Premiers Pas en Programmation Objet : les Classes et les Objets

Formateur : CHACHIA ABDELILAH

Dans la première partie de ce cours, nous avons appris à manipuler des objets de type simple : entiers, doubles, caractères, booléens. Nous avons aussi appris comment utiliser les tableaux pour stocker des collections d'objets de même type : tableaux d'entiers, tableaux de booléens. . .Cependant, la majorité des programmes manipulent des données plus complexes. Pour ecrire un logiciel bancaire, il faudra représenter dans notre langage de programmation l'ensemble des informations caractérisant un compte et coder les actions qui s'effectuent sur les comptes (retrait, dépot) ; un logiciel de bibliothèque devra représenter l'ensemble des informations caractéristiques des livres et coder les opérations d'ajout ou de retrait d'un livre . . ..

L'approche Orientée Objet, que nous allons aborder dans ce chapitre, consiste à rendre possible dans le langage de programmation la définition d'objets (des livres, des comptes . . .) qui ressemblent à ceux du monde réel, c'est à dire caractérisés par un état et un comportement. L'état d'un compte, pourra être défini par son numéro, le nom de son titulaire, son solde ; son comportement est caractérisé par les opérations de dépot, de retrait et d'affichage du solde.

Dans nos programmes nous aurons plusieurs objets comptes. Chacuns ont un état qui leur est propre, mais ils ont les mêmes caractéristiques : ce sont tous des comptes. En programmation Orientée Objet, nous dirons que ces différents objets comptes sont des objets instances de la classe Compte. Une classe est un prototype qui définit les variables et les méthodes communes à tous les objets d'un même genre. Une classe est un *patron d'objets*. Chaque *classe* définit la façon de créer et de manipuler des *Objets* de ce type.

A l'inverse, un objet est toujours un exemplaire, une instance d'une classe (son patron).

Ainsi, pour faire de la programmation Objet, il faut savoir concevoir des classes, c'est à dire définir des modèles d'objets, et créer des objets à partir de ces classes.

Concevoir une classe, c'est définir :

1. **Les données** caractéristiques des objets de la classe. On appelle ces caractéristiques les *variables d'instance*.
2. **Les actions** que l'on peut effectuer sur les objets de la classe . Ce sont les *méthodes* qui peuvent s'invoquer sur chacuns des objets de la classe.

Ainsi, chaque objet crée possèdera :

1. **Un état**, c'est à dire des valeurs particulières pour les variables d'instances de la classe auquel il appartient.

2. **Des méthodes** qui vont agir sur son état.

2.1 Définir une Classe

Une classe qui définit un type d'objet a la structure suivante :
- Son nom est celui du type que l'on veut créer.
- Elle contient les noms et le type des caractéristiques (les variables d'instances) définissant les objets de ce type.
- Elle contient les méthodes applicables sur les objets de la classe.

Pour définir une classe pour les aires des carrés on aurait par exemple :

```
class Carre {
    double  longeur;
    double  aire;


    void afficher(){
      System.out.println("longeur:  "+ this.longeur);
    }
    void calculaire(){
      this.aire = this.longeur * this.longeur;
    }
    void SetLongeur(double longeur1){
        this.longeur = longeur1 ;
     }
}
```

2.1.1 Les Variables d'instances

La déclaration d'une variable d'instance se fait comme une déclaration de variable locale au main ou à un sous programme : on donne un nom, précédé d'un type. La différence est que cette déclaration se fait **au niveau de la classe** et non à l'intérieur d'un sous programme.

Ainsi l'état de chaque objet instance de la classe Carre que nous créerons par la suite sera constitué d'une valeur pour chacune de ces deux variables d'instances. Pour chaque objet instance de la classe Carre nous pourrons connaitre la valeur de son longeur, son aire. Ces valeurs seront **propres à chaque objet**.

2.1.2 Les méthodes : premier aperçu

Nous n'allons pas dans ce paragraphe décrire dans le détail la définition des méthodes d'objets, mais nous nous contentons pour l'instant des remarques suivantes : une classe définissant un type d'objets comportera autant de méthodes qu'il y a d'opérations utiles sur les objets de la classe.

La définition d'une méthode d'objet (ou d'instance) ressemble à la défintion d'un sous programme : un type de retour, un nom, une liste d'arguments précédés de leur type. Ce qui fait d'elle une méthode d'objet est qu'elle ne comporte pas le mot clé `static`. Ceci (plus le fait que les méthodes

sont dans la classe Carre) indique que la méthode va pouvoir être invoquée (appelée) sur n'importe quel objet de type Carre et modifier son état (le contenu de ses variables d'instances).

