Chapitre 9

Les principes du concept objet

Introduction

Les objets sont beaucoup plus qu'une structure syntaxique.

Ils sont régis par des principes essentiels, qui constituent les fondements de la programmation objet.

Dans ce chapitre nous verrons :

- La communication objet (les objets en mémoire, les passages par référence).
- Que les objets contrôlent leur fonctionnement (concepts d'encapsulation, de protection des données).
- La notion d'héritage entre classes.
- Le polymorphisme.

1. La communication objet

- En définissant une classe, le développeur créé un modèle, qui décrit les fonctionnalités des objets utilisés par le programmeur.

- Les objets sont créés en mémoire à partir de ce modèle par copie des données et des méthodes.

- **`New`** permet d'initialiser les données de l'objet et fournit l'adresse où se trouve les informations stockées.

1.1 Les données *static*

- Si le mot- clé ***static*** est placé devant une variable ou une méthode :
  - ⇒ Réservation d'un seul et unique emplacement en mémoire.
  - ⇒ Espace accessible pour tous les objets du même type.

- Si le mot- clé ***static*** n'apparaît pas
  - ⇒ Réservation, à chaque appel de l'opérateur **new**, d'un espace mémoire.

Exemple Compter des cercles :

```
public class Cercle
{
  public int x, y, rayon;    // position du centre et rayon
  public static int nombre; // nombre de cercle

  public void créer()
  {
    Scanner clavier = new Scanner(System.in);
    System.out.print("Position en x : ");
    x = clavier.nextInt();
    System.out.print("Position en y : ");
    y = clavier.nextInt();
    System.out.print("Rayon          : ");
    rayon = clavier.nextInt();
    nombre = nombre + 1;
  }
  ...
} // Fin de la classe cercle
```

- Dans la classe **`Cercle`** :
  - **`x`**, **`y`**, **`rayon`** sont des **variables d'instance**
  - **`nombre`** est une **variable de classe**

- Différenciées par la présence ou non du mot-clé **`static`**.

- La variable de classe **`nombre`** est un espace mémoire commun à tous les objets de la même classe.

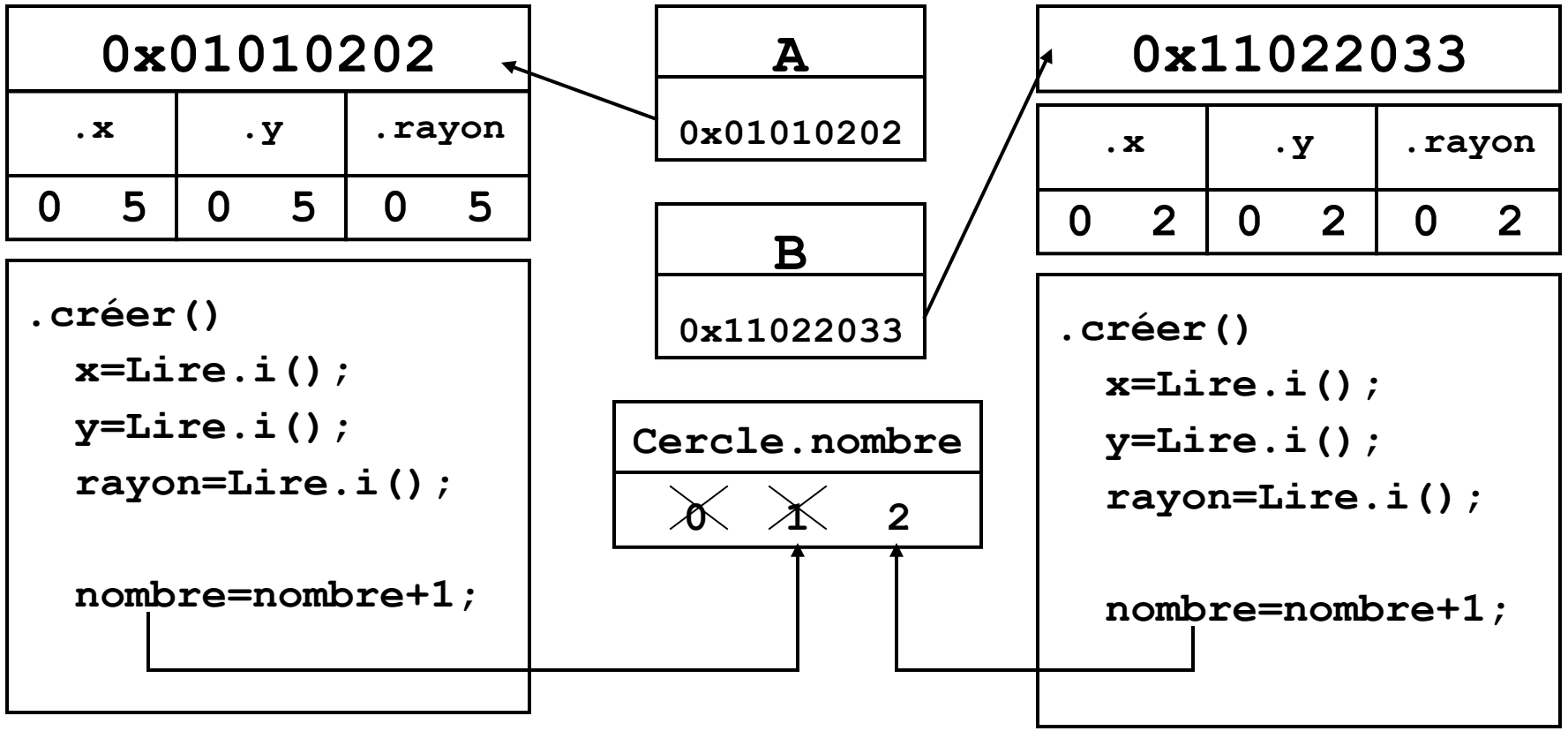
→ *il n'est donc pas nécessaire de créer d'instance d'une classe (d'objet) pour accéder à une variable ou une méthode déclarée **"static"**, on peut utiliser directement le nom de la classe.*

