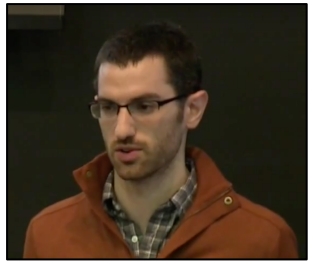
non-strict functions, bottom and Scala by-name parameters

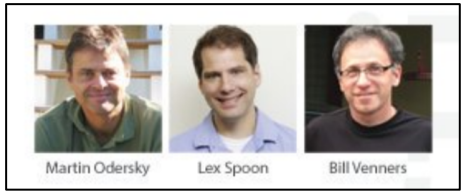
‘a close l⊙⊙k’

through the work of

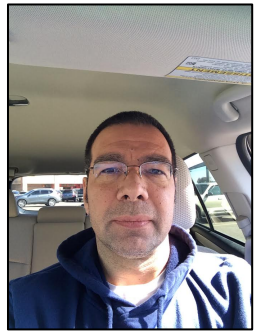
Runar Bjarnason
@runarorama



Paul Chiusano
@pchiusano



Martin Odersky @odersky
Bill Venners @bvenners
Lex Spoon
https://www.linkedin.com/in/lex-spoon-65352816/



Alvin Alexander
@alvinalexander

Mark Lewis
@DrMarkCLewis

Aleksandar Prokopec
@alexprokopec

slides by  @philip_schwarz

Strict and non-strict functions

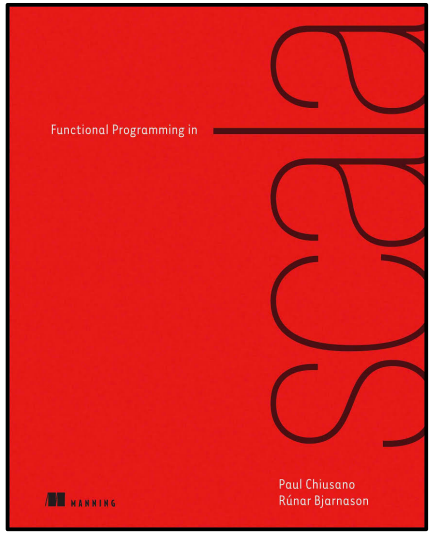
Before we get to our example of lazy lists, we need to cover some basics. What are **strictness** and **non-strictness**, and how are these concepts expressed in **Scala**?

Non-strictness is a property of a function. To say a function is non-strict just means that the function may choose not to evaluate one or more of its arguments. In contrast, a strict function always evaluates its arguments. Strict functions are the norm in most programming languages, and indeed most languages only support functions that expect their arguments fully evaluated. **Unless we tell it otherwise, any function definition in Scala will be strict** (and all the functions we've defined so far have been strict). As an example, consider the following function:

```
def square(x: Double): Double = x * x
```

When you invoke `square(41.0 + 1.0)`, the function `square` will receive the evaluated value of `42.0` because it's strict. If you invoke `square(sys.error("failure"))`, you'll get an exception before `square` has a chance to do anything, since the `sys.error("failure")` expression will be evaluated before entering the body of `square`.

Although we haven't yet learned the **syntax for indicating non-strictness in Scala**, you're almost certainly familiar with the concept. For example, the short-circuiting **Boolean functions** `&&` **and** `||`, found in many programming languages including **Scala**, **are non-strict**. You may be used to thinking of `&&` and `||` as built-in syntax, part of the language, but **you can also think of them as functions that may choose not to evaluate their arguments**.



Functional Programming in Scala

Runar Bjarnason @runarorama

Paul Chiusano @pchiusano

The function **&&** takes two **Boolean** arguments, but only evaluates the second argument if the first is **true** :

```
scala> false && { println("!!"); true } // does not print anything
res0: Boolean = false
```

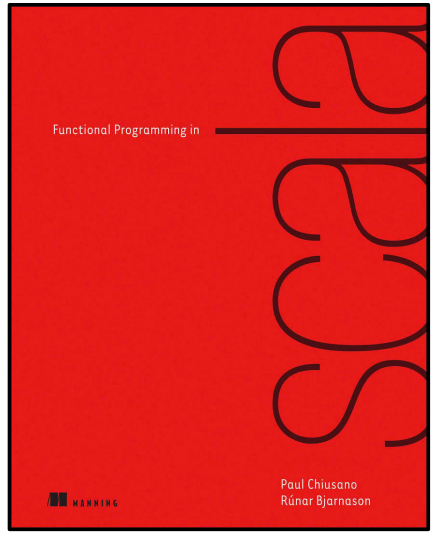
And **||** only evaluates its second argument if the first is **false** :

```
scala> true || { println("!!"); false } // doesn't print anything either
res2: Boolean = true
```

Another example of **non-strictness** is the **if** control construct in **Scala**:

```
val result = if (input.isEmpty) sys.error("empty input") else input
```

Even though **if** is a built-in language construct in **Scala**, it can be thought of as a function accepting three parameters: a condition of type **Boolean**, an expression of some type **A** to return in the case that the condition is **true**, and another expression of the same type **A** to return if the condition is **false**. **This if function would be non-strict, since it won't evaluate all of its arguments**. To be more precise, we'd say that **the if function is strict in its condition parameter**, since it'll always evaluate the condition to determine which branch to take, and **non-strict in the two branches** for the `true` and `false` cases, since it'll only evaluate one or the other based on the condition.



