Séquence 12

Surcharge de méthodes et de constructeurs

Dans une classe on peut avoir plusieurs méthodes différentes avec le même nom. Ces méthodes sont différentes parce qu'elles ne comportent pas les mêmes instructions. Pour pouvoir savoir laquelle des méthodes qui portent le même nom est appelée lors d'une invocation de méthode, il faut que les paramètres soient différents par le type et/ou par le nombre de telle sorte qu'il n'y a pas de doute possible : une seule méthode correspond aux valeurs données entre parenthèse dans l'invocation de méthode.

1 Un exemple de méthode surchargée : System.out.println

Pour réaliser un affichage, on utilise la méthode `System.out.println`. La méthode, vraiment ? Non, pas la méthode, **les** méthodes. Il y en a plusieurs, ce que nous allons voir dans la documentation de la classe.

`System` est une classe du paquetage `java.lang`. Dans cette classe, `out` est une variable statique et le type de cette variable est `java.io.PrintStream`. Allons consulter la documentation de cette classe pour y trouver les méthodes `println`.

— `void println()`
Terminates the current line by writing the line separator string.
— `void println(boolean x)`
Prints a boolean and then terminate the line.
— `void println(char x)`
Prints a character and then terminate the line.
— `void println(char[] x)`
Prints an array of characters and then terminate the line.
— `void println(double x)`
Prints a double and then terminate the line.
— `void println(float x)`
Prints a float and then terminate the line.
— `void println(int x)`
Prints an integer and then terminate the line.
— `void println(long x)`
Prints a long and then terminate the line.

— `void println(Object x)`
  Prints an Object and then terminate the line.
— `void println(String x)`
  Prints a String and then terminate the line.

On voit qu'il y a 10 méthodes différentes : une sans paramètres qui permet simplement de passer à la ligne et 9 qui prennent un paramètre mais qui ont toutes un type différent pour ce paramètre. Lors de l'invocation, le type de la valeur donnée entre parenthèse permet de choisir quelle méthode doit être exécutée. Ce choix est fait lors de la compilation.

Par exemple, dans l'invocation `System.out.println(12.5);`, la valeur entre parenthèse est de type `double`. C'est donc la méthode `println` qui prend un double en paramètre qui est appelée.

2 Exemple de surcharge de constructeurs

Une classe peut comporter plusieurs constructeurs à condition que leurs paramètres soient différents par leur nombre et/ou par leurs types. Prenons l'exemple de la classe `java.util.Date`. Suit un extrait de la documentation.

— `Date()`
  Allocates a Date object and initializes it so that it represents the time at which it was allocated, measured to the nearest millisecond.
— `Date(int year, int month, int date)`
  Deprecated. As of JDK version 1.1, replaced by Calendar.set(year + 1900, month, date) or GregorianCalendar(year + 1900, month, date).
— `Date(int year, int month, int date, int hrs, int min)`
  Deprecated. As of JDK version 1.1, replaced by Calendar.set(year + 1900, month, date, hrs, min) or GregorianCalendar(year + 1900, month, date, hrs, min).
— `Date(int year, int month, int date, int hrs, int min, int sec)`
  Deprecated. As of JDK version 1.1, replaced by Calendar.set(year + 1900, month, date, hrs, min, sec) or GregorianCalendar(year + 1900, month, date, hrs, min, sec).
— `Date(long date)`
  Allocates a Date object and initializes it to represent the specified number of milliseconds since the standard base time known as "the epoch", namely January 1, 1970, 00 :00 :00 GMT.
— `Date(String s)`
  Deprecated. As of JDK version 1.1, replaced by DateFormat.parse(String s).

On voit qu'il y a 6 constructeurs différents dont 4 sont **Deprecated.** , c'est-à-dire obsolètes et qu'il ne faut plus utiliser.

Le premier n'a pas de paramètre. C'est pour construire un objet représentant la date et l'heure du moment où le constructeur est appelé. Le second prend en paramètres trois entiers pour l'année, le mois, le jour. Le troisième prend 5 entiers en paramètres : l'année, le mois, le jour, les heures et les minutes. Le suivant prend un entier de plus pour les secondes. Le cinquième, qui n'est pas obsolète, prend en paramètre une valeur de type `long` (nombre entier sur 8 octets). C'est le nombre de millisecondes écoulées depuis le premier janvier 1970 à 0 heures.

Cette référence au premier janvier 1970 peut sembler exotique. Elle se justifie par le fait que cette date est souvent prise comme origine dans le décompte du temps dans les ordinateurs, notamment dans les systèmes d'exploitation Unix (Linux, MacOs). Noter que le nombre donné en paramètre peut être négatif pour des dates antérieures au premier janvier 1970.

