
# Chapitre 4: L'approche objets
# Classes et objets

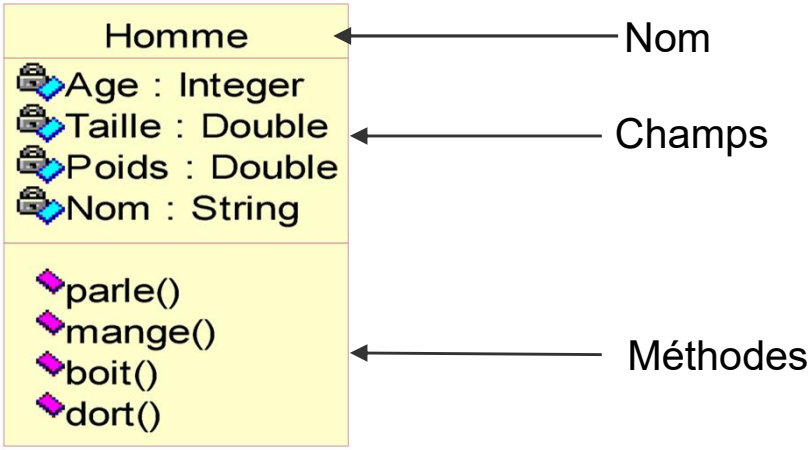
Par Aïcha El Golli

aicha.elgolli@essai.ucar.tn

# LA CLASSE (1) : DÉFINITION

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

# La classe (2) : représentation graphique




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

La **programmation orientée objet** met les objets au cœur de la conception des applications (contrairement à la programmation procédurale qui centre la conception sur les traitements appliqués aux données).

Une **application orientée objet** est constituée d'un ensemble d'objets qui communiquent en s'échangeant des **messages** (par l'invocation de leurs méthodes).

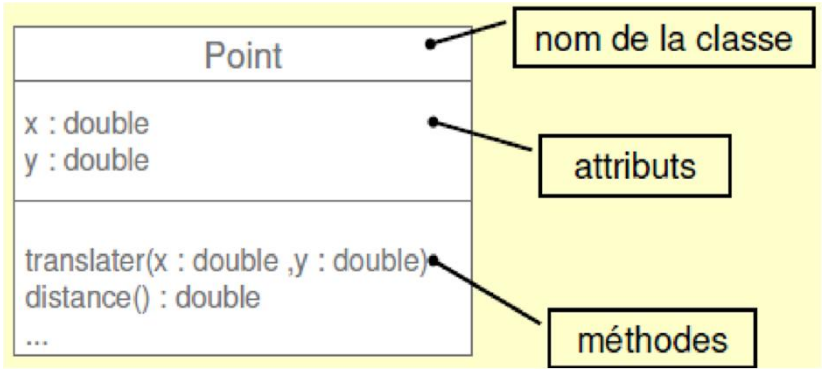
La syntaxe de base pour la **déclaration d'une classe** est la suivante :

```
modificateurs class nom_de_la_classe {
déclaration_de_champs
déclaration_de_constructeurs
déclaration_de_méthodes
}
```

Les **modificateurs** (liste de mots-clés) seront vus ultérieurement (public pour l'instant).

Les **noms de classes** sont habituellement écrits avec la première lettre en majuscule

# SYNTAXE DE DÉFINITION D'UNE CLASSE




Nom de la Classe

Exemple : Une classe définissant un point

```
public class  Point{
  double x; // abscisse du point
  double y; // ordonnée du point
  public Point() { // Constructeur 1
    x = 0.0;
    y = 0.0;
  }
  public Point(double px, double py) { // Constructeur 2
    x = px;
    y = py;
  }

// translate de point de (dx,dy)
  public void translater (double dx, double dy) {
    x = x+dx;
    y = y+dy;
  }
 // calcule la distance du point à l'origine
  public double distance() {
    return Math.sqrt(x*x+y*y);
          }
}
```

Attributs

Méthodes

# LA CLASSE

```
public class Point{
  private double x; // abscisse du point
  private double y; // ordonnée du point
/**
 * Constructeur 1 (par défaut)
 */
  public Point() {
    x = 0.0;
    y = 0.0;
    }
  /**
   * Constructeur 2 (par complet)
   */
  public Point(double px, double py) {
    setX(px);
    setY(py);
  }

 public double getX() {
return x;
}
public void setX(double x) {
this.x = x;
}
public double getY() {
return y;
}
public void setY(double y) {
this.y = y;
}
/** translate de point de (dx,dy)
  * @param dx
  * @param dy
  */
public void translate (double dx, double dy) {
    SetX( x+dx);
    SetY( y+dy);
  }

 /**calcule la distance du point à l'origine
  *
  * @return la distance calculée du point à l'origine
  */

  public double distance() {
    return Math.sqrt(x*x+y*y);
}
}
```

