POO en JAVA

Classes et Objets

Classes et Objets en Java

Les Bases des Classes en Java

Introduction 1/2

- Java est un langage OO pur et donc les classes constituent la structure sous-jacente de tous les programmes Java .

- Tout ce que nous voulons représenter en Java doit être encapsulé dans une classe qui définit l'*«état»* et le *«comportement»* des composants de base du programme connus sous le nom d'*objets*.

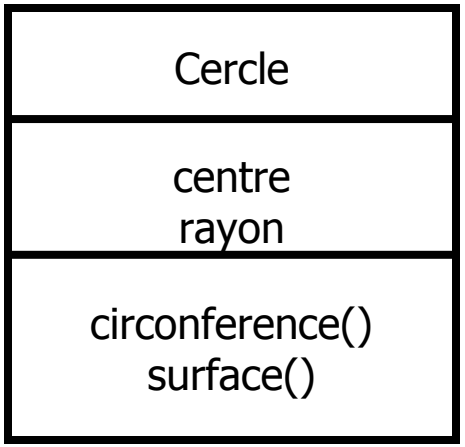
Introduction 2/2

- Les Classes créent des objets et les objets utilisent les Méthodes pour communiquer entre eux. Ils offrent une façon pratique pour le regroupement ou le rassemblement (packaging) d'un groupe d'éléments et fonctions qui travaillent sur des données liées logiquement.

- Une classe sert essentiellement comme modèle pour un objet et se comporte comme un type de données de base "int". Il est donc important de comprendre comment les champs et les Méthodes sont définies dans une classe et comment ils sont utilisés pour construire des programmes Java qui intègrent les concepts OO de base tels que l'encapsulation, l'héritage et le polymorphisme.

Classes

- Une *classe* est une collection de *d'attributs* (données) et *méthodes* (procedure ou fonction) qui opèrent sur ces données.



Les Classes

- Une *classe* est une collection *d'attributs* (données) et *méthodes* (procédure ou fonction) qui opèrent sur ces données.
- La syntaxe de base pour une définition de classe :

```
class NomClasse
{
        [déclaration d'attributs ]
        [déclaration de méthodes]
}
```

- Déclaration de la Classe Cercle – pas d'attributs, pas de méthodes

```
public class Cercle {
    // ma classe cercle
}
```

Les Classes

- Une *classe* est une collection de *d'attributs* (données) et *méthodes* (procédure ou fonction) qui opèrent sur ces données.
- La syntaxe de base pour une définition de classe :
- Déclaration de la Classe Cercle – pas d'attributs, pas de méthodes

```
public class Cercle {
        // ma classe cercle
}
```

La déclaration d'une classe se fait dans un fichier avec l'extension ".java" qui porte le même nom que la classe. En l'occurrence le code (méthodes et attributs) de la classe *Cercle* se trouve dans le fichier *Cercle.Java.* Un fichier ".java" ne peut contenir qu'une seule classe *public*.

Ajout d'attributs: La Classe Cercle avec attributs

- Ajouter les *attributs*

```
public class Cercle {
        public double x, y;  // coordonées du centre
        public double r;     //  rayon du cercle


}
```

- Les *attributs* (données) sont aussi appellés des varaibles *d'instance* .

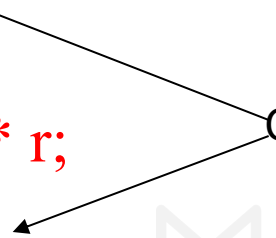
Ajout de Méthodes

- Une classe avec seulement des champs (attributs) de données n'a aucune vie. Les Objets crées par une telle classe ne peuvent répondre à aucun message.
- Les Méthodes sont déclarés dans le corps de la classe mais immédiatement après la déclaration des champs (attributs) de données.
- La forme générale d'une déclaration de méthode est:

```
type NomMéthode (liste-paramètres)
{
        corps-Méthode;
}
```

Ajout de Méthodes à la Classe Cercle

```
public class Cercle {

    public double x, y; // centre du cercle
    public double r;    // rayon du cercle

    //Méthodes pour retourner la circonférence et la  surface
    public double circonference() {
                return 2*3.14*r;
    }
    public double surface() {
                return 3.14 * r * r;
    }
}
```



Abstraction de Données

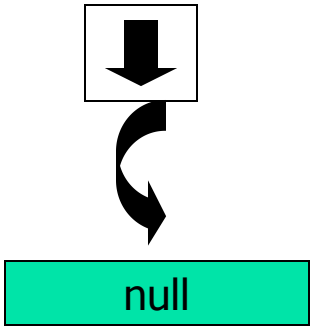
- Déclaration de la classe Cercle, a crée un nouveau type de données – Abstraction de Données
- On peut définir des variables ***(objets/instances)*** de ce type:

```
Cercle  unCercle;
Cercle  bCercle;
```

Classe Cercle (suite)

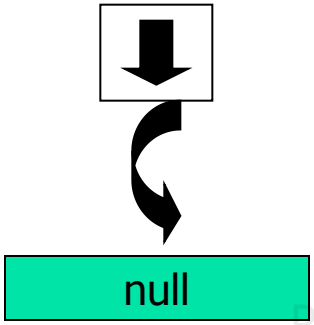
- unCercle, bCercle font simplement référence à un objet Cercle , mais ne sont pas des objets eux-mêmes.

unCercle



Pointe sur rien (référence Null)

bCercle

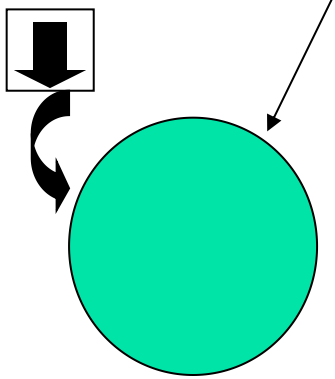


Pointe sur rien (référence Null)

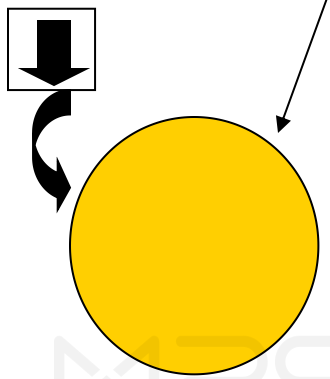
Création des objets (instance) d'une classe

- Les objets sont crées dynamicament utilisant le mot clé *new*.
- unCercle et bCercle font référence à des objets Cercle

unCercle = new Cercle() ;



bCercle = new Cercle() ;



