ECOLE MAROCAINE DES SCIENCES DE L'INGENIEUR

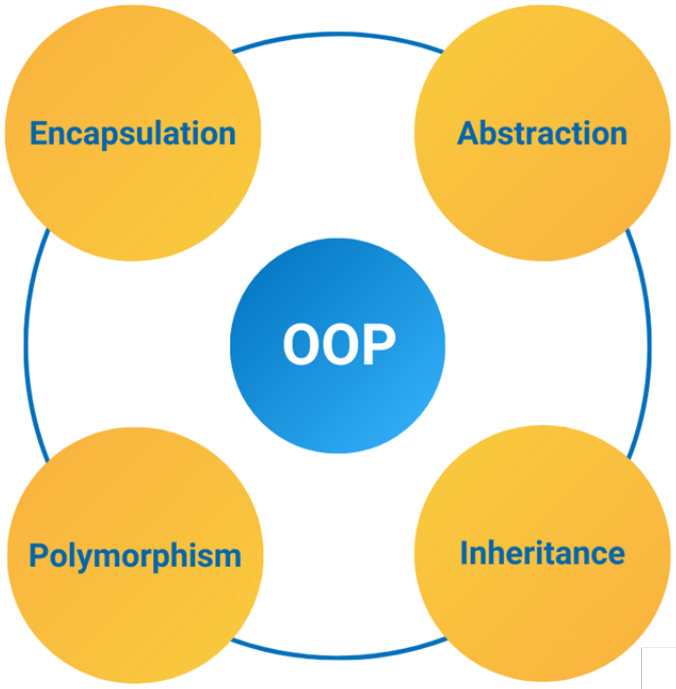
Membre de HONORIS UNITED UNIVERSITIES

CLASSES ET OBJETS

3IIR - POO2

POO

- **Concepts clés => classes et objets.**
  - ***Le code du programme est structuré autour d'objets qui sont des instances de classes.***
- **Principes =>** Abstraction, Encapsulation, Héritage, polymorphisme
  - **Favorise la réutilisabilité, la modularité et la maintenance du code.**



ABSTRACTION

- Permet de représenter de **manière simplifiée et pertinente,** des objets du monde réel, selon le contexte du problème et de la solution proposée.
- Consiste à **cacher les détails internes du fonctionnement** d'un objet et à **n'exposer que les fonctionnalités essentielles.**

✓ **Avantages** :

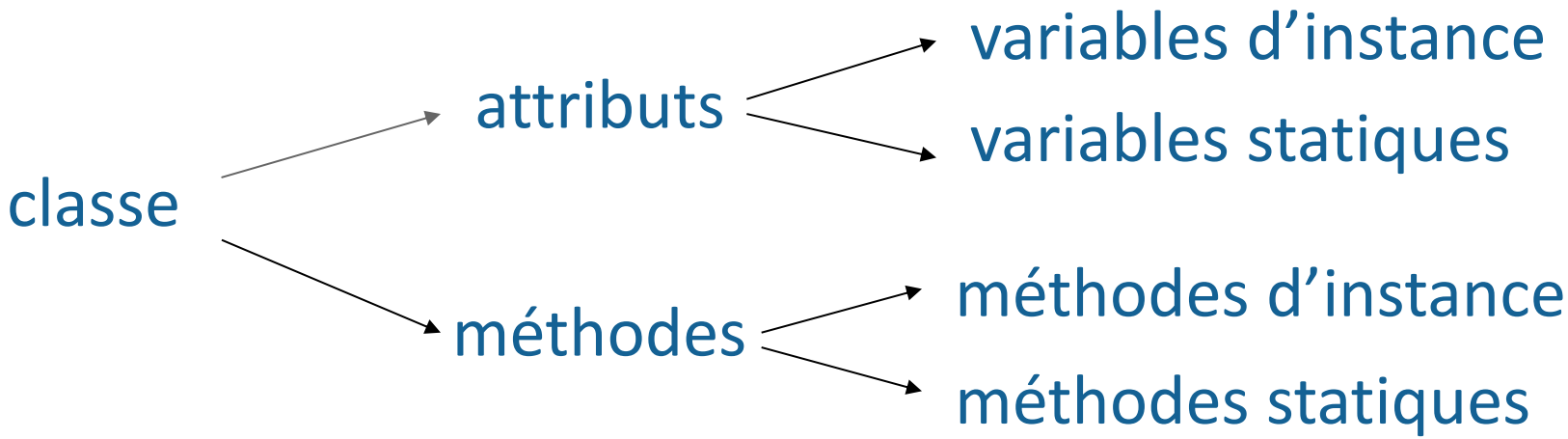
- **Gestion de la complexité** : les utilisateurs se concentrent sur **ce que fait un objet** plutôt que sur comment il le fait.
- **Contrat pour les développeurs** : permet de fournir une interface commune, garantissant uniformité et évolution facile du code.

