Exercise 9:
A DSL for reactive animations

TECHNISCHE UNIVERSITÄT DARMSTADT

Software Engineering Design & Construction
WS 2016/17 - Dr. Michael Eichberg, M.Sc. Matthias Eichholz

Although this exercise is not graded, it is highly recommended to also do them on your own. Just looking at a solution is much easier in comparison to actually coming up with it. Support can be found in the office hours of our tutors and in the forum at `https://www.fachschaft.informatik.tu-darmstadt.de/forum/viewforum.php?f=234`.

Task 1 Order Management System II

For the first milestone of the order management system (OMS) for TastyPizza Inc., you have already implemented a class hierarchy for representing the products offered by TastyPizza Inc. For the second milestone, you should extend the OMS library further. The following new requirements need to be fulfilled by your OMS:

- An order should maintain a list of ordered products with all their associated information.
- A bill printer that uses the interface introduced in the last subtask to print a bill in a simple but readable format to a string. If the order contains alcohol, a notice for the delivery boy should be printed to verify the customer's age.

Use the Visitor Pattern to implement the bill printer. Use a flexible variant of the visitor with two methods per product: `def startVisit(...)` and `def endVisit(...)`. It would visit some objects twice, for example, a pizza once via `startVisit()`, then all its toppings, then the same pizza again via `endVisit()`. This will make certain printing tasks easier.
Start by extending the existing product trait with a method `def accept(v: ProductVisitor)`.

```
trait Product {
  def name: String
  def price: Price
  def accept(v: ProductVisitor)
}
```

Provide at least 3 test cases that reflect the workflow of an order and print a bill to the console.

Task 2 A DSL for Reactive Animations

In this exercise, you will develop a domain specific language for animations, using the REScala[1] framework. Your solution should be as simple as possible. As a first step, make yourself familiar with the code. Add your implementations in the corresponding places and test them thoroughly. Ultimately, your task for this exercise is to develop an animation DSL that allows clients to repeatedly animate an integer value from 0 to 100 in 2 seconds as follows:

```
val anim = animate (0 ->> 100) in (2 secs) repeat
val signal = anim.start()
```

There are three ingredients to this DSL:

a) paths, as in `(0 ->> 100)` above,

b) time, as in `(2 secs)` above, and

c) building the animation including the final REScala signal.

In order to be able to develop this DSL, you need to understand two concepts of Scala: infix and postfix notation and implicit conversions.

[1] `http://www.rescala-lang.com`

Infix and postfix notation

First, the expression `0 ->> 100` above is equivalent to `0.->>(100)`, that is an invocation of method `->>` on object `0` with the argument `100`[2]. The first notation is called *infix* notation since it is similar to infix operators, such as the common `+` on integers. In Scala, `1 + 2` is also equivalent to `1.+(2)`. You can use this syntax for every method.
Second, the syntax `(2 secs)` is equivalent to `2.secs`, which in turn is equivalent to `2.secs()`, i.e., an invocation of method `secs` on object `2`. This notation is called *postfix* notation. Make sure that you always wrap an expression in postfix notation in parentheses, otherwise the compiler might get confused (for a good reason that is beyond the scope of this exercise). Alternatively, if you prefer the dot notation, you can just write `2.secs` in this exercise.

Implicit conversions

If you invoke a method on an object that the compiler cannot find (because the class of the object does not define it), it tries to implicitly convert the object into one that has the given method. To define an implicit conversion, we write a function that has the **`implicit`** modifier. For example, since integers do not have a method `secs`, we can use a little trick to trigger an implicit conversion as follows:

```
object Time {
  implicit def long2TimeOps(t: Long): LongTimeOps = new LongTimeOps(t)
}

class LongTimeOps(val t: Long) {
  def secs: Time = ...
}

class Time(val nanos: Long) {
  ...
}
```

We can now write the following:

```
import Time._

val t: Time = 2.secs
```

The import declaration makes sure the compiler knows about our implicit conversion. When it encounters `2.secs` it looks for method `secs` in the Int class (again `0` is an object with a class in Scala). Since it cannot find it, it searches for implicit conversions in scope from integers to some class that defines method `secs`. It will find our `long2TimeOps` that we just imported and will apply it. The last line will effectively be rewritten by the compiler to

```
val t: Time = long2TimeOps(2).secs
```

As you can see, now everything is correct, since `secs` is a method in class `LongTimeOps`, of which an instance is returned by conversion `long2TimeOps`.

Task 3 Time

Implement all methods you find in the `Time` and `LongTimeOp` classes that are given in the code template. In order to make sure one can use doubles for times, as in `(2.5 secs)`, implement the following implicit conversion together with class `DoubleTimeOps`:

```
implicit def double2TimeOps(t: Double): DoubleTimeOps = ???
```

Task 4 Paths

We want to animate one dimensional paths of `Double` values such as `0 ->> 100` ("from 0 to 100") above. We can represent such a path as a simple function that takes a `Double` from `0.0` to `1.0` and returns a `Double`:

```
type Path = Double => Double
```

For example, the path `10 ->> 100` can be represented as the Scala function `x => 10 + 90*x`.
Using a similar approach as for class `Time`, use an implicit conversion to write a small DSL to create linear paths, e.g., `x ->> y` for any integers or doubles `x` and `y`.

[2] In Scala, integer values such as 0 or 100 can indeed be seen as objects with a class.

Task 5 Animation

Using `Time` and `Path`, develop the animation DSL. Implement method `animate` and class `Animation`, so that you can achieve the syntax in the above example:

```
object Animation {
  def animate(f: Path): Animation = ???
}

class Animation(...) {
  def in(t: Time): Animation = ???
  def repeat: Animation = ???
  def start(): Signal[Double] = ???
}
```

Your `Animation` class should be immutable, i.e., its method should create new animation objects. The `start` method should create a new signal using a timer signal supplied by `Clock.time`. Hint: the bounce and repeat behavior can be modelled as a function converting the current time from `Clock.time` to an interval in $[0;1]$ that is used as an input to the path function.
The code template contains an example how to draw a ball that follows the animated signal. If you have implemented everything correctly until here, you should see a ball repeatedly moving on a line from left to right when you run the `Main` application.
Implement two more animations with your DSL. One that moves a ball on a sine wave forth and back and one that perpetually moves a ball in a circle (in one direction and *not* forth and back). Hint: you might need to create more than one signal per animation. Add the corresponding drawing code in the `Main` object so that you can visually verify that your animation is behaving as expected.