Ingénierie du Web Avancé

Diplôme du Cycle des Études Supérieures Spécialisées

Développement Orienté Objet & Java-JEE

A. EL HIBAOUI

Faculté des Sciences de Tétouan – Université Abdelmalek Essaâdi
Département Informatique
2021–2022

aelhibaoui@uae.ac.ma

Plan du cours

1. Programmation orientée Objet en Java
2. Bases du langage Java
3. Principes de la programmation objet et UML
4. Spring framework – Spring Web – Spring Data – Spring Security – Microservices et Spring Cloud

Structure d'un programme Java

Un programme Java est constitué d'un ou plusieurs fichiers source. Chaque fichier source peut contenir, dans l'ordre :

1. une déclaration de paquetage (optionnelle)
2. une ou plusieurs importation(s) de paquetage (optionnelles)
3. une ou plusieurs déclaration(s) de classes : au moins une obligatoire.

Pour pratiquer le langage Java, c'est-à-dire pour saisir, compiler et exécuter vos programmes, il vous faut un environnement de développement Java, comportant un compilateur, une machine Java et des bibliothèques au moins la bibliothèque standard. Vous obtenez tout cela en installant le JDK (Java Development Kit), que vous pouvez télécharger gratuitement depuis le site de Sun, à l'adresse http://java.sun.com/javase/downloads.
Quel que soit le système d'exploitation que vous utilisez, vous pouvez vous constituer un environnement de travail plus agréable en installant le JDK puis un éditeur de textes orienté Java. Vous en trouverez plusieurs sur le web, par exemple NetBeans, à l'adresse http://www.netbeans.org/.

Un exemple de programme Java

Ci-dessous, un simple programme Java.

FirstProg.java

```
// Mon premier programme Java
class FirstProg {
   public static void main (String[] args){
   System.out.println("Salam Alikom");
   }
}
```

Compilation et exécution d'un programme Java

Il est important de sauver ce fichier sous le nom de la classe *principale*, c'est-à-dire au nom FirstProg.java (un autre nom provoquera une erreur lors de la compilation). A l'aide d'un environnement de développement Java intégré JDK compilez le programme avec la commande suivante :

```
javac FirstApp.java
```

Le compilateur devrait créer un fichier nommé `FirstProg.class`. Désormais, vous pouvez exécuter votre programme grâce à la machine virtuelle Java, en tapant la commande suivante :

```
java FirstApp
```

Il ne faut pas spécifier l'extension (.class) au risque d'obtenir une erreur !
Votre programme Java, une fois compilé, peut être exécuté sur n'importe quel environnement matériel pour lequel existe une machine virtuelle. Cela vous permet de diminuer votre dépendance vis-à-vis d'un système d'exploitation ou du matériel.

Notre exemple affichera à l'écran le texte `Salam Alikom`

Les paquetages

- • **Le paquetage anonyme**
  Dans ce fichier, aucune déclaration de paquetage n'est écrite à l'entête du fichier et pourtant le programme s'est compilé et il s'est exécuté.
  Cependant la classe ainsi déclarée appartienne à un paquetage anonyme par défaut, fourni par le système.
  Toutes vos classes appartiendront à ce paquetage tant que vous ne ferez pas une déclaration spécifique de paquetages en tête du fichier.
- • **Le paquetage java.lang**
  Bien que cela n'apparaisse pas, il y a tout un ensemble classes importées dans tout code Java; il s'agit des classes présentes dans le paquetage java.lang.
  Ce paquetage comporte de nombreuses classes de base (System, Math, Thread etc ...)

Quelques paquetages

La Table 1 présente les principaux paquetages de Java 2

java.applet	Fournit les classes nécessaires à la réalisation d'Applet
java.awt	Classes nécessaires pour la réalisation d'interface graphique, pour le fenêtrage et le graphisme ainsi que les images.
java.awt.event	Gestion des événements déclenchés par des composants AWT.
java.awt.font	Gestion des polices de caractères.
java.awt.image	Gestion des images.
java.io	Gestion des flux (octets et caractères) d'entrées sorties
java.lang	Classes fondamentales du langage (System, Math, Numériques, Chaines, Thread, Objet ...)
java.net	Gestion du réseau des connexions.
java.rmi	Gestion d'invocation de méthodes sur une machine distante
java.security	Sécurité
java.sql	Gestion des échanges avec les bases de données
java.text	Gestion de la localisation, au niveau du texte, des dates etc ...
java.util	Très nombreuses classes fournissant des services de types stockage de données (pile, file, vecteur, liste, collection, table de hash, dictionnaire) mais aussi calendrier, temporisateur ...
java.util.jar	Gestion des JAR (archives java compressées)
java.util.zip	Gestion de la compression
javax.swing	Remplace AWT ; gestion de l'interface du graphisme, du fenêtrage de façon indépendante du serveur de fenêtre de l'OS hôte.
javax.swing.event	Gestion des événements générés par les composants Swing
javax.swing.table	Fournit des classes et interfaces pour faire face à java.awt.swing.JTable]].
javax.swing.text	Gestion des textes, sous classes pour interprétés directement le format rtf ou le format html
javax.swing.tree	Gestion du composant Tree, permet d'afficher graphiquement une structure d'arbre (répertoire de fichiers par exemple)

Table: Principaux paquetages de Java 2

Considérations lexicales

Un fichier source Java, en dehors des séparateurs (espace, virgule, point-virgule etc.) et des commentaires utilise pour manipuler les différents types de données les éléments suivants :

- Jeu de caractères
- commentaires
- identificateurs (noms des variables par ex.)
- littéraux (valeurs numériques par ex.)
- opérateurs (opérateurs logiques par ex.)
- mots réservés (instructions par exemple)
- déclarations de classes ou d'interface

Jeu de caractères

Java utilise le codage des caractères UTF-16, une implémentation de la norme Unicode, dans laquelle chaque caractère est représenté par un entier de 16 bits, ce qui offre $2^{16} = 65536$ possibilité au lieu des 128 (ASCII) ou 256 (Latin-1) imposés par d'autres langages de programmation. Il y a donc de la place pour la plupart des alphabets nationaux ; en particulier, tous les caractères français y sont représentés sans créer de conflit avec des caractères d'autres langues. Cela vaut pour les données manipulées par les programmes (caractères et chaînes) et aussi pour les programmes eux-mêmes. Il est donc possible en Java de donner aux variables et aux méthodes des noms accentués.

Conseil.

Il est conseillé de ne pas utiliser la possibilité de nommer les variable avec des accents car vous n'êtes jamais à l'abri de devoir un jour consulter ou modifier votre programme à l'aide d'un éditeur de textes qui ne supporte pas les accents.

