

# JAVA



Moussa Camara

[Msa.camara@gmail.com](mailto:Msa.camara@gmail.com)

Tèl: 06 50 14 63 51

# JAVA: C'EST QUOI DÉJÀ?

- Le Java est un langage informatique orienté « objet » au sens technique du terme. Il se veut relativement simple d'utilisation et compatible avec un maximum de systèmes d'exploitation.
- Le langage Java a été créé en 1995 par James Gosling et Patrick Naughton. Il s'inspire du langage C++. Il est désormais édité par la société Oracle. Java est présent sur la plupart des systèmes d'exploitation informatique.
- Il se base sur un langage orienté objet et familier avec notamment l'utilisation de terme anglais provenant du langage humain. Il est omniprésent sur les PC, et il peut également être présent sur d'autres supports comme les smartphones et les tablettes, bien que moins courant. Nombreux sont les sites utilisant Java, pour y accéder il est donc nécessaire que l'utilisateur télécharge gratuitement le logiciel Java permettant de lire le langage ces sites.

# JAVA: ET POURQUOI ?

- La dernière version de Java comprend d'importantes améliorations en matière de performances, de stabilité et de sécurité pour les applications Java exécutées sur votre ordinateur. L'installation de cette mise à jour gratuite garantit que les applications Java sont toujours exécutées de manière sécurisée et efficace.
- Ses avantages:
  - Portabilité, il existe des JVM pour l'ensemble des OS du marché.
    - La machine virtuelle Java (**Java virtual machine**) est un appareil informatique fictif qui exécute des programmes compilés sous forme de bytecode Java.
  - Le bytecode exporté est léger.
    - bytecode destiné à regrouper des instructions exécutables par une machine virtuelle java.
  - Sécurité, la JVM lance de nombreuses vérifications sur le bytecode.

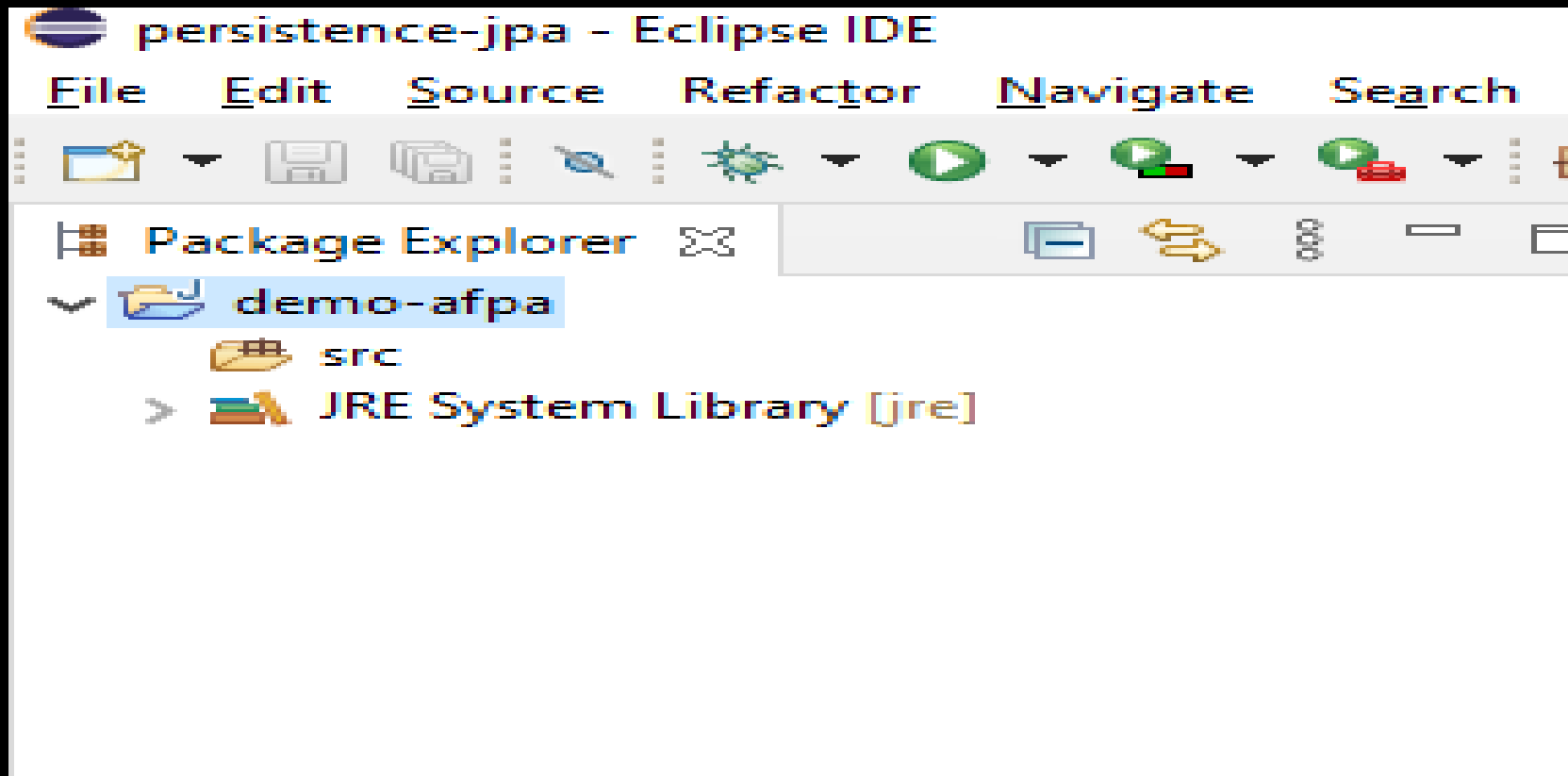
# JAVA: INCOVÉNIENTS

- Ses inconvénients:
  - Lenteur de l'exécution (compilation + interprétation) mais d'autres alternatives existent. Quelques règles fondamentales
  - Le nom d'une classe commence toujours par une majuscule.
  - Les mots contenus dans un identificateur commencent par une majuscule : HelloWorld.
  - Les constantes sont en majuscules.
  - Les propriétés et les méthodes débutent par une minuscule.

# JAVA: INSTALLATION

- On aura besoin de trois prérequis:
  - Java: <https://www.java.com/fr/download/>
  - Le JDK: <https://www.oracle.com/java/technologies/javase-downloads.html>
  - Si vous ne l'avez pas encore: télécharger l'IDE que vous souhaitez
    - Eclipse: <https://www.eclipse.org/downloads/>
    - Netbeans: <https://netbeans.apache.org/download/index.html>
    - IntelliJ: <https://www.jetbrains.com/fr-fr/idea/>

# JAVA: INSTALLATION



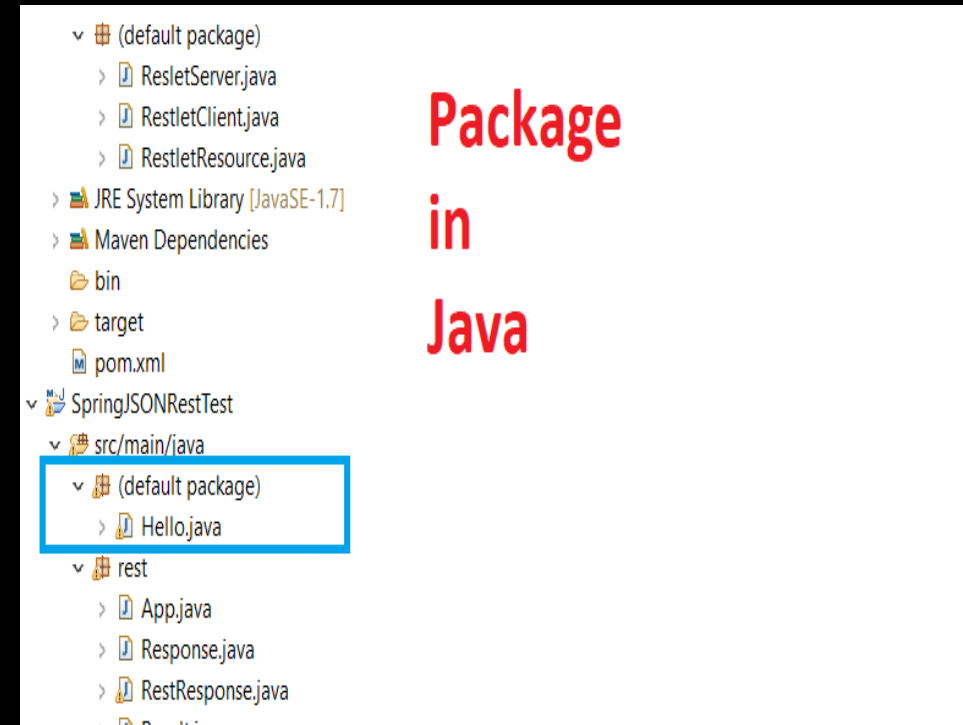
# JAVA: ARCHITECTURE

- Le dossier src contiendra tout notre code source et dans lequel nous passerons 99%de notre temps
- Le dossier jre contient tout le JDK c'est-à-dire toutes les pakcages dépendance de notre application.



# JAVA: PREMIÈRE APPLICATION

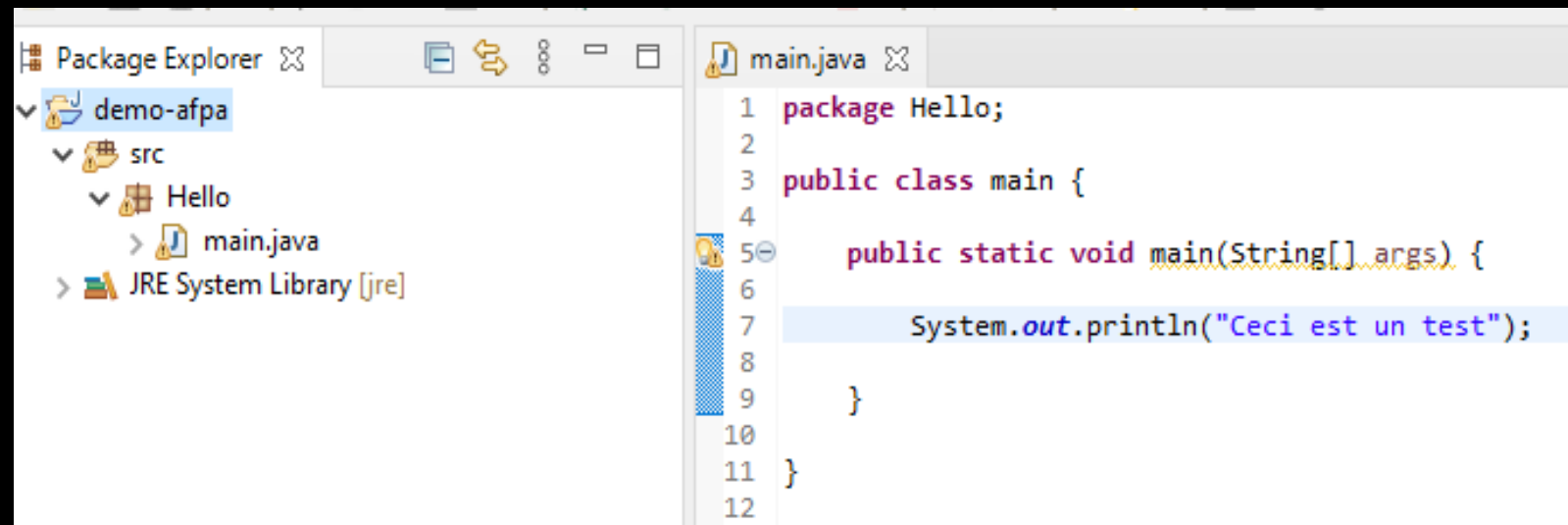
- Maintenant que nous avons tout installer, nous allons créer notre première application web.
- Commençons par le package:
  - Un package permet de regrouper les classes en ensemble pour faciliter la modularité.





# JAVA: MAIN

- Créons une classe main
- La méthode **main** constitue la partie principale du programme, permettant l'exécution d'une application Java.
- `public` indique que la méthode peut être appelée par n'importe quel objet. ... Lorsque la méthode `main` est trouvée dans une classe, elle devient la méthode à partir de laquelle démarre automatiquement le programme.



# JAVA – SYSTEM.OUT.PRINTLN()

- Explication: `System.out.println(« Ceci est un test ... coco »);`
- System : C'est le nom de la classe standard qui contient les objets qui encapsule la norme I/O dispositifs de votre système.
- out : l'objet out représente le flux de sortie(l. e commandement fenêtre) et est le membre de données statiques de la classe System
- println : Le println() est méthode de out objet prend la chaîne de texte comme argument et l'affiche à la norme résultats l. e sur l'écran du moniteur .

System -Classe

out -objet statique

println() -méthode

# JAVA: ORIENTÉE OBJET

- Comme évoqué un peu plus haut, java est un langage informatique orienté « objet » mais c'est quoi la programmation orientée objet déjà ?

