Surcharge et documentation

Arnaud Labourel arnaud.labourel@univ-amu.fr

2 mars 2022

Section 1

Surcharge de méthode/constructeurs

Méthodes ayant le même nom

Dans une classe, plusieurs méthodes peuvent avoir le même nom.

C'est ce qu'on appelle la surcharge de méthode.

Il est par contre nécessaire que la séquence dans l'ordre des types des arguments soit différente pour chaque méthode ayant le même nom.

La méthode est choisie par le compilateur de la façon suivante :

- Le nombre de paramètres doit correspondre
- Les affectations des paramètres doivent être valides

Exemple de surcharge de méthode

```
public class DataArtist {

    public void draw(String s) {
        /* ... */
    }
    public void draw(int i) {
        /* ... */
    }
    public void draw(double f) {
        /* ... */
    }
    public void draw(int i, double f) {
        /* ... */
    }
}
```

Exemple de surcharge de méthode

```
class Adder {
  public static int add(int intVal1, int intVal2) {
    System.out.println("integer");
    return intVal1 + intVal2;
  }

  public static double add(double doubleVal1,
                           double doubleVal2) {
    System.out.println("double");
    return doubleVal1 + doubleVal2;
  }
}
```

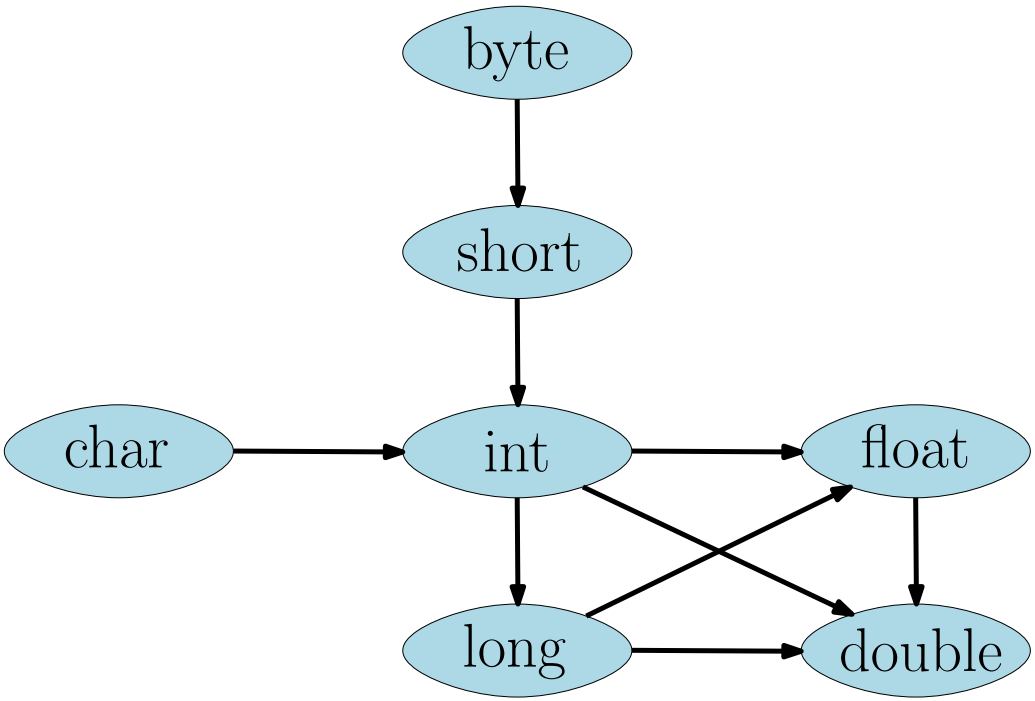
Exemple de surcharge de méthode

```
int intValue = 1;
double doubleValue = 2.2;
double result1 = Adder.add(doubleValue, doubleValue);
// → double

int result3 = Adder.add(intValue, intValue);
// → int
```

Promotion pour les types primitifs

Si les types des arguments ne correspondent pas aux types des paramètres, les types des arguments sont promus en suivant les flèches:




```
double result2 = Adder.add(intValue, doubleValue);
// → double
```

Plusieurs constructeurs

C'est exactement les mêmes règles qui s'appliquent pour les constructeurs d'une classe.

```
public class Point{
    public int x, y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public Point() {
        this(0, 0);
    // this appelle le constructeur de la classe.
    }
    public Point(Point p) {
        this(p.x, p.y);
    }
```