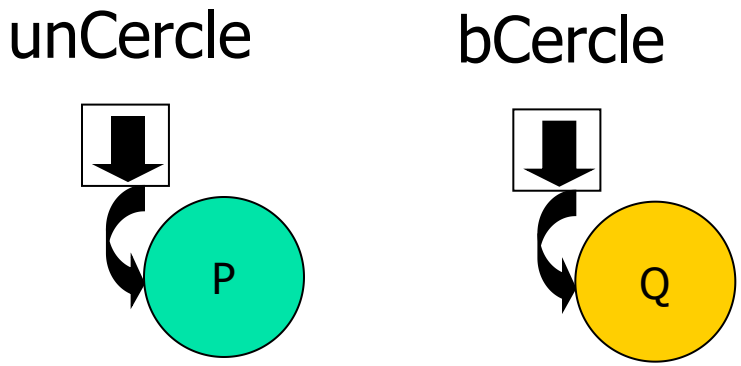
Création des objets d'une classe

```
unCercle = new Cercle();
bCercle = new Cercle() ;

bCercle = unCercle;
```

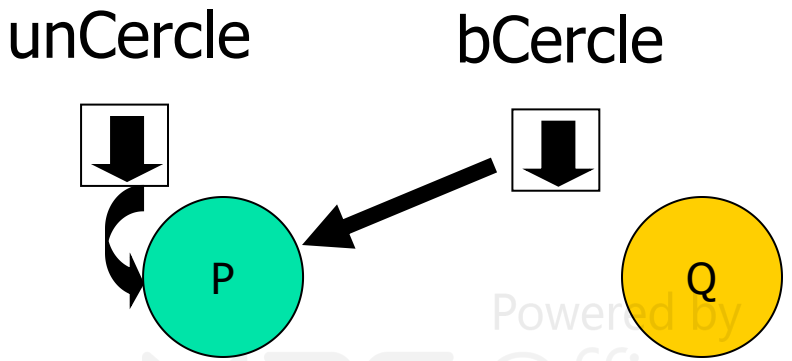
Creation des objets d'une classe

```
unCercle = new Cercle();
bCercle = new Cercle() ;

bCercle = unCercle;
```

Avant instruction

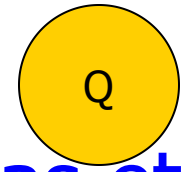


Après instruction



Garbage collection (Automatique)

Ramasse miettes

- L'objet 

n'a pas une référence et ne peux pas être utilisé dans le futur.

- L'objet devient un candidat pour la "garbage collection" automatique .

- Java collecte automatiquement le garbage *(miettes)* periodiquement et libère la mémoire utilisée pour être utilisée dans le futur.

Accés aux Données Objet/Cercle

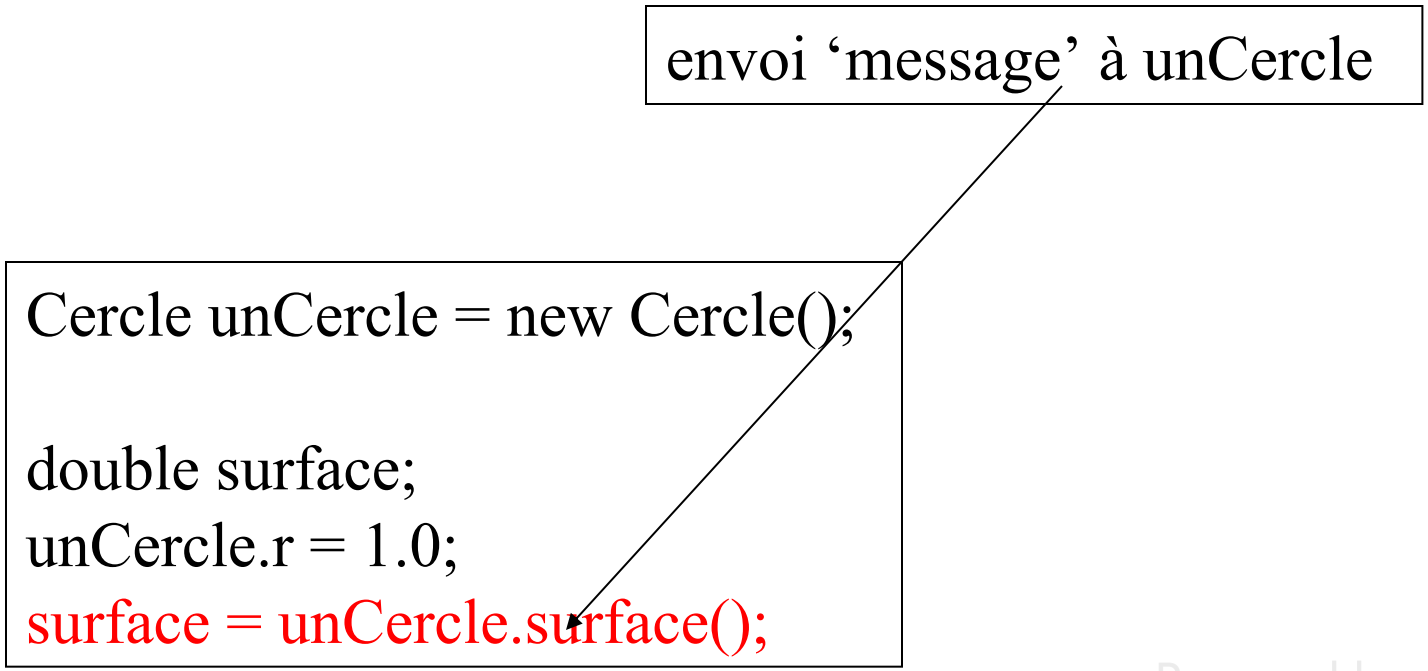
- Similaire à la syntaxe C pour l'accés aux données définies dans une structure.

NomObjet.NomVariable
NomObjet.NomMéthode(liste-paramètres)

```
Cercle unCercle = new Cercle();

unCercle.x = 2.0 // initialise centre et rayon
unCercle.y = 2.0
unCercle.r = 1.0
```

Exécution de Méthodes dans un Object/Cercle

- Utilisation des méthodes d'objets :



Utilisation de la Classe Cercle

```
class monMain
{
	public static void main(String args[])
	{
		Cercle unCercle;  // création de référence
		unCercle = new Cercle(); // création d'objet
		unCercle.x = 10;  // affectation de valeurs aux champs de données
    (attributs)
		unCercle.y = 20;
		unCercle.r = 5;
		double surface = unCercle.surface(); // invocation de méthode
		double circonf = unCercle.circonference();
		System.out.println("rayon="+unCercle.r+" surface="+surface);
		System.out.println("rayon="+unCercle.r+" circonference ="+circonf);
	}
}
```

```
rayon=5.0 surface=78.5
rayon=5.0 circonference =31.400000000000002
```

Classes et Objets en Java

Constructeurs, Surcharge, Membres Statiques

Accés/Modification des données membres

```
class monMain
{
	public static void main(String args[])
	{
		Cercle unCercle;  //
		unCercle = new Cercle(); //
		unCercle.x = 10;  // affectation de valeurs aux champs de données
    (attributs)
		unCercle.y = 20;
		unCercle.r = 5;
		double surface = unCercle.surface(); //
		double circumf = unCercle.circonference();
		System.out.println("rayon="+unCercle.r+" surface="+surface);
		System.out.println("rayon="+unCercle.r+" circonference ="+circonf);
	}
}
```

```
rayon=5.0 surface=78.5
rayon=5.0 circonference =31.400000000000002
```

Meilleure façon d'Initialisation ou d'Accés aux Données Membres x, y, r

- Lorsqu'il existe beaucoup de *données* à MàJ/accéder et aussi pour développer un code lisible, ce est genéralement fait par la définition de méthodes spécifiques pour chaque cas.
- Pour initialiser/MàJ une valeur:
  - unCercle.setX(10)
- pour accéder/lire une valeur:
  - unCercle.getX()
- Ces méthodes sont informellement appellées Accesseurs/Mutateurs ou les Méthodes Setters/Getters . liées au principe de l'encapsulation.