**POO**

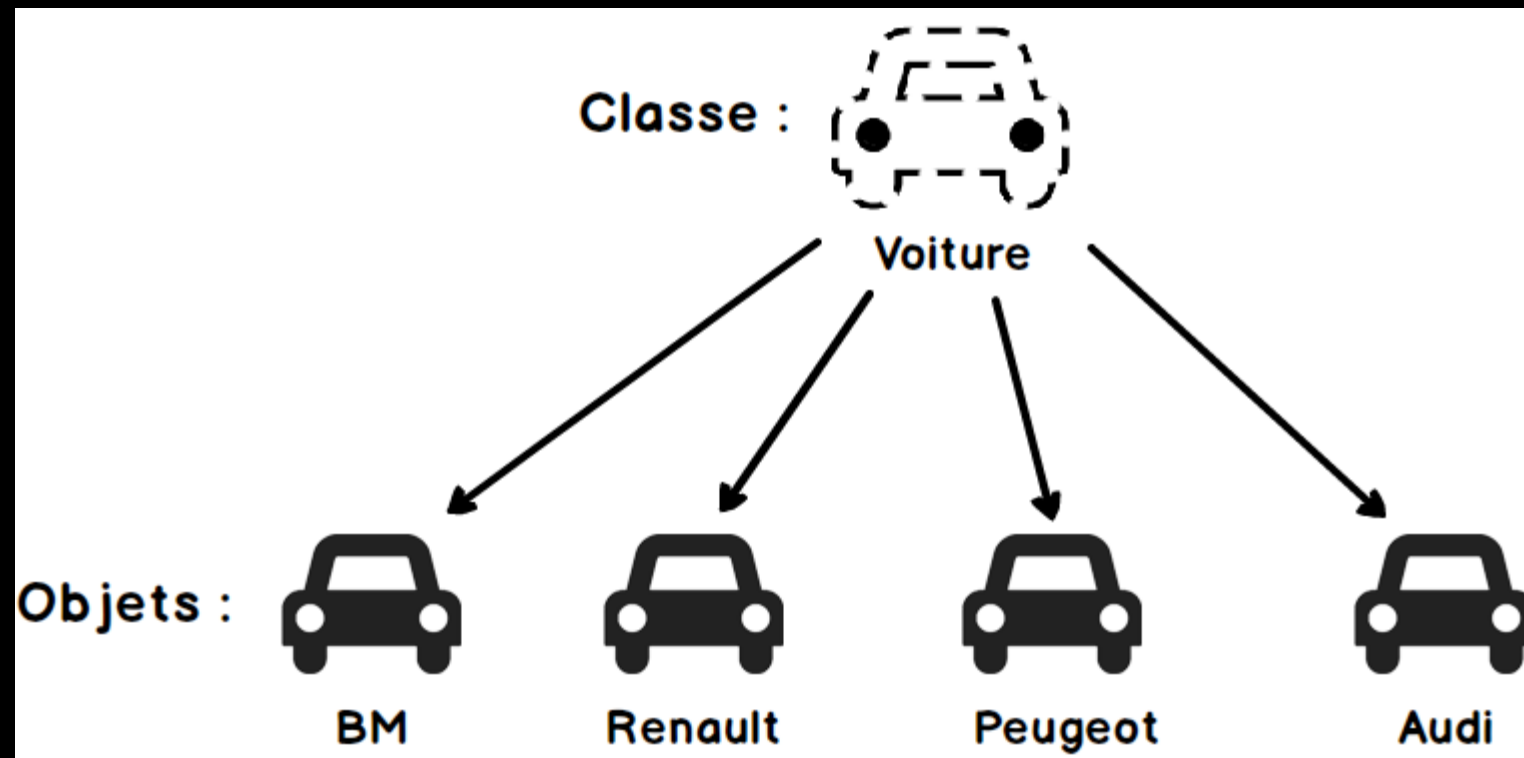
PROGRAMMATION ORIENTEE OBJET

# LA POO: PROGRAMMATION ORIENTÉE OBJET

- La programmation orientée objet a été inventée dans le but de structurer la conception de logiciels, de manière à les rapprocher de la réalité.
- En effet, on peut observer que dans la vie courante, chaque objet peut être décomposé en un ensemble d'objets plus petits.
- En Java, il suffira donc de partir de la description globale d'un type d'objet pour ensuite définir chacun de ses sous-ensembles dérivés.
- Les définitions les plus générales et les plus abstraites effectuées au sein d'un programme, pouvant donner naissance à tout un ensemble d'objets dérivés, répondront au nom de classes.

# RETENONS POO

On y reviendra en plus détails un peu tard



# JAVA: LES VARIABLES

- UNE VARIABLE EST UN CONTENEUR QUI PEUT CONTENIR À PEU PRÈS TOUT. UN NOMBRE, UNE CHAÎNE DE CARACTÈRE, UN TABLEAU...
- Les variables en langage Java sont typées, c'est-à-dire que les données contenues dans celles-ci possèdent un type, ainsi elles sont donc stockées à une adresse mémoire et occupent un nombre d'octets dépendant du type de donnée stockée.

Data Type	Description	Size	Range	Example
int	Signed 2's complement integer	32 bits	- 2,147,483,648 to 2,147,483,647	int rollNo = 34; int totalMarks = -234;
float	Single precision IEEE 754 floating point	32 bits	3.4e-038 to 3.4e+038	float height = 5.1f;
double	Floating point numbers	64 bits	1.7e-308 to 1.7e+038	double average = 567.23;
char	Single character	16 bits	0 to 65,536	char grade = 'A';
short	Signed 2's complement integer.	16 bits	-32768 to 32767	short age = 23;
long	Signed 2's complement integer	64 bit	9,223,372,036,854,775,808 to 9,223,372,036,854,755,807	long creditMoney = 34567;
byte	Signed 2's complement integer	8 bit	-128 to 127	byte age = 23;
boolean	True or False	Not precisely defined	true or false	boolean webCheck = true;

# JAVA: LES VARIABLES

- Il existe deux sortes de variables :
  - les types primitifs qui contiennent des valeurs élémentaires
    - Exemple : int, float , boolean.
  - les références aux objets qui contiennent . . . des références aux objets.
    - Exemple : String, Chien, Joueur,...



# JAVA: TYPES PRIMITIFS

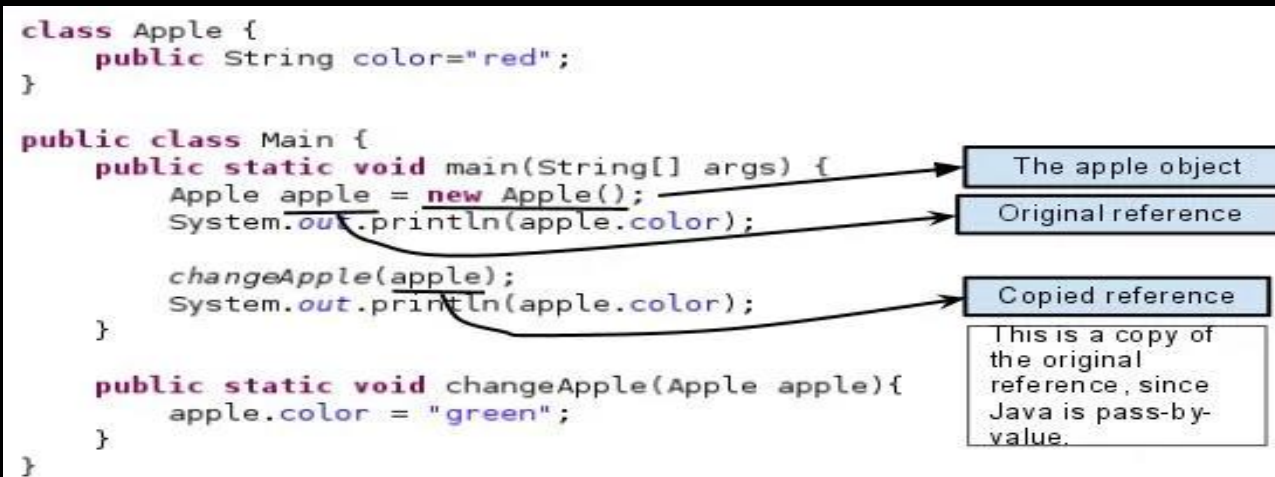
## les types primitifs

10/04/2011

Type primitif	Taille	Minimum	Maximum	Type wrapper
boolean	-	-	-	Boolean
char	16-bit	Unicode 0	Unicode 216-1	Character
byte	8-bit	-128	+127	Byte
short	16-bit	-215	+215-1	Short
int	32-bit	-231	+231-1	Integer
long	64-bit	-263	+263-1	Long
float	32-bit	IEEE754	IEEE754	Float
double	64-bit	IEEE754	IEEE754	Double

# JAVA: RÉFÉRENCES D'OBJETS

- Il n'existe pas de variable objet
  - Il n'y a que des variables références (ou références) à des objets
  - Une variable référence décrit comment accéder à un objet.
  - Elle ressemble à un pointeur ou une adresse mais ce n'est pas un pointeur, ni une adresse.
- 
- Une variable primitive contient des bits qui représentent sa valeur.
  - Une variable référence contient des bits qui indiquent comment accéder à l'objet.



# JAVA: RÈGLES SUR LES VARIABLES

- Concernant le nom de nos variables, nous avons une grande liberté dans le nommage de celles-ci mais il y a quand même quelques règles à respecter :
  - Le nom d'une variable doit obligatoirement commencer par une lettre ou un underscore (\_) et ne doit pas commencer par un chiffre ;
  - Le nom d'une variable ne doit contenir que des lettres, des chiffres et des underscores mais pas de caractères spéciaux ;
  - Le nom d'une variable ne doit pas contenir d'espace.
  - Le nom ne doit pas être un mot réservé du langage.

Nom de variable correct	Nom de variable incorrect	Raison
Variable	Nom de Variable	comporte des espaces
Nom_De_Variable	123Nom_De_Variable	commence par un chiffre
nom_de_variable	toto@mailcity.com	caractère spécial @
nom_de_variable_123	Nom-de-variable	signe - interdit
_707	transient	nom réservé

# JAVA: VARIABLES, VALEURS ET AFFECTATIONS

- ✓ Affectation de variables de types primitifs

L'exécution de la séquence suivante :

```
int x, y ;  
x = 1234 ;  
y = x ;  
x = x + 66 ;  
System.out.println( « x vaut » + x ) ;  
System.out.println( « y vaut » + y ) ;
```

- affichera :

x vaut 1300

y vaut 1234

- Les valeurs de x et y sont **distinctes**.

# JAVA: VARIABLES, VALEURS ET AFFECTATIONS

- ✓ Affectation d'une référence

Supposons que la classe *Personne* dispose de la méthode *integrerSociete* qui permet d'affecter une valeur à la variable d'instance *societe*.

Alors l'exécution de la séquence :

```
Personne dupont, martin;  
dupont = new Personne (« Dupont »);  
dupont.afficher() ;  
martin = dupont;  
martin.integrerSociete (« Java SARL ») ;  
dupont. afficher() ;
```

• affichera :

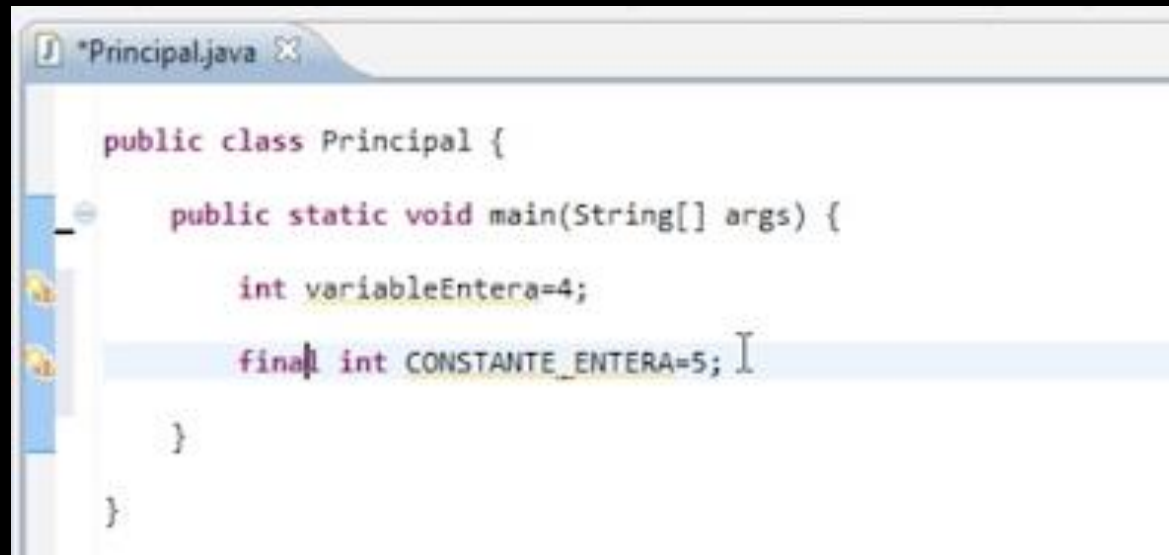
- Je m'appelle DUPONT
- Je ne suis pas salarié
- Je m'appelle DUPONT
- Je travaille chez Java SARL

- Les références dupont et martin désignent le même objet .