Functional Programming in Scala

Runar Bjarnason @runarorama

Paul Chiusano @pchiusano

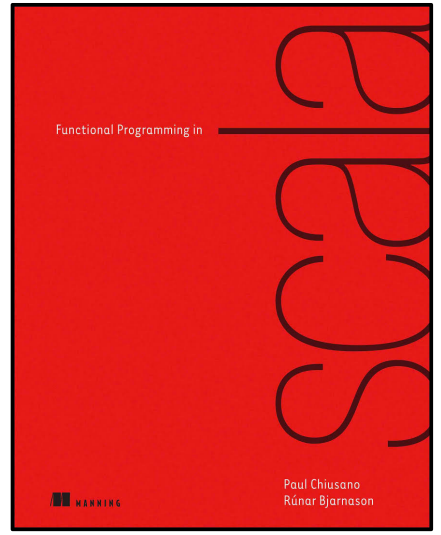
In Scala, we can write non-strict functions by accepting some of our arguments unevaluated. We'll show how this is done explicitly just to illustrate what's happening, and then show **some nicer syntax for it that's built into Scala**. Here's a nonstrict `if` function:

```
def if2[A](cond: Boolean, onTrue: () => A, onFalse: () => A): A =
  if (cond) onTrue() else onFalse()

if2(a < 22,
  () => println("a"),
  () => println("b")
)
```

The function literal syntax for creating a `() => A`

The arguments we'd like to pass unevaluated have a `() =>` immediately before their type. A value of type `() => A` is a function that accepts zero arguments and returns an `A`[3]. In general, the unevaluated form of an expression is called a thunk, and we can force the thunk to evaluate the expression and get a result. We do so by invoking the function, passing an empty argument list, as in `onTrue()` or `onFalse()`. Likewise, callers of `if2` have to explicitly create thunks, and the syntax follows the same conventions as the function literal syntax we've already seen.

Overall, this syntax makes it very clear what's happening—**we're passing a function of no arguments in place of each non-strict parameter, and then explicitly calling this function to obtain a result in the body.**



Functional Programming in Scala

3 In fact, the type `() => A` is a syntactic alias for the type `Function0[A]`.

But this is such a common case that **Scala provides some nicer syntax**:

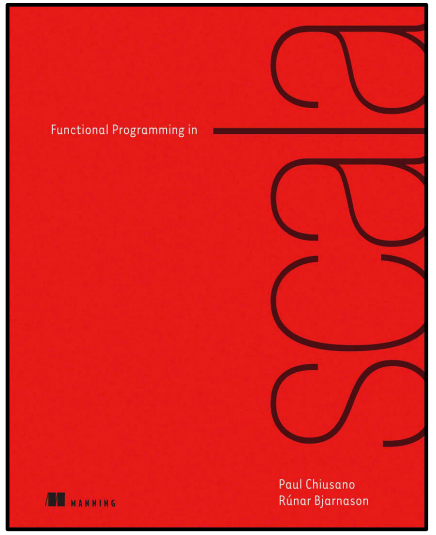
```
def if2[A](cond: Boolean, onTrue: => A, onFalse: => A): A =
  if (cond) onTrue else onFalse
```

The arguments we'd like to pass unevaluated have an arrow => immediately before their type. In the body of the function, we don't need to do anything special to evaluate an argument annotated with =>. We just reference the identifier as usual. Nor do we have to do anything special to call this function. We just use the normal function call syntax, and **Scala** takes care of wrapping the expression in a **thunk** for us:

```
scala> if2(false, sys.error("fail"), 3)
res3: Int = 3
```

With either syntax, **an argument that's passed unevaluated to a function will be evaluated once for each place it's referenced in the body of the function**. That is, **Scala won't (by default) cache the result of evaluating an argument**:

```
scala> def maybeTwice(b: Boolean, i: => Int) = if (b) i+i else 0
maybeTwice: (b: Boolean, i: => Int)Int

scala> val x = maybeTwice(true, { println("hi"); 1+41 })
hi
hi
x: Int = 84
```



Functional Programming in Scala

Runar Bjarnason @runarorama

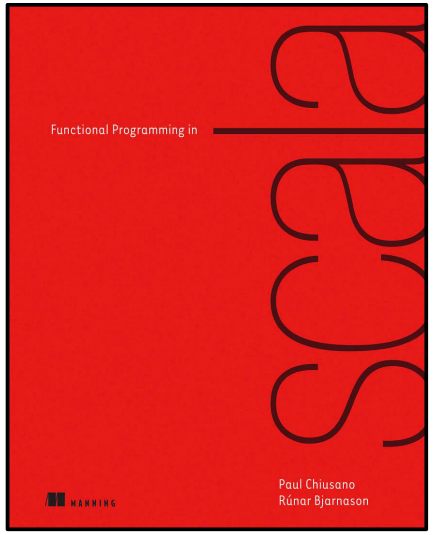
Paul Chiusano @pchiusano

Here, `i` is referenced twice in the body of `maybeTwice`, and we've made it particularly obvious that it's evaluated each time by passing the block `{println("hi"); 1+41}`, which prints `hi` as a **side effect** before returning a result of `42`. The expression `1+41` **will be computed twice** as well. **We can cache the value explicitly if we wish to only evaluate the result once, by using the `lazy` keyword**:

```
scala> def maybeTwice2(b: Boolean, i: => Int) = {
     |   lazy val j = i
     |   if (b) j+j else 0
     | }
maybeTwice2: (b: Boolean, i: => Int)Int


scala> val x = maybeTwice2(true, { println("hi"); 1+41 })
hi
x: Int = 84
```

Adding the `lazy` keyword to a `val` declaration will cause Scala to delay evaluation of the right-hand side of that `lazy val` declaration until it's first referenced. It will also cache the result so that subsequent references to it don't trigger repeated evaluation.