CLASSES

- La classe est **l'unité de base de l'abstraction**, elle permet de définir un **nouveau type.**
- Une classe contient :
  - Des **attributs** qui correspondent aux caractéristiques qu'auront en commun tous les objets de la classe.
  - Des **méthodes**, qui correspondent aux opérations réalisables sur les objets de la classe (comportements).

CLASSES

- Tout code Java est défini dans des **classes** ou des interfaces ou des énumérations.



CLASSES

- Il est recommandé de mettre chaque définition de classe dans un fichier (.java) séparé.

Point.java

```
package exemples.formes;

class Point {

    float x; // variable d'instance
    float y;

    void afficher()
    {
        System.out.println ("x = " + x + " y = " + y) ;
    }
}
```

Attributs

Méthode

Le nom d'une classe commence par une majuscule

Si le fichier contient une classe "public", il doit porter le nom de cette classe.

La classe **Point** se trouve dans le fichier **Point.java** qui se trouve dans le dossier **exemples/ formes.**

CLASSES

- **Toute classe doit faire partie d'un package.**
  - Les packages sont comme des dossiers où les fichiers, contenant le code source des classes, sont sauvegardés.
  - Les packages permettent d'organiser les classes tout en contrôlant leur visibilité et celle de leurs membres.

```
package <nom du package>;

class <NomDeLaClasse> {

}
```

- Si vous ne spécifiez pas de package, la classe sera placée dans le package par défaut. ***Cette pratique est déconseillée.***

CLASSES

- Les attributs sont initialisés par défaut :
  - **Types primitifs**

    `boolean` : **`false`**

    `char` : `'\u0000'` (caractère nul)

    `byte`, **`short`**, **`int`**, **`long`** : 0

    `float`, **`double`** : 0.0
  - **Type référence** : null

- Par défaut (aucun modificateur d'accès spécifié)
  - **Les attributs et les méthodes d'une classe, A, sont visibles pour toutes les classes du même package que A .**

OBJETS

Un objet est une instance spécifique de la classe en **mémoire**.