# JAVA: LES CONSTANTES

- Parfois, on manipule des données qui restent constantes durant toute l'exécution du programme.
- Pour stocker de telles valeurs, on peut utiliser des constantes qui sont des variables dont on ne peut changer la donnée qu'elles contiennent. Pour signaler qu'une variable est une constante, on utilise le mot réservé **final** lors de sa déclaration

A screenshot of a Java IDE window titled "Principal.java". The code defines a public class "Principal" with a public static void main method. Inside the main method, there are two integer declarations: "int variableEntera=4;" and "final int CONANTE\_ENTERA=5;". The word "final" is highlighted in red in the original image. The code is as follows:

```
public class Principal {  
    public static void main(String[] args) {  
        int variableEntera=4;  
        final int CONANTE_ENTERA=5;  
    }  
}
```

# JAVA - FINAL

- **final** : ce modificateur impose que la déclaration comporte une valeur d'initialisation et empêche toute affectation ultérieure d'une autre valeur. Il permet de définir une information constante.
  - Par ailleurs, lorsque l'on déclare un élément **final**, le compilateur est à même d'optimiser le code afin d'améliorer sa vitesse d'exécution.
- **static** : ce modificateur indique que toutes les instances de la classe se partageront un exemplaire unique : une variable de classe. Les éléments **static** d'une classe existent quel que soit le nombre d'instances.

- Exemple

```
class Personne {  
    private static final String PAS_DE_SOCIETE = «?» ;  
    private String nom;  
    private String societe;  
    ... // etc .  
}
```



# JAVA: MOTS RÈSERVÈS

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

# JAVA: LES TABLEAUX


Un tableau est un ensemble d'éléments basé sur un même type : par exemple, un tableau d'entiers, un tableau de chaînes de caractères, ...

Un tableau sera introduit par la syntaxe [] qui pourra être placée à la suite du type utilisé pour les éléments stockés dans ce tableau.

```
int [] tab = {1,2,3,4,5};  
System.out.println("Le premier element:" +tab[0]);
```

```
//Tous les elements de mon tableau  
for(int i = 0; i < tab.length; i++) {  
    System.out.println(tab[i]);  
}
```

```
// Autres methodes comme forEach  
for(int index : tab) {  
    System.out.println(" index : "+index);  
}
```

- 
- En Java, il n'y a pas que les tableaux qui permettent de stocker un ensemble de valeurs. Il y a aussi les collections Java. En fait, ces collections sont des classes qui, pour les principales, sont localisées dans le paquetage `java.util`.
  - Ces classes sont dites génériques, car elles permettent de typer les éléments qui seront stockés dans ces collections.
  - Une des différences entre un tableau Java et une collection Java:
    - Un tableau a une taille fixe
    - Une collection est dynamique (tant qu'il y a de la mémoire de disponible, on peut continuer à ajouter des éléments dans une collection).

- Par exemple, la classe `java.util.ArrayList` implémente un tableau d'éléments, mais extensible en fonction des besoins.

```
// Création de notre collection et ajouts de données
```

```
ArrayList<String> coll = new ArrayList<>();
```

```
coll.add( "azerty" );
```

```
coll.add( "qwerty" );
```

```
// Combien y a t'il de données dans la collection ?
```

```
System.out.println( "Size == " + coll.size() );
```

```
// Affichage des données de la collection
```

```
for (String string : coll) {
```

```
    System.out.println( string );
```

```
}
```



# EXERCICES

# JAVA: TABLEAUX MULTIDIMENSIONNELS

- Imaginons un jeu d'échec à développer. Ce jeu se joue sur un plateau de cases horizontale et verticale.

```
int[][] echiquier = {  
    {  
        1,2,3,4  
    },  
    {  
        5,6,7,8  
    }  
};  
  
//Boucles sur les deux tableaux  
for(int i = 0; i< echiquier.length; i++) {  
    for(int j = 0; j< echiquier[i].length; j++) {  
        System.out.println("\n echiquier : "+echiquier[i][j]);  
    }  
}
```

# JAVA: LES OPÉRATEURS

- Opérateurs arithmétiques:
- Voici la liste des différents opérateurs de calcul disponible en java.

+	<u>opérateur d'addition</u>	<u>Ajoute deux valeurs</u>
-	<u>opérateur de soustraction</u>	<u>Soustrait deux valeurs</u>
*	<u>opérateur de multiplication</u>	<u>Multiplie deux valeurs</u>
/	<u>opérateur de division</u>	<u>Divise deux valeurs</u>
%	<u>opérateur modulo</u>	<u>Retourne le reste de la division</u>
=	<u>opérateur d'affectation</u>	<u>Affecte une valeur à une variable</u>



# JAVA:

- Opérateurs de comparaison:
- Ils comparent des nombres ou des expressions qui retournent TRUE (vrai) si la comparaison réussit ou False (Faux) si elle échoue. Quand les opérandes sont de types différents, ils sont convertis avant comparaison.

# JAVA:

- Opérateurs de logiques:
- Aussi appelés opérateurs de booléens, ces opérateurs servent à vérifier deux ou plusieurs conditions.

Signe	Nom	Exemple	Signification
&&	et	(condition1) && (condition2)	condition1 <u>et</u> condition2
	ou	(condition1)    (condition2)	condition1 <u>ou</u> condition2
<u>Exemple :</u>			
<pre>if(a &lt;= 18 &amp;&amp; b == 20){     //instruction }</pre>			
<u>Exemple :</u>			
<pre>if(a == 18    b == 20){     //instruction }</pre>			
<u>Exemple :</u>			
<pre>if((a == 18 &amp;&amp; b == 20)    (a == 20 &amp;&amp; bb == 18)) {     //instruction }</pre>			

# JAVA: LA CONCATÉNATION

- Le terme “concaténer” signifie “joindre deux chaînes bout à bout pour n’en former qu’une seule”.
- `String chaine1 = “Bonjour”;`
- `String chaine2= “tout le monde”;`
- `String chaine3 = chaine1 + chaine2;`
- `System.out.println(chaine3); // Bonjour tout le monde`

# JAVA: LES CONDITIONS

- QUAND VOUS DÉVELOPPEZ UNE APPLICATION VOUS AVEZ BESOINS PAR MOMENT DE SAVOIR SI UNE VARIABLE EXISTE OU SI ELLE CONTIENT QUELQUE CHOSE OU ENCORE SI ELLE EXISTE ET QU'ELLE CONTIENT BIEN UNE VALEUR.... EN FAIT VOUS ALLER CHERCHER À TESTER PLUSIEURS CAS POSSIBLE AVANT DE FAIRE UNE ACTION.
- PAR EXEMPLE, SUR UN SITE DE VENTE DE EBOOKS EN LIGNE, VOUS NE LAISSEREZ L'UTILISATEUR TÉLÉCHARGER SON EBOOK QUE LORSQUE VOUS AVEZ VÉRIFIÉ SI VOUS AVEZ LES CORDONNÉES DE LA PERSONNE ET SI LE PAIEMENT EST BIEN VALIDÉ OU SI LE PRIX EST GRATUIT ET QUE VOUS N'AVEZ PAS BESOINS DES CORDONNÉES DE LA PERSONNE AVANT DE VALIDÉ VOTRE PROCESSUS.
- C'EST LÀ QUE RENTRE EN JEUX LES CONDITIONS. ILS VONT JUSTEMENT NOUS PERMETTRE DE DIRE À NOTRE APPLICATION SI LE PRIX N'EST PAS GRATUIT IL FAUT QUE J'AI REÇU LE PAIEMENT AVANT DE LAISSER L'UTILISATEUR TÉLÉCHARGER SON LIVRE.

# JAVA: LES STRUTURES CONDITIONNELLES

- L'instruction if dans sa forme basique ne permet de tester qu'une condition or la plupart du temps on aimerait pouvoir choisir les instructions à executer en cas de realization de la condition... L'expression if ... else permet d'executer une autre série en cas de non-realization de la condition.

Vous pouvez ajouter des conditions supplémentaires à l'aide de else if, voici la syntaxe :

```
if ( première condition ) {  
    }  
else if ( seconde condition ) {  
    }  
else{  
    }  
}
```

Noté que "else if" est sur deux mots. Si vous utilisez celui-ci en un seul mot, vos conditions ne marchera pas.

La condition if (si) ;  
La condition if... else (si... sinon) ;  
La condition if... elseif... else (si... sinon si...  
sinon).



# JAVA: SWITCH CASE

L'instruction *switch* permet de faire plusieurs tests de valeurs sur le contenu d'une même variable. Ce branchement conditionnel simplifie beaucoup le test de plusieurs valeurs d'une variable, car cette opération aurait été compliquée (mais possible) avec des *if* imbriqués. Sa syntaxe est la suivante :

```
switch      (Variable)      {  
    case      Valeur1:  
        Liste      d'instructions;  
        break;  
    case      Valeur2:  
        Liste      d'instructions;  
        break;  
    default:  
        Liste      d'instructions;  
        break; }  
}
```

Les parenthèses qui suivent le mot clé *switch* indiquent une expression dont la valeur est testée successivement par chacun des *case*. Lorsque l'expression testée est égale à une des valeurs suivant un *case*, la liste d'instruction qui suit celui-ci est exécuté. Le mot clé *break* indique la sortie de la structure conditionnelle. Le mot clé *default* précède la liste d'instructions qui sera exécutée si l'expression n'est jamais égale à une des valeurs.

# JAVA: LES BOUCLES: WHILE

- LES BOUCLES NOUS PERMETTENT DE RÉPÉTER UN CODE TANT QU'UNE CONDITION EST VRAIE OU QU'ON DISE À LA BOUCLE DE S'ARRÊTER. LA BOUCLE ARRÊTERA SON EXÉCUTION AUSSITÔT QUE LA CONDITION N'EST PLUS RESPECTÉE.
- EXPLICATION
  - DANS L'EXEMPLE, LA VARIABLE INDEX EST DÉCLARÉ AVEC UNE VALEUR DE 0. A CHAQUE PASSAGE DANS LA BOUCLE ELLE EST INCRÉMENTER ET AFFICHE UN ALERT. TANT QUE LA VALEUR DE INDEX EST INFÉRIEUR À 3 UN ALERT S'AFFICHERA À CHAQUE PASSAGE DANS LA BOUCLE. LA FONCTION ALERT SERA DONC EXÉCUTER 3 FOIS.



# JAVA: LES BOUCLES: FOR

- SIMILAIRE À LA BOUCLE WHILE, TOUTE LA PARTIE 'LOGIQUE' DE LA BOUCLE SE FAIT À UN ENDROIT CE QUI REND LE CODE PLUS FACILE À LIRE.

```
int somme =0;

for( int i =0; i <=10; i++){
    Somme++;
}
```

# JAVA: LES BOUCLES: DO WHILE

Cette structure est très proche de la structure while. Sa syntaxe est :

```
do{  
    instructions;  
    while (condition);
```

Dans cette boucle **faire\_tant\_que**, la condition est évaluée après l'exécution du corps de la boucle. Elle est au minimum exécutée une fois même si la condition à tester est fausse au départ

```
int i = 100, j = 0, somme = 0 ;  
do{  
    somme += j;  
    j++;}  
while (j <= i); //Si oublié génère une erreur de compilation
```

A la sortie de la boucle, la variable somme contient la somme des 100 premiers entiers.

# JAVA: FOREACH

- La boucle « for each » s'introduit en Java via le mot clé for, mais avec une syntaxe légèrement différente de celle utilisée pour le for traditionnel.
- Le « for each » permet principalement de parcourir des tableaux ou des collections pour en manipuler tous les éléments.

- Exemple:

```
int []notes = {15,10,18,16};
```

```
for(int indes : notes){  
    System.out.println(" forEach==:" + indes);  
}
```

# EXERCICES

- Reprendre les 3 exercices de la serie sur les tableaux:
  - Faire un programme permettant de vérifier si une année est bissextile ou pas.
  - Faire un programme permettant de vérifier si un nombre est premier ou pas.
- Exercice 3 : Soit le tableau suivant :  
Semaine = ['lundi', 'mar', 'mercredi', 'jeudi', 'vendredi', 'samedi', 'dimanc'] ;
  1. Retirer la dernière valeur du tableau
  2. Afficher les valeurs du tableau
  3. Ajouter la valeur la valeur 'dimanche' à la fin du tableau
  4. Remplacer le mar par mardi
  5. Afficher le nombre de valeurs du tableau
  6. Afficher la 5ème valeur

# JAVA: LES FONCTIONS

- On appelle fonction un sous-programme qui permet d'effectuer un ensemble d'instruction par simple appel de la fonction dans le corps du programme principal.
- Les fonctions permettent d'exécuter dans plusieurs parties du programme une série d'instructions, cela permet une simplicité du code et donc une taille de programme minimale. D'autre part, une fonction peut faire appel à elle-même, on parle alors de fonction récursive (il ne faut pas oublier de mettre une condition de sortie au risque sinon de ne pas pouvoir arrêter le programme...).
- Une méthode est une fonction faisant partie d'une classe. Elle permet d'effectuer des traitements sur (ou avec) les données membres.