Le sixième et dernier constructeur prend en paramètre une chaîne de caractère décrivant la date en anglais. Par exemple `"February 18, 1928"`. Celui-ci, comme ceux qui prennent des entiers, est obsolète.

Lors de l'appel au constructeur avec l'instruction new, il est facile de déterminer quel constructeur doit être utilisé selon le type et le nombre de valeurs données entre parenthèses.

Quelques exemples :

— `new Date("February 18, 1928")` invocation avec un paramètre de type `String`, c'est le sixième constructeur qui est appelé.

— `new Date(1955-1900,12-1,12,12,12)` invocation avec 5 paramètres de type `int` : c'est le troisiéme constructeur qui est appelé.

3 Écrire des méthodes et cosntructeurs surchargés

Pour utiliser la surcharge dans les classes que nous créons rien du plus simple : il suffit de mettre plusieurs constructeurs et/ou plusieurs méthodes ayant le même nom avec des paramètres différents.

Prenons comme exemple une classe représentant un panier de produits. Chaque produit a un nom, une référence, une quantité. Voici la classe des produits.

```
public class Produit{
    private String nom;
    private int reference;
    private int quantite;
    public Produit(String n, int r, int q){
        nom=n;
        reference=r;
        quantite=q;
    }
    public String getNom(){
        return nom;
    }
    public int getReference(){
        return reference;
    }
    public int getQuantite(){
        return quantite;
    }
    public void setQuantite(int q){
        quantite = q;
    }
    public String toString(){
        return nom + "␣ref:␣" + reference + "␣quantite:␣" + quantite;
    }
}
```

Nous allons écrire une classe des paniers de produits (ensemble de plusieurs produits). Pour avoir plusieurs produits dans un panier, nous allons utiliser un `ArrayList` de produits. Nous allons surcharger le constructeur pour permettre d'initialiser le panier avec 0, 1 ou plusieurs produits.

Nous aurons une méthode ajouter permettant d'ajouter des produits au panier. Nous la surchargerons pour ajouter un seul ou plusieurs produits.

Nous aurons une méthode pour retirer des produits en les identifiant par leur nom ou leur référence. Nous donnerons aussi la possibilité de retirer une partie des produits en indiquant quelle quantité il faut retirer sur un produit présent dans le panier.

```
mport java.util.ArrayList;
public class Panier{
    private ArrayList<Produit> produits;
    public Panier(){
        produits = new ArrayList<Produit>();
    }
    public Panier(Produit prod){
      produits = new ArrayList<Produit>();
      produits.add(prod);
    }
    public Panier(Produit[] tab){
        produits = new ArrayList<Produit>();
        for (int i=0; i<tab.length; i++){
            produits.add(tab[i]);
        }
    }
    public void ajouter(Produit prod){
        produits.add(prod);
    }
    public void ajouter(Produit[] tab){
        for (int i=0; i<tab.length; i++){
            produits.add(tab[i]);
        }
    }
    private int indexOf(String nom){
        int idx = 0;
        while(idx<produits.size() && ! produits.get(idx).getNom().equals(nom)){
            idx = idx + 1;
        }
        if (idx<produits.size())
            return idx;
        else
            throw new IllegalArgumentException("Produit␣" + nom + "␣non␣trouve");
    }
     private int indexOf(int ref){
        int idx = 0;
        while(idx<produits.size() && produits.get(idx).getReference()!=ref){
            idx = idx + 1;
        }
        if (idx<produits.size())
            return idx;
        else
            throw new IllegalArgumentException("Produit␣" + ref + "␣non␣trouve");
    }
    public void retirer(String nom){
        int idx = indexOf(nom);
        produits.remove(idx);
    }
    public void retirer(String nom, int quant){
```

```
        int idx = indexOf(nom);
        Produit prod = produits.get(idx);
        int qte = prod.getQuantite();
        if (qte > quant)
            prod.setQuantite(qte-quant);
        else
            throw new IllegalStateException("Pas␣assez␣de␣produits␣" + nom
                                            + "␣pour␣en␣retirer␣" + quant);
    }
    public void retirer(int ref){
        int idx = indexOf(ref);
        produits.remove(idx);
    }
    public void retirer(int ref, int quant){
        int idx = indexOf(ref);
        Produit prod = produits.get(idx);
        int qte = prod.getQuantite();
        if (qte > quant)
            prod.setQuantite(qte-quant);
        else
            throw new IllegalStateException("Pas␣assez␣de␣produits␣" + ref
                                            + "␣pour␣en␣retirer␣" + quant);
    }
}
```

4 Compléments sur les constructeurs

4.1 Constructeur par défaut

Si l'on ne définit pas un constructeur dans une classe, il en existe quand même un qui est ajouté implicitement par Java dans chaque classe qui ne comporte pas de constructeur. Ce constructeur par défaut n'a pas de paramètre et il appelle le constructeur sans paramètre de la super-classe.