Exécution de l'application compter des cercles :

```
public class CompterDesCercles
{
  public static void main(String [] arg)
  {
    Cercle A = new Cercle();
    A.créer();
    System.out.println("Nombre de cercle : "+Cercle.nombre);

    Cercle B = new Cercle();
    B.créer();
    System.out.println("Nombre de cercle : "+Cercle.nombre);
  }
}
```

CompterDesCercles



1.2 Passage de paramètres par référence

• Jusqu'à présent, nous avons passé les paramètres **par valeur** :

⇒ Impossible de modifier plusieurs données dans une fonction et de les retourner.

⇒ Utilisation d'une autre technique de passage des paramètres : **passage par référence**.

• Le passage des paramètres par référence permet la modification des données d'un objet.

• Utilisé lorsqu'on passe en paramètre un objet.

Exemple : échanger la position de deux cercles. Passage d'un objet **`Cercle`** en paramètre

```
public class Cercle
{
   public void échangerAvec( Cercle autre )
   {
      int tmp;

      tmp = x;
      x = autre.x;
      autre.x = tmp;

      tmp = y;
      y = autre.y;
      autre.y = tmp;
   }

   ...
}
```

```
public class EchangerDesCercles
{
   public static void main( String [] arg )
   {
      Cercle A  = new Cercle();
      A.créer();
      System.out.println("Le cercle A");
      A.afficher();

      Cercle B = new Cercle();
      B.créer();
      System.out.println("Le cercle B");
      B.afficher();
```

```
        B.échangerAvec(A);

        System.out.println("Après echange");

        System.out.println("Le cercle A");
        A.afficher();

        System.out.println("Le cercle B");
        B.afficher();
    }
} // Fin de la classe EchangerDesCercles
```

2. Les objets contrôlent leur fonctionnement

- Des outils de **création** des objets
- Des outils appropriés pour **utiliser** un objet

2.1 La notion d'*encapsulation*

• La manipulation des objets doit se faire avec soin.

• On ne doit pas pouvoir modifier des objets sans utiliser les méthodes prévues à cet effet.

• Exemple de la classe Cercle :

- La modification des coordonnées du centre doit se faire avec la méthode **`deplacer()`**:

```
A.deplacer(2,4);
```

- Pour l'instant on peut aussi directement modifier les coordonnées du centre :

```
A.x=2; A.y=4;
```

2.2 La protection des données

- 3 niveaux de protection :
  - ***public*** : accessible par tous les objets de l'application.
  - ***private*** : accessible seulement par les membres de la classe.
  - ***protected*** : accessible seulement par les membres de la classe ou par les membres d'une sous-classe (voir la notion d'héritage).
- Par défaut, c'est le type **public** qui est employé.