Ceci a aussi des conséquences sur le code de la méthode comme par exemple l'apparition du mot clé `this`, sur lequel nous reviendrons lorsque nous saurons comment invoquer une méthode sur un objet.

2.2 Utiliser une Classe

Une fois définie une classe d'objets, on peut utiliser le nom de la classe comme un nouveau type : déclaration de variables, d'arguments de sous programmes .... On pourra de plus appliquer sur les objets de ce type toutes les méthodes de la classe.

2.2.1 Déclarer des objets instances de la classe

Si la classe Carre est dans votre répertoire de travail, vous pouvez maintenant écrire une autre classe, par exemple `test` qui, dans son main, déclare une variable de type Carre :

```
public class Test {
  public static void main (String [] arguments){
    Carre c1 = new Carre();
  }
}
```

Comme pour les tableaux, une variable référencant un objet de la classe Carre, doit recevoir une valeur, soit par une affectation d'une valeur déjà existante, soit en créant une nouvelle valeur avec `new` avant d'être utilisée. On peut séparer la déclaration et l'initialisation en deux instructions :

```
public class Test {
  public static void main (String [] arguments){
    Carre c1;
    c1 = new Carre();
  }
}
```

Après l'exécution de `c1 = new Carre();` chaque variable d'instance de `c1` a une valeur par défaut. Cette valeur est 0.00 pour longeur et aire.

2.2.2 Accéder et modifier les valeurs des variables d'instances d'un objet

La classe Carre définit la forme commune à tous les carrés. Toutes les variable de type Carre auront donc en commun cette forme : une longeur, un aire. En revanche, elles pourront représenter des carrés différents.

Accéder aux valeurs des variables d'instance

Comment connaître la longeur du carre `c1` ? Ceci se fait par l'**opérateur noté par un point** :

```
public class Test {
  public static void main (String [] arguments){
        Carre c1 = new Carre();
      System.out.println(c1.longeur);
```

```
  }
}
```

La dernière instruction à pour effet d'afficher à l'écran la valeur de la variable d'instance longeur de `c1`, c'est à dire l'entier 0.00. Comme le champ longeur est de type double, l'expression `c1.longeur` peut s'utiliser partout où un entier est utilisable :

```
public class Test {
  public static void main (String [] arguments){
    Carre c1 = new Carre();
    double x;
    double []tab = {2.00,4.00,6.00};
    tab[1]= c1.longeur;
    x = c1.longeur+34 / (c1.longeur +4);

  }
}
```

Modifier les valeurs des variables d'instance

Chaque variable d'instance se comporte comme une variable. On peut donc lui affecter une nouvelle valeur :

```
public class Test {
  public static void main (String [] arguments){
    Carre c1 = new Carre();
    Carre c2 = new Carre();
    c1.longeur =10.3;
    c2.longeur =15;
    System.out.println
    ("valeur de longeur de c1:" + c1.longeur);
    System.out.println
    ("valeur de longeur de c2:" + c2.longeur );


  }
}
```

c1 représente maintenant le carré numero 1 qui a comme longeur 10.3 . et c2 le carré numero 2 qui a comme longeur 15.

Affectation entre variables référençant des objets

l'affectation entre variables de types Carre est possible, puisqu'elles sont du même type, mais le même phenomène qu'avec les tableaux se produit : les 2 variables référencent le même objet et toute modification de l'une modifie aussi l'autre :

```
public class TestBis {
  public static void main (String [] arguments){
    Carre c1 = new Carre();
```

```
    Carre c2 = new Carre();
    c1.longeur =10.3;
    Carre c2 = new Carre();
    c2 = c1;
    System.out.println
    ("valeur de longeur de c1:" + c1.longeur);
    System.out.println
    ("valeur de longeur de c2:" + c2.longeur );


  }
}
```

Trace d'exécution :

```
%> java TestBis
valeur de longeur c1: 10.3
valeur de longeur c2: 10.3
```

2.2.3 Invoquer les méthodes sur les objets.

Une classe contient des variables d'instances et des méthodes. Chaque objet instance de cette classe aura son propre état, c'est à dire ses propres valeurs pour les variables d'instances. On pourra aussi invoquer sur lui chaque méthode non statique de la classe. Comment invoque-t-on une méthode sur un objet ?

> Pour invoquer la méthode `afficher()` sur un objet `c1` de la classe Carre il faut écrire : `c1.afficher();` .