As a final bit of terminology, we say that **a non-strict function in Scala takes its arguments by name rather than by value.**



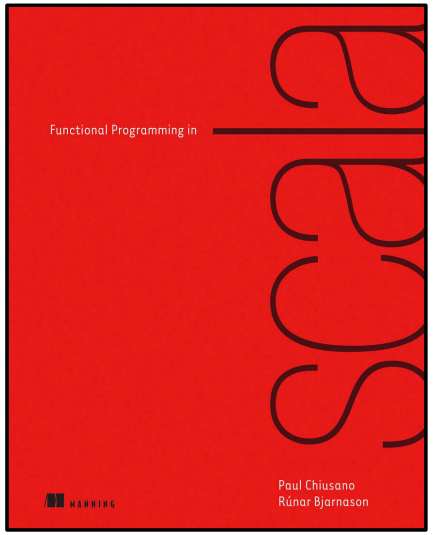
Functional Programming in Scala

Runar Bjarnason @runarorama

Paul Chiusano @pchiusano

Formal definition of strictness

If the evaluation of an expression **runs forever** or **throws an error** instead of returning a definite value, we say that **the expression doesn't terminate**, or that it **evaluates to bottom**. **A function `f` is strict if the expression `f(x)` evaluates to bottom for all `x` that evaluate to bottom**.



Functional Programming in Scala

(by Paul Chiusano and Runar Bjarnason)

@pchiusano @runarorama

Bottom

Navigation

- Haskell
- Wiki community
- Recent changes
- Random page

Tools

- What links here
- Related changes
- Special pages
- Printable version
- Permanent link
- Page information
- Cite this page

The term **bottom** refers to a computation which never completes successfully. That includes a computation that fails due to some kind of error, and a computation that just goes into an infinite loop (without returning any data).

The mathematical symbol for bottom is '⊥'. That's Unicode character 22A5 hex = 8869 decimal. Also available in HTML as '⊥' and in LaTeX as '\bot' (within math mode). In plain ASCII, it's often written as the extremely ugly character sequence '`_|_`'.

Bottom is a member of any type, even the trivial type () or the equivalent simple type:

```
data Unary = Unary
```

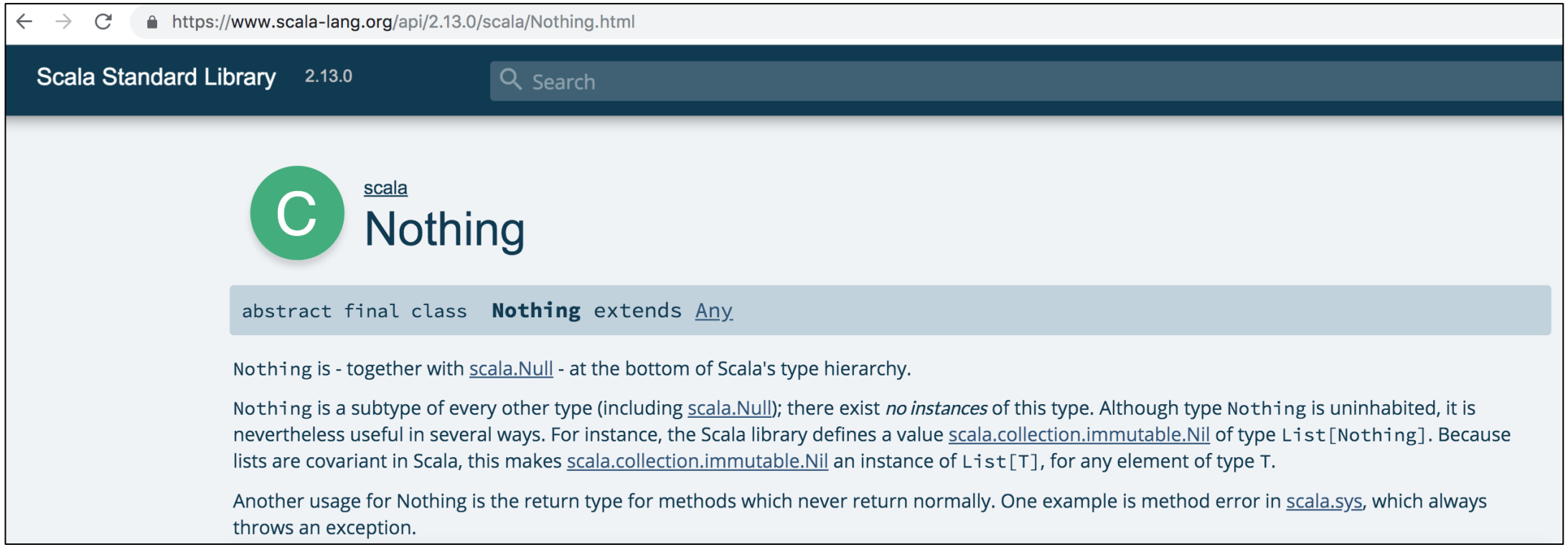
If it were not, the compiler could solve the halting problem and statically determine whether any computation terminated (though note that some languages with dependent type systems, such as Epigram, can statically enforce termination, based on the type for particular programs, such as those using induction).

