# Chapitre IV – Classes, attributs, méthodes

# Généralités

On s'intéresse d'abord à aux entités que l'on va manipuler avant de s'intéresser à la façon de les manipuler.
Une classe est un bloc de base d'un programme Java.
Définir une classe c'est construire un type de données structuré grâce à ses *attributs* et lui adjoindre des *méthodes* permettant sa manipulation (initialisation, modification, consultation, suppression ...) par l'intermédiaire de ses *attributs*.
Une classe est donc un modèle pour un ensemble de données.
On trouve donc la notion d'*encapsulation* : sous une même entité - la *classe* - sont regroupées les données - les *attributs* - et les moyens - les *méthodes* - de les manipuler.
Un objet est *une instance* d'une classe - une ≪ application concrète ≫ de cette classe pour un usage particulier dans un programme.

# Généralités

D'une façon générale, toute classe est construite ainsi :

```
class nomClasse {
// les attributs
// les méthodes
// des classes internes éventuelles
```

Un attribut est une variable mais toute variable n'est pas un attribut puisqu'elle peut être locale à une méthode.
Parmi les méthodes il y a des *constructeurs* de même nom que la classe, **sans type de retour** : ils servent à créer des objets.

# Généralités

D'une façon générale, toute classe est construite ainsi :

```
class nomClasse {
// les attributs
// les méthodes
// des classes internes éventuelles
```

Un attribut est une variable mais toute variable n'est pas un attribut puisqu'elle peut être locale à une méthode.

Parmi les méthodes il y a des *constructeurs* de même nom que la classe, **sans type de retour** : ils servent à créer des objets.

# Généralités

D'une façon générale, toute classe est construite ainsi :

```
class nomClasse {
// les attributs
// les méthodes
// des classes internes éventuelles
```

Un attribut est une variable mais toute variable n'est pas un attribut puisqu'elle peut être locale à une méthode.
Parmi les méthodes il y a des *constructeurs* de même nom que la classe, **sans type de retour** : ils servent à créer des objets.

# Un premier exemple

```
public class Point{
 private float abs;
 private float ord;
 //constructeurs
public Point(float xa, float xb){
abs=xa;ord=xb;}

public Point(){}

//méthodes en écriture
public void setAbs(float x){
this.abs=x;
}
public void setOrd(float y){
 this.ord=y;
}
//méthodes en lecture
public float getAbs(){
return this.abs;
}
public float getOrd(){
return this.ord;
}
```

# suite

```
// méthode statique
public static boolean sontAlignes(Point A, Point B, Point C){
return ((B.ord−A.ord)(C.abs−A.abs)==
(C.ord−A.ord)(B.abs−A.abs));
}
// méthode d'instance
public  void afficher(){
System.out.println("abscisse =" + this.abs +
", ordonnée=" + this.ord);
}}
```

this Surcharge Accesseurs constructeur

# Attributs et méthodes : généralités

Dans la définition d'une classe il y a deux catégories d'attributs et de méthodes :

- les attributs et méthodes de classe ou statique ; ils sont précédés du modificateur `static`
- les attributs et méthodes d'instance

A l'intérieur de la définition de la classe les attributs et méthodes statiques de cette classe seront accessibles directement sans préfixer leur nom par le nom de la classe. Par exemple sontAlignes.
Dans un programme dès qu'une classe est invoquée, le bytecode de cette classe est chargée en mémoire et les attributs et méthodes statiques n'étant liés qu'à la classe et non à une instance sont immédiatement disponibles. Même si plusieurs instances de cette classe existent **il n'y aura qu'une version d'un attribut de classe**.

# Attributs

Si une variable est déclarée `static`, tous les objets de la classe partageront cette variable. Donc, toute modification d'une variable statique dans un objet quelconque de la classe est répercutée dans tous les objets de la classe.

# Exemple

```
public class UneClasse {
int x;
static int y ;
}
public class EssaiUneClasse{
public static void main ( String [ ] arg ) {
UneClasse c= new UneClasse();
UneClasse cc= new UneClasse();
c.x=1;
c.y=2;
cc.x=2*c.x;
cc.y=1;
System.out.print(c.x + ",");
System.out.println(c.y);
System.out.print(cc.x + ",");
System.out.println(cc.y);
}}
```

L'affichage donne :

1,1
2,1

- 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 tableaux et types d'objet
- le nom véritable d'un attribut de classe y pour la classe UneClasse est UneClasse.y

# Portée des variables

**Règle** : LES VARIABLES SONT CONNUES ET SEULEMENT CONNUES À L'INTÉRIEUR DU BLOC DANS LEQUEL ELLES SONT DÉCLARÉES.
On parle de variable locale lorsqu'une variable est paramètre d'une méthode ou bien déclarée dans un bloc d'exécution comme dans une méthode par exemple.
Une variable locale n'est pas un attribut et *doit être explicitement initialisée* avant d'être utilisée.
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. Le mieux est d'éviter ce genre de conflit.

