Scala objet : la classe `Rational`

Dans ce TP nous allons développer une classe pour représenter les nombres rationnels $\mathbb{Q}$ (c'est-à-dire les fractions $\frac{a}{b}$ où a et b sont des entiers relatifs). Contrairement aux classes `Java` ou `C++` que vous avez pu écrire, nous allons adopter le style purement fonctionnel, plus sûr, que Scala préconise : l'état d'un objet ne sera jamais modifié après son instantiation. Chaque question de ce TP élabore sur le code de la question précédente. On conseille donc de maintenir un fichier `Rational.scala`, et le compiler et tester à chaque fois en chargeant son contenu dans l'interprète : entrer "`:load Rational.scala`" dans l'interprète a le même effet que d'y entrer tout le contenu du fichier `Rational.scala`.

1. Déclarez une classe `Rational` à deux paramètres entiers, qui initialiseront les champs `num` (numérateur) et `den` (dénominateur). Instancier un objet `x` qui vaudra $\frac{6}{3}$.
2. La valeur de `x` n'est pas affichée par l'interprète ; à la place on lit l'adresse mémoire de l'objet instancié. Pour que l'interprète affiche les rationnels, on doit implémenter une méthode `toString()`. Complétez la classe `Rational` en y ajoutant une méthode :

   ```
   override def toString() : String
   ```

   Le modifieur `override` indique qu'on veut *surcharger* cette méthode. [1]
3. On peut placer dans le corps d'une classe :
   — des déclarations de méthodes (`def`)
   — des déclarations de champs (`val`), ou
   — des expressions quelconques (exemple : `2+2`). Ces expressions sont évaluées à chaque instanciation d'un objet de la classe ; leur valeur est oubliée, mais si l'évaluation échoue par une exception, alors l'instanciation échoue.

   D'autre part, il existe une fonction pré-définie de signature :

   ```
   def require(requirement: Boolean): Unit
   ```

   qui ne renvoit rien si `requirement` est vrai, mais échoue avec une exception si `requirement` est faux.

   Servez-vous des deux points précédents pour compléter `Rational` de façon à rejeter les instanciations avec un dénominateur nul.
4. Ajouter à `Rational` une méthode de signature :

   ```
   def unary_-()
   ```

   qui renvoit l'opposé de `this`. Cette méthode ne modifiera pas `this`, elle créera et renverra un nouveau rationnel résultat. Évaluez l'expression `- x`. [2]
5. Ajoutez une méthode de signature :

   ```
   def +(that: Rational): Rational
   ```

1. Une méthode de ce nom existe déjà par défaut dans cette classe, héritée de la super-classe `Any` ; c'est elle qui affiche l'adresse mémoire de l'objet.
2. On rappelle que les méthodes nommées unary_XXX peuvent être appelées non seulement en écrivant comme d'habitude `x.unary_XXX()` mais aussi comme opérateur préfixe en écrivant `XXX x`.

qui calcule l'addition de deux rationnels. Pour rappel :

$$\frac{a}{b} + \frac{c}{d} = \frac{ad}{bd} + \frac{bc}{bd}$$

Cette méthode ne modifiera pas ses deux opérandes, elle créera et renverra un nouveau rationnel résultat. Calculer la valeur de l'expression `x + x`.[3] Comme vous le voyez, les fractions ne sont pas simplifiée.

6. On rappelle l'équation de simplification des fractions :

$$\frac{a}{b} = \frac{a/\operatorname{pgcd}(a,b)}{b/\operatorname{pgcd}(a,b)}$$

Ajoutez une méthode auxilliaire de signature :

```
private def pgcd(a: Int, b: Int): Int =
```

qui calcule le PGCD de `a` et `b`. On utilisera la définition récursive du PGCD :

$$\begin{aligned} \operatorname{pgcd}(a,b) &= a && \text{si } b = 0 \\ \operatorname{pgcd}(a,b) &= \operatorname{pgcd}(b, a \bmod b) && \text{si } b \neq 0 \end{aligned}$$

Ajoutez un champ `p` à la classe `Rational`, initialisé avec le PGCD de `den` et `num`, puis modifiez la classe de façon à simplifier tout rationnel lors de son instantiation. Que vaut maintenant `x + x` ?

7. Complétez `Rational` avec des méthodes correspondant aux autres opérations arithmétiques :

```
def -(that: Rational)
def *(that: Rational)
def /(that: Rational)
```

8. Que valent les expressions suivantes :

```
val one = new Rational(1,1)
val half = new Rational(1,2)
val third = new Rational(1,3)
one + third * half
(one + third) * half
one - one - one
half + 1
```

Vérifiez que les règles de précédence et d'associativité correspondent bien aux notions mathématiques d'addition, soustraction, multiplication. Pourquoi la dernière ligne échoue-t-elle ?

9. Surchargez les méthodes `+`, `-`, `*` et `/` de façon à accepter un argument de type `Int`. Assurez-vous maintenant que `half + 1` ait la bonne valeur. Evaluez maintenant `1 + half` ? Pourquoi est-ce que cela échoue ?

10. Déclarez une fonction :

```
implicit def intToRational(x:Int) = new Rational(x, 1)
```

en dehors de la classe `Rational`. Le modifieur `implicit` indique que ce n'est pas une fonction comme les autres : à tous les endroits où elle est visible, le compilateur a le droit de l'insérer pour résoudre des problèmes de type. Ici, on déclare donc la possibilité de convertir implicitement un entier en rationnel. Ré-évaluez `1 + half` maintenant.

3. On rappelle qu'une expression infixe `A B C` est interprétée comme un appel à la méthode `B` : `A.B(C)`.