Informatique - Java

Nicolas Rousset

Java - les classes

Java étant particulièrement orienté objet, les classes sont omniprésentes.
Ainsi même si l'on veut définir des fonctions qui ne sont pas rattachées à une classe particulière, on est obligée de créer une classe et de les définir comme static liés à cette classe.

Qu'est-ce qu'une classe ?

On peut définir une classe de deux façons :

- comment étant la représentation d'un objet modélisé
- comme étant un rassemblement de variables (les attributs) et des méthodes pour les traiter

Par exemple si l'on souhaite modéliser un avion, on pourra avoir des classes Passagers, Moteurs, Pilote, Aile, etc ...

Toutefois nous verrons que les classes ne se rattachent pas toujours à un objet réel que l'on souhaite modéliser.

Qu'est-ce qu'une classe ? (2)

Si l'on prend un exemple simple, une personne, on la représentera par une classe Personne. Celle-ci possèdera des attributs, par exemple :

- un nom sous forme de String
- un prénom sous forme de String
- un numéro de sécurité sociale sous forme de Long

On est donc obliger pour représenter un objet réel de déterminer l'ensemble des données qui le qualifie. On en aura donc une représentation limitée.

Les attributs d'une classe peuvent très bien être une autre classe.

Classe et instance de classe

Il faut distincter deux élements :

- la classe elle-même
- les instances de cette classes

Par exemple si l'on définit une classe Voiture, celle-ci représente le concept de général de voiture, et chaque voiture sera représentée par une instance.

Définition d'une classe

On définit une classe ainsi :

```
public class Personne
{
    private String prenom;
    private String nom;
    private int age;
}
```

Il s'agit d'une classe possédant 3 attributs, un prénom sous forme de champ texte, un nom sous forme de champ texte, un age.
Les attributs sont toujours **private**, sauf exception.

Définition d'une classe (2) - les modifiers

Lorsque l'on déclare une classe, on utilisera un certain nombre de qualificatifs sur les attributs et les méthodes. Les principaux sont :

Les modifiers d'accès :

- public : la méthode est accessible depuis n'importe quel endroit du code (normalement on ne l'utilise pas pour un attribut)
- protected : la méthode ou l'attribut est accessible à l'intérieur de la classe et des classes dérivées (voir plus loin)
- private : la méthode ou l'attribut est accessible uniquement à l'intérieur de la classe

Définition d'une classe (3) - les modifiers

Au delà des modifiers d'accès, il existe :

- final (attributs) : indique qu'une valeur ne peut être modifiée une fois initialisée. La valeur doit donc être initialisée dans le constructeur
- static (attributs ou méthode) : indique que l'attribut ou la méthode est rattachée à la classe. Pour un attribut il est donc commun à toutes les instances de la classe. Pour une méthode, cela signifie qu'elle ne peut accéder qu'à des méthodes et des attributs statiques.
- abstract : s'applique à la classe ou la méthode pour signifier qu'elle n'est pas définie. Dès lors qu'une classe est asbtraite (soit déclarée comme telle, soit parce qu'elle possède au moins une méthode abstraite) elle doit être dérivée et ne peut être implémentée directement

Définition d'une classe (4) - getters and setters

private => par défaut on n'expose pas les attributs (pas d'accès ou de modification directe)

Du coup on définit des méthodes associées pour accéder et modifier les valeurs. Un attribut peut n'avoir ni l'un ou l'autre.

```
public class Personne
{
    // Convention de nommage, les noms commencent par une minuscule
    private String prenom;
    private String nom;
    private int age;

    // convention de nommage, get + le nom de l'attribut avec une majuscule au début
    public String getNom()
    {
        return nom;
    }

    // même chose, set + le nom de l'attribut avec une majuscule au début
    public void setNom(String nom)
    {
        this.nom = nom;
    }
}
```

Définition d'une classe - les constructeurs

Le constructeur est, comme son nom l'indique, la méthode qui est appelée pour construire la classe, pour l'initialiser.
Il se déclare comme une fonction avec le même nom que la classe, et en omettant le type de retour :

```
public class Personne
{
    private String prenom;
    private String nom;
    private int age;

    public Personne()
    {
    }

    // même chose, set + le nom de l'attribut avec une majuscule au début
    public Personne(String prenom, String nom, int age)
    {
        // Ici le this sert à distinguer le paramêtre
        // prénom de l'attribut prénom
        this.prenom = prenom;
        this.nom = nom;
        this.age = age;
    }
}
```

Comme toute méthode, un constructeur peut être surchargé : il peut y avoir plusieurs constructeurs, forcément de même nom, pourvu que les attributs soient différents. Le compilateur choisira le bon.

On appelle le constructeur avec l'opérateur **new** :

```
// Appel du constructeur à 3 arguments
Personne personne = new Personne( "Jean", "Dupont", 18 );
// Appel du constructeur par défaut
Personne johnDoe = new Personne();
```

Note : Si il n'y a aucun constructeur, Java en créé un par défaut qui ne fait rien (tous les variables sont initialisées à leurs valeurs par défaut).
Dès lors qu'aun moins un constructeur est créé, il n'y a plus de constructeur créé, il faut le faire soi-même.