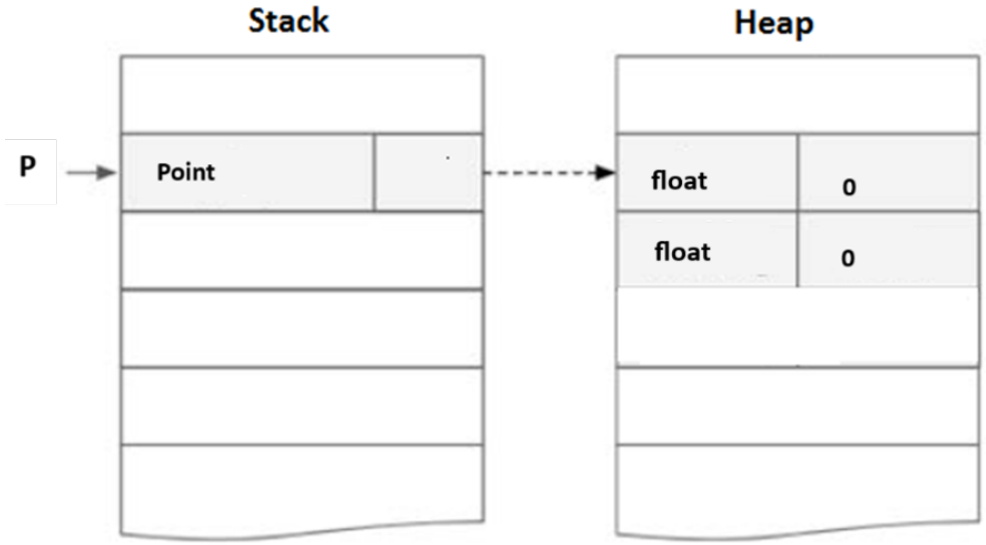
- Pour utiliser une classe, il faut créer une instance de cette classe.
  - **Déclaration** : `Point P;` **/* déclare une référence à un objet Point */**
  - **Instanciation** : `P = new Point();` **/*alloue la mémoire pour un objet Point */**
    - → ***L'instanciation avec new ci-dessus, alloue la mémoire pour un objet de type Point, appelle le constructeur par défaut puis retourne une référence à l'objet créé. Cette référence est stockée dans la variable P.***
- ***On dit "l'objet P", mais il faut comprendre que P n'est qu'une référence à l'objet Point crée.***

OBJETS

- **En java, une variable d'un type classe est de type référence .**
  - ***Une référence est l'adresse de l'emplacement mémoire*** ou se trouvent les données de l'objet.
  - Une référence permet d'accéder à un objet sans avoir besoin de connaître son adresse mémoire.

```
Point P = new Point();
```

Après l'allocation de la mémoire (Heap), une référence (ie. adresse) à l'emplacement mémoire alloué est stockée dans la variable P.



En Java, vous ne pouvez pas manipuler directement les adresses mémoire comme en C ou C++

ENCAPSULATION

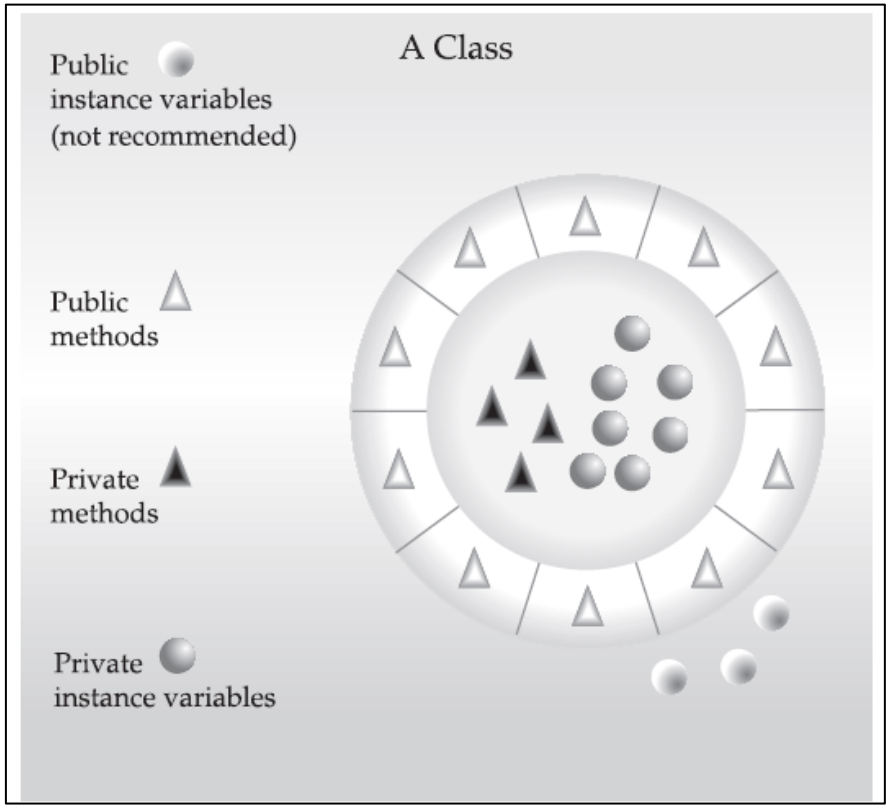
- **Objectifs**:
  - Restreindre l'accès aux données de l'objet.
  - Protéger l'état interne de l'objet et contrôler comment ses données sont modifiées.
- Pour mettre en œuvre ce concept :
  - Les **attributs** de la classe doivent être **privés**.
  - On n'accède aux attributs privés que par **des méthodes publiques** prévues à cet effet.

```
package exemples.formes;

class Point {

 private float x;

 private float y;

public void afficher()
{
  System.out.println ("x = " + x + " y = " + y) ;
}

}
```

