Exercises — Week Seven

Truly Functional

Spring term 2016

You need to have installed the current Scala distribution and the current Java distribution before commencing these exercises. Don't forget the documentation for both distributions as they will come in very useful.

For some of these exercises you will need the test code you used in a previous exercise sheet, namely the classes below the `atomicscala` folder on the repo. Any source code examples are available under the `week07` folder.

`map` & `reduce`

1. (a) Modify the following code:

   ```
   import atomic.AtomicTest._

   val v = Vector(1, 2, 3, 4)
   v.map(n => n + 1) is Vector(2, 3, 4, 5)
   ```

   so the anonymous function multiplies each value by 11 and adds 10.

   Satisfy the following tests:

   ```
   val v = Vector(1, 2, 3, 4)
   v.map(/* Fill this in */) is Vector(21, 32, 43, 54)
   ```

   (b) Can you replace `map` with `foreach` in the above solution? What happens? Test the result.

   (c) Rewrite the solution for the previous exercise part using `for`. Was this more or less complex than using `map`? Which approach has the greater potential for errors?

2. Rewrite the following code:

   ```
   import atomic.AtomicTest._

   val v = Vector(1, 2, 3, 4)
   v.map(n => n + 1) is Vector(2, 3, 4, 5)
   ```

   using a `for` loop instead of `map`, and observe the additional complexity this introduces.

3. Rewrite the following code:

```
// Reduce.scala
import com.atomicscala.AtomicTest._

val v = Vector(1, 10, 100, 1000)
v.reduce((sum, n) => sum + n) is 1111
```

using `for` loops.

4. Use `reduce` to implement a method `sumIt` that takes a variable argument list and sums those arguments.

   Satisfy the following tests:

   ```
   sumIt(1, 2, 3) is 6
   sumIt(45, 45, 45, 60) is 195
   ```

Traits and Inheritance

5. (a) Define a trait called `Shape` and give it three abstract methods:
      i. `sides` returns the number of sides;
      ii. `perimeter` returns the total length of the sides;
      iii. `area` returns the area.

   Implement `Shape` with three classes: `Circle`, `Rectangle`, and `Square`. In each case provide implementations of each of the three methods. Ensure that the main constructor parameters of each shape (e.g., the radius of the circle) are accessible as fields.

   Tip: The value of π is accessible as `math.Pi`.

   (b) The solution from part a delivered three distinct types of shape. However, it doesn't model the relationships between the three correctly. A `Square` isn't just a `Shape` — it's also a type of `Rectangle` where the width and height are the same.

   We want to avoid case-class-to-case-class inheritance, so refactor the solution to the last exercise so that `Square` and `Rectangle` are subtypes of a common type `Rectangular`.

   (c) i. Make `Shape` a *sealed trait*.
      ii. Then write a singleton object called `Draw` with an `apply` method that takes a `Shape` as an argument and returns a description of it on the console.

   For example:

   ```
   Draw(Circle(10)) // returns "A circle of radius 10cm"

   Draw(Rectangle(3, 4))
      // returns "A rectangle of width 3cm and height 4cm"

   // and so on...
   ```

iii. Finally, verify that the compiler complains when you comment out a `case` clause.

(d) Write a sealed trait `Colour` to make our shapes more interesting.

i. give `Colour` three properties for its RGB values;

ii. create three predefined colours: `Red`, `Yellow`, and `Pink`;

iii. provide a means for people to produce their own custom `Colour`s with their own RGB values;

iv. provide a means for people to tell whether any `Colour` is "light" or "dark".

Note: A lot of this is left deliberately open to interpretation. The important thing is to practice working with traits, classes, and objects. Decisions such as how to model colours and what is considered a *light* or *dark* colour can either be left up to you or discussed with other class members.

(e) Edit the code for `Shape` and its subtypes to add a colour to each shape.

Finally, update the code for `Draw.apply` to print the colour of the argument as well as its shape and dimensions (hint: you may want to deal with the colour in a helper method):

i. if the argument is a predefined colour, print that colour by name:

ii. if the argument is a custom colour rather than a predefined one, print the word "light" or "dark" instead.

6. Scala developers don't just use types to model data. Types are a great way to put artificial limitations in place to ensure we don't make mistakes in our programs. In this exercise we will see a simple (if contrived) example of this using types to prevent division by zero errors.

Dividing by zero is a tricky problem — it can lead to exceptions. The JVM has us covered as far as floating point division is concerned but integer division is still a problem:

```
scala> 1.0 / 0.0
res0: Double = Infinity

scala> 1 / 0
java.lang.ArithmeticException: / by zero
```

Let's solve this problem once and for all using types!

