de pt1 que devient pt2 ?
- Les variables d'instance x et y de pt2 ont également été modifiées, bien qu'on ne l'ait pas fait explicitement.
- L'affectation de la valeur de pt1 à pt2 a véritablement créé à partir de pt2 une référence à l'objet pointé par pt1.

# Exercice

## 4.3 Exercice Ville

- En vous inspirant des membres famille, créer des villes appartenant à un pays





# Conversion de type de données primaires et d'objets

- Parfois, certaines valeurs sont stockées dans un type qui ne convient pas pour ce que vous voulez en faire
- Instance d'une mauvaise classe, nombre à virgule flottante (float ) au lieu d'entier (int), entier au lieu d'une chaîne.

**Le mécanisme consistant à transformer la valeur d'un type vers un autre est une conversion.**

- La conversion est un mécanisme qui, à partir d'un objet ou d'une valeur d'un certain type, crée un autre objet ou une autre valeur de type différent.
- Le résultat de la conversion est un nouvel objet ou une nouvelle valeur
- La conversion n'affecte pas l'objet ou la valeur d'origine.



# Conversion de types primaires

- La conversion des types primaires consiste à convertir la valeur d'un type primaire dans une autre.

**Les valeurs booléennes ne peuvent pas être converties en d'autres types primaires.**

- Dans la plupart des cas, une **conversion vers un type le plus grand** apporte une **précision supérieure** et aucune information n'est perdue lors de la conversion.
- *Toutefois, la conversion d'entiers à valeurs virgule flottante fait exception à cette règle générale*
- La conversion vers un type plus petit peut entraîner une perte de précision
  - un int ou un long en un float,
  - un long en un double





# Conversion de types primaires

## 4.4 Conversion primaire

- La conversion explicite utilise le format suivant :

**(typename) value**

- L'expression suivante divise x par y et convertit le résultat en type int :

**(int) (x / y);**

- La **conversion est prioritaire sur les opérations arithmétiques**, d'où l'emploi des parenthèses dans notre exemple, pour que la conversion porte bien sur le résultat de la division

- Autrement, le **valeur de x serait d'abord convertie, puis divisée par y**, ce qui pourrait donner un résultat différent.



# Conversions des objets

- Les instances d'une classe peuvent aussi être converties en instances d'autres classes, à une condition :

**les classes des objets à convertir et convertis doivent être liées par le mécanisme d'héritage.**

- Autrement dit, la conversion d'un objet n'est possible que vers une instance **d'une sous-classe ou d'une super-classe de sa propre classe.**

- En effet, les instances de sous-classes contiennent généralement toutes les informations des instances des super-classes.

- Notation :

**(classname) object**

- Dans cette instruction, classname est le nom de la classe dans laquelle object (une référence à un objet) doit être converti.

- La **conversion crée une référence**, du type classname, à l'ancien objet.

- **L'ancien objet continue d'exister** sans subir de changement.



# Les wrappers

- Chacun des types primitifs peut être "enveloppé" dans un objet provenant d'une classe prévue à cet effet et appelée **Wrapper** (mot anglais signifiant *enveloppeur*)
- Les enveloppeurs sont donc des objets représentant un type primitif
- Avantages :
  - Les Wrapper peuvent être utilisés comme n'importe quel objet, ils ont donc leurs propres méthodes
- Inconvénients :
  - L'objet enveloppant utilise plus d'espace mémoire que le type primitif.
    - Par exemple, un int prends 4 octets en mémoire mais un Integer utilisera 32 octets sur une machine virtuelle en 64 bits (20 octets en 32 bits).
  - L'objet enveloppant est immuable :
    - Il ne peut pas être modifié, toute modification de sa valeur nécessite de créer un nouvel objet et de détruire l'ancien, ce qui augmente le temps de calcul



## Les wrappers

Enveloppeur	Type primitif
Character	char
Byte	byte
Short	short
Integer	int
Long	long
Float	float
Double	double
Boolean	boolean

# Transformation de types primaires en objet et inversement

- Grâce aux méthodes de classe définies dans ces classes, on peut créer un équivalent objet pour tous les types de base, à l'aide de l'opérateur new.
- La ligne de code suivante crée une instance de la **classe Integer** avec la valeur 35 :

```
Integer objet_entier = new Integer(35);
```

- Par exemple, **intValue()** extrait une valeur de type int d'un objet Integer :

```
int entier = objet_entier.intValue() ; // retourne 35
```

- Java fournit les classes spéciales suivantes correspondant aux types primaires : Boolean, Character, Double, Float, Integer et Long.



## Comparaison d'objets

- Seuls les opérateurs `==` (égal) et `!=` (différent) peuvent être utilisés pour des comparaisons d'objets
- Pour obtenir un résultat significatif lors de la comparaison d'objets d'une classe, il faut créer des méthodes spéciales dans la classe et les appeler.
- Cependant, **la classe String** possède une méthode appelée **`equals ( )`**
- Cette méthode teste chaque caractère de deux chaînes et retourne la valeur `true` s'il y a égalité parfaite



# Identification de la classe d'un objet

- Identifier la **classe d'un objet** attribué à la variable **obj** :

**String name = obj.getClass().getName() ;**

- **getClass()** est une méthode de la classe Object. Elle est donc disponible **pour tous les objets**.
- De l'appel de cette méthode **résulte un objet de la classe Class**.
- Il contient une méthode appelée **getName()** qui retourne le nom de la classe **sous forme de chaîne**.





# Identification de la classe d'un objet

- L'opérateur **instanceof** permet aussi de réaliser des tests.
- Il a deux opérandes : **un objet, à gauche, et le nom d'une classe, à droite.**
- L'expression retourne **true si l'objet appartient à la classe désignée ou à l'une de ses sous-classes, et false dans le cas contraire :**

```
"foo" instanceof String           // true
Point pt = new Point(10,10);
pt instanceof String              // false
```

- Java, Variété des modificateurs :
  - Stratégie de composition des classes
  - **Contrôle d'accès à une classe**, méthode ou variable
    - **public, protected et private**
  - **static** : Création de méthode et variables de classe
  - **final** : finaliser l'implémentation de classes, méthodes et variables
  - **abstract** : classes et méthodes abstraites, ne permettant pas d'instanciation
  - **synchronized et volatile** (threads)



# Modificateurs public, private, protected

- Permettent l'encapsulation (visibilité des méthodes et variables)
- Par défaut → Accessibles par autres classes du même package
- **Accès public (public)** → Tout est disponible
- **Accès privé (private)**
  - Masquer une variable ou méthode (non utilisable par une autre classe)
  - Empêcher la modification d'une variable ou l'exécution d'une méthode
  - Seulement visible dans la classe où elles sont déclarées
- **Accès protégé (protected)**
  - Limiter l'utilisation des variables ou méthodes :
    - Aux sous-classes d'une classe ;
    - Aux autres classes du même package.



# Principaux package Java

## ■ **java.lang :**

- ses classes concernent le langage lui-même.
- object, String et System en font partie.
- Ce package contient aussi des classes spéciales destinées à la représentation des types de données primaires (Integer, character, Float, etc.).

## ■ **java.util :**

- ces classes sont des utilitaires, comme Date, ou de simples classes de collections, comme Vector et Hashtable ...

## ■ **java.io :**

- ces classes écrivent ou lisent dans les flux d'entrées/sorties (comme l'entrée ou la sortie standard), ou encore, gèrent les fichiers



# Principaux package Java

## ■ **java.net :**

- ce sont des classes de gestion de réseau, comme socket et URL (une classe permettant de référencer des documents sur le World Wide Web).

## ■ **java.awt / javax.swing :**

- c'est l'Abstract Windowing Toolkit, ou jeu d'outils de fenêtrage abstrait.
- ces classes permettent d'implémenter une interface utilisateur graphique. Window, Menu, Button, Font, checkBox, etc., en font partie.
- Ce package contient aussi des mécanismes de gestion des événements système, et des classes de traitement d'images (le package java.awt.Image).

## ■ **java.applet / javax.swing.JApplet :**

- Ces classes servent à implémenter des applets Java.



## **5. Branchements conditionnels, boucles, structures de données**

- Les tableaux fournissent un moyen de **stocker une collection d'éléments** sous forme d'entité unique.
- Le tableau a **un certain nombre de cellules**, chacune d'elle comportant un élément individuel.
- On peut ajouter des éléments aux cellules ou supprimer selon les besoins.
- Contrairement à ceux d'autres langages, les tableaux, en Java, sont de **véritables objets**, pouvant être traités comme n'importe quels autres objets du langage.
- Les tableaux contiennent des **types de données quelconques (de type primaire ou objet)**,
- mais il est **impossible de stocker plusieurs types différents dans un même tableau**.





# Déclaration de tableaux

- Pour créer un tableau, il faut définir une variable dans laquelle il sera stocké.
- Exemples de déclaration de variables tableau :

```
String difficultWords[ ] ;  
Point hits[];  
int temps[];
```
- Autres exemples de déclaration :

```
String[] difficultWords;  
Point[] hits;  
int[] temps;
```



# Création des objets tableau

- La seconde étape consiste à créer un objet tableau et à l'attribuer à cette variable.

- Il y a deux façons de le faire :

- en utilisant new,

- ```
String[] names = new String[10];
```

- en initialisant directement le contenu du tableau.

- ```
String[] pays = { "France", "Allemagne", "Espagne",  
                  "Italie" "Suisse" };
```



# Parcours des objets tableau

## 5.1 TestTableau

- Accès aux valeurs par son indice :

```
mon_tableau[indice];
```

```
Ex : mon_tableau[1] = 15;
```

- Pour ne pas sortir des limites des tableaux dans les programmes, leur longueur doit être testée avec **la variable objet length (variable d'instance)**, disponible pour tous les types d'objets tableau :

```
int longueur = tableau_1.length // retourne 10
```

- Remarque :
  - Pour **les tableaux** : **length** est une variable d'instance
  - Pour **les String** : **length()** est une méthode d'instance



# Tableaux multidimensionnels

- Java ne supporte pas directement les **tableaux multidimensionnels**.
- Cependant, il est possible de déclarer et créer un tableau de tableaux (lesquels peuvent aussi contenir des tableaux, et ainsi de suite, sans aucune limitation du nombre de dimensions ainsi obtenues), et d'accéder à ses éléments.

```
int coords[][] = new int[12][12];  
    coords[0][0] = 1;  
    coords[0][1] = 2;
```



# Blocs d'instructions

- Un bloc d'instructions est un groupe d'instructions entouré d'accolades ({}). Il se place comme une simple instruction.

```
{  
  .Bloc d 'instructions  
  .  
}
```

- La règle est que l'on peut placer un bloc partout où l'on pourrait placer une instruction, et que chaque instruction du bloc s'exécute en séquence.



# Branchement conditionnel if

- Ce branchement conditionnel contient le mot clé if suivi d'un test booléen et d'une instruction ou d'un bloc d'instructions à exécuter si le résultat du test est vrai (true).

- Exemple :

```
if (x < y)
    System.out.println("x est plus petit que y");
```

- Le mot clé optionnel else permet à une autre instruction de s'exécuter si le résultat du test est faux (false) :

```
if (x < y)
    System.out.println("x est plus petit que y");
else
    System.out.println("y est plus grand.");
```

# Opérateur conditionnel

- L'opérateur conditionnel, aussi appelé **opérateur ternaire** (3 opérandes), est une alternative aux mots clés if et else.
- Il est fait pour les conditions simples ou très courtes et suit la syntaxe suivante :

**test ? resultat\_vrai : resultat\_faux;**

- test est une expression qui retourne la valeur true ou false
- Si le résultat est vrai (true), l'opérateur retourne la valeur de resultat\_vrai, sinon la valeur de resultat\_faux.
- Exemple :
- Les valeurs de x et y sont testées et la plus petite est retournée et attribuée à la variable smaller :

```
int smaller = x < y ? x : y;
```

- Equivalence if :

```
int smaller  
if x < y smaller = x  
else  smaller = y;
```





# Branchement conditionnel SWITCH

## 5.2 NumberReader

- Permet d'alléger des tests compliquant le code.
- Cette forme d'ordre if présente des if imbriqués.

```
switch (test)
{
case valueOne:
    resultOne;
    break;
case valueTwo:
    resultTwo;
    break;
case valueThree:
    resultThree;
    break;

default:
    defaultresult;
```



## Branchement conditionnel SWITCH, exemples

```
switch (oper)
{
case '+':
    addargs(arg1, arg2);
    break;
case '-':
    subargs(arg1, arg2);
    break;
case '*':
    multargs(arg1, arg2);
    break;
case '/':
    divargs(arg1, arg2);
    break;
}
```

```
switch (x)
{
case 2:
case 4:
case 6:
case 8:
    System.out.println(
        "x est un nombre
        pair");
    break;
default:
    System.out.println(
        "x est un nombre
        impair");
}
```

- Les boucles for exécutent une instruction ou un bloc d'instructions jusqu'à ce qu'une certaine condition soit remplie.
- Les boucles for sont souvent utilisées pour des itérations simples (compteurs) d'instructions, mais on peut les utiliser pour pratiquement toutes sortes de boucles.
- Les boucles for ont la syntaxe suivante :

```
for (initialisation ; test ; incrément)
{
    instruction;
}
```



## Boucle while

- La boucle while exécute une instruction ou un bloc tant qu'une condition particulière reste vraie. Sa syntaxe est la suivante :

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

- **condition est un test booléen**, comme pour les instructions if et for.
- Si la valeur retournée est true, la boucle **while exécute Instruction**, puis teste à nouveau la condition.
- Ceci se répète **jusqu'à obtention d'une condition fausse**.



## Boucle do while

### 5.4 CopyArrayWhile

- Les boucles do et while se ressemblent, mais do exécute une instruction ou un bloc donné jusqu'à ce que la condition soit fausse.
- La boucle while testant la condition avant toute exécution, le corps de la boucle n'est jamais exécuté si la condition est fausse dès le départ.
- A l'inverse, do exécute les instructions du corps de la **boucle au moins une fois avant de tester la condition**. La syntaxe d'une boucle do est la suivante :

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



## Exercice

### 5.5 Exercices

- Donner les multiples de 3 existants avant un nombre spécifié
- Donner les nombres premiers existant avant un nombre spécifié



## **6. Gestion des exceptions**





# Utilisation des exceptions

- Une **exception** est une **interruption de l'exécution d'un programme** suite à une erreur.
- Par exemple : une division par zéro provoque une exception de type `ArithmeticException`.
- Java permet non seulement la gestion des exceptions, mais aussi la création d'exceptions spécifiques.
- Les exceptions sont des instances de sous-classes **`java.lang.Error`** et **`java.lang.Exception`**



# Utilisation des exceptions

- Exemple très simple où on génère une exception de division par zéro :

```
class DivParZero {  
    public static void main (String argv[] ) {  
        int zero=0;  
        zero = 1997/zero;    }  
}
```

- En exécutant ce programme, on obtient l'affichage suivant :

```
java.lang.ArithmeticException: / by zero at  
DivParZero.main(DivParZero.java:4)
```

- On distingue :

- Le **nom complet de l'exception** qui a été levée  
(*java.lang.ArithmeticException*)
- Un **message** précisant la **cause de cette erreur** (*/ by zero*).
- L'indication de la classe, de la méthode et du numéro de ligne où s'est produite cette exception (*DivParZero.main(DivParZero.java:4)*)



# Utilisation des exceptions

- On utilise deux mots clefs **try et catch()**, permettant de gérer (intercepter) des exceptions afin de provoquer l'exécution de certaines instructions spécifiques.
- Utilisation de try et catch :

```
try
{
    // zone contenant des instructions
    //pouvant lever des exceptions
}
catch (NomException e)
{
    // traitement de l'exception
}
```



# Utilisation des exceptions

- Un bloc **try{ }** contient un ensemble d'instructions qui peuvent lever des exceptions durant leur exécution.
- On indique donc que l'on va gérer tout ou partie de ces exceptions.
- Cette gestion s'effectue dans un ou **plusieurs blocs catch() { }**
- Dans un tel bloc, on attrape une exception précisée dans l'instruction **catch()**
- Il est important de noter qu'une **exception est également un objet** en Java.
- On peut donc définir **autant de bloc catch()** qu'il y a **d'exceptions susceptibles d'être levées**.

## Exception, exemple

```
class DivParZero {  
public static void main (String argv[] ) {  
    int zero=0;  
    try { zero = 1997/zero; }  
    catch (ArithmeticException e ){  
        System.out.println("Une exception arithmétique a été  
        levée");  
        System.out.println("Message : " + e.getMessage());  
        System.out.println("Pile :");  
        e.printStackTrace(); }  
}}
```

A l'exécution, on obtient :

```
Une exception arithmétique a été levée  
Message : / by zero  
Pile :  
java.lang.ArithmeticException: / by zero at  
DivParZero.main(DivParZero.java:4)
```



## Utilisation des exceptions

- Nous avons donc ici installé un gestionnaire d'exceptions autour de l'affectation provoquant une **division par zéro**.
- Un bloc **catch()** interceptant une exception arithmétique a été défini.
- Dans ce bloc, nous signalons quelle exception a été levée, puis nous affichons le message associé grâce à la méthode **getMessage()** appliquée à l'objet e du type **ArithmeticException**.
- Enfin, nous appelons la méthode **printStackTrace()** qui va nous renvoyer le message affiché par défaut.
- Il faut également savoir qu'il existe un dernier mot clé, **finally**, qui définit un bloc dont les instructions sont systématiquement exécutées, qu'une exception soit levée ou non.