Comme l'illustre l'exemple suivant :

```
public class TestAfficher {
  public static void main (String [] arguments){
      Carre c1 = new Carre();
      Carre c2 = new Carre();
      c1.longeur =100;
      c1.afficher();
      c2.afficher();


  }
}
```

L'expression `c1.afficher();` invoque la méthode `afficher()` sur l'objet `c1`. Cela a pour effet d'afficher à l'écran la longeur de `c1` c'est à dire 100. L'expression `c2.afficher();` invoque la méthode `afficher()` sur l'objet `c2`. Cela affiche la longeur de `c2` c'est à dire 0.

```
> java TestAfficher
solde: 100
solde: 0
```

Ainsi, les méthodes d'objets (ou méthodes non statiques) s'utilisent par invocation sur les objets de la classe dans lesquelles elles sont définies. l'objet sur lequel on l'invoque ne fait pas partie de la liste des arguments de la méthode. Nous l'appellerons l'objet courant.

2.3 Retour sur les méthodes non statiques

Dans une classe définissant un type d'objet, on définit l'état caractéristiques des objets de la classe (les variables d'instances) et les méthodes capables d'agir sur l'état des objets (méthodes non statiques). Pour utiliser ces methodes sur un objet `x` donné, on ne met pas x dans la liste des arguments de la méthode. On utilisera la classe en déclarant des objets instances de cette classe. Sur chacun de ces objets, la notation pointée permettra d'accéder à l'état de l'objet (la valeur de ses variables d'instances) ou de lui appliquer une des méthodes de la classe dont il est une instance.
Par exemple, si `c1` est un objet instance de la classe Carre `c1.longeur` permet d'accéder a la longeur de ce carre, et `c1.calculaire()` permet d'invoquer la méthode calculaire sur `c1`.

2.3.1 Les arguments des méthodes non statiques

Contrairement aux sous programmes statique que nous écrivions jusqu'alors, on voit que les méthodes non statiques ont un **argument d'entrée implicite**, qui ne figure pas parmi les arguments de la méthode : l'objet sur lequel elle sera appliqué, que nous avons déjà appelé *l'objet courant*.

Par exemple, la méthode `afficher` de la classe Carre n'a aucun argument : elle n'a besoin d'aucune information supplémentaire à l'objet courant.
Une méthode d'objet peut cependant avoir des arguments. C'est le cas par exemple de SetLongeur : on fournit une longeur pour mettre a jour la valeur de sa longeur.

Les seuls arguments d'une méthode non statique sont les informations nécessaires à la manipulation de l'objet courant (celui sur lequel on invoquera la méthode), jamais l'objet courant lui même.

2.3.2 Le corps des méthodes non statiques

Les méthodes non statiques peuvent consulter ou modifier l'état de l'objet courant. Celui ci n'est pas nommé dans la liste des arguments. Il faut donc un moyen de désigner l'objet courant dans le corps de la méthode.
C'est le role du mot clé `this`. Il fait référence à l'objet sur lequel on invoquera la méthode. A part cela, le corps des méthodes non statiques est du code Java usuel.

Par exemple dans la définition de `afficher` :

```
void afficher(){
  System.out.println("longeur"+ this.longeur);
}
```

`this.longeur` désigne la valeur de la variable d'instance longeur de l'objet sur lequel sera invoqué la méthode.

Lors de l'exécution de `c1.afficher()`, `this` désignera `c1`, alors que lors de l'exécution de `c2.afficher()` , `this` désignera `c2`.

En fait, lorsque cela n'est pas ambigu, on peut omettre `this` et écrire simplement le nom de la méthode sans préciser sur quel objet elle est appelée. Pour la méthode `afficher` cela donne :

```
void afficher(){
  System.out.println("longeur"+ longeur);
}
```

2.3.3 Invocation de méthodes avec arguments

Lorsqu'une méthode d'objet a des arguments, on l'invoque sur un objet en lui passant des valeurs pour chacun des arguments.

Voici un exemple d'invoquation de SetLongeur :

```
public class testDepot {
    public static void main (String [] arguments){

     Carre c1 = new Carre();
     c1.longeur =100;
     c1.afficher();
     c1.SetLongeur(800);
     c1.afficher();


    }
}
```

2.3.4 Lorsque les méthodes modifient l'état de l'objet

La méthode SetLongeur modifie l'état de l'objet courant. L'invocation de cette méthode sur un objet modifie donc l'état de cet objet. Dans notre exemple d'utilisation, le premier `c1.afficher()` affiche 100, alors que le second `c1.afficher()` affiche 800. Entre ces deux actions, l'exécution de `c1.SetLongeur(800)` a modifié l'état de `c1`.