# JAVA: LES MÉTHODES

- Ainsi, on peut avoir des types de données des int, String, Boolean, ...

Exemple:

```
Public static int carre(int nbre){  
    Return nbre * nbre  
}
```

- Si une fonction ou méthode ne retourne rien alors on doit bien le préciser avec void

```
Public static void carre(int nbre){  
    System.out.println("Ceci est un test");  
}
```

# JAVA: MOT-CLÉ STATIC

Une méthode statique appartient à une classe et une méthode non statique appartient à un objet d'une classe.  
Quand une méthode est statique, cela veut tout simplement dire à java qu'on veut un accès direct à la méthode.

Exemple d'une méthode statique:

```
public static void main(String[] args) {  
    //Il n'y a pas d'objet créer ici car afficher() est  
    une méthode statique  
    afficher();  
}  
  
public static void afficher(){  
    System.out.println("Appel de la méthode statique");  
}
```

Exemple d'une méthode non statique

```
public static void main(String[] args) {  
    Personne moussa = new Personne();  
    //L'objet est crée ici, car afficher() est une méthode  
    non statique  
    moussa.afficher();  
}  
  
public void afficher(){  
    System.out.println("Appel d'une méthode non  
    statique");  
}
```



# JAVA: MOT-CLÉ FINAL

- En langage Java, le mot-clé **final** indique qu'un élément ne peut être changé dans la suite du programme. Il peut s'appliquer aux méthodes et attributs d'une classe ainsi que la classe elle-même. Aussi, il peut s'appliquer sur les paramètres d'une méthode et sur les variables locales.

Exemple: on garantit qu'on ne peut plus modifier ni a, ni b dans les méthodes de calcul proposées.

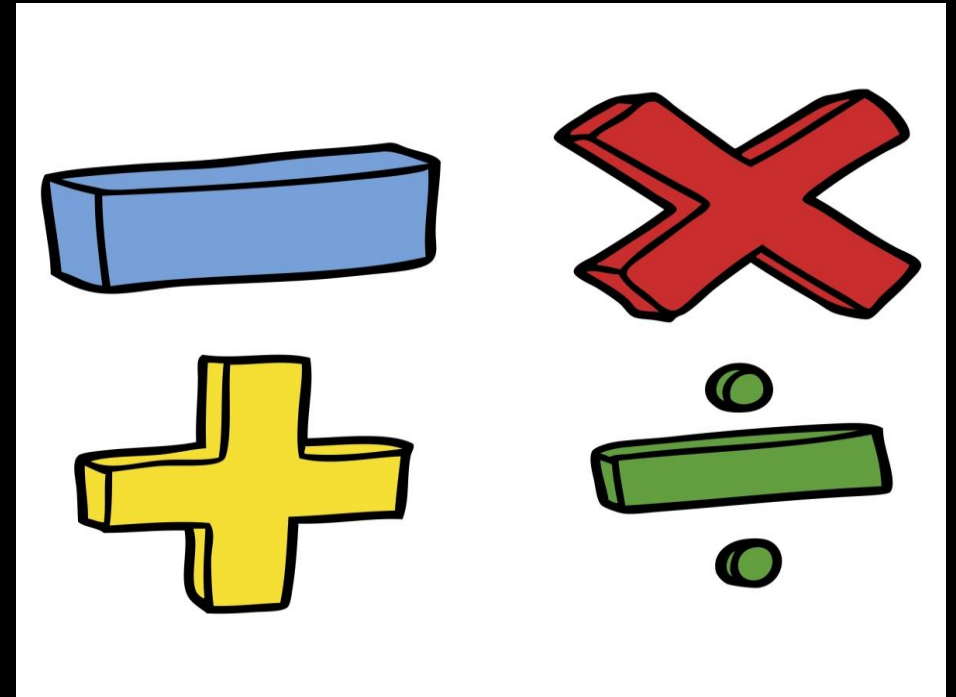
```
public static int addition(final int a, final int b){  
    return a + b;  
}
```

```
public static void test( final int value )  
{  
    while( value > 0 ) {  
        System.out.println( value-- );  
  
        // Ne compile pas!  
        //Car value est FINAL  
    }  
}
```

# EXERCICES - CALCUL

Faire des methodes:

- Addition
- Souscription
- Multiplication
- Division
- Modulo



# JAVA: MÉTHODES À NOMBRE DE VARIABLE DE PARAMÈTRE

- Nous avons jusqu'ici les fonctions sans ou avec paramètres. Quand on définit des paramètres d'une fonction, nous sommes obligés de mettre le nombre de paramètres prévu mais ça ... c'était avant
- En java, on peut définir des fonctions sans limite de paramètre ... et oui sans limite

```
public static String ma_fonction( int... values ) {  
    System.out.println( "1st value: " + values[0] );  
    return "Nb elements:"+values.length;  
}
```

# JAVA: CLASS JAVA.UTIL

- Le paquetage java.util contient des classes utilitaires, en particulier celles qui servent à manipuler les dates et les collections.
- java.util.Collection
- java.util.List
- java.util.Map: contient les **pairs** <clé, valeur>. Chaque pair est connu comme entrée. Map contient des éléments à clé unique.
- java.util.Iterator: est un patron de conception comportemental qui permet de parcourir une structure de données complexe de façon séquentielle sans exposer ses détails internes.
- java.util.Vector: permet de stocker des objets dans un tableau dont la taille évolue avec les besoins.
- java.util.Hashtable: implémente l'interface Map. Chaque liste est identifiée par sa clé donc elle permet de créer une collection d'objets associés à des noms. Elle est similaire à HashMap mais elle est synchronisée.

# JAVA: LES PACKAGES

- Java permet de regrouper les classes en ensembles appelés packages afin de faciliter la modularité.
- chaque paquetage porte un nom. Ce nom est soit un simple identificateur ou une suite d'identificateurs séparés par des points. L'instruction permettant de nommer un paquetage doit figurer au début du fichier source (.java) comme suit:

```
----- Fichier Exemple.java -----  
    package nomtest; // 1ere instruction  
    class MTest {}  
    public class Exemple { // code ...}  
----- Fin du Fichier -----
```



java.applet	Classes de base pour les applets
java.awt	Classes d'interface graphique AWT
java.io	Classes d'entrées/sorties (flux, fichiers)
java.lang	Classes de support du langage
java.math	Classes permettant la gestion de grands nombres.
java.net	Classes de support réseau (URL, sockets)
java.rmi	Classes pour les méthodes invoquées à partir de machines virtuelles non locales.
java.security	Classes et interfaces pour la gestion de la sécurité.
java.sql	Classes pour l'utilisation de JDBC.
java.text	Classes pour la manipulation de textes, de dates et de nombres dans plusieurs langages.
java.util	Classes d'utilitaires (vecteurs, hashtable)
javax.swing	Classes d'interface graphique

# JAVA

- La classe Scanner simplifie la lecture de données sur l'entrée standard (clavier) ou dans un fichier.
- Pour utiliser la classe Scanner, il faut d'abord l'importer :
  - `import java.util.Scanner;`
- Ensuite il faut créer un objet de la classe Scanner :
  - `Scanner sc = new Scanner(System.in);`



- Pour récupérer les données, il faut faire appel sur l'objet `sc` aux méthodes décrites ci-dessous.
- Ces méthodes parcourent la donnée suivante lue sur l'entrée et la retourne :
  - `String next()` donnée de la classe `String` qui forme un mot,
  - `String nextLine()` donnée de la classe `String` qui forme une ligne,
  - `boolean nextBoolean()` donnée booléenne,
  - `int nextInt()` donnée entière de type `int`,
  - `double nextDouble()` donnée réelle de type `double`

# EXAMPLE: SCANNER

```
public static void main (String [] args)
{
    System.out.println("Choisissez un nombre.");
    Scanner scanner = new Scanner(System.in);
    int choix = scanner.nextInt();

    System.out.println("Voici ton choix: " +choix);
}
```

# EXERCICE

- Faire un programme permettant à l'utilisateur de saisir un mot et verifie si ce dernier est un palindrome ou pas.
- PS: Utiliser Scanner.in

# CORRECTION PALINDROME

## Palindrome

```
package Hello;
import java.util.Scanner;
public class Palindrome {
    public static String inverse(String chaine) {
        String inversion = "";
        for (int i = 0; i < chaine.length(); i++) {
            inversion +=
                chaine.charAt(chaine.length() - i - 1);
        }

        return inversion;
    }
}
```

## Main.java

```
Scanner entree = new Scanner(System.in);
System.out.println("Indiquez la chaîne de caractères");
String chaine = entree.nextLine();

System.out.println(chaine);

String inserveChaine = Palindrome.inverse(chaine);
System.out.println("L'inverse de " + chaine + " est " +
    inserveChaine);

if (inserveChaine.equals(chaine))
    System.out.println(chaine + " est un palindrome");
else
    System.out.println(chaine + " n'est pas un palindrome");
}
```

# JAVA: LES OBJETS

Nous allons maintenant rentrer dans le vif du sujet à savoir la programmation orientée objet ou POO.

programmation orientée objet est un style de programmation qui s'articule autour d'objets renfermant des données et des mécanismes.

Son avantage:

- Elle traduit un univers complexe en de petits ensembles simples et autonomes (Objets),
- Elle permet de sécuriser nos données en définissant des règles d'accès et de modification,
- Elle permet de travailler à plusieurs sur un même projet composé de plusieurs briques.



# JAVA: EXEMPLE POO

- Supposons que nous voulons faire un gateau
  - On a besoin des elements de base un moule qui va permettre la creation du gateau
    - Le moule represente notre **classe**
    - Le gateau represente notre **objet**





# JAVA: POO - CLASSE

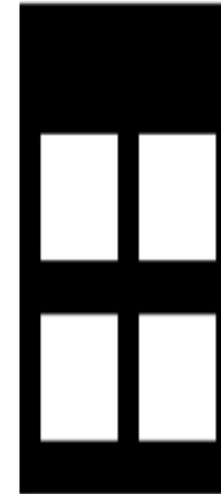
- La classe est la structure d'un objet. Il s'agit du plan de tout ce qui compose l'objet. La classe est composée de deux parties:

les attributs ( ou propriétés )  
les méthodes

- Les attributs sont les données associées à l'objet.
- les méthodes sont des fonctions qui sont associées à la classe.

Prenons un exemple concret...

Patron = Classe



titre
auteur
date
image
message

- Le mot-clé **class** sert simplement à créer la classe.
- Il est suivi du nom de la classe ainsi que du bloc d'instruction dans lequel sera spécifié l'ensemble des attributs et méthodes.

Exemple:

```
public class Personne {  
  
    public String nom;  
    public String prenom;  
  
    public Personne(String nom, String prenom) {  
        this.nom = nom;  
        this.prenom = prenom;  
    }  
  
    @Override  
    public String toString() {  
        return nom + " " + prenom ;  
    }  
}
```

# POO – INSTANCIER UNE CLASSE

- Instancier une classe, c'est se servir d'une classe afin qu'elle nous crée un objet. En gros, une instance est un objet.
- L'instanciation d'une classe est la phase de création des objets issus de cette classe.
- Lorsque l'on instancie une classe, on utilise le mot-clé *new* suivant du nom de la classe. Cette instruction appelle la méthode constructeur ( `__construct()` ) qui construit l'objet et le place en mémoire.
- EXEMPLE: **`$article = new Article();`**

# POO – ATTRIBUTS METHODES

- La classe est composée de deux parties: les attributs ( ou propriétés )et les méthodes.
- Les « attributs » (aussi appelés « données membres ») sont les caractères propres à un objet.