Commentaires

Il y a trois sortes de commentaires. D'abord, un texte compris entre // et la fin de la ligne :

```
int nbLignes;          // nombre de variables du systeme
```

Ensuite, un texte compris entre /* et */, qui peut s'étendre sur plusieurs lignes :

```
/* Recherche de l'indice ik du ``pivot maximum''
    c.-a-d. tel que k <= ik < nl et
   a[ik][k] = max { |a[k][k]|, ... |a[nl - 1][k]| } */
```

Enfin, cas particulier du précédent, un texte compris entre /** et */ est appelé commentaire de documentation et est destiné à l'outil `javadoc` qui l'exploite pour fabriquer la documentation d'une classe à partir de son texte source. Cela ressemble à ceci (un tel commentaire est associé à l'élément, ici une méthode, qu'il procède) :

```
/**
 * Resolution d'un systeme lineaire
 *
 * @param a Matrice du systeme
 * @param x Vecteur solution du systeme
 * @return <tt>true</tt> si la matrice est inversible, <tt>false</tt> sinon.
*/
public boolean resolution(double a[][], double x[]) {
                ...
```

Les identificateurs

Les critères pour constituer des identificateurs valides sont habituels : Séquences illimitées d'alphanumériques Unicode, l'initiale devant être une lettre.
L'alphabet utilisable est très notablement étendu (par rapport uax autres langages) puisque vous pouvez utiliser tout caractère Unicode.
Cependant bien que les lettres accentuées ou accompagnées de signes diacritiques soient acceptées, leur usage n'est pas recommandé. Le particularisme d'écriture ainsi introduit est dommageable à la portabilité du code.

Note

il vaut mieux éviter le symbole "$" qui est utilisé par le compilateur, notamment dans la nomination des classes internes compilées.

Évidemment les mots réservés du langage ne font pas des identificateurs valides. De même les littéraux booléens **false**, **true** et le littéral **null** sont exclus.

Les littéraux

Un littéral est la représentation de la valeur soit d'un type primitif, soit du type chaîne (**String**), ou du type nul (qui a une seule valeur littérale null).
En dehors de **null** les littéraux peuvent appartenir à l'une des 5 catégories suivantes :

1. entiers
2. réels
3. caractères
4. chaînes de caractères
5. booléens.

Les entiers

Les entiers peuvent être écrits en base décimale, octale ou hexadécimale :

```
int a = 21 ;// decimal
int b = 025 ; // octal
int c = 0x15 ; // hexadecimal
```

Les littéraux numériques entiers sont encodés par défaut en int (32 bits) sauf si vous ajoutez un “l” ou un “L” à la fin du littéral ; dans ce cas la valeur littérale est encodée en long (64 bits)

```
short a = 123 ; // 123 est encode en int puis transtype en short
short aa = (int) 123 ; // idem
int b = 123 ;
int c = 123L ; // erreur compilation il faut declarer a comme long
```

Les réels

Les réels peuvent être écrits avec la notation décimale ou exponentielle :

```
double f = 1.23 ;
double g = 1.23E2 ;// l'ecriture 1.23 E2 provoque une erreur
```

Les littéraux numériques décimaux sont encodés par défaut en double (64 bits).

```
float h = 1.23 ; // provoque une erreur
float f = (float)1.23 ; // ecriture acceptee
```

Les caractères

Les caractères sont insérés dans le fichier source à l'aide de guillemets simples ;

```
char ch = 'A' ;
char tabul = '\t' ;
```

Certaines séquences de caractères ont les significations particulières habituelles :

\n	nouvelle ligne
\t	tabulation
\b	espace arrière
\r	retour chariot
\f	nouvelle page

Caractères Unicode: Java dans un souci de portabilité et de localisation utilise l'Unicode qui définit le codage des caractères sur 16 bits. Les caractères non imprimables peuvent être représenté aussi par la séquence \u*xxxx*, où *xxxx* sont les quatre chiffres hexadécimaux du code numérique du caractère. Par exemple, la chaîne suivante comporte les quatre caractères A, espace (code 32, ou 20 en hexadécimal), B et ñ (code 241, ou F1 en hexadécimal) :

```
"A\u0020B\u00F1" // la chaine de quatre caracteres "A B~n"
```

es chaînes de caractères

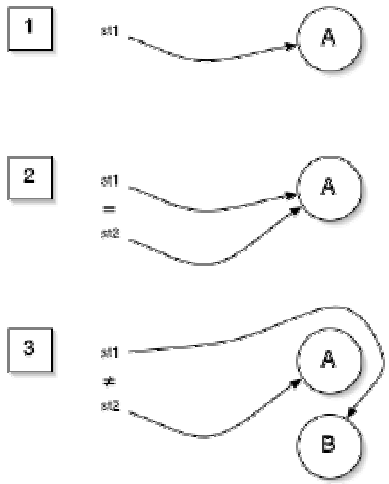
Les chaînes de caractères littérales seront insérées à l'aide de doubles guillemets.

```
String s = "Fichier introuvable" ;
```

Attention les chaînes de caractères de type String sont immuables :

```
String st1 = ''A'' ;     // st1 reference un objet de type String dont la ''valeur'' est A
String st2 = st1 ;  // st2 reference le meme objet que st1.
st1 = ''B'';     //La ''valeur'' de l'objet reference par st1 est maintenant B.
```

En fait un nouvel objet de type String a été créé avec comme "valeur" B et la référence à ce nouvel objet String a été affecté à la variable st1 ; maintenant st1 et st2 ne référence plus le même objet. C'est ce qui est représenté dans la série des 3 schémas de la figure 1 :



StringBuffer

Il existe une autre entité, en Java, capable de gérer des chaînes de caractères. Il s'agit de la classe **StringBuffer**. Les objets de cette classe ne sont pas immuables, comme le montre l'exemple ci-dessous :

```
StringBuffer sb1 ;
sb1 = new StringBuffer("X");     //creation intialisation de l'objet sb1 de type StringBuffer
StringBuffer sb2 = sb1;      /*sb2 est cree et recoit la reference contenue dans sb1, sb1 et sb2 referenc
sb2.append("Y");     /*la methode append() est appliquee a l'objet sb2, elle a pour effet de concatener l
```

Les booléens

Booléens

Les booléens peuvent prendre les valeurs littérales **true** ou **false**.

```
bool a=false;
...
if (a){
...
}else {
...
}
```

Les différents types de données

Les types des données manipulées par les programmes Java se répartissent en deux catégories :

- les types primitifs,
- les objets, c'est-à-dire les tableaux et les classes.

Types primitifs

La table 2 récapitule ce qu'il faut savoir sur les huit types primitifs.

Type	description	taille	ensemble de valeurs	valeur initiale
Nombres entiers				
byte	Octet	8 bits	de -128 à 127	0
short	Entier court	16 bits	de -32768 à 32767	0
int	Entier	32 bits	de -2147483648 à 2147483647	0
long	Entier long	64 bits	de -2^{63} à $2^{63}-1$	0
Nombres flottants				
float	Flottant simple précision	32 bits	de $-3,4028235 \times 10^{+38}$ à $-1,4 \times 10^{-45}$, 0 et de $1,4 \times 10-45$ à $3,4028235 \times 10^{+38}$	0.0
double	Flottant double précision	64 bits	de $-1,7976931348623157 \times 10+308$ à $-4,9 \times 10^{-324}$, 0 et $4,9 \times 10^{-324}$ à $1,7976931348623157 \times 10+308$	0.0
Autres types				
char	Caractère	16 bits	Tous les caractères Unicode	'\u0000'
boolean	Valeur booléenne	1 bit	false, true	false

Table: Types primitifs

Remarque

Classes correspondante : `Float, Double.`

- Le type décimal donne lieu à deux types primitifs : float (32 bits) et le double (64 bits).
- Le dépassement de capacité inférieure retourne 0 (il y a des 0^+ et des 0^-)
- La division par 0 retourne $\pm$ **Infinity** suivant le signe du numérateur.
- Le quotient $0/0$ retourne **NaN** (Not a Number).