2.3.5 Lorsque les méthodes retournent un résultat

Les méthodes non statiques peuvent evidemment retourner des valeurs. On pourrait par exemple modifier `deposer` pour qu'en plus de modifier l'état de l'objet, elle retourne le nouveau solde en résultat :

```
  class Carre {

     double   longeur;
     double  aire;

      void afficher(){
        System.out.println("longeur"+ this.longeur);
                }
      double GetLongeur(){
         return this.longeur;

                }
  }
```

La methode GetLongeur retourne la valeur de la longeur.

```
public static void main (String [] arguments){

      Carre c1 = new Carre();
      c1.longeur =100;
      System.out.println(c1.GetLongeur())
      ;

}
```

2.4 Les classes d'objets peuvent aussi avoir des méthodes statiques

Lorsqu'on définit une classe caractérisant un ensemble d'objets, on définit des variables d'instance et des méthodes non statiques. Ces méthodes définissent un comportement de l'objet courant.

Mais les classe peuvent aussi avoir des méthodes statiques.

Déclarer une méthode statique signifie que cette méthode n'agit pas, ni ne connaît en aucune manière, l'objet courant. Autrement dit, c'est une méthode qui n'agit que sur ses arguments et sur sa valeur de retour. Techniquement, cela signifie qu'une telle méthode ne peut jamais, dans son corps, faire référence à l'objet courant, ni aux valeurs de ses variables d'instance.

```
class Carre {

  double  longeur;
  double  aire;

    void afficher(){
      System.out.println("longeur"+ this.longeur);
                }
    double GetLongeur(){
       return this.longeur;

              }
    public static  void Bonjour(){
      System.out.println("bonjour");
                }

}
```

2.5 Les constructeurs.

Revenons un instant sur la création d'objets.
Que se passe-t-il lorsque nous écrivons : Carre `c1= new` Carre`()` ?
L'opérateur `new` réserve l'espace mémoire nécessaire pour l'objet `c1` et initialise les données avec des valeurs par défaut. Une variable d'instance de type `int` recevra 0, une de type `boolean` recevra `false`, une de type `char` recevra `'\0'` et les variables d'instances d'un type objet recevra `null`. En effet, lorsqu'on écrit Carre`()`, on appelle une sorte de mé-thode qui porte le même nom que la classe. Dans notre exemple, on voit que le constructeur Carre qui est appelé n'a pas d'arguments.

Un constructeur ressemble à une méthode qui porterai le même nom que la classe, mais ce n'est pas une méthode : la seule façon de l'invoquer consiste à employer le mot clé `new` suivi de son nom, c'est à dire du nom de la classe [1]. Ceci signifie qu'il s'exécute avant toute autre action sur l'objet, lors de la création de l'objet.

2.5.1 Le constructeur par défaut

La classe Carre ne contient pas explicitement de définition de constructeur. Et pourtant, nous pouvons, lors de la création d'un objet, y faire référence après l'opérateur `new`. Cela signifie que Java fournit pour chaque classe définie, un constructeur par défaut. Ce constructeur par défaut :

- a le même nom que la classe et
- n'a pas d'argument.

Le constructeur par défaut ne fait pratiquement rien. Voilà a quoi il pourrait ressembler :

2.5.2 Définir ses propres constructeurs

Le point intéressant avec les constructeurs est que nous pouvons les définir nous mêmes. Nous avons ainsi un moyen d'intervenir au milieu de `new`, d'intervenir lors de la création des objets, donc avant toute autre action sur l'objet.

Autrement dit, en écrivons nos propres constructeurs, nous avons le moyen, en tant que concepteur d'une classe, d'intervenir pour préparer l'objet à être utilisé, avant toute autre personne utilisatrice des objets.

attention : Dès que nous définissons un constructeur pour une classe, le constructeur par défaut n'existe plus.

Un constructeur se définit comme une méthode sauf que :

1. Le nom d'un constructeur est toujours celui de la classe.
2. Un constructeur n'a jamais de type de retour.

Dans une classe, on peut définir autant de constructeurs que l'on veut, du moment qu'ils se différencient par leur nombre (ou le type) d'arguments . Autrement dit, on peut surcharger les constructeurs.

```
class Carre {

  double  longeur;
  double  aire;
     public  Carre(){
          this.longeur=1.2;
     }
     void afficher(){
         System.out.println("longeur"+ this.longeur);
     }
     double GetLongeur(){

              return this.longeur;

     }
     public static  void Bonjour(){

              System.out.println("bonjour");

     }
}
```

2.6 Protection des données

Il est souvent souhaitable de protéger les données contenues dans les variables d'un objet contre des modifications incontrôlées. Par exemple, pour les comptes bancaires, le numéro de compte ne doit pas changer. Il faut le protéger contre une affectation opérée par erreur ou par malveillance. Seul le constructeur doit affecter une valeur à ce numéro.