MODIFICATEURS D'ACCÈS

Les modificateurs d'accès décrivent la visibilité des classes, des attributs et des méthodes.

- **Classe :**
  - **par défaut :** visible dans son propre package;
  - **public** : visible partout;
- **Attribut ou méthode d'une classe A**
  - **private** : visible uniquement dans la classe A;
  - **par défaut :** visible uniquement pour les classes du même package que A.
  - **public** : visible partout si A est public;
  - **protected** : visible pour les classes du même package que A et aussi pour les sous-classes de A, même dans d'autres packages.



MODIFICATEURS D'ACCÈS

Point.java

```
package exemples.formes;

class Point {

 private float x;

 float y;

 public char nom;


}
```

Main.java

```
package exemples.formes;

public class Main {

public static void main (String[] args){

     Point P1 = new Point();

     P1.x = 5;
     P1.y = 6;
     P1.nom ='A';

     }
}
```

Ce code ne compile pas, trouvez l'erreur !

MODIFICATEURS D'ACCÈS

formes/Point.java

```
package exemples.formes;

class Point {

 private float x;

 private float y;

 public Point() { x = 1; y = 1;}

 …

}
```

test/Main.java

```
package exemples.test;

import exemples.formes.Point; // pour utiliser Point

class Main {

public static void main (String[] args){

     Point P1 = new Point();

     }
}
```

Ce code ne compile pas, trouvez l'erreur !

MODIFICATEURS D'ACCÈS

test/Main.java

```
package exemples.test;

import exemples.formes.Point;

class Main {

public static void main (String[] args){

     Point P1 = new Point();

     P1.x = 5;
     P1.y = 6;
     P1.nom ='A';

     }
}
```

```
package exemples.formes;
public class Point {

 private float x;

 float y;

 public char nom;


}
```

formes/Point.java

Ce code ne compile pas, trouvez l'erreur !

CONSTRUCTEUR PAR DÉFAUT

- Un constructeur est une méthode spéciale de la classe qui est appelée automatiquement, pour initialiser les attributs de l'objet lors de sa création.
- **Si aucun constructeur n'est défini dans la classe, Java fournit un constructeur par défaut qui ne fait rien.**

```java
package exemples.formes;

class Point {

 private float x=1;

 private float y;

public void afficher()
 {
  System.out.println ("x = " + x + " y = " + y) ;
 }
}
```

Point.java

```java
package exemples.formes;

class Main {

public static void main (String[] args){

     Point P = new Point();

     P.afficher() ;

     }
}
```

Main.java

Donnez la sortie du programme !

CONSTRUCTEUR SANS PARAMÈTRES

- Vous pouvez définir explicitement un constructeur sans paramètres pour initialiser les attributs d'un objet avec des valeurs par défaut.