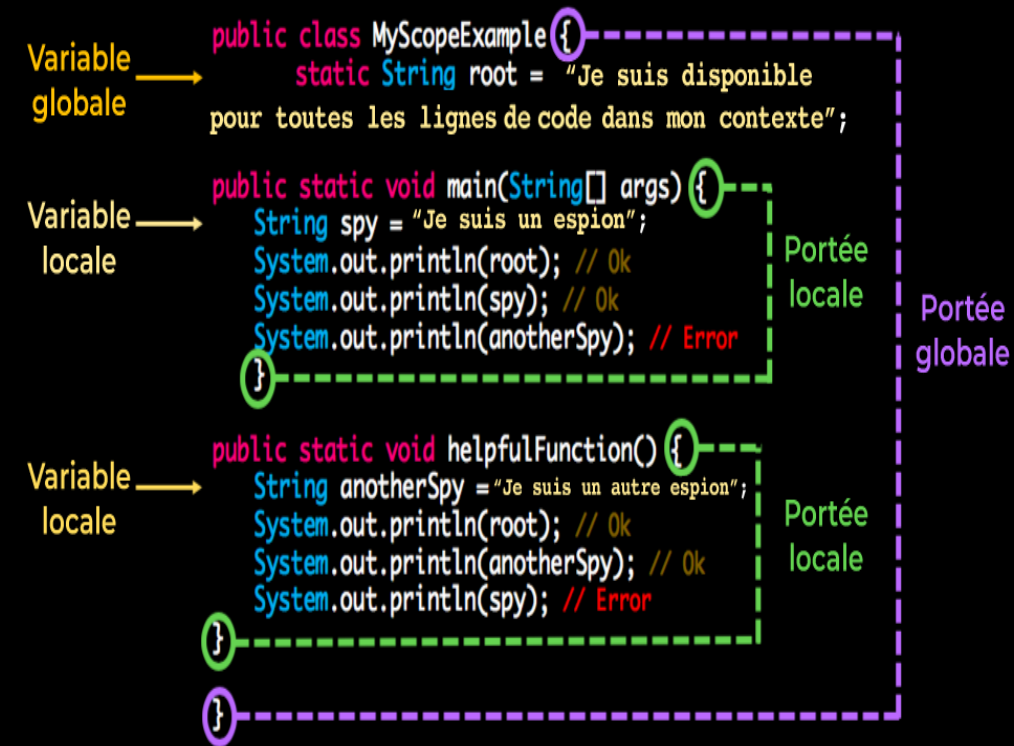
Une personne, par exemple, possède différents attributs qui lui sont propres comme le nom, le prénom, la couleur des yeux, le sexe, la couleur des cheveux, la taille...

- Les « méthodes » sont les actions applicables à un objet.

Un objet personne, par exemple, dispose des actions suivantes :  
manger, dormir, boire, marcher, courir..

# JAVA: LES PORTÉES DE VARIABLES

- En programmation informatique, une variable locale est une variable qui ne peut être utilisée que dans la fonction ou le bloc où elle est définie.
- La variable locale s'oppose à la variable globale qui peut être utilisée dans tout le programme.



# PRÉSENTATION DU TYPE ABSTRAIT

## PERSONNE

- Débutons notre apprentissage de la P.O.O. en établissant le cahier des charges du type abstrait Personne, représentant le concept d'un salarié d'une entreprise.
- Spécifions ce cahier des charges simplifié par la description minimale suivante :
  1. Une personne est décrite par deux informations qui la caractérisent :
    - son nom.
    - le nom de l'entreprise qui l'emploie.
  2. Une personne doit être capable de préciser qui elle est, en affichant ses informations, lorsqu'on le lui demande.

# PRÉSENTATION DU TYPE ABSTRAIT PERSONNE

- On distingue déjà, à la simple lecture de ce cahier des charges, deux types d'éléments liés à une personne :
  - ✓ Des caractéristiques physiques (les données) . C'est-à-dire la valeur de ses attributs.
  - ✓ Des caractéristiques comportementales. (les actions). C'est-à-dire les ordres auxquels une personne peut répondre lors de sollicitations extérieures. Ici, il s'agit pour une personne, de la capacité à s'afficher.



# LE TYPE PERSONNE EN JAVA

```
public class Personne {  
  
    //-----  
    // Les caractéristiques physiques  
    //-----  
    private String nom ;  
    private String societe ;  
  
    //-----  
    // Les caractéristiques comportementales  
    //-----  
  
    // 1- Construit un objet Personne de société inconnue et de nom correspondant au paramètre nom  
    public Personne (String nom) {  
        this.nom = nom.toUpperCase();  
        societe = "?"; // Par convention, la chaîne de caractères ? stipule que la personne n'est pas employée  
    }  
    public void integrerSociete (String entreprise) {  
        societe = entreprise ;  
    }  
  
    // 2 - Affiche les caractéristiques de la personne  
    public void afficher() {  
        System.out.print (« Je m'appelle » + nom) ;  
  
        if (societe.equals (« ? » ))  
            System.out.println (« et je ne suis pas salarié. »);  
        else  
            System.out.println (« et je travaille chez : » societe);  
    }  
  
} // class Personne
```

# EXEMPLES D'UTILISATION À PARTIR DU TYPE PERSONNE

```
Personne p1, p2, p3;
```

```
p1 = new Personne() ;// Refusé à la compilation
```

```
p2 = new Personne("Martin");// OK
```

```
p3 = new Personne(2); // Refusé à la compilation
```

```
p2.nom = "Dupond";// Refusé à la compilation
```

```
System.out.println("Je m'appelle"+ p2.nom); // Refusé à la compilation
```

```
p2.afficher();
```

```
// affiche « Je m'appelle MARTIN Je ne suis pas salarié. »
```

```
p2.integrerSociete("Java SARL"); //OK
```

```
p2.afficher();
```

```
// affiche « Je m'appelle MARTIN et je travaille chez Java SARL »
```

# JAVA: VERSION 1 DE LA CLASSE PERSONNE

Pour le cahier des charges N°1 ...

Une personne est décrite par deux informations :

son nom, enregistré en majuscules.

la société qui l'emploie.

Une personne doit être capable de s'identifier, en affichant les informations qui la caractérisent :



```
class Personne {
```

```
    // Deux variables d'instance (pour l'instant non protégées)
```

```
        public String nom;
```

```
        public String societe;
```

```
    // Une méthode
```

```
    public void afficher() {
```

```
        System.out.println ("Je m'appelle " + nom);
```

```
        System.out.println ("Je travaille chez " + societe);
```

```
    }
```

```
} // class Personne, version provisoire
```

Quelques problèmes résident dans le code ci-dessus :

On peut instancier (c'est-à-dire créer) des objets Personne indéterminés :

```
Personne p = new Personne() ;
```

On peut modifier, accidentellement ou non, les variables d'instance d'un objet Personne, une fois instancié.

```
p.nom = "durand" ; // nom en minuscules
```

On ne peut pas appliquer des règles de gestion comme par exemple, respecter la règle suivante : « Une Personne doit toujours avoir un nom qui ne peut être modifié; elle peut, en revanche, changer de société »

Tous ces problèmes, qui visent à apporter de la sécurité et garantir la cohérence des données, vont être petit à petit résolus, grâce aux mécanismes de la P.O.O. que nous allons mettre en œuvre.

# UN NOUVEAU CAHIER DES CHARGES POUR LE TYPE ABSTRAIT PERSONNE

Complétons et précisons les règles liées aux objets de type *Personne*

- ✓ 1 - Une personne est décrite par deux informations :  
son **nom** et la **société** qui l'emploie.
- ✓ 2 - Une personne a **toujours** un nom. Ce nom doit être invariable.  
C'est une chaîne de caractères, exprimés en **majuscules**.
- ✓ 3 - Une personne n'est **pas nécessairement associée** à une société : elle peut être considérée comme un indépendant ou comme une personne sans activité.
- ✓ 4 - Une personne doit pouvoir indiquer, lors de l'affichage, si elle est, ou non, salariée ( c'est à dire si sa société est identifiée et différente de notre convention « ? » ).
- ✓ 5 - Lorsqu'une personne est associée à une société, le nom de cette société ne peut excéder **30 caractères**, dans laquelle les lettres sont en majuscules.
- ✓ 6 - Une personne **peut changer de société** , ou perdre toute association à une société.
- ✓ 7 - Enfin, une personne est capable de **s'identifier**, en affichant les informations qui la caractérisent.

Toutes ces **nouvelles contraintes** vont être mises en œuvre grâce aux concepts de la **P.O.O.**

- Une règle d'or, absolue :

En **Programmation Orientée Objet**, l'accès aux informations doit être contrôlé.

Si la classe **Personne** définit deux variables d'instance **nom** et **societe**, le développeur qui utilise (après l'avoir créé) un objet **Personne** ne doit pas pouvoir affecter **sans contrôle** une nouvelle valeur à l'une ou l'autre de ces variables d'instances.

C'est la garantie de la cohérence de l'objet.




# JAVA: TYPE D'ACCÈS PUBLIC OU PRIVÉ ?

```
class Personne {  
  
    private String nom;  
    private String societe;  
  
    public void afficher() {  
        System.out.println(«Je m'appelle » + nom);  
        System.out.println(«Je travaille chez » + societe );  
    }  
} // class Personne, version provisoire
```

Les variables d'instance étant maintenant *private*, si on écrit :

```
Personne p = new Personne();  
    p.nom = «durand» ; // Impossible d'accéder au champ (private)
```

La compilation refuse la seconde instruction : *p.nom*, *privée*, est **inaccessible** pour l'utilisateur de la classe *Personne*.

- 
- Accès public ou private ?
  - **public:** Les variables et méthodes de la classe MaClasse, définies avec le modificateur public sont accessibles partout où MaClasse est instanciable.
  - **private:** Les variables et méthodes d'une classe MaClasse, définies avec private ne sont accessibles que dans la classe MaClasse.

# JAVA: LES DEUX ACTEURS DE LA PROGRAMMATION OBJET

- ✓ Le concepteur de la classe *MaClasse* :

Celui qui définit *MaClasse* : dans les méthodes de *MaClasse*, il a directement accès aux variables d'instance et aux méthodes privées.

- ✓ Un utilisateur de la classe *MaClasse* :

Celui qui instancie *MaClasse* : il n'a directement accès qu'aux variables et méthodes publiques de *MaClasse*.

# JAVA: CONTROLE DES INSTANCIATIONS

- ✓ L'instanciation par défaut

Avec la définition de la classe *Personne* de la page précédente, si nous exécutons :

```
Personne p=new Personne() ;  
p.afficher() ;
```

nous voyons s'afficher :

Je m'appelle *null*  
Je travaille chez *null*

- ✓ Quand le concepteur de la classe n'a pas spécifié de mécanisme d'instanciation explicite, *Java* en fournit un par défaut, appelé constructeur par défaut.
- ✓ Le *constructeur* est une méthode particulière en ce sens où elle porte le même nom que la classe à laquelle elle appartient.
- ✓ Le *constructeur par défaut* initialise chaque variable d'instance *vi* avec une valeur par défaut :
  - *null* si *vi* est de type objet,
  - valeur par défaut du type *type* si *vi* appartient à un *type* primitif.

# JAVA: DÉFINIR UN CONSTRUCTEUR

```
• class Personne {  
    private String nom;  
    private String societe;  
  
    public Personne (String  
    patronyme) {  
        nom = patronyme.toUpperCase();  
        // Cahier des charges, règle 2  
    }  
    //.... cf page précédente  
    // ....  
} // class Personne
```

*Personne (String  
Personne*

```
Personne p = new  
Personne(«durand») ;
```

```
Personne p = new Personne() ;
```

*// Erreur de compilation : il n'y a plus de constructeur par défaut !!*

**NB** : En **Java**, tous les objets instanciés sont créés dans le tas.

# JAVA: VERSION2 DE LA CLASSE PERSONNE

```
class Personne {  
    private String nom;  
    // Convention : l'absence de société sera matérialisée par '?'  
    private String societe;  
  
    // Construit un objet Personne de nom invariable et de societe inconnue  
    public Personne ( String nom ) {  
        // Qualification avec this pour distinguer la v.i. du paramètre  
        this.nom = nom.toUpperCase();  
        societe = new String(«?»);  
    }  
  
    public void afficher() {  
        System.out.println ( "Je m'appelle " + nom );  
        if ( societe.equals( «?» ) )  
            System.out.println( "Je ne suis pas salarié" );  
        else  
            System.out.println ( "Je travaille chez " + societe );  
    } // class Personne, version 2
```



# JAVA: UTILISATION DE THIS

Dans un **constructeur**, le mot-clef **this** désigne l'objet en phase de construction.

Dans une méthode, ce mot-clef désigne l'objet qui traite le message :

```
// Une autre méthode de la classe Personne  
public void affecter( Personne personne ) {  
    this.societe = personne.societe;  
    // Ici this est facultatif  
}
```

Le mot clé **this** permet de désigner l'objet dans lequel on se trouve, c'est-à-dire que lorsque l'on désire faire référence dans une fonction membre à l'objet dans lequel elle se trouve, on utilise this.

Grâce à cette variable spéciale, il est possible dans une fonction membre de faire référence aux propriétés situées dans le même objet que la fonction membre.