Pour assurer cette protection des données, il existe la possibilité de déclarer les variables comme privées, c'est-à-dire qu'on ne peut les utiliser qu'à l'intérieur de la classe. On utilise pour cela le mot-clé `private`.

Pour permettre de connaître la valeur d'une variable privée sans permettre de la modifier, on peut écrire une méthode qui renvoie la valeur de la variable. Une telle méthode s'appelle un *accesseur*.

```
class Carre {

  private double  longeur;
  double aire;
     public  Carre(longeur){
          this.longeur=longeur;
     }
     void afficher(){
         System.out.println("longeur"+ this.longeur);
     }

   double GetLongeur(){

          return this.longeur;
   }

  double SetLongeur(longeur1){

        this.longeur=longeur1;

  }
  public static  void Bonjour(){

          System.out.println("bonjour");

     }
}
```

Si maintenant, dans une autre classe, on essaie de modifier le contenu d'une variable par une affectation, cela cause une erreur.

```
public class Autre{
    public static void main(String[] args){
        Carre c1 = new Carre(150);
        c1.solde = 100;


    }
}
```

```
> javac Autre.java
Autre.java:4: longeur has private access in Carre
        c1.longeur = 100;
```

On utilise le mot-clé `public` pour permettre l'utilisation d'une composante (variable ou méthode) par n'importe quelle classe. Ce mot-clé s'oppose à `private`. Si l'on ne précise rien (ni public, ni private), l'effet est intermédiaire : le composant est utilisable par certaines autres classes mais pas toutes.

En pratique, toute variable d'instance est normalement `private`. Les méthodes et constructeurs de la classes sont généralement déclarés `public`.

Une classe typique a donc la forme :

```
public class Personne {
        private String nom; // ne change pas une fois l'objet créé
        private String prenom; // ne change pas une fois l'objet créé
        private String adresse; // peut changer une fois l'objet créé

         public Personne(String nom, String prenom, String adresse) {
                this.nom= nom;
                this.prenom= prenom;
                this.adresse= adresse;
         }

         public String getNom() {
                return nom;
         }

         public String getPrenom() {
                return prenom;
         }

         public String getAdresse() {
                return adresse;
         }

         public void setAdresse(String adresse) {
                this.adresse= adresse;
         }
 }
```

2.7 Les variables statiques (ou de classe)

Nous savons déjà que les classes peuvent contenir

1. des variables d'instances
2. des constructeurs
3. des méthodes statiques ou non statiques.

Les classes peuvent contenir une quatrième sorte d'éléments : des variables statiques.

2.7.1 Déclaration de variables statiques

On les declarent en faisant précéder la déclaration usuelle du mot clé `static`. Par exemple :

```
static int a;
```

A titre d'exemple, ajoutons à la classe Carre une variable statique `nb` :

```
public class Carre {

   // ---Les variables d'instances ---
   double longeur;


   // ---La variable statique ---
   static int nb ;
   // ---le reste est inchange ___
```

```
-- jc amlqrpsarcsp
  public  Carre(longeur){

      this.longeur=longeur;
       nb++;    /* il compte le nombre de creation de carre */
      }
```

2.7.2 Rôle et comportement des variables statiques

Chaque objet à sa propre copie des variables d'instances. Elles représentent l'état personnel de l'objet. En revanche, il y a une seule copie des variables d'instances par classe. Tous les objets instances de la classe partagent la même copie. On peut accéder au contenu des variables d'instances par la classe ou par les objets instances de la classe. Et voici des exemples d'utilisation de la variable statique `nb` ajoutée à la classe Carre:

```
class public class TestStatic {

  public static void main (String [] arguments){


     Carre c1= new Carre(100);
     Carre c2 = new Carre(150);
     System.out.println(Carre.nb); // acces a nb par la classe Carre.
     System.out.println (Carre.nb);


    }

 }
```

Cet exemple nous montre bien que les variables statiques sont globales à tous les objets instances d'une classe. Ainsi, dans une classe, nous avons deux sortes d'éléments :

- les variables d'instances et les méthodes non statiques qui agissent sur les variables d'instances propres à chaque objet. C'est pourquoi ces méthodes sont aussi appelées méthodes *d'instances*. Chaque objet crée a sa propre copie des variables et méthodes d'instances.
- Les variables et méthodes statiques, dites aussi variables et méthodes *de classe*. Une seule et même copie de ces éléments est partagée par tous les objets de la classe.

Les variables de classe, comme `nb` sont bien sur accessibles dans la classe ou elles sont définies et en particulier dans les méthodes de la classe. Ce sont des variables globales à la classe.