Bottom can be expressed in Haskell thus:

```
bottom = bottom
```

or

```
bottom = error "Non-terminating computation!"
```




```scala
case object Nil extends List[Nothing] {
  override def head: Nothing = throw new NoSuchElementException("head of empty list")
  override def headOption: None.type = None
  override def tail: Nothing = throw new UnsupportedOperationException("tail of empty list")
  override def last: Nothing = throw new NoSuchElementException("last of empty list")
  override def init: Nothing = throw new UnsupportedOperationException("init of empty list")
  override def knownSize: Int = 0
  override def iterator: Iterator[Nothing] = Iterator.empty
  override def unzip[A1, A2](implicit asPair: Nothing => (A1, A2)): (List[A1], List[A2]) = EmptyUnzip
```

```scala
package object sys {

  /** Throw a new RuntimeException with the supplied message.
   *
   *  @return   Nothing.
   */
  def error(message: String): Nothing = throw new RuntimeException(message)
```

```
scala> scala.sys.error("boom")
java.lang.RuntimeException: boom
  at scala.sys.package$.error(package.scala:27)
  ... 29 elided

scala>
```

9.5 By-name parameters

…suppose you want to implement an assertion construct called **myAssert**. [3] The **myAssert** function will take a function value as input and consult a flag to decide what to do. If the flag is set, **myAssert** will invoke the passed function and verify that it returns **true**. If the flag is turned off, **myAssert** will quietly do nothing at all.

Without using by-name parameters, you could write `myAssert` like this:

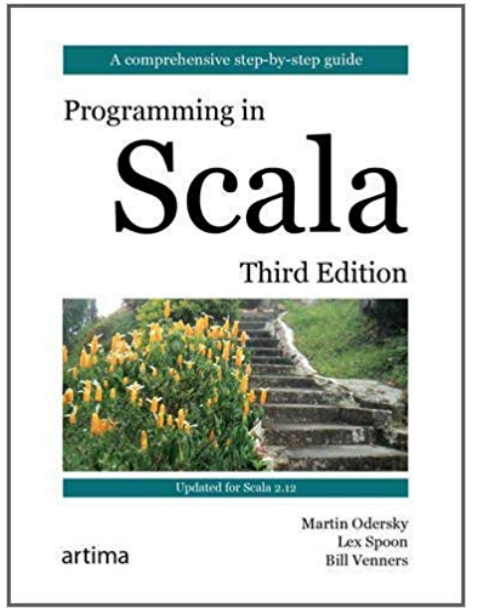
```
var assertionsEnabled = true

def myAssert(predicate: () => Boolean) =
  if (assertionsEnabled && !predicate())
    throw new AssertionError
```

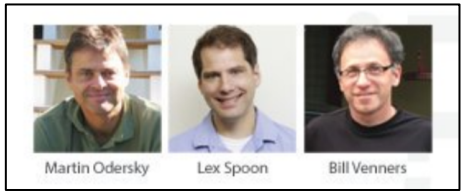
The definition is fine, but using it is a little bit awkward:

```
myAssert(() => 5 > 3)
```

You would really prefer to leave out the **empty parameter list and => symbol** in the function literal and write the code like this:

```
myAssert(5 > 3) // Won't work, because missing () =>
```





Martin Odersky @odersky

Bill Venners @bvenners

Lex Spoon
https://www.linkedin.com/in/lex-spoon-65352816/

[3] You'll call this **myAssert**, not **assert**, because Scala provides an **assert** of its own, which will be described in Section 14.1.

By-name parameters exist precisely so that you can do this. To make a **by-name parameter**, you give the parameter a type starting with `=>` instead of `() =>`. For example, you could change `myAssert`'s `predicate` parameter into a **by-name parameter** by changing its type, "`() => Boolean`", into "`=> Boolean`". Listing 9.5 shows how that would look:

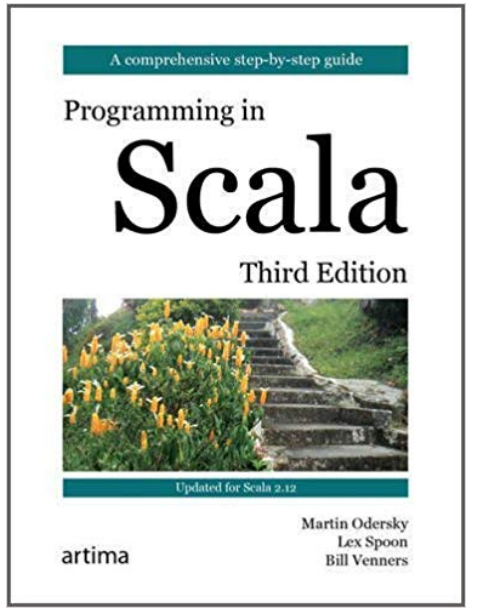
```
def byNameAssert(predicate: => Boolean) =
  if (assertionsEnabled && !predicate)
    throw new AssertionError
```

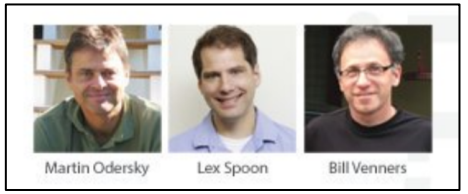
Listing 9.5 - Using a by-name parameter.