# JAVA: RÉCAPITULONS

- ✓ Le constructeur par défaut est encore appelé constructeur sans paramètres ou bien encore constructeur implicite.
- ✓ Un constructeur est une méthode particulière - en général *public* - et dont l'identificateur est le même que celui de la classe et qui est toujours définie sans type de renvoi (même pas *void*).
- ✓ Dès qu'un constructeur explicite est défini, le constructeur par défaut n'est plus disponible, sauf si le développeur le rétablit en définissant explicitement un constructeur sans paramètres.

Le constructeur garantit que l'objet va être dans un état initial satisfaisant dès sa création, puisqu'il permet la valorisation des variables d'instance de tout objet.

# JAVA: RÉCAPITULONS

- ✓ Un objet est responsable des tâches qu'il exécute, et lui seul.
- ✓ Grâce à l'encapsulation, on ne permet pas l'accès direct aux champs d'un objet ( qui peut être vu comme une boîte noire ).
- ✓ Les objets possédant les mêmes propriétés (données et comportements) sont décrits par une même classe.
- ✓ En Programmation Orientée Objets (P.O.O.) , on cherche à bâtir des objets ( et donc des classes ) spécialisés en cherchant à favoriser l'utilisation de classes déjà existantes.

On veillera à ce qu'une classe soit la plus indépendante possible, tout en évitant qu'elle n'implémente trop de fonctionnalités. La conception, le débogage et la réutilisabilité s'en trouvant nettement plus favorisés.

# JAVA: DÉFINISSONS DES ACCESSEURS

## ✓ Comment changer de société ?

Nous n'avons pour l'instant **aucun moyen** d'associer une société à une personne puisque la variable d'instance **societe** est **privée**.

Il faut donc définir de nouvelles **méthodes publiques** dans la classe qui permettront de **modifier** et **d'afficher** la société d'une personne.

## ✓ Créer deux méthodes supplémentaires

```
// Accès en consultation
public String lireSociete() {
    return societe;
}
// Accès en modification
public void changerSociete(String entreprise) {
    // Attention au cahier des charges !!
    societe = entreprise;
}
```

Les méthodes *lireSociete* et *changerSociete* sont respectivement des accesseurs en lecture (*accessor*) et en écriture (*mutator*) de la variable d'instance *societe*.



Gérer l'accessibilité des variables *privées*

On peut **limiter** l'accès aux données à la lecture (avec uniquement un **accesseur en consultation**) ou **étendre** l'accès en lecture/écriture (avec deux accesseurs en **consultation** et en **modification**).

Gérer l'intégrité des données

L'**accesseur en modification** comporte **souvent du code de contrôle** qui permet de valider la nouvelle valeur de la variable.

# JAVA: UNE CLASSE PERSONNE ROBUSTE

```
Améliorons notre classe Personne :
public String lireNom() { // Accès en consultation
    return nom;
}
public String lireSociete() { // Accès en consultation
    return societe;
}

public void quitterSociete() {
    if (societe.equals(" ? »)) { // La personne n'est pas rattachée à une société
        afficher (); // on décide d'arrêter l'application
        System.out.println ("Impossible de quitter la société");
        System.exit(1); // Arrêt de l'exécution, code erreur 1
    }
    societe = " ? » ; // Ici, il y a bien une société à quitter, on applique la convention
}

// Méthode-filtre : renvoie le paramètre nomSociete s'il représente un nom de société
// acceptable selon les règles établies
private String validerSociete(String nomSociete ) {
    if (nomSociete.length() > 30 || nomSociete .equals("?")) {
        // En Java, || représente l'opérateur logique OU
        System.out.println(« Classe Personne, société incorrecte : »+ nomSociete );
        System.exit(2); // Arrêt exécution,
        // Ici, on est sûr que nomSociete est valide : on le retourne
        return nomSociete ;
    }
}

public void affecterSociete(String entreprise) {
    // Avant d'aller dans une société, il faut avoir quitté la précédente
    if ( ! societe.equals(" ? ») ) {
        afficher();
        System.out.println ("Erreur : 1- quitterSociete , puis 2-affecterSociete");
        System.exit(1);
    }
    societe = validerSociete( entreprise ).toUpperCase();
}
```

# JAVA: UN OU PLUSIEURS CONSTRUCTEURS ?

- ✓ Revenons sur le premier constructeur

```
public Personne (String nom) {  
    // Construit 1 objet Personne de nom invariable et de societe inconnue  
    this.nom = nom.toUpperCase() ;  
    societe = new String («?» ) ;  
}
```

Pour instancier l'individu *Dupont*, de la société **Java SARL**, il faut exécuter :

```
Personne dupont = new Personne (« Dupont ») ;  
dupont.affecterSociete («Java SARL») ;
```

- On peut légitimement souhaiter faire cette instanciation en une seule opération.



- ✓ Un deuxième constructeur

```
public Personne (String nom, String entreprise) {  
    // Construit un objet Personne de nom fixe et de societe connue  
    this.nom = nom.toUpperCase();  
    societe = validerSociete(entreprise).toUpperCase() ;  
}
```

- ✓ Notion de signature

Pour le compilateur, il n'y a pas d'ambiguïté entre les deux constructeurs. Ainsi, l'exécution de :

```
new Personne(«Dupont», «Java SARL»);
```

fait appel au **deuxième constructeur** car celui-ci utilise deux chaînes en paramètre.

Plus généralement, la **signature** d'un constructeur ou d'une méthode comprend son **identificateur**, la **liste** et les **types** de ses **paramètres**.

Le compilateur détermine **la méthode à exécuter** en **fonction** de sa **signature**.  
Des **méthodes différentes** peuvent porter le même nom, à partir du moment où **leur signature respective diffère**.

### ✓ Exemple d'utilisation

La classe **System** fournit une **variable de classe** publique, **out**, que l'on exploite dans l'instruction suivante :

```
System.out.println(« Utilisation du flux de sortie »);
```

```
// Elle fournit également une méthode de classe publique : exit :  
System.exit(0) ;
```

# JAVA: RÉCAPITULATIONS

## ✓ Définir une classe

- Variables d'instances généralement **privées**.
- Instanciation :
  - ◆ Pas de constructeur : Initialisation par défaut.
  - ◆ Un ou plusieurs constructeurs **explicites** : plus de constructeur par défaut (sauf si un constructeur est défini sans arguments).
  - ◆ Comme pour toutes les méthodes, il peut **exister plusieurs constructeurs**. Il s'agit là à la technique de surcharge.
- Méthode d'instance
  - ◆ L'objet est un paramètre implicite de la méthode, accessible si nécessaire via la notation **this**.
- Variables et méthodes de classe
  - ◆ définies avec le modificateur **static**.
  - ◆ une méthode de classe ne dispose pas de la référence **this**.

# JAVA: RÉCAPITULATIONS

## ✓ Conventions d'écriture

Dans la pratique professionnelle, on utilise des règles de nommage consistant à utiliser des **verbes à l'infinitif** pour les méthodes : *"quitterSociete(...)"*, *"afficher()"*, *lireNom()*, .....

De même, pour les **accesseurs**, on adopte les conventions suivantes : les préfixes **get** et **set** : **getNom** au lieu de *lireNom*, **setSociete** au lieu de *integrerSociete*.

- Cette convention est celle attendue par la technologie des **JavaBeans**, pour retrouver dynamiquement ces accesseurs et valoriser les **v.i.s.**

# JAVA: VERSION FINALE DE LA CLASSE PERSONNE

```
class Personne {  
    // La variable de classe matérialisant le non rattachement à  
    une // société selon notre convention .  
    private static final String PAS_DE_SOCIETE = "?";  
    // Les variables d'instance de type String  
    private String nom;  
    private String societe;  
  
    private String validerSociete(String entreprise) {  
        // .....  
    }  
    //-----  
    // Deux constructeurs pour instancier  
    //-----  
    public Personne (String nom) {  
        // Construit un objet Personne de societe inconnue  
        this.nom = nom.toUpperCase();  
        societe = PAS_DE_SOCIETE;  
    }  
    public Personne (String nom, String societe) {  
        // Construit un objet Personne de nom et  
societe connus // ...  
    }  
    //-----  
    // Accesseurs en consultation  
    //-----  
    public String getNom() {  
        // ...  
    }  
    public String getSociete() {  
        // ...  
    }  
}
```

```
//-----  
// Accesseur en modification  
//-----  
public void setSociete(String entreprise) {  
    // ...  
}  
public boolean etreSalarie() {  
    return ! societe.equals(PAS_DE_SOCIETE);  
}  
public void quitterSociete() {  
    // ...  
}  
//-----  
// Afficher les caractéristiques d'un objet  
//-----  
public void afficher () {  
    System.out.println (« Je m'appelle » + nom);  
    if ( ! etreSalarie() )  
        System.out.println (« Je ne suis pas employé  
d'une entreprise »);  
    else  
        System.out.println (« Je travaille chez » +  
societe);  
}  
  
} // class Personne, dernière version
```

# JAVA: ECRITURE ET EXÉCUTION D'UN PROGRAMME JAVA

- Deux types d'applications
  - les applications autonomes (stand-alone), exécutées via un appel au système d'exploitation.
  - les applets, exécutées sous le contrôle d'un navigateur Web, ou par l'intermédiaire de l'appletViewer, les servlets exécutées par un serveur Web.
- Structure d'une application stand-alone
  - Exemple: Le programme Personne.java

```
class Personne { // Définition des variables
// Définition des méthodes
public static void main(String args[]) {
    // Méthode principale appelée au début de l'exécution
}
} // ... class Personne
```

Personne.java

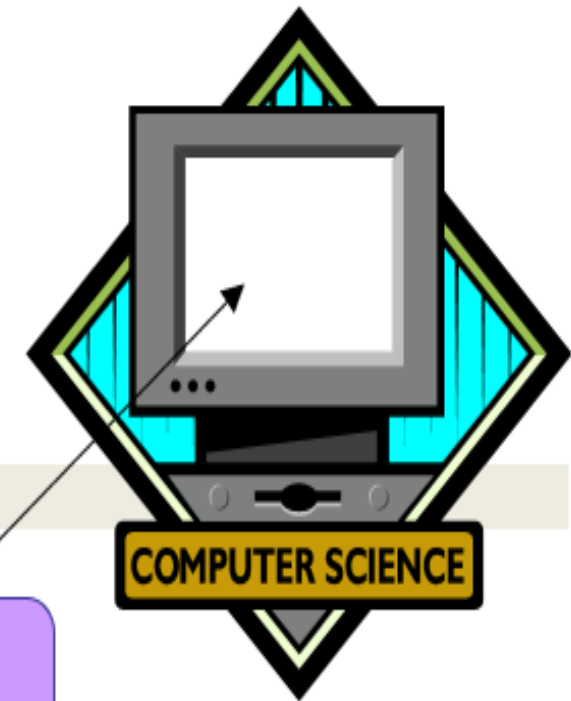
```
class Personne  
{...};
```

Personne.class

bytecode

COMPILATION

EXECUTION





# JAVA: UNE CLASSE USUELLE: LA CLASSE STRING

- ✓ La classe *String* permet la manipulation de chaînes caractères.
- ✓ Les objets *String* sont constants.

```
String uneChaine, uneAutre;  
uneChaine = « Voici une chaîne de caractères »;  
uneAutre  = « en voilà une autre ... »;  
String encoreUneAutre = new String (« Je suis une chaîne ») ;
```

- La méthode *length()* fournit la longueur d'une chaîne de caractères. La méthode *charAt()* renvoie un caractère de rang donné.

```
String uneChaine = « Voici une chaîne de caractères » ;  
int longueur= uneChaine.length() ;  
char caractere = uneChaine.charAt( 0 ); // caractere prend la valeur 'V'
```

- La méthode *equals* permet de comparer deux chaînes.

```
String chaine1 = « Voici une chaîne » ;  
String chaine2= « et une autre »;  
if ( chaine2.equals( chaine1 ) ) [ ... ] else [ ... ]
```

# JAVA: AUTRES MÉTHODES

```
String chaine1 = « Voici une chaîne »;  
String chaine2 = chaine1.substring(2,5);
```

```
// Extraction de sous-chaîne : chaine2 vaut "ici"  
// ch.substring( deb, fin+1) pour obtenir les caractères de rang ' deb à fin'
```

```
String chaine3 = chaine1.trim().toUpperCase();  
// Ecrémage puis majuscule : chaine3 vaut "VOICI UNE CHAINE »
```

```
String chaine4 = chaine1.replace( 'i', ' ?' ) ;  
// Remplacement des occurrences d'un caractère : chaine4 vaut «"Vo?c? une chaîne »
```

```
int rangDeH = chaine1.indexOf( 'h' );  
// rangDeH vaut 11, ( -1 si la lettre était absente de chaîne1 )
```

```
String chaine5 = String.valueOf( 20.6 );  
// Représentation d'une valeur numérique sous forme de chaîne ( méthode de classe )
```

# JAVA: IMPORTATIONS

## ✓ Les packages

Des bibliothèques de classes appelées **packages** standard peuvent être utilisées afin d'accéder à des classes d'utilité générale :

- ▶ ***java.lang*** : *String, StringBuffer, Math, System ...*

Ce package est importé par défaut.

