Greetings - Scala fundamentals (part 7)

Aims

We're going to continue working on our `GreeterApplication` and explore some more concepts of functional programming.

- Anonymous Functions
- Mapping over Collections
- Higher-Order Functions
- Partial Functions
- Function Composition

Prerequisites

You have completed Greetings - Scala fundamentals (part 6).

Lesson steps

Anonymous functions

In this lesson, we're going to explore some of the more functional ways that we can approach our application.

Our `Person` class has a collection of `[BankAccount]`. If we wanted to work out the total balance of these accounts, we might be tempted to use the for loops, which we learned about during our flow control lesson, to achieve this. This isn't a good idea, because it would mean that we had to create a *mutable variable* to add each of the account balances to.

We should look to implement it in an *immutable* manner.

We can call a method on our `List[BankAccount]` which *will* iterate over each element (bank account) in the list. For each iteration it will pass in an *accumulator*. This *accumulator* is an **immutable value** which will be passed through with the running balance at that position in the list.

The signature for this method is:

```
list.foldLeft(startingValue)(f: (accumulator, account) => ...)
```

Lets break this down.

- We first start with our `list` and then call the `foldLeft` method. This will iterate over the `List[BankAccount]` as long as the list is not empty.
- Inside our first set of `()` parenthesis, we pass in our starting value. In this instance `0.00`.
- You will notice we have a second set of `()` parenthesis. This is going to be where we can pass our *anonymous* function as an argument.

Tip

> an anonymous function is a function that has no name assigned

Further reading
Scala Anonymous Functions

Our example when implemented will be:

```
bankAccounts.foldLeft(0.00)((acc, account) => acc + account.balance)
```

We could have implemented this by using a method to add the two values:

```
def add(acc : Double, account : BankAccount) = acc + account.balance

bankAccounts.foldLeft(0.00)((acc, account) => add(acc, account))
```

A nicer implementation is to have assigned our *anonymous* function to a value rather than creating a method:

```
val add = (acc : Double, account : BankAccount) => acc + account.balance

bankAccounts.foldLeft(0.00)(add)
```

This will tell the Scala compile to create an anonymous function on each iteration that will accept two arguments, and return a `Double` as the return type. This is the same signature that is provided in the first example `(acc, account) => acc + account.balance`.

Run your application and see the output.

> An anonymous function is applied to each element in a collection. This is normally presented in the following syntax:
> `(arguments : T ...) => expression`

> As with `def` you can accept multiple arguments such as `(one : T, two : T) => ...`
> `T` here is the data type of the argument, again this could be a primitive type or a trait, abstract class, etc.

Further reading
A Tour of Scala: Anonymous Function Syntax | The Scala Programming Language

There is an even simpler implementation we could have chosen and that is:

`def totalBalance = bankAccounts.map(_.balance).sum`.

Run your application and see the output.

Grab a cup of tea! We will break this down as we're introducing a number of concepts here.

Mapping

Now, we are introducing the `map` method. You **will** come across this method a lot in Scala. It has a number of uses:

Tip

> A predicate is a filter which is used to determine if a condition is met, you have used a predicate in the form of an If condition.

1. Iterating over each element in a collection and applies a predicate function on that element. It creates a new collection with the result of the predicate function applied to each and every element in the collection
2. Operating on `Option[T]` to pull out the value `T` if the Option has a value. More on this later!
3. Waiting for a `Future[T]` to complete and return the value of `T` when the value is resolved. Also known as a *callback*. Again, more on that later!

Further reading
Scala Tutorial - Map Function Example

Tip

> Be careful not to confuse `map()` method with the collection type `Map()`

Since we now know that calling `map()` on an `Iterable` collection will apply that function to each and every element in that list we can now understand what the `_` underscore operator is doing.

Similar to the `map` method, you will come across the `_` operator in Scala **a lot**!

Further reading
What are all the uses of an underscore in Scala? - Stack Overflow

In this instance, we have chosen to use the `_` operator as a *short hand* syntax for `(x => x)`.
In the long form, we are creating an *immutable* val `x` for each iteration of our collection; of which the value of `x` would be a `BankAccount`.
The short form is telling Scala that we want to apply the expression that is after the `_` operator to every element in the collection.