Now you can leave out the empty parameter in the property you want to assert. The result is that **using `byNameAssert` looks exactly like using a built-in control structure**:

```
byNameAssert(5 > 3)
```

A **by-name type**, **in which the empty parameter list, `()`, is left out, is only allowed for parameters**. There is no such thing as a by-name variable or a by-name field.





Martin Odersky @odersky

Bill Venners @bvenners

Lex Spoon
https://www.linkedin.com/in/lex-spoon-65352816/

Now, you may be wondering why you couldn't simply write `myAssert` using a plain old `Boolean` for the type of its parameter, like this:

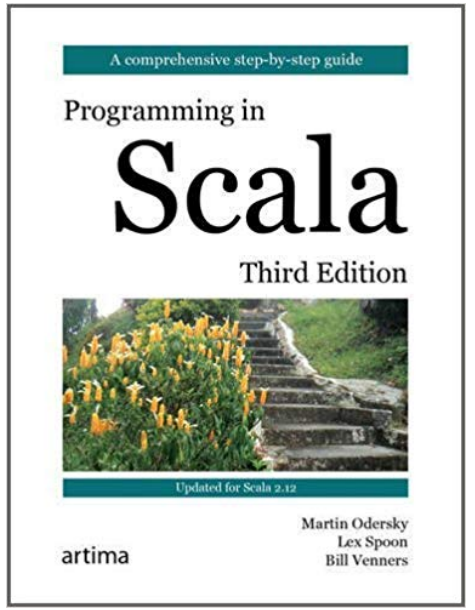
```
def boolAssert(predicate: Boolean) =
  if (assertionsEnabled && !predicate)
    throw new AssertionError
```

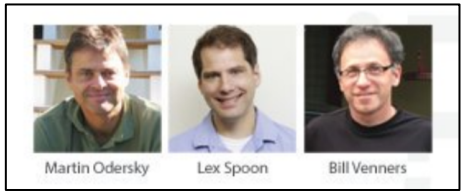
This formulation is also legal, of course, and the code using this version of `boolAssert` would still look exactly as before:

```
boolAssert(5 > 3)
```

Nevertheless, **one difference exists between these two approaches that is important to note**. Because the type of `boolAssert`'s parameter is `Boolean`, **the expression inside the parentheses in `boolAssert(5 > 3)` is evaluated *before* the call to `boolAssert`**. The expression `5 > 3` yields `true`, which is passed to `boolAssert`. **By contrast**, because the type of `boolAssert`'s `predicate` parameter is `=> Boolean`, **the expression inside the parentheses in `byNameAssert(5 > 3)` is *not* evaluated before the call to `byNameAssert`**. Instead a **function value** will be created whose `apply` method will evaluate `5 > 3`, and this function value will be passed to `byNameAssert`.

The **difference between the two approaches**, therefore, is that **if assertions are disabled, you'll see any side effects that the expression inside the parentheses may have in `boolAssert`, but not in `byNameAssert`**.





Martin Odersky @odersky

Bill Venners @bvenners

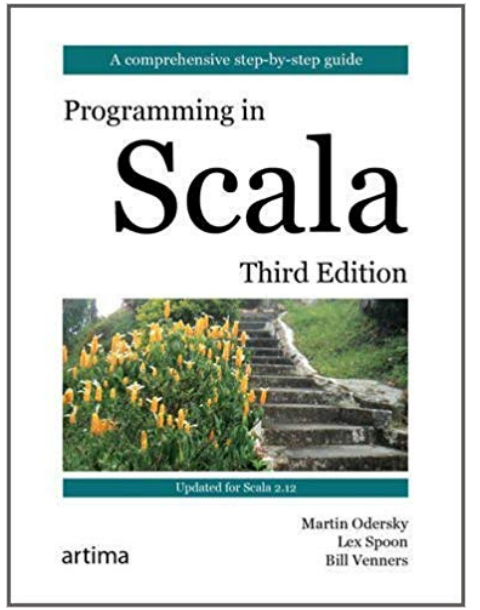
Lex Spoon
https://www.linkedin.com/in/lex-spoon-65352816/

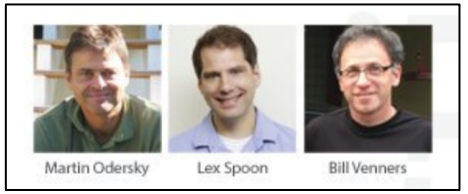
For example, if assertions are disabled, attempting to assert on "`x / 0 == 0`" **will yield an exception** in **`boolAssert`**'s case:

```
scala> val x = 5
x: Int = 5

scala> var assertionsEnabled = false
assertionsEnabled: Boolean = false

scala> boolAssert(x / 0 == 0)
java.lang.ArithmeticException: / by zero
  ... 29 elided

scala>
```

But attempting to assert on the same code in **`byNameAssert`**'s case **will *not* yield an exception**:

```
scala> byNameAssert(x / 0 == 0)

scala>
```





Martin Odersky @odersky

Bill Venners @bvenners

Lex Spoon
https://www.linkedin.com/in/lex-spoon-65352816/

Background: By-name parameters

"**By-name**" **parameters** are quite different than **by-value parameters**. Rob Norris, (aka, "tpolecat") makes the observation that you can think about the two types of parameters like this:

- A **by-value parameter** is like receiving a **val field**; its body is **evaluated once**, **when the parameter is bound to the function**.
- A **by-name parameter** is like receiving a **def method**; its body is **evaluated whenever it is used inside the function**.