# ATTRIBUT

- utilisation dans le cas général : usage de l'opérateur **. ("point")**

Point p = new Point();
double abs=p**.x;**

- utilisation depuis une méthode :

accès direct pour l'instance courante

```
class Point {
// ...
double distance(Point autre){
return Math.sqrt((x-autre.x)*(x-autre.x)
+(y-autre.y)*(y-autre.y));}  }
```

### Déclaration des attributs

Une déclaration d'attribut est de la forme :

```
type nomAttribut;

ou

type nomAttribut = expressionInitialisation;
```

type simple (pas Objet) :
char
int
byte
short
long
double
float
boolean

type structuré (Objet) : type est le nom d'une classe connue dans le contexte de compilation et d'exécution

Nécessaire si votre classe utilise une autre classe d'un autre package et qui n'est pas le package java.lang

```
import java.awt.Color;

class Point {
   double x = 0;
   double y = 0;
   Color c;

   ...
}
```

Un point a une couleur définie par un objet de type Color

### Syntaxe JAVA : visibilité des attributs

Les attributs sont des variables « globales » au module que constitue la classe : ils sont accessibles dans toutes les méthodes de la classe.

```
class Point {
   double x;
   double y;

   void translater(double dx,double dy){
      x += dx;
      y += dy;
   }

   double distance() {
      double dist;
      dist = Math.sqrt(x*x+y*y);
      return dist;
   }
}
```

# MÉTHODE


- appel "direct" depuis une autre méthode de la même classe :

```
class Point {
//...
void afficherDistance(){
double a=distance();
System.out.println(a);
}  }
```

- appel des méthodes dans les autres cas :
  - usage de l'opérateur **. (point) :**

```
Point p1 = . . . ;
double d=p1.distance();
```

- surcharge : plusieurs méthodes de même nom, mais de *signatures différentes (i.e.* avec types de paramètres différents)

```
class Point {
   double x;
   double y;

   void translater(double dx,double dy){
         x += dx;
         y += dy;
   }

   double distance() {
         double dist;
         dist = Math.sqrt(x*x+y*y);
         return dist;
   }
}
```

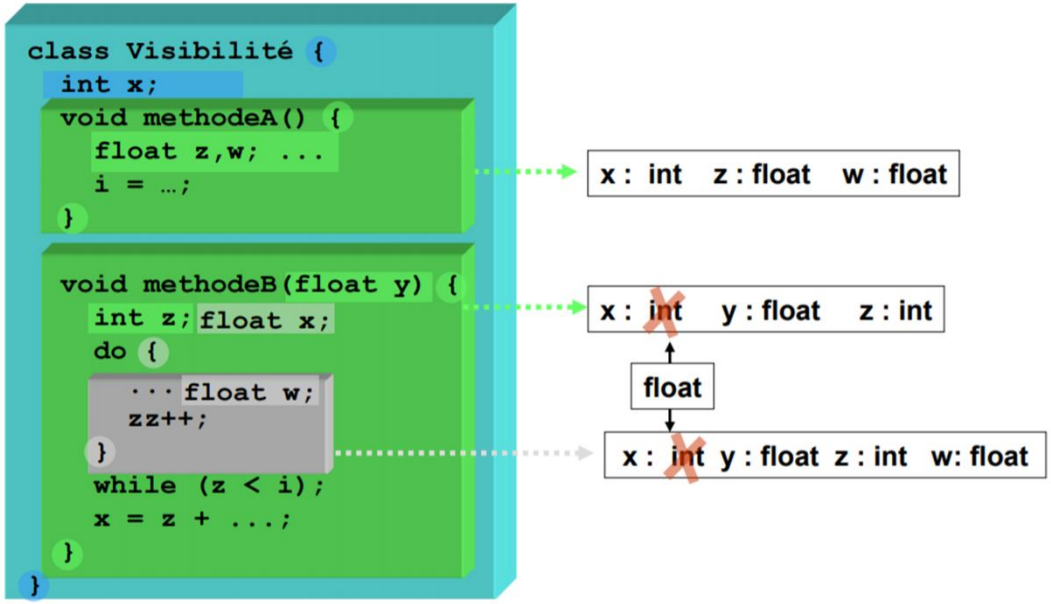
Deux classes différentes peuvent avoir des membres de nom identique

```
class Cercle {
   double x;  // abscisse du centre
   double y;  // ordonnée du centre
   double r;  // rayon

   void translater(double dx,double dy){
         x += dx;
         y += dy;
   }

   ...
}
```

Attention !! la redéfinition d'une variable masque la définition au niveau du bloc englobant.



