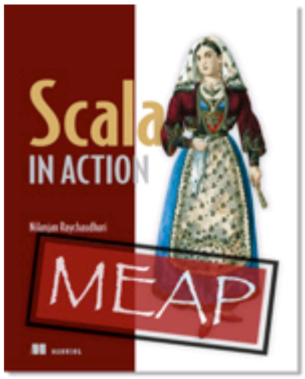
Building your own function objects

An article from



Scala in Action
EARLY ACCESS EDITION

Nilanjan Raychaudhuri
MEAP Release: March 2010
Softbound print: Spring 2011 | 525 pages
ISBN: 9781935182757

A *function object* is an object that you can use as a function. That doesn't help you much, so why don't I show you an example? Here, we create a function object that wraps a `foldLeft` method:

```
object foldl {
  def apply[A, B](xs: Traversable[A], defaultValue: B)(op: (B, A) => B) =
            (defaultValue /: xs)(op)
}
```

Now we can use our `foldl` function object like any function:

```
scala> foldl(List("1", "2", "3"), "0") { _ + _ }
res0: java.lang.String = 0123

scala> foldl(IndexedSeq("1", "2", "3"), "0") { _ + _ }
res24: java.lang.String = 0123

scala> foldl(Set("1", "2", "3"), "0") { _ + _ }
res25: java.lang.String = 0123
```

To treat an object as a function object, all you have to do is declare an `apply` method. In Scala, `<object>(<arguments>)` is syntactic sugar for `<object>.apply(<arguments>)`. Now, because we defined the first parameter as `Traversable`, we can pass `IndexedSeq`, `Set`, or any type of collection to our function.

The expression `(defaultValue /: xs)(op)` might look a little cryptic, but the idea isn't to confuse you but to demonstrate the alternative syntax for `foldLeft`, `/:`. When an operator ends with `:`, then the right-associativity kicks in.

When declaring function objects, it's a good idea to extend one of the `Function` traits defined in the Scala library. In Scala, the `Functon1` trait is declared as follows:

```
trait Function1[-T1, +R] extends AnyRef {
  def apply(v: T1): R
}
```

Here, `1` stands for "function with one parameter." Similarly, Scala defines a function trait for two or more parameters. In the following example, we're building an increment function that takes an integer parameter and increments its value with `1`:

```
object ++ extends Function1[Int, Int]{
  def apply(p: Int): Int = p + 1
}
```

The shorthand and equivalent implementation of this function would be:

```
val ++ = (x: Int) => x + 1
```

There's one more way we could define our function object, and that's using the alternative notation of `function =>`:

```
object ++ extends (Int => Int) {
  def apply(p: Int): Int = p + 1
}
```

In the last case, we're using the shorthand notation of `Function1`, `Int => Int`. We've used similar syntactic notation when declaring function parameters. In Scala, there's always another way to define a construct, and it's up to you to decide which one you like the most. We can now pass the function objects that we created as parameters to other functions. Here, we're invoking our `map` function with our new `++` function:

```
scala> map(List(10, 20, 30), ++)
res1: List[Int] = List(11, 21, 31)
```

This is the same as invoking with an anonymous function:

```
scala> map(List(10, 20, 30), (x: Int) => x + 1)
res2: List[Int] = List(11, 21, 31)
```

It's also the same as passing the `Function` trait:

```
scala> map(List(10, 20, 30), new Function1[Int, Int] {
              def apply(p: Int) = p + 1
          })
res3: List[Int] = List(11, 21, 31)
```

One drawback of using the `Function` trait is that you're restricted to only one `apply` method. When passing an existing function (not a function object) as a parameter, Scala creates a new anonymous function object with an `apply` method, which invokes the original function. This is called *eta-expansion*.

One interesting subtrait of the `Function1[-P, +R]` trait is `PartialFunction`, which allows you to define a function that's not defined for all `P` and allows you compose with other functions.

Function1 is also defined as Function

Because the `Function1` trait is used so often, Scala defines a type alias called `Function`. You won't find any reference of this trait in scaladoc because this type alias is defined inside the `Predef` class as follows:

```
type Function[-A, +B] = Function1[A, B]
```

`type` is a keyword in Scala, and it's used to type variables. It's similar to `typedef` in C.

Another helpful use of type variables is as a representation of a complicated type.

```
type MILS = Map[Int, List[String]]

val mapping: MILS = Map(

      1 -> List("one", "uno"), 2 -> List("two", "dos"))
```

Why is the function1 trait contravariant for parameter and covariant for return type?

If you're interested in why it's both covariant and contravariant at the same time, please read on. To answer the question, let's consider what would happen when it's opposite; for instance, covariant for parameter and contravariant for return value. Now imagine we have a covariant parameter, as in the following example:

```
val addOne: Any => Int = { x: Int => x + 1 }
```

Because `Int` is subtype of `scala.Any`, the covariant parameter should allow the preceding code to compile. But, the loophole in this code is that we can now invoke `addOne` with any type of parameter as long as it's a subtype of `scala.Any`. This could cause all sorts of problems for a function that expects only `Int`. Scala, being a type-safe language, doesn't allow you to do that. The only other option we have is to declare the parameter type as invariant, but that would make the `Function1` trait inflexible. Now if we make the return type a contravariant, then we could declare a method like the following:

```
val asString: Int => Int = { x: Int => (x.toString: Any) }
```

This code should also be valid because `Any` is a super type of `Int`, and a contravariant allows us to go from a narrower type to a more generic type. The following piece of code should also be valid, right?

```
asString(10) + 20
```

But we end up adding `20` to a string value, and that could be problematic. Scala's strong type system implementation stops you from making this kind of mistake when dealing with parameter types and return types. To have a flexible and type-safe `Function1` trait, the only possible implementation would be to have the parameter contravariant and the return type covariant.

Summary

The knowledge of type parameterization helps us in exploring type-variance concepts and the type-safety features of Scala. It's important to understand this concept for building generic, type-safe, reusable components in Scala. We explored the usage and importance of higher-order functions such as `map` and how they help the Scala library, especially the collection library, provide rich, useful APIs. Using higher-order functions, you can easily encapsulate common programming idioms.