Those statements aren't 100% accurate, but they are decent analogies to start with.

A little more accurately, the book Scala Puzzlers says that **by-name parameters** are "**evaluated only when they are referenced inside the function**." The Scala Language Specification adds this:

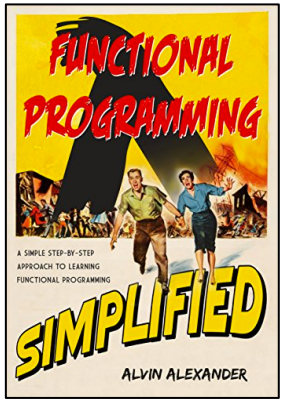
> *This (**by-name**) indicates that the argument is **not evaluated** at the point of **function application**, but instead is **evaluated** at **each use** within the function.*

According to Wikipedia these terms date back to a language named ALGOL 60 (yes, the year 1960). **But for me, the term "by-name" isn't very helpful**. When you look at those quotes from the Puzzlers book and the Language Specification, you see that they both say, "**a by-name parameter is only evaluated when it's accessed inside a function**." Therefore, **I find that the following names are more accurate and meaningful than "by-name"**:

- Call **on access**
- Evaluate **on access**
- Evaluate **on use**
- Evaluate **when accessed**
- Evaluate **when referenced**

However, because I can't change the universe, I'll continue to use the terms "**by-name**" and "call **by-name**" in this lesson, but **I wanted to share those alternate names, which I think are more meaningful**.

Alvin Alexander @alvinalexander



Why have by-name parameters?

Programming in Scala, written by **Martin Odersky** and **Bill Venners**, provides a great example of **why by-name parameters were added to Scala**. Their example goes like this:

1. Imagine that Scala does not have an assert function, and you want one.
2. You attempt to write one using function input parameters, like this:

```
def myAssert(predicate: () => Boolean) =
  if (assertionsEnabled && !predicate())
    throw new AssertionError
```

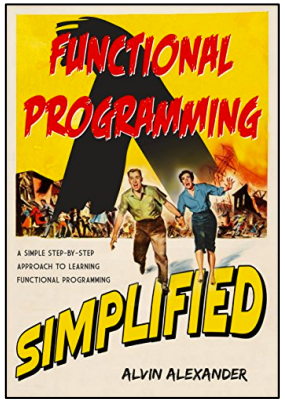
That code uses the "function input parameter" techniques I showed in previous lessons, and assuming that the variable assertionsEnabled is in scope, it will compile just fine.

The problem is that when you go to use it, you have to write code like this:

```
myAssert(() => 5 > 3)
```

Because **myAssert** states that predicate is a function that takes no input parameters and returns a Boolean, that's how you have to write this line of code. **It works, but it's not pleasing to the eye**.

Alvin Alexander @alvinalexander



The solution is to change predicate to be a **by-name parameter**:

```
def byNameAssert(predicate: => Boolean) =
  if (assertionsEnabled && !predicate)
    throw new AssertionError
```

With that simple change, you can now write assertions like this:

```
byNameAssert(5 > 3)
```

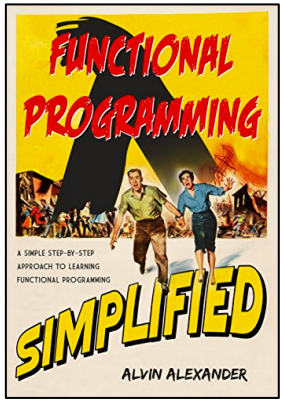
That's **much more pleasing to look at** than this:

```
myAssert(() => 5 > 3)
```

Programming in Scala states that this is **the primary use case for by-name parameters**:

> *The result is that using **byNameAssert** looks exactly **like using a built-in control structure**.*

Alvin Alexander @alvinalexander



```
def timer[A](blockOfCode: => A) = {
  val startTime = System.nanoTime
  val result = blockOfCode
  val stopTime = System.nanoTime
  val delta = stopTime - startTime
  (result, delta/1000000d)
}
```

The **timer** method uses the **by-name** **syntax to accept a block of code as an input parameter**. Inside the **timer** function there are three lines of code that deal with determining how long the **blockOfCode** takes to run, with this line sandwiched in between those time-related expressions:

```
val result = blockOfCode
```

That line (a) executes **blockOfCode** and (b) assigns its return value to result. Because **blockOfCode** is defined to return a generic type (A), it may return Unit, an Int, a Double, a Seq[Person], a Map[Person, Seq[Person]], whatever.