- ✓ **Un constructeur doit avoir le même nom que la classe**
- ✓ **Un constructeur ne doit pas avoir de type de retour même pas void.**

Point.java

```
package exemples.formes;

class Point {

 private float x;

 private float y;

public Point() { x = 1; y = 1;}

public void afficher()
{
  System.out.println ("x = " + x + " y = " + y) ;
}

}
```

- La définition explicite d'un constructeur dans une classe **désactive** la génération du constructeur par défaut implicite.

CONSTRUCTEUR AVEC PARAMÈTRES

- Le constructeur avec paramètres permet à l'utilisateur de fournir des valeurs spécifiques pour initialiser les attributs de l'objet au moment de sa création.

```java
package exemples.formes;

class Point {

 private float x;

 private float y;

public Point() { x = 1; y = 1;}

public Point(float a, float b) { x = a; y = b;}

public void afficher()
 {
  System.out.println ("x = " + x + " y = " + y) ;
 }

}
```

Point.java

```java
package exemples.formes;

class Main {

public static void main (String args[]){

     Point P1 = new Point();

     P1.afficher() ;

     Point P2 = new Point(1,2);

     P2.afficher() ;

     }
}
```

Main.java

BLOC D'INITIALISATION D'INSTANCE

- Un bloc d'initialisation d'instance est un bloc de code situé en dehors des méthodes.
- Un bloc d'initialisation d'instance s'exécute chaque fois qu'une instance de la classe est créée, juste avant l'appel du constructeur.

```
package exemples;
class Exemple
{
    {
    System.out.println("Bloc d'initialisation ");
    }
    public static void main(String[] args)
    {
    System.out.println("Dans main ");
    }
}
```

Donnez la sortie de ce programme ?

BLOC D'INITIALISATION D'INSTANCE

→ Un bloc d'initialisation d'instance sert à initialiser les attributs d'instance avant l'appel du constructeur.

```
package exemples;
class Exemple {

int num;

Exemple()
{
      System.out.println(" Constructeur sans paramètres, num = " + num);
}
Exemple(int x)
{
      num += x;
      System.out.println(" Constructeur avec paramètres, num = " + num);
}
{
      num = 10;
      System.out.println(" bloc d'initialisation, num =" + num);
}

public static void main(String[] args) {
      new Exemple();
      new Exemple(5);
 }
}
```

Donnez la sortie de ce programme ?

BLOC D'INITIALISATION D'INSTANCE

```
package exemples;
class Exemple {

Exemple(int x) {
  num += x;
  System.out.println(" Constructeur avec paramètres, num = " + num);
}

public static void main(String[] args) {
          new Exemple(5);
}
private int num;

{
 num = 10;
 System.out.println(" bloc d'initialisation num = " + num ");
}

}
```

Donnez la sortie du programme ?

BLOC D'INITIALISATION D'INSTANCE

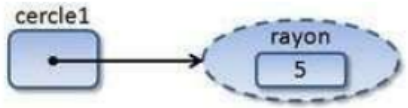
```
package exemples;
class Exemple {

private int num;

Exemple() {
   System.out.println("num = " + num);
}

Exemple(int x) {
  num += x;
  System.out.println("num = " + num);
}

{
  int temp = (int) (Math.random() * 100);
  num = temp % 10 == 0 ? temp + 1 : temp; // éviter les multiples de 10
  System.out.println(" Bloc d'initialisation ");
}

public static void main(String[] args) {
  new Exemple();
  new Exemple(5);
}
}
```

Un bloc d'initialisation d'instance est utile :

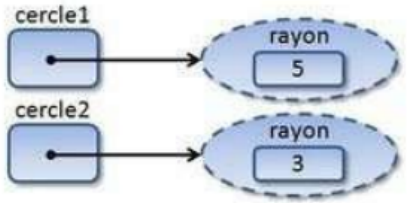
- **Lorsque la logique d'initialisation est plus complexe qu'une simple affectation.**
- **Pour éviter de répéter le code d'initialisation si plusieurs constructeurs existent.**

