Scala fundamentals - Implicits basic

Aims

In this session we are going to demystify what *Implicit Values*, *Implicit Functions* and *Implicit Classes* are in Scala.

Prerequisites

You have completed up to part 10 of Scala fundamentals.

Lesson steps

> This may take a while, grab a cup of tea!

TLDR; Scala has yet more keywords that we haven't explored, they are `implicit` and `implicitly[T]`.

You may have seen code like the following:

```
implicit val executionContext = Cafe.ec
```

We are going to explore what the implicit keyword can do for us in Scala.

Get ready for a deep dive!

Implicit Parameters

In Scala we have already discovered values in the form of:

```
val name : String = "Adam"
```

We also know to use this in a function, we pass it in as a parameter to that function.

```
def sayName(name : String) = println(name)

val name : String = "Adam"
sayName(name)
```

What would we have to do if we wanted to change the format of the string that is returned from `sayName()`. We could either pass in another argument of some higher-order function we would apply on the string, alternatively we can pass in an implicit argument.

The syntax for an implicit set of arguments are as follows:

```
def f(a : T)(implicit f : I)
```

As you've probably noticed, we have an additional set of parenthesis `()` after our first set of arguments. We have also including the keyword `implicit` at the start of the argument list to specify that these arguments will be provided *implicitly*.

Let's take a look at an example:

```
// Define a trait to express an interface on how to format a string
trait Formatter {
    def apply(s : String) : String
}

// Implement a concrete instance of the formatter to transform to uppercase
class UpperCaseFormatter extends Formatter {
    override def apply(s: String): String = s.toUpperCase()
}

// Create an instance of the formatter, and make it available implicitly
implicit val upperCaseFormatter : Formatter = new UpperCaseFormatter()

// specify that sayName() accepts an implicit formatter
def sayName(name : String)(implicit formatter : Formatter) = {
    formatter.apply(name)
}
```

When we execute the code above, `sayName("Adam")` will return the value `ADAM`. How do you think this works?

What we have expressed here, is that the `sayName` function accepts an implicit value as an argument. Scala takes care of this for us at runtime, and in fact calls the function as

`sayName("Adam")(formatter)`.
We then use the value of `Formatter` inside of the function to format the string.

You will come across implicits a lot in Scala, particularly in Play Framework which we will be working with later.

Task

- Implement the above code and try it out for yourself
- What happens if you implement more than one implicit formatter? What if we create a `class LowerCaseFormatter`?

Let's take a look at another example, we are going to implement a function that accepts an implicit argument of `Logger` class. The `class Logger` is responsible for logging out to the console, alternatively it could log out JSON to a logging service such as Kibana.

```
object ImplicitExample {

  trait Logger[T] {
    def log(f : T) : T
  }

  class IntLogger() extends Logger[Int] {
    override def log(f: Int) : Int = {
      println(f)
      f
    }
  }

  implicit val logger = new IntLogger()

  def double(n : Int)(implicit logger : Logger[Int]) = {
    logger.log {
      n * 2
    }
  }

  double(4)

}
```

Let's break this down.

We have firstly implemented an interface that each logger must conform to, we have done this by implementing a trait named `Logger`:

```
trait Logger[T] {
  def log(f : T) : T
}
```

Here we have expressed that our Logger is generic by having a type `[T]`, this means we can implement a Logger that can work with any data type, that may be `String`, `Double` `Person`. i.e. `Logger[Int]`.
Since we have implemented generics, our `log(f : T)` method now accepts an `Int` and returns an `Int` we implemented a `Logger[Int]`.

We have a concrete implementation of the `Logger[Int]` below that:

```
class IntLogger() extends Logger[Int] {
    override def log(f: Int) : Int = {
      println(f)
      f
    }
}
```

We now have a concrete implementation of our generic `Logger`. We have extended `Logger[Int]` and implemented the `log` method which we have chosen to perform a `println` and return the value.

We have then made an instance of the `class IntLogger` available implicitly within scope of the ImplicitExample object.

```
implicit val logger = new IntLogger()
```

We have then implemented a double function which accepts an `Int` as an argument, and an implicit logger. `def double(n : Int)(implicit logger : Logger[Int])`. We have then used our concrete implicit logger within our double function:

```
logger.log {
    n * 2
}
```

Task

- Try and implement an implicit `Logger[String]`

So that is enough on implicit values!

Implicit functions ("view bounds")

A view bound specifies that a type can be "*viewed as*" another type, therefore it enables the use of some type `A` as *if* it were some type `B`.

An implicit function is a function that is implicitly available in scope that can be used at runtime like any other function without explicitly calling the function. An example of this would be to do a type conversion from a `Double` into a `String`. The implicit function for that would look like:

```
implicit def doubleToString(n : Double) : String = {
    println(s"I am converting in the implicit")
    s"$n"
}
```

Here we have declared an implicit function in scope that accepts a `Double` and returns a `String`. We have used string interpolation to convert the data type of the argument.

We can then use this implicit function as follows:

```
val num : Double = 2.00
val converted : String = num
```

We haven't explicitly called the function in the values above. We've allowed scala to determine this at runtime. By specifying that the value `converted` must be a type `String`, scala has then looked for an implicit function that can accept a `Double` and return a `String`. The `implicit def doubleToString(...)` meets that signature.

Task

- Try that example out in a Scala project to see it working for yourself.

Implicit Classes

We are now going to look at what Implicit Classes are in Scala.

As we have seen above, we can also use the implicit keyword for classes. The signature for

this is `implicit class ClassName() {}`. As you can see all be have done is prepend the implicit keyword to our class signature.

There are rules that must be met when implementing implicit classes:

- They must defined inside another `trait` / `class` / `object`
- They can only take one non-implicit argument in their constructor
- There may not be any method, member or object in scope with the same name as the implicit class
    - This means it **cannot** be a case class

Following the rules above, the following declaration is a valid implicit class:

```
trait Implicits {

    implicit class NumberImplicit(n : Int) {

        def double() = n * 2

        def add(m : Int) = n + m

        def subtract(m : Int) = n - m
    }
}
```

The `NumberImplicit` class accepts *one* argument in it's constructor, is within a `trait` and does not have nay members, methods, values or objects in scope of the same name!

So what does our implicit class do?

We can use our implicit class in the following way:

```
2 double
3 add 4
5 subtract 2
```

Are you wondering why we haven't had to instantiate our NumberImplicit class? i.e. `new NumberImplicit()`.

We haven't had to instantiate this as Scala will provide an instance for us at runtime, and in the examples above we have called a function that is available implicitly by the class. For instance, the `add()` function is used from within our implicit class as it's available in scope.

The following two calls are equivalent:

```
2 add 4
2.add(4)
```

Task

- Implement the above solution and try it out
- Implement a `multiply()` method which will accept one argument and return an Int.

Further Reading

Implicit classes

Recap

We have look at how to declare and use implicit values, how to do type conversions with implicit functions and how to extend functionality with implicit classes, even for primitive types such as `Int`.

Resources

- Implicit classes