# Syntaxe

La syntaxe est la suivant
$\langle$ modificateur $\rangle$ $\langle$ type-retour$\rangle$ $\langle$ nom$\rangle$ ($\langle$ liste-param$\rangle$) {bloc }
avec

- modificateur = public, static ...
- type-retour = type de la valeur renvoyée ou void
- liste-param = couples de `type identificateur` séparés par des virgules.

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

# Sémantique

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.

# Méthode statique

Si une méthode est déclarée static, tous les objets de la classe partageront cette méthode.
Régle : une méthode statique n'a pas directement accès aux variables non statiques.

# Exemple

```
public class EssaiUneMethodeStat {
int x;
public static void main ( String [ ] arg ) {
System.out.print("la valeur x vaut " + x);
}}
```

cela provoque une erreur de compilation car `main` est une méthode statique et `x` est un attribut d'instance.

# Méthode d'instance

Une méthode qui n'est pas déclarée `static` est toujours utilisée *en référence à un objet*. On parle de méthode *d'objet ou d'instance*.

# Appel d'une méthode

- pour appeler une méthode statique : on écrit `nomClasse.nomMethode` si `nomMethode` est une méthode statique de la classe `nomClasse`
- pour appeler une méthode d'objet : on écrit `nomObjet.nomDeMethode(...)`

Par exemple, la classe `System` contient un attribut statique `out` de type `PrintStream`, classe contenant la méthode d'objet `println()`. Donc l'objet `System.out` appelle une méthode d'objet de nom `println()`.

# Appel d'une méthode

- pour appeler une méthode statique : on écrit `nomClasse.nomMethode` si `nomMethode` est une méthode statique de la classe `nomClasse`
- pour appeler une méthode d'objet : on écrit `nomObjet.nomDeMethode(...)`

Par exemple, la classe `System` contient un attribut statique `out` de type `PrintStream`, classe contenant la méthode d'objet `println()`. Donc l'objet `System.out` appelle une méthode d'objet de nom `println()`.

# Appel d'une méthode

- pour appeler une méthode statique : on écrit `nomClasse.nomMethode` si `nomMethode` est une méthode statique de la classe `nomClasse`
- pour appeler une méthode d'objet : on écrit `nomObjet.nomDeMethode(...)`

Par exemple, la classe `System` contient un attribut statique `out` de type `PrintStream`, classe contenant la méthode d'objet `println()`. Donc l'objet `System.out` appelle une méthode d'objet de nom `println()`.

# Appel d'une méthode

- pour appeler une méthode statique : on écrit `nomClasse.nomMethode` si `nomMethode` est une méthode statique de la classe `nomClasse`
- pour appeler une méthode d'objet : on écrit `nomObjet.nomDeMethode(...)`

Par exemple, la classe `System` contient un attribut statique `out` de type `PrintStream`, classe contenant la méthode d'objet `println()`. Donc l'objet `System.out` appelle une méthode d'objet de nom `println()`.

# Désigner l'objet courant dans une méthode d'instance

Une méthode d'instance est appelée par un objet de la classe dans laquelle est définie cette méthode.
Pour désigner cet objet qui appelle la méthode *dans le corps de la méthode* on utilise le mot clef `this`.
(voir la classe `Point` et la méthode `afficher`)

# Surcharge

Une méthode de nom donné peut posséder plusieurs définitions une même classe, chacune de ces définitions se distinguant des autres au travers de la liste de ses paramètres.
Par contre le type du résultat n'intervient pas dans cette différenciation.
Par exemple, la classe Point pourrait avoir une autre méthode à 4 paramètres

```
public static boolean sontAlignes(Point A, Point B,
Point C, Point D){... }
```

# Exemple

```
class EssaiThis {
// méthode d'instance
void faire(EssaiThis instance){
System.out.print(instance == this) ;
}
public static void main(String[] args) {
EssaiThis instance1 =new EssaiThis ;
EssaiThis instance2 =new EssaiThis ;
instance1.faire(instance2) ;
instance1.faire(instance1) ;
}}
```

Quel sera l'affichage à l'exécution de la méthode `main` ?

false true

# Exemple

```
class EssaiThis {
// méthode d'instance
void faire(EssaiThis instance){
System.out.print(instance == this) ;
}
public static void main(String[] args) {
EssaiThis instance1 =new EssaiThis ;
EssaiThis instance2 =new EssaiThis ;
instance1.faire(instance2) ;
instance1.faire(instance1) ;
}}
```

Quel sera l'affichage à l'exécution de la méthode `main` ?