# Utilisation des exceptions

## 6.1 DivParZero

```
Finally
{
    System.out.println("Fin du calcul");
}
System.out.println("Fin du programme");
```

■ Si on exécute le programme ainsi modifié, on obtient :

```
Une exception arithmétique a été levée
Message : / by zero
Pile :
java.lang.ArithmeticException: / by zero at
DivParZero.main(DivParZero.java:4)
Fin du calcul
Fin du programme
```





## Utilisation des exceptions, résumé

```
try
{
    Lève l'exception
    Lorsque l'exception est levée le reste du
    bloc d'instructions n'est pas exécuté
}
catch (Classe de l'exception levée e )
{
    Traitement de l'exception
}
finally
{
    Traitement systématiquement effectué
}
```



# **Exceptions personnalisées**



## Utilisation des exceptions

- D'une manière générale, une exception est une rupture de séquence déclenchée par une instruction ***throw*** comportant une expression de type classe.
- *Il y a alors branchement à un ensemble d'instructions nommé "gestionnaire d'exception"*
- Le choix du bon gestionnaire est fait en fonction du type de l'objet mentionné à ***throw***



## Déclencher une exception

- On considère la classe **Point**, permettant de manipuler des points à des positions  $x, y$
- On souhaite s'assurer que ces derniers ne sont pas négatifs
- Au sein du constructeur, on vérifie la validité de ces derniers en regard des paramètres fournis
- Lorsque l'un d'entre eux est incorrect, nous "déclenchons" une exception à l'aide de l'instruction ***throw***
- Création d'une classe **ErrCoord** héritant de la classe **Exception** :

```
class ErrCoord extends Exception
{ }
```



# Utilisation du gestionnaire d'exception

## 6.2 Exception personnalisée

- Inclure dans le bloc **try**, les instructions susceptibles de provoquer une exception :

```
try
{
    // instructions
}
```

- Utilisation du bloc **catch**, pour intercepter et traiter l'exception :

```
catch (ErrConst e)
{
    System.out.println ("Erreur construction ") ;
    System.exit (-1) ;
}
```



## Gérer plusieurs exceptions

- Par exemple, nous considérons une classe Point munie :
  - du constructeur précédent, déclenchant toujours une exception ErrConst,
  - d'une méthode deplace qui s'assure que le déplacement ne conduit pas à une coordonnée négative ; si tel est le cas, elle déclenche une exception ErrDepl
  - Création d'une ErrDepl levée par la méthode deplace :

```
public void deplace (int dx, int dy) throws ErrDepl
{
    if ( ((x+dx)<0) || ((y+dy)<0)) throw new ErrDepl() ;
    x += dx ; y += dy ;
}
```



# Gérer plusieurs exceptions

## 6.3 Exceptions personnalisées multiples

- On a potentiellement 2 exceptions ErrConst et ErrDepl, interceptée dans le bloc try
- Puis traiter dans leur bloc respectif catch :

```
try
{
    // bloc dans lequel on souhaite detecter les exceptions //
    ErrConst et ErrDepl
}
catch (ErrConst e) // gestionnaire de l'exception ErrConst
{
    System.out.println ("Erreur construction ") ;
    System.exit (-1) ;
}
catch (ErrDepl e) // gestionnaire de l'exception ErrDepl
{
    System.out.println ("Erreur déplacement ") ;
    System.exit (-1) ;
}
```





# Transmission d'informations au gestionnaire

- On peut transmettre des informations au gestionnaire d'exception :
  - par le biais de l'objet fourni dans l'instruction ***throw***,
  - par l'intermédiaire du **constructeur** de l'objet **exception**



# Transmission d'informations par throw

## 6.4 Exception personnalisée throw

- Pour transmettre des informations par le biais de ***throw*** :
  - prévoir les champs appropriés dans la classe d'Exception
  - On peut aussi y ajouter des méthodes

# Transmission d'informations par le constructeur

## 6.5 Exception personnalisée constructeur

- Dans certains cas, on peut se contenter de ***transmettre un "message"*** au gestionnaire, sous forme d'une information de type chaîne.
- On peut aussi exploiter une particularité de la classe *Exception* :
  - Celle-ci dispose d'un **constructeur à un argument de type *String***
  - Dont on peut récupérer la valeur à l'aide de la méthode ***getMessage()***
- Pour en bénéficier, il suffit de prévoir dans la classe exception, un constructeur à un argument de type *String*, qu'on retransmettra au constructeur de la super-classe *Exception*



# Gestion des exceptions - Exercice

## 6.6 Exercice Exception personnalisée

- A partir d'un tableau de Personne (classe Personne), vérifier :
  - Que le nom et prénom ne soit pas nuls
  - Que l'age soit positif



# **7. Variables dynamiques**

## **Les collections**

### **Premier exemple avec Vector**

# Classe Vector

- Appartient aux collections
- Implémente une matrice extensible d'objets , package **java.util**
- avant Java 1.5 :
  - Pas de typage, chaînage d'objets anonymes
  - Obligation de caster lors de la récupération des valeurs
- depuis Java 1.5 :
  - Typage possible des objets chaînés
  - Pas d'obligation de cast
- Création :
  - Initialement vide :  
avant 1.5 : `Vector v = new Vector();`  
depuis 1.5 : `Vector<type objet> v = new Vector<type objet>();`
  - Taille prédéfinie :  
avant 1.5 : `Vector v = new Vector(25);`  
depuis 1.5 :
  - Taille prédéfinie, avec pas d'incrémentatation :  
avant 1.5 : `Vector v = new Vector(25, 5);`  
depuis 1.5 : `Vector v = new Vector(25, 5);`



## Classe Vector - méthodes

add()	Ajout d'un élément	v.add("Watson");
lastElement()	Position sur la dernière occurrence du vecteur	v.lastElement();
get(indice)	Récupère l'élément à la position indice	v.get(0);
remove(indice)	Suprime l'élément à la position indice	v.remove(3);
set(indice)	Changer la valeur à la position indice	v.set(1, "Woods");
clear()	Efface l'objet Vector	v.clear();
contains()	Recherche par comparaison d'un élément précis	boolean isThere = v.contains("O'Meara");





## Classe Vector - méthodes

indexOf()	Retourne l'indice d'un élément sélectionné	<code>int i = v.indexOf("Nicklaus");</code>
removeElement()	Supprime un élément par valeur	<code>v.removeElement("Woods");</code>
size()	Taille de la liste de vecteurs	<code>int size = v.size();</code>
setSize()	Force la taille de la liste de vecteurs	<code>v.setSize(10);</code>
trimToSize()	Force la capacité de la liste de vecteurs	<code>v.trimToSize();</code>
capacity	Donne la capacité de la liste de vecteurs	<code>int capacity = v.capacity();</code>

- On peut implémenter **l'interface Iterator** pour le parcours de la liste de vecteurs séquentiellement
- Suivant le type de collection :
  - Offre un niveau d'abstraction (ne tient pas compte de l'implémentation de la collection)
  - Permet d'itérer sur la collection, lorsque le type de collection ne permet pas l'utilisation de boucles classiques
- Exemple :
  - Création d'un itérateur, et affectation au vecteur v :

```
Iterator<type objet de v> it = v.iterator();
```

- Parcours du vecteur :

```
while(it.hasNext())  
{  
    System.out.println("occurence : "+it.next());  
}
```



# Interface Iterator

## 7.1 Les vecteurs

- Permet le parcours par itérations d'une liste d'éléments

<code>public boolean hasNext();</code>	Détermine si la structure de données contient des éléments supplémentaires.
<code>public Object next();</code>	Récupère l'élément suivant dans une structure. S'il n'y a plus d'éléments, la méthode <code>next ( )</code> générera une exception <code>NoSuchElementException</code> .
<code>public void remove();</code>	Supprime l'élément de la liste



# Lecture du clavier

## 7.2 Lecture Clavier

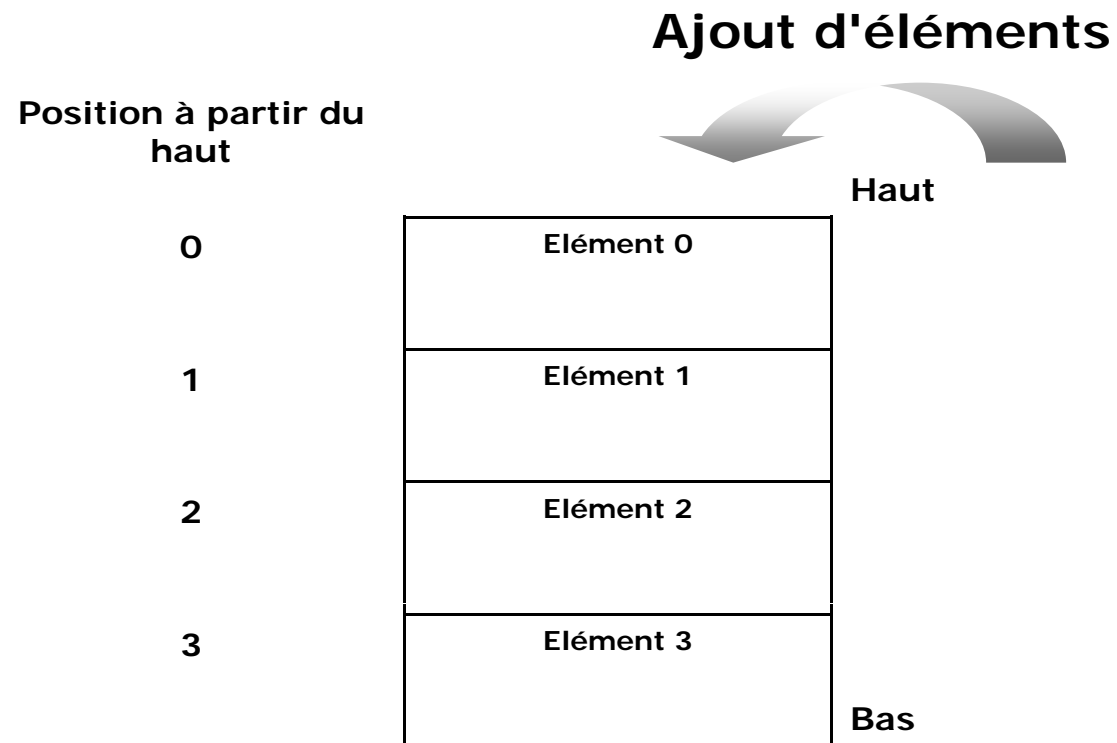
- Utilisation de l'objet **System.in** : entrée standard
- Utilisation de flux en entrée (vu plus loin : InputStream)
- Exemple :

```
// classe fournissant des fonctions de lecture au clavier
import java.io.* ;
public class Clavier
{
    public static void main (String args){
        String ligne_lue = null ;
        try
        {
            InputStreamReader lecteur = new InputStreamReader (System.in) ;
            BufferedReader entree = new BufferedReader (lecteur) ;
            ligne_lue = entree.readLine();
            System.out.println(ligne_lue);
        }
        catch (IOException err)
        {
            System.exit(0) ;
        }
    }
}
```

### 7.3 Exercice SaisieVecteur

- Exemple : Saisie d'informations aux clavier et mise en vecteur de chaque ligne saisie
- Puis après saisie, affichage du contenu du vecteur

- Utilisée pour les structures de piles





## Classe Stack - méthodes

- Création : Un seul constructeur :

```
Stack s = new Stack();
```

push()	Ajout d'un élément	s.push("Un");
pop()	Récupère et supprime un élément (au sommet)	s.pop();
peek()	Atteindre le premier élément de la pile	s.peek();
search()	Retourne la distance séparant l'élément recherché du sommet de la pile	s.search("Two");
empty()	Vrai si la pile est vide, faux sinon	s.empty();

**Possibilité d'utilisation d'indices**



# Classe Stack - exemple

## 7.4 Les Piles

```
Stack s = new Stack(); // Création de la pile s
```

```
s.push("Un");           // Ajout des valeurs  
s.push("Deux");  
s.push("Trois");  
s.push("Quatre");  
s.push("Cinq");  
s.push("Six");
```

```
String s1 = (String)s.pop(); // s1 vaut "Six"  
String s2 = (String)s.pop(); // s2 vaut "Cinq"  
String s3 = (String)s.peek(); // s3 vaut "Quatre"
```

```
int i = s.search("Deux"); // retourne 2
```

```
boolean isEmpty = s.empty(); // retourne false
```





## **8. Création des classes et des applications**



## Création d'une classe

- Pour définir une classe, on utilise le mot clé `class` et le nom de la classe :

```
class MaClasse  
{  
}
```

- Par défaut, les classes héritent de la classe `object`.
- Si la classe nouvellement créée est une sous-classe d'une autre classe particulière (ce qui veut dire qu'elle hérite de cette classe), on utilise `extends` pour indiquer la super-classe de la nouvelle classe :

```
class MaClasse extends MaSuperClasse  
{  
}
```



## Création de variables d'instance

- Les variables d'instance se déclarent dans la définition de classe.
- Les variables sont considérées comme des variables d'instance lorsqu'elles sont déclarées en dehors d'une définition de méthode.
- Cependant, l'usage veut que la plupart des variables d'instance soient définies aussitôt après la première ligne de la définition de classe.
- Exemple de définition :
- Cette définition de classe comporte cinq variables d'instance :

```
class Bicycle extends VehiculeSansMoteur
{
    String bikeTypeBicycle;
    int NbVitesse;
    int Pignons;
    int VitesseAvant;
    int VitesseArriere;
}
```

- Une **constante**, est une 'variable' dont la valeur ne change jamais.
- Les constantes servent à définir des valeurs communes à toutes les méthodes d'un objet, et à donner des noms significatifs aux valeurs immuables de l'objet donné.
- En Java, on peut créer des **variables d'instance constantes**, ou des **variables de classe constantes**, mais pas de variables locales constantes.
- Pour déclarer une constante, placez le **mot clé final** devant la déclaration de la variable, et initialisez-la :

```
final float pi = 3.14592;
```



## Création de variables de classe

- Les variables de classe sont globales pour une classe et toutes ses instances.
- Les variables de classe sont utiles pour faire communiquer entre eux différents objets d'une même classe, ou pour garder la trace de propriétés communes à un groupe d'objets.
- Pour déclarer une **variable de classe**, utilisez le mot **static** dans la déclaration de classe :

```
static int sum;  
static final int maxObjets = 10;
```



# Définition de méthodes

- Une définition de méthode comporte **quatre parties essentielles** :
  - Le **nom** de la méthode,
  - Le **type d'objet** / de donnée primaire **retourné par la méthode**
  - **Une liste d'arguments**,
  - Le **corps de la méthode**.

```
TypeRetourné nomMéthode (liste de arguments)
{
    Corps de la méthode
}
```

- Les **trois premières parties** de la définition d'une méthode forment ce que l'on appelle **sa signature**, qui résume les informations les plus importantes sur la méthode elle-même.





# Définition de méthodes

## 8.1 RangeClass

- Si il y a un type retourné :

```
typeRetourné nomMéthode(liste d'arguments) {  
...  
return; }
```

- S'il n'y a aucun type retourné :

```
void nomMéthode(liste d'arguments) {  
...  
}
```

- **Référence à l'objet courant** (celui qui contient la méthode), ou de se référer aux variables d'instance de cet objet, ou de passer cet objet comme argument à une autre méthode.

- Objet courant : **mot clé this**, à n'importe quel endroit où l'objet pourrait apparaître :

- en notation point pour se référer aux variables d'instance de l'objet, comme argument d'une méthode, comme valeur de retour de la méthode courante, etc.

- Exemples :

```
t = this.x;    // la variable d'instance x de cet objet
```

```
this.maMethode(this); // appelle la méthode maMethode,  
// définie dans la classe de l'objet courant, et lui passe  
// l'objet courant.
```

- **On peut omettre complètement le mot clé this.**
- Référence simplement par nom : variables d'instance, méthodes définies dans la classe courante (this est implicite dans de telles références).

- Ainsi, les deux premiers exemples ci-avant peuvent s'écrire :

```
t = x;          // la variable d'instance x de cet objet  
maMethode(this); // appelle la méthode maMethode,  
// définie dans la classe de l'objet courant, et lui passe  
// l'objet courant.
```

- On peut ou non omettre le mot clé this selon qu'il existe ou non des variables de même nom déclarées dans la portée locale
- *Les méthodes de classe, c'est-à-dire les méthodes déclarées avec le mot clé static, ne peuvent pas utiliser this*

## 8.2 ScopeTest

```
Classe MaClasse
{
    déclaration variables de classes //(*)
    déclaration variables d'instance //(**)

    voir méthode M1() {
        déclaration variables locales //(**)
    }
    ...
}
```

- (\*) Connues de tous les objets de la classe
- (\*\*) Connues de l'objet concerné
- (\*\*) Connues de la méthode seule



## Méthode de classe

- Pour définir une méthode de classe, utilisez le mot clé **static** devant la définition de la méthode, comme vous le feriez pour créer une variable de classe.
- Par exemple, la méthode de classe max pourrait avoir la signature ci-après :

```
static int max(int arg1, int arg2)
{
}
```



## **9. Compléments sur les méthodes**

- En Java, les méthodes peuvent être **polymorphes** (ou *surchargées/overloading*) :
- **Plusieurs versions d'une méthode du même nom**, mais avec des signatures et des définitions différentes.
- Comportements différents selon les arguments passés à la méthode.
- Généralement pour aboutir au mêmes résultats avec des arguments différents
- Exemple : **méthode Date() polymorphe** servant à définir un objet Date.



- Java distingue les méthodes polymorphes selon :
  - le type des arguments passés
  - le nombre des arguments passés
- mais ne tient pas compte du type de retour.

***Si vous tentez de créer deux méthodes ayant le même nom et la même liste de paramètres, mais des types de retour différents, une erreur se produira lors de la compilation.***

- Servent à **initialiser les nouveaux objets** lors de leur création (méthode).
- Elles **ne peuvent pas être appelées directement** ; elles sont en revanche appelées automatiquement par Java lorsque vous créez un nouvel objet.
- Si aucune méthode constructeur n'est définie pour une classe, un objet peut quand même être créé (constructeur par défaut, mais initialisation des variables d'instance à faire par la suite).
- La définition de méthodes constructeurs dans les classes permet :
  - De fixer les valeurs initiales des variables d'instance
  - D'appeler des méthodes basées sur ces variables ou sur d'autres objets, ou de calculer les propriétés initiales de l'objet.
- Les **constructeurs peuvent être polymorphes**, idem méthodes ordinaires

- Les constructeurs ressemblent beaucoup aux méthodes ordinaires, à deux différences essentielles près :
  - Les constructeurs portent toujours le même nom que la classe.
  - Les constructeurs n'ont pas de type de retour.

# Constructeurs - exemple

## 9.2 Constructeur

```
class Personne {
    String nom;int age;

    Personne(String n,int a){
        nom = n;
        age = a;}

    void printPersonne(){
        System.out.print("Hello, mon nom est " + nom);
        System.out.println(". J'ai " + age + " ans.");}

    public static void main (String args[]){
        Personne p;
        p = new Personne("Laura",20);
        p.printPersonne();
        System.out.println("-----");
        p = new Personne("Tommy",6);
        p.printPersonne();
        System.out.println("-----");}}
```



## Redéfinition de méthode

- A l'appel d'une méthode depuis un objet, Java cherche la définition de la méthode dans la classe de cet objet (même signature)
- S'il ne la trouve pas :
  - il remonte dans la hiérarchie des classes jusqu'à ce qu'il trouve la bonne définition (liens d'héritage)
- Dans certaines situations, on veut qu'un objet réagisse à l'appel d'une méthode avec un comportement différent de celui prévu dans la définition de niveau supérieur.
  - Pour cela, vous devez redéfinir la méthode en question.

***Une redéfinition de méthode consiste à définir dans une sous-classe une méthode déjà définie dans une super-classe avec la même signature.***

***Si cette méthode est appelée, c'est la méthode de la sous-classe qui est trouvée et exécutée à la place de celle de la super-classe.***



## Redéfinition de méthode

- Pour redéfinir une méthode :
  - Créer dans une sous-classe une méthode ayant la même signature :
    - nom,
    - type de retour
    - liste d'arguments
  - Que la méthode définie dans l'une des super-classes de la sous-classe en question.
- Java exécutant d'abord la première définition de méthode qu'il trouve et qui comporte la bonne signature,
- La définition d'origine de la méthode est ainsi effectivement "masquée".



