Scala fundamentals - Currying and Partially applied Parameters (part 13)

Aims

We will be covering the following topics:

- Currying
- Partially applied parameters

Prerequisites

You have completed all previous 12 parts of the Scala fundamentals.

Lesson steps

In this session, we're going to be looking at two new concepts: currying and partially applied functions.

Currying

In simple terms, currying is the process of transforming a function that takes multiple arguments into a sequence of functions that each have only a single parameter. An example would be taking this function, which has two parameters…

```
def add(x: Int, y: Int): Int = {
  x + y
}
```

…and transforming it into this:

```
def add(x: Int)(y: Int): Int = {
  x + y
}
```

This is actually short-hand for the following:

```
def add(x: Int): (Int => Int) = {
  (y: Int) => {
    x + y
  }
}
```

Further reading

Currying | Scala documentation

Although our functions may result in the same output, there are differences between them. Both functions have, for example, different types.

- Our original `add(x: Int, y: Int)` function is of type `(Int, Int) => Int`.
- Our curried `add(x:Int)(y:Int)` function is of type `Int => (Int => Int)`.

How do we work with this function? If we wanted to simply call our curried function, we could do this:

```
val onePlusFive = add(1)(5) // 6
```

In this example, the value `1` is passed to the first function in our curried sequence of functions. The value `5` is passed to the second function in our curried sequence of functions.

Instead of providing parameters to both of our functions, however, we can opt to only provide one. This will give us a **partially applied function**:

```
val addFour = add(4)_
val twoPlusFour = addFour(2) // 6
assert(onePlusFive == twoPlusFour) // true
```

In the example above, when write `add(4)_` our underscore becomes a placeholder for the second function in our sequence. The value `addFour` becomes a function in itself, which we can then call as `addFour(2)`.

Further reading

Partially applied functions

Through currying, we've started to see how we might implement and use partially applied functions. Now, let's get into this topic in more detail. We'll start by declaring an anoymous function:

```
scala> val sum = (a: Int, b: Int, c: Int) => a + b + c
sum: (Int, Int, Int) => Int = <function3>
```

Instead using currying, we are going to partially apply the function to create a *function value*.

```
scala> val f = sum(1, 2, _: Int)
f: Int => Int = <function1>
```

In the example above, `f` becomes a function of type `Int => Int`. The function has been partially applied, in that we have supplied values for the parameters `a` and `b` - but not `c`. We've specified that the final missing parameter of `sum` is `_: Int`.

Having defined our partially applied function, we can call `f` like so:

```
f(10) // 13
```

When we create our partially applied function, we can choose to apply - or not - whichever parameters we want.

```
scala> val f = sum(1, _: Int, 2)
f: Int => Int = <function1>

scala> val g = sum(_: Int, 1, 2)
g: Int => Int = <function1>
```

Above, we have altered which of the parameters is not applied. In these examples, the resulting functions are all basically the same: two of the parameters are already applied, and one is not.

Further reading

- How to use partially applied functions in Scala
- Currying in the Real World

Recap

In this session, we've looked at the concept of currying. We've also looked at how we can define partially applied functions.

Further reading

The Neophyte's Guide to Scala Part 11: Currying and Partially Applied Functions

Resources

- Currying | Scala documentation
- Currying Functions in Java & Scala
- How to use partially applied functions in Scala
- Currying in the Real World
- The Neophyte's Guide to Scala Part 11: Currying and Partially Applied Functions