Section 2

Commentaires et documentation

Commentaires inutiles

```
String get(String[] source, int index) {
    // Teste si l'index est dans les limites du tableau.
    if (index < 0 || index >= source.length)
        return null;
    return source[index];
}
```

Si un commentaire semble nécessaire, le remplacer par une méthode :

```
boolean indexIsInBounds(String[] source, int index) {
    return index >= 0 && index < source.length;
}
String get(String[] source, int index) {
    if (!indexIsInBounds(source, index)) return null;
    return source[index];
}
```

Commentaires inutiles

Les commentaires se désynchronisent souvent du code :

```
/* doit toujours retourner true. */
boolean isAvailable() {
    return true;
}
```

risque de devenir un jour :

```
/* doit toujours retourner true. */
boolean isAvailable() {
    return false;
}
```

Commentaires inutiles = répétition

Commentaires inutiles

Des commentaires qui peuvent sembler utiles :

```
/* une méthode qui retourne que les carrés : */
List<Rectangle> get(List<Rectangle> list) {
    /* le résultat sera stocké dans cette liste : */
    List<Rectangle> list2 = new ArrayList<Rectangle>();
    for (Rectangle x : list)
        if (x.w == x.h /* un carré ? */)
            list2.add(x);
    return list2;
}
class Rectangle {
    public int w; /* largeur */
    public int h; /* hauteur */
}
```

Commentaires inutiles

On peut se passer de commentaire en rajoutant une méthode et en nommant correctement les éléments du code.

```
List<Rectangle> findSquares(List<Rectangle> rectangles) {
    List<Rectangle> squares = new ArrayList<Rectangle>();
    for (Rectangle rectangle : rectangles)
        if (rectangle.isSquare())
            squares.add(rectangle);
    return squares;
}
class Rectangle {
    private int width, height;
    boolean isSquare() {
        return width == height;
    }
}
```

Commentaires **utiles**

Des commentaires pour décrire les tâches à réaliser peuvent être utiles

```
void processElement(Stack<Formula> stack,
                    String element) {
    // TODO : prendre en compte les signes '-' et '/'
    switch (element) {
        case "+": processSum(stack); break;
        case "*": processProduct(stack); break;
        default : processInteger(stack, element);
        break;
    }
}
```

Commentaires **utiles**

Documentation ou spécification du comportement d'une méthode :

```
/**
 * Returns <tt>true</tt> if this list contains the
 * specified element. More formally, returns
 * <tt>true</tt> if and only if this list contains
 * at least one element <tt>e</tt> such that
 * <tt>(o==null ? e==null : o.equals(e))</tt>.
 *
 * @param o element whose presence in this list is
 * to be tested
 * @return <tt>true</tt> if this list
 * contains the specified element
 */
public boolean contains(Object o) {
    return indexOf(o) >= 0;
```

JavaDoc

JavaDoc permet de générer automatiquement une documentation du code à partir de commentaires associés aux classes, méthodes, propriétés, ...

La documentation contient :

- Une description des membres : attributs et méthodes (publics et protégés) des classes
- Une description des classes, interfaces, ...
- Des liens permettant de naviguer entre les classes
- Des informations sur les implémentations et extensions

Un bloc de commentaire Java commençant par /** deviendra un bloc de commentaire Javadoc qui sera inclus dans la documentation du code source.

JavaDoc

Tag	Description
@author	pour préciser l'auteur de la fonctionnalité
@deprecated	indique que l'attribut, la méthode ou la classe est dépréciée
@return	pour décrire la valeur de retour
{@code `literal`}	Formate `literal` en code
{@link `reference`}	permet de créer un lien

Pour générer la javadoc en IntelliJ

Tools → generate Javadoc

JavaDoc

```
/**
 * The {@code Byte} class wraps a value of primitive
 * type {@code byte} in an object.  An object of type
 * {@code Byte} contains a single field whose type is
 * {@code byte}.
 *
 * <p>In addition, this class provides several methods
 * for converting a {@code byte} to a {@code String}
 * and a {@code String} to a {@code byte}, as well as
 * other constants and methods useful when dealing
 * with a {@code byte}.
 *
 * @author  Nakul Saraiya
 * @author  Joseph D. Darcy
 */
```