Programmation orientée objet

Série 8

Joel Cavat / 2020

Exercices théoriques

8.1 Exercice *(`Pair`)*

Réaliser une classe `Pair` immuable qui représente un couple `(x,y)`. Cette classe permet d'encapsuler deux valeurs de types différents. Ce type de structure est utile lorsque nous souhaitons retourner plusieurs valeurs.

Réalisez les fonctionnalités suivantes:

- obtenir la valeur de gauche
- obtenir la valeur de droite
- mapper la valeur de gauche à l'aide d'une fonction passée en argument
- mapper la valeur de droite à l'aide d'une fonction passée en argument
- mapper les deux valeurs à l'aide de deux fonctions passées en argument

Cette classe servira de base pour d'autres types de couples (`UniquePair` et `NumericPair`).

8.2 Exercice *(`NumberPair` et `UniquePair`)*

A l'aide de la classe `Pair`, écrivez une classe `NumberPair` qui garantit que les paramètres soient numériques. Ecrivez une autre classe `UniquePair` qui garanti que les paramètres soient du même type.

Note: Nous autorisons exceptionnellement l'héritage de classes à classes de façon totalement maîtrisée et dans l'optique d'approfondir les paramètres bornés. Nous pourrions la justifier ainsi: Nous souhaitons instancier une `Pair`. De plus, nous aimerions pouvoir substituer `Pair` par `NumberPair` et `UniquePair` partout ou `Pair` est utilisé.

Imaginez une solution (sans la mettre en oeuvre) qui supprimerait totalement l'héritage tout en permettant la contrainte ci-dessus.

8.3 Exercice *(StatusApp.java)*

Complétez le code du fichier `StatusApp.java` pour que l'extrait programme ci-dessous s'exécute correctement:

```java
public class StatusApp {
    public static void log(Object o) { System.out.println(o); }
    public static Status<Integer, String> random() {
        int chance = new Random().nextInt(3);
        if( chance == 0 ) {
            return Status.onWith(42);
        } else {
            return chance == 1 ? Status.off() : Status.errorWith("Oups");
        }
    }
    public static void main(String[] args) {
        Status<Integer, String> s = random();
        if( s.isOn() ) {
            log("Status On: " + s.get());
        } else if( s.isError() ) {
            log("Status Unstable: " + s.getError() );
        } else {
            log("Status Off");
        }
        s.accept(
            i -> log("Status On: " + i) ,
            () -> log("Status Off"),
            e -> log("Status Unstable: " + e)
        );
    }
}
```

8.4 Exercice *(`Box` comparable)*

L'interface `Comparable<T>` indique qu'il est possible de comparer un type. Elle oblige à redéfinir la méthode `int compareTo(T o)`

La classe `Integer` hérite de `Comparable<Integer>` ; la classe String hérite quant à elle de `Comparable<String>` :

```
public final class Integer implements Comparable<Integer> ... {
  ...
  public int compareTo(Integer anotherInteger) { ... }
}

public final class String implements Comparable<String> ... {
  ...
  public int compareTo(String anotherString) { ... }
}
```

Modifiez la classe `Box<T>` pour permettre de comparer deux boîtes:

- Une `Box` doit respecter l'interface `Comparable`
- Pour comparer des `Box`s, il suffit de comparer la valeur qu'ils encapsulent. Le paramètre est donc lui aussi comparable !
- Créer une méthode statique utilitaire pour préciser si une `Box` est plus grande qu'une autre
  - par ex.:

```
Box<Integer> b1 = new Box<>(1);
Box<Integer> b2 = new Box<>(2);
Util.isBigger(b1, b2); // doit retourner false
```

8.5 Exercice *(`IntegerStack` mais en générique !)*

Reprenez l'exercice sur la pile d'entiers et rendez-là générique.