# LA RÉFÉRENCE « THIS »

Le mot clé **this** est utilisé pour référencer **l'objet à l'intérieur duquel le code est exécuté**.

On peut considérer que l'objet this est **implicitement passé en référence** lors de l'invocation de chaque méthode d'instance possibilité dans les méthodes de désigner *explicitement l'instance courante:*

- pour accéder aux attributs "masqués" par paramètres :

```
class Cercle {
double rayon=0;
// ...
double comparerA(double rayon){
return this.rayon-rayon;
} }
```

- pour appeler une fonction (ou méthode) avec l'instance courante en paramètre :

```
class Cercle {
void dessinerSur(Ecran e){
// ...
}}
class Ecran {
// ...
void tracer(Cercle c){
c.dessinerSur(this);
}}
```

# L'INSTANCIATION

- Instanciation : concrétisation d'une classe en un objet « *concret* ».
  - Dans nos programmes Java nous allons définir des classes et ***instancier*** ces classes en des objets qui vont interagir. Le fonctionnement du programme résultera de l'interaction entre ces objets « ***instanciés*** ».
  - En Programmation Orientée Objet, on décrit des classes et l'application en elle-même va être constituée des objets instanciés, à partir de ces classes, qui vont communiquer et agir les uns sur les autres.

Instance

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

Exemple : si nous avons une classe voiture, alors votre voiture est une instance particulière de la classe voiture.

- Classe = concept, description
- Objet = représentant ***concret*** d'une classe

# OBJET: INSTANCIATION D'UNE CLASSE

instance d'une classe création uniquement par **new** :

```
Point p1;  //p1 : référence non initialisée

p1 = new Point(); //p1 : référence à une
 instance
```

destruction automatique par le "Garbage Collector" quand il n'y a plus aucune référence vers l'objet :

```
Point p2=new Point();

// ...

p2=p1;

// => l'objet Point créé par new pour le p2
initial sera détruit.
```

# ENCAPSULATION ET CONTRÔLE D'ACCÈS

Pour chaque attribut et chaque méthode d'une classe visibilité possible :

– **public:** visible depuis les instances de toutes les classes d'une application. En d'autres termes, son nom peut être utilisé dans l'écriture d'une méthode de ces classes.

– **private:** visible uniquement depuis les instances de sa classe. En d'autres termes, son nom peut être utilisé uniquement dans l'écriture d'une méthode de sa classe.

– **protected**

– par défaut, accès "**package**"

# Encapsulation et contrôle d'accès

En toute rigueur, il faudrait toujours que :

- les attributs ne soient pas visibles,
  - Les attributs ne devraient pouvoir être lus ou modifiés que par l'intermédiaire de méthodes prenant en charge les vérifications et effets de bord éventuels.
- les méthodes "utilitaires" ne soient pas visibles,
- seules les fonctionnalités de l'objet, destinées à être utilisées par d'autres objets soient visibles.

⇒C'est la notion d'encapsulation

Nous verrons dans la suite que l'on peut encore affiner le contrôle d'accès

- les attributs déclarés comme privées (**private**) sont totalement protégés
  - *ne sont plus directement accessibles depuis le code d'une autre classe*

code écrit en dehors de la classe Point

```
Point p1 = new Point();
p1.x = 10; p1.setX(10);
p1.y = 10; p1.setY(10);
Point p2 = new Point();
p2.x = p1.x;
p2.y = p1.x + p1.y;
p2.setX(p1.getX());
p2.setY(p1.getX()+p1.getY());
```

- *pour les modifier il faut passer par une méthode de type procédure*
- *pour accéder à leur valeur il faut passer par une méthode de type fonction*

```
public class Point {
   private double x;
   private double y;
    public void translater(int dx, int dy) {
         x += dx; y += dy;
    }
    public double distance() {
         return Math.sqrt(x*x+y*y);
    }
    public void setX(double x1){
         x = x1;
    }
    … idem pour y
    public double getX(){
         return x;
    }
    … idem pour y
}
```

getters et setters : standard JavaBeans peuvent être générés automatiquement par IDE

Un objet ne peut être utilisé que de la manière prévue à la conception de sa classe, sans risque d'utilisation incohérente -> Robustesse du code.