# Appel de la méthode d'origine

## 9.3 NamedPoint

■ Pour appeler la méthode d'origine depuis l'intérieur d'une définition de méthode, utilisez le mot clé **super**, qui fait remonter l'appel de méthode vers le haut de la hiérarchie :

■ Exemple :

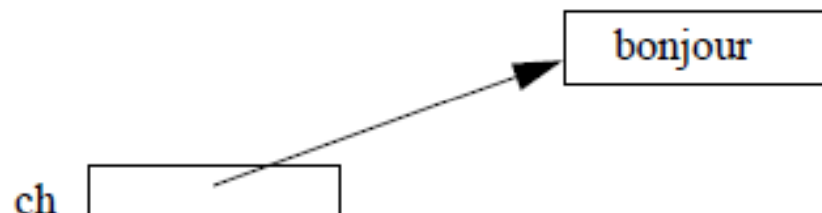
```
void myMethod (String a, String b)
{
    // faire quelque chose ici
    super.myMethod(a, b);
    // éventuellement autres instructions
}
```





## **10. Chaînes de caractères et types énumérés**

- Comme toute déclaration d'une variable objet, l'instruction :  
**String ch ;**
  - déclare que **ch** est destinée à contenir une **référence à un objet de type String**.
- La notation : **"bonjour"**
  - désigne en fait **un objet de type String** (ou, pour être plus précis, sa référence), **créé automatiquement** par le compilateur.
- Ainsi, avec **ch = "bonjour" ;**
  - on aboutit à une situation qu'on peut schématiser ainsi :

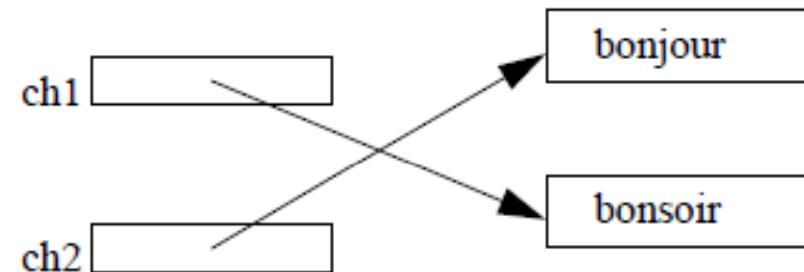
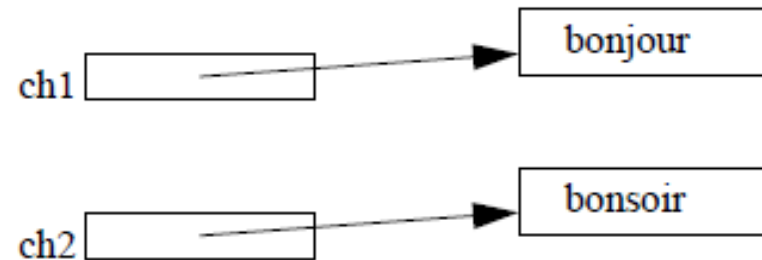


# Un objet String n'est pas modifiable

- **Pas de méthode** permettant d'en **modifier la valeur**
  - On manipule des références à des objets
  - celles-ci peuvent voir leur valeur évoluer au fil du programme
- Exemple :

```
String ch1, ch2, ch ;  
ch1 = "bonjour" ;  
ch2 = "bonsoir" ;
```

```
ch = ch1 ;  
ch1 = ch2 ;  
ch2 = ch ;
```



*Les deux objets de type chaîne n'ont pas été modifiés, mais les références `ch1` et `ch2` l'ont été*



## Longueur d'une chaîne - length

- La méthode **length()** permet d'obtenir la longueur d'une chaîne :
  - nombre de caractères qu'elle contient

- Exemple :

```
String ch = "bonjour" ;  
int n = ch.length() ;           // n contient 7  
ch = "hello" ; n = ch.length () ; // n contient 5  
ch = "" ; n = ch.length () ;    // n contient 0
```

- Remarque

- Contrairement aux tableaux(où **length** désignait un champ),
- C'est une méthode pour les objets String

# Accès aux caractères d'une chaînes - charAt

- La **méthode charAt** permet d'accéder à un caractère de rang donné d'une chaîne
  - Le premier caractère porte le rang 0)
  - Exemple :

```
String ch = "bonjour" ;  
ch.charAt(0); // correspond au caractère 'b',  
ch.charAt(2); // correspond au caractère 'n'.
```

- Remarque : Possibilité d'utiliser la boucle **for each** (JDK 5.0)
- Exemple :

```
String mot = "Langage Java";  
for (char c : mot.toCharArray()) System.out.println(c);
```

*Cette boucle ne permet que des consultations, et en aucun cas des modifications*

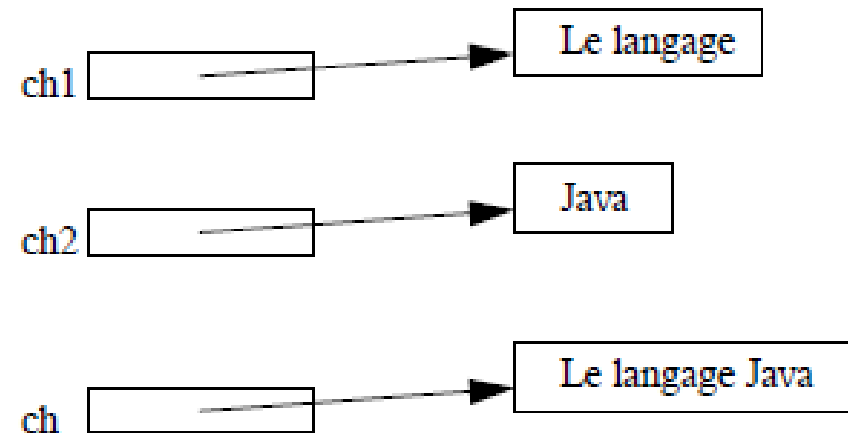
*Ici, cela ne constitue pas une contrainte puisqu'un objet de type String n'est, de toute façon, pas modifiable*

# Concaténation de chaînes

- L'opérateur `+` est défini lorsque ses deux opérandes sont des chaînes
- Il fournit en résultat une nouvelle chaîne formée de la concaténation des deux autres

■ Exemple :

```
String ch1 = "Le langage " ;  
String ch2 = "Java" ;  
String ch = ch1 + ch
```



## Conversion des opérandes de l'opérateur +

- Java autorise à mélanger chaînes et expressions d'un type primitif.

- Exemple :

```
int n = 26 ;  
System.out.println ("n = " + n) ; // affiche : n = 26
```

- Lorsque **l'opérateur +** possède un opérande de type String, l'autre est converti en chaîne

- Exemple :

```
int n = 26 ;  
String titre = new String ("resultat : ") ;  
String monnaie = "$"  
String resul = titre + n + " " + monnaie ;  
System.out.println (resul) ; // affichera : resultat : 26 $
```



## Recherche dans une chaîne - indexOf

- La méthode **indexOf** permet de rechercher, à partir du début d'une chaîne ou d'une position donnée :
  - La première occurrence d'un caractère donné,
  - La première occurrence d'une autre chaîne.
- Dans tous les cas, elle fournit :
  - La position du caractère (ou du début de la chaîne recherchée) si une correspondance a effectivement été trouvée,
  - La valeur -1 sinon.
- **lastIndexOf()**, Même principe, mais en examinant la chaîne depuis sa fin.

### ■ Exemple :

```
String mot = "anticonstitutionnellement" ;  
int n ;  
n = mot.indexOf ('t') ;           // n vaut 2  
n = mot.lastIndexOf ('t') ;      // n vaut 24  
n = mot.indexOf ("ti") ;         // n vaut 2  
n = mot.lastIndexOf ("ti") ;     // n vaut 12  
n = mot.indexOf ('x') ;          // n vaut -1
```



# Comparaisons de chaînes

- Opérateurs == et !=

- rôle de ces opérateurs dans le cas d'un objet : comparaison d'une référence à un objet) s'appliquent aux chaînes (String)

- ils comparent les références fournies comme opérandes (et non les objets référencés).

***Deux chaînes de valeurs différentes ont toujours des références différentes***

***En revanche, deux chaînes de même valeur ne correspondent pas nécessairement à un seul et même objet***

***Exemple :***



# Comparaisons de chaînes

## 10.1 Les chaînes

- La **méthode equals** compare le **contenu** de deux chaînes :

- Exemple :

```
String ch1 = "hello" ;  
String ch2 = "bonjour" ;  
.....  
ch1.equals(ch2) // cette expression est fausse  
ch1.equals("hello") // cette expression est vraie
```

- La méthode **equalsIgnoreCase** effectue la même comparaison, mais sans distinguer les majuscules des minuscules

- Exemple :

```
String ch1 = "HeLlo" ;  
String ch2 = "hello" ;  
ch1.equalsIgnoreCase(ch2) // cette expression est vraie  
ch1.equalsIgnoreCase("hello") // cette expression est vraie
```

# Comparaisons de chaînes

- La méthode **compareTo** s'utilise ainsi :  
`chaîne1.compareTo(chaîne2)`
- Elle fournit :
  - Un entier **négatif** si chaîne1 arrive **avant** chaîne2,
  - Un entier **nul** si chaîne1 et chaîne2 sont **égales** (on a alors `chaîne1.equals(chaîne2)`),
  - Un entier **positif** si chaîne1 arrive **après** chaîne2.

chaîne1	chaîne2	chaîne1.compareTo(chaîne2)
"bonjour"	"monsieur"	négatif
"bonj"	"bonjour"	négatif
"paris2"	"paris10"	positif (car '2' > '1')
"Element"	"element"	négatif (car 'E' < 'e')
"Element"	"élément"	négatif (car 'E' < 'é')
"element"	"élément"	égatif (car 'e' < 'é')



## Modification de chaînes

- Les objets de type **String** ne sont **pas modifiables**
- La classe String dispose de quelques méthodes qui créent une nouvelle chaîne obtenue par transformation de la chaîne courante.

- **Remplacement de caractères :**

- La méthode **replace** crée une chaîne en remplaçant toutes les occurrences d'un caractère donné par un autre.

- Exemple :

```
String ch = "bonjour" ;  
String ch1 = ch.replace('o', 'a') ; // ch n'est pas modifiée  
// ch1 contient "banjaur"
```



# Modification de chaînes

- Extraction de **sous-chaînes** :
- La méthode **substring** permet de créer une nouvelle chaîne en extrayant de la chaîne courante :
  - soit tous les caractères **depuis une position donnée**
  - Exemple :

```
String ch = "anticonstitutionnellement" ;  
String ch1 = ch.substring (5) ; // ch n'est pas modifiée  
// ch1 contient "onstitutionnellement"
```
  - soit tous les caractères **compris entre deux positions données** (la première incluse, la seconde exclue) :
  - Exemple :

```
String ch = "anticonstitutionnellement" ;  
String ch1 = ch.substring (4, 16) ; // ch n'est pas modifiée  
// ch1 contient "constitution"
```

- **Passage en majuscules ou en minuscules**
- La méthode **toLowerCase** crée une nouvelle chaîne en remplaçant toutes les majuscules par leur équivalent en minuscules
- La méthode **toUpperCase** crée une nouvelle chaîne en remplaçant toutes les minuscules par leur équivalent en majuscules.
- Exemple :

```
String ch = "LanGaGE_3" ;  
String ch1 = ch.toLowerCase() ; // ch est inchangée  
// ch1 contient "langage_3"  
.....  
String ch2 = ch.toUpperCase() ; // ch n'est pas modifiée  
// ch2 contient "LANGAGE_3"
```





## Modification de chaînes

- **Suppression des séparateurs de début et de fin :**
- La méthode **trim** crée une nouvelle chaîne en supprimant les éventuels séparateurs de début et de fin (espace, tabulations, fin de ligne) :
- Exemple :

```
String ch = " \n\tdes separateurs avant, pendant\t\n et apres \n\t ";  
String ch1 = ch.trim() ;  
// ch n'est pas modifiée, ch1 contient la chaîne :  
//      "des separateurs avant, pendant\t\n et apres"
```



# Arguments de la ligne de commande

- L'en-tête de la méthode main :

```
public static void main (String args[])
```

- Elle reçoit un argument du type **tableau de chaînes** destiné à contenir les éventuels arguments fournis au programme lors de son lancement.
- Lorsque le programme est lancé à partir d'une ligne de commande, ces **arguments sont indiqués à la suite de l'appel du programme**
- Avec des environnements de développement intégré, la démarche est différente.
- Facile de récupérer ses arguments dans la méthode main, la longueur du tableau reçu indique le nombre d'arguments

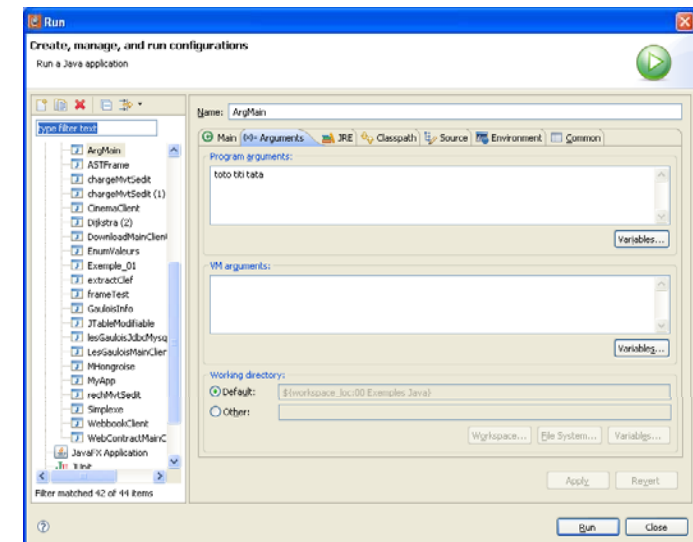
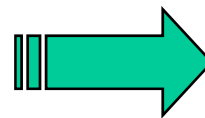
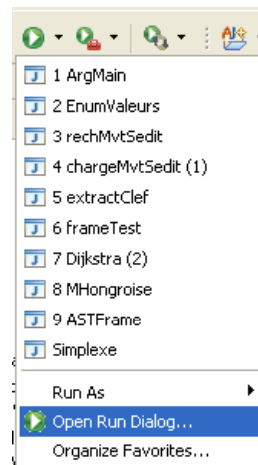
# Arguments de la ligne de commande

## 10.2 ArgMain

■ Exemple :

```
public class ArgMain
{
    public static void main (String args[])
    {
        int nbArgs = args.length ;
        if (nbArgs == 0) System.out.println ("pas d'arguments");
        else
        {
            for (int i=0 ; i<nbArgs ; i++)
                System.out.println ("argument numero " + i + " = " +
args[i]) ;
        }
    }
}
```

Eclipse :





## Classe StringBuffer

- Les objets de type **String** ne sont pas modifiables, mais il est possible de les employer pour effectuer la plupart des manipulations de chaînes
- La modification d'une chaîne ne peut se faire qu'en créant une nouvelle chaîne
- Dans les programmes manipulant intensivement des chaînes, la **perte de temps** qui en résulte peut devenir gênante.
- Java dispose de la **classe StringBuffer** destinée à la manipulation de chaînes, dans laquelle les **objets sont modifiables**.



# Classe StringBuffer

- La classe **StringBuffer** :
  - dispose de fonctionnalités classiques
  - n'a aucun lien d'héritage avec String
  - ses méthodes ne portent pas toujours le même nom
- On peut créer un objet de type **StringBuffer** à partir d'un objet de type **String**.



# Classe StringBuffer

- Les méthodes :

- **Modification** d'un caractère de rang donné : **setCharAt**,

- **Accès** à un caractère de rang donné : **charAt**,

- **Ajout** d'une chaîne en fin : la méthode **append** accepte des arguments de tout type primitif et de type String,

- **Insertion** d'une chaîne en un emplacement donné : **insert**,

- **Remplacement** d'une partie par une chaîne donnée : **replace**,

- **Conversion** de StringBuffer en String : **toString**.

# Classe StringBuffer

## 10.3 TestStb

### ■ Exemple :

```
class TestStB
{
    public static void main (String args[])
    {
        String ch = "la java" ;
        StringBuffer chBuf = new StringBuffer (ch) ;
        System.out.println (chBuf) ;
        chBuf.setCharAt (3, 'J'); System.out.println (chBuf)
;
        chBuf.setCharAt (1, 'e') ; System.out.println (chBuf)
;
        chBuf.append (" 2") ; System.out.println (chBuf) ;
        chBuf.insert (3, "langage ") ; System.out.println
(chBuf) ;
    }
}
```

```
la java
la Java
le Java
le Java 2
le langage Java 2
```





## Les types énumérés

- À l'origine, Java ne disposait pas des "**types énumérés**", présents dans bon nombre d'autres langages
- Cette lacune a été comblée par la version **1.5**
- Il est dorénavant **possible de définir** en Java un **type** dont on choisit explicitement **les identificateurs des constantes**.
- Ce type, implémenté sous forme **d'une classe**, dispose d'autres propriétés intéressantes comme la possibilité de lui **ajouter des méthodes spécifiques**



## Les types énumérés - définition

- **type énuméré / classe d'énumération** : types dont les valeurs, en nombre fini, sont définies par des identificateurs

- Exemple :

```
enum Jour { lundi, mardi, mercredi, jeudi, vendredi,  
          samedi, dimanche }
```

- Les valeurs de ce **type nommé Jour** sont les **7 constantes** notées :

`lundi, mardi, mercredi, jeudi, vendredi, samedi et dimanche`



## Les types énumérés - définition

- On peut alors définir classiquement des objets de type Jour :

```
Jour courant, debut ;
```

- et leur affecter des valeurs constantes de ce même type :

```
courant = Jour.mercredi ;  
debut = Jour.lundi ;
```

- **Jour** est une classe, bien que le mot-clé **class** ne figure pas dans sa définition (sa présence constituerait une erreur)

- Les **7 constantes** (lundi, mardi, ...) en sont **des instances** (et non des champs) **finales**, donc **non modifiables**, comme on peut le souhaiter



## Les types énumérés - définition

- Les **types énumérés** dérivent de la classe **Enum**.
- Il ne faut pas oublier de "**préfixer**" les constantes du type énuméré par le **nom de la classe** correspondante.
- Par exemple, si l'on utilise **mardi** à la place de **Jour.mardi**, on obtiendra une erreur de compilation.



## Les types énumérés - comparaisons

- **Comparaisons d'égalité :**

- Exemple :

```
if (courant == Jour.dimanche) ...
```

- On peut utiliser indifféremment les opérateurs == ou **equals**,

- Le premier porte sur l'adresse de l'objet (constant)

- Le second sur sa valeur, car les objets constants du type ne sont instanciés qu'une seule fois.



## Les types énumérés - comparaisons

- **Comparaisons basées sur un ordre**
- Pas possible d'utiliser les opérateurs arithmétiques usuels sur les types énumérés.
- On peut recourir à la méthode **compareTo** (héritée de la classe Enum) qui se fonde sur l'ordre dans lequel on a déclaré les constantes.
- Exemple :

```
courant = Jour.mercredi ;  
courant.compareTo(Jour.mardi) sera positif,  
courant.compareTo(Jour.vendredi) sera négatif,  
courant.compareTo(Jour.mercredi) sera nul
```

## Les types énumérés - comparaisons

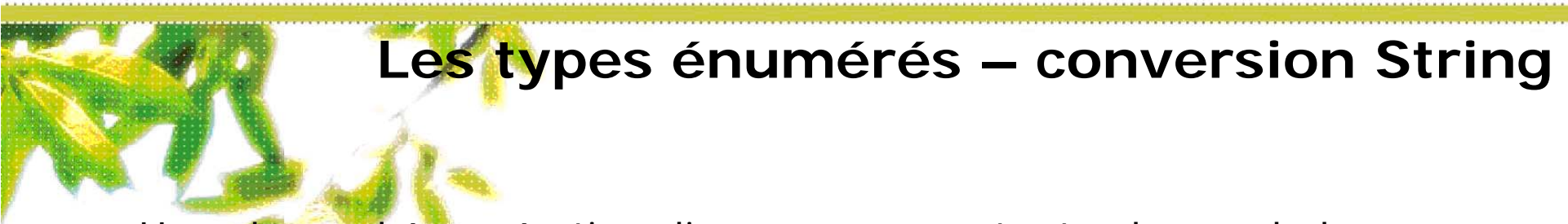
- Il existe également dans la classe **Enum**, une méthode nommée **ordinal()**
- Fournit le rang d'une valeur donnée dans la liste des constantes du type
- La première valeur est de rang 0.
- Exemple :

```
mardi.ordinal();           // vaut 1
```

- Après l'affectation :

```
courant = mercredi ; // courant.ordinal() vaut 2
```





## Les types énumérés – conversion String

- Une classe d'énumération dispose, comme toute classe, de la méthode **toString**
  - Fournit une chaîne correspondant au libellé de la constante.
  - Exemple :  
`Jour.lundi.toString()`
    - aura comme valeur la chaîne 'lundi' .
- Méthode réciproque de **toString** : **valueOf**
  - Fournit l'objet constant d'un type énuméré correspondant à un libellé qu'on indique sous forme de chaîne.
  - Exemple :  
`Jour.valueOf ("mardi")`
    - fournit la référence à l'objet constant Jour.mardi.