Mot clé this

Lorsque l'on est à l'intérieur d'une méthode non static (constructeur compris) on se réfère nécessairement à une instance de la classe.
Le mot clé **this** permet de désigner l'instance en cours de traitement.

Il est la plupart du temps implicite; lorsque l'on utilise le nom d'un attribut, le compilateur sait par défaut qu'il s'agit du nom de l'attribut en cours.
Il est donc optionnel la plupart du temps. Quelques cas où il ne l'est pas :

Désambiguation

Lorsque l'on a définit un paramètre de la fonction qui a le même nom qu'un attribut (ce qui est possible), le compilateur choisira la valeur la plus spécifique, ie l'argument. Il faut donc utiliser this pour dire que l'on désigne l'attribut, comme dans le constructeur ci-dessus :

```
public Personne(String prenom, String nom, int age)
{
    // Ici le this sert à distinguer le paramêtre
    // prénom de l'attribut prénom
    this.prenom = prenom;
    this.nom = nom;
    this.age = age;
}
```

Appel d'un constructeur au sein d'un autre

On peut, dans le code d'un constructeur appeler un autre constructeur de la même classe, avec le mot clé **this**.
Par exemple :

```
// On appelle l'autre coonstructeur en lui passant les
// valeurs par défaut définies ci-après
public Personne()
{
    this("John", "Doe", -1);
}

public Personne(String prenom, String nom, int age)
{
    // Ici le this sert à distinguer le paramêtre
    // prénom de l'attribut prénom
    this.prenom = prenom;
    this.nom = nom;
    this.age = age;
}
```

Attention ! Ne pas confondre avec le mot clé **super** qui sert lui à appeler le constructeur de la classe mère.

Passer l'objet courant en argument

On utilisera this lorsque l'on veut passer l'objet courant en tant qu'argument.
Par exemple, si l'on souhaite maintenir une liste de tous les instances de la classe :

```
private static List<Personne> listePersonnes = new ArrayList<Personne>();

public Personne()
{
    listePersonnes.add(this);
}
```

this désigne ici l'objet en cours de construction.

Héritage de classe

Une des propriétés les plus importantes des classes en Java et en programmation est la possibilité d'**héritage**.
On peut définir des classes qui hérite d'une autre.

Que cela signifie-t-il ?

Qu'on définit une nouvelle classe qui possède tous les attributs de l'ancienne, mais à laquelle on peut rajouter ou redéfinir des propriétés.

```
public class Chien
{
    private nbPattes = 4;
    private String nom;

    public Chien(String nom)
    {
        this.nom = nom;
    }

    public void aboyer()
    {
        System.out.println("Wouaf wouaf");
    }

    public abstract void afficherTaille();
}

public class Chihuahua extends Chien
{
    public Chihuahua(String nom)
    {
        super(nom);
    }

    public void afficherTaille()
    {
        System.out.println("Je suis petit");
    }
}

public class TerreNeuve extends Chien
{
    private int nbDePersonnesSauvees;

    public TerreNeuve(String nom)
    {
        super(nom);
    }

    public void afficherTaille()
    {
        System.out.println("Je suis très grand");
    }

    public void sauverDeLaNoyade()
    {
        nbDePersonnesSauvees++;
    }
}

public class Rottweiller extends Chien
{
    public Rottweiller(String nom)
    {
        super(nom);
    }

    public void afficherTaille()
    {
        System.out.println("Je suis grand");
    }

    public void aboyer()
    {
        System.out.println("WOUAF GRR WOUAFF GRR WOUAFF");
    }
}
```

Héritage de classe

Dans l'exemple précédent on a définit un concept général de Chien, et ensuite trois types de Chien particulier.
Tous les Chien possèdent les actions de base **afficherTaille()** et **aboyer()**.

Pour l'action **afficherTaille()** elle doit nécessairement être redéfinie dans une classe héritée.

Dans le cas du TerreNeuve on a défini un attribut et une méthode qui n'existe que pour lui.

```
private int nbDePersonnesSauvees;

public void sauverDeLaNoyade()
{
    nbDePersonnesSauvees++;
}
```

Cette action n'existe que pour le TerreNeuve, mais il possède également l'action Aboyer.

Pour le Rottweiller, on a redéfinit l'action de base aboyer.

Maintenant, le fait que ces 3 classes sont de type Chien, permet de leur appliquer des traitements communs :

```
List<Chien> chenil = new ArrayList();

chenil.add( new Chihuahua("Tobi") );
chenil.add( new TerreNeuve("Brontosaure") );
chenil.add( new Rottweiller("Bichon") );

for( Chien monChienQuiVaAboyer : chenil )
{
   monChienQuiVaAboyer.aboyer();
}
```