Accesseurs/Mutateurs "Getters/Setters"

```
public class Cercle {
    public double x,y,r;

    //Méthodes pour retourner la circonference et la surface
    public double getX() { return x;}
    public double getY() { return y;}
    public double getR() { return r;}
    public double setX(double x_in) { x = x_in;}
    public double serY(double y_in) { y = y_in;}
    public double setR(double r_in) { r = r_in;}

}
```

Accesseurs/Mutateurs "Getters/Setters"

```
class monMain
{
	public static void main(String args[])
	{
		Cercle unCercle;  // creating reference
		unCercle = new Cercle(); // creating object
		unCercle.setX(10);
		unCercle.setY(20);
		unCercle.setR(5);
		double surface = unCercle.surface(); // invoking méthode
		double circumf = unCercle.circonference();
		System.out.println("rayon="+unCercle.getR()+" surface="+surface);
		System.out.println("rayon="+unCercle.getR()+" circonference ="+circumf);
	}
}
```

```
rayon=5.0 surface=78.5
rayon=5.0 circonference =31.400000000000002
```

Initialisation des Objets

- Lorsque les objets sont crées, les champs de données (attributs) sont initialisés par défaut *(numérique(0), boolean(false),caractère('\u0000', référence(null))*, à moins que ses utilisateurs le font explicitement par d'autres valeurs. Par exmple,
  - NomObjet.attribut1 = val; // Ou
  - NomObjet.Setattribut1(val);// val n'est une valeur par défaut du type de attribut1
- Dans beaucoup de cas, ça a du sens si cette initialisation peut être effectuée par défaut sans que les utilisateurs les initialisent explicitement.
  - Par exemple, si on crée un objet de la classe appellée "Alphabet", il est naturel de présumer que l'attribut *lettre* est initialisé à *'a' (au lieu du caractère '\u0000' qui n'a aucune signification pour l'aphabet du langage naturel.*) sauf autre indication d'être spécifié différement.

```
class Alphabet
{
        char lettre;
        …
}
 Alphabet  alphabet1 = new  Alphabet();
```

  - Quelle est la valeur de " Alphabet.lettre" ?
- En Java, cela peut être réalisé par un mécanisme appelé ***constructeurs***.

Qu'est ce qu'un Constructeur?

- Un Constructeur est une méthode spéciale qui est invoquée “automatiquement” au moment de la création de l'objet .
- Un Constructeur est normalement utilisé pour l'initialisation des objets avec des valeurs par défaut à moins que d'autres valeurs différentes sont fournies.
- Un Constructeur a le même nom que le nom de la classe.
- Un Constructeur ne peut pas retourner de valeurs (il ne s'agit pas de void).
- Une classe peut avoir plus d'un constructeur tant qu'ils ont des signatures différentes (i.e., syntaxe des arguments d'entrées différente).

Définition d'un Constructeur

- Comme toute autre méthode

```
public class NomClasse{

    // Attributs (champs de données)…

    // Constructeur
    public NomClasse()
    {
        // Déclarations de corps Méthodes initialisant les
        //champs de données    }

    //Méthodes pour manipuler les champs de données
}
```

- Invocation:
  - Il n'y a AUCUNE instruction d'appel explicite nécessaire: Lorsque l'instruction de création d'objet est exécutée, la méthode du constructeur sera exécuté automatiquement.

Définition d'un Constructeur: Exemple

```
public class Alphabet {
    char lettre;

    // Constructeur
    public Alphabet()
    {
        lettre = 'a';
    }
    //Méthodes de mise à jour ou d'accès à lettre
    public void suivant()
    {
        lettre = (char)((int)lettre + 1);
    }
    public void precedent()
    {
        lettre = (char)((int)lettre - 1);;
    }
    char getLettre()
    {
      return lettre;
    }
}
```

Tracer la valeur de *lettre* à chaque instruction. Quelle est la valeur de sortie ?

```
class maClasse {
  public static void main(String args[])
  {
        Alphabet alphabet1 = new Alphabet();
        alphabet1.suivant();
        char x = alphabet1.getLettre(); // x=='b'
        alphabet1.suivant();
        char y =  alphabet1.getLettre(); // y=='c'

        System.out.println( x +" "+ alphabet1.getLettre());//
  }
}
```

Un *Alphabet* avec Valeur Initiale fournie par l'utilisateur?

- Cela peut être fait par l'ajout d'une autre méthode constructeur à la classe.

```
public class Alphabet {
    char lettre;

    // Constructeur 1
    public Alphabet()
    {
        lettre = 'a';
    }
    public Alphabet(char lettreInit )
     {
         lettre = lettreInit;
     }
}

// Un nouvel Utilisateur de la Classe: Utilisant les deux
// constructeurs
    Alphabet alphabet1 = new Alphabet();
    Alphabet alphabet2 = new Alphabet ('t');
```

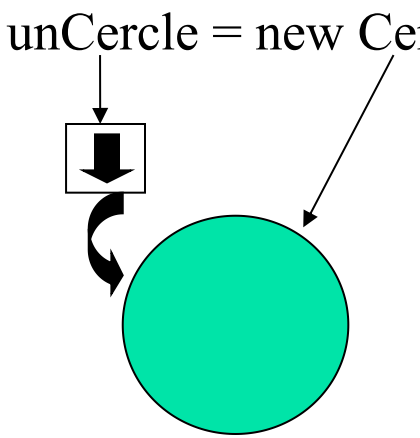
Ajout d'un Constructeur à plusieurs Paramètres à la Classe Cercle

```
public class Cercle {
     public double x,y,r;
    // Constructeur
    public Cercle(double centreX, double centreY,
                                                  double rayon)
    {
          x = centreX;
          y = centreY;
          r = rayon;
     }
    //Méthodes pour retourner la circonference et la surface
     public double circonference() { return 2*3.14*r; }
     public double surface() { return 3.14 * r * r; }
}
```

Constructeurs initialisant les Objets

- Rappelons l'ancien Segment de code suvant :

```
Cercle unCercle = new Cercle();
unCercle.x = 10.0; // initialise centre et rayon
unCercle.y = 20.0
unCercle.r = 5.0;
```



Au moment de la création le centre et le rayon ne sont pas définis.