Dans une classe qui s'appelle `ExempleConstructeur`, le constructeur par défaut aura le code suivant :

```
  public ExempleConstructeur(){
    super();
  }
```

Ce constructeur par défaut peut créer une erreur de compilation dans le cas où la super-classe ne comporte pas de constructeur sans paramètre. L'appel `super();` est alors incorrect. Dans ce cas, on est obligé d'écrire un constructeur dans la sous-classe.

Ci-dessous un code qui provoque une erreur de ce type et le message obtenu à la compilation.

```
public class Entier{
    protected int x;
    public Entier(int xx){
        x = xx;
    }
}
public class EntierSigne extends Entier{
```

```
    protected boolean positif;
}
```

```
> javac EntierSigne.java
EntierSigne.java:1: error: constructor Entier in class Entier cannot
                                              be applied to given types;
public class EntierSigne extends Entier{
       ^
  required: int
  found: no arguments
  reason: actual and formal argument lists differ in length
1 error
```

4.2 Appel à un autre constructeur

La première ligne d'un constructeur doit contenir un appel à un constructeur de la super-classe utilisant le mot-clé `super`. Si cet appel n'apparaît pas, implicitement, le langage considère qu'il y un appel au constructeur sans paramètre de la super-classe. S'il n'y a pas de constructeur sans paramètre dans la super-classe, cela provoque une erreur.

Le code suivant montre à l'exécution que le constructeur de la super-classe est exécuté même si l'appel à `super` n'apparaît pas explicitement.

```
public class SuperClasse{
    public SuperClasse(){
        System.out.println("Constructeur␣sans␣parametre␣de␣SuperClasse");
    }
}
public class SousClasse extends SuperClasse{
    public SousClasse(int x){
        System.out.println("Constructeur␣de␣SousClasse");
    }
    public static void main(String[] args){
        SousClasse objet = new SousClasse(5);
    }
}
```

A l'exécution :

```
> java SousClasse
Constructeur sans parametre de SuperClasse
Constructeur de SousClasse
```

Bien qu'il n'y ait pas d'appel avec `super` dans le constructeur de `SousClasse`, l'appel a lieu et le constructeur de `SuperClasse` est exécuté.

4.3 Alternative à super

La première instruction d'un constructeur peut être un appel à un constructeur de la super-classe utilisant le mot-clé `super`, mais cela peut également être un appel à un autre constructeur de la sous-classe. Dans ce cas, le mot-clé utilisé est `this`.

```
public class AutreSousClasse extends SuperClasse{
    protected int xx, yy;
    public AutreSousClasse(int x){
        super();
        xx = x;
        System.out.println("constructeur␣AutreSousClasse(int␣x)");
    }
    public AutreSousClasse(int x,int y){
        this(x);
        System.out.println("constructeur␣AutreSousClasse(int␣x,int␣y)");
    }
    public static void main(String[] args){
        AutreSousClasse objet1 = new AutreSousClasse(5);
        System.out.println("======");
        objet1 = new AutreSousClasse(5,10);
    }
}
```

L'exécution du code donne l'affichage suivant.

```
> java AutreSousClasse
Constructeur sans parametre de SuperClasse
constructeur AutreSousClasse(int x)
======
Constructeur sans parametre de SuperClasse
constructeur AutreSousClasse(int x)
constructeur AutreSousClasse(int x,int y)
```

On voit que l'exécution du second constructeur, au moyen de l'appel à `this`, appelle le premier constructeur et celui-ci appelle le constructeur de la super-classe. Celui-ci appelle à sont tour le constructeur sans paramètre de `Object` qui est la super-classe de `SuperClasse`. Cet appel au constructeur de la classe `Object` ne se voit pas parce que celui-ci ne fait pas d'affichage.

Dans la construction d'un objet par un constructeur, il y a un processus qui commence avec l'exécution d'un constructeur de la classe `Object`, puis un constructeur d'une de ses sous-classes, etc, jusqu'à la classe instanciée par le `new`.

Nousa vons vu qu'un objet a plusieurs types situés sur un chemin de l'arbre d'héritage qui part de la racine `Object` et va jusqu'à la classe instanciée. Un constructeur de chaque type est exécutée à la création de l'objet en commençant par ceux qui sont en haut de l'arbre d'héritage.