## Les types énumérés – itérations

- Besoin de parcourir les différentes valeurs d'un type énuméré.
- Impossible d'utiliser la boucle **for** usuelle => les opérateurs arithmétiques ne s'appliquent pas à un type énuméré.
- Utilisation de la boucle **for... each**, introduite elle aussi par le JDK 5.0
  - Nécessité de créer d'abord un **tableau des valeurs du type**
  - On peut ensuite itérer avec la boucle **for... each**.
  - Ce tableau peut être obtenu à l'aide de la méthode **values()** de la classe Enum.

# Les types énumérés – itérations

## 10.4 TestEnum

### ■ Exemple :

```
Jour.values()
```

- fournit un tableau formé des 7 valeurs du type Jour, exactement comme si nous avions procédé ainsi :

```
Jour[0] = Jour.lundi ;  
Jour[1] = Jour.mardi ;  
...  
Jour[7] = Jour.dimanche ;
```

- On peut alors appliquer ainsi la boucle for... each à ce tableau :  

```
for (Jour j : Jour.values() )  
    // faire quelque chose avec j
```

# Les types énumérés – Ajout de méthodes

## 10.5 EnumMethode

- Possible de définir des **méthodes spécifiques** à l'intérieur de la classe constituée par un type énuméré.

- Exemple :

- ajout à notre type Jour, une méthode nommée affiche qui affiche simplement le nom du jour :

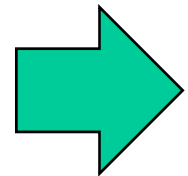
```
enum Jour
{
    lundi, mardi, mercredi, jeudi, vendredi, samedi, dimanche ;
    // notez le ";" ici
    public void affiche ()
    {
        System.out.println (this.toString() ) ;
    }
}
```

- Un point-virgule termine la liste des constantes du type
- et précède les définitions de méthodes.

# Les types énumérés – Exercice commenté

## 10.6 Examen

- Calcul de résultats d'examen d'un ensemble d'élèves
- Chaque élève est représenté par un objet de type *Eleve*, comportant *obligatoirement les champs suivants* :
  - le nom de l'élève (type *String*),
  - son admissibilité à l'examen, sous forme d'une valeur d'un type énuméré comportant les valeurs suivantes : N (non admis), P (passable), AB ( *Assez bien*), B (*Bien*), TB (*Très bien*).
- Les élèves sont fournis par un tableau de chaînes placé dans le programme principal
- pour chaque élève, on lit au clavier 3 notes d'examen, on calcule la moyenne et renseigne convenablement le champ d'admissibilité, suivant les règles usuelles :
  - moyenne < 10 : Non admis
  - 10 <= moyenne <12 : Passable
  - 12 <= moyenne <14 : Assez bien
  - 14 <= moyenne <16 : Bien
  - 16 <= moyenne : Très bien
- On affiche l'ensemble des résultats en fournissant en clair la mention obtenue





# Les types énumérés – Exercice commenté

## 10.6 Exercice Examen

- 3 classes :
  - Examen.java : Programme principal
  - Clavier.java : permet de lire des entrée clavier dans un type donné (fournie)
  - Eleve.java : Classe de définition des élèves

# Les types énumérés – Exercice commenté

## 10.6 Examen

- **Eleve.java** : Classe de définition des élèves

```
class Eleve
{
    private String nom ;
    private Mention resul ;

    public Eleve (String n)
    {
        this.nom = n;
        this.resul = Mention.NC ; // valeur par défaut
    }

    public String getNom(){return this.nom ;}
    public Mention getResul() {return this.resul ;}

    public void setNom(String nom) { this.nom = nom;}
    public void setResul(Mention r) { this.resul = r ;}

}
```



# Les types énumérés – Exercice commenté

## 10.6 Examen

- **Examen.java** : Programme principal

```
// Définition de l'énumération
enum Mention
{
    NA ("Non admis"), P ("Passable"), AB ("Assez bien"),
    B ("Bien"), TB ("Tres bien"), NC ("Non connu");

    private Mention (String d) {mention = d ;}

    public String details() { return mention;}

    private String mention;
}
...
```

# Les types énumérés – Exercice commenté

## 10.6 Examen

■ **Examen.java** : Programme principal

```
...
public class Examen {
    // méthode qui demande au clavier trois notes pour un nom
    // donné et fournit en retour la moyenne correspondante
    static public double moyenne (String n)
    {
        System.out.println ("donnez les trois notes de
        l'eleve " + n) ;
        double som = 0. ;
        for (int i=0 ; i<3 ; i++)
        {
            double note = Clavier.lireDouble() ;
            som += note ;
        }
        double moyenne = som / 3. ;
        return moyenne ;
    }
    ...
}
```

# Les types énumérés – Exercice commenté

## 10.6 Examen

■ **Examen.java** : Programme principal

```
...
// méthode qui définit la mention en fonction de la
// moyenne
static public Mention resul (double m)
{
    if ( m<10. ) return Mention.NA ;
    if ( m<12.0) return Mention.P ;
    if ( m<14.0) return Mention.AB ;
    if ( m<16.0) return Mention.B ;
    return Mention.TB ;
}
...
```

# Les types énumérés – Exercice commenté

## 10.6 Examen

■ **Examen.java** : Programme principal

```
...
public static void main (String args[])
{
    String noms[] = { "Dutronc", "Dunoyer", "Lechene",
                      "Dubois", "Frenet" } ;
    // creation du tableau d'eleves
    int nel = noms.length ;

    Eleve eleves [] = new Eleve [nel] ;
    for (int i=0 ; i<nel ; i++)
        eleves [i] = new Eleve (noms[i]) ;
    // lecture des notes et détermination du résultat de
    // chaque élève
    for (Eleve el : eleves)
    {
        double moy = moyenne (el.getNom()) ;
        el.setResul ((resul(moy))) ;
    }
}
```

# Les types énumérés – Exercice commenté

## 10.6 Examen

- **Examen.java** : Programme principal

```
...  
    // affichage résultats  
    System.out.println ("Resultats : ") ;  
    for (Eleve el : eleves)  
        System.out.println (el.getNom() + " - " +  
                             el.getResul().details()) ;  
    }  
}
```



# **11. La programmation générique**



# Programmation générique

- Ecrire un code unique utilisable avec des objets ou variables quelconques
- Deux aspects différents :
  - Le type en question est effectivement quelconque, par exemple écrire un programme de tri pour des structures différentes :
    - Double, String ou autre ..
  - Existe depuis 1<sup>o</sup> version de Java (toute classe dérive de la classe Object)
- Depuis 1.5 : Le type est non précisé lors de l'écriture de la classe ou de la méthode, on utilise dans ce cas un 'paramètre de type' utilisable aussi bien dans des classes que des méthodes
- On parle alors de **classes ou de méthodes génériques**





# Notion de classe générique

- Exemple :
  - Classe permettant de manipuler des couple d'objets, c'est-à-dire la réunion de 2 objets d'un même type
  - Elle contient :
    - Un constructeur
    - Une méthode affichant les valeurs du couple : **affiche()**
    - Une méthode donnant la valeur du 1° dans le couple : **getPremier()**

# Notion de classe générique

- Classe : Couple.Java

"paramètre de type" nommé ici T

```
class Couple<T>
{
    private T x, y ; // les deux elements du couple

    public Couple (T premier, T second)
    {
        x = premier;
        y = second ;
    }

    public void affiche ()
    {
        System.out.println ("premiere valeur : " + x + " -
        deuxieme valeur : " + y) ;
    }
    T getPremier () { return x ;}
}
```



# Utilisation de classe générique

## 11.1 Couple

- Lors de la déclaration d'un type couple, on devra préciser le type réellement utilisé

- Par exemple :  

```
// ci est un Couple d'objets de type Integer  
Couple <Integer> ci ;
```


```
// cp est un Couple d'objets de type Point  
Couple <Point> cp ;
```

- Seule obligation : Le type doit obligatoirement être une classe
- On ne peut pas utilisé les type de variables primaires :
- Par exemple :

```
Couple <int> c ; // erreur : int n'est pas une classe
```

- Mais plutôt utiliser les wrapper :

```
ci = new Couple<Integer> (oi1, oi2) ;
```



# Classe générique à plusieurs types

## 11.2 Couple2

- Une classe générique peut contenir plusieurs types différents
- Par exemple :
  - Reprise de l'exemple précédent en utilisant un couple de 2 objets différents

```
class Couple<T, U>
{
    private T x ; // le premier element du couple
    private U y ; // le second element du couple

    public Couple (T premier, U second){
        x = premier ; y = second ;
    }
    public T getPremier (){ return x ;}

    public void affiche (){
        System.out.println ("premiere valeur : " + x + " -
            deuxieme valeur : " + y) ;
    }
}
```

# Classe générique - Limitations

- On ne peut **pas** instancier un objet d'un type paramétré :

```
class <T> Exple
```

```
...
```

```
x = new T() ;
```

```
// interdit d'instancier un objet de type paramétré T
```

- De même on ne peut pas instancier de **tableaux d'éléments de type paramétrés**
- Pas de création de classe génériques héritant de Throwable, d'Exception ou de Error
- Un champ static ne peut pas être de type générique :

```
class Couple <T>
```

```
{
```

```
    static T compte ; // erreur de compilation
```

```
    ....
```

```
}
```

### 11.3 Méthode générique

- Par exemple : méthode permettant de tirer au hasard un élément d'un tableau fourni en argument et de type quelconque :

```
static <T> T hasard (T [] valeurs)
{
    // choisir au hasard une position i dans le tableau
    // valeurs
    return valeurs[i] ;
}
```

- Utilisation de la méthode, sur un tableau d'éléments de type Integer, puis String :

```
Integer[] tabi = { 1, 5, 8, 4, 9} ;
Integer n = hasard (tabi) ;
```

```
String [] tabs = { "bonjour", "salut", "hello"} ;
String s = hasard (tabs) ;
```

# Méthodes génériques à plusieurs arguments

## 11.4 Méthode générique 2

- Méthode générique à plusieurs arguments :
  - Par exemple :
    - considérons une méthode à deux arguments d'un même type, à savoir une méthode tirant au hasard un objet parmi deux objets de même type *T fournis en arguments*.
  - Elle pourrait se présenter ainsi :

```
public class MethGen2
{
    static <T> T hasard (T e1, T e2)
    {
        double x = Math.random() ;
        if (x < 0.5) return e1 ;
        else return e2 ;
    }
}
```





## Méthodes génériques – Effets de compilation

- Remarque : Le type Double n'est compatible avec aucun autre (sauf lui-même et Number : super classe des numériques)
- Exemples d'appel de la méthode générique :
- Déclarations utilisées :

```
Integer n1 = 2 ; // conversion automatique de int en Integer  
Integer n2 = 5 ; // idem  
Double x1 = 2.5 ; // conversion auto de double en Double
```

- Exemple 1 : Appel accepté

```
hasard (n1, n2) ; // deux arguments du même type Integer
```

# Méthodes génériques – Effets de compilation

- Exemple 2 :

```
Integer n1 = 2;  
Integer n2 = 5;  
Double x1 = 2.5
```

```
hasard (n1, x1) ;  
// accepte bien que les arguments soient de types  
// différents
```

- Pourquoi :

- En fait, la méthode hasard est compilée comme suit :

```
static Object hasard (Object e1, Object e2)  
{  
    double x = Math.random() ;  
    if (x < 0.5) return e1 ;  
    else return e2 ;  
}
```

- Donc compatible avec l'appel effectué

## Méthodes génériques – Effets de compilation

- Si l'on souhaite d'avantage de vérification pour la compilation, il est possible d'imposer le type de retour :

- **nomClasse<Type>.nomMéthode()**

- Par exemple :

```
Integer n1 = 2;
```

```
Integer n2 = 5;
```

```
Double x1 = 2.5
```

```
Hasard.<Double> (n1, x1) ;
```

```
// accepte bien que les arguments soient de types
```

```
// différents
```

- Force le compilateur à vérifier que n1 et x1 sont des types compatibles
- Ce qui provoque une erreur de compilation, puisque Integer et Double ne sont pas compatibles

# Méthodes génériques – Effets de compilation

## ■ Autre exemple :

```
Integer n1 = 2;
```

```
Integer n2 = 5;
```

```
Double x1 = 2.5
```

```
Hasard.<Number> (n1, x1) ;
```

```
// accepte cet appel, les deux types étant compatibles
```

```
// avec Number
```



## **12. Les collections**

# Les collections

java.util.\*

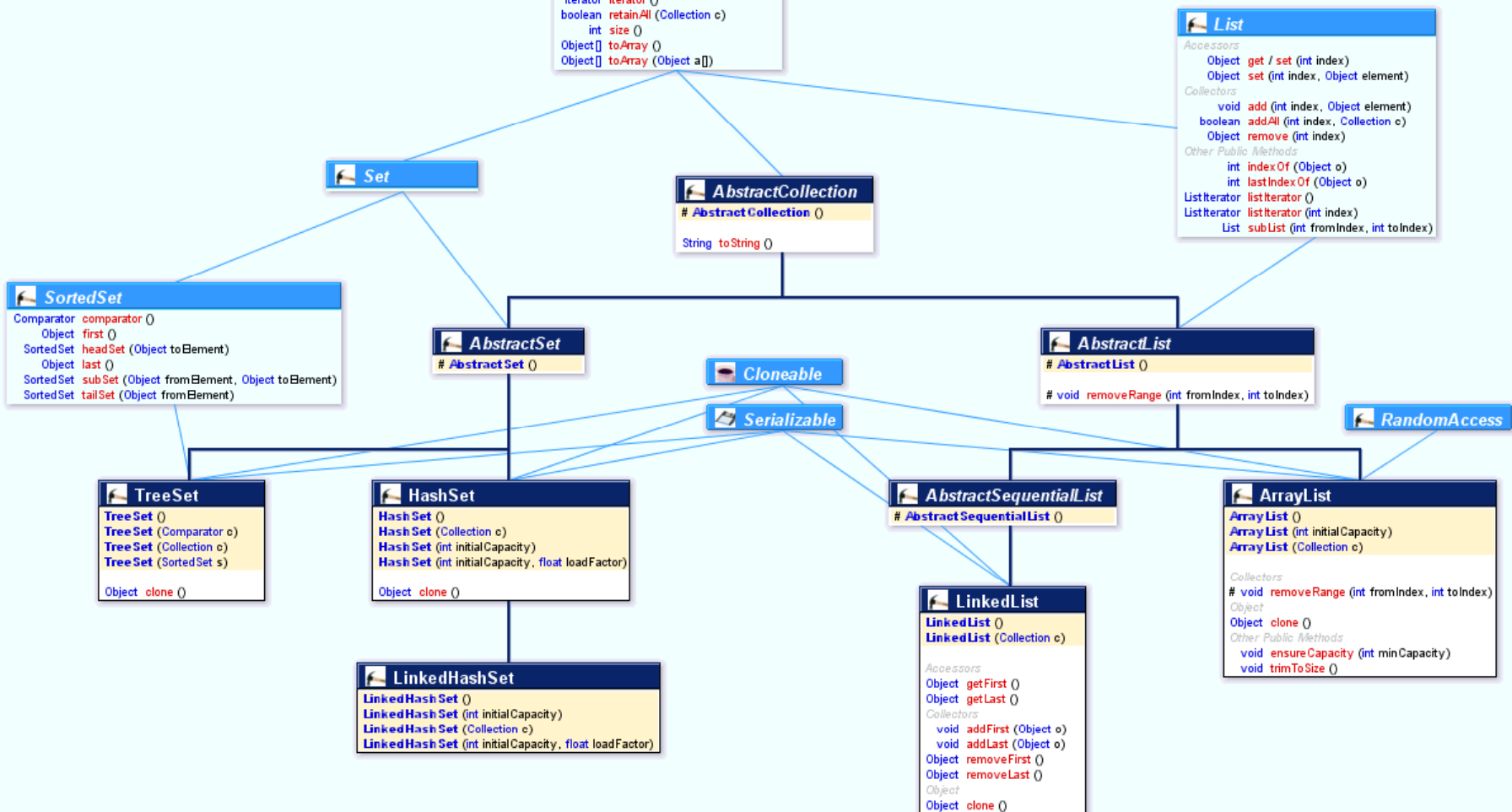
## Collection

Methods declared in Interfaces are hidden in subtypes

See also: [Legacy Collection Diagram](#)



www.falkhausen.de Version 0.9 Copyright 2002-04 by Markius Falkhausen. All rights reserved.





# Les collections

- La version 2 de Java a élargi et harmonisé la bibliothèque de classes utilitaires (**java.util**)
- On y trouve désormais des classes permettant de manipuler les principales structures de données :
  - Les vecteurs dynamiques,
  - Les ensembles,
  - Les listes chaînées,
  - Les queues
  - Les tables associatives.
- Les concepteurs de cette bibliothèque ont cherché à privilégier la simplicité, la concision, l'homogénéité, l'universalité et la flexibilité
- Ainsi que les classes implémentent une même interface (**Collection**) qu'elles complètent de fonctionnalités propres



- Deux concepts distincts :

- **Les Collections :**

- Groupe d'éléments individuels, souvent associé à une règle définissant leur comportement.
  - Par exemple :
  - Une **List doit garder les éléments dans un ordre précis,**
  - **un Set ne peut** contenir de doublons

- **Les Maps :**

- Map : ensemble de paires clef – valeur
- **Une Map** peut renvoyer un **Set de ses clefs, une Collection de ses valeurs, ou un Set de ses paires**
- **Les Maps, comme les tableaux, peuvent facilement être étendus dans de multiples dimensions** sans ajouter de nouveaux concepts :
  - il suffit de créer une **Map dont les valeurs** sont des **Maps**



# Concepts généraux des collections et Map

- Elles sont manipulées par le biais de **classes génériques** implémentant l'interface **Collection<E>**,
  - **E** représentant **le type des éléments** de la collection.
  - Tous les éléments d'une même collection sont donc de **même type E** (ou d'un **type dérivé de E**).
- *Avant JDK 5.0 :*
  - Collection contenant des objets hétérogènes
  - Obligation de cast des éléments
- *A partir de JDK 6.0 :*
  - Collection paramétrée par le type
  - Pas d'éléments hétérogènes
  - Pas d'obligation de cast des éléments

- Objets permettant de parcourir les éléments d'une collection
- De deux ordres :
  - **monodirectionnels** :
    - Le parcours de la collection se fait d'un début vers une fin ; on ne passe qu'une seule fois sur chacun des éléments ;
  - **bidirectionnels** :
    - Le parcours peut se faire dans les deux sens ; on peut avancer et reculer à sa guise dans la collection.