Conversions entre types primitifs

Le transtypage entre entier et flottant est autorisé.

La conversion élargisssante (i.e. vers un type possédant une capacité supérieure) est automatique.

Tandis que la conversion restrictive nécessite l'usage de l'opérateur de transtypage **()**.

De plus en cas de transtypage restrictif entre valeurs numériques il faut prendre garde au dépassement de capacité qui peut altérer totalement le résultat.

Les types composites ou références

Java selon Sun

Java selon Sun[a]

[a] http://java.sun.com/docs/overviews/java/java-overview-1.html

Java: A simple, object-oriented, network-savvy, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded, dynamic language.

C'est-à-dire, Sun définit Java comme :

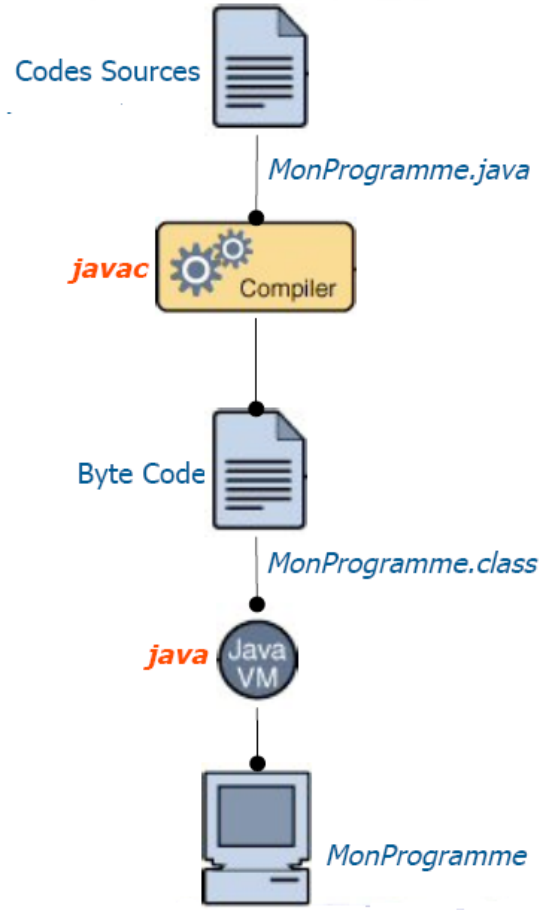
- Simple
- Orienté objet
- Réparti
- Interprété
- Sûr
- Portable
- Performant
- Multitâches
- Dynamique ...

Structure des fichiers sources

- Un programme java = une collection de définitions de classes.
- Un fichier source (`fichier.java`) peut contenir plusieurs définitions de classes et les classes d'une même application peuvent être dans des fichiers sources différents.

▷ **Principe de fonctionnement**

- **Code source :** Fichiers utilisés lors de la phase de programmation
- **Byte-code :** Fichiers obtenus par compilation et destinés à être exécutés sur la machine virtuelle
- **Machine virtuelle :** Programme interprétant le byte-code. La machine virtuelle dépend du système d'exploitation, mais elle est capable d'exécuter tout programme Java même s'il a été compilé avec un autre système d'exploitation



Compilateur et machine virtuelle

- **Versions de la machine virtuelle :**
  - Java 2 Micro Edition (Java ME) : les terminaux portables
  - Java 2 Standard Edition (Java SE) : les postes clients
  - Java 2 Enterprise Edition (Java EE) : un serveur d'application
- **Environnements :**
  - SDK/JDK (Software/Java Development Kit) fournit un compilateur et une machine virtuelle
  - JRE (Java Runtime Environment) fournit uniquement une machine virtuelle.
- **Version actuelle de Java (à voir) :**
  - Actuellement "Java SE 6.0"
  - Bientôt Java SE 7.0

Environnement de développement (IDE)

Statistiques d'après www.developpez.net

Affichage des résultats du sondage: Quel EDI Java utilisez vous en 2009 ?

Eclipse	188	49,47%
NetBeans	160	42,11%
IntelliJ	10	2,63%
RAD / WSAD	1	0,26%
JDeveloper	2	0,53%
JCreator	3	0,79%
MyEclipse	4	1,05%
JBuilder (>= 2007)	2	0,53%
JBuilder (<= 2006)	1	0,26%
BEA Workshop Studio	1	0,26%
Editeurs de texte avancés (Emacs, VI, JEdit, UltraEdit, ...)	6	1,58%
Autre (précisez)	2	0,53%

Votants: **380**. Vous ne pouvez pas participer à ce sondage.

Figure: Eclipse (plus complet) – NetBeans

Structures de contrôle

Actions séléctives : si ... sinon

Algorithmique

```
Si condition alors
    action 1;
Sinon
    Action 2
Fin si
```

Java

```
if(condition){
    action 1;
    }
else {
    action 2;
    }
```

Structures de contrôle

Actions séléctives : si ... sinon

Exemple de SI en Java

```
int plusGrand(int a,int b){
     if (a>b) return a;
     else return b;
}
```

Structures de contrôle

Actions séléctives : selon

Algorithmique

```
selon variable
 debut
  val 1 : action 1;
  val 1 : action 1;
  ...
  val p : action p;
  sinon : action;
 fin;
```

Java

```
switch(variable){

  case val_1 : action_1;
  case val_2 : action_2;
    ...
  case val_p : action_p;
  default  : action;
}
```

Structures de contrôle

Actions séléctives : selon

Exemple de Selon en Java

```
switch (n){
   case 1 : n++;
   case 2 : n--;
   case 3 : ++n;
   default : n=0;
}
```

Un mot sur `break` et `continue`

Les instructions `break` et `continue` facilitent respectivement la sortie complète et l'arrêt pour l'itération en cours d'une boucle suite à une condition particulière nécessitant une action précise.

Structures de contrôle

Boucles itératives : boucle pour

Algorithmique

```
Pour i <- valInit jusqu a n pas de k faire
 debut
  Action 1;
  ...
  Action p;
FinPour;
```

Java

```
for(i=valInit; i<=n ; i++){
   action 1;
   ...
   action p;
}
```

Structures de contrôle

Boucles itératives : boucle pour

Exemple de la boucle Pour en Java

```
for (i=1;i<=n;i++){
  if(i%2==0)
     System.out.println("nombre pair");
  else
     System.out.println("nombre impair");
}
```

Structures de contrôle

Boucles itératives : boucle Tant que

Algorithmique

```
tant que (condition) faire
    action;
fin tant que;
```

Java

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

Structures de contrôle

Boucles itératives : boucle tant que

Exemple de la boucle Tant que en Java

```
i=1;
while (i<=n){
  if(i%2==0)
     System.out.println("nombre pair");
  else
     System.out.println("nombre impair");
  i++;
}
```