Exemple

```
public class CerclePrivé
{
   private int  x, y, rayon ;

   public void créer()
   {
      Scanner clavier = new Scanner(System.in);
      System.out.print("Position en x : ");
      x = clavier.nextInt();
      System.out.print("Position en y : ");
      y = clavier.nextInt();
      System.out.print("Rayon          : ");
      rayon = clavier.nextInt();
   }
```

```
...

public void déplacer(int nx, int ny)
{
   x = nx;
   y = ny;
}

public void afficher()
{
   System.out.println("Centre en " + x + "," + y);
   System.out.println("Rayon : " + rayon);
}

...
```

```
    ...

    public void échangerAvec( CerclePrivé autre )
    {
        int tmp;

        tmp = x;
        x = autre.x;
        autre.x = tmp;

        tmp = y;
        y = autre.y;
        autre.y = tmp;
    }

}   // Fin de class CerclePrivé
```

Utilisation de la classe **`CerclePrive`** :

```
public class FaireDesCerclesPrives
{
  public static void main( String [] arg )
  {
    CerclePrivé A = new CerclePrivé();
    Scanner clavier = new Scanner(System.in);

    A.afficher();
    System.out.println("Entrez le rayon : ");
    A.rayon = clavier.nextInt();
    System.out.println("Cercle de rayon:" + A.rayon);
  }
}
```

Erreur : accès en écriture à une donnée privée

Erreur : accès en lecture à une donnée privée

Les données privées ne peuvent être consultées ou modifiées que par des méthodes de la classe où elles sont déclarées.

Les autres classes ne pourront y accéder qu'au travers de méthodes publiques :

- **Méthodes pour l'accès en consultation**
- **Méthodes pour l'accès en modification**

2.3 Contrôler les données

Exemple : contrôler les données d'un cercle

⇒ On veut éviter que le rayon soit négatif ou qu'il dépasse la valeur 100.

```
public class CercleControle
{
   private int x, y, r ;

   public void modifier_rayon()
   {
      Scanner clavier = new Scanner(System.in);
      do
      {
         System.out.print("Rayon : ");
         rayon = clavier.nextInt();
      } while ( rayon < 0 || rayon >  100);
   }
   ...
}
```

```
public class FaireDesCerclesControles
{
   public static void main( String [] arg )
   {
      CercleControle A = new CercleControle();
      A.modifier_rayon();
      A.afficher();
   }
}
```

⇒ La modification de la variable **privée** **`rayon`** passe obligatoirement par l'utilisation de la méthode **publique** **`modifier_rayon()`**

⇒ La valeur saisie pour le rayon est vérifiée.

⇒ L'utilisateur ne peut pas mettre directement une valeur pouvant être erronée dans la variable **`rayon`**.

2.4 La notion de constante

- Possibilité de déclarer des constantes

⇒ Utilisation du mot- clé **`final`** :

```
public final int LargeurEcran = 1024;
```

- Des variables de classes peuvent être constantes :

```
public final static int LargeurEcran = 1024;
```

2.5 Méthodes privées

Possibilité de déclarer les méthodes d'une classe (fonctions) en mode **private**.

Exemple : méthode de la classe **`Cercle`**, de contrôle de la validité du rayon d'un cercle.

```
private int rayonVerifie()
{
    ...
}
```

⇒n'est visible et donc utilisable que dans la classe **`Cercle`**.

3. Les constructeurs

• Les constructeurs sont utilisés pour initialiser correctement les données d'un objet au moment de la création de l'objet en mémoire.

• Le constructeur par défaut initialise :
  – les variables numériques à 0 ou 0.0
  – Les caractères à '\0'
  – Null pour les objets (chaînes de caractères, etc.)

• **`Cercle C = new Cercle();`**
  – **`Cercle()`**, est le constructeur par défaut.

3.1 Définir le constructeur d'une classe

Possibilité de définir son propre constructeur. Le constructeur est une méthode portant le **même nom** que la classe.

Exemple :

```
public class Cercle
{
   public Cercle()
   {
      System.out.print("Position en x : ");
      x = clavier.nextInt();
      System.out.print("Position en y : ");
      y = clavier.nextInt();
      r = rayonVérifié();
   }
   ...
```

Remarques

- Attention : un constructeur n'est pas typé (ne rien placer devant dans l'en-tête, pas même **`void`**).

- Le constructeur est appelé par l'opérateur **`new`** :

```
Cercle A = new Cercle();
```

3.2 Surcharge de constructeur

• Possibilité de définir plusieurs constructeurs pour une même classe.

⇒ **Surcharge** du constructeur

```
public Cercle(int centrex, int centrey)
{
    x = centrex ;
    y = centrey;
}
```

• Le constructeur à utiliser est déterminé en regardant la liste des paramètres attendus par chaque constructeur.

Remarque

- Il est possible de surcharger n'importe quelle méthode d'une même classe, c'est à dire d'avoir plusieurs méthodes portant le même nom.
- La méthode à utiliser est déterminée en regardant la liste des paramètres attendus par chaque méthode.