`false` `true`

# Exemple

```
class EssaiThis {
// méthode d'instance
void faire(EssaiThis instance){
System.out.print(instance == this) ;
}
public static void main(String[] args) {
EssaiThis instance1 =new EssaiThis ;
EssaiThis instance2 =new EssaiThis ;
instance1.faire(instance2) ;
instance1.faire(instance1) ;
}}
```

Quel sera l'affichage à l'exécution de la méthode `main` ?

`false true`

# Modificateur `final`

- `final` unAttribut : aucune modification de unAttribut après son initialisation
- `final` uneMéthode : aucune modification de uneMéthode après sa déclaration (donc pas de redéfinition dans une sous-classe)
- `static final` ... uneValeur = ... : uneValeur est une constante.

# Modificateur `final`

- `final` unAttribut : aucune modification de unAttribut après son initialisation
- `final` uneMéthode : aucune modification de uneMéthode après sa déclaration (donc pas de redéfinition dans une sous-classe)
- `static final` ... uneValeur = ... : uneValeur est une constante.

# Modificateur `final`

- `final` unAttribut : aucune modification de unAttribut après son initialisation
- `final` uneMéthode : aucune modification de uneMéthode après sa déclaration (donc pas de redéfinition dans une sous-classe)
- `static final` ... uneValeur = ... : uneValeur est une constante.

Java fournit 3 niveaux de protection pour les membres d'une classe. Chaque attribut et chaque méthode d'une classe peut être :

- 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. Les données peuvent modifiées par une méthode de la classe, d'une autre classe ou depuis la fonction `main`. Il est alors déclaré avec le modificateur `public`.
- 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**. Il est alors déclaré avec le modificateur `private`.
- visible uniquement depuis les instances de sa classe et de ses sous-classes. Il est alors déclaré avec le modificateur `protected`.

Par défaut si les données sont déclarées sans type de protection, elles sont `public`.

## bonnes habitudes

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

C'est la notion d'*encapsulation* : les données doivent pouvoir être contrôlées ainsi que les comportement de l'objet. On vient de voir la protection des données. Maintenant on va voir le contrôle de l'accès puis les constructeurs.

# Méthodes d'accès aux données

Lorsque les données sont totalement protégées `private`, elles ne sont plus accessibles depuis une autre classe ou depuis la fonction `main` d'un autre programme.
Il faut donc créer *dans la classe* des méthodes d'accès aux données de la classe.
Il y a deux types d'accès à envisager :

- en consultation ou lecture, on parle d'accesseur en consultation,
- en modification ou écriture, on parle d'accesseur en modification.

Par convention les méthodes d'accès en lecture doivent commencer par `get` et les méthodes d'accès en écriture doivent commencer par `set`.
(voir la classe `Point`)

# Constructeurs

Un constructeur est une méthode particulière qui sert à initialiser un objet.
Si on omet d'écrire explicitement un constructeur dans une classe alors Java crée un *constructeur par défaut* qui a les caractéristiques suivantes :

- il a le nom de la classe
- son en-tête ne contient aucun type de retour ni `void`
- il n'a pas de paramètre
- il a un corps vide
- il est invoqué avec new

Java permet la définition de plusieurs constructeurs dans une même classe qui auront tous le même nom que le constructeur par défaut : on dit alors que le constructeur est *surchargé* (lorsqu'il y en a au moins deux).

# Constructeurs

ATTENTION : dès qu'un constructeur est explicitement définie dans une classe, le constructeur par défaut n'existe plus.
Donc si on veut avoir une version du constructeur par défaut parmi d'autres constructeur, il faut le mettre explicitement. ((voir la classe `Point`)

# Disparition d'un objet

Un objet disparaît lorsque plus aucune référence sur lui n'existe. L'espace qu'il occupe en mémoire peut alors être récupéré. Java possède un *ramasse-miettes* (*garbage collector*) qui se charge de cette récupération sans intervention nécessaire du programmeur.

# La classe `String`

La classe `String` est une classe prédéfinie de Java.
Elle comporte de nombreuses méthodes permettant la manipulation de chaînes de caractères.
C'est une classe *finale* c'est-à-dire que ses méthodes sont elles aussi finales et ne peuvent donc pas être surchargées (on ne peut pas utiliser des méthodes de même nom).
Les objets de la classe `String` sont *immuables*, c'est-à-dire qu'ils ne peuvent pas être modifiés, mais, à chaque « modification » est créé un nouvel objet.
Les méthodes de la classe `String` sont nombreuses (voir l'API).