Structures de contrôle

Boucles itératives : boucle répéter

Algorithmique

```
repeter
    action;
jusqu a (condition);
```

Java

```
do{
    action;
  }
while(!condition);
```

Structures de contrôle

Boucles itératives : boucle répéter jusqu'à

Exemple de la boucle répéter jusqu'à en Java

```
i=1;
do{
   if(i%2==0)
      System.out.println("nombre pair");
   else
      System.out.println("nombre impair");
   i++;
}
while(i!=n);
```

Tableaux en Java

```
//Création d'un tableau d'entiers de taille 10
int [] tabl1 = new int[10];

//Création d'un tableau d'entier de taille 10 en deux fois
int[] tabl2;
tabl2 = new int[10];

// Création d'un tableau d'entiers de taille 10 et remplissage initial
 int tabl3 = new int[] {1,2,3,4,5,6,7,8,9,10};

//Création d'un tableau d'objets de taille 10
String[] tabS1 = new String [10];

// Création d'un tableau d'objets de taille 10 en deux fois
String[] tabS2;
tabS2=new String [10];

// Création d'un tableau d'objets de taille 4 et remplissage initial

String[] tabS3 = new String [5]={"Mohamed"; "Abou Baker"; "Ali"; "Omar"; "Othman"};
```

Les tableaux ont une taille fixe.
Ils ne sont pas redimmensionnables.

Tableaux en Java

En Java, les tableaux sont indexés à partir du 0.

Taille d'un tableau

```
int[] tabl=new int []{13,24,35,45,56,67,78,89,90,12};
System.out.println("La taille du tableau est "+tabl.length);
```

```
La taille du tableau est 10
```

Accès aux valeur d'un tableau

```
System.out.println("Accès aux valeurs indicées de 0 à 9 ");
for(int i=0; i<10; i++)
    System.out.print("\t"+tabl[i]);
```

```
Accès aux valeurs indicées de 0 à 9
13   24   35   45   56   67   78   89   90   12
```

Dépassement de capacité

Accès à la valeur indicée 10

```
System.out.println("Accès à la valeur indicée 10 ");
System.out.println(tabl[10]);
```

```
Accès à la valeur indicée 10
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 10
    at Tableau.main(Tableau.java:15)
```

- En java, les tableaux sont considérées comme des classes sans méthode.
- La variable membre `length` donne la longueur du tableau.
- Le dernier élément d'un tableau de taille n se trouve à l'index n-1.

Valeurs par défaut d'un tableau

Type primitifs

```
int[] tabI=new int [10];
System.out.println("Accès aux valeurs indicées de 0 à 9 ");
for(int i=0; i<10; i++)
    System.out.print("\t"+tabI[i]);
```

```
Accès aux valeurs indicées de 0 à 9
0   0   0   0   0   0   0   0   0   0
```

Valeur par défaut pour des objets

```
Personne[] tabP=new Personne [10];
System.out.println("Accès aux valeurs indicées de 0 à 9 ");
for(int i=0; i<10; i++)
    System.out.print("\t"+tabP[i]);
```

```
Accès aux valeurs indicées de 0 à 9
null    null    null    null    null    null    null    null    null    null
```

Valeurs par défaut d'un tableau

- Les variables de types primitifs sont initialisées à 0 pour les types numériques et à false pour les booléens.
- Les références sur les objets sont initialisées à null.

Initialisation

Initialisation des valeurs du tableau

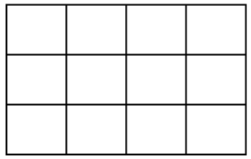
```
System.out.println("Initialisation des valeurs du tableau");
for(int i=0; i<10; i++)
     tabP[i]=new Personne();

System.out.println("Accès aux valeurs indicées de 0 à 9 ");
for(int i=0; i<10; i++)
    System.out.print("\t"+tabP[i]);
```

```
Initialisation des valeurs du tableau
Accès aux valeurs indicées de 0 à 9
Personne@e53108     Personne@f62373     Personne@19189e1    Personne@1f33675
Personne@7c6768     Personne@1690726     Personne@5483cd        Personne@9931f5
Personne@19ee1ac    Personne@1f1fba0
```

- Une référence indique un emplacement mémoire.
- Cet emplacement ne représente rien en dehors de la JVM.

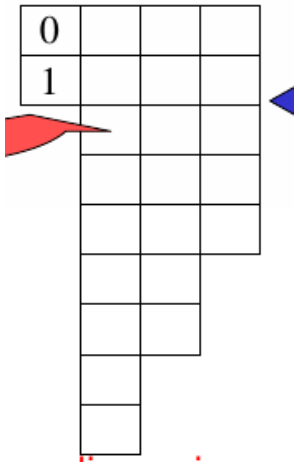
Tableaux à plusieurs dimensions

```
int[][] matI = new int [4][3];
```



```
int[][] mat2 = new int [][]{new int [2], new int [9], new int [7],new int [5] };
```

```
//remplissage du 1er tableau
for(int i=0; i<mat2[0].length; i++){
   mat2[0][i]=i;
} // end of for()
```



En java, les tableaux à plusieurs dimensions sont vus comme des tableaux de tableaux (de tableaux, de tableaux, etc)

Passage de tableaux en paramètres

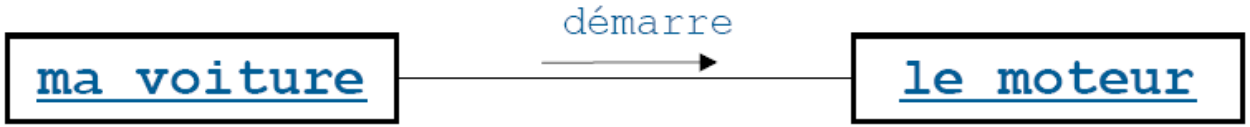
```
public class Tableau {
    public static void modifieTableau( int [] tab){
        for(int i =0; i< tab.length; i++)
            tab[i] *=2;
        }
    public static void modifieValeur(int i){
        System.out.println("La valeur de i au début de modifieValeur est "+i);
        i*=2;
        System.out.println("La valeur de i a la fin de modifieValeur est "+i);
    }
    public static void main(String [] s){
        int [] tabI= new int[]{1,2,3,4,5};
        System.out.println();
        System.out.println("\n Acces aux valeurs indexees de 0 a 4 avant modifieTableu");
        for (int i=0; i < tabI.length; i++)
            System.out.println(tabI[i]);
        modifieTableau(tabI);
        System.out.println("\n Acces aux valeurs indexees de 0 a 4 apres modifieTableu");
         for (int i=0; i < tabI.length; i++)
            System.out.println(tabI[i]);
        System.out.println("\n La valeu de tabl[2] avant modifieValeur est "+tabI[2]);
        modifieValeur(tabI[2]);
        System.out.println("\n La valeu de tabl[2] après modifieValeur est "+tabI[2]);
    }
}
```

Passage de tableaux en paramètres

```
 Acces aux valeurs indexees de 0 a 4 avant modifieTableu
1
2
3
4
5

 Acces aux valeurs indexees de 0 a 4 apres modifieTableu
2
4
6
8
10

 La valeu de tabI[2] avant modifieValeur est 6
La valeur de i au début de modifieValeur est 6
La valeur de i a la fin de modifieValeur est 12

 La valeu de tabI[2] après modifieValeur est 6
```