# Itérateurs mono – interface iterator

- Chaque classe collection dispose d'une méthode nommée **iterator** fournissant un itérateur monodirectionnel
- Objet d'une classe implémentant l'**interface Iterator<E>** (Iterator avant le JDK 5.0).
- Associé à une collection donnée, il possède les propriétés suivantes :
  - À un instant donné, un itérateur indique une **position courante** désignant soit un élément donné de la collection, soit la fin de la collection.
  - Le **premier appel** de la **méthode iterator** sur une collection donnée fournit comme position courante, **le début de la collection**.
  - La **méthode next** avance l'itérateur **d'une position**.
  - La méthode **hasNext** de l'itérateur permet de savoir si **l'itérateur est ou non en fin de collection**,



# Itérateurs mono – interface iterator

- Canevas de parcours d'une collection :

```
// depuis JDK 5.02
Iterator<E> iter = c.iterator();
while ( iter.hasNext() )
{
    E o = iter.next() ;
    // utilisation de o
}
```

```
// avant JDK 5.0
Iterator iter = c.iterator();
while ( iter.hasNext() )
{
    Object o = iter.next() ;
    // utilisation de o
}
```

## Itérateurs mono – interface iterator - remove

- **remove** : permet de supprimer le dernier objet renvoyé par **next()**
- Exemple : supprimer d'une collection c tous les éléments vérifiant une condition :

```
// depuis JDK 5.0
Iterator<E> iter = c.iterator() ;
while (c.iter.hasNext())
{
    E = iter.next() ;
    if (condition) iter.remove() ;
}
```

```
// avant JDK 5.0
Iterator iter = c.iterator() ;
while (c.iter.hasNext())
{
    Object o = iter.next() ;
    if (condition) iter.remove() ;
}
```

## Itérateurs mono – interface iterator - remove

- **remove()** ne travaille pas directement avec la position courante de l'itérateur, mais avec la dernière référence renvoyée
- Exemples :

```
Iterator<E> iter ;      // Iterator iter ; <-- avant JDK 5.0
iter = c.iterator();
iter.remove() ;         // incorrect
```

```
Iterator <E> iter ;      // Iterator iter ; <-- avant JDK 5.0
iter = c.iterator() ;
iter.next () ;           // se place après le premier objet
iter.remove() ;          // supprime le premier objet
```





## Itérateurs bi – interface ListIterator

- **ListIterator** : fournit un **itérateur bidirectionnel**
  - Collections parcourables dans les deux sens
  - hérite de **iterator**
- Méthodes supplémentaires :
  - **previous** et **hasPrevious**, complémentaires de **next** et **hasNext**,
  - mais aussi, des méthodes d'insertion d'un élément à la position courante (**add**) ou de modification de l'élément courant (**set**).



## Collections – Structure générale

### Collection

List	implémentée par <i>LinkedList</i> , <i>ArrayList</i> et <i>Vector</i>
Set	implémentée par <i>HashSet</i>
SortedSet	implémentée par <i>TreeSet</i>
NavigableSet	implémentée par <i>TreeSet</i> (Java 6)
Queue (JDK 5.0)	implémentée par <i>LinkedList</i> , <i>PriorityQueue</i>
Deque (Java 6)	implémentée par <i>ArrayDeque</i> , <i>LinkedList</i>



## **Les listes chaînées - LinkedList**



## Listes chaînées - LinkedList

- La classe **LinkedList** permet de manipuler des listes dites "**doublement chaînées**"
  - À chaque élément de la collection, on associe **deux informations supplémentaires** qui sont :
    - les références à l'élément **précédent** et au **suivant**.
  - Parcourue à l'aide d'un **itérateur bidirectionnel** de type **ListIterator**
  - Permet des **ajouts ou des suppressions** à une **position donnée**



## LinkedList – Construction et parcours

- Comme toute collection, une liste peut être construite vide ou à partir d'une autre collection c :

```
/* création d'une liste vide */
LinkedList<E> l1 = new LinkedList<E> () ;
// LinkedList l1=new LinkedList() ;           <-- avant JDK 5.0

/* création d'une liste formée de tous les éléments de la collection c */
LinkedList<E> l2 = new LinkedList<E> (c) ;
// LinkedList l2 = new LinkedList (c) ;       <-- avant JDK 5.0
```

- La classe **LinkedList** dispose des méthodes spécifiques :

- **getFirst()** et **getLast()** fournissant respectivement le premier ou le dernier élément de la liste.



## LinkedList – Ajout d'un élément

- La méthode **add** de **ListIterator** permet d'ajouter un élément à la position courante
- Rappelons que la "**position courante**" utilisée par **add** est toujours définie :
  - si la liste est vide ou si l'on n'a pas agit sur l'itérateur, l'ajout se fera en **début de liste**,
  - si **hasNext** vaut **false**, l'ajout se fera en **fin de liste**.

```
LinkedList <E> ; // LinkedList l ; <-- avant JDK 5.0
.....
ListIterator <E> iter ; // ListIterator iter ; <-- avant JDK 5.0
iter = l.listIterator () ;
/* iter désigne initialement le début de la liste */

/* actions éventuelles sur l'itérateur (next et/ou previous) */

iter.add(elem) ; /* ajoute l'élément elem à la position courante */
```



# LinkedList – Suppression d'un élément

## 12.1 Les LinkedList

- La méthode **remove** de **ListIterator** permet de supprimer un élément à la position courante
- Si la liste est **vide**, elle retourne la valeur **0**





## LinkedList – Exercice

### 12.2 Exercice LinkedListEleve

- Saisir une liste d'élèves dans une LinkedList, et l'afficher en utilisant la classe Eleve, et clavier pour la saisie
- Utiliser un ListIterator pour parcourir la liste



# Les vecteurs dynamiques - classe ArrayList



# Les vecteurs dynamiques - classe ArrayList

- La classe **ArrayList** offre des fonctionnalités d'accès rapide comparables à celles d'un tableau d'objets

- Construction

- Comme toute collection, un vecteur dynamique peut être construit vide ou à partir d'une autre collection c

```
/* vecteur dynamique vide */
ArrayList <E> v1 = new ArrayList <E> () ;
// ArrayList v1 = new ArrayList () ; <-- avant JDK 5.0

/* vecteur dynamique contenant tous les éléments de la
collection c */
ArrayList <E> v2 = new ArrayList <E>(c) ;
// ArrayList v2 = new ArrayList (c) ; <-- avant JDK 5.0
```

# Les vecteurs dynamiques - classe ArrayList

- Ajout d'un élément :
  - **add (elem)** qui se contente d'ajouter l'élément elem en fin de vecteur
  - **add (i,elem)** ajout d'un élément elem en un rang i, tous ses suivants étant décalés d'une position
- Suppression d'un élément
  - Méthode spécifique **remove** permettant de supprimer un élément de rang donné
  - Elle fournit en retour l'élément supprimé.
  - Les éléments suivants doivent être décalés d'une position.

```
ArrayList <E> v ;  
// ArrayList v ; <-- avant JDK 5.0  
.....  
/* suppression du troisième élément de v qu'on obtient dans o  
*/  
E o = v.remove(3) ;  
// Object o = v.remove (3) ; <-- avant JDK 5.0  
v.removeRange (3, 8) ;  
// supprime les éléments de rang 3 à 8 de v
```

# Les vecteurs dynamiques - classe ArrayList

## 12.3 Les ArrayList

- On peut connaître la valeur d'un élément de rang **i** par **get(i)**.

```
// depuis le JDK 5.0
for (E e : v)
{
    // utilisation de e
}

// avant JDK 5.0
for (int i=0 ; i<v.size() ; i++)
{
    // utilisation de v.get(i)
}
```

- On peut remplacer par **elem** la valeur de l'élément de **rang i** par **set(i, elem)**

- Exemple : comment remplacer par la référence null, tout élément d'un vecteur dynamique **v** vérifiant une condition :

```
for (int i = 0 ; i < v.size() ; i++)
    if (condition) set(i, null) ;
```



# Les vecteurs dynamiques - classe ArrayList

## 12.4 Exercice ArrayListEleve

- Reprendre le programme LinkedListEleve :
  - Le renommer en ArrayListEleve
  - Implémenter un ArrayList à la place de la LinkedList
  - On peut utiliser la classe Iterator pour son parcours



## **Les ensembles – HashSet et TreeSet**



- Deux classes implémentent la notion d'ensemble :
  - **HashSet** et **TreeSet**
  - Un ensemble est une **collection non ordonnée** d'éléments,
  - Aucun élément **ne pouvant apparaître plusieurs fois** dans un même ensemble.
  - **HashSet** qui recourt à une technique dite de hachage
  - **TreeSet** qui utilise un arbre binaire pour ordonner complètement les éléments

- Construction et parcours :

- Comme toute collection, un ensemble peut être construit vide ou à partir d'une autre collection :

```
// ensemble vide
HashSet<E> e1 = new HashSet<E> () ;
// HashSet e1 = new HashSet() ; <-- avant JDK 5.0

// ensemble contenant tous les éléments de la collection c
HashSet<E> e2 = new HashSet<E>(c) ;
// HashSet e2 = new HashSet(c) ; <-- avant JDK 5.0

// ensemble vide
TreeSet<E> e3 = new TreeSet<E>() ;
// TreeSet e3 = new TreeSet() ; <-- avant JDK 5.0

// ensemble contenant tous les éléments de la collection c
TreeSet<E> e4 = new TreeSet<E>(c) ;
// TreeSet e4 = new TreeSet(c) ; <-- avant JDK 5.0
```

- Les deux classes **HashSet** et **TreeSet** disposent de la méthode **iterator()** prévue dans l'interface **Collection**
- Elle fournit un **itérateur monodirectionnel** (**Iterator**) permettant de parcourir les différents éléments de la collection :

```
HashSet<E> e ; // ou TreeSet<E> e
// HashSet e ; ou TreeSet e ; <-- avant JDK 5.0
.....
Iterator<E> it = e.iterator() ;
// Iterator it = e.iterator() ; <-- avant JDK 5.0

while (it.hasNext())
{
    E o = it.next() ;
    // Object o = it.next() ; <-- avant JDK 5.0
    // utilisation de o
}
```

- Ajout d'un élément :

- Impossible d'ajouter un élément à une position donnée (les ensembles ne disposent **pas d'un itérateur bidirectionnel**)

- La seule façon d'ajouter un élément à un ensemble : **add()**

- Elle s'assure en effet que l'élément n'existe pas déjà :

```
HashSet<E> e ; E elem ;  
// HashSet e ; Object elem ; <-- avant JDK 5.0  
.....  
  
boolean existe = e.add (elem) ;  
if (existe) System.out.println (elem + " existe deja") ;  
else System.out.println (elem + " a ete ajoute") ;
```

- Suppression d'un élément :

- **remove ()** :

- renvoie true si l'élément a été trouvé (et donc supprimé)
    - renvoie false dans le cas contraire

- **Exemple :**

```
TreeSet<E> e ; E o ;  
// TreeSet e ; Object o ; <-- avant JDK 5.0  
.....  
boolean trouve = e.remove (o) ;  
if (trouve) System.out.println (o + " a ete supprime") ;  
else System.out.println (o + " n'existe pas ") ;
```

- La méthode **remove** de l'itérateur monodirectionnel permet de supprimer l'élément courant :

```
TreeSet<E> e ;  
// TreeSet e ; <-- avant JDK 5.0  
.....  
Iterator<E> it = e.iterator () ;  
// Iterator it = e.iterator() ; <-- avant JDK 5.0  
  
it.next () ; it.next () ;  
/* deuxième élément = élément courant */  
it.remove () ;  
/* supprime le deuxième élément */
```

- Enfin, la méthode **contains** permet de tester l'existence d'un élément

## ■ Opérations ensemblistes :

■ Les méthodes **removeAll**, **addAll** et **retainAll**, applicables à toutes les collections, vont prendre un intérêt tout particulier avec les ensembles

■ Exemple : *si e1 et e2 sont deux ensembles :*

### ■ **e1.addAll(e2) :**

- Place dans e1 tous les éléments présents dans e2.
- Après exécution, la réunion de e1 et de e2 se trouve dans e1

### ■ **e1.retainAll (e2) :**

- Garde dans e1 ce qui appartient à e2.
- Après exécution, on obtient l'intersection de e1 et de e2 dans e1

### ■ **e1.removeAll (e2) :**

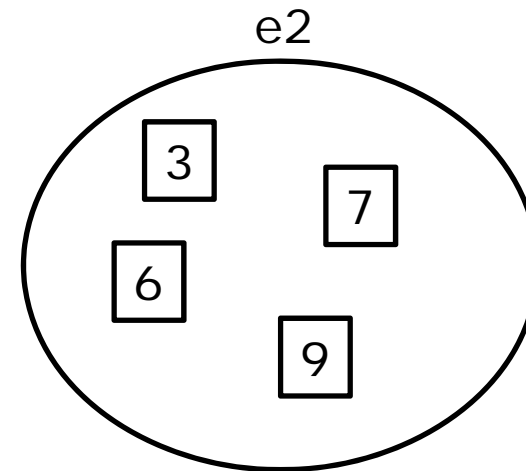
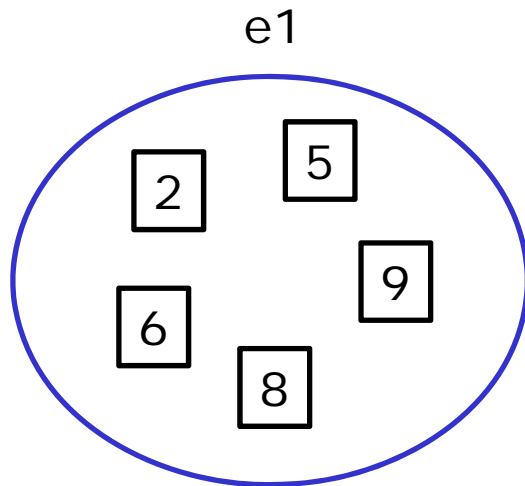
- Supprime de e1 tout ce qui appartient à e2.
- Après exécution, on obtient le "complémentaire de e2 par rapport à e1" dans e1



# Ensembles - HashSet

## 12.5 Les ensembles - HashSet

```
int t1[] = {2, 5, 6, 8, 9} ;  
int t2[] = { 3, 6, 7, 9} ;  
HashSet <Integer> e1 = new HashSet <Integer>(), e2 = new  
HashSet<Integer> () ;  
  
for (int v : t1) e1.add (v) ;  
for (int v : t2) e2.add (v) ;  
System.out.println ("e1 = " + e1) ;  
System.out.println ("e1 = " + e2) ;
```

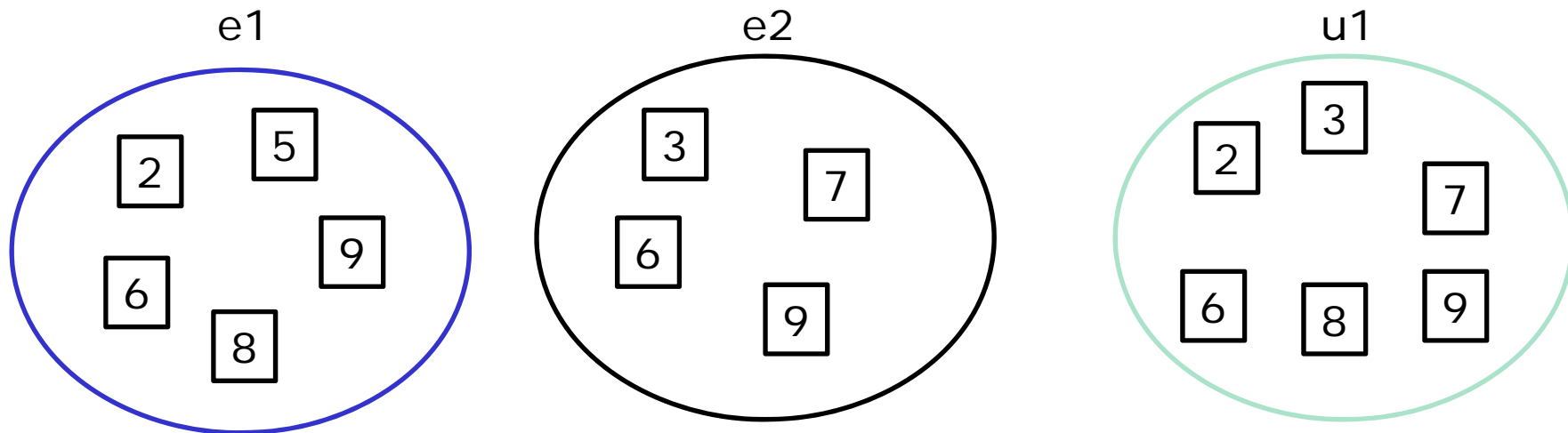


# Ensembles - HashSet

## 12.5 Les ensembles - HashSet

```
// reunion de e1 et e2 dans u1
HashSet <Integer> u1 = new HashSet <Integer> () ;

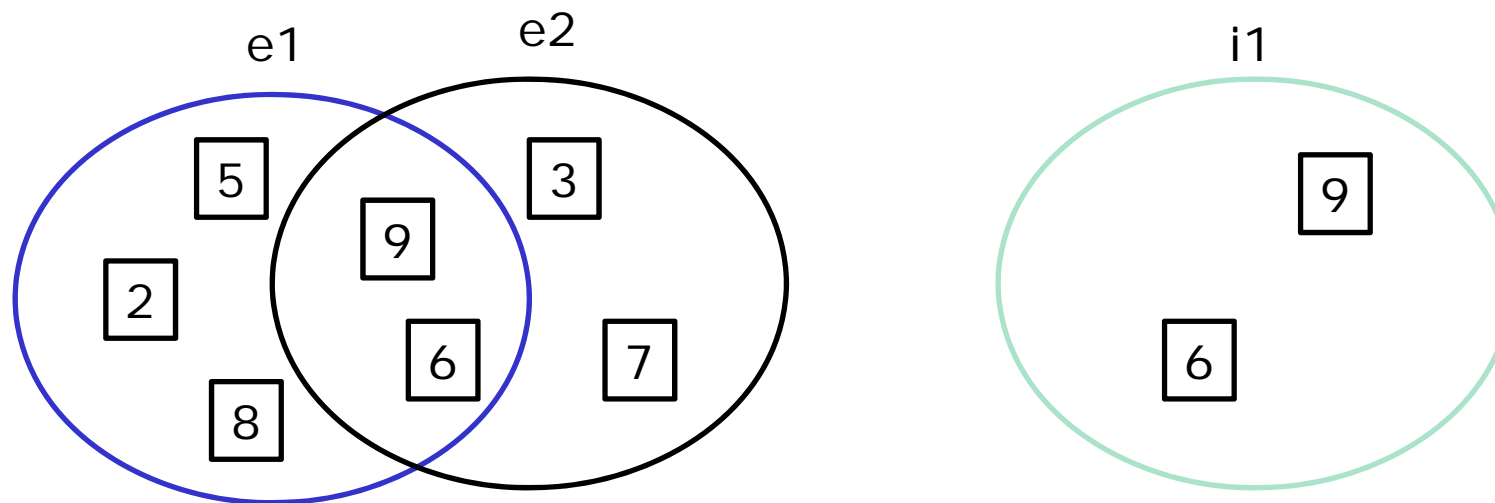
copie (u1, e1) ; // copie e1 dans u1
u1.addAll (e2) ;
System.out.println ("u1 = " + u1) ;
```



# Ensembles - HashSet

## 12.5 Les ensembles - HashSet

```
// intersection de e1 et e2 dans i1  
HashSet <Integer> i1 = new HashSet <Integer> () ;  
copie (i1, e1) ;  
i1.retainAll (e2) ;  
  
System.out.println ("i1 = " + i1) ;
```



- Utilise une table de hachage :
  - méthode de **hashCode()** = **fonction de hachage**
  - Permet d'associer un entier à la valeur d'un élément permettant l'optimisation de la structuration des informations
  - Possibilité de définir ses propres règles :
    - La méthode **equals** :
      - Elle définit l'appartenance d'un élément à l'ensemble
    - La méthode **hashCode** :
      - Pour ordonnancer les éléments d'un ensemble

# Ensembles – HashSet – Table de hachage

## 12.6 Les ensembles – HashSet hashCode()

```
class Point {
    private int x, y;

    Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public int hashCode() {
        // Ici la table de hachage determine
        // que le code de hachage de l'élément est la somme
        // de ses coordonnées
        return x + y;
    }
    public boolean equals(Object pp) {
        // Méthode déterminant l'égalité de 2 éléments
        // ils sont égaux si leurs coordonnées sont égales
        Point p = (Point) pp;
        return ((this.x == p.x) & (this.y == p.y));
    }
    public void affiche() {
        System.out.print("[ " + x + " " + y + " ] ");
    }
}
```