OPÉRATIONS SUR LES OBJETS

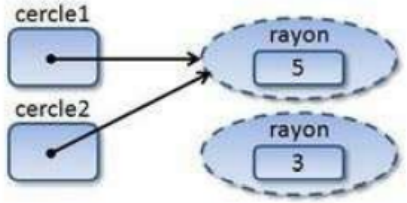
```
Cercle cercle1;
cercle1 = new Cercle(5);
```



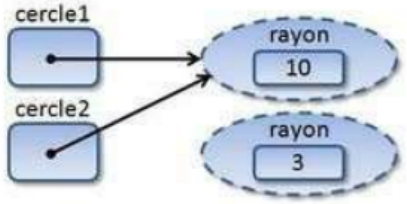

```
Cercle cercle2 = new Cercle(3);
```




```
cercle2 = cercle1;
```




```
cercle1.Rayon = 10;
```



Affectation

- L'affectation cercle2 = cercle1 provoque la copie de la référence cercle1 dans cercle2 .
- Après affectation, les deux variables font référence au même objet.

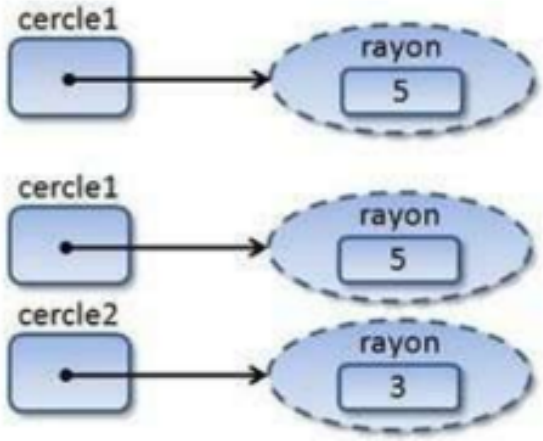
Pour créer une copie d'un objet; redéfinissez la méthode clone()

OPÉRATIONS SUR LES OBJETS

Comparaison

```
Cercle cercle1;
cercle1 = new Cercle(5);
```

```
Cercle cercle2 = new Cercle(3);
```



- L'expression **cercle1 == cercle2** compare les références et non le contenu des objets eux-mêmes. Elle renvoie toujours false.

Pour comparer les objets et non leurs références; redéfinissez la méthode equals.

DESTRUCTION DES OBJETS

- En java, il n'est pas nécessaire de libérer manuellement l'espace mémoire alloué avec new car Java dispose d'une fonctionnalité de libération mémoire automatique appelée ramasse-miettes (**Garbage collector**).
  - En C++, la libération de la mémoire dynamique est de la responsabilité du programmeur, ce qui est source de problèmes complexes.
- Le système de gestion automatique de la mémoire (Garbage collector), s'occupe de libérer les objets qui ne sont plus utilisés.
  - Les objets qui n'ont plus de références pointant vers eux à un moment donné sont instinctivement libérés, et la mémoire qu'ils occupent est libérée.

MÉTHODES

- Il est recommandé de nommer les méthodes avec des **verbes** et de suivre la convention **camelCase**.

Les **variables locales** d'une méthode doivent être explicitement initialisées avant d'être utilisées.

Exemple.java

```
package exemples;
class Exemple {

public void afficherMessage()
{
  String message;
  System.out.println(message);
}

public static void main(String[] args) {

  Exemple ex = new Exemple();
  ex.afficherMessage();

}

}
```

Ce code ne compile pas, trouvez l'erreur !