Java et les objets

- Tout est objet (ou presque). Un objet est défini par :
  - Une identité : permet de distinguer un objet d'un autre objet (type)
  - Un état : représenté par des attributs (variables) qui stockent des valeurs
  - Un comportement : défini par des méthodes (procédures ou fonctions) qui modifient des états
- Les objets sont décrits grâce à leur classe : chaque objet est une instance d'une classe.
  - La classe décrit les attributs de ses instances. On dit aussi les variables d'instance ou même les données membres.
  - La classe décrit le comportement de ses instances sous la forme de méthodes utilisées par les instances pour « réagir à des messages » qui leurs sont envoyés.
- Les objets communiquent entre eux par des messages.
  - Un objet peut recevoir un message qui déclenche une méthode qui modifie son état et / ou une méthode qui envoie un message à un autre objet



Programmation objet : classe java

```java
package voiture;

public class Voiture{
    // variables d'instances
    private int puissance ;
    private boolean estDemarree ;
    private float vitesse ;

    // Constructeur
    public  Voiture(){...}

    //trois méthodes
    public void demarre(){...}
    public void accelere(float v) {...}
    public int getPuissance() {...}
```

← Nom du package
← Nom de la classe

/!\ Cette classe doit être enregistrée dans un fichier qui porte le même nom que la classe Voiture.java

La compilation se fait à l'aide de la commande :

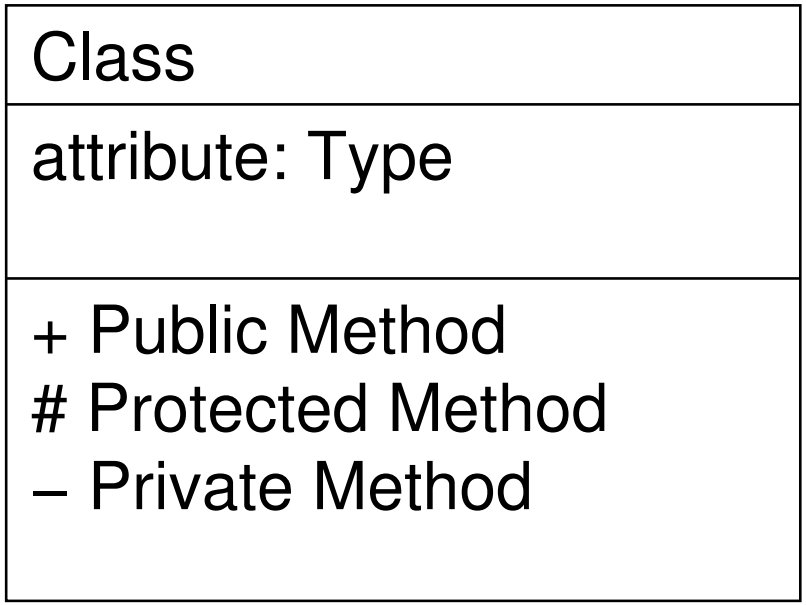
```
javac Voiture.java
```

Pour compiler, il est nécessaire d'avoir une méthode

```java
    public static void main(String agrv[])
```

qui contient les instructions à exécuter

Programmation objet et UML



Descente aux enfers : écrire du Java

En Java, pas de code exécutable (instructions) hors des méthodes

Tout programme comprend au moins une classe

Package

- **Un package** = regroupement de classes et d'interfaces (voir plus loin) associées à une fonctionnalité
  - **But** : regrouper les classes afin d'organiser des libraires de classes Java
  - **Avantage** : réduit le risque de collision de noms
- Toute classe Java est un élément d'un package
- La première chose à écrire dans le fichier est le nom du package auquel la classe appartient
- Exemples de packages
  - `java.lang` : rassemble les classes de base Java (Object, String, ...)
  - `java.util` : rassemble les classes utilitaires (Collections, Vector, ...)
  - `java.io` : lecture et écriture

Elaboration d'un package

On définit un package par le mot réservé **package**, suivi du nom du package et d'un point virgule

```
package monpackage;
public class A{ ... }
public class B{ ... }
public class C{ ... }
```

- La définition du package doit être la première instruction du fichier.
- Si la définition de package est omise, un package par défaut sera défini par le compilateur nommé **unamed**

Utilisation d'un package

Il suffit de préfixer le nom de la classe par le nom du package.

```
package monpackage;

public class A{ ...}
public class B{ ...}
public class C{ ... }
```

```
package test;

public class T{
    monpackage.A unA = new monpackage.A();
    ...}
```

C'est très lourd, utilisé import

Le mot réservé import permet d'ouvrir le contexte d'un package. Attention au collision des noms et il n'est pas nécessaire d'importer le package dont fait partie la classe

```
package test;
import monpackage.A;
public class T{
    A unA = new A();
    ... }
```

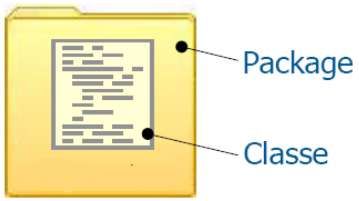
Utilisation d'un package : comportement du compilateur

Lorsqu'il y a une référence à une classe, le compilateur la recherche dans le package par défaut (java.lang) et dans tous les packages importés. S'il ne trouve pas la classe en question, il lève une erreur de résolution de symbole. Il est nécessaire de fournir au compilateur explicitement l'information pour savoir où se trouve la classe :

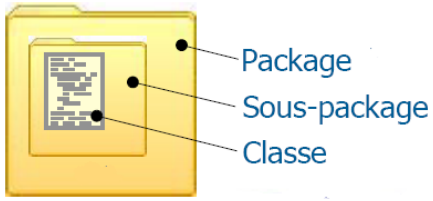
- Utilisation d'import (classe ou paquetage) et/ou
- Nom du paquetage avec le nom de la classe
  - import java.lang.String; // Ne sert à rien puisque par défaut
  - import java.io.ObjectOutput;
  - import monpackage.*;
  - import java.lang.*; // Ne sert à rien puisque par défaut

Packages et répertoires

- Chaque classe correspond à un fichier
- Chaque package (ou sous-package) correspond à un répertoire : le package monpackage correspond au répertoire monpackage.



- Un package peut contenir :
  - Des classes ou des interfaces
  - Des sous-packages



- La hiérarchie des répertoires donne des noms composés pour les packages : monpackage.xyzzy correspondra au répertoire monpackage/xyzzy

Le nom des packages est toujours écrit en minuscules

A propos du nom du package

Chaque logiciel code les caractères comme il l'entend. Ils ne sont d'accord que sur les chiffres et les lettres sans accents.
Or, les règles ci-dessus introduisent une correspondance entre des noms internes à Java et des noms de fichiers gérés par l'OS.
Il y a de grandes chances que si vos noms de packages ou de classes possèdent des lettres accentuées, la correspondance avec les fichiers ne se fasse pas correctement.

Pas d'accent dans les noms de packages ni dans les noms de classes et pas d'espace d'ailleurs, s'il s'agit d'un nom composé.

Visibilité

L'instruction import nomPackage.* ne concerne que les classes du package indiqué. Elle ne s'applique pas aux classes des sous-packages.