# Ensembles – HashSet – Table de hachage

## 12.6 Les ensembles – HashSet hashCode()

```
import java.util.*;

public class EnsPt1 {
    public static void main(String args[]) {
        Point p1 = new Point(1, 3), p2 = new Point(2, 2);
        Point p3 = new Point(4, 5), p4 = new Point(1, 8);
        Point p[] = { p1, p2, p1, p3, p4, p3 };

        HashSet<Point> ens = new HashSet<Point>();
        for (Point px : p)
        {
            System.out.print("le point ");
            px.affiche();
            boolean ajoute = ens.add(px);
            if (ajoute) System.out.println(" a ete ajoute");
            Else System.out.println("est deja present");
            System.out.print("ensemble = ");
            affiche(ens);
        }
    }
}
```

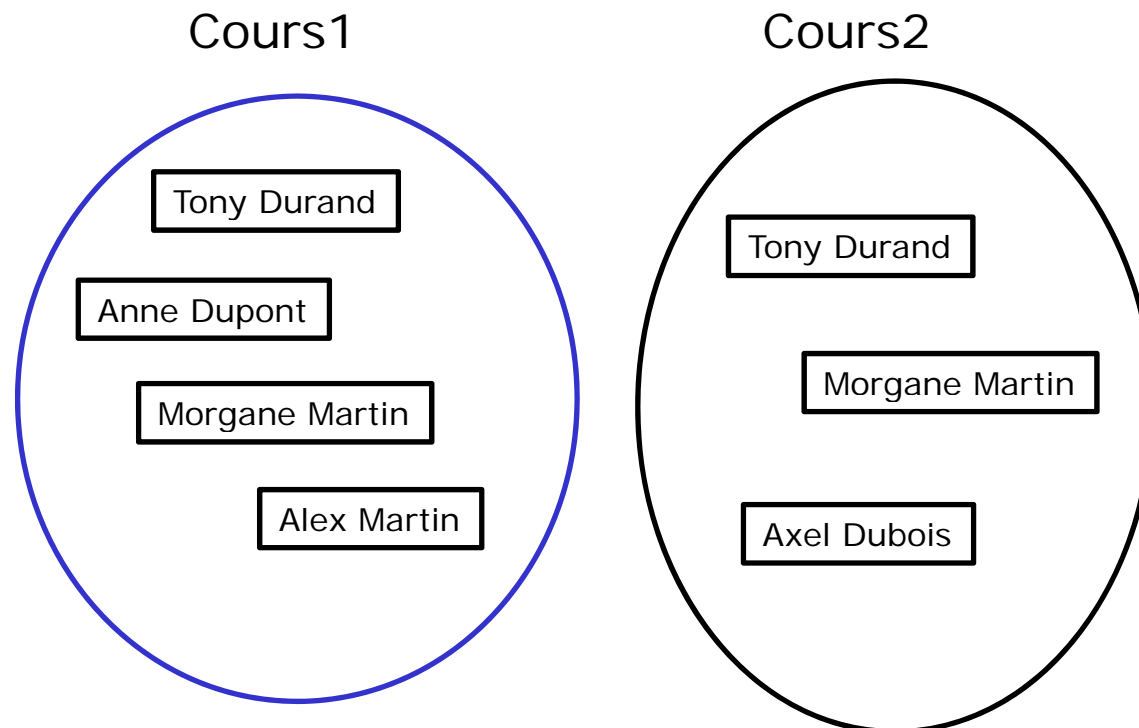
...

*On affichera : le point [1 3] est deja present du fait de la redéfinition de la méthode equals*

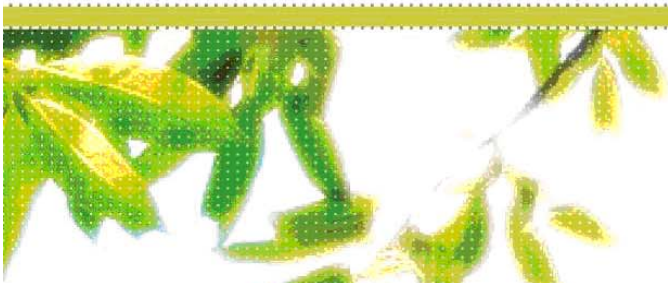
# Ensembles - HashSet

## 12.7 Exercice HashSetEleve

- Reprendre le programme des élèves (LinkedList ou ArrayList)
  - Ajouter un id à chaque élève (pour le hachage)
  - Créer deux Hashset cours1 et cours2
  - Faire l'union dans un 3° Hashset (tous les élèves)
  - Faire l'intersection dans un 4° HashSet (élèves participants à tous les cours)
  - Faire la différence dans un 5° HashSet (élèves ne participant qu'au cours 1)







## Ensembles - TreeSet

### 12.8 Les ensembles – TreeSet

- Utilise un **arbre binaire** pour ordonnancer les éléments de l'ensemble
- Redéfinition de la méthode **compareTo** pour définir **les règles d'égalité**
  - La recherche dans cet arbre d'un élément est généralement moins rapide que dans une table de hachage mais plus rapide qu'une recherche séquentielle
- On n'utilise pas les méthodes hashCode et equals
- La classe TreeSet dispose de deux méthodes spécifiques **first()** et **last()** (premier et le dernier élément de l'ensemble)

### 12.9 Les ensembles – TreeSet Comparable

- La redéfinition de la méthode **compareTo** pour définir **les règles d'égalité** implique l'implémentation de l'interface **Comparable** pour les objets constituant l'arbre



## Ensembles - HashSet

### 12.10 Exercice TreeSetEleve

- Reprendre le projet HashSetEleve, pour implémenter un TreeSet



## **Les queues - L'interface Queue**



## Les queues - L'interface Queue JDK 5.0

- **Nouvelle interface** destinée à la gestion des files d'attente (ou queues).
- Il s'agit de structures dans lesquelles on peut :
  - introduire un nouvel élément, si la queue n'est pas pleine,
  - prélever le premier élément de la queue,
  - **add()** : permet d'ajouter un nouvel élément
    - ne déclenche pas d'exception quand la queue est pleine ; renvoie simplement la valeur false.
- Le prélèvement du premier élément de la queue peut se faire :
  - de façon **destructive**, à l'aide de la méthode **poll** : l'élément ainsi prélevé est supprimé de la queue ; la méthode renvoie null si la queue est vide,
  - de façon **non destructive** à l'aide de la méthode **peek**.



# Les queues - L'interface Queue JDK 5.0

- Classes implémentant **l'interface Queue** :

- **LinkedList**, modifiée par le Java 5, pour y intégrer les nouvelles méthodes.

- **PriorityQueue**, introduite par Java 5, permet de choisir une relation d'ordre ;



- dans ce cas, le type des éléments doit implémenter l'interface **Comparable** ou être doté d'un comparateur approprié.

- Les éléments de la queue sont alors ordonnés par cette relation d'ordre

- Le prélèvement d'un élément porte alors sur le "premier" au sens de cette relation (on parle du "plus prioritaire", d'où le nom de PriorityQueue).



## Les queues à double entrée Deque (Java 6)

- **Java 6** a introduit une nouvelle interface **Deque**, dérivée de **Queue**, destinée à gérer des **files d'attente à double entrée**,
  - C'est-à-dire dans lesquelles on peut réaliser l'une des opérations suivantes à l'une des extrémités de la queue :
    - ajouter un élément,
    - examiner un élément,
    - supprimer un élément.



# Les queues à double entrée Deque (Java 6)

- Pour chacune des ces **6 possibilités (3 actions, 2 extrémités)**, il existe deux méthodes :
  - l'une déclenchant une exception quand l'opération échoue (pile pleine ou vide, selon le cas),
  - l'autre renvoyant une valeur particulière (null pour une méthode de prélèvement ou d'examen
  - false pour une méthode d'ajout).
- Liste de ces méthodes (First correspondant aux actions sur la tête, Last aux actions sur la queue et e désignant un élément)

	Exception	Valeur spéciale
Ajout	<code>addFirst (e)</code> <code>addLast (e)</code>	<code>offerFirst ()</code> <code>offerLast ()</code>
Examen	<code>getFirst ()</code> <code>getLast()</code>	<code>peekFirst ()</code> <code>peekLast ()</code>
Suppression	<code>removeFirst ()</code> <code>removeLast ()</code>	<code>pollFirst ()</code> <code>pollLast ()</code>



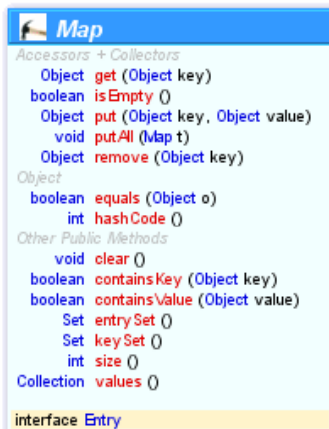
## **13. Les tables associatives HashMap et TreeMap**

# Les Tables - HashMap et TreeMap

java.util.\*

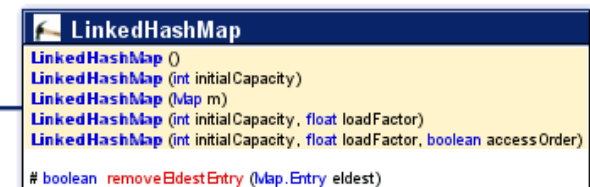
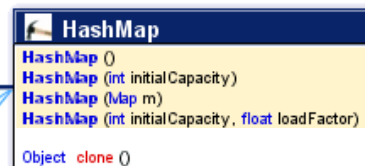
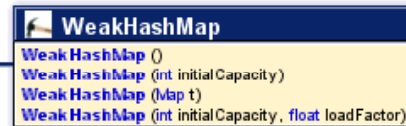
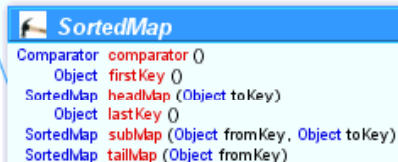
## Map

Methods declared in Interfaces are hidden in subtypes



**Cloneable**

**Serializable**





## Les Tables - HashMap et TreeMap

- Une table associative permet de conserver une information associant deux parties nommées **clé et valeur**
- *Elle est principalement destinée à retrouver la valeur associée à une clé donnée*
- Les exemples les plus caractéristiques de telles tables sont :
  - **le dictionnaire** : à un mot (clé), on associe une valeur qui est sa définition,
  - **l'annuaire usuel** : à un nom (clé), on associe une valeur comportant le numéro de téléphone et, éventuellement, une adresse
  - **l'annuaire inversé** : à un numéro de téléphone (qui devient la clé), on associe une valeur comportant le nom et, éventuellement, une adresse



# Les Tables - Implémentation

- **Deux types** d'organisation rencontrés pour les ensembles :
  - table de hachage : classe ***HashMap***
  - arbre binaire : classe ***TreeMap***
- Dans les deux cas, seule la clé sera utilisée pour ordonnancer les informations
- **HashMap** :
  - on se sert du **code de hachage** des objets formant les clés
- **TreeMap** :
  - on se sert de la **relation d'ordre** induite par *compareTo* ou par un comparateur fixé à la construction
- Elles implémentent **l'interface Map**

## Les Tables – Ajout d'information

- Ajouter d'une **clé/valeur** à une table : méthode **put()**
- Exemple :
  - *Si K désigne le type des clés et V celui des valeurs :*

```
// creation d'une table vide
HashMap <K, V> m = new HashMap <K, V> () ;
.....
HashMap <String, Integer> m = new HashMap <String,
Integer> () ;

// ajoute a m, un element de cle "m" et de valeur 3
m.put ("m", 3) ;
```

- Si la **clé existe déjà**, la valeur associée remplacera l'ancienne (une clé donnée ne pouvant figurer qu'une seule fois dans une table)
- **put** fournit en retour soit **l'ancienne valeur** si la clé existait déjà, soit **null**
- Les **clés** et les **valeurs** doivent être des **objets**

## Les Tables – Recherche d'information

- On obtient la valeur associée à une clé donnée à l'aide de la méthode ***get()*** , laquelle fournit **null**
- si la clé cherchée n'est pas présente (*K étant le type de la clé*) :

```
K o = get ("x") ; //fournit la valeur associee a la cle "x"  
if (o == null)  
    System.out.println("Aucune valeur associee a la cle x");
```

- La méthode ***containsKey()*** permet de savoir si une clé donnée est présente



## Les Tables – Suppression d'information

- On peut supprimer un élément d'une table en utilisant la méthode `remove()`
- *Elle fournit en retour l'ancienne valeur associée si la clé existe ou la valeur `null` dans le cas contraire :*

```
K cle = "x" ;
K val = remove (cle) ; // supprime l'element (cle + valeur)
de cle "x"

if (val != null)
    System.out.println ("On a supprime l'element de cle " +
        cle + " et de valeur" + val) ;
else
    System.out.println ("la cle " + cle + " n'existe pas") ;
```



## Les Tables – Parcours , notion de vue

- En théorie, les types **HashMap** et **TreeMap** ne disposent pas d'itérateurs.
- La méthode nommée **entrySet()**, "voir" une table comme un ensemble de "paires"
- Une paire n'étant rien d'autre qu'un élément de type **Map.Entry** réunissant deux objets (de types a priori quelconques).
- Les méthodes **getKey()** et **getValue()** du type **Map.Entry** permettent d'extraire respectivement la clé et la valeur d'une paire

## Les Tables – Parcours , notion de vue

- Canevas de parcours d'une table :

```
HashMap <K, V> m ;
.....
// entrees est un ensemble de "paires"
Set <Map.entry<K, V> > entrees = m.entrySet () ;

// iterateur sur les paires
Iterator <Map.entry<K, V> > iter = entrees.iterator() ;

// boucle sur les paires
while (iter.hasNext()) {
    // paire courante
    Map.Entry <K, V> entree = (Map.Entry)iter.next() ;
    // cle de la paire courante
    K cle = entree.getKey () ;
    // valeur de la paire courante
    V valeur = entree.getValue() ;
    .....
}
```



# Les Tables – HashMap et TreeMap

## 13.1 Les tables – HashMap

## 13.2 Les tables – TreeMap

- Autres vues associées à une table :

- Ensemble des clés : méthode **keySet()** :

```
HashMap m ;
```

```
.....
```

```
Set cles = m.keySet ( ) ;
```

- On peut parcourir classiquement cet ensemble à l'aide d'un itérateur

- La suppression d'une clé (clé courante ou clé de valeur donnée) entraîne la suppression de l'élément correspondant de la table *m*

- Collection des valeurs à l'aide de la méthode **values()** :

```
Collection valeurs = m.values ( ) ;
```



# **15. Les annotations**



# Les annotations

- Java SE 5 a introduit les annotations qui sont des métas données incluses dans le code source
- Les annotations ont été spécifiées dans la JSR 175
- But est d'intégrer au langage Java des métas données
- Des métas données étaient déjà mises en oeuvre avec Javadoc ou exploitées par des outils tiers notamment XDoclet (outil open source)
- Javadoc utilisait des métas données en standard pour générer une documentation automatique du code source
- Javadoc propose l'annotation `@deprecated` qui bien qu'utilisé dans les commentaires permet de marquer une méthode comme obsolète et faire afficher un avertissement par le compilateur



## Les annotations - Présentation

- Elles apportent une standardisation des métas données dans un but généraliste
- Ces métas données associés aux entités Java peuvent être exploitées à la compilation ou à l'exécution.
- Elles peuvent être utilisées avec quasiment tous les types d'entités et de membres de Java : packages, classes, interfaces, constructeurs, méthodes, champs, paramètres, variables ou annotations elles même
- Java propose plusieurs annotations standards et permet la création de ses propres annotations.
- Une annotation s'utilise en la faisant précéder de l'entité qu'elle concerne par le caractère @.





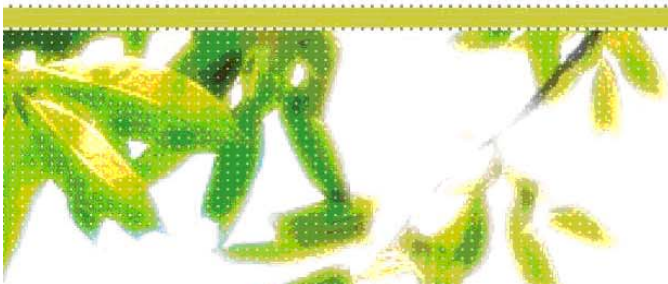
# Les annotations - Catégories

- Il existe plusieurs catégories d'annotations :
  - les marqueurs (markers) : (exemple : `@Deprecated`, `@Override`, ...)
  - les annotations paramétrées (exemple : `@MonAnnotation("test")` )
  - les annotations multi paramétrées : exemple : `@MonAnnotation(arg1="test 3", arg2="test 2", arg3="test3")`
- Les arguments fournis en paramètres d'une annotation peuvent être de plusieurs types :
  - les chaînes de caractères,
  - les types primitifs,
  - les énumérations,
  - les annotations,
  - le type Class
- Les annotations sont définies dans un type d'annotation



## Les annotations – Mise en oeuvre

- Par convention, les annotations s'utilisent sur une ligne dédiée
- Leur utilisation concernent plusieurs fonctionnalités :
  - Utilisation par le compilateur pour détecter des erreurs ou ignorer des avertissements
  - Documentation
  - Génération de code
  - Génération de fichiers



# L' annotation @Deprecated

## 14.1 Annotation Deprecated

- Marqueur qui précise que l'entité concernée est obsolète et qu'il ne faudrait plus l'utiliser
- Elle peut être utilisée avec une classe, une interface ou un membre (méthode ou champ)
- Les entités marquées avec l'annotation @Deprecated devrait être documentée avec le tag @deprecated de Javadoc en lui fournissant la raison de l'obsolescence et éventuellement l'entité de substitution



# L' annotation @Override

## 14.2 Annotation Override

- Marqueur utilisé par le compilateur pour vérifier la réécriture de méthode héritée.
- **@Override** s'utilise pour annoter une méthode qui est une réécriture d'une méthode héritée
- Le compilateur lève une erreur si aucune méthode héritée ne correspond.



# L' annotation @SuppressWarnings

## 14.3 Annotation SuppressWarning

- Elle permet de demander au compilateur d'inhiber certains avertissements qui sont pris en compte par défaut
- La liste des avertissements utilisables dépend du compilateur
- Un avertissement utilisé dans l'annotation non reconnu par le compilateur ne provoque pas d'erreur mais éventuellement un avertissement
- Le compilateur fourni avec le JDK supporte les avertissements suivants :

Nom	Rôle
deprecation	Vérification de l'utilisation d'entités déclarées deprecated
unchecked	Vérification de l'utilisation des generics
fallthrough	Vérification de l'utilisation de l'instruction break dans les cases des instructions switch
path	Vérification des chemins fournis en paramètre du compilateur
serial	Vérification de la définition de la variable serialVersionUID dans les beans
finally	Vérification de la non présence d'une instruction return dans une clause finally



## Annotations communes

- Les annotations communes sont définies par la JSR 250 et sont intégrées dans Java 6
- But est de définir des annotations couramment utilisées pour éviter leur redéfinition pour chaque outil qui en aurait besoin
- Les annotations définies concernent :
  - La plate-forme standard dans le package `javax.annotation` (`@Generated`, `@PostConstruct`, `@PreDestroy`, `@Resource`, `@Resources`)
  - La plate-forme entreprise dans le package `javax.annotation.security` (`@DeclareRoles`, `@DenyAll`, `@PermitAll`, `@RolesAllowed`, `@RunAs`)



# L'annotation @Generated

- De plus en plus d'outils ou de frameworks génèrent du code source pour faciliter la tâche des développeurs
- Le code ainsi généré peut être marqué avec l'annotation @Generated

- Exemple :

```
@Generated(  
    value = "entite.qui.a.genere.le.code",  
    comments = "commentaires",  
    date = "12 April 2008"  
)  
  
public void toolGeneratedCode(){  
}
```

- L'attribut obligatoire value permet de préciser l'outil à l'origine de la génération
- Les attributs facultatifs comments et date permettent respectivement de fournir un commentaire et la date de génération.





