Programmation 2 : Sixième cours

Arnaud Labourel arnaud.labourel@univ-amu.fr

4 novembre 2019

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
 */
```

Structure d'un projet et paquets

Structure d'un projet

En Java, un projet peut être découpé en paquets (`package`).

Les paquets permettent de :

- associer des classes afin de mieux organiser le code
- de créer des modules indépendants réutilisables
- d'avoir plusieurs classes qui possèdent le même nom (du moment qu'elles ne sont pas dans le même paquet)

Un paquet (`package`) :

- est une collection de classes
- peut contenir des sous-paquets

Lors de l'exécution...

Java utilise l'arborescence de fichier pour retrouver les fichiers `.class`

- Une classe (ou une interface) correspond à un fichier `.class`
- Un dossier correspond à un paquet

Les `.class` du paquet `com.univ_amu` doivent :

- être dans le sous-dossier com/univ_amu
- le dossier com doit être à la racine d'un des dossiers du `ClassPath`

Le `ClassPath` inclut :

- le répertoire courant
- les dossiers de la variable d'environnement CLASSPATH
- des archives JAR
- des dossiers précisés sur le ligne de commande de java (`-classpath path`)

Lors de la compilation... (1/2)

Le mot-clé `package` permet de préciser le paquet des classes ou interfaces définies dans le fichier :

```
package com.univ_amu;
public class MyClass { /* ... */ }
```

Java utilise l'arborescence pour retrouver le code des classes ou interfaces :

- Une classe (ou une interface) `MyClass` est cherchée dans le fichier `MyClass.java`
- Le fichier `MyClass.java` est cherché dans le dossier associé au paquet de `MyClass`

Lors de la compilation... (2/2)

```
package com.univ_amu;
public class MyClass { /* ... */ }
```

Dans l'exemple précédent, il est donc conseillé que le fichier :

- se nomme `MyClass.java`
- se trouve dans le dossier `com/univ_amu` (Par défaut, la compilation crée `MyClass.class` dans `com/univ_amu`)

Utilisation d'une classe à partir d'un autre paquet

Accessibilité :

- Seules les classes publiques sont utilisable à partir d'un autre paquet
- Un fichier ne peut contenir qu'une seule classe publique

On peut désigner une classe qui se trouve dans un autre paquet :

```
package com.my_project;
    public class Main {
    public static void main(String[] args) {
        com.univ_amu.MyClass myClass =
            new com.univ_amu.MyClass();
    }
}
```

Importer une classe

Vous pouvez également importer une classe :

```
package com.my_project;
import com.univ_amu.MyClass;

public class Main {
    public static void main(String[] args) {
        MyClass myClass = new MyClass();
    }
}
```

Deux classes de deux paquets différents peuvent avoir le même nom :

- Exemple : `java.util.List` et `java.awt.List`
- Attention de choisir le bon import

Importer un paquet

Vous pouvez également importer toutes les classes d'un paquet :

```
package com.my_project;
import com.univ_amu.*;

public class Main {
    public static void main(String[] args) {
        MyClass myClass = new MyClass();
    }
}
```

Remarques :

- Les classes des sous-paquets ne sont pas importées
- Le paquet `java.lang` est importé par défaut

Importer des membres statiques

Depuis Java 5, il est possible d'importer directement des méthodes ou attributs de classes (`static`).

La syntaxe est la suivante :

```
import static my_package.my_class.my_static_member;
```

Exemple :

```
import static java.lang.Math.PI;
import static java.lang.Math.pow;
```

Exemple sans import statique

```
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello World!");
    System.out.println(
      "A circle with a diameter of 5 cm has");
    System.out.println("a circumference of "
                       + (Math.PI * 5) + " cm");
    System.out.println("and an area of "
            + (Math.PI * Math.pow(2.5, 2))
            + " sq. cm");
  }
}
```

Exemple d'import statique

```
import static java.lang.Math.PI;
import static java.lang.Math.pow;
import static java.lang.System.out;

public class HelloWorld {
    public static void main(String[] args) {
        out.println("Hello World!");
        out.println(
            "A circle with a diameter of 5 cm has");
        out.println("a circumference of "
                + (PI * 5) + " cm");
        out.println("and an area of "
            + (PI * pow(2.5, 2)) + " sq. cm");
    }
}
```

Un exemple

Le fichier com/univ_amu/HelloWorld.java :

```
package com.univ_amu;
    public class HelloWorld {
    public static void main(String[] arg) {
        System.out.println("Hello world ! ");
    }
}
```