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

public class Essai {
    ...
    public Essai() {
        Date myDate = new Date(...);
        ZipFile myZip = new ZipFile(...);
        ...
    }
    ...
}
```

`Essai` utilise les classes `Date` du package `java.util`
et `ZipFile` du package `java.util.zip`

Classes

Les classes

Introduction

- Les objets sont les acteurs principaux d'un programme java. Leur rôle principal est d'emmagasiner des informations et à mettre à disposition une collection d'opérations qui permettent d'accéder à ces informations et/ou de les modifier
- Les objets ne sont pas des structures mais des acteurs réactifs : réagissent aux messages
- Une classe est le support de l 'encapsulation : c'est un ensemble de données (attributs) et de fonctions regroupées dans une même entité. Une classe est une description abstraite d'un objet. Les fonctions qui opèrent sur les données sont appelées des méthodes. Instancier une classe consiste à créer un objet sur son modèle. Entre classe et objet il y a, en quelque sorte, le même rapport qu'entre type et variable.

Déclaration d'une classe

Une classe comporte sa déclaration, des variables et la définition de ses méthodes. Elle se compose en deux parties : un en-tête et un corps. Le corps peut être divisé en 2 sections : la déclaration des variables et la définition des méthodes.

Syntaxe

```
[<modificateur_de_classe>] class <nom_de_classe> [extends <nom_de_Superclasse>]
[implemenents <Interface_1> <Interface_2> ...<Interface_n>] {
        // les variables
        ...
        // les méthodes
        ...
}
```

Modificateurs de classe

Les modificateurs de classe sont des mots réservés optionnels qui spécifient la portée, ainsi que le statut de la classe au sein du programme :

Modificateur	**Rôle**
public	La classe est accessible partout (dans le package et les classes qui importe le package). *Toute classe publique doit être déclarée dans un fichier qui porte le même nom .java* *Un fichier ne peut contenir qu'une classe publique*
(package)	Si aucun modificateur n'est spécifié, la classe peut être utilisée seulement à l'intérieur du package, même si on importe le package
abstract	La classe contient des méthodes abstraites, qui n'ont pas de définition explicite. Elle ne peut pas être instanciée : il faut définir une classe qui hérite de cette classe et qui implémente les méthodes abstraites
final	La classe ne peut pas être modifiée, sa redéfinition grâce à l'héritage est interdite. Les classes déclarées final ne peuvent donc pas avoir de sous-classes filles.

Héritage

- Le mot extends est optionnel. Si une classe Y hérite de la classe X :
  1. Toutes les opérations applicables aux instances de X sont applicables aux instances de Y
  2. Toutes les variables de la classe X qui ne sont pas déclarées privées (private) peuvent être manipulées directement par les méthodes de la classe Y
- X est dite superclasse de Y et Y est dite une sousclasse de X
- Par héritage successif, il est possible de définir une hiérarchie d'héritage. Si on omet le mot extends la classe hérite de la classe Objet. En haut de la hiérarchie, on trouve la classe **Objet** dont hérite toutes les classes Java

Implémentation

- Le mot implements permet de spécifier une ou des interfaces que la classe implémente. Ce mot est optionnel.
- Une interface est une spécification qui permet de définir un nouveau type de données, définit uniquement par la liste des opérations qui pourront lui être appliquées (sans implémentation de ces opérations)
- Une classe qui implémente une interface doit donner l'implémentation de TOUTES les méthodes de l'interface.
  Cela permet de récupérer quelques avantages de l'héritage multiple :
  - Typage multiple : supposons que X implements Y, Z
  - Une instance de X est à trois types : le type X, Y et Z

Les variables

Une classe spécifie le type des informations qui seront emmagasinées par la classe ou par les instances de la classe. Ces informations sont définies par :

1. Les variables d'instances (ou attributs d'instances)
2. Les variables de la classe

Les variables d'instances

Les variables d'instances spécifient le type des informations qui seront emmagasinées par les instances (objets) de la classe.

Syntaxe

[<modificateur_de_variable>] <type> <nom> [= <valeur_initiale>] ;

Modificateurs des variables d'instances :

Ils spécifient la porté de la variable. La portée d'une variable s'étend à toute la classe, dans laquelle elle est déclarée.

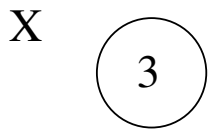
Modificateur	**Rôle**
public	La variable est accessible en lecture/écriture par les méthodes de toutes les classes. *Ce n'est pas très propre (contraire au principe d'encapsulation)*
protected	L'accès est autorisé aux sous-classes et les classes du package
private	Juste pour la classe (inaccessible à l'extérieur et aux sousclasses)
(package)	Si aucun modificateur n'est spécifié, l'accès est autorisé aux classes du package courant
final	Une variable dont la valeur ne changera pas (ou presque). Si le type de la variable est un type primitif, alors c'est une constante. Si le type est objet, la référence à l'objet ne doit pas changer (mais la valeur de l'objet le peut) . *Dans tous les cas, il faut initialiser la variable, soit directement soit dans le constructeur (sinon erreur).*

Types des variables d'instances :

En java les variables sont des références (adresses mémoires) sauf les types primitifs

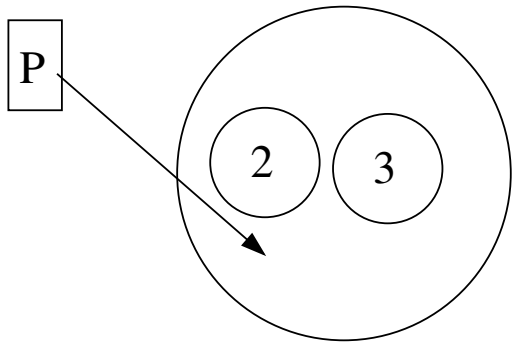
- **Types primitifs** : En java, il y a 8 types de primitifs (entier, entier long, ...). Dans ce cas, la variable contient directement la valeur du type de base.
  L'autoboxing introduit depuis la version 5.0 convertit de manière transparente les types primitifs en références

```
int x =3;
```



- **Types objets** : dont le nom désigne celui d'une instance de classe ou d'une interface. C'est des références (pointeurs selon C)

```
Point p = new Point(2,3);
```



Les variables d'instances : initialisation

Une variable peut recevoir une valeur au moment de sa déclaration

```
Class Test {

 private   int a=5; // variable privée
 public    int b=6; // variable publique
 protected int c=7; // variable protégée
           int d=8; // variable du paquet
 final     int e=9; // une constante

}
```

Les variables de classe

Les variables de classe sont utilisées pour stocker les informations globales relatives à la classe. Une variable de classe est introduit par le mot réservé static

```
public class A{
    public static int a =1;
    public static final Color b;
    ...
}
```

Si la variable de classe n'est pas initialisée par le programmeur, le compilateur l'initialise lui-même : pour les objet c'est null, les entiers 0, ...
On peut accéder aux variables de classe qui sont publiques sans passer par un objet **A.b**

Constructeur des variables de classe :

L'équivalent du constructeur existent pour les variables de classes. Il est invoqué au moment d'utilisation de la classe. A l'instar des méthodes de classe, ce constructeur ne peut accéder qu'aux variables de classes

Syntaxe

