docs.scala-lang.org

Higher-order Functions

5-7 minutes

Higher order functions take other functions as parameters or return a function as a result. This is possible because functions are first-class values in Scala. The terminology can get a bit confusing at this point, and we use the phrase “higher order function” for both methods and functions that take functions as parameters or that return a function.

In a pure Object Oriented world a good practice is to avoid exposing methods parameterized with functions that might leak object’s internal state. Leaking internal state might break the invariants of the object itself thus violating encapsulation.

One of the most common examples is the higher-order function `map` which is available for collections in Scala.

```
val salaries = Seq(20000, 70000, 40000)
val doubleSalary = (x: Int) => x * 2
val newSalaries = salaries.map(doubleSalary) //
List(40000, 140000, 80000)
```

`doubleSalary` is a function which takes a single Int, `x`, and returns `x * 2`. In general, the tuple on the left of the arrow `=>` is a parameter list and the value of the expression on the right is what gets returned. On line 3, the function `doubleSalary` gets applied to each element in the list of salaries.

To shrink the code, we could make the function anonymous and pass it directly as an argument to map:

```
val salaries = Seq(20000, 70000, 40000)
val newSalaries = salaries.map(x => x * 2) //
List(40000, 140000, 80000)
```

Notice how `x` is not declared as an Int in the above example. That’s because the compiler can infer the type based on the type of function map expects (see Currying). An even more idiomatic way to write the same piece of code would be:

```
val salaries = Seq(20000, 70000, 40000)
val newSalaries = salaries.map(_ * 2)
```

Since the Scala compiler already knows the type of the parameters (a single Int), you just need to provide the right side of the function. The only caveat is that you need to use _ in place of a parameter name (it was `x` in the previous example).

Coercing methods into functions

It is also possible to pass methods as arguments to higher-order functions because the Scala compiler will coerce the method into a function.

```
case class WeeklyWeatherForecast(temperatures:
Seq[Double]) {

  private def convertCtoF(temp: Double) = temp *
1.8 + 32

  def forecastInFahrenheit: Seq[Double] =
temperatures.map(convertCtoF) // <-- passing the
method convertCtoF
}
```

Here the method `convertCtoF` is passed to the higher order function `map`. This is possible because the compiler coerces `convertCtoF` to the function `x => convertCtoF(x)` (note: `x` will be a generated name which is guaranteed to be unique within its scope).

Functions that accept functions

One reason to use higher-order functions is to reduce redundant code. Let's say you wanted some methods that could raise someone's salaries by various factors. Without creating a higher-order function, it might look something like this:

```
object SalaryRaiser {

  def smallPromotion(salaries: List[Double]):
List[Double] =
    salaries.map(salary => salary * 1.1)

  def greatPromotion(salaries: List[Double]):
List[Double] =
    salaries.map(salary => salary *
math.log(salary))

  def hugePromotion(salaries: List[Double]):
List[Double] =
    salaries.map(salary => salary * salary)
}
```

Notice how each of the three methods vary only by the multiplication factor. To simplify, you can extract the repeated code into a higher-order function like so:

```
object SalaryRaiser {
  private def promotion(salaries: List[Double],
promotionFunction: Double => Double): List[Double]
=
    salaries.map(promotionFunction)

  def smallPromotion(salaries: List[Double]):
List[Double] =
    promotion(salaries, salary => salary * 1.1)

  def greatPromotion(salaries: List[Double]):
List[Double] =
    promotion(salaries, salary => salary *
math.log(salary))

  def hugePromotion(salaries: List[Double]):
List[Double] =
    promotion(salaries, salary => salary * salary)
}
```

The new method, `promotion`, takes the salaries plus a function of type `Double => Double` (i.e. a function that takes a Double and returns a Double) and returns the product.

Methods and functions usually express behaviours or data transformations, therefore having functions that compose based on other functions can help building generic mechanisms. Those generic operations defer to lock down the entire operation behaviour giving clients a way to control or further customize parts of the operation itself.

Functions that return functions

There are certain cases where you want to generate a function. Here's an example of a method that returns a function.

```
def urlBuilder(ssl: Boolean, domainName: String):
(String, String) => String = {
  val schema = if (ssl) "https://" else "http://"
  (endpoint: String, query: String) =>
s"$schema$domainName/$endpoint?$query"
}

val domainName = "www.example.com"
def getURL = urlBuilder(ssl=true, domainName)
val endpoint = "users"
val query = "id=1"
val url = getURL(endpoint, query) //
"https://www.example.com/users?id=1": String
```

Notice the return type of urlBuilder `(String, String) => String`. This means that the returned anonymous function takes two Strings and returns a String. In this case, the returned anonymous function is `(endpoint: String, query: String) => s"https://www.example.com/$endpoint?$query"`.