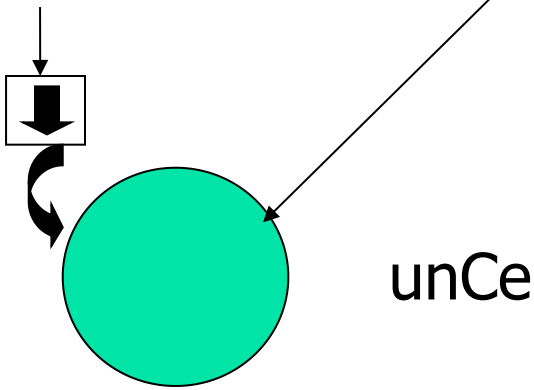
Ces valeurs sont explicitement définies (initialisées) plus tard.

Constructeurs initialisant les Objets

- **Constructeur avec arguments**

```
Cercle unCercle = new Cercle(10.0, 20.0, 5.0);
```

unCercle = new Cercle(10.0, 20.0, 5.0) ;



unCercle est crée avec *centre (10, 20)*
et *rayon 5*

Constructeurs Multiples

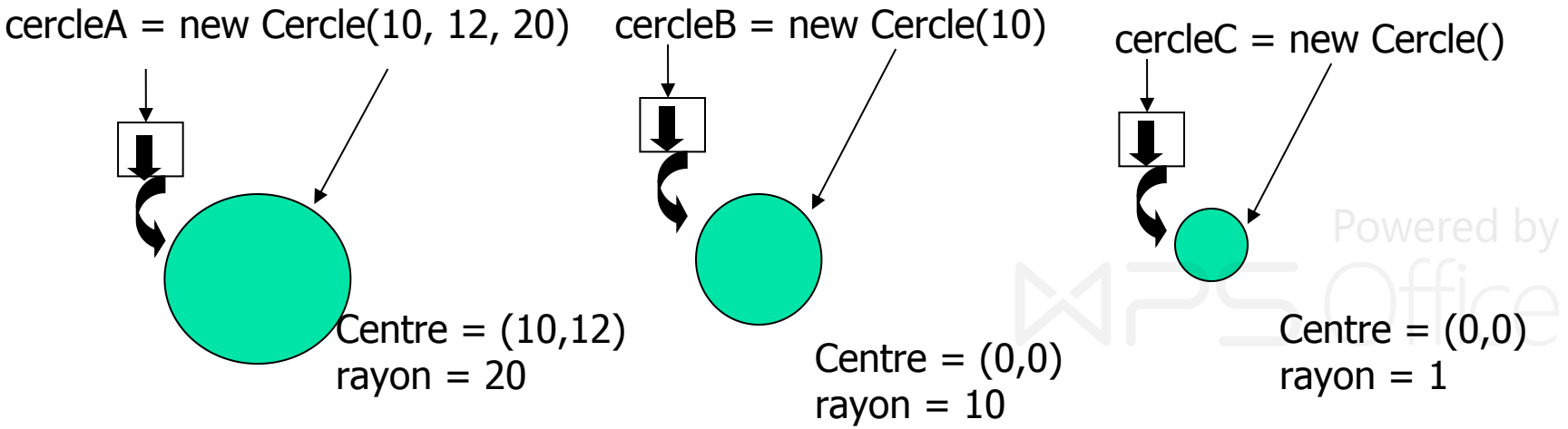
- Parfois nous voulons initialiser de plusieurs façons différentes, en fonction des circonstances.
- Ceci peut être soutenu par l'existence de plusieurs constructeurs ayant des arguments d'entrée différents.

Plusieurs Constructeurs

```
public class Cercle {
    public double x,y,r; // variables d'instance
    // Constructeurs
    public Cercle(double centreX, double cenreY, double rayon) {
        x = centreX;  y = centreY; r = rayon;
    }
    public Cercle(double rayon) { x=0; y=0; r = rayon; }
    public Cercle() { x=0; y=0; r=1.0; }

     //Méthodes pour retourner la circonference et la surface
    public double circonference() { return 2*3.14*r; }
    public double surface() { return 3.14 * r * r; }
}
```

Initialisation par des constructeurs

```
public class TestCercles {

        public static void main(String args[]){
            Cercle cercleA = new Cercle( 10.0, 12.0, 20.0);
            Cercle cercleB = new Cercle(10.0);
            Cercle cercleC = new Cercle();
        }
}
```



Constructeur synthétisé (ou par défaut)
Constructeur par défaut (ou sans arguments)

- Si une classe ne définit pas explicitement un constructeur, il y a toujours un constructeur synthétisé (c'est un constructeur sans argument) .
- Du moment que l'on définisse un constructeur, le constructeur synthétisé n'est plus disponible.
- Il est généralement nécessaire d'avoir unconstructeur sans argument dans une classe (constructeur par défaut).
- Un constructeur est en général une méthode publique.

Surcharge de Méthode

- Les constructeurs ont tous le même nom.
- Les Méthodes sont distinguées par leurs signatures:
  - nom
  - nombre d'arguments
  - types d'arguments
  - position des arguments
- Cela signifie, une classe peut également avoir plusieurs Méthodes usuelles avec le même nom.

- Ne pas confondre avec *redéfinition de méthode* (à venir).

Un Programme avec surcharge de Méthode

```
// Compare.java: a class comparing different items
class Compare {
   static int max(int a, int b)
   {
        if( a > b)
                return a;
        else
        return b;
   }
   static String max(String a, String b)
   {
        if( a.compareTo (b) > 0)
                return a;
        else
                return b;
   }

  public static void main(String args[])
  {
        String s1 = "Rabat";
        String s2 = "Casablanca";
        String s3  = "Méknès";

        int a = 10;
        int b = 20;

        System.out.println(max(a, b)); // quel nombre est plus grand
        System.out.println(max(s1, s2)); // quelle ville est plus grande
        System.out.println(max(s1, s3)); // quelle ville est plus grande
  }
}
```

Le mot clé *this*

- Le mot clé *this* utilisé fait référence à l'objet lui même.
  Il est genéralement utilisé pour accéder aux membres de la classe (depuis ses propres méthodes) quand ils ont le même nom que ceux passés en arguments.

```
public class Cercle {
    public double x,y,r;
    // constructeur
    public Cercle (double x, double y, double r) {
        this.x = x;
        this.y = y;
        this.r = r;
    }
  //Méthodes pour retourner la circonference et la surface
}
```

Le mot clé *this*

- Le mot clé *this* peut aussi être utilisé pour éviter la réécriture du code. Il sert à appeler un constructeur à l'intérieur d'un autre (à condition que cette instruction soit la première du constructeur appelant).

```
public class Cercle {
    public double x,y,r;
    // constructeurs
    public Cercle (double x, double y, double r) {
        this.x = x;
        this.y = y;
        this.r = r;
    }
 public Cercle(double rayon) {this(0.0, 0.0, rayon ); }
 //Méthodes pour retourner la circonference et la surface
}
```

Le mot clé *this*

- Une variable locale ou un paramètre de même nom qu'un attribut, masque ce dernier dans la portée de la variable locale ou du paramètre. On peut toutefois utiliser le mot reservé *this* accéder à l'attribut. (Voir exemple précedent)

- à l'intérieur d'une classe, l'appel d'un membre de la classe n'a pas besoin d'être qualifié. La cible est l'instance courante (qui est désigné par la réference *this* .