# Les annotations @Resource et @Resources

- L'annotation **@Resource** définit une ressource requise par une classe
- Elles possèdent plusieurs attributs :

Attribut	Description
authenticationType	Type d'authentification pour utiliser la ressource (Resource.AuthenticationType.CONTAINER ou Resource.AuthenticationType.APPLICATION)
description	Description de la ressource
mappedName	Nom de la ressource spécifique au serveur utilisé (non portable)
name	Nom JNDI de la ressource
shareable	Booléen qui précise si la ressource est partagée
type	Le type pleinement qualifié de la ressource



## Les annotations **@PostConstruct** et **@PreDestroy**

- Les annotations **@PostConstruct** et **@PreDestroy** permettent désigner des méthodes qui seront exécutées après l'instanciation d'un objet et avant la destruction d'un objet
- Ces deux annotations ne peuvent être utilisées que sur des méthodes.
- Ces annotations sont par exemple utiles dans **Java EE** et en relation avec les **EJB**



## **15. Utilisation des dates**

- Pas toujours simple à mettre en œuvre :
- Il existe plusieurs calendriers, le plus utilisé est le calendrier Grégorien (début à la naissance de Jésus Christ)
- Il comporte de nombreuses imperfections :
  - Le nombre de jours d'un mois varie selon le mois et l'année ...
  - Le format textuel des dates diffère selon la Locale utilisée
  - L'existence des fuseaux horaires (date/heure différente selon la localisation géographique)
  - La possibilité de prendre en compte un décalage horaire lié à l'heure d'été et à l'heure d'hiver
  - ...

- Calcul du temps avec Java :
  - Nombre de millisecondes écoulées depuis un point d'origine défini (1er janvier 1970)
  - Ceci permet de définir un point dans le temps de façon unique.
- L'utilisation de dates en Java est de surcroît plus compliquée à cause de l'API historique qui permet leur gestion car elle n'est pas toujours intuitive



# Classes standards pour manipuler les dates

## 15.1 Fabrique de dates

■ La gestion et des traitements sur les dates sont réparties sur plusieurs classes :

■ **java.util.Date** : elle encapsule un point dans le temps

■ **java.util.Calendar** et **java.util.GregorianCalendar** : elle permet la manipulation d'une date

■ **java.util.TimeZone** et **java.util.SimpleTimeZone** : elle encapsule un fuseau horaire à partir du méridien de Greenwich (GMT) et les informations relatives aux décalages concernant l'heure d'été et l'heure d'hiver

■ **java.text.DateFormat**, **java.text.SimpleDateFormat** : elle permet de convertir une date en chaîne de caractères et vice versa

■ **java.text.DateFormatSymbols** : elle permet de traduire les différents éléments d'une date (jour, mois, ...)

- Utilise une variable de type **long** : représente par le nombre de millisecondes écoulées entre le 1 janvier 1970 à minuit heure GMT et l'instant concerné
- Depuis la version 1.1, toutes les méthodes permettant de manipuler la date sont ***deprecated***.
- Par défaut, le temps est obtenu en utilisant la méthode **System.currentTimeMillis()** (précision dépendante du système d'exploitation)



- Permet une représentation et une manipulation dans un **calendrier** et un **fuseau horaire**
- Nécessaire d'**initialiser le point dans le temps**, avant de pouvoir **utiliser** l'instance de **Calendar**
- Une nouvelle instance de la classe est toujours initialisée avec le **point dans le temps courant**
- Pour encapsuler un autre point, il faut obligatoirement après l'instanciation **utiliser une des méthodes** de la classe pour **modifier** le point encapsulé

- Pour accéder aux propriétés de la date une seule méthode **get()**
  - Utilise en paramètre le nom de la propriété souhaitée et qui retourne toujours une **valeur de type int**
- La classe Calendar définit des **constantes de type int** pour le **nom de ces propriétés** :
  - La classe Calendar définit aussi plusieurs constantes qui contiennent les valeurs possibles pour certaines propriétés.
  - Leur utilisation est fortement recommandée
- Par exemple :
  - La valeur d'un mois varie de 0 à 11 correspondants aux constantes **Calendar.JANUARY** à **Calendar.DECEMBER**
  - Calendar définit aussi la constante **UNDECIMBER** qui représente le treizième mois de l'année requis par certains calendriers

- **Trois façons** de manipuler la date :
  - **set()** : permet de modifier un élément de la date
  - **add()** : permet de modifier un élément de la date en tenant compte des impacts sur les autres éléments qui composent la date
  - **roll()** : identique à la méthode add() mais sans affecter les autres éléments de la date
- La date encapsulée dans **Calendar** peut être manipulée de deux manières :
  - Directement par un calcul sur le nombre de millisecondes écoulées depuis le 1er janvier 1970
  - En agissant sur les éléments qui composent la date en utilisant les méthodes dédiées



## Java.util.GregorianCalendar

- La classe **java.util.GregorianCalendar** est la seule implémentation concrète de la **classe Calendar** fournie en standard
- Cette implémentation correspond au calendrier Grégorien
- La **méthode isLeapYear()** permet de savoir si l'année encapsulée par la classe est bissextile

# java.util.TimeZone et java.util.SimpleTimeZone

## 15.3 TimeZone

- **TimeZone** est utilisée par **Calendar** pour déterminer la date suivant le fuseau horaire
- Les fuseaux horaires ont 2 décalages :
  - Par rapport au méridien de Greenwich (GMT)
  - En regard des heures été ou hivers (DST : daylight savings time)
  - La classe **TimeZone** encapsule un nom long et un nom court qui permet d'identifier le fuseau horaire
- La méthode **String[] getAvailableIDs()** permet d'obtenir les noms des **TimeZones** définis en standard
- Java 6 : 597 **TimeZones** fournis
- La classe permet d'obtenir une instance de **TimeZone** à partir de son identifiant en utilisant la méthode **getTimeZone()**
- ou celle correspondant à la **Locale** courante en utilisant la méthode **getDefault()**



# java.util.SimpleDateFormat

## 15.4 SimpleDateFormat

- Pour manipuler les formats, cette classe utilise un modèle (pattern) sous la forme d'une chaîne de caractères
- Plusieurs méthodes pour obtenir le modèle par défaut de la Locale courante :
  - **getTimeInstance(style)**
  - **getDateInstance(style)**
  - **getDateTimeInstance(styleDate, styleHeure)**
- Pour chacune de ces méthodes, quatre styles sont utilisables :
  - SHORT,
  - MEDIUM,
  - LONG
  - et FULL
- Ils permettent de désigner la richesse des informations contenues dans le modèle pour la date et/ou l'heure



## **16. Gestion des flux**



- Flux = système de communication Java implémenté dans **java.io** (**input/output** ou **Entrées/sorties**)
  - Différents types de flux :
    - Les **flux d'octets** pour les entiers et type de données simples
    - Les **flux de caractères** pour des fichiers texte et autres ressources textuelles
- On peut traiter toutes les données quelque soit leur origine (internet, cdrom, disque dur ...)
- Un flux d'entrée envoie des données de la **source vers le programme**
- Un flux de sortie envoie des données d'un **programme jusqu'à sa destination**



# Flux d'octets et de caractères

## ■ Flux d'octets :

- Véhiculent des valeurs comprises entre 0 et 255
- Permettent d'exprimer :
  - données numériques
  - programmes exécutables
  - pseudo-code
  - etc...

## ■ Flux de caractères :

- Véhiculent les données de type texte (fichier texte, page web ...)

## ■ Flux d'entrée :

1°) création d'un objet **FileInputStream** associé à la source de données

2°) lecture des informations du fichier – **read()** qui retourne un octet lu à partir du fichier

3°) fermeture du fichier – **close()**

## ■ Flux de sortie

1°) création d'un objet **FileOutputStream** associé à la source de données

2°) écriture d'un octet dans le fichier – **write()**

3°) fermeture du fichier – **close()**

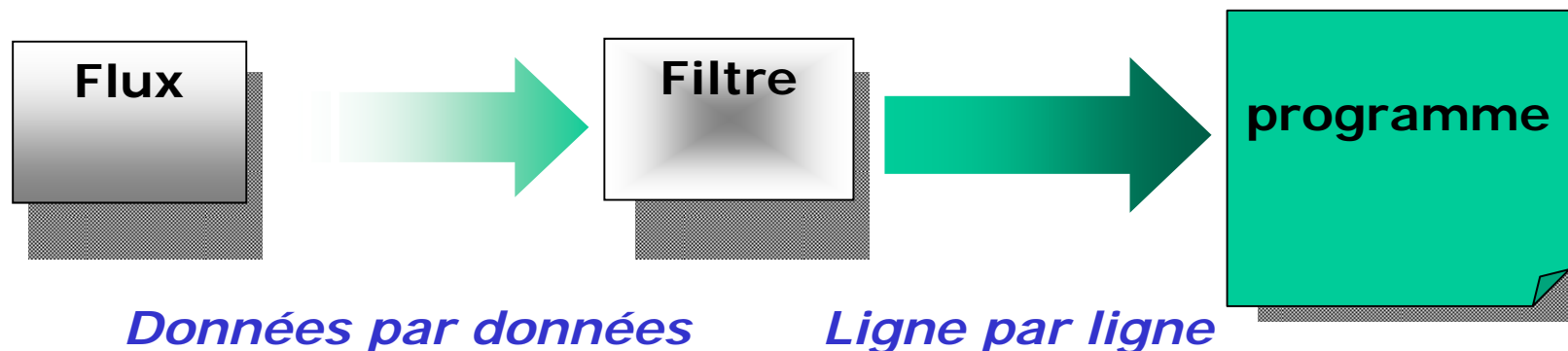
# Filtrer un flux

- L'utilisation d'un filtre permet par association à un flux **d'optimiser la lecture ou l'écriture des données**

- Filtre = flux modifiant la gestion d'un flux

- Utilisation d'un filtre :

- Création du flux de données
  - Association du filtre au flux
  - Lecture ou écriture des données à partir du filtre
  - Fermeture flux de données



- Deux classes abstraites :
  - **InputStream** : entrées de flux
  - **OutputStream** : sortie de flux
  - **On ne peut pas créer d'objets directement (classes abstraites)**
- À partir d'elles, utilisation des sous-classes :
- **FileInputStream et FileOutputStream :**
  - **Flux d'octets** stocké dans des fichiers
- **DataInputStream et DataOutputStream.**
  - **Flux d'octets filtré** à partir duquel des données telles que des entiers et des nombres à virgule flottante peuvent être lus.



# Flux d'entrée d'octets

## 16.1 ReadBytes

- Objet : **FileInputStream**

- Constructeur : **FileInputStream(String)**, String étant le nom du fichier concerné

- Exemple :

```
FileInputStream fis = new FileInputStream("scores.dat");
```

- Méthodes :

- **read()** – lecture d'un octet, retourne -1 à la fin du flux

- **read ( byte[ ] , int, int)** - Les arguments de cette méthode sont les suivants

- une matrice d'octets où seront stockées les données

- l'élément de la matrice dans lequel le premier octet des données doit être stocké

- le nombre d'octets à lire.

- Objet : **FileOutputStream**

- Constructeur : **FileOutputStream(String,boolean),**

- String étant le nom du fichier concerné

- boolean permet de préciser si le fichier doit être complété ou non

- Exemple :

- ```
FileOutputStream s = new FileOutputStream("scores.dat");
```

- Méthodes :

- **write(int)** – écriture d'un octet

- **write( byte[ ] , int, int)** - Les arguments de cette méthode sont les suivants

- une matrice d'octets où seront lues les données

- l'élément de la matrice dans lequel le premier octet des données doit être lu

- le nombre d'octets à écrire.



- Ils permettent de traiter les informations par volume ou par paquet, donc plus rapidement
- Les filtres peuvent fonctionner via **un tampon (ou buffer)**
- Un tampon est **un emplacement de stockage**
  - Les données peuvent être conservées avant qu'un programme lisant ou écrivant ces données en ait besoin.
  - Grâce à un tampon, vous pouvez obtenir des données sans revenir constamment à la source d'origine des données.



# Flux à tampon pour Flux d'E/S

## 16.3 Flux à tampon

- Remplit un tampon de données qui n'ont pas été encore traitées.
  - Si le programme a besoin des données, il va dans le tampon puis dans la source s'il ne trouve pas les données
- Objet : **BufferedInputStream**
- Constructeurs :
  - **BufferedInputStream(InputStream)**
    - Crée un flux d'entrée à tampon pour l'objet InputStream spécifié
  - **BufferedInputStream(InputStream,int)**
    - Crée le flux à tampon InputStream avec un tampon de taille int.
- Pour flux de sortie :
  - Objet : **BufferedOutputStream**
  - Constructeurs : même définition que pour les entrées

- Utilisé avec des données non représentées par des octets ou des caractères.
- Ces flux filtrent un flux d'octets pour que les différents types puissent être lus (**boolean, byte, double, float, int, long ...**)
  - Objet : **DataInputStream** (entrée), **DataOutputStream** (sortie)
  - Constructeur :
    - **DataInputStream(InputStream),**
    - **DataOutputStream(OutputStream)**
  - **InputStream** et **OutputStream** étant le flux d'entrée/sortie existant

- Méthodes :
  - La liste suivante indique les méthodes de lecture et d'écriture applicables, respectivement, aux flux d'entrée et aux flux de sortie
    - **readBoolean(),writeBoolean(boolean) ;**
    - **readByte(),writeByte(integer) ;**
    - **readDouble(),writeDouble(double) ;**
    - **readFloat(),writeFloat(float) ;**
    - **readInt(),writeInt(int) ;**
    - **readLong(),writeLong(long) ;**
    - **readShort(),writeShort(int).**
    - ...
  - Chacune des méthodes d'entrée retourne le type de données primitif indiqué par le nom de la méthode.
  - Par exemple, la méthode **readFloat ()** retourne une valeur **float**

- Les flux de caractères servent à travailler avec presque tout texte représenté par le jeu de caractère ASCII ou l'Unicode
  - Exemples de fichiers :
    - fichiers de texte brut,
    - documents HTML,
    - fichiers source Java,
    - etc...
- Ce sont des sous-classes des classes **Reader** et **Writer**



## Flux de caractères en entrée

- Objet : **FileReader**
- Constructeur : **FileReader(String)**,
  - String étant le nom du fichier concerné
- Exemple :  
**FileReader look = new FileReader("index.html");**
- Méthodes :
  - **read( )** - Retourne le caractère suivant du flux sous forme d'entier.
  - **read (char[], int, int)** - Lit les caractères de la matrice de caractères spécifiée,
    - depuis le point de départ indiqué par le premier entier,
    - et en tenant compte du nombre de caractères indiqué par le second entier.

# Flux de caractères en entrée - exemple

- La méthode suivante charge un fichier texte (readme.txt)

FileReader text, et affiche ses caractères :

```
FileReader text = new FileReader("readme.txt");
int inByte;
do {
    inByte = text.read();
    if (inByte != -1)
        System.out.print( (char)inByte );
} while (inByte != -1);
System.out.println("");
text.close();
```

- La méthode **read()** d'un flux de caractères retourne un entier,
  - Vous devez convertir l'entier en caractère avant de l'afficher,
  - Chaque caractère possède un code numérique représentant sa position dans le jeu de caractères Unicode.
  - L'entier lu à partir du flux est son code numérique.





## Flux de caractères en entrée

- Si vous souhaitez lire une ligne de texte au lieu de lire un fichier caractère par caractère :
  - Utiliser la classe **BufferedReader** associée à un objet **FileReader**
  - La classe **BufferedReader** lit un flux d'entrée de caractères et le place dans un tampon pour plus d'efficacité
  - Il faut disposer d'un objet **Reader** existant pour créer une version à tampon



## Flux de caractères en entrée

- Les méthodes constructeur suivantes peuvent servir à créer une classe **BufferedReader** :
  - **BufferedReader (Reader) :**
    - Crée un flux de caractères à tampon associé à l'objet Reader spécifié
  - **BufferedReader(Reader, int) :**
    - Crée un flux de caractères à tampon associé à l'objet Reader spécifié, avec un tampon dont la taille est int



## Flux de caractères en entrée

### 16.5 Flux de caractères en entrée

- Un flux de caractères à tampon est lu à l'aide des méthodes **read ( )** et **read (char(), int, int)** décrites pour l'objet **FileReader**
  - Vous pouvez lire une ligne de texte en utilisant la méthode **readLine()**
  - Elle retourne un objet String contenant la prochaine ligne de texte du flux
  - A l'exclusion du ou des caractères représentant la fin d'une ligne.
  - Si la fin du flux est atteinte, la valeur de la chaîne retournée sera égale à null
- Une fin de ligne est indiquée par
  - un caractère de nouvelle ligne (' \n') ;
  - un caractère de retour chariot (' \r');
  - un retour chariot suivi d'une nouvelle ligne.



## Flux de caractères en sortie

- La classe **FileWriter** sert à écrire un flux de caractères dans un fichier.
  - C'est une sous-classe de la classe **OutputStreamWriter**, dont le comportement convertit les codes de caractères Unicode en octets.
  - Constructeurs :
    - **FileWriter(String)**
      - La chaîne indique le nom du fichier vers lequel le flux de caractères sera dirigé
    - **FileWriter(String, boolean)**
      - L' argument boolean optionnel doit être égal à true s'il faut ajouter le fichier à la fin d'un fichier texte existant.



# Flux de caractères en sortie

- Méthodes de la classe **FileWriter** :

- **write(int)** : Ecrit un caractère.

- **write(char[], int, int)** :

- Ecrit des caractères de la matrice de caractères spécifiée, du point de départ indiqué par 1<sup>er</sup> entier, et en tenant compte du nbre de caractères indiqué par le 2<sup>e</sup> entier.

- **write (String, int, int)** :

- Ecrit des caractères de la chaîne spécifiée, en partant du point de départ indiqué, et en tenant compte du nombre de caractères indiqué

- Exemple :

```
FileWriter letters = new FileWriter("alphabet.txt");  
for (int i = 65; i < 91; i++) letters.write( (char)i );  
letters.close();
```

- **close()** sert à fermer le flux quand tous les caractères ont été envoyés au fichier de destination.



## Flux de caractères en sortie

- La classe **BufferedWriter** peut servir à écrire un flux de caractères à tampon.
- Les objets de cette classe sont créés à l'aide des méthodes constructeur **BufferedWriter(Writer)** ou **BufferedWriter(Writer, int)**
  - Writer peut être n'importe quelle classe de flux de sortie de caractères, telle que FileWriter.
  - Le second argument, facultatif, est un entier indiquant la taille du tampon à utiliser.
- La classe **Bufferedwriter** possède les trois mêmes méthodes de sortie que la classe FileWriter :
  - **write(int),**
  - **write(char[], int, int))**
  - **write(String, int, int)**



## Flux de caractères en sortie

### 16.6 Flux de caractères en sortie

- Autre méthode de sortie utile : **newLine()**, envoie le caractère de fin de ligne préféré de la plate-forme sur laquelle le programme s'exécute.
- **close ()** est appelée pour fermer le flux de caractères à tampon et vérifier que toutes les données du tampon sont envoyées vers la destination du flux.





## Flux de caractères en sortie - exercice

### 16.7 Exercice Flux de caractères

- Rechercher dans un fichier les différentes occurrences d'une chaîne de caractères spécifiée
- Utiliser la classe Clavier pour entrer les 2 paramètres
- Afficher en fin de programme, l'ensemble des lignes trouvées et leur n° de ligne

- Classe **File** (java.io), représente une référence à un fichier ou à un dossier.
- Méthodes :
  - **File (String)** : Crée un objet File avec le dossier spécifié -aucun nom de fichier n'étant indiqué, l'objet ne fait donc référence qu'à un dossier de fichier.
  - **File (String, String)** : Crée un objet File avec le chemin de dossier et le nom spécifiés.
  - **File(File, String)** : Crée un objet File dont le chemin est représenté par File spécifié, et dont le nom est spécifié par String.
  - **exists ()** : retourne une valeur booléenne indiquant si le fichier existe
  - **length ()** : retourne un entier long indiquant taille du fichier en octets.
  - **renameTo (File)** : renomme le fichier pour lui faire adopter le nom spécifié par l'argument File. Une valeur booléenne est retournée, indiquant si l'opération a réussi.