# Quelques méthodes

- `int length()` $\rightsquigarrow$ renvoie la longueur de la chaîne objet (soit le nombre de caractères espace compris),
- `char charAt(int index)` $\rightsquigarrow$ renvoie le caractère en position index sachant que le premier caractère de la chaîne est en position 0,
- `String toLowerCase()` $\rightsquigarrow$ renvoie une chaîne composée des mêmes caractères que l'objet en minuscules,
- `String toUpperCase()` $\rightsquigarrow$ renvoie une chaîne composée des mêmes caractères que l'objet en majuscules.

`"abc".charAt(1)` retourne le caractère `b`
`"X2001".toLowerCase()` retourne la chaîne `x2001`.

# méthodes de comparaison

- `boolean equals(String s)` $\rightsquigarrow$ teste l'égalité de l'objet `String` à `s`
- `int compareTo(String s)` $\rightsquigarrow$ compare l'objet à `s` selon l'ordre alphabétique ; elle retourne un entier **négatif** si l'objet est **avant s**, **0** si l'objet est **égal à s** et un entier **positif** si l'objet est **après s**.
- `boolean startsWith(String prefix)` $\rightsquigarrow$ teste si l'objet commence avec la chaîne prefix
- `boolean endsWith(String suffix)` $\rightsquigarrow$ teste si l'objet termine avec la chaîne suffix
- `int indexOf(String facteur)` $\rightsquigarrow$ retourne -1 si facteur n'est pas un facteur de l'objet et retourne le plus petit indice du début de facteur dans l'objet sinon.
- `int lastIndexOf(String facteur)` $\rightsquigarrow$ retourne -1 si facteur n'est pas un facteur de l'objet et retourne le plus grand indice du début de facteur dans l'objet sinon.

# Exemples

`"chocolat".compareTo("chou à la crème")` renvoie un nombre négatif
`"A200002".indexOf("00")` renvoie 2
`"A200002".lastIndexOf("00")` renvoie 4.

# méthodes de « modification »

- `String substring(int debut, int fin)` $\rightsquigarrow$ renvoie la sous-chaîne de l'objet depuis l'indice debut jusqu'à l'indice fin-1.
- `String concat(String s)` $\rightsquigarrow$ crée une nouvelle chaîne composée de l'objet suivi de s.
- `String replace(char x, char y)` $\rightsquigarrow$ crée une nouvelle chaîne en remplaçant dans l'objet toutes les occurrences du caractère x par le caractère y.

`"chocolat".substring(2,5)` est la chaîne "oco".
`"j'aime ".concat("le chocolat")` est la chaîne "j'aime le chocolat".
`"chocolat".replace('o','u')` est la chaîne "chuculat".
**Remarque** : le terme modification n'est pas très appropriée puisqu'il y a en fait création d'une nouvelle chaîne.

# Remarque

Les arguments de la méthode `System.out.println` sont de type `String`.
Pourtant on a jusqu'à présent mélanger des variables de type primitif et des chaînes de caractères en les séparant par $+$ : en fait, avec cette méthode tout nombre est converti en une chaîne de caractère composée des chiffres qui le composent et de même tout booléen est transformé en la chaîne `true` ou `false` selon sa valeur.

# La classe `StringBuffer`

Si l'on veut modifier l'intérieur d'une chaîne sans en créer une nouvelle il faut utiliser la classe `StringBuffer` : `StringBuffer` est une classe du paquetage java.lang comme la classe `String`.
Elle possède aussi la méthode `char charAt(int index)` qui retourne le caractère d'indice index.
Pour modifier une chaîne on peut utiliser la méthode `void setCharAt(int index, char nouveau)` qui remplace le caractère d'indice index de l'objet par le caractère nouveau.

# lien avec String

Un constructeur de la classe `StringBuffer` accepte en paramètre un objet de la classe `String`.

```
String s="hello";
StringBuffer sbf = new StringBuffer(s);
```

Inversement la méthode d'instance `toString` invoqué par un objet de type `StringBuffer` renvoie un objet de type `String` référençant la même chaîne que l'objet de type `StringBuffer`.

```
StringBuffer sbf = new StringBuffer("Where is it?");
String s = sbf.toString();
```

# Classe interne

Une classe interne est une classe définie à l'intérieur d'une autre classe. L'accès à cette classe interne peut être `public`, `protected` ou `private`.
Si la classe interne est déclarée

- `static`, elle a accès à tous les attributs et méthodes `statiques` de sa classe englobante.
- sinon elle a accès à tous les attributs et méthodes de sa classe englobante.

Depuis l'extérieur on pourra faire référence à une méthode d'une classe interne (accessible !) en préfixant son identificateur de la façon suivante :

```
ClasseExterne.ClasseInterne.uneMéthodeInterne();
```

Lors de la compilation, le compilateur génère une fichier `ClasseExterne$ClasseInterne.class`