Exercises — Scala (Week Five)

Practice with classes

Spring term 2017

Overview

We now have enough of the syntax to have some fun playing with classes.

Learning Objectives

- Practice with classes.
- Practice with case classes.
- Practice with companion classes.

Testing

This exercise sheet does not come with any existing code or test harness — you are required to write appropriate tests using the `ScalaTest` framework. You should follow *good practice* with respect to `sbt` folder structure and build files (Hint: copy them from an earlier week).

The Questions

1. (a) Implement a `Counter` class. The constructor should take an `Int`. The methods `inc` and `dec` should increment and decrement the counter respectively returning a new `Counter`. HereâĂŹs an example of the usage:

    ```
    scala> new Counter(10).inc.dec.inc.inc.count
    res02: Int = 12
    ```

   (b) Augment the `Counter` to allow the user can optionally pass an `Int` parameter to inc and dec. If the parameter is omitted it should default to 1.

   (c) Reimplement `Counter` as a case class, using `copy` where appropriate. Additionally initialise `count` to a default value of `0`.

   (d) Here is a simple class called `Adder`:

    ```
    class Adder(amount: Int) {
      def add(in: Int) = in + amount
    }
    ```

    Extend `Counter` to add a method called `adjust`. This method should accept an `Adder` and return a new `Counter` with the result of applying the `Adder` to the `count`.

2. (a) Implement a companion object for a `Person` class containing an `apply` method that accepts a whole name as a single string rather than individual first and last names.

   Tip: you can split a `String` into an `Array` of components as follows:

```
scala> val parts = "John Doe".split(" ")
parts: Array[String] = Array(John, Doe)

scala> parts(0)
res36: String = John
```

   (b) What happens when we define a companion object for a case class?

   Take our `Person` class and turn it into a case class. Make sure you still have the companion object with the alternate `apply` method as well.

3. (a) Write two classes, `Director` and `Film`, with fields and methods as follows:

   - `Director` should contain:
     - a field `firstName` of type `String`
     - a field `lastName` of type `String`
     - a field `yearOfBirth` of type `Int`
     - a method called `name` that accepts no parameters and returns the full name
   - `Film` should contain:
     - a field `name` of type `String`
     - a field `yearOfRelease` of type `Int`
     - a field `imdbRating` of type `Double`
     - a field `director` of type `Director`
     - a method `directorsAge` that returns the age of the director at the time of release
     - a method `isDirectedBy` that accepts a `Director` as a parameter and returns a `Boolean`

   You will find appropriate demo data on the repo under the folder `scala-exercises`; you will need to adjust your constructors so that the code works without modification.

   Implement a method of `Film` called `copy`. This method should accept the same parameters as the constructor and create a new copy of the film. Give each parameter a default value so you can copy a film changing any subset of its values:

```
highPlainsDrifter.copy(name = "L’homme des hautes plaines")
// returns Film("L’homme des hautes plaines", 1973, 7.7, /* etc */)

thomasCrownAffair.copy(yearOfRelease = 1968,
  director = new Director("Norman", "Jewison", 1926))
// returns Film("The Thomas Crown Affair", 1926, /* etc */)

inception.copy().copy().copy()
// returns a new copy of ‘inception‘
```

(b) Write companion objects for `Director` and `Film` as follows:

- The `Director` companion object should contain:
  - an `apply` method that accepts the same parameters as the constructor of the class and returns a new `Director`;
  - a method `older` that accepts two `Directors` and returns the oldest of the two.
- The `Film` companion object should contain:
  - an `apply` method that accepts the same parameters as the constructor of the class and returns a new `Film`;
  - a method `highestRating` that accepts two `Films` and returns the highest `imdbRating` of the two;
  - a method `oldestDirectorAtTheTime` that accepts two `Films` and returns the `Director` who was oldest at the respective time of filming.

(c) We can dispose of much of the *boilerplate* by converting the `Director` and `Film` classes to *case classes*. Do this conversion and work out what code we can remove.