```
$ javac com/univ_amu/HelloWorld.java
$ ls com/univ_amu
HelloWorld.java HelloWorld.class
$ java com.univ_amu.HelloWorld
Hello world !
```

Quelques remarques

Nommage des paquets :

- Les noms de paquets sont écrits en minuscules
- Pour éviter les collisions, on utilise le nom du domaine à l'envers ⇒ `com.google.gson`, `com.oracle.jdbc`
- Si le nom n'est pas valide, on utilise des underscores : ⇒ `com.univ_amu`

Fichier JAR (Java Archive) :

- est une archive ZIP pour distribuer un ensemble de classes Java
- contient un *manifest* (qui peut préciser la classe qui contient le main)
- peut également faire partie du `ClassPath`
- peut être généré en ligne de commande (`jar`) ou avec un IDE

Classe interne

Classe imbriquée statique

Il est possible de définir une classe à l'intérieur d'une autre (classe imbriquée ou *nested class*) :

```
public class LinkedList {
    public static class Node {
        private String data; private Node next;
        public Node(String data, Node next) {
            this.data = data; this.next = next;
        }
    }
}
```

Il est possible d'instancier la classe interne sans qu'une instance de `LinkedList` existe car elle est statique :

```
LinkedList.Node node = new LinkedList.Node("a", null);
```

Classe imbriquée statique

Rappel

Une classe non-imbriquée publique (`public`) doit être dans un fichier portant son nom.

Interdit !

Fichier `LinkedList.java`

```
public class LinkedList {/*...*/}
public class Node {/*...*/}
```

$\Rightarrow$ erreur à la compilation :

```
Error:(9, 8) java: class Node is public, should be
  declared in a file named Node.java
```

Remarque

Une classe imbriquée peut être publique et accessible depuis l'extérieur.

Classe imbriquée statique

Il est également possible de la rendre privée à la classe `LinkedList` :

```
public class LinkedList {
    private static class Node {
        private String data;
        private /*LinkedList.*/Node next;
        public Node(String data, Node next) {
            this.data = data;
            this.next = next;
        }
    }
}
```

Dans ce cas, seules les méthodes de `LinkedList` pourront l'utiliser. Notez que des méthodes statiques définies dans `LinkedList` peuvent également utiliser cette classe interne du fait qu'elle soit statique.

Classe imbriquée statique

Exemple d'implémentation de méthodes dans la classe `LinkedList` :

```
public class LinkedList {
    /* Code de la classe interne statique Node. */
    private Node first = null;
    public void add(String data) {
        first = new Node(data, first);
    }
    public void print() {
        Node node = first;
        while (node!=null) {
            System.out.print("["+node.data+"]");
            node = node.next;
        }
    }
}
```

Classe imbriquée statique

Exemple d'utilisation de la classe précédente :

```
LinkedList list = new LinkedList();
list.add("a");
list.add("b");
list.add("c");
list.print();
```

Cet exemple produit la sortie suivante :

```
[c] [b] [a]
```

Classe imbriquée statique

Une classe imbriquée statique ne peut accéder qu'aux attributs et méthodes statiques de la classe qui la contient :

```
public class LinkedList {
    private static class Node {
        /* Champs et méthodes de Node. */
        boolean isFirst() {
            return this==first; // interdit !
        }
    }
    private Node first;
    /* Autres champs et méthodes de LinkedList. */
}
```

Classe interne

En revanche, si la classe interne n'est pas statique, elle peut accéder aux champs de classe qui la contient :

```
public class LinkedList {
    private class Node {
        /* Champs et méthodes de Node. */
        private boolean isFirst() {
            return this==first; // autorisé !
        }
    }
    private Node first;
    /* Autres champs et méthodes de LinkedList. */
}
```

Classe interne

Java insère dans l'instance de `Node` une référence vers l'instance de `LinkedList` qui a permis de la créer :

```
public class LinkedList {
  private class Node {
   /* Champs et méthodes de Node. */
    private boolean isFirst() {
      return this==/*référenceVersLinkedList.*/first;
    }
  }
  public void add(String data) {
    first = new Node(data, first);
  }
  /* Autres champs et méthodes de la classe LinkedList.
}
```

Classe interne

Il est possible d'utiliser la méthode `isFirst` dans `LinkedList` :

```
public class LinkedList {
  /* Code de la classe interne statique Node
     et champs et méthodes de la classe LinkedList. */
  public void print() {
    Node node = first;
    while (node!=null) {
      System.out.print("["+node.data
          +","+node.isFirst()+"]");
      node = node.next;
    }
  }
}
```

Classe interne

Exemple d'utilisation de la classe précédente :

```
LinkedList list = new LinkedList(); list.add("a");
list.add("b");
list.add("c");
list.print();
```

Cet exemple produit la sortie suivante :

```
[c,true][b,false][a,false]
```