# LES CONSTRUCTEURS (1)

L'appel de new pour créer un nouvel objet déclenche, dans l'ordre :

- L'allocation mémoire nécessaire au stockage de ce nouvel objet et l'initialisation par défaut de ces attributs,
- L'initialisation explicite des attributs, s'il y a lieu,
- L'exécution d'un constructeur.

Un constructeur est une méthode d'initialisation.

# LES CONSTRUCTEURS (2)

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

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

Un **constructeur** est une sorte de méthode qui sera invoquée lors de la création d'un objet de cette classe (opérateur **new**).

Un **constructeur** doit porter le **nom de la classe** et ne doit **pas** comporter **de type de retour** dans sa déclaration (même pas void).

Le but du constructeur est d'**initialiser l'objet** (notamment la valeur de ses champs).

Un constructeur retourne implicitement une **référence** à une instance de la classe (objet).

Si aucun constructeur n'est défini dans une classe, *Java* fournit un **constructeur par défaut**, sans argument, et qui se charge uniquement de créer les champs et de leur attribuer une valeur initiale (valeur par défaut ou valeur de l'expression d'initialisation).

# LES CONSTRUCTEURS (3)

Personne.java

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

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

```
public class Application
{
  public static void main(String args[])
  {
    Personne ali = new Personne()
    ali.nom=" Ali" ;
  } }
```

Va donner une erreur à la compilation

# LES CONSTRUCTEURS (4)

Comme les méthodes, les constructeurs peuvent être **surchargés** (en suivant les même règles).

Lorsqu'une classe possède des constructeurs multiples (surchargés), il est possible, dans le corps d'un constructeur, d'invoquer un autre constructeur, un utilisant la syntaxe spéciale :

**this(***expr1*, *expr2*, ...**)**

Dans la déclaration de la classe **Point**, le premier constructeur aurait pu être écrit ainsi :

```
public Point() {          // Constructeur 1
    this(0.0, 0.0);               // Invocation du Constructeur 2
}
```

***Restriction importante* : *L'appel à* this(...) *doit apparaître comme première instruction dans un constructeur.***

Attention :Si le programmeur crée un constructeur (même si c'est un constructeur avec paramètres), le constructeur par défaut n'est plus disponible. Attention aux erreurs de compilation !

# LES CONSTRUCTEURS (5)

Personne.java

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

Redéfinition d'un Constructeur sans paramètres

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

# CONSTRUCTEUR (6)

On peut invoquer un constructeur depuis un autre avec this(. . .) :

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

conseil :

– faire d'abord un constructeur "complet" avec tous les paramètres possibles,

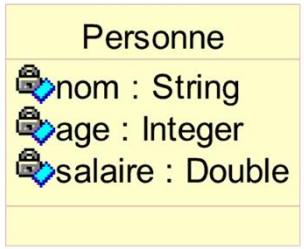
– faire en sorte que tous les autres constructeurs (en général un constructeur *sans argument et* un *pour construire par recopie d'un objet de même type) appellent le constructeur "complet"*

# CLASSE ET OBJET EN JAVA

Du modèle à …

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

... des instances de cette classe




```
class Personne
{  String nom;
 int age;
 double salaire;
public Personne(String n, String p, double a)
  {    nom=n;   prenom=p; salaire = a;  }
public Personne()
 {this(null,null,0);     //appelle personne(null,null,0)   }
}
```

```
Personne jean, pierre;
jean = new Personne ();
pierre = new Personne ();
```

L'opérateur d'instanciation en Java est new :

MaClasse monObjet = new MaClasse();

- La classe **Personne** permet de déclarer des objets de ce type :

Personnes jean; // Déclaration d'un objet de type Personne

- **La déclaration ne crée pas un objet** mais uniquement une **référence vers un objet** du type mentionné (comme pour les tableaux).
- Pour **créer un objet** on utilise l'opérateur **new** suivi du nom de la classe et d'une liste facultative d'arguments entre parenthèses.

**new** *nom_de_la_classe*(*expr1*, *expr2*, *...* **)**

new Personne(« jean », « blin », 2000)

- L'opérateur new fait appel à l'un des **constructeurs** de la classe en fonction du profil des paramètres transmis (constructeur par défaut ou l'un des constructeurs définis explicitement dans la classe).

# OBJETS, TABLEAUX, TYPES DE BASE

Lorsqu'une variable est d'un type objet ou tableau, ce n'est pas l'objet ou le tableau lui-même qui est stocké dans la variable mais une référence vers cet objet ou ce tableau (on retrouve la notion d'adresse mémoire ou du pointeur en C).

