ASSIGNMENT 9 – DSLS

Advanced programming paradigms

Question 1 – *Implicit conversions*

You will develop a program that enables easy temperature conversions. It uses a a common type for every temperature – `Temperature` – which should be declared `abstract` and `sealed` (all the sub-classes declared in this source file are the only subclasses allowed). Two sub-types of this class are `Celsius` and `Kelvin`.

(a) Write the required *implicit conversions* and the required code to allow the following code to run:

```
1 val a: Celsius = 30
2 val b: Kelvin = 30
3 val c: Kelvin = Celsius(10)
4 val d: Celsius = c
5 val e: Temperature = d
6
7 println(a) // Should print "30° C"
8 println(b) // Should print "30 K"
```

(b) What will you get on the console if you try to print `e`?

10 C

(c) Why is it interesting to have the `Temperature` class? Explain!

Collection of Temperatures, object companion with implicits inside

Question 2 – *Interactive lab*

In this interactive lab, we will discover together how a classical example of interactive drawing program can be adapted to express more simply complex ideas thanks to a an internal DSL[1].

(a) In this first part, we will discover the example code and see how to make it work

1. In the Scala IDE, make a new Scala project lab1 with a Scala application `Lab1` and paste in all this code. Run the program. What happens? Why? (Look at code in the `App` object.)
2. Have a look at the code. You'll see a bunch of geometric shapes: `Circle`, `Rectangle`, `Arrow`, all of which extend `Drawable`. There is also a class `Point` for specifying points with (x, y) values.
3. Next, there is a `Drawing` – a Swing component to which you add `Drawable` objects.
4. Then there are a bunch of effects. An effect is something that happens as time elapses—moving a shape, making it visible or invisible, etc. The effects are pretty simple—their act method is called many times, and it does something, such as moving the center (`MoveEffect`) or changing the transparency (`HideEffect`).
5. More interesting are the effect combinators. One wants to say *"Do these two effects together, and then do the other effect."* When `e1` and `e2` are effects, then you can make a `TogetherEffect(e1, e2)` that runs them in parallel, and an `InOrderEffect(e1, e2)` that runs `e1` and then `e2`, and also a `BackwardsEffect(e1)` where time runs backwards.
6. So, now put these to use. Make it so that `c1` moves as before, and when it is done, then `c2` moves to `new Point(200, 200)` in 6000 milliseconds.
7. Make it so that, as `c2` moves, it also hides in 3000 milliseconds.

(b) In this second part, we will start making our *Effect* DSL.

1. That API is a mess. We really want to say

[1] This interactive lab was given by C. Horstmann during his invited lecture in 2015

```
e1 followedBy (e2 and e3)
```

2. Make it so. Simply define methods `followedBy` and `and` in the `Effect` class that return an `InOrderEffect` or a `TogetherEffect`.
3. Also define a method `reversed` that makes a `ReverseEffect` and try out

```
e1 followedBy (e2 and (e3 followedBy (e3 reversed)))
```

4. What does it do?
5. Maybe that's nicer with operators? Make it so one can write

```
e1 ==> (e2 || (e3 ==> -e3))
```

6. How did you do that?

(c) Code blocks

1. What if we want to make another change to those shapes? Maybe we want a circle to grow. Sure, one could write a `GrowEffect`. But wouldn't it be nicer if we could just specify the grow behavior in a code block? Like

```
val e = update(2000) { t => c2.radius = 30 + 20 * t }
```

to indicate that `c2` should have its radius changed from 30 to 50 in 2000 milliseconds.

2. So, we need an `UpdateEffect`. It should take
   - A duration
   - A block of code for updating, with a `Double` parameter and `Unit` result

   Its `act` method simply calls the code block with `completion(t)`, which ranges from 0 to 1 as the timer tick ranges from 0 to the duration of the effect. Implement the class. Then implement a curried `update` method (for simplicity, in the `Lab1` object) that makes an instance, given an `Int` and a code block. Then start the effect `e` above.
3. Explain what happens if you try

```
val e = update(2000) { c2.radius = 30 + 20 * _ }
```

(d) Implicit conversions

1. We want to repeat an effect `n` times. That's easy, thanks to the miracle of recursion:

```
abstract class Effect ... {
  ...
  def times(n: Int): Effect = {
    if (n == 1) this else new InOrderEffect(this, times(n - 1))
  }
}
```

Add that method, and then change the `Lab1` object to call `d.start(e1 times 3)`
What happens?

2. What happens when you try

```
d.start(3 times e1)
```

Why?

3. Ok, that can't work — `times` isn't a method of `Int`. So that's where implicit conversions come in. We need to convert `Int` to some object, say `EffectInt`, with a `times` method. Make such a class and method, and then try out

```
1 d.start(new EffectInt(3) times e1)
```

4. Sure, that works, but it's ugly from the point of view of a DSL. Make an implicit conversion from `Int` to `EffectInt`. Just place it inside the `Lab1` object. Then try

```
1 d.start(3 times e1)
```

and rejoice.

(e) Implicit parameters

1. Right now, the code for making an arrow between two `Drawable` is

```
1 d += new Arrow(c1, c2)
```

2. Really, in a DSL, we'd like to say

```
1 c1 --> c2
```

3. What about the `d+=` ?
4. That's boring — of course we need to add the arrow to the component so that it gets painted.
5. Relieving boredom is what implicit parameters are for. Define a `->` method on the `Drawable` class with a regular parameter `to: Drawable` and an implicit parameter `d` of type `Drawing`. Make it call `d += new Arrow(this, to)`.
6. Now replace `d += new Arrow(c1, c2)` with `c1 -> c2` in `Lab1`. What happens?
7. That couldn't have worked. There is no implicit `Drawing` anywhere. Add `implicit` before the declaration of the `Drawing` instance `d` in `Lab1`. Now what happens?