Travaux pratiques scala 1 : programmation objet

Jonathan Lejeune

Objectifs

Ce sujet de travaux pratiques vous permettra de vous exercer sur la syntaxe scala et à la programmation objet.

Exercice 1 – Catalogue de produits

Le package de travail de cet exercice est `datacloud.scala.tpobject.catalogue`.

Dans cet exercice nous souhaitons programmer un objet catalogue qui associe un nom de produit (de type `String`) à un prix (de type `Double`).

Question 1

Créer un trait scala `Catalogue` qui offre les méthodes suivantes :

- `getPrice` qui pour un nom de produit donné renvoi son prix. Si le produit n'existe pas dans le catalogue, la méthode renvoie `-1.0`.
- `removeProduct` qui pour un nom de produit donné, efface ce dernier du catalogue.
- `selectProducts` qui pour un prix minimum et un prix maximum donnés, renvoie un `Iterable` de nom de produit dont le prix est compris entre ces bornes.
- `storeProduct` qui pour un nom de produit et un prix donnés ajoute un produit dans le catalogue. Si le produit existe déjà, le prix est tout simplement mis à jour.

Question 2

Créer une classe scala `CatalogueWithMutable` qui implante le trait `Catalogue` de la question précédente et se base sur une `Map` mutable.

Question 3

Tester votre classe avec la classe Junit suivante fournie dans les ressources de TP :

`datacloud.scala.tpobject.catalogue.test.CatalogueWithMutableTest`

Question 4

Créer une classe scala `CatalogueWithNonMutable` qui implante le trait `Catalogue` de la question précédente et se base sur une `Map` non mutable.

Question 5

Tester votre classe avec la classe Junit suivante fournie dans les ressources de TP :

`datacloud.scala.tpobject.catalogue.test.CatalogueWithNonMutableTest`

Exercice 2 – Structure de données en arbre binaire

Le package de travail de cet exercice est `datacloud.scala.tpobject.bintree`.

Soit le fichier IntTree.scala (fourni) :

```
package datacloud.scala.tpobject.bintree

sealed abstract class IntTree
case object EmptyIntTree extends IntTree
case class NodeInt(elem : Int, left : IntTree, right : IntTree) extends IntTree
```

Question 1

Créer un `object` scala `BinTrees` et y définir les fonctions suivantes :

- `contains` qui pour un `IntTree` et élément de type `Int` donnés renvoie vrai si l'élément est dans l'arbre, faux sinon.
- `size` qui renvois le nombre d'éléments d'un `IntTree` donné
- `insert` qui pour un `IntTree` et élément de type `Int` donnés, construit un nouvel IntTree en insérant l'élément dans l'IntTree donné tout en en équilibrant la taille des branches. Autrement dit, l'élément sera toujours inséré dans la branche de l'arbre qui comporte le moins d'élément

Vous utiliserez le mécanisme de pattern matching et la récursivité.

Question 2

Tester avec la classe Junit suivante fournie dans les ressources de TP :

`datacloud.scala.tpobject.bintree.test.IntTreeTest`

Question 3

Créer un fichier Tree.scala et y les généraliser types `IntTree`, `EmptyIntTree` et `NodeInt` respectivement en `Tree`, `EmptyTree` et `Node` afin de traiter des arbres de n'importe quel type `A`.

Question 4

Enrichir `BinTrees` des fonctions `contains`, `size` et `insert` pour les `Tree`.

Question 5

Tester avec la classe Junit suivante fournie dans les ressources de TP :

`datacloud.scala.tpobject.bintree.test.TreeTest`

Exercice 3 – Vecteur d'entier

Le package de travail de cet exercice est `datacloud.scala.tpobject.vector`.

Question 1

Écrire une classe scala `VectorInt`

- qui possède un attribut immutable `elements` qui est un tableau de `Int`, que l'on spécifiera à l'instanciation de l'objet
- et qui offre comme méthodes :
  - `length:Int` qui renvoie la taille du vecteur
  - `get(i:Int):Int` qui renvoie la valeur présente à l'indice `i` compris entre 0 et `length -1`

- ◇ `tostring` qui renvoie une représentation String du vecteur au format `(v0 v1 v2 ..)`
- ◇ `equals(a:Any):Boolean` qui teste l'égalité avec un objet `a`. Vous utiliserez un pattern matching : si a est de type `VectorInt` et que toutes les valeurs sont égales alors retourne vrai, faux sinon.
- ◇ `+(other:VectorInt):VectorInt` qui fait la somme de deux vecteurs
- ◇ `*(v:Int):VectorInt` qui multiplie toutes les valeurs du vecteur par le scalaire `v`
- ◇ `prodD(other:VectorInt):Array[VectorInt]` : (noté $\otimes$ en mathématiques) et qui calcul le produit dyadique entre deux vecteurs, où le résultat est une matrice que nous représentons ici par un tableau de vecteur (un vecteur est une ligne de la matrice). Un produit dyadique $u \otimes v$ de deux vecteurs de même taille revient à multiplier u en tant que vecteur colonne par v en tant que vecteur ligne. Par exemple :

$$u \otimes v \rightarrow \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} \otimes \begin{bmatrix} v_1 & v_2 & v_3 \end{bmatrix} = \begin{bmatrix} u_1v_1 & u_1v_2 & u_1v_3 \\ u_2v_1 & u_2v_2 & u_2v_3 \\ u_3v_1 & u_3v_2 & u_3v_3 \end{bmatrix}$$

Dans le même fichier (`VectorInt.scala`) on déclarera également un objet compagnon à la classe (un object scala `VectorInt`) et qui possède une méthode `implicit` permettant de convertir implicitement et automatiquement un tableau de `Int` en `VectorInt`

Question 2

Tester avec la classe Junit suivante fournie dans les ressources de TP :

`datacloud.scala.tpobject.vector.test.VectorIntTest`