Lorsqu'une variable est d'un type de base, la variable contient la valeur.

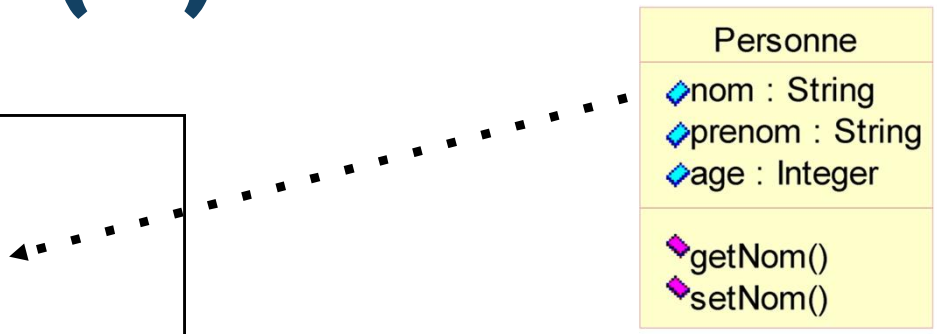
La référence est, en quelque sorte, un pointeur pour lequel le langage assure une manipulation transparente, comme si c'était une valeur (pas de déréférencement).

Par contre, du fait qu'une référence n'est pas une valeur, c'est au programmeur de prévoir l'allocation mémoire nécessaire pour stocker effectivement l'objet (utilisation du new).

# ACCÈS AUX ATTRIBUTS D'UN OBJET (1)

Personne.java

```
public class Personne
{
  Private String nom;
  private String prenom;
  private int age;
  public void setNom(String unNom)
   {
     nom = unNom;
   }
  public String getNom()
  {
    return (nom);
  }
}
```



# ACCÈS AUX ATTRIBUTS D'UN OBJET (2)

Application.java

```
public class Application
{
  public static void main(String args[])
  {
    Personne jean = new Personne()
    jean.setNom ("Jean" ) ;
    jean.setPrenom ( "Pierre" );
  }
}
```

Remarque :

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

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

# NOTION DE MÉTHODES ET DE PARAMÈTRES (1)

La notion de méthodes dans les langages objets

Proches de la notion de procédure ou de fonction dans les langages procéduraux.

La méthode c'est avant tout le regroupement d'un ensemble d'instructions suffisamment générique pour pouvoir être réutilisées

Comme pour les procédures ou les fonctions (au sens mathématiques) on peut passer des paramètres aux méthodes et ces dernières peuvent renvoyer des valeurs (grâce au mot clé return).