Create an object called `divide` with an `apply` method that accepts two `Int`s and returns `DivisionResult`. `DivisionResult` should be a sealed trait with two subtypes: a `Finite` type encapsulating the result of a valid division, and an `Infinite` type representing the result of dividing by 0.

Here are some examples:

```
scala> divide(1, 2)
res7: DivisionResult = Finite(0)

scala> divide(1, 0)
res8: DivisionResult = Infinite
```

Finally, write a sample function that calls `divide`, matches on the result, and returns a sensible description.

7. Create a simple model for publisher data. Code a set of traits and classes according to the following description:

   - A *publication* is a *book* or a *periodical*.
   - A *book* has an *author* while a *periodical* has an *editor*.
   - *Periodical*s have many *issues*, each of which has a *volume* and an *issue* number.
   - A *manuscript* is a document of a certain *length* written by an *author*.
   - A *book* is a *manuscript*, but an *issue* of a *periodical* contains a sequence of *manuscripts*.

   Tip: a sequence of type `A` has type `Seq[A]`.

8. To implement a `LinkedList` in Scala we need to combine our knowledge of generic types with our existing knowledge of sealed traits. We can define a linked list as a sealed trait `LinkedList[A]` with two subtypes:

   - a class `Pair[A]` with two fields, `head` and `tail`:
     - `head` is the item at this position in the list;
     - `tail` is another `LinkedList[A]` — either another `Pair` or an `Empty`;
   - a class `Empty[A]` with no fields.

   (a) Start by writing the simplest trait and classes you can so that you can build a list. You should be able to use your implementation as follows:

   ```
   val list: LinkedList[Int] = Pair(1, Pair(2, Pair(3, Empty())))

   list.isInstanceOf[LinkedList[Int]] // returns true

   list.head      // returns 1 as an Int
   list.tail.head // returns 2 as an Int
   list.tail.tail // returns Pair(3, Empty()) as a LinkedList[Int]
   ```

   [Yes, this is very similar to the example from class.]

   (b) Now we have our `LinkedList` class, let's give it some useful methods. Define the following:

   i. a method called `length` that returns the length of the list;
   ii. a method `apply` that returns the `n`th item in the list;
   iii. a method `contains` that determines whether or not an item is in the list.

   In each case, start by writing an abstract method definition in `LinkedList`. Think about the types of the arguments and the types of the results. Then implement the method for `Empty` — it should be pretty easy to provide a default implementation for an empty list. Finally, implement the method on `Pair`. The implementation will be recursive and defined in terms of `head` and `tail`.

   Hint: If you need to signal an error in your code (there's one situation in which you will need to do this), consider throwing an exception (although we will dispense with this approach later in the course). Here is an example:

```
throw new Exception("Bad things happened")
```

(c) Implement the `Pair` class. It should store two values — `one` and `two` — and be generic in both arguments. Example usage:

```
scala> val pair = Pair[String, Int]("hi", 2)
pair: Pair[String,Int] = Pair(hi,2)

scala> pair.one
res13: String = hi

scala> pair.two
res14: Int = 2
```

9. Implement a trait `Sum[A, B]` with two subtypes `Left` and `Right`. Create type parameters so that `Left` and `Right` can wrap up values of two different types.

   Hint: you will need to put both type parameters on all three types. Example usage:

```
scala> Left[Int, String](1).value
res24: Int = 1

scala> Right[Int, String]("foo").value
res25: String = foo

scala> val sum: Sum[Int, String] = Right("foo")
sum: Sum[Int,String] = Right(foo)

scala> sum match {
     |   case Left(x) => x.toString
     |   case Right(x) => x
     | }
res26: String = foo
```

10. In a previous exercise question you "solved" the problem of dividing by zero by defining a type called `DivisionResult`. This forced us to handle the possibility of a division by zero in order to access the value.

    With our knowledge of generics we can now generalise `DivisionResult` to encapsulate potential errors of any type. Modify `DivisionResult` to create a generic trait called `Maybe` with two subtypes, `Full` and `Empty`. Rewrite `divide` to return a `Maybe[Int]`.

    Example usage:

```
divide(1, 0) match {
  case Full(value) => println(s"It's finite: ${value}")
  case Empty()     => println(s"It's infinite")
}
```

11. Sum types and product types are general concepts that allow us to model almost any kind of data structure. We have seen two methods of writing these types, traits and generics — when should we consider using each?