```
static{
    a=1;
    ...
}
```

Les méthodes

Toutes les méthodes sont déclarées à l'intérieur d'une classe. La définition d'une méthode comporte deux parties :

1. **Entête :** qui définit le nom de la méthode, sa portée, la liste de ses paramètres (la signature de la méthode) et éventuellement le type de retour
2. **Corps :** qui contient les instructions de la méthode

Syntaxe :

```
[<modificateur_de_méthode>] <type_de_retour> <nom_de_méthode> ( [<liste_des_paramètres >])
[<exception>]{
        <corps_de_méthode>
}
```

Modificateurs des méthodes :

Ils spécifient la porté de la méthode. La portée d'une méthode s'étend à toute la classe, dans laquelle elle est déclarée.

Modificateur	**Rôle**
public	La méthode peut être invoquée par toutes les classes.
protected	L'invocation est autorisé aux sous-classes et aux classes du package courant
private	Une méthode interne, inaccessible à l'extérieur et aux sous-classes
(package)	Si aucun modificateur n'est spécifié, l'invocation est limitée aux classes du package courant
abstract	Une méthode abstraite qui sans instruction. La classe par conséquence devient abstraite (non instanciable)
static	La méthode est dite dans ce cas de classe (par opposition à une méthode d'instance qui ne comporte pas le mot static). *Une méthode de classe ne peut accéder que aux variables de classes, alors que les méthodes d'instance peuvent accéder à toutes les variables.* Utile pour des calculs intermédiaires internes à une classe ou pour retourner la valeur d'une variable privée.

Modificateurs des méthodes :

Pour « **demander** » à un objet d'effectuer un traitement il faut lui envoyer un message :

- Pour les méthodes d'instance : le message est envoyé à l'objet

<référence_objet>.<nom_méthode>([<liste_des_paramètres>])

- Pour les méthodes de classe

<nom_classe >.<nom_méthode>([<liste_des_paramètres>])

Ne pas oublier les parenthèses pour les appels aux méthodes.

Type de retour des méthodes :

On distingue deux catégories de méthodes :

1. **Les procédures :** ne retourne aucune valeur.
   Dans ce cas, on utilise le mot réservé void comme type de retour.

2. **Les fonctions :** retourne toujours une valeur.
   La valeur retournée par l'instruction return doit être du même type que le type de retour de la fonction.

Paramètres des méthodes :

Un paramètre d'une méthode peut être :

- Une variable de type simple
- Une référence d'un objet typée par n'importe quelle classe

Les paramètres d'une méthode sont définit dans une liste où chaque déclaration d'un paramètre est séparée par une virgule.

La déclaration d'un paramètre est composée du type de paramètre suivie par le nom du paramètre.

```
(type1 var1, type2 var2, type3 var3, ..., typen varn)
```

Paramètres des méthodes :

Varargs est une nouveauté Java 5 permettant de passer en paramètre un nombre indéfini de valeurs de même type. Avant la version Java 5, il fallait passer en paramètre un tableau d'un type donné pour réaliser la même chose.

Syntaxe

La syntaxe de varargs est la suivante : utilisation de « ... » en dernier

```
public ajouterClient(String... tab)
```

Du côté de la méthode où le varargs est défini, les données sont manipulées comme un tableau

```
public ajouterClient(String... tab) {
    for (String current : tab) {
          System.out.println(current)
    }
}
```

Du côté client qui fait un appel à la méthode, les données peuvent être envoyées comme un ensemble de paramètres

```
MyClass.ajouterClient("Ahmed", "Mohammed", "Hamid");
MyClass.ajouterClient();
```

Passage des paramètres

- En Java tout est passé par valeur
  - Les paramètres effectifs d'une méthode
  - La valeur de retour d'une méthode (si différente de void)
- **Les types simples**
  - Leur valeur est recopiée
  - Leur modification dans la méthode n'entraîne pas celle de l'originale
- **Les objets**
  - Leur modification dans la méthode entraîne celle de l'originale!!!
  - Leur référence est recopiée et non pas les attributs

Les méthodes : constructeur

- **Un constructeur** est une méthode spéciale utilisée pour initialiser les objets lors de leur création.
- Un constructeur est automatiquement invoqué lors de la création d'un objet avec new
- Mise à part l'absence du type de retour, la syntaxe de déclaration d'un constructeur est essentiellement identique à celle d'une méthode
- Notons toutefois les points suivantes :
  - Le nom du constructeur est le même que le nom de la classe
  - Un constructeur ne retourne rien
  - Seuls les modificateurs public, private et protected son possibles

Les méthodes : rôle constructeur

- Le rôle du constructeur est une méthode qui sert principalement à initialiser les variables d'instance.

- Rappelons que la création d'un objet comprend deux étapes :
  1. Allocation d'une zone mémoire pour contenir les variables d'instances de l'objet. Ces dernières sont initialisées avec leurs valeurs par défaut (pour l'un objet c'est null) ou par les valeurs initiales spécifiés par le programmeur

  2. Invocation du constructeur : de nouvelles valeurs peuvent être affectés aux variables d'instances et aux variables des classes.

- Toute classe Java possède au moins un constructeur.
  Si une classe ne définit pas explicitement de constructeur, un **constructeur par défaut** sans arguments et qui n'effectue aucune initialisation particulière est invoquée

Les méthodes : constructeur par défaut

Si (et seulement si) le programmeur a écrit une classe sans spécifié de constructeur, le compilateur génère un constructeur par défaut, sans paramètres qui ne fait rien.
Ce constructeur a l'allure de

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

Cela suppose que la superclasse dispose d'un constructeur par défaut (la classe objet a au moins un). Sinon le programmeur est invité par le compilateur à corriger son erreur.

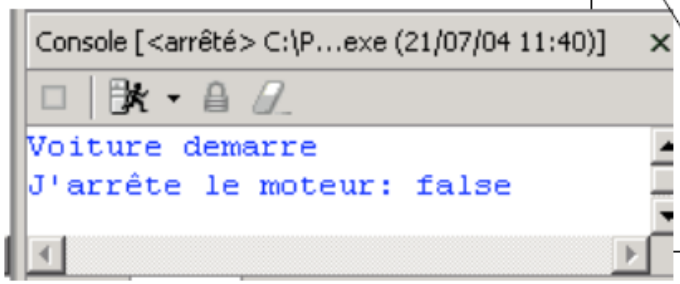
Les méthodes : destructeur

```
public class Voiture {

    private boolean estDemarree;
    ...

    protected void finalize() {
        estDemarree = false;
        System.out.println("Moteur arrêté");
    }
    ...
}
```

Pour être sûr que finalize s'exécute il faut absolument appeler explicitement *System.gc()*

```
public class TestMaVoiture {

    public static void main (String[] argv) {
        // Déclaration puis création de maVoiture
        Voiture maVoiture = new Voiture();
        maVoiture.demarre();
        // maVoiture ne sert plus à rien
        maVoiture = null;

        // Appel explicite du garbage collector
        System.gc();

