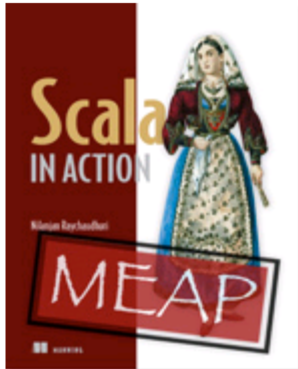
MANNING PUBLICATIONS

Defining functions

An article from



[Scala in Action](#)
EARLY ACCESS EDITION

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

This article is taken from the book Scala in Action*. The author explains how to define a function and pass functions as a parameter to another function.*

Tweet this button! (instructions [here](#))

Get **35% off** any version of *Scala in Action* with the checkout code **fcc35**.
Offer is only valid through www.manning.com.

Functions are building blocks in Scala. In this article, we're going to explore that topic a little. To define a function in Scala, you need to use the `def` keyword followed by the method name, parameters, optional return type, `=`, and the method body. Figure 1 shows the syntax of the Scala function declaration.

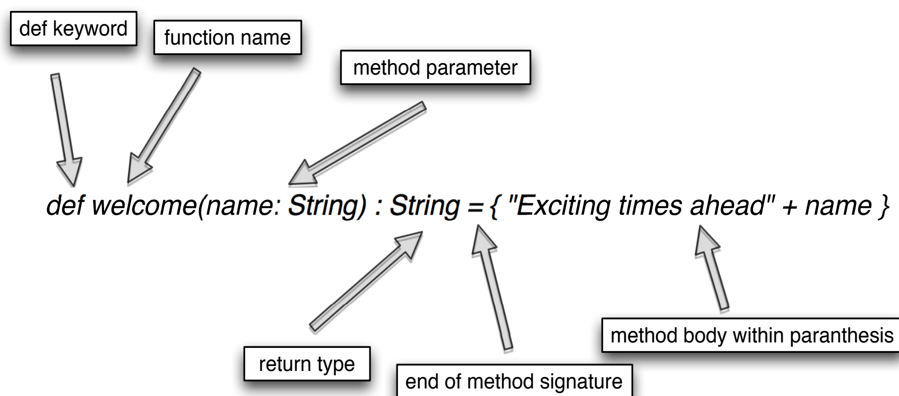


Figure 1 The syntax of the Scala function declaration

You need to use a colon (:) to separate the parameter list from the return type. In case of multiple parameters, they are separated by comma (,). The equals sign (=) is used as a separator between the method signature and the method body.

Let's drop the parameter for the time being; we'll come back to it a little later. We'll create a Scala function without parameters.

```
scala> def myFirstMethod():String = { "exciting times ahead" }
myFirstMethod: ()String
```

The return type of a Scala function is optional because Scala infers the return type of a function automatically. There are situations where it doesn't work, but we'll worry about that later. So, we improve our `myFirstMethod` a little bit by removing the return type.

```
scala> def myFirstMethod() = { "exciting times ahead" }
myFirstMethod: ()java.lang.String

scala> myFirstMethod()
res6: java.lang.String = exciting times ahead
```

The significance of = after the method signature is not only to separate the signature from the method body but also to tell the Scala compiler to infer the return type of your function. If you omit that, Scala won't infer your return type.

```
scala> def myFirstMethod() { "exciting times ahead" }
myFirstMethod: ()Unit

scala> myFirstMethod()
```

In this case, when you invoke the function using the function name and `()`, you'll get no result. If you look at the REPL output, you'll notice that the return type of our function is no longer `java.lang.String`; it's `Unit`. `Unit` in Scala is like `void` in Java, and it means that the method doesn't return anything.

NOTE

Scala type inference is quite powerful, but use it carefully. For example, if you're creating a library and plan to expose your functions as a public API, it's a good practice to specify the return type for the users of the library. In any case, if you think it's not clear from the function what its return type is, either try to improve the name so that it communicates its purpose better or specify the return type.

Our `myFirstMethod` is simple; it returns the string `"exciting times ahead"`, and when you have a function like that, you also drop the curly braces from the method body.

```
scala> def myFirstMethod() = "exciting times ahead"
myFirstMethod: ()java.lang.String
```

If you invoke the function, you'll get the same result. In Scala, it's always possible to take out unnecessary syntax noise from the code. Because we aren't passing any parameters, we could take out our unused `()` from the declaration, and it will almost look like a variable declaration, except that instead of using `var` or `val` we're using `def`.

```
scala> def myFirstMethod = "exciting times ahead"
myFirstMethod: java.lang.String
```

When calling the function, you could also lose the parentheses:

```
scala> myFirstMethod
res17: java.lang.String = exciting times ahead
```

Now, let's come back to parameters. We have a function called `max` that takes two parameters and returns the one that's the greater of them.

```
scala> def max(a: Int, b: Int) = if(a > b) a else b
max: (a: Int,b: Int)Int

scala> max(5, 4)
res8: Int = 5

scala> max(5, 7)
```

```
res9: Int = 7
```

By now, you probably have figured out that specifying `return` is optional in Scala. You don't have to specify the `return` keyword to return anything from the function. The value of the last expression will be returned from the function. In the previous case, `if` condition evaluates to `true`, then `a` is the last expression that get executed, so `a` will be returned; otherwise `b` will be returned. Even though the return type is optional, you do have to specify the type of the parameters when defining functions. Scala type inference will figure out the type of parameters when you invoke the function but not during function declaration.