SURCHARGE DE MÉTHODES

- La surcharge de méthodes nous permet de déclarer deux méthodes ou plus dans la même classe avec le même nom mais avec soit **un nombre différent de paramètres**, soit **des paramètres de types différents.**
- ***Il est impossible de surcharger une méthode en modifiant uniquement son type de retour.***

PASSAGE DES ARGUMENTS

- **Un argument de type primitif** (int, char, boolean, etc.) est **passé par valeur** :
  - La valeur de l'argument est copiée dans le paramètre . Ainsi, les modifications du paramètre dans la méthode n'ont pas d'impact sur l'argument.
- Un **argument de type référence** (objet, tableau, chaîne de caractères) **transmet une référence :**
  - **La référence est copiée dans le paramètre => Paramètre et argument concernent la même zone mémoire.** Ainsi, les modifications du paramètre dans la méthode modifient aussi l'argument.

PASSAGE DES ARGUMENTS

- Un paramètre **varargs**, permet de passer **un nombre variable d'arguments** de même type.

- Utilisez les trois points … pour déclarer un paramètre **varargs.**
- Le paramètre **vararg** est traité comme un tableau à l'intérieur de la méthode.
- Une méthode ne peut avoir **qu'un seul paramètre varargs.**
- Le paramètre **varargs** doit être le dernier de la liste des paramètres de la méthode.

Paramètre varargs

```
package exemples;

class Exemple {

public static void afficherNombres(int ... nombres)
{
     for (int n : nombres)
     {
       System.out.print(n + " ");
     }
}

public static void main(String[] args)
{
     Exemple.afficherNombres(4,5,6,7);
}

}
```

LA RÉFÉRENCE THIS

- Le mot-clé **this** fait référence à **l'objet courant** => l'objet appelant la méthode.
  - **Le compilateur transmet implicitement la référence de l'objet courant à la méthode. C*ette référence est stockée dans* this*.***
- Le mot-clé **this** a deux objectifs importants :
  - **Accéder à l'objet courant à l'intérieur de la méthode.**
  - **Faire la distinction entre les variables d'instances et les paramètres ayant le même nom.**

```
public Point(float x, float y)
{
 this.x = x;
 this.y = y;
}
```

LE MOT-CLÉ THIS

- En java, le mot-clé **this** permet également d'appeler un autre constructeur de la même classe.

```
package exemples.formes;
public class Rectangle {

private float l;
private float h;

public Rectangle(){
  this(1,1); // cette instruction doit être la première
}

public Rectangle(float l, float h)
{
  this.l = l;
  this.h = h;
}

public Rectangle(float a) //pour construire un carré
{
  this(a,a); // Appel du constructeur avec paramètres
}
}
```

Rectangle.java

ACCESSEURS

- **Un accesseur est une méthode publique qui permet d'accéder à un attribut privé.**
  - Un accesseur en lecture (**getter**) permet de lire la valeur d'un attribut privé.
  - Un accesseur en écriture (**setter**) permet de modifier la valeur d'un attribut privé.
- En Java, les accesseurs sont des méthodes de la forme **get**NomAttribut et **set**NomAttribut.

```java
package exemples.formes;

public class Cercle {

private int rayon;

public Cercle(int r)
{
     rayon = r > 0 ? r : 0;
}

public int getRayon()
{
     return rayon;
}
public void setRayon(int r)
{
      rayon = r > 0 ? r : 0;
}

}
```

Cercle.java

ATTRIBUT FINAL

- Un **attribut constant** est déclaré avec le mot clé **final.**

Un **attribut final** doit être initialisé **soit lors de sa déclaration**, **soit par tous les constructeurs de la classe**.

- **Rappel** => le mot clé final peur aussi être attribué aux :
  - Paramètres des méthodes.
  - Variables locales des méthodes.

```
package exemples.formes;
public class Point {

private float x,y;

private final char nom;

public Point(float x, float y, char c)
{
     this.x = x;
     this.y = y;
     this.nom = c;
}
public Point(){
     this.x = 0;
     this.y = 0;
}

}
```

Ce code ne compile pas, trouvez l'erreur !