In this instance `def totalBalance = bankAccounts.map(_.balance).sum` we are saying that we want to create a **new** `List` which will contain the balance of each account. This creates a `List[Double]`.

i.e. `List(1000.00, 1000.00)`

Then, we have called the `sum()` method on the `List[Double]` which will *iterate* over each `Double` and total the values.

As a result, we will be returned with a single `Double` with the total value of £2000.00 in this instance.

This is how it could have been done long form with all the types annotated:

```
def totalBalance = {
  val balances : Seq[Double] = bankAccounts.map(_.balance)
  balances.sum
}
```

Higher-order functions

Let's continue on the theme of anonymous functions, however, now we're going to discuss the term higher-order functions.

> Scala allows the definition of higher-order functions. These are functions that *take other functions as parameters*, or whose *result* is a function.

An example definition of a method that takes a function as an argument is

```
def sumAndMultiplyBy(f: Double => Double) = ...
```

Here we can see we are declaring a function by the `def` keyword. We are then stating that it's parameter list is a single argument which is `f: Double => Double`. This states we have created an anonymous function names `f` which it's argument is a single `Double` which returns a `Double`.

An example of this implemented would be:

```
def sumAndMultipleBy(f: Double => Double) = bankAccounts.map(x => f(x.balance)).sum
```

In the example above, the parameter list of our method is `f: Double => Double`. This means that it will accept a function that will take a `Double` and return a `Double`.

Inside your `GreeterApplication` object implement the following:

```
Prompt.reply(person.sumAndMultipleBy(_ * 2))
```

This will invoke our higher-order function, we have then provided an anonymous function which will take each element, using the `_` operator signifying the element at the current position, then multiply this by two using the `*` operator.
Our `sumAndMultipleBy` function will then apply this to each element in the list and provide a number in each iteration as `_`. Once this is complete we then `sum` the total value again.

Run your application to test this out!

> Don't forget to commit your code.

Partial Functions

We are now going to introduce the concept of a partial function. What on earth is a partial function, how can it be partial when surely it's 'whole'?

Well, imagine the following problem. You want to create a function that will only work for a subset of all the possible input values. Likewise, you want to define functions that deal with a subset of these inputs and compose them together to handle every case.

This is when we can use a `PartialFunction[-A, +B]`.

Let's look at a real world example:

```
val divide = (x : Int) => 42 / x
```

Here we have declared an anonymous function which is assigned to the val divide.

We can then invoke this by doing `divide(0)`. This code will return us a `ArithmeticException`, this is because we cannot divide 42 by 0.
We could typically handle this exception to safely handle when we get an ArithmeticException, which we will cover in more depth later. However, we can define a partial function which will only work for a subset of all the values given, in this instance, **not** zero.

```
val divide = new PartialFunction[Int, Int] {
  def apply(x : Int) = 42 / x
  def isDefinedAt(x : Int) = x != 0
}
```

All `PartialFunction`s must implement an `apply` and `isDefinedAt` method. Let's see what each is doing:

- `apply()` is the function that gets called when passing in a value to the function call. i.e. `divide(0)` is the same as `divide.apply(0)`. This will calculate the value and return an `Int`.
- `isDefinedAt` is the function which checks if the input can be accepted by the function. i.e. can we pass `0` as a parameter to the function. This returns us a `Boolean` value.

You will notice we declared the `PartialFunction` to accept an `Int` as it's input, and an `Int` as it's return type. Denoted by the type parameters `PartialFunction[Int, Int]`.

Tip

> The syntax here is quite different to what we have seen before, we have typically declared an argument and return type in the following way for a function:
>
> ```
> def doSomething(input : Int) : Int
> ```

We can then call the divide function in the following way:

```
divide.isDefinedAt(1) // returns true
divide.isDefinedAt(0) // returns false

// if allowed, then run the function
if(divide.isDefinedAt(1)) divide(1) // returns 42
```

This isn't all you can do with `PartialFunction`s. We will look how we can use them with collections.