Now you can use the **timer** function for all sorts of things. It can be used for something that isn't terribly useful, like this:

```
scala> val (result, time) =
timer(println("Hello"))
Hello
result: Unit = ()
time: Double = 0.081619
```

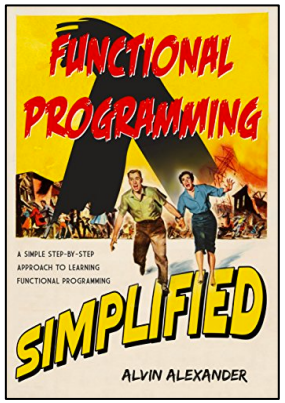
It can be used for an algorithm that reads a file and returns an iterator:

```
scala> def readFile(filename: String) = io.Source.fromFile(filename).getLines
readFile: (filename: String)Iterator[String]

scala> val (result, time) = timer(readFile("/etc/passwd"))
result: Iterator[String] = non-empty iterator
time: Double = 3.004355
```

Or it can be used for just about anything else:

```
scala> val (result, time) = timer{ someLongRunningAlgorithmThatReturnsSomething }
```

Alvin Alexander @alvinalexander



“When is my code block run?”

A great question right now is, “**When are my by-name parameters executed?**” In the case of the timer function, it executes the **blockOfCode** when the second line of the function is reached. But if that doesn’t satisfy your curious mind, you can create another example like this:

```
def test[A](codeBlock: => A) = {
  println("before 1st codeBlock")
  val a = codeBlock
  println(a)
  Thread.sleep(10)
  println("before 2nd codeBlock")
  val b = codeBlock
  println(b)
  Thread.sleep(10)
  println("before 3rd codeBlock")
  val c = codeBlock
  println(c)
}
```

If you paste that code into the Scala REPL, you can then test it like this:

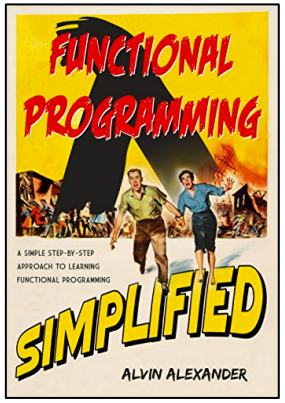
```
scala> test( System.currentTimeMillis )
```

That line of code will produce output like this:

```
before 1st codeBlock
1563698622029
before 2nd codeBlock
1563698622041
before 3rd codeBlock
1563698622053

scala>
```

As that output shows, the block of code that is passed in is executed each time it’s referenced inside the function.

Alvin Alexander @alvinalexander



7.8 Basic Argument Passing

Now that we have looked at the way memory is organized for variable declarations and the **aliasing** issue, we should take another look at **what happens when you call a function**. The two are actually very similar. When you pass values into a function, the function has local `val`s with the local argument names, but they reference the same objects that the outside variables referenced. Consider the following code from a script.

```
def getAndClear(arr:Array[Int],index:Int):Int = {
  val ret=arr(index)
  arr(index)=0
  ret
}
val numbers=Array(7,4,9,8,5,3,2,6,1)
val place=5
val value=getAndClear(numbers,place)
// Other stuff
```

The function is passed an `Array` of `Int`s and an `index` for a locaton in that array. It is supposed to return the value at that location and also "clear" that location. The meaning of "clear" here is to store a zero at that location. To see how this works look at figure 7.5 which shows the arrangement of memory. The function is defined and the variables `numbers` and `place` are both declared and initialized. We get new objects for them to reference.

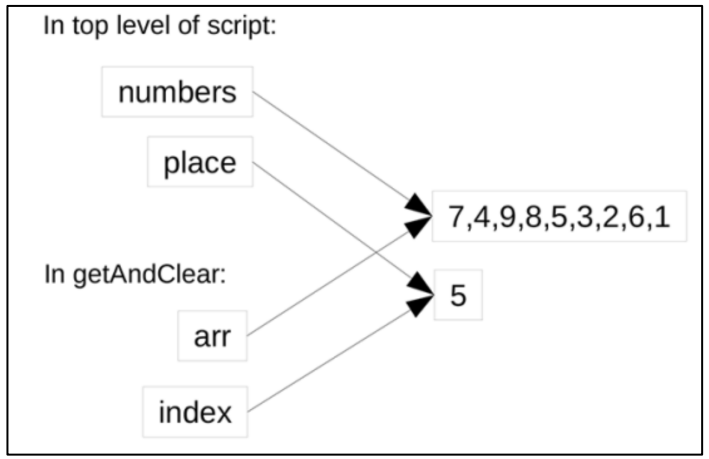
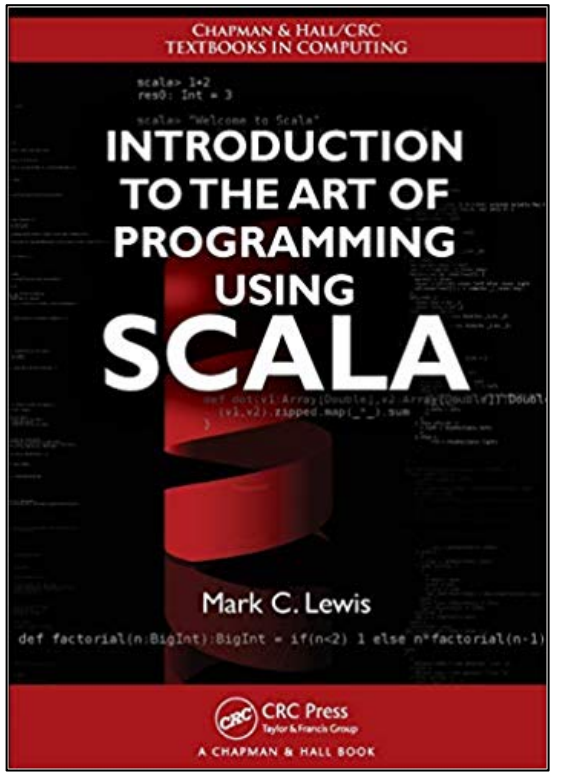


Figure 7.5 - The memory layout from the getAndClear script. The arguments passed into the function become **aliases** for the objects created at the top level of the script.