Les Membres *Static*

- Java supporte la définition de méthodes et variables globales qui peuvent être accedées sans créer les objets de la classe. De tels membres sont appelés membres Statiques.
- Définir une variable en la marquant par le mot clé static.
- Cette caractéristique est utile lorsqu'on veut créer une variable commune à toutes les instances d'une classe.
- L'un des exemples le plus commun est d'avoir une variable qui tient le compte du nombre d'objets d'une classe qui ont été crées
- Note: Java crée seulement une copie pour une variable statique qui peut être utilisée même *si la classe n'est jamais instantiée*.

Variables *Static*

- Définir utilisant *static:*

```
public class Cercle {
      // variable de classe, une pour la classe Cercle, combien de
//cercles
      public static int nbrCercles;

       //variables d'instance, une pour chaque instance d'un Cercle
      public double x,y,r;
      // Constructeurs...
}
```

- Accés par le nom de la classe (NomClasse.NomVarStat):

```
nCercles = Cercle.nbrCercles;
```

Variables *Static* - Exemple

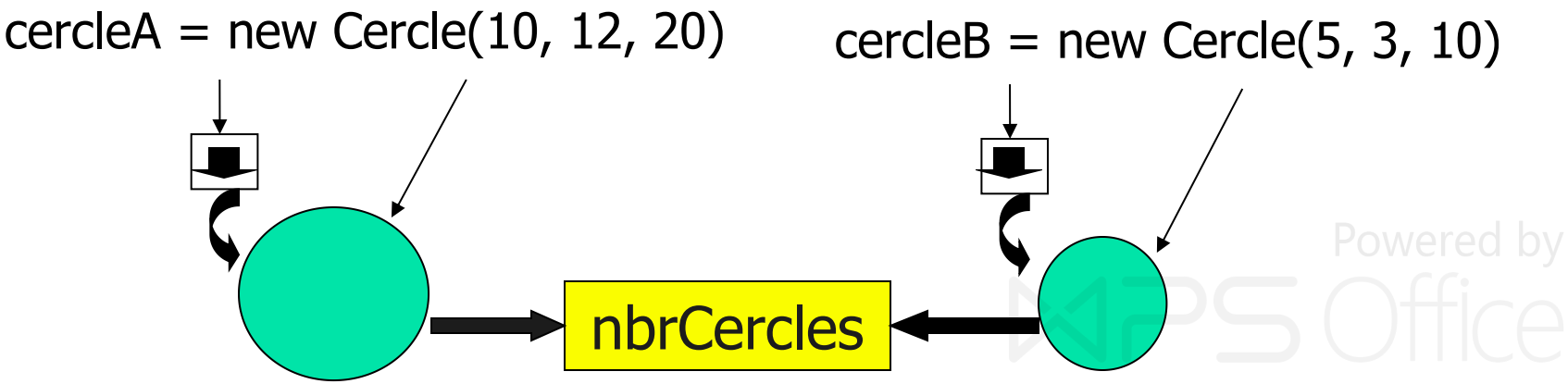
- Utilisation des variables *static :*

```
public class Cercle {
     / variable de classe, une pour la classe Cercle, combien de
//cercles

     private static int nbrCercles = 0;
     private double x,y,r;

     // constructeurs...
     Cercle (double x, double y, double r){
          this.x = x;
          this.y = y;
          this.r = r;
          nbrCercles++;
     }
}
```

Variables de Classe - Exemple

- Utilisation des variables *static :*

```
public class CompteCercles {

        public static void main(String args[]){
            Cercle cercleA = new Cercle( 10, 12, 20);    // nbrCercles = 1
            Cercle cercleB = new Cercle( 5, 3, 10);      // nbrCercles = 2
    }
}
```



Variables d'Instance Vs Static

- Variables d'Instance : Une copie par object. Chaque objet a sa propre variable d'instance.
  - E.g. x, y, r (centre et rayon dans le cercle)

- Variables Static : Une copie par classe.
  - E.g. nbrCercles (nombre total d'objets cercle creés)

Méthodes *Static*

- Une classe peut avoir des méthodes qui sont définis comme statiques (e.g., méthode main).
- Les méthodes *Static* peuvent etre accédés sans utiliser des objets. Aussi, il n'existe AUCUN besoin de créer des objets.
- Ils sont prefixés par les mots clés “static”
- Les méthodes *Static* sont généralement utilisées pour un groupe de libraries de fonctions reliées qui ne dépendent pas de données membres de sa classe. Par exemple, les fonctions de la librairie Math.

Classe Comparator avec méthodes Static

```
// Comparator.java: Une classe avec des Méthodes statiques de Comparaison  de données
class Comparator {
      public static int max(int a, int b)
      {
            if( a > b)
                  return a;
            else
                  return b;
      }

      public static String max(String a, String b)
      {
            if( a.compareTo (b) > 0)
                  return a;
            else
                  return b;
      }
}

class maClasse {
  public static void main(String args[])
  {
      String s1 = "Rabat";
      String s2 = "Casalanca";
      String s3  = "Méknès";

      int a = 10;
      int b = 20;

      System.out.println(Comparator.max(a, b)); // quel nombre est plus grand
      System.out.println(Comparator.max(s1, s2)); // quelle ville est plus grande
      System.out.println(Comparator.max(s1, s3)); // quelle ville est plus grande
  }
}
```

Directement accessible à l'aide de NomClasse (PAS d'objets)

Restrictions sur les méthodes *Static*

- Ils peuvent seulement appeler d'autres méthodes *static*.
- Ils peuvent seulement accéder qux données *static*.
- Ils ne peuvent pas faire réference à “this” ou “super” (plus tard) en aucune façon.

Règles sur les membres *static*

- Un consturcteur ne peut être *Static*.
- Les variables et méthodes de classe peuvent être appelés sans qualification dans toute méthode (de classe ou d'instance).
- Les variables et méthodes d'instance ne peuvent être appelés sans qualification que dans des méthodes d'instance.