- ▶ ***java.util, java.net, java.awt***, qui doivent être importés explicitement.

## ✓ Instruction d'importation

```
import java.util.Date ; // Permet l'utilisation de la classe Date .  
import java.util.* ;  
import java.io.* ;
```

// Permet l'utilisation de toutes les classes des 2 packages util et io.

Si une classe n'est pas trouvée, elle sera recherchée dans les **packages importés**.

Les instructions d'importation doivent se trouver en tête du fichier.

# JAVA: IMPORTATIONS

L'accès aux classes peut se faire **sans importation** mais au prix d'une **plus grande complexité d'écriture** :

```
java.util.Date aujourd'hui = new java.util.Date() ;
```

au lieu de :

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

```
...
```

```
Date uneDate = new Date() ;
```

# JAVA: TP

- Écrivez une classe représentant une ville.
- Elle doit avoir les propriétés nom et département et une méthode affichant « la ville X est dans le département Y ».
- Créez une ville et département par défaut Paris
- Créez des objets ville, affectez leurs propriétés, et utilisez la méthode d'affichage.
- Remplacer les informations de la ville par défaut par Lyon
- Afficher la nouvelle ville

CORRECTION EN LIVE – DEPARTMENT-VILLE



# JAVA: TP

- Mini Tp – Exo1 -----
  - Créer une classe personne avec:
    - les attributs nom,prenom,age
    - Les méthodes: getPrenom, getNom, getAge, setprenom, setNom,setAge
  - Afficher le prenom, nom et age de la personne



# EXERCICE LA TABLE DE MULTIPLICATION

- Faire un programme qui demande à l'utilisateur de saisir un chiffre et lui retourne sa table de multiplication



# CORRECTION

```
public static void main (String [] args)
{
    System.out.println("Choisissez un nombre.");
    Scanner scanner = new Scanner(System.in);
    int choix = scanner.nextInt();

    System.out.println("Voici ton choix: " +choix);

    for(int index=0; index<=10; index++) {
        System.out.println(choix+"*"+index+"="+ (choix*index));
    }
}
```



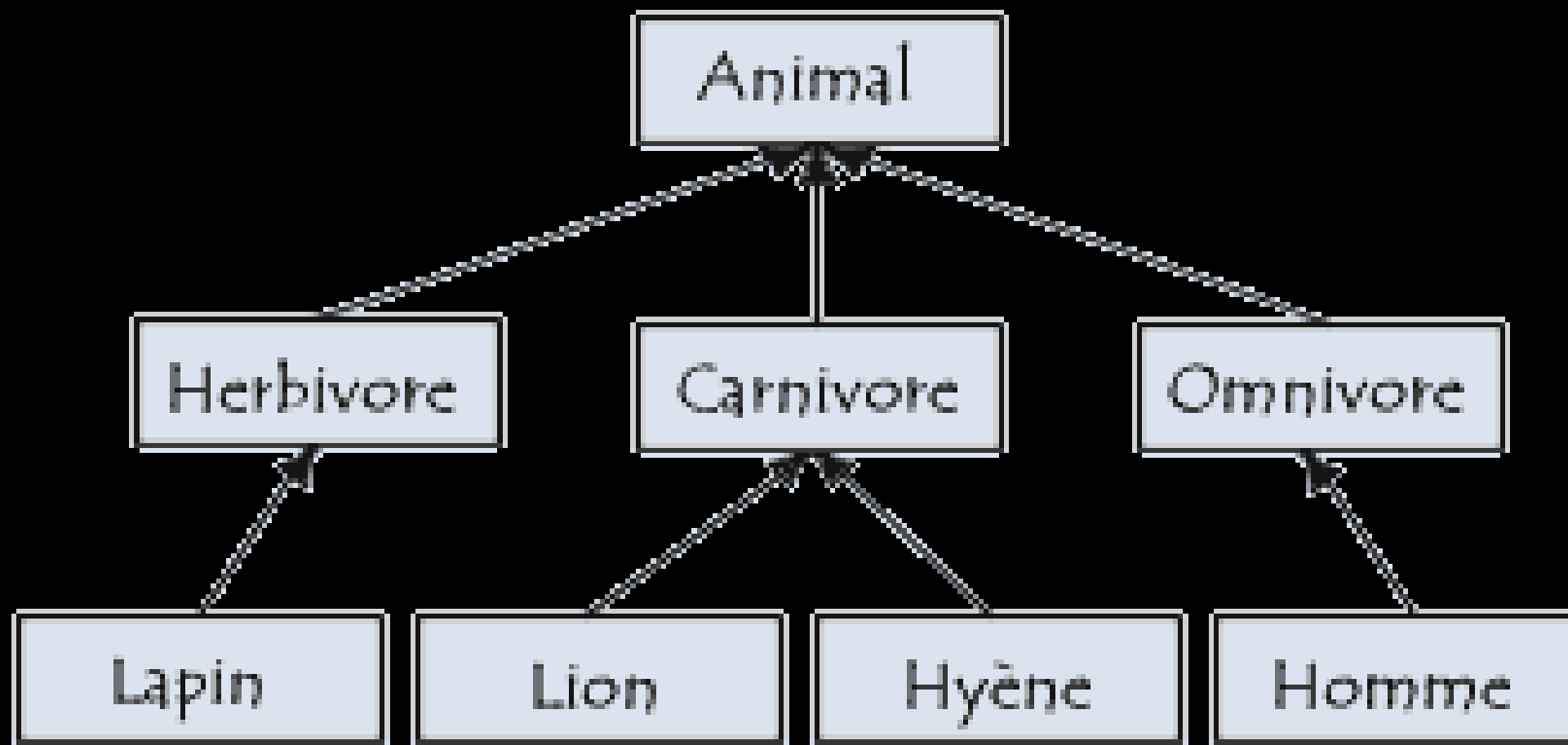
# TP – LIVRE ET LISTE ÉTUDIANTS



# JAVA: L'HERITAGE

- L'héritage est un mécanisme qui facilite la réutilisation du code et la gestion de son évolution. Grâce à l'héritage, les objets d'une classe ont accès aux données et aux méthodes de la classe parent et peuvent les étendre. Les sous classes peuvent redéfinir les variables et les méthodes héritées. Pour les variables, il suffit de les redéclarer sous le même nom avec un type différent. Les méthodes sont redéfinies avec le même nom, les mêmes types et le même nombre d'arguments, sinon il s'agit d'une surcharge. L'héritage successif de classes permet de définir une hiérarchie de classe qui se compose de super classes et de sous classes.
- Une classe qui hérite d'une autre est une sous classe et celle dont elle hérite est une super\_classe.
- Une classe peut avoir plusieurs sous classes. Une classe ne peut avoir qu'une seule classe mère

# EXAMPLE



# JAVA: L'HERITAGE

- Syntaxe

```
class Mere{
```

```
}
```

```
class Fille extends Mere{
```

```
}
```

# JAVA:HERITAGE

## Class Parent

```
package Heritage;

public class Parent {
    public String parent(){
        String mere=" Je suis la
classe Mère.";
        return mere;
    }
}
```

## Class Fille

```
package Heritage;

public class Fille {
    public String fille(){

        return " Je suis la classe
fille.";
    }
}
```



# POO – SURCHARGE DE METHODE

- On a créé une classe fille qui hérite de la classe mère avec la méthode parent(). Si on crée une autre méthode parent() dans la classe fille celle-ci va écraser la méthode parent qui est appelée dans la fille.
- Pour utiliser ces deux méthodes avec le même nom, on peut surcharger les méthodes.
- Ainsi la méthode parent de la classe mère est combinée dans la classe fille avec le mot-clé **super**

# JAVA: SUPER

- Le mot clé super est utilisé afin d'appeler les méthodes de la classe parente appelé aussi superclass.
- Il est également souvent utilisé dans la fonction constructeur de l'enfant pour appeler la fonction de constructeur parent.

```
package Heritage;

public class Animal {

    public String identifier()
    {
        return "Je suis un animal";
    }
}
```

```
public class Chat extends Animal{

    public String identifier()
    {
        return super.identifier() +"Je
suis un chat";
    }
}
```

# EXERCICE

- Mini tp
  - Créer des types d'animaux (chat, chien, lion,...) qui héritent de la classe animal
  - Chaque animal a son caractérisque:
    - le chat miaou
    - le chien ouaf
    - le lion rugi
- Afficher le caractérisque

# CORRECTION

```
package Heritage;

public class Chat extends Animal{

    public String identifier()
    {
        return super.identifier() +"Je suis un chat";
    }

    public String crier()
    {
        return "Miaou";
    }
}
```

```
package Heritage;

public class Chien extends Animal {

    public String identifier()
    {
        return super.identifier() +"Je suis un
chien";
    }

    public String crier()
    {
        return "Ouaf ouaf";
    }
}
```

# JAVA: ENCAPSULATION

- Le principe d'encapsulation correspond tout simplement à la façon dont on définit la visibilité et l'accessibilité de nos propriétés et méthodes de classe.
- L'encapsulation est, avec l'héritage, l'un des concepts fondamentaux de la programmation orientée objet.
- On va pouvoir définir trois niveaux d'accessibilité différents pour nos propriétés et méthodes grâce aux trois mots clefs suivants :
  - public
  - private
  - protected

# JAVA: ENCAPSULATION

```
private int age;  
  
private String nom;  
  
private String prenom;  
  
protected String test_change;
```

```
public static void main(String[] args) {  
    // TODO Auto-generated method stub  
    Encaps test = new Encaps();  
  
    //System.out.println(test.nom());  
    test.setNom("Homer");  
    System.out.println(test.getNom());  
  
    test.test_change = "Moussa";  
    System.out.println(test.test_change);  
}
```



**EXERCICE**

TP - HERITAGE

EXERICICE-heritage.



# JAVA: POO - CLASS ABSTRACT

- En programmation orientée objet (POO), une classe **abstraite** est une classe dont l'implémentation **n'est pas complète** et qui **n'est pas instanciable**. Elle sert de base à d'autres classes dérivées (héritées).
- Une classe abstraite c'est un peu vague. On a défini des attributs et méthodes dans notre classe mais on ne sait pas forcément comment ça va être géré
- Le mécanisme des classes abstraites permet de définir des comportements (méthodes) dont l'implémentation (le code dans la méthode) se fait dans les classes filles. Ainsi, on a l'assurance que les classes filles respecteront le contrat défini par la classe mère abstraite. Ce contrat est une interface de programmation.

```
package Abstract;

abstract class Personne {

    abstract protected String manger();

    abstract protected String marcher();

    abstract protected String dormir();
}
```

```
package Abstract;

public class humain extends Personne {

    public String marcher() {
        return "Je suis la classe marcher";
    }

    public String manger() {
        return "Je suis la classe manger";
    }

    public String dormir() {
        return "Je suis la classe dormir";
    }
}
```

# JAVA: LES INTERFACES

- En programmation orientée objet, une **interface** est un ensemble de signatures de méthodes publiques d'un objet.
- Il s'agit donc d'un moyen d'imposer une **structure** à nos classes, c'est-à-dire d'obliger certaines classes à implémenter certaines méthodes.
- Techniquement, une interface est une classe entièrement abstraite. Son rôle est de décrire un comportement à notre objet. Les interfaces ne doivent pas être confondues avec l'héritage : **l'héritage représente un sous-ensemble**
- (exemple : un magicien est un sous-ensemble d'un personnage). Ainsi, une voiture et un personnage n'ont aucune raison d'hériter d'une même classe.
- Par contre, une voiture et un personnage peuvent tous les deux se déplacer, donc **une interface représentant ce point commun** pourra être créée.

# JAVA: LES INTERFACES

```
package Interface;

public interface Article {

    public String getTitre();
    public String getAuteur();
    public String getResume();
    public String getContenu();

}
```

```
package Interface;

public class main {

    public static void main(String[] args) {
        // TODO Auto-generated method stub

        BlogArticle blog = new BlogArticle();

        System.out.println(blog.getAuteur());

    }

}
```

```
package Interface;

public class BlogArticle implements Article {

    public String getTitre(){
        return "Cool le P00";
    }

    public String getAuteur(){
        return "Moussa Camara";
    }

    public String getResume(){
        return "Ceci est le résumé";
    }

    public String getContenu(){
        return "Ceci est un contenu ...coco...";
    }

}
```

# JAVA: L'HERITAGE

- Une classe n' étend pas une interface, elle l'**implémente**.
- On utilise pour cela le mot-clé **implements**.
- Une classe peut implémenter autant d'interfaces que l'on veut. Une classe concrète doit obligatoirement fournir une implémentation pour toutes les méthodes déclarées par toutes les interfaces qu'elle implémente, soit elle-même, soit une de ses super classes.

