Lecture 16

A powerful feature of Scala is that can interoperate almost seamlessly with Java code. So, we can use any Java library from Scala.

Imagine we were using a Java library for vectors, such as the one shown in fig. 26.1a. Although this is high-quality Java code, it can feel very cumbersome to use from Scala. Compared to the terse Scala programs we've been writing, we have to write lots of verbose method calls:

```
val v1 = new Vector2D(2, 3)
val v2 = v1.add(v1).mul(3)
```

If the library had been written in Scala, it may look more like the code in fig. 26.1b, which would allow us to write the following code instead:

```
val v1 = new Vector2D(2, 3)
val v2 = (v1 + v1) * 3
```

Imagine that this is a large, full-featured, and well-tested library, so it would be foolish to rewrite the whole thing, just to get a nicer API.

One solution is to "wrap" the Java API in Scala class, as shown below:

```
class VecAdaptor(vec: Vector2D) {
  def +(other: VecAdaptor)= new VecAdaptor(vec.add(other.vec))
  def *(factor: Double) = new VecAdaptor(vec.mul(factor))
  def unary_- = new VecAdaptor(vec.neg())
}
```

This is an example of the *adaptor pattern*. Imagine that the Java class had several other methods that we had not bothered to adapt. If we wanted to use them too, we'd have to either add adaptor methods for all them (which would be a lot of code), or we'd have to continuously wrap and un-wrap in Scala, which would very tiresome.

Implicit Classes

In principle, we want to take an existing class, `Vector2D`, and add the methods `+`, `*`, etc. to it. However, classes cannot be modified after they are declared, so we need to create a wrapper. However, instead of explicitly wrapping the class, we can create an *implicit class* as follows:

```
implicit class RichVector2D(vec: Vector2D) {
  def +(other: Vector2D): Vector2D = vec.add(other)
  def *(factor: Double): Vector2D = vec.mul(factor)
  def unary_- = vec.neg()
}
```

Given this definition, when Scala sees the following code:

```
val v1 = new Vector2D(2, 3)
val v2 = (v1 + v1) * 3
```

It will automatically insert the wrapper and turn it into the following:

```
val v1 = new Vector2D(2, 3)
val v2 = new RichVector2D(new RichVector2D(v1) + v1) * 3
```

Here is a simpler example:

```
(new Vector2D(50, 60)) * 30
```

Normally, this expression would not type-check, since `Vector2D` does not have a method `*` that accepts integers. However, there is an implicit class that wraps `Vector2D` that does accept integers, so the expression is transformed to:

```java
public class Vector2D {
  private Double x;
  private Double y;

  public Vector2D(Double x, Double y) {
    this.x = x;
    this.y = y;
  }

  public Vector2D mul(Double factor) {
    return new Vector2D(factor * x,
                        factor * y);
  }

  public Vector2D add(Vector2D other) {
    return new Vector2D(this.x + other.x,
                        this.y + other.y);
  }

  public Vector2D neg() {
    return new Vector2D(-x, -y);
  }
}
```

(a) A simple vector library in Java.

```scala
class Vector2D(x: Double, y: Double) {
  def +(other: Vector2D): Vector2D = {
    new Vector2D(this.x + other.x,
                 this.y + other.y)
  }

  def *(factor: Double): Vector2D = {
    new Vector2D(factor * x,
                 factor * y)
  }

  def unary_- = new Vector2D(-x, -y)
}
```

(b) A similar library in Scala.

```scala
(new RichVector2D(new Vector2D(50, 60))) * 30
```

The `RichVector2D` implicit class lets us write `(new Vector2D(50, 60)) * 30`, but we can't write `30 * (new Vector2D(50, 60))`, as it assumes that builtin integers have a `*` method that accepts vectors. However, we can create an implicit class that wraps built-in integers too:

```scala
implicit class RichInt(n: Int) {
  def *(v: Vector2D): Vector2D = v.mul(n)
}
```

Scala's type-checker rewrites `3 * ...` to `(new RichInt(3)) * ...`. Notably, we didn't have to make any changes to the built-in `Int` type. (Not that we could, since it is truly built in to Scala.)

Implicit Functions

Although we can use standard mathematical notation to manipulate vectors, we are still stuck creating constant vectors as follows:

```scala
val v1 = new Vector2D(2, 3)
```

Suppose we wanted to just write `(2, 3)` to denote a vector. However, we aren't invoking a method in above, implicit classes can't help.

However, can use implicit functions:

```scala
implicit def pointToVector2D(pt: (Int, Int)): Vector2D = {
  new Vector2D(pt._1, pt._2)
}

val v1: Vector2D = (2, 3)
```

Warning

Code that uses a lot of implicit conversions can become very unreadable very quickly. So, you should use them with care. Moreover, when there are several implicit classes available, it may be ambiguous which conversion is used. When there is ambiguity, Scala will simply signal an error. When designing Scala APIs, it is considered good practice to place implicits in their own object, so that users have to opt-in to using them. For example, we may declare our implicit conversions for vectors as shown in fig. 26.2. Now, to use these implicits, we would have to write:

```scala
import Implicits._
```

```
object Implicits {

  implicit class RichVector2D(vec: Vector2D) {
    def +(other: Vector2D): Vector2D = vec.add(other)
    def *(factor: Double): Vector2D = vec.mul(factor)
    def unary_- = vec.neg()
  }

  implicit def pointToVector2D(pt: (Int, Int)): Vector2D = new Vector2D(pt._1, pt._2)

}
```

Figure 26.2: Implicits should be placed in their own object.

This is a signal to the reader that implicit conversions are being used.

> **Think!** If you know an untyped language such as JavaScript or Python, think about what it would take to do something similar to implicit classes.

Built-in Implicits in Scala

Scala has several implicits that are available by default. Many of these implicits are used to add features to builtin Java classes that Scala also uses. For example, Scala arrays are just Java arrays. However, you can use methods such as `.map` on arrays in Scala. This is achieved using implicits conversions. From the Scala console, if you enter `:implicits -v`, you can see the complete list of implicits that are available.