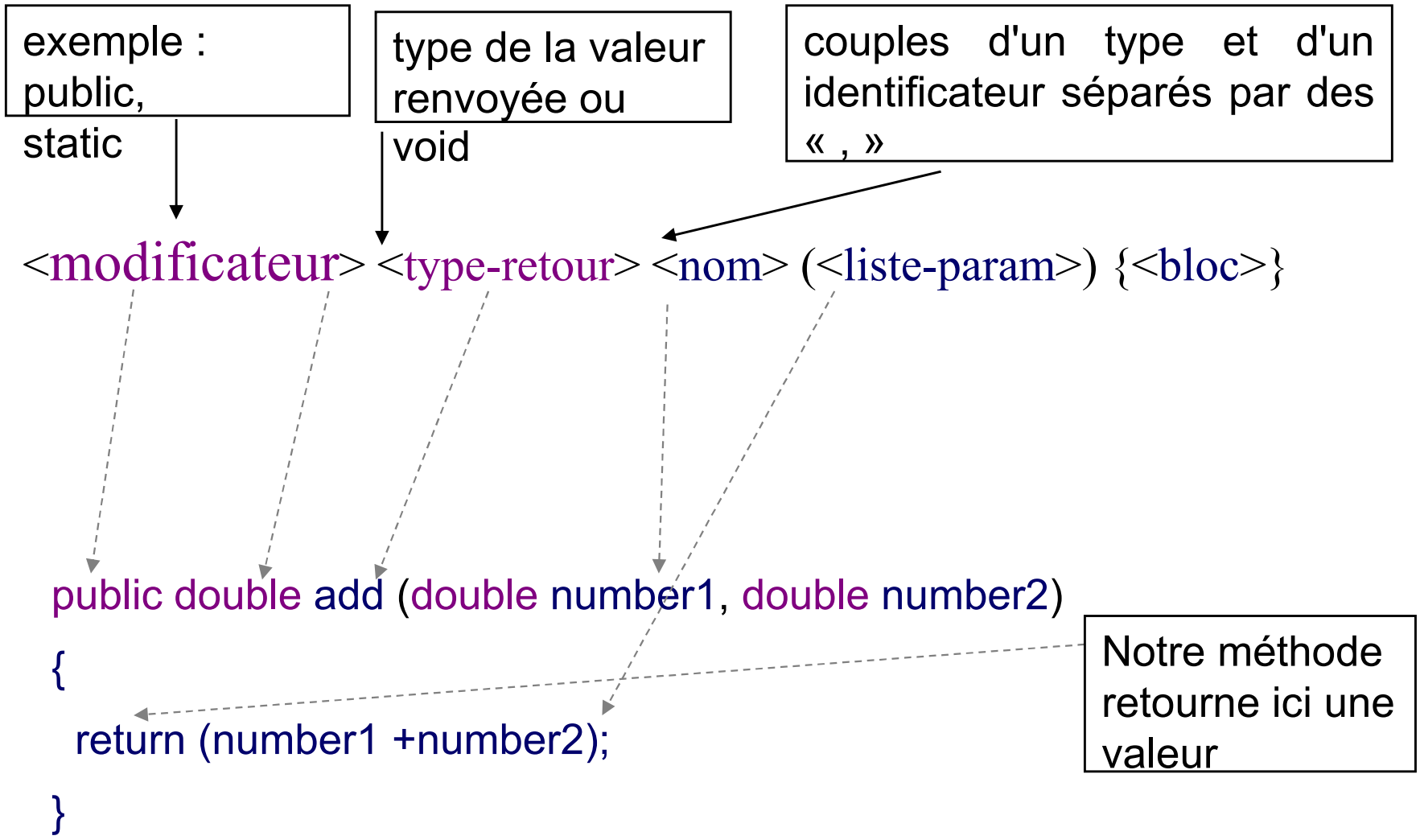
# MODE DE PASSAGE DES PARAMÈTRES

Java n'implémente qu'un seul mode de passage des paramètres à une méthode : le passage par valeur.

Conséquences :

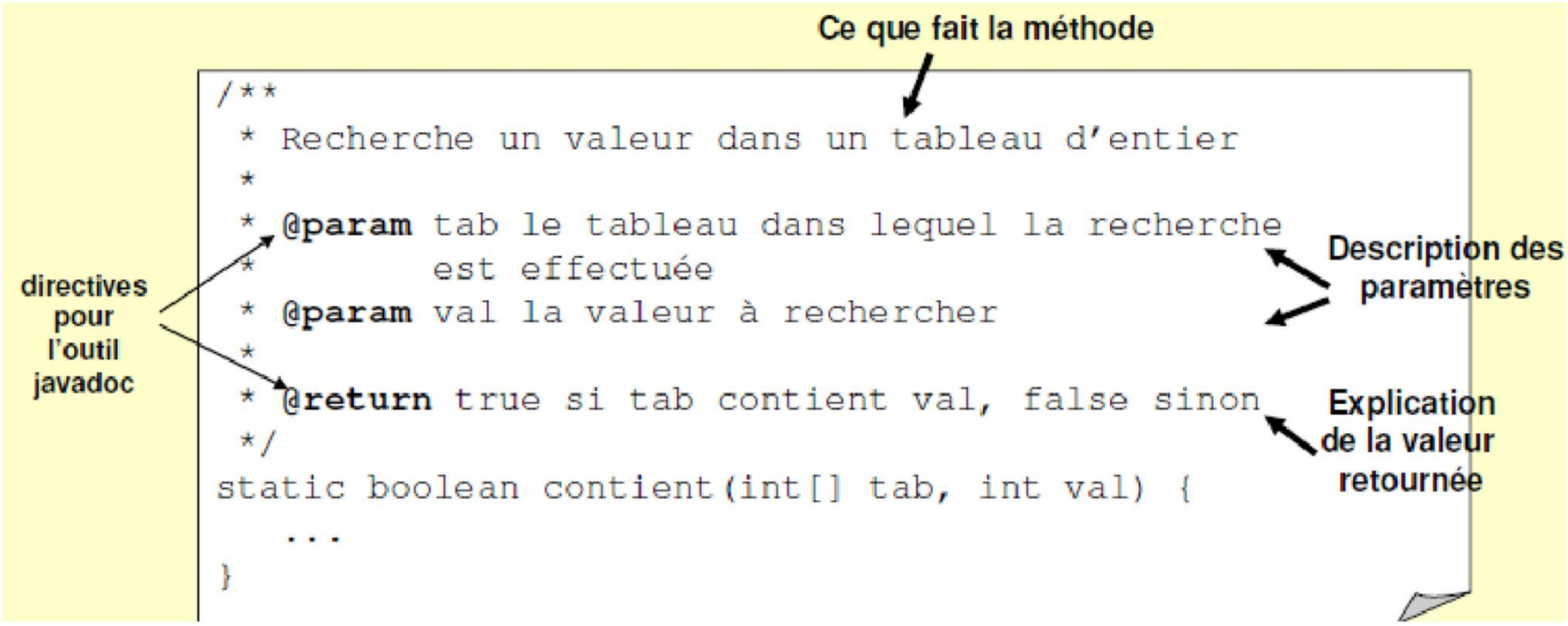
- l'argument passé à une méthode ne peut être modifié,
- si l'argument est une instance, c'est sa référence qui est passée par valeur. Ainsi, le contenu de l'objet peut être modifié, mais pas la référence elle-même.