ATTRIBUT STATIQUE

- Un **attribut statique** est une variable partagée par tous les objets de la classe.
  - Utilisé pour stocker une valeur **commune à tous les objets**
  - Déclaré avec le mot clé **`static`**
- À **un attribut statique** correspond **une unique zone mémoire qui est allouée au démarrage du programme.**
  - L'attribut statique est donc lié à la classe elle même, et non à un objet spécifique de la classe.

```java
package exemples.formes;
public class Point {

private float x,y;

private static int nb; //initialisé à 0

public Point(float x, float y)
{
     this.x = x; this.y = y;
     nb++;
}
public Point(){
     this.x = 0; this.y = 0;
     nb++;
}

}
```

Point.java

ATTRIBUT FINAL STATIC

- Une constante partagée est déclarée en tant que **`static final.`**

```
package exemples.formes;
public class Cercle {

private int rayon;

static final double PI = Math.PI;

public Cercle(int r)
{
     rayon = r > 0 ? r : 0;
}

public double perimetre()
{
     return 2*PI*rayon;
}

}
```

Cercle.java

MÉTHODE STATIQUE

- Une méthode statique est déclarée avec le mot clé **static**
- Une méthode statique peut être appelée sans créer d'instance de la classe. Pour cela, on préfixe le nom de la méthode statique avec le nom de la classe => **NomClasse.nomMéthodeStatique**
- **Il est préférable** de définir une méthode statique pour accéder en ***lecture/écriture*** à un attribut **static** privé.

```java
package exemples.formes;
public class Point {

private float x,y;

private static int nb; //compteur d'instances

public static int getNb()
{
  return nb;
}

}
```

Point.java

```java
package exemples.formes;

class Main {

public static void main (String args[]){

     Point P1 = new Point();

     Point P2 = new Point(1,2);

     System.out.println("Nombre = " + Point.getNb());
     }
}
```

Main.java

MÉTHODE STATIQUE

- Une méthode statique est aussi utilisée pour définir un comportement indépendant de tout objet de la classe.

```
package jeu;
public class Joueur
{
     String nom;

public Joueur(String n)
{
     nom = n;
}

public Joueur() { this("");}

}
```

Joueur.java

```
package jeu;
public class OutilsJeu
{
static java.util.Random unDé = new java.util.Random();

static int jet()
{
   return Math.abs(unDé.nextInt()) % 6 + 1;
}

}
```

OutilsJeu.java

MÉTHODE STATIQUE

```
package jeu;

import java.util.Scanner;

public class PartieJeu
{

public static void main(String[] arg) {

Joueur joueur = new Joueur();

Scanner sc= new Scanner(System.in);

Joueur.nom= sc.nextLine();

System.out.println(joueur.nom + " joue " + OutilsJeu.jet());

sc.close();
}
}
```

PartieJeu.java

MÉTHODE STATIQUE

Une méthode statique ne peut pas :

- Appeler directement une méthode non statique de la classe.
- Accéder directement à un attribut non statique de la classe.
- Faire référence à **this** (ou **super**).

Ce code ne compile pas, trouvez l'erreur !

```
package exemples;
public class A
{
     static private final int p = 20 ;

     private int m ;

     void g(int n)
     {
        m = n ;
     }

     static int f(int n)
     {
       m = n;
     }
     static int f1(A oA)
     {
       oA.m = p;
       g(p);
       oA.g(p);
     }

}
```

BLOC STATIC

- Si vous devez effectuer des calculs pour initialiser vos variables statiques, vous pouvez déclarer un bloc statique qui sera exécuté exactement une fois, lorsque la classe est chargée pour la première fois.

```
public class EssaiStatic
{

static int a =3;
static int b;

static void meth(int x )
{
System.out.println(" x = " + x);
System.out.println(" a = " + a);
System.out.println(" b = " + b);
}
static
{
   System.out.println(" static block" );
   b = a * 4;

}
public static void main(String[] arg) {
  meth(42);
}
}
```