Classes et Objets en Java

Passage de Paramètres , Délegation, Controle de Visibilité

Méthodes *Static* dans la classe Cercle

- Comme les *variables de classe*, *les méthodes de classes* peuvent en avoir aussi:

```
public class Cercle {

      //Une méthode de classe
      public static Cercle plusGrand(Cercle a, Cercle b) {
            if (a.r > b.r) return a; else return b;
       }
}
```

- Accédés par le nom de la classe

```
Cercle c1= new Cercle();
Cercle c2 = new Cricle();
Cricle c3 = Cercle.plusGrand(c1,c2);
```

Méthodes *Static* dans la classe Cercle

- Les méthodes de Classe peuvent seuleument accéder aux variables et méthodes *static* .

```
public class Cercle {
      // variable de classe, une pour la classe Cercle, combien de cercles
      private static int nbrCercles = 0;
      public double x,y,r;
      public static void printnbrCercles(){
       // nbrCercles est une variable Static
            System.out.println("Number of Cercles = "  + nbrCercles);
      }
      // Ceci n'est pas VALIDE
      public static void printrayon(){
      {
     // nbrCercles est une variable d'instance (non Static)
            System.out.println("rayon = " +  r);
      }
}
```

Retour à HelloWorld
[le Système invoque la méthode *static* main]

// HelloWorld.java: Hello World program

```
class HelloWorld
{
    public static void main(String args[])
    {
      System.out.println("Hello World");
    }
}
```

Retour aux Constantes
[final peut aussi être déclarée en tant que *static*]

```
class NombresAuCarré{

    static final int MAX_NUMBER = 25;
    public static void main(String args[]){
        final int MAX_NUMBER = 25;
        int lo = 1;
        int auCarré= 0;
        while (auCarré <=  MAX_NUMBER){
            auCarré = lo * lo;     // Calcule le carré
            System.out.println(auCarré);
            lo = lo + 1;       /* Calcule la nouvelle valeur  lo */
        }
    }
}
```

Passage de Paramètres

- Les paramètres de Méthodes qui sont des objets sont passés par réference.
- Copie de la reférence de *l'objet* est passée à la méthode, valeur originale unchangée (pas de copie de l'objet).
- Toute méthode (non *"static"*) posséde un argument caché désignant l'objet appelant

Passage de Paramètres - Exemple

```
public class Test Reference{
    public static void main (String[] args)
    {
        Cercle c1 = new  Cercle(5, 5, 20);
        Cercle c2 = new Cercle(1, 1, 10);
        System.out.println ( “c1 rayon = “ + c1.getrayon());
        System.out.println ( “c2 rayon = “ + c2.getrayon());
        testeurParametre(c1, c2);
          System.out.println ( “c1 rayon = “ + c1.getrayon());
          System.out.println ( “c2 rayon = “ + c2.getrayon());
    }

     …….. cont
```

Passage de Paramètres - Exemple

```
    public static  void  testeurParametre(Cercle cercleA, Cercle
cercleB)
    {
        cercleA.setrayon(15);
        cercleB = new Cercle(0, 0, 100);

        System.out.println ( “cercleA rayon = “ + cercleA.getrayon());
        System.out.println ( “cercleB rayon = “ + cercleB.getrayon());
    }

}
```

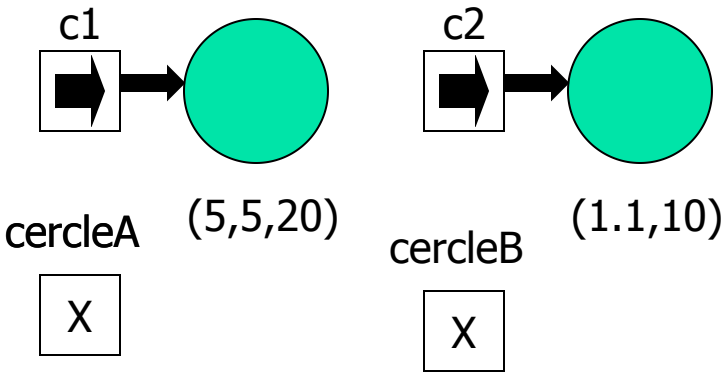
Passage de Paramètres - Exemple

- Sortie –

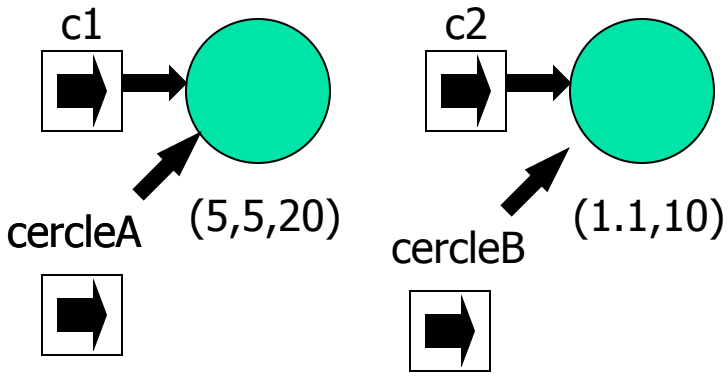
```
c1 rayon =  20.0
c2 rayon = 10.0

cercleA rayon = 15.0
cercleB rayon = 100.0

c1 rayon =  15.0
c2 rayon = 10.0
```

Passage de Paramètres - Exemple

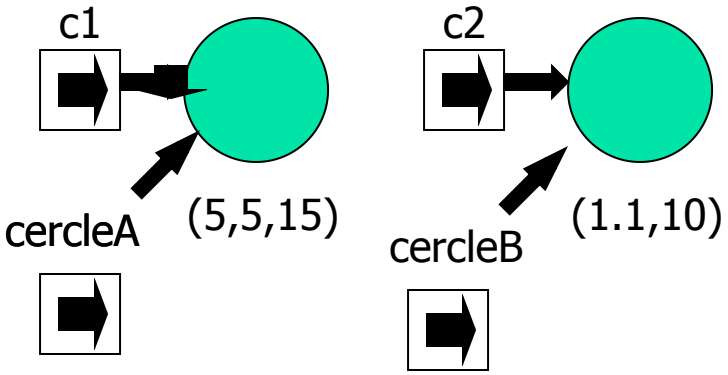
ETAPE1 – Avant appel de testeurParametre()



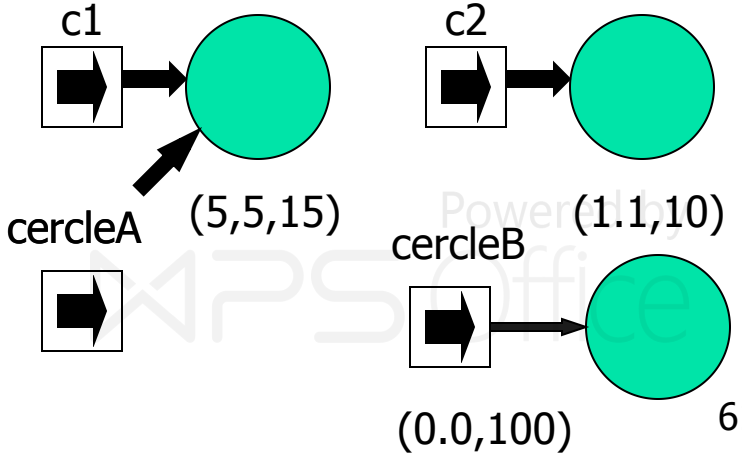
ETAPE2 – testeurParametre(c1, c2)