# NOTION DE MÉTHODES ET DE PARAMÈTRES (2)



# COMMENTAIRE DES FONCTIONS

Toute déclaration de méthode doit **TOUJOURS** être précédée de son commentaire documentant (exploité par l'outil javadoc)



# PORTÉE DES VARIABLES (1)

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

```
public class Portee
{
  int i, j, k;
  public void meth1(int z)
  {
     int j,r;
     r = z;
    j=6;
   this.j=r+j;

  }
 public void meth2()
 {
   k = r;
 }
}
```

Ce sont 2 variables différentes

k est connu au niveau de la méthode meth2() car déclaré dans le bloc de la classe. Par contre r n'est pas défini pour meth2(). On obtient une erreur à la compilation

# PORTÉE DES VARIABLES (2)

En cas de conflit de nom entre des variables locales et des variables globales, c'est toujours la variable la plus locale qui est considérée comme étant la variable désignée par cette partie du programme

?Attention par conséquent à ce type de conflit quand on manipule des variables *globales*.

C'est le j défini en local qui est utilisé dans la méthode meth1()

```
public class Portee
{
  int i, k, j;
  public void meth1(int z)
  {
     int j,r;
     j = z;
  }
}
```

# DESTRUCTION D'OBJETS

Java n'a pas repris à son compte la notion de destructeur telle qu'elle existe en C++ par exemple.

C'est le ramasse-miettes (ou Garbage Collector - GC en anglais) qui s'occupe de collecter les objets qui ne sont plus référencés.

Le ramasse-miettes fonctionne en permanence dans un thread de faible priorité (en « *tâche de fond* »). Il est basé sur le principe du compteur de références.

Il est possible au programmeur d'indiquer ce qu'il faut faire juste avant de détruire un objet.

C'est le but de la méthode finalize() de l'objet.

Cette méthode est utile, par exemple, pour :

- fermer une base de données,
- fermer un fichier,
- couper une connexion réseau,
- etc.

# TABLEAUX D'OBJETS

ATTENTION : un tableau d'objets est en fait un tableau de **références vers objet :**

```
Point [] tabP= new Point[10];  // tabP[0]==null
```

il faut donc allouer les objets eux-mêmes ensuite :

```
for (int i=0; i<10; i++)
 tabP[i]=new Point();
```

# VARIABLES/ATTRIBUT DE CLASSE

Il peut s'avérer nécessaire de définir un attribut dont la valeur soit partagée par toutes les instances d'une classe. On parle de variable de classe.

Ces variables sont, de plus, stockées une seule fois, pour toutes les instances d'une classe.

Mot réservé : static

Accès :

- depuis une méthode de la classe comme pour tout autre attribut,
- via une instance de la classe,
- à l'aide du nom de la classe.

# VARIABLES/ATTRIBUT DE CLASSE

```
public class Personne
{
  public static int compteur = 0;
  public Personne ()
  {
    compteur++;
  }
}

public class AutreClasse
{
 public void uneMethode()
  {
    int i = Personne.compteur;
  }
}
```

Variable de classe

Utilisation de la variable de classe compteur dans le constructeur de la classe

Utilisation de la variable de classe compteur dans une autre classe

# MÉTHODES DE CLASSE

Il peut être nécessaire de disposer d'une méthode qui puisse être appelée sans instance de la classe. C'est une méthode de classe.

On utilise là aussi le mot réservé **static**

Puisqu'une méthode de classe peut être appelée sans même qu'il n'existe d'instance, une méthode de classe ne peut pas accéder à des attributs non statiques. Elle ne peut accéder qu'à ses propres variables et à des variables de classe.

