

Programmation Orientée Objets avec Java

Prof. Yassin Rekik et Prof. Stéphane Malandain – 2023-2024

(Basé sur le cours de Joël Cavat)



Chapitre 6

Les types imbriqués

Concepts traités

2

- Les classes imbriqués
- Les classes anonymes
- Enumérations avancées

Les types imbriqués

Les types imbriqués

4

- Un type imbriqué est généralement appelé `inner class` ou `nested class`
- Il s'agit de la déclaration d'un type dans un autre
- Plusieurs types d'imbrication :
 - Les classes imbriquées statiques ou non statiques
 - Les classes anonymes
- Les interfaces fonctionnelles seront abordées ultérieurement dans un cours dédié.

Imbrication simple

5

- Les classes imbriquées (`nested class` ou `inner class`) mettent à disposition un mécanisme qui permet de déclarer des classes (ou des interfaces) dans une autre classe ou dans une autre interface.
- Cela permet de réduire la complexité de manière locale
- Cela permet également de retourner des objets qui n'existeraient pas sans leur objet englobant.

Les types imbriqués

6

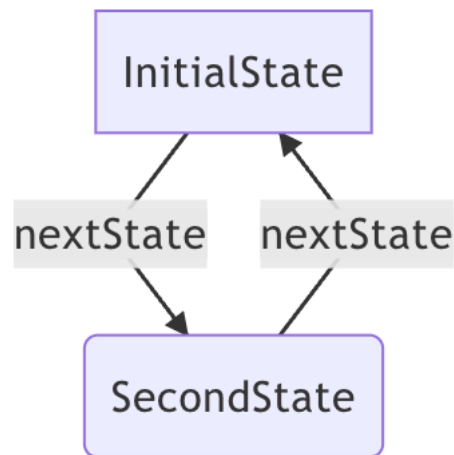
```
1  class OuterClass {  
2      ...  
3      static class StaticNestedClass {  
4          ...  
5      }  
6      class InnerClass {  
7          ...  
8      }  
9  }
```

- Une classe interne peut être statique si un objet ne possède aucune référence de sa classe englobante
- Un objet d'une classe qui n'est pas statique est rattaché à une instance existante de la classe englobante

Exemple d'utilisation

7

- Nous voulons réaliser un automate à états fini (*finite state machine*) composé de deux états tout en ayant une référence qui puisse pointer sur un des deux.
- Un objet state peut être de 2 types : `InitialState` ou `SecondState`
- En BNF cela donne : `State ::= InitialState | SecondState`



Exemple d'utilisation

8

```
1  // Mise en oeuvre d'une machine d'états finis
2  public interface State {
3      static class InitialState implements State {
4          public boolean isRunning() { return false; }
5          public State nextState() { return new SecondState(); }
6      }
7      static class SecondState implements State {
8          public boolean isRunning() { return true; }
9          public State nextState() { return new InitialState(); }
10     }
11     default boolean isRunning() { return true; }
12     public static State initial() { return new InitialState(); }
13     State nextState();
14 }
```

InitialState et SecondState sont indépendants d'une quelconque instance englobante. Ils sont donc statiques

Quiz

9

Que retourne ces deux expressions ?

```
1  State.initial().nextState().isRunning();  
2  State.initial().nextState().nextState().isRunning();
```

Réponse

10

```
1  State.initial().nextState().isRunning(); // => true
2  State.initial().nextState().nextState().isRunning(); // => false
```

Explications :

```
1  State.initial() // ==> objet de type: InitialState
2  |   |   |   |   // InitialState redéfinit nextState et retourne un SecondState
3  State.initial().nextState() // ==> objet de type: SecondState
4  State.initial().nextState().isRunning(); // SecondState redéfinit isRunning et retourne true
5  State.initial().nextState().nextState(); // SecondState redéfinit nextState et retourne un InitialState
6  State.initial().nextState().nextState().isRunning(); //==> InitialState redéfinit isRunning en retournant false
```

Map.Entry (statique)

- Les interfaces peuvent également être imbriquées.
- `Entry<K, V>` est une interface statique interne à l'interface `Map<K, V>`
- Représente un tableau associatif (couple clé-valeur)
- L'interface `Map` fournit une méthode statique qui retourne un objet d'une classe respectant l'interface `Entry`.
- L'objet retournée est immuable

```
1    Map.Entry<Integer,String> tuple = ...
```