We can re-write the top code into a more concise syntax:

```
val divide : PartialFunction[Int, Int] = {
  case d : Int if d != 0 => 42 / d
}
```

The above code achieves the same result, except we have used the `case` keyword instead.

If we ran the above example on a collection using the map function, this will return a `MatchError`. This is because we do not have a `case` in which we can handle the input value.

For example:

```
List(0, 1, 2, 3, 4).map(divide)
```

The above would fail as there is not a `case` statement inside of the partial function that will handle zero.

We can correct this in the following way:

```
List(0, 1, 2, 3, 4).collect(divide)
```

This does not throw a `MatchError`, this is because the `collect` function is implemented to call the `isDefinedAt()` function on each element first, to see if it can handle the input. If not, it discards it from the collection and continues.

Tip

> A `PartialFunction` is a subtype of `Function`.
> Therefore, whenever we need to pass in a function as an argument to a function (higher-order function), we can also supply a `PartialFunction` instead, using composition to handle a range of inputs.

Function composition

Let's take another look at using a `PartialFunction` as well as `collect`, we're going to return a `List` of `Int` where the numbers are even.

```
val isEven : PartialFunction[Int, String] = {
  case x: Int if x % 2 == 0 => s"$x is even"
}

val numbers = 1 to 5
val evenNumbers = numbers collect isEven
```

In the code above, we have created a `PartialFunction` which accepts an `Int` and returns a `String`. Inside the `case` statement, we are checking if the number is even `x % 2 == 0`. If this expression is true, we return a string. i.e. `2 is even`.

We have then generated a `List[Int]` using the `Range` function `to`. This gives us `List(0,1,2,3,4,5)`.
We are then calling `collect` on the list passing in our function. If we have of used `map` instead of `collect`, this would have thrown a `MatchError` because we can't handle odd numbers.

We can fix this by using function composition.

```
val isEven : PartialFunction[Int, String] = {
  case x: Int if x % 2 == 0 => s"$x is even"
}
```

```
val isOdd : PartialFunction[Int, String] = {
  case x : Int if x % 2 == 1 => s"$x is odd"
}

val isEvenOrOdd = isEven orElse isOdd

val numbers = 1 to 5
val allNumbers : IndexedSeq[String] = numbers map isEvenOrOdd
```

Here we have defined another `PartialFunction` which we have assigned to `isOdd`. This Partial functions accepts an odd number and returns a `String`.

We have then created a new function by chaining two together, `isEven orElse isOdd`. This allows us to compose functions together out of smaller functions.

`PartialFunction` also has a `compose` and `andThen` function, let's take a look at these.

```
val double = (n : Int) => n * 2
val plusOne = (n : Int) => n + 1

val doublePlusOne = double andThen plusOne

val numbers = List(1, 2, 3, 6, 8, 12, 45, 68)
numbers.map(doublePlusOne)
```

Here we have defined two anonymous functions, one which doubles the number, and the other which adds one. I have then created another function out of our two anonymous functions by using composition. By using the `andThen` function, I had said for our new function `doublePlusOne`, taking each number in the list, double the value then add one to the result. This would give us `List(3, 5, 7, 13...)`.

If we changed `andThen` to be `compose`, we would be given `List(4, 6, 8, 14...)`. This is because it would be the same as writing:

```
def double(n : Int) = n * 2
def plusOne(n : Int) = n + 1

def doublePlusOneComposed(n : Int) = double(plusOne(n))
numbers.map(doublePlusOneComposed)
```

This would call the `doublePlusOneComposed` function on each element in the list, first calling `plusOne()` on each element, and pass that value to the `double()` function.

Take some time to review this section in depth.

Recap

In this session, we have covered:

- Anonymous Functions
- Mapping
- Higher-Order Functions
- Partial Functions

Resources

- Scala Tutorial - Map Function Example
- What are all the uses of an underscore in Scala? - Stack Overflow
- Scala Anonymous Functions
- A Tour of Scala: Anonymous Function Syntax | The Scala Programming Language