Signalons enfin qu'une méthode **static** ne peut pas être redéfinie, ce qui signifie qu'elle est automatiquement **final**.

On trouve un certain nombre de méthodes statiques dans l'API. C'est le cas de toutes les méthodes de la classe java.lang.Math : on pourra ainsi utiliser Math.sin(...), Math.log(...) , de même que la constante statique Math.PI.

# PASSAGE DES PARAMÈTRES

- **par valeur** pour les types "primitifs" :

```
public static void echange(int i, int j){
   // i et j = copies locales des valeurs
   int tmp=i;
   i = j;
   j = tmp;
}
public static void main(String[] args){
   int a=1,b=2;
   echange(a,b);
   // a et b ne sont pas modifiés
}
```

---

- **par valeur (i.e. recopie)** des références :

```
public static void ech(String s1,
                       String s2) {
   // s1 et s2 = copies locales des références
   String tmp=s1;
   s1 = s2;
   s2 = tmp;
}
public static void main(String[] args) {
   String a="oui",b="non";
   ech(a,b);
   // a et b ne sont pas modifiés
}
```

# PASSAGE DES PARAMÈTRES

- la référence est passée par valeur (i.e. le paramètre est une copie de la référence), **mais le contenu de l'objet référencé peut être modifié par la fonction** (car la copie de référence pointe vers le même objet…) :

```
public static void increm(int t[]) {
  for (int i=0; i<t.length; i++)
     t[i]++;
}

public static void tronquer(StringBuffer b){
  b.deleteCharAt(b.length()-1);
}

public static void main(String[] args){
  int tab[]={1,2,3};
  increm(tab);
  // tab vaut {2,3,4}
  StringBuffer buf=new StringBuffer("oui");
  tronquer(buf);
  // buf vaut "ou"
}
```

# SURCHARGE

- En Java, il y a possibilité de définir (dans la même classe) plusieurs fonctions de même nom, différenciées par le type des arguments :

```
class NomClasse {
 public static int longueur(String s){
   return s.length();
 }
 public static int longueur(int i){
   String chiffres;
   chiffres=String.valueOf(i);
   return chiffres.length();
 }
 public static void main(String [] args) {
    int k=12345;
    String s="oui";
    int lk=longueur(k);   // lk vaut 5
    int ls=longueur(s);   // ls vaut 3
 }
}
```

- appel de fonction : les arguments doivent être du type prévu ou d'un type convertible automatiquement vers le type prévu

# FONCTIONS À NOMBRE VARIABLE D'ARGUMENTS

- Possibilité de *terminer* la liste des paramètres d'une fonction par une « ellipse » (`type...`) permettant un appel avec en argument un nombre variable (éventuellement nul) d'argument(s) *de même type* (sauf si le type est `Object`)
- Exemple :

```
public static double[] f(int exp,
                         double... nb) {
  double[] res = new double[nb.length];
  for (int i=0; i<res.length; i++)
    res[i] = Math.pow(nb[i],exp);
  return res;
}
//…
double[] t1 = f(2, 2.5);
double[] t2 = f(2, 1.1, 2.2, 3.3);
double[] t3 = f(2);
```

- Le paramètre avec « ellipse » est traité dans la fonction comme un tableau, et au niveau des appels comme un nombre quelconque d'arguments de même type.
- Possibilité d'arguments de types hétérogènes en utilisant `Object...` comme paramètre d'ellipse.

# CLASSE POINT (en java)

```
public class Point {

private double x;

private double y;

Point(){this(0.,0.)}

Point(double x, double y){

setX(x);setY(y);}

public double getX() {

return x;}

public void setX(double x) {

this.x=x;}
```

```
public double getY() {
return y;}

public void setY(double y)
{this.y=y;}

boolean compare(Point p){
return ((p.x==x)&&(p.y==y));}

public String toString() {
return (“(“+x+”, “+y+”)”);
} }
```

# CLASSE POINT (en Python)

```
class Point2D:

    def __init__(self, x=0.0, y=0.0):
        self.set_X(x)
        self.set_Y(y)

    def get_X(self):

        return self.__x

    def get_Y(self):

        return self.__y
    def set_X(self, x):
        self.__x = x

    def set_Y(self, y):
        self.__y = y
    def compare(self, p):
        return (self.__x == p.__x) and (self.__y == p.__y)
    def __str__(self):
        return "("+str(self.__x)+", "+str(self.__y)+")"
```