Exemple :

```
public class Cercle
{
  public void modifier (int nx)
  public void modifier (int nx, int ny)
  public void modifier (int nx, int ny, int nr)
```

4. L'héritage

Concept permettant de créer de nouvelles classes en réutilisant les fonctionnalités d'une autre classe, en lui apportant des variations.

$\Rightarrow$ Les méthodes définies pour un ensemble de données sont réutilisables pour des variantes de cet ensemble.

Exemple :

Création d'une classe **`Forme`** contenant l'ensemble des comportements géométriques.

$\Rightarrow$ Peut être réutilisée pour la classe **`Cercle`**.

$\Rightarrow$ Possibilité d'ajouter de nouveaux comportements supplémentaires à la classe **`Forme`**.

4.1 La relation « est un »

Pour déterminer si une classe B **Hérite** d'une classe A, il suffit de savoir s'il existe une relation « **est un** » entre B et A.

Syntaxe :

```
Class B extends A
{
    // données et méthodes de la classe B
}
```

- **B** est une **sous-classe** de A (B = classe dérivée)
- **A** est une **super-classe** (A = classe de base, classe mère)

4.2 Exemple

Un cercle « est une » forme géométrique

```
public class Forme
{
   protected int  x, y, couleur;

   public Forme(int nx, int ny)
   {
      x = nx ;
      y = ny ;
      couleur = 0;
   }
   ...
```

```
public void afficher()
{
  System.out.println("Position en " + x + "," + y);
  System.out.println("Couleur : " + couleur);
}

public void échangerAvec(Forme autre)
{
  int tmp;
  tmp = x;
  x = autre.x;
  autre.x = tmp;
  tmp = y;
  y = autre.y;
  autre.y = tmp;
}
```

```
    public void déplacer(int nx, int ny)
    {
      x = nx;
      y = ny;
    }

} // Fin de la classe Forme
```

Utilisation :

```
public class Cercle extends Forme
{
   public final static int  TailleMax = 100;
   private int  rayon;

   public Cercle(int xx, int yy)
   {
      super(xx, yy);
      couleur = 10;
      rayon = rayonVérifié();
   }

   public void afficher()
   {
      super.afficher();
      System.out.println("Rayon : " + rayon);
   }
```

```
    private int rayonVérifié()
    {
        int tmp;
        Scanner clavier = new Scanner(System.in);

        do
        {
            System.out.print("Rayon : ");
            tmp = clavier.nextInt();
        } while ( tmp < 0 || tmp > TailleMax) ;
        return tmp;
    }

} // Fin de la classe Cercle
```

4.3 Le constructeur d'une classe héritée

- Les classes dérivées possèdent leur propre constructeur.

- Le constructeur de la classe **Cercle** appellera le constructeur de la classe **Forme**.

- S'il n'y a pas de constructeur, le compilateur cherche le constructeur par défaut (sans paramètre) de la classe supérieure (**Forme**).

Attention : problème s'il n'y a pas de constructeur sans paramètre dans la classe supérieure.

⇒ On peut appeler directement un constructeur de la classe mère depuis le constructeur de la classe avec le mot clé **super** :

```
public Cercle(int xx, int yy)
{
    super(xx, yy);
    couleur = 10;
    rayon = rayonVérifié();
}
```

5. Le polymorphisme

Polymorphisme : une méthode peut se comporter différemment suivant l'objet sur lequel elle est appliquée.

Exemple : la méthode **afficher()** :

```
public class FormerDesCercles
{
  public static void main(String [] arg)
  {
    Cercle A  = new Cercle(5, 5);
    A.afficher(); // utilisation de la méthode définie
                  // dans Cercle
    Forme F = new Forme (10, 10);
    F.afficher(); // utilisation de la méthode définie
                  // dans Forme
  }
}
```

@override

Si dans une classe dérivée on redéfinit une méthode d'une classe de base, on peut faire précéder cette méthode par le mot clef **@override** :

```
public class Cercle extends Forme
{
    ...
    // On suppose qu'on a une méthode tracer() dans Forme
    @override
    public void tracer(Graphics g)
    {
        g.drawOval(x-rayon/2, y-rayon/2, rayon*2, rayon*2);
    }
```

Cette annotation sert pour deux choses :

- pour la génération de documentation avec JavaDoc.
- le compilateur vérifiera que dans la classe de base il y a bien une méthode qui a ce prototype.

@Override permet au compilateur de vérifier l'authenticité de la surcharge, ce qui permet d'éviter les erreurs de frappe :

```
public abstract class Parent {
    public void doThat() {...}
}
```

```
public class Child extends Parent {
   @Override
   public void doTaht() {...}
}
```

→ provoquera une erreur de compilation car **`doTaht`** n'existe pas dans la classe Parent.