        // Fin du programme
        System.exit(0);
        System.out.println("Message non visible");
    }
```

Force le programme à se terminer



Surcharge des méthodes

- On peut donner le même nom à deux ou plusieurs méthodes qui accomplissent en principe la même fonction logique = surcharge de méthodes.

- **Où surcharger une méthode?**
  On peut surcharger une méthode soit dans la même classe, soit dans une sousclasse qui surcharge les méthode de la superclasse.

- **Condition à respecter :** signatures différentes.
  La surcharge est effective dès lors qu'il y a une modification des arguments de la méthode.
  Si c'est la même signature, il s'agit d'une **redéfinition** (donc dans une sousclasse)

Surcharge des méthodes

```
public class Voiture {
    private double vitesse;
    ...

    public void accelere(double vitesse) {
        if (estDemarree) {
            this.vitesse = this.vitesse + vitesse;
        }
    }
    public void accelere(int vitesse) {
        if (estDemaree) {
            this.vitesse = this.vitesse +
                (double)vitesse;
        }
    }
...}
```

```
public class TestMaVoiture {

    public static void main (String[] argv) {
        // Déclaration puis création de maVoiture
        Voiture maVoiture = new Voiture();

        // Accélération 1 avec un double
        maVoiture.accelere(123.5);
        // Accélération 2 avec un entier
        maVoiture.accelere(124);
}}
```

Convention en Java

- Conventions des noms
  - unpackage // (tout est en miniscule)
  - CeciEstUneClasse // (le 1er caractère est en majuscule)
  - celaEstUneMethode(...)
  - getUnAttribut()
  - setUnAttribut (...)
  - jeSuisUneVariable
  - JE_SUIS_UNE_CONSTANTE
- Une classe par fichier et un fichier par classe
  - Exceptionnellement, les classes internes (Inner Class)
- Documentation du code source

  // Je suis un commentaire

  /* Encore un commentarire */

  Utilisation de l'outil Javadoc (voir plus loin)
- Mise en forme
  - Facilité de lecture
  - Crédibilité assurée
  - Indentation à chaque niveau

```
if (b == 3) {
    if (cv == 5) {
        if (q) {
            ...
        } else {
            ...
        }
        ...
    }
    ...
}
```

Préférer

```
if (b == 3) {
if (cv == 5) {
if (q) {
...
} else {...}
...
}
...
}
```

Éviter

Structure d'un objet

Un objet est constitué d'une partie « statique » et d'une partie « dynamique »

- **Partie statique :**
  - Ne varie pas d'une instance de classe à une autre
  - Permet d'activer l'objet
  - Constituée des méthodes de la classe
- **Partie dynamique :**
  - Varie d'une instance de classe à une autre
  - Varie durant la vie d'un objet
  - Constituée d'un exemplaire de chaque attribut de la classe

Cycle de vie d'un objet

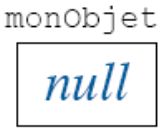
1. **Création**
   - Usage d'un Constructeur
   - L'objet est créé en mémoire et les variables de l'objet sont initialisés
2. **Utilisation**
   - Usage des Méthodes et des variables d'instance (non recommandé)
   - Les variables de l'objet peuvent être modifiées ou consultées

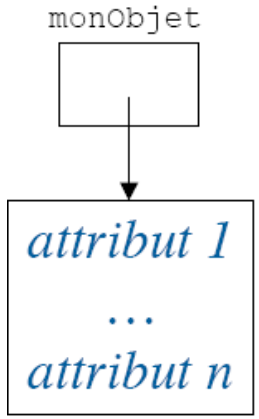
L'utilisation d'un objet non construit provoque une exception de type `NullPointerException`

3. **Destruction et libération de la mémoire**
   - Usage (éventuel) d'un Pseudo-Destructeur
   - L'objet n'est plus référencé, la place mémoire occupée est récupérée

Cycle de vie d'un objet

- La création d'un objet à partir d'une classe est appelée **une instanciation**
  - L'objet créé est une instance de la classe
- **Déclaration**
  - Définit le nom et le type de l'objet
  - Un objet seulement déclaré vaut null
- **Création et allocation de la mémoire**
  - Appelle de méthodes particulières :
    new Constructeurs (...)
  - La création réserve la mémoire et initialise les variables d'instances
  - Renvoi d'une référence sur l'objet nouvellement créé





Le mots réservé super

Le mot clé **super** désigne la superclasse.

- **super** peut être utilisé dans un constructeur pour invoquer le constructeur de la **super-classe**.

```
public unConstructeur(){
    super (3,4);  // Invocation du constructeur
                  // de la super-classe
      :
}
```

- **super** peut être utilisé également pour invoquer la version originale d'une méthode qui a été redéfinie.

```
public void m(){
   // Redéfinition de la méthode m
   super.m ();  // Invocation de la méthode originale
      :
}
```

Le mot réservé this

Le mot-clé **this** désigne l'objet qui reçoit le message.

- **this** peut être utilisé dans un constructeur pour invoquer un autre constructeur défini dans la même classe.

```
public constructeurNo1(int val){
  x = val;
  y = val ;
}
 public constructeurNo2(){
  this (0);  // invocation du premier constructeur
  :
}
```

- **this** est souvent utilisé comme paramètre effectif, pour offrir à un autre objet une référence sur soi-même.

```
nomMaitre.tonEsclave(this);
   // pour signigier à l'objet "monMaitre" qu'il dispose
   // en ma personne d'un nouvel esclave
```

Encapsulation

Lorsqu'on écrit un programme à plusieurs, on veut « protéger » son code des maladresses des autres programmeurs.

- Ou même de soi-même, si on y revient 6 mois plus tard, dans l'urgence par exemple.
- On peut rendre les attributs inaccessibles à l'extérieur de la classe :

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

Si certains attributs doivent néanmoins être accessibles, on peut définir des méthodes d'accès :

```
class Point {
      private int x, y ;
      int get_x() { return this.x ; }
      void set_x(int new_x) { this.x = new_x ; }
}
```

On appelle parfois ces méthodes **getter** et **setter**.

Encapsulation

Ce qu'il ne faut pas faire non-plus

- Certains pensent qu'il suffit de définir des méthodes d'accès pour écrire du code da,s le style Orienté Objet, car alors, la règle qui veut qu'on accède aux attributs uniquement dans leur propre classe est formellement respectée :

```
pt.setX(pt.getX()+deltax);
pt.setY(pt.getY()+deltay);
```

JAMAIS !

- Il existe en Java une construction qui permet de spécifier un ensemble de messages : l'interface.

```
interface Deplacable{
    void deplaceToi( int dx, int dy);
}
```

- Dans une interface, on donne seulement des en-têtes de méthodes, sans leur corps.

Encapsulation

- L'encapsulation par l'utilisation d'interfaces est formellement parfaite: l'utilisateur d'un objet ne connaît pas sa structure (ses variables).
- Mais bien sûr, il suffit de truffer les interfaces de getters et setters pour réduire cette encapsulation à une encapsulation de pure forme.
- L'encapsulation alourdit l'écriture programme.
  - Attributs privés
  - Utilisation d'interfaces pour les types des variables
- Le but à atteindre, c'est le découplage des classes entre-elles.
  - Pouvoir changer une classe sans changer aucune autre
  - Pouvoir penser à une classe en oubliant les autres