## 16.8 Les fichiers

### ■ Méthodes :

- **delete ()** ou **deleteOnExit ()** doivent être appelées pour supprimer un fichier ou un dossier.
- **mkdir ()** peut servir à créer le dossier spécifié par l'objet File sur lequel elle est appelée.
- Aucune méthode n'est disponible pour inverser la suppression d'un fichier ou d'un dossier.
- **Attention :**
  - pour chacune des méthodes, génération d'une exception **securityException** si le programme ne se trouve pas dans les conditions de sécurité requises pour effectuer l'opération de fichier en question
  - Il faut donc prévoir pour elles un **bloc try. . . catch** ou une clause throws dans une déclaration de méthode.



## **17. Connexion aux bases de données JDBC**

## **JDBC : Java DataBase Connectivity**

Jeux de classes Java permettant le développement d'application base de données client-serveur ou en architecture n tiers

### **Obstacle majeur :**

Variété des formats base de données → multiples méthodes d'accès

### **Avantage :**

Utilisation du standard SQL pour accéder aux bases de données relationnelles

Utilisation du package : **java.sql**



## Pilote base de données

JDBC est conforme au modèle Java (indépendance de la plate-forme)

Mais utilise des gestionnaires de pilotes (pilotes nécessaires à l'utilisation de la base de données concernée)

Un pilote par base de données, voire un pilote par version de base de données

Exemple de pilote base de données Windows : ODBC

ODBC, l'interface commune conçue par Microsoft pour accéder aux bases de données SQL, est gérée sur un système Windows par l'Administrateur de source de données ODBC.



## Accès à une base de données

Tâches associées à l'utilisation d'une base de données (6 étapes) :

- 1) Chargement d'un pilote JDBC
- 1) Définition de l'URL de connexion
- 2) Connexion
- 1) Création d'une instruction, exécution
- 1) Traitement des résultats.
- 1) Fermeture de la connexion.





## Chargement d'un pilote

- On n'a pas besoin de créer une instance de classe.
- On utilise le DriverManager (gestionnaire de pilotes) par la méthode **Class.forName**
- Il faut connaître le type de base de données que l'on utilise, ainsi que le pilote associé.
- Chargement d'un pilote :  
**Class.forName( "jdbc.piloteXYZ" );**
- Exemple : Chargement du pilote JDBC-ODBC Bridge  
**Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver" );**



## Définition de l'URL de connexion

- Création d'un objet de connexion : classe `Connection`
- Appel du driver utilisé

### Syntaxe :

```
Connection ma_Connexion = DriverManager.getConnection(url,  
    "MonLogin", "MonMotDePasse");
```

- `url` : nom de la source de donnée
- `MonLogin` : user d'accès à l'application
- `MonMotDePasse` : mot de passe de l'utilisateur



## Définition de l'URL de connexion, exemple

### **Exemple :**

Utilisation d'une base de données baseTest en utilisant le bridge JDBC-ODBC, pour le user jmf, mot de passe : jmf.

Spécification de l'url : jdbc:odbc:baseTest

```
Connection maConnexion = DriverManager.getConnection(  
    "jdbc:odbc:baseTest"; "jmf"; "jmf" );
```

La spécification de l'url de connexion varie suivant le protocole de connexion utilisé



## Instruction JDBC, objet Statement

- Un objet Statement permet de véhiculer les ordres SQL vers la base de données
- Il faut une instance de connexion active pour créer un objet Statement
- Exemple : Avec utilisation d'une connexion maConnexion  
Création de l'objet Statement :

```
Statement stmt = maConnexion.createStatement();
```



# Interrogation, ExecuteQuery

A partir de l'objet Statement, utilisation de la méthode executeQuery :

## **Syntaxe :**

```
Objet_Statement.executeQuery("requete");
```

## **Exemple :**

Chargement et exécution d'une instruction d'interrogation :

```
stmt.executeQuery("SELECT * FROM *");
```

Le résultat de la requête est récupéré dans un objet ResultSet.



## Mise à jour, ExecuteUpdate

- A partir de l'objet Statement, utilisation de la méthode executeUpdate :
- **Syntaxe :**
- `Objet_Statement.executeUpdate("requete");`
- Elle permet la mise à jour de la structure, ainsi que des données.
- **Exemple :**
- Chargement et exécution d'une instruction de mise à jour :  
`Stmt.executeUpdate(" CREATE TABLE ELEVE(ID_ELEVE  
INTEGER,NOM_ELEVE VARCHAR(32)");`  
`Stmt.executeUpdate(" INSERT INTO ELEVE VALUE (1, " Dupont  
Georges" );`
- Le résultat de l'instruction peut prendre différentes valeurs :
  - Nombre de lignes de table modifiés
  - 0 dans le cadre de la mise à jour de structure de table

Les objets permettant pour obtenir des informations SGBD sont :

## **DataBaseMetaData :**

- Contient les informations sur la base de données (des *métadonnées*),
- c'est-à-dire le nom de la table, les index, la version de la base, ...

## **ResultSet:**

- Objet contenant les informations sur une table ou le résultat d'une requête.
- L'accès aux données se fait colonne par colonne, mais il est éventuellement possible d'accéder indépendamment à chaque colonne par son nom

## **ResultSetMetaData:**

- Un objet contenant des informations sur le nom et le type des colonnes d'une table



- L'objet **ResultSet** est l'objet le plus important de JDBC.
- Presque toutes les méthodes et requêtes retournent les données sous forme d'un objet **ResultSet**.
- Cet objet contient un nombre donné de colonnes repérées chacune par un nom, ainsi qu'une ou plusieurs lignes contenant les données, et auxquelles il est possible d'accéder séquentiellement une à une du haut vers le bas.
- Ainsi, afin d'exploiter un objet **ResultSet**, il est nécessaire de récupérer le nombre de colonnes de celui-ci, à l'aide de l'objet **ResultSetMetaData**.



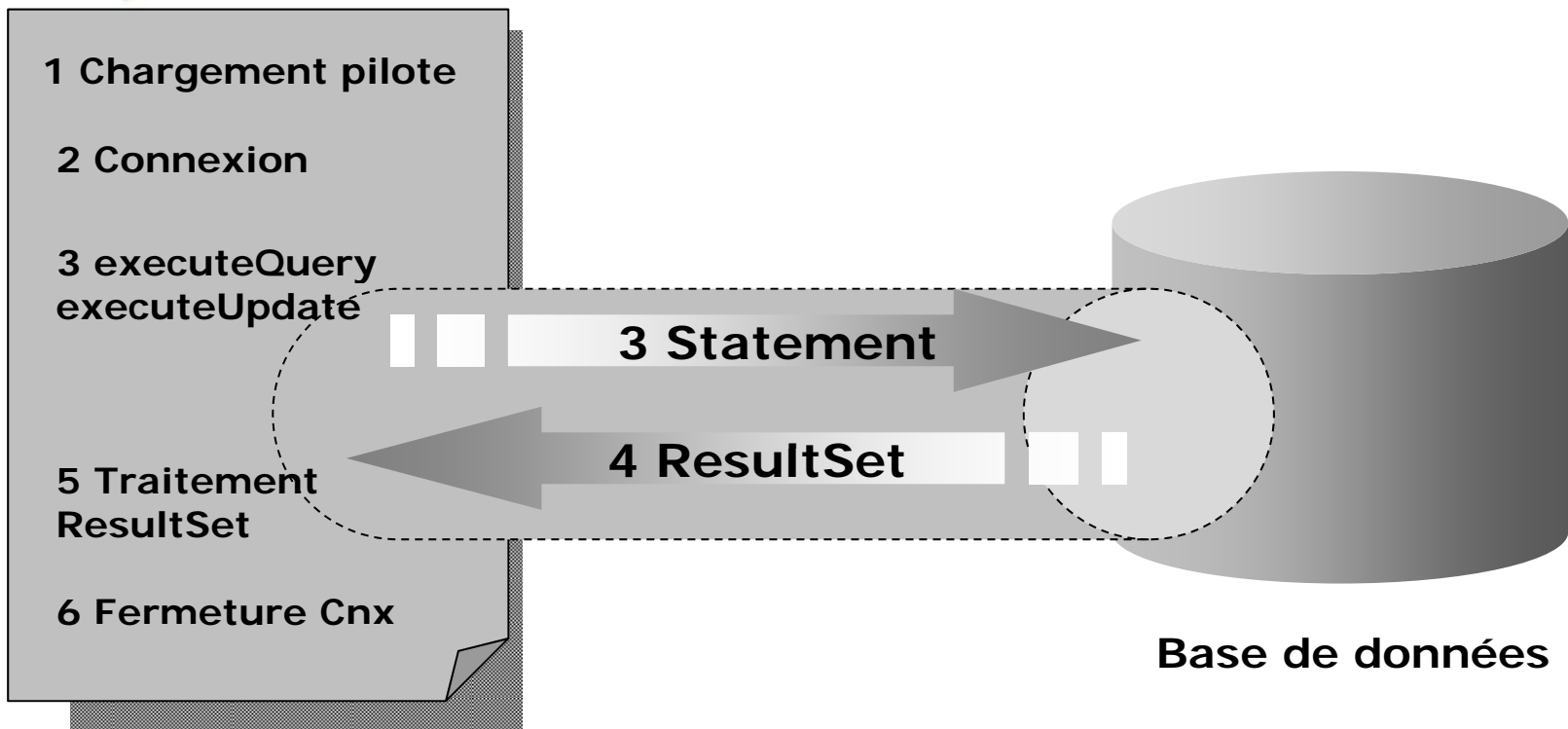
## ResultSet – Principales Méthodes

- **getInt(int)**: récupère sous forme d'entier le contenu d'une colonne désignée par son numéro
- **getInt(String)**: récupère sous forme d'entier le contenu d'une colonne désignée par son nom
- **getFloat(int)**: récupère sous forme de réel le contenu d'une colonne désignée par son numéro
- **getFloat(String)**: récupère sous forme de flottant le contenu d'une colonne désignée par son nom
- **next()**: déplace le pointeur de colonne sur la colonne suivante
- **close()** : "ferme " l'objet
- **getMetaData()**: retourne les métadonnées de l'objet (l'objet *ResultSetMetaData*)

- **getColumnCount()**: récupère le nombre de colonnes
- **getColumnName(int)**: récupère le nom de la colonne spécifiée
- **getColumnLabel(int)**: récupère le label de la colonne spécifiée
- **getColumnType(int)**: récupère le type de données de la colonne spécifiée

# JDBC – Synoptique de fonctionnement

Programme Java





# Objet PreparedStatement

Cet objet permet la préparation d'instructions SQL paramétrables par la biais de la sous-classe de la classe Statement.

Cet objet permet lors de l'utilisation multiple d'une requête SQL, de réduire le temps de traitement comparativement à un objet Statement.

## Principe :

- Affecter une instruction SQL précompilée par le SGBDR (plus rapide)
- Passer les paramètres utiles à l'objet

## Syntaxe :

```
PreparedStatement objet = ma_connexion.prepareStatement(  
"Requête SQL + paramètres ?");
```



# Objet PreparedStatement

## Exemple :

```
PreparedStatement updateEleve = conn.prepareStatement(  
" INSERT INTO ELEVE VALUE (?, ?) ");
```

- Il convient ensuite de fournir la valeur des paramètres à la requête.
- Pour cela, il faudra utiliser l'une des méthodes **setXXX** définies dans la classe PreparedStatement.
- Si la valeur que vous voulez substituer au point d'interrogation est un int Java, vous devrez appeler la méthode setInt.
- Si la valeur est un String Java, vous devrez appeler la méthode setString, etc...
- En indiquant le n° du paramètre concerné

## Exemple :

```
updateEleve.setInt(1,1);  
updateEleve.setString(2,"Georges Dupont");
```



# Utilisation des transactions

- jeu d'instructions exécutées ensembles de façon unitaire, donc toutes les instructions sont exécutées, ou aucune.
- Permettent de faire des mises à jour combinées, dans le cadre par exemple de dépendance entre clefs étrangères et primaires.

## Mode auto-commit :

- A la création d'une connexion, elle est en mode auto-commit.
  - ➔ Chaque instruction SQL est traitée comme une transaction et prendra automatiquement effet après sa bonne exécution (instruction terminée).
  - ➔ Une instruction est terminée quand tous ses résultats et toutes ses mises à jour ont été rapportées.
  - ➔ La manière d'autoriser deux ou plusieurs instructions à être groupées dans une transaction est de mettre hors service le mode auto-commit.
  - ➔ C'est ce que nous faisons à la ligne suivante, où conn est notre connexion active :

```
conn.setAutoCommit(false);
```





# Utilisation des transactions

- Une fois que le mode auto-commit est désactivé, aucune instruction SQL ne prendra effet jusqu'à ce que la méthode commit soit employée.

Dans le cas d'une transaction de requêtes groupées, on effectuera :

- Désactivation du mode auto-commit
- Exécution des requêtes
- Commit (pour prise en compte et mise à jour base de données)
- Activation du mode auto-commit

## Activation

```
conn.commit();  
conn.setAutoCommit(true);
```



## Utilisation de RollBack

- La méthode **rollback()** **annule une transaction** et remet les valeurs qui ont été modifiées à leurs valeurs originales.
- Si vous essayez d'exécuter une ou plusieurs instructions dans une transaction et que vous avez une SQLException, vous devrez utiliser la méthode rollback() pour annuler la transaction et la recommencer depuis le début.
- C'est la seule façon d'être sûr de ce qui a été pris en compte sur la base de données, et ce qui ne l'a pas été.
- Une SQLException vous indique que quelque chose ne fonctionne pas, mais elle ne vous indique pas ce qui a été pris en compte ou pas.
- Donc vous ne pouvez pas compter sur le fait que rien n'a été mis à jour sur la BD.



## Procédures stockées

Ordres SQL stockés sur la base de données (très rapide)

### Syntaxe :

```
create procedure nom_procédure as requête
```

### Exemple :

```
String createProcedure = "create procedure  
VOIR_ELEVES as SELECT NOM_ELEVE FROM ELEVE  
order by NOM_ELEVE";  
Statement stmt = conn.createStatement();  
stmt.executeUpdate(createProcedure);
```

La procédure VOIR\_ELEVES sera compilée et stockée dans la base de données comme un objet pouvant être appelé de façon similaire à l'appel d'une méthode.



## Appel des procédures stockées

### Appel d'une procédure stockée :

Créer un objet **CallableStatement** (sous-classe Statement)

Comme avec les objets Statement et PreparedStatement, ceci est fait avec une connexion ouverte, il contient l'appel d'une procédure, il ne contient pas la procédure elle-même.

### Syntaxe :

```
CallableStatement objet = conn.prepareCall("{call  
nom_procedure_stockée}");
```

### Exemple :

```
CallableStatement cs = conn.prepareCall("{call VOIR_ELEVES}");  
ResultSet rs = cs.executeQuery();
```

- La méthode pour exécuter cs est executeQuery car cs appelle une procédure stockée qui contient une requête et produit un resultset.
- Si la procédure avait contenue une mise à jour ou une des instructions DDL, la méthode executeUpdate aurait été utilisée.



## Fermeture de connexion

Utilisation de la méthode close sur la connexion

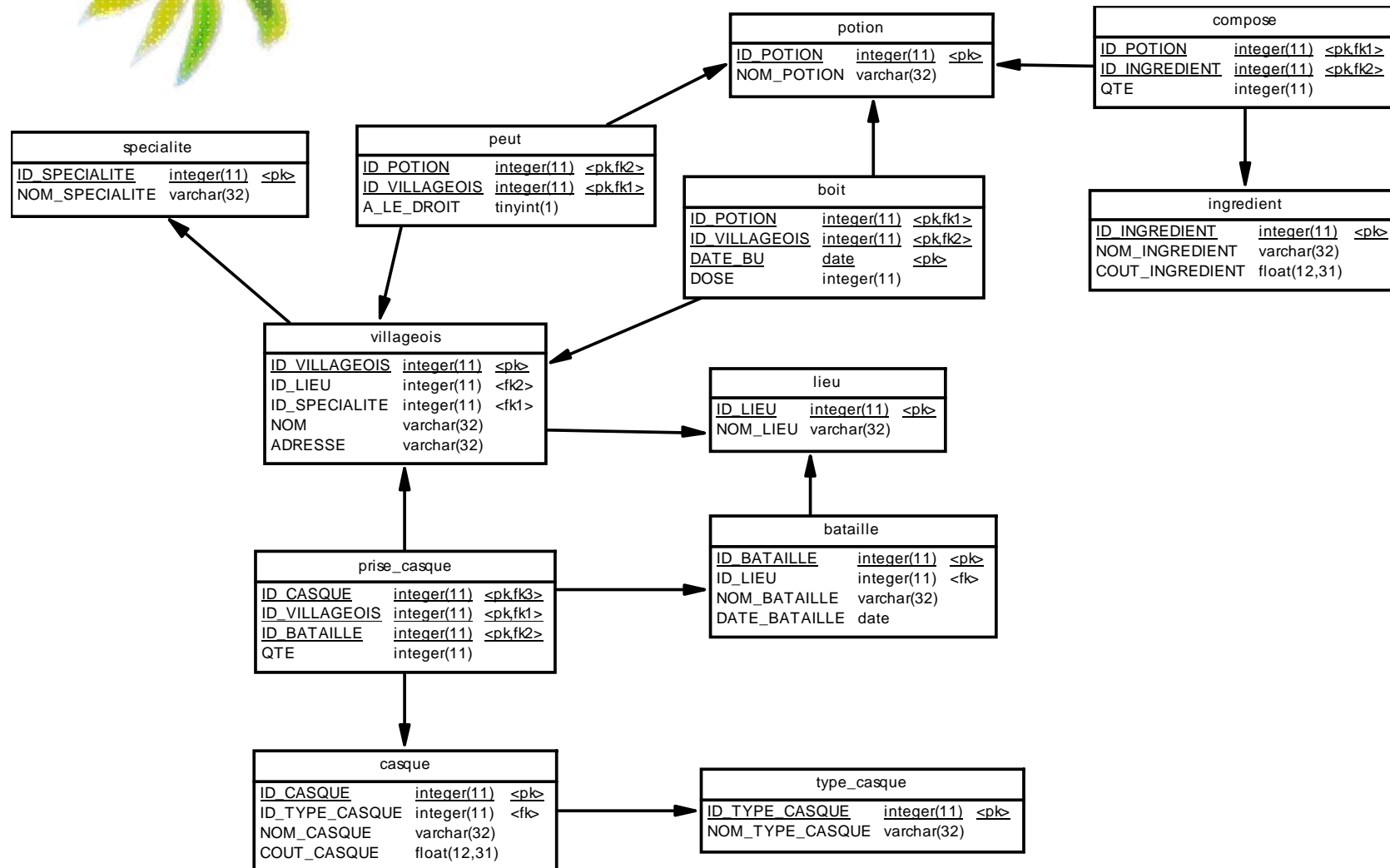
**Syntaxe :**

```
Connexion.close( )
```

# JDBC - Exemple

## 17.1 JDBC

- Liste des potions et des ingrédients de la base de données lesGaulois



# JDBC - Exemple

## 17.2 Exercice JDBC

- Exporter la liste des villageois et leur spécialités dans un fichier en sortie