# DIFFÉRENCE ENTRE ABSTRACT ET INTERFACE

## Abstraite

- Une classe Abstraite:
  - C'est une classe qui ne s'instancie pas permet de factoriser du code
  - Une classe qui ne peut qu'étendre d'une et une seule classe

**Exemple:**

**Class test extends Articles**

## Interface

- Une Interface:
  - Represente en quelques sortes une API
  - Fournit un contrat de services
  - Une classe qui peut implementer plusieurs interface

**Exemple:**

**Class test implements Articles, User, Comments**

# DIFFERENCE ENTRE HÉRITAGE ET IMPLÉMENTATION

## Classes abstraites et Interfaces : A ne pas confondre

L'héritage représente un sous-ensemble, par exemple :

- Un chien est un animal (Chien hérite d'Animal)
- Un Airbus est un avion (Airbus hérite d'Avion)

Un avion et un animal n'ont pas de raison d'hériter de la même classe, cependant les deux peuvent se déplacer, il est possible de mettre en place une interface possédant ce point commun.



# DIFFERENCE ENTRE HÉRITAGE ET IMPLÉMENTATION

- **Implements** (en français : met en oeuvre) permet d'implémenter une interface dans une classe, toutes les méthodes de l'interface doivent être redéclarées dans la classe.
- **Extends** permet de faire hériter deux classes ou deux interfaces. (Contrairement aux classes, l'héritage multiple entre interfaces est possible en les séparant par des virgules).

# JAVA: EXEMPLE : HERITAGE ET INTERFACE

- EXEMPLE : Heritage et interface

```
public interface Motorise {  
    // notre interface  
    public void faisLePlein() ;  
}
```

```
public class Transport {  
    // une instance de Transport ne sait pas toujours  
    // faire le plein  
    public void roule() {}  
}
```

```
public class Voiture extends Transport implements Motorise {  
    public void conduit() {}  
    public void faisLePlein() {}  
}
```

```
public class Avion extends Transport implements Motorise {  
    public void vole() ;  
    public void faisLePlein() {}  
}
```

```
public class Velo extends Transport {  
    // ne sait pas faire le plein  
    public void pedale() ;  
}
```

# JAVA: TP

- Créer une classe Véhicule
- 2 classes qui en héritent : **Voiture** et **Moto** qui vont contenir des méthodes pour retourner :
  - le nombre de roues (lié au type de véhicule)
  - le type de carburant (essence ou diesel)
  - la vitesse max
- Ajouter un constructeur
- Instancier un véhicule de chaque type
- Ajouter 2 attributs contenanceReservoir et contenuReservoir
- Créer une classe Pompe (à essence) avec 3 attributs : typeCarburant, contenance et contenu
- Dans Vehicule, ajouter une méthode fairePlein() qui prend une Pompe en paramètre, qui remplit le reservoir du Vehicule et enlève autant d'essence à la Pompe
- \*\*\*\* bonus :
  - vérifier qu'on fait le plein à une pompe du même type de carburant que le véhicule

# JAVA – CORRECTION TP EN LIVE



# JAVA - POLYMORPHISME

- Le polymorphisme dans java veut simplement dire qu'une classe peut prendre plusieurs formes et c'est d'autant plus vrai avec les classes qui hérite d'une classe supérieure.
- Le polymorphisme peut être vu comme la capacité de choisir dynamiquement la méthode qui correspond au type réel de l'objet

# JAVA- POLYMORPHISME

- Le polymorphisme consiste à exploiter cela en fournissant un B dans les expressions "qui attendent" un A.

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

```
Voiture pv = new Polo();  
Voiture fv = new Ford();  
Voiture mv = new Moto();
```

```
System.out.println(pv.toString());  
System.out.println(fv.toString());  
System.out.println(mv.toString());  
}
```

```
public class Polo extends Voiture {  
    private final int roues = 4;
```

```
    public Polo() {  
        super();  
    }
```

```
public class Ford extends Voiture {  
    private final int roues = 4;
```

```
    public Ford() {  
        super();  
    }
```



# JAVA – EXERCICE - VEHICULE

- Exo:
- Ecrire les classes nécessaires au fonctionnement du programme suivant (en ne fournissant que les méthodes nécessaires à ce fonctionnement).

```
public class TestMetiers {  
    public static void main(String[] argv) {  
        Personne[] personnes = new  
            Personne[3];  
        personnes[0] = new Menuisier("Paul");  
        personnes[1] = new Plombier("Jean");  
        personnes[2] = new  
            Menuisier("Adrien");  
        for (int i = 0; i < personnes.length;  
            i++)  
            personnes[i].affiche();  
    }  
}
```



# JAVA: LES EXCEPTIONS

- Les exceptions représentent le mécanisme de gestion des erreurs intégré au langage Java. Il se compose d'objets représentant les erreurs et d'un ensemble de trois mots clés qui permettent de détecter et de traiter ces erreurs (try, catch et finally ) mais aussi de les lever ou les propager (throw et throws).
- Lors de la détection d'une erreur, un objet qui hérite de la classe Exception est créé (on dit qu'une exception est levée) et propagé à travers la pile d'exécution jusqu'à ce qu'il soit traité.
- Ces mécanismes permettent de renforcer la sécurité du code Java.

# JAVA: LES MOTS CLÉS TRY, CATCH ET FINALLY

```
try {  
  
    // Bloc d'instructions pouvant déclencher une erreur  
  
} catch ( Exception exception ) {  
  
    // Bloc d'instructions pour traiter une éventuelle exception  
  
} finally {  
  
    // Bloc d'instructions à exécuter dans tous les cas,  
    // que le try se soit exécuté en succès ou en erreur.  
  
}
```

# JAVA: LES EXCEPTIONS

- Le bloc **try** permet de définir un ensemble d'instructions à surveiller : la traduction du mot try en français est « essayer ». Ce bloc est obligatoire. Il ne peut pas y avoir de bloc catch (ou finally) sans bloc try associé et les accolades y sont obligatoires.
- Si une exception est déclenchée durant l'exécution du bloc try, il sera alors suspendu et l'exécution se poursuivra dans un bloc catch associé ou dans le bloc finally.
- Il peut y avoir un ou plusieurs blocs **catch** associé à un bloc try. Chaque bloc catch est associé à un type d'erreur (une classe d'exception) à traiter : c'est ce qui est dit entre les parenthèses qui suivent le mot clé catch.
- Vous pouvez ajouter un dernier bloc à votre instruction try / catch : le bloc finally. Ce bloc est facultatif, mais si vous le fournissez, il devra obligatoirement être positionné en dernier. Il permet de dire ce que vous souhaitez faire en fin d'instruction try et ce quelle que soit la manière dont il se termine (en succès ou en échec).

# JAVA: LES FICHIERS / DOSSIERS

- Pour aller loin, Java nous permet aussi de créer des fichiers ou dossiers, les déplacer ou même les supprimer en résumé faire ce qu'on veut avec nos fichiers
- Commeçons par la création de fichiers
- Le constructeur le plus simple pour un objet File prend un nom de fichier complet. En l'absence de chemin, Java utilise le répertoire courant. Par exemple :

# JAVA – LES FICHIERS

- Les tampons (buffers) sont des objets qui contiennent des données de différents types primitifs (byte, char, short, int, long, float et double).

# JAVA - TYPE GÉNÉRIQUE

- Une classe générique est une classe Java qui peut être réutilisée pour des objets de différents types. Prenons l'exemple de la classe `ArrayList` de Java 4, qui permet de stocker des objets dans une liste.
- Cette classe expose entre autres une méthode `get()`, et une méthode `add(Object)`.

# JAVA – LES TYPAGES

- Exemple

```
package type_generique;

public class TestGen<T> {

    private T money;

    public TestGen(T money) {
        super();
        this.money = money;
    }
}
```

```
public class Main {

    public static void main(String[] args) {

        TestGen euro = new TestGen<Character>('e');
        TestGen euro2 = new
        TestGen<String>("euro");

        TestGen euro3 = new TestGen<Integer>(5);
    }
}
```

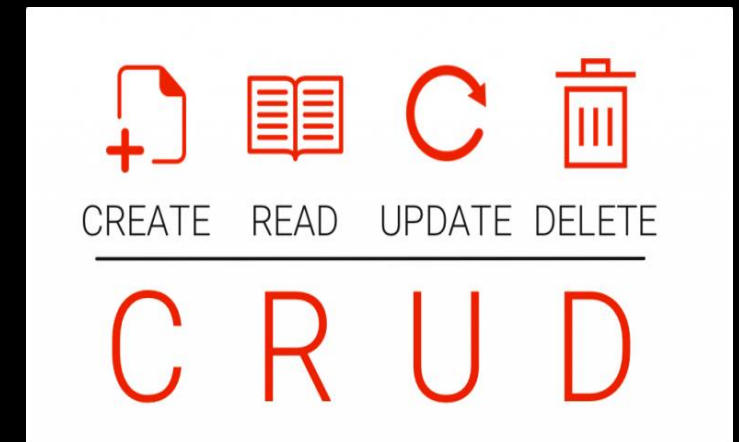


# JAVA – DAO (DATA ACCESS OBJECT)

- C'est quoi encore un truc ?
- Un objet d'accès aux données (en anglais data access object ou DAO) est un patron de conception (c'est-à-dire un modèle pour concevoir une solution) utilisé dans les architectures logicielles objet.
- Les objets en mémoire vive sont souvent liés à des données persistantes (stockées en base de données, dans des fichiers, dans des annuaires...).
- Le modèle DAO propose de regrouper les accès aux données persistantes dans des classes à part, plutôt que de les disperser.

# JAVA - DAO

- Le pattern DAO (Data Access Object) permet de faire le lien entre la couche métier et la couche persistante, ceci afin de centraliser les mécanismes de mapping entre notre système de stockage et nos objets Java. Il permet aussi de prévenir un changement éventuel de système de stockage de données (de PostgreSQL vers Oracle par exemple).
- La couche persistante correspond, en fait, à notre système de stockage et la couche métier correspond à nos objets Java, mapper sur notre base. Le pattern DAO consiste à ajouter un ensemble d'objets dont le rôle sera d'aller :
  - lire ;
  - écrire ;
  - modifier ;
  - supprimer ;



# JAVA – CRÉONS NOTRE PREMIER DAO

```
public static Connection getConnection() {
    String url = "jdbc:mysql://localhost/";
    String dbName = "ecole_afpa";
    String user="root";
    String pwd="";
    Connection connect = null;
    try {
        Class.forName("com.mysql.jdbc.Driver");

        try {
            connect = DriverManager.getConnection(url+dbName,user,pwd);
            System.out.println("OK for connect");

        } catch (SQLException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    } catch (ClassNotFoundException e) {e.printStackTrace();}
    return connect;
}
```

# JAVA – CLASS METIER

- Nous allons maintenant nous attaquer à la couche métier de notre application.
- Pour une meilleure gestion de nos entités, on doit pouvoir:
  - Ajouter , Modifier, supprimer, lister une ou toutes les informations

# DAO- CLASS METIER

```
package dao;

import java.util.List;

public interface IDAO<T> {
    public void create(T object);
    public List<Client> read();
    public void update(T object);
    public void delete(T object);
    public void findById(T object);
}
```

# JAVA - DAO

- Maintenant que nous avons notre interface (model), nous pouvons dorénavant effectuer des actions communes à toutes les entités.
- Commençons par le **C**RUD

```
@Override
public void create(Client object) {
    try {
        PreparedStatement sql =
            connect.prepareStatement("INSERT INTO
            client (nom, prenom) VALUES (?,?)");

        sql.setString(1, object.getNom());
        sql.setString(2, object.getPrenom());

        sql.executeUpdate();
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

# JAVA - CRUD

- Affichage de la liste des clients avec id et nom.

```
@Override
public List<Client> read() {
    List<Client> listCli= new ArrayList<>();
    try {
        System.out.println("OK for connect");

        PreparedStatement sql = connect.prepareStatement("select * from client");
        ResultSet rs = sql.executeQuery();

        while(rs.next()) {
            System.out.println(rs.getString("nom"));
            Client cli = new Client();
            cli.setNom(rs.getString("nom"));
            cli.setPrenom(rs.getString("prenom"));

            listCli.add(cli);
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
    return listCli;
}
```



# JAVA - CRUD

- Update client

```
@Override
public void update(String table, String nouvelle){
try {
    PreparedStatement ps = connect.prepareStatement("UPDATE "+table+ " SET nom = ? "
+ " WHERE id = ?");

    ps.setString(1, nouvelle);
    ps.setInt(2, 5);

    ps.executeUpdate();
    ps.close();

    System.out.println("updated !");

} catch (SQLException e) {
    e.printStackTrace();
}
}
```

# JAVA - CRUD

- DELETE CLIENT

```
@Override
public void delete(String table) {

    try {
        PreparedStatement ps = connect.prepareStatement("Delete FROM "+ table+
            " WHERE id=?");

        ps.setInt(1, 1);
        ps.execute();
        ps.close();
        System.out.println("Deleted !");

    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```