ETAPE3 – circlA.setrayon(15)

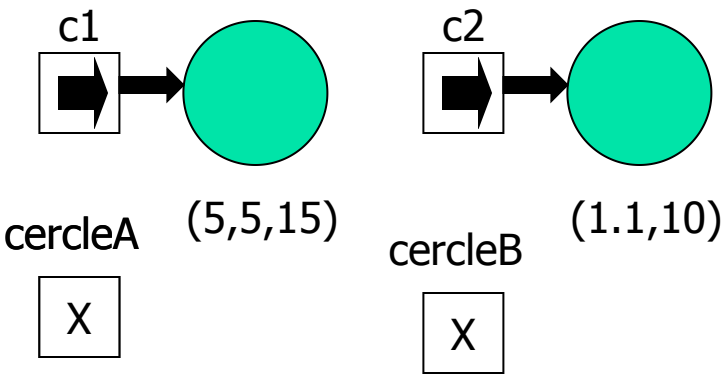


ETAPE4 – circlB = new Cirlce(0,0,100)



Passage de Paramètres - Exemple

ETAPE5 – Après Retour de testeurParametre



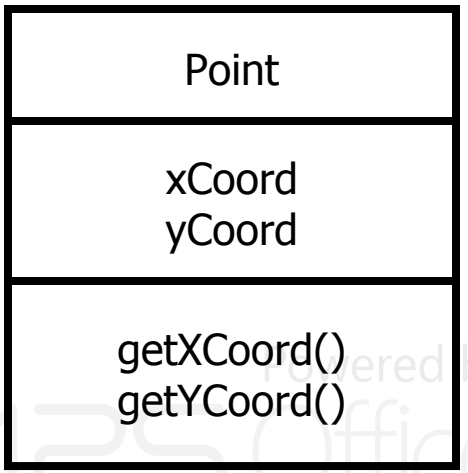
Délegation

- Abilité d'une classe à déleguer ses responsabilités à une autre classe.

- Une façon de permettre à un objet l'invocation des services d'autres objets au travers la contenance.

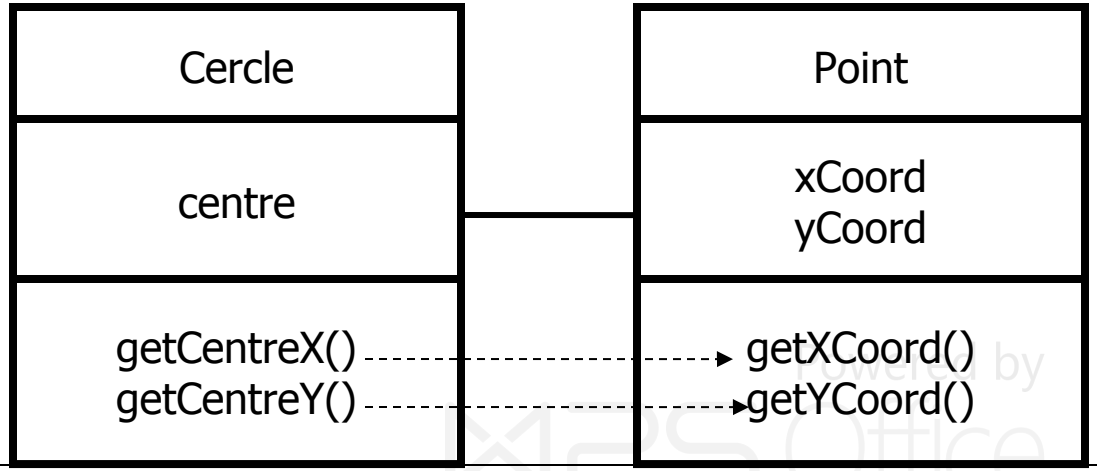
Délégation - Exemple

```
public class Point {
        private double xCoord;
        private double yCoord;
        // constructeur
              ……….

        public double getXCoord(){
              return xCoord;
        }
        public double getYCoord(){
              return yCoord;
        }
}
```



Délégation - Exemple

```
public class Cercle {
        private Point centre;
        public double getCentreX(){
            return centre.getXCoord();
        }
        public double getCentreY(){
            return centre.getYCoord();
        }
}
```



Contrôle de Visibilité : Protection de données et Encapsulation

- Java fournit le contrôle sur la *visibilité* des variables et méthodes, *l'encapsulation,* enfermant en toute sécurité les données à l'intérieur d'une *capsule* qu'est la classe.
- Prévient les programmeurs de dépendre des détails de l'implémentation d'une classe, pour pouvoir faire des mises à jour sans soucis
- Aide à proteger contre des faux usages ou erreurs accidentelles.
- Garde un code élégant et propre (facile à maintenir)

Modificateurs de Visibilité : Public, Private, Protected

- *Public:* mot clé appliqué à une classe, la rend disponible/visible partout. Appliqué à une méthode ou une variable, la rend complétement visible (accéssible à toute classe).
- *Private:* les attributs ou méthodes d'une classe sont seulement visible à l'intérieur d'une classe.
- *Protected:* Voir Plus loin (Héritage).
- Les classes sont généralement groupées en *packages.* Toute classe appartient à un package.