Exemples connus

12

Map.Entry (statique)

```
1  Map.Entry<Integer, String> oneEntry = Map.entry(1, "one");
2  int key = oneEntry.getKey(); // key = 1
3  String value = oneEntry.getValue(); // value = "one"
```

- Un import `static` permet de simplifier la syntaxe de la méthode `entry` pour peupler un tableau associatif sans devoir préciser le type englobant :

```
1  import static java.util.Map.entry;
2  import static java.util.Map.ofEntries;
3
4  Map<Integer, String> map = ofEntries( // au lieu de Map.ofEntries()
5      entry(1, "one"), // au lieu de Map.entry()
6      entry(2, "two")
7  )
```

Exemples connus

13

Iterator (non statique)

L'`iterator` est un exemple de classe interne non statique. Elle doit référencer une collection existante

Exemple d'un itérateur :

```
1  Iterator<String> it = List.of("Hello", "world").iterator();
2  // it possède une référence sur la liste List.of("Hello", "world")
3
4  while (it.hasNext()) {
5      String oneString = it.next();
6      /* do something with oneString */
7  }
8
```

Exemples connus

14

Iterator (non statique)

```
1 public class ArrayList<E> implements List<E> ... {
2     public Iterator<E> iterator() { return new Itr(); }
3     ...
4     private class Itr implements Iterator<E> {
5         ...
6         // a accès à tous les membres de l'objet `ArrayList`
7     } }
8
```

Classes anonymes

15

- Une classe anonyme est une manière de déclarer une classe et d'instancier un objet de celle-ci en même temps.
- Très utile si nous souhaitons instancier qu'un seul objet d'une classe sans avoir à déclarer la classe dans un fichier

- Soit l'interface suivante :

```
1 public interface Hello {  
2     void greetings();  
3 }
```

- Nous pouvons déclarer une classe qui respecte cette interface et créer directement un objet de cette manière :

```
1 Hello h = new Hello() {  
2     public void greetings() { System.out.println("Hi guys!"); }  
3 };  
4 h.greetings(); // Affiche Hi guys!
```

Classes anonymes

16

- Une classe anonyme est une manière de déclarer une classe et d'instancier un objet de celle-ci en même temps.
- Très utile si nous souhaitons instancier qu'un seul objet d'une classe sans avoir à déclarer la classe dans un fichier

- Soit l'interface suivante :

```
1  public interface Hello {  
2      void greetings();  
3  }
```

- Ou encore de cette manière :

```
5  
6  // ou directement  
7  new Hello() {  
8      public void greetings() { System.out.println("Hi!"); }  
9  }.greetings(); // Affiche Hi!  
10
```


Classes anonymes

17

- Cette pratique est très utilisée en programmation événementielle
- Extrait d'une interface graphique qui décrit l'action à réaliser lors d'un clic sur un bouton :

```
1  JButton myButton = new JButton("click me");
2  myButton.addActionListener(new ActionListener() {
3      @Override
4      public void actionPerformed(ActionEvent ev) {
5          System.out.println("Someone has clicked on the button");
6      }
7  });
8
```

Interfaces fonctionnelles et lambdas

18

- Depuis Java 8, une interface qui ne contient qu'une méthode abstraite est promue automatiquement en une fonction permettant la syntaxe lambda. Ce type d'interface est appelée **interface fonctionnelle**.
- Le code précédent peut s'écrire ainsi :

```
1  JButton myButton = new JButton("click me");  
2  myButton.addActionListener(ev -> System.out.println(" button clicked !"));  
3
```

Interfaces fonctionnelles et lambdas

19

Si l'on reprend l'exemple précédent, nous pouvons déclarer `hello` comme une interface fonctionnelle :

```
1  @FunctionalInterface
2  public interface Hello {
3      void greetings();
4  }
```

Au lieu d'écrire ceci :

```
1  Hello h = new Hello() {
2      public void greetings() { System.out.println("Hi guys!"); }
3  };
4  h.greetings();
5
```

Interfaces fonctionnelles et lambdas

20

Nous pouvons donc directement l'écrire ainsi :

```
1  Hello h = () -> System.out.println("Hi guys!");  
2  h.greetings();
```

La méthode `greetings` est bien une méthode qui ne prend aucun argument `()` et retourne `void`, d'où la syntaxe `() -> System.out.println("Hi guys")`. L'interface `Hello` est donc considérée comme une fonction: `() -> ()`.