If you have a background in Haskell, OCaml, or any other type of inferred programming language, then the way Scala parameters are defined would feel a bit weird. The reason is that Scala doesn't use the Hindley-Milner algorithm to infer type; instead Scala's type inference is based on declaration-local information, also known as local type inference. Type inference is out of scope for this book, but if you're interested you can read about the Hindley-Milner type inference algorithm[1] and why it's useful.[2]

Sometimes it becomes necessary to create a function that will take an input and create a `List` from it. But, the problem is that you can't determine the type of input yet. Someone could use your function to create a `List` of `Int`, and another person could use it to create a `List` of `String`. In cases like these, you create a function in Scala by parameterized type. The parameter type will be decided when you invoke the function.

```
scala> def toList[A](value:A) = List(value)
toList: [A](value: A)List[A]

scala> toList(1)
res16: List[Int] = List(1)

scala> toList("Scala rocks")
res15: List[java.lang.String] = List(Scala rocks)
```

When declaring the function, we denote the unknown parameterized type as `A`. Now, when our `toList` is invoked, it replaces the `A` with the type of the given parameter. In the method body, we create an instance of immutable `List` by passing the parameter and, from the REPL output, it's clear that `List` is also using a parameterized type.

If you're a Java programmer, then you'll find lots of similarities between Java generics and Scala parameterized types. The only difference to remember for now is that Java uses angle brackets (<>) and Scala uses square brackets (`[]`). Another Scala convention for naming the parameterized types is that they normally start at `A` and go up to `Z`, as necessary. This contrasts with the Java convention of using `T`, `K`, `V`, and `E`.

Function literals

In Scala, you can also pass functions as a parameter to another function and, most of the time in those cases, we provide inline definition of the function. This passing of functions as a parameter is sometimes loosely called *closure* (passing a function isn't always necessarily closure). Scala provides a shorthand way to create a function in which you write only the function body; they're called *function literals*. Let's put that to a test. In this test, I want to add all the elements of a `List` using function literals. This demonstrates a simple use of function literals in Scala. Here, we're creating a `List` of even numbers:

```
scala> val evenNumbers = List(2, 4, 6, 8, 10)
evenNumbers: List[Int] = List(2, 4, 6, 8, 10)
```

Now, to add all the elements of `List` (`scala.collection.immutable.List`), we could use the `foldLeft` method defined in `List`. The `foldLeft` method takes two parameters: an initial value and a binary operation. It applies the binary operation to the given initial value and all the elements of the list. It expects the binary operation as a function that takes two parameters of its own to perform the operation, which in our case will be addition. So, if we can create a function that will take two parameters and add them, then we'll be finished with the test. The `foldLeft` function will call our function for every element in the `List` starting with the initial value.

```
scala> evenNumbers.foldLeft(0) { (a: Int, b:Int) => a + b }
res19: Int = 30
```

In this case, the function `(a: Int, b:Int) => a + b` is called an *anonymous* function or a function without a predefined name. We can improve our function by taking advantage of Scala's type inference.

[1] http://en.wikipedia.org/wiki/Type_inference#Hindley-Milner_type_inference_algorithm
[2] www.codecommit.com/blog/scala/what-is-hindley-milner-and-why-is-it-cool

```
scala> evenNumbers.foldLeft(0) { (a, b) => a + b }
res20: Int = 30
```

Usually, we have to specify the type of the parameter for top-level functions because Scala can't infer the parameter types when declared but, for anonymous functions, Scala inference can figure out the type from the context. In this case, we're using a list of integers and `0` as our initial value and, based on that, Scala could easily infer the type of `a` and `b` as integer. Scala allows us to go even further with our anonymous function, where we can drop the parameters and only have the method body to make it function literal. But, in this case, the parameters will be replaced by underscores (_). An underscore has a special meaning in Scala and, in this context, it's a placeholder for a parameter; in our case, we'll use two underscores:

```
scala> evenNumbers.foldLeft(0) { _ + _ }
res21: Int = 30
```

Each underscore represents a parameter in our function literal. In Scala, underscores can be used in various places, and their meaning is determined solely by the context and where they're used. Sometimes, it gets a little confusing, so always remember that the value of the underscore is based on where it's being used. Function literals are a common idiom in Scala, and you'll find occurrences of them in Scala libraries and codebases. In the following example, `_.isUpper` is a function literal.

```
val hasUpperCase = name.exists(_.isUpper)
```

In this case, we're invoking the given function literals for each character in the `name` string; when it finds an uppercase character, it will exit.

Using Scala closure and first-class functions: an example

A *closure* is any function that closes over the environment in which it's defined. For example, closure will keep track of any variable changes outside the function that's referred to inside the function.