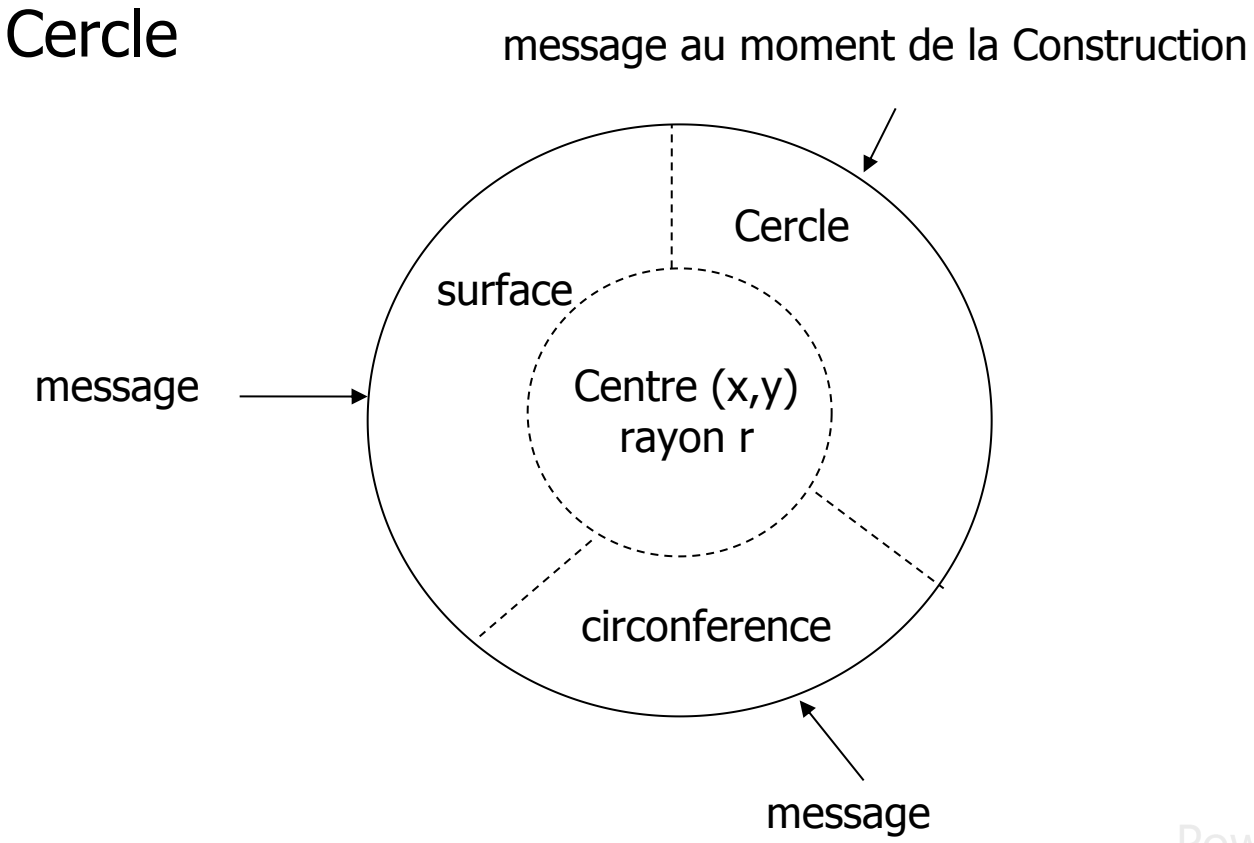
===>

  - Si Aucun modificateur de visibilité n'est specifié, alors un membre de la classe se comporte comme *public* dans son *package* et *private* dans les autres *packages* (Visibilité au niveau package / visibilité par défaut). C-à-d une méthode ou une varable est accessible à toute classe du même *package*.

Visibilité

```
public class Cercle {
    private double x,y,r;

        // constructeur
     public Cercle (double x, double y, double r) {
            this.x = x;
            this.y = y;
            this.r = r;
     }
     //Méthodes pour retorner la circonference et la surface
    public double circonference() { return 2*3.14*r;}
    public double surface() { return 3.14 * r * r; }
}
```

Visibilité



Accesseurs – "Getters/Setters"

```
public class Cercle {
    private double x,y,r;

    //Méthodes pour manipuler les variables privés (x, y, z)
    public double getX() { return x;}
    public double getY() { return y;}
    public double getR() { return r;}
    public double setX(double x) { this.x = x;}
    public double serY(double y) { this.y = y;}
    public double setR(double r) { this.r = r;}

}
```


Plus sur la "Visibilité" lors de la Discussion sur les packages et l'héritage

Classes et Objets en Java

Relations entre classes

Réutilisation
Association et Composition

Réutilisation ?

- Comment utiliser une classe comme matériau pour concevoir une autre classe répondant à de nouveaux besoins ?
- Quels sont les attentes de cette nouvelle classe ?
  - besoin des «services» d'une classe existante (les données structurées, les méthodes, les 2).
  - faire évoluer une classe existante (spécialiser, ajouter des fonctionnalités, ...) (plus loin).

Réutilisation ?

- Quelle relation existe entre les 2 classes ?
- Dans une conception objet, des relations sont définies, des règles d'association et de relation existent.
- Un objet fait appel un autre objet.
- Un objet est crée à partir d'un autre : il hérite (plus loin).

La relation *client/serveur*
Association

- Un objet o1 de la classe C1 utilise un objet o2 de la classe C2 via son interface (attributs, méthodes).
- o1 délègue une partie de son activité et de son information à o2.

La relation *client/serveur - Association/Agrégation*

- o2 a été construit et existe par ailleurs.
- Faire une référence dans C1 vers un objet de C2;
- Attention o2 est autonome, indépendant de o1, il peut être partagé.
- C'est une **association ou agrégation.**

Le client

```
public class C1
{
  private C2 o2;
  ...
}
```


Le serveur

```
public class C2
{
   ...
   ...
}
```

La relation *client/serveur - Association/Agrégation*

Le client

```
public class Cercle
{
  private Point centre;
  ...
}
```

Le serveur

```
public class Point
{
   ...
   ...
}
```

- *Un Point est associé à un Cercle. Une association entre Point et Cercle.*
- La relation ou l'association entre Point et Cercle du genre *"a un/has a"*.
- Un Cercle *"a un/posséde"* un centre (Point). C'est une **agrégation.**

La relation *client/serveur* *Composition*

- Comment faire pour que o1 possède son o2 à lui.
- La solution se trouve dans le constructeur de C1 : doter o1 d'un objet o2 qui lui appartienne. C'est une **composition.**

Le client

```
public class C1
{
  private C2 o2;
  ...
  C1(...){
     o2 = new C2(...);
  }
  ...
}
```

Le serveur

```
public class C2
{
    ...
    ...
}
```

La relation *client/serveur - Composition*

Le client

```
public class Cercle
{
  private Point centre;
  ...
  public Cercle(...){
     centre = new Point(...);
  }
  ...
}
```

Le serveur

```
public class Point
{
    ...
    ...
}
```

L'exemple du cercle

- Le cercle c1 a un centre. C'est le point p1.
- Utilisation d'un objet de type Point, car la classe existe déjà.
- Si agrégation, tout changement effectué sur p1 a des conséquences sur c1.
  - Si on déplace p1, on déplace tous les cercle ayant comme centre p1. Composition ?
- Mais plusieurs cercle peuvent partager le même centre ... Impossible si composition.
  - ça se discute !

Nouveau problème

- Si le serveur est insuffisant, incomplet, inadapté ?
  - un objet *Point* répond à un besoin «mathématique».
  - Si on veut en faire un objet graphique, avec une couleur et qu'il soit capable de se dessiner ?
  - Mais en gardant les propriétés et les fonctionnalités d'un *Point*.
- La solution en POO : **l'héritage.**
  - **Définir une nouvelle classe à partir de la définition d'une classe existante.**
  - **Spécialiser, augmenter.**

POO en JAVA

Classes et Objets

Support de présentation
http://www.cloudbus.org/~raj/254/

https://perso.limsi.fr/allauzen/cours/cours11_12/MANJAVA/pac_2.ppt