In our example, we'll try to add support for the word `break`. Scala doesn't have `break` or `continue`. Once you get comfortable with Scala, you won't miss them because Scala's support of functional programming style reduces the need for having `break` or `continue`. But, let's assume you have a situation where you think having a `break` would be helpful. Scala is an extensible programming language, so let's extend it to support `break`.

We'll use the Scala exception-handling mechanism to implement `break` in Scala. Throwing an exception will help us to break the sequence of execution, and the `catch` block will help us reach the end of the call. Because `break` isn't a keyword, we can use it to define our function that will throw an exception.

```
def break = new RuntimeException("break exception")
```

If you've used exception handling in Java, C#, or Ruby, it should be easy to follow. Now, let's create the main function that will take the operation that needs a breakable feature. We'll make it obvious and call it `breakable`:

```
def breakable(op: => Unit) { … }
```

What's this `op: => Unit`? The special right arrow (=>) lets Scala know that the `breakable` function expects a function as a parameter. The right side of the => defines the return type of the function; in this case, it's `Unit` (similar to Java `void`) and `op` is the name of the parameter. Since we haven't specified anything on the left side of the arrow, it means that the function we're expecting as a parameter doesn't take any parameter for itself. But, if you expect a function parameter that takes two parameters, like `foldLeft`, you have to define it as follows:

```
def foldLeft(initialValue: Int, operator: (Int, Int) => Int)= { ... }
```

The `breakable` function that we declared takes a no-parameter function and returns `Unit`. Now, using these two functions, we could simulate the break. Let's look at an example function that needs to break when the environment variable `SCALA_HOME` isn't set; otherwise, it must do the work:

```
def install = {
  val env = System.getenv("SCALA_HOME")
  if(env == null) break
  println("found scala home lets do the real work")
}
```

Now inside the `breakable` function we need to catch the exception that will get raised when `break` is called from the `install` function.

```
try {
    op
} catch { case _ => }
```

That's it. Listing 1 holds the complete code.

Listing 1 `breakable`, `break`, and `install` functions

```
val breakException = new RuntimeException("break exception")

def breakable(op: => Unit) {
  try {
    op
  } catch { case _ => }
}

def break = throw breakException

def install = {
  val env = System.getenv("SCALA_HOME")
  if(env == null) break
  println("found scala home lets do the real work")
}
```

To invoke the `breakable` function, pass the method name that needs a `breakable` feature, like `breakable(install)`, or you could inline the `install` function and pass it as a closure.

```
breakable {
  val env = System.getenv("SCALA_HOME")
  if(env == null) break
  println("found scala home lets do the real work")
}
```

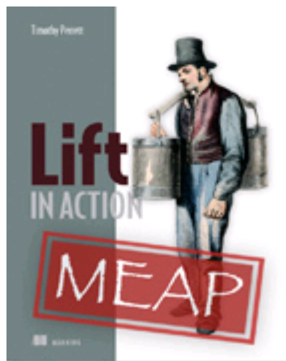
In Scala, if the last argument of a function is of `function` type, then you can pass it as closure. This syntax sugar is useful in creating domain-specific languages. In the next chapter we'll look into how closures are converted into objects; remember, everything in Scala is an object.

Scala already provides `breakable` as part of the library. Look for `scala.util.control.Breaks`. You should use `Breaks` if you have a need for a break. Again, I'd argue that once you look into functional programming with Scala in detail, you'll probably never have a need for `break`.

Summary

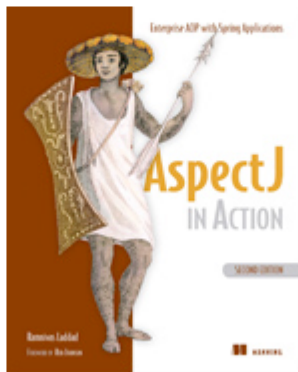
In this article we discuss defining functions. We said that, to define a function in Scala, you need to use the `def` keyword followed by the method name, parameters, optional return type, `=`, and the method body. Scala provides a shorthand way to create a function in which you write only the function body; they're called *function literals*. A *closure* is any function that closes over the environment in which it's defined.

Here are some other Manning titles you might be interested in:



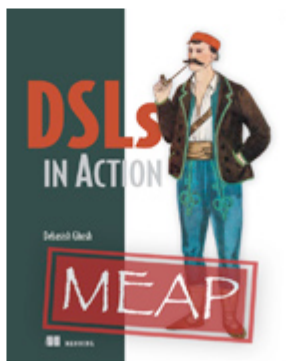
[Lift in Action](#)
EARLY ACCESS EDITION

Timothy Perrett
MEAP Release: April 2010
Softbound print: February 2011 | 450 pages
ISBN: 9781935182801



[AspectJ in Action, Second Edition](#)
IN PRINT

Ramnivas Laddad
MEAP Release: October 2009
September 2009 | 568 pages
ISBN: 1933988053



[DSLs in Action](#)
EARLY ACCESS EDITION

Debasish Ghosh
MEAP Began: October 2009
Softbound print: December 2010 (est.) | 375 pages
ISBN: 9781935182450

For Source Code, Sample Chapters, the Author Forum and other resources, go to
http://www.manning.com/raychaudhuri