Mark Lewis
@DrMarkCLewis

7.9 Pass-By-Name

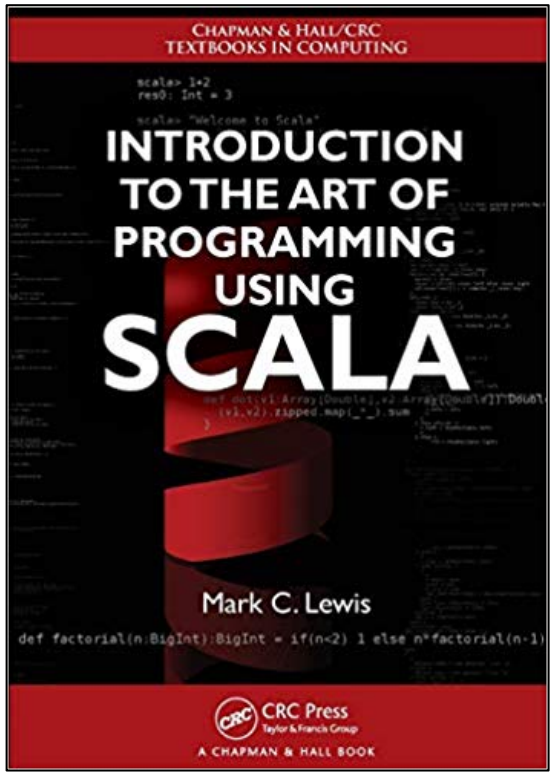
Earlier we looked at the way in which arguments are passed in Scala. This is a style called PASS-BY-VALUE. The function gets a copy of what the variable was in the calling code. The function can not change that variable, it can only change what it refers to if what it refers to is mutable. Some languages provide an alternate way of passing things called PASS-BY-REFERENCE. When something is passed by reference, the function has the ability to change the variable in the calling function itself. The fact that all variables in Scala basically are references blurs the line between the two, but fundamentally Scala only allows pass-by-value and the function can only modify things seen to the outside if the variable refers to a mutable object.

While Scala does not provide a true pass-by-reference, it does provide a passing style that few other languages provide, PASS-BY-NAME. The idea of **pass-by-name** is that **the argument is passed not as a value, but as a THUNK that is basically a set of code that will be executed and give a value when the parameter is used in the function. You can imagine pass-by-name** as **automatically creating a function that takes no argument and returns a value that will be executed each time the argument is used**. To help you understand this and see the syntax, we will give a simple example. We will start making a basic increment function in the way we are used to doing.

```
scala>   def incr(n:Int):Int = {
     |     println("About to increment.")
     |     n+1
     |   }
incr: (n: Int)Int
```

The print statement is just to help us keep track of what is happening when. Now we will write the same function again, but this time pass the argument **by name** instead.

```
scala>   def incrByName(n : =>Int):Int = {
     |     println("About to increment.")
     |     n+1
     |   }
incrByName: (n: => Int)Int
```



Mark Lewis
@DrMarkCLewis

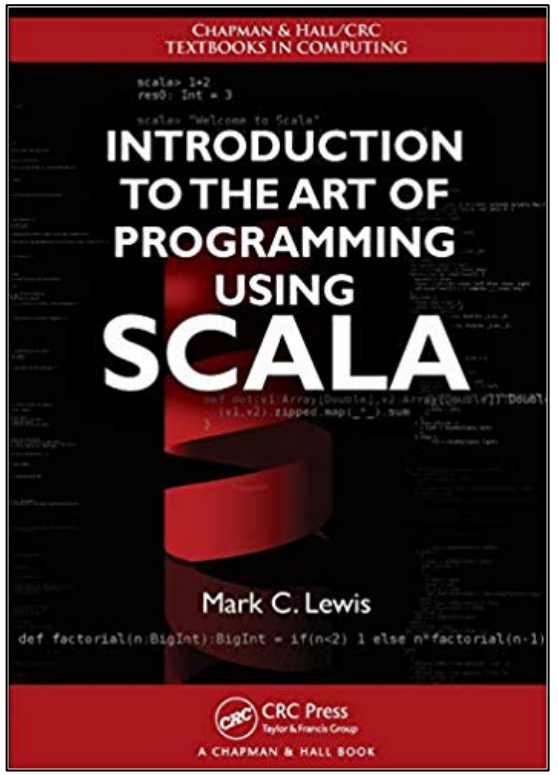
The syntax for passing an argument by name is to put a rocket before the type of the argument. This is the same arrow used in function literals. If we were to put empty parentheses in front of it to get `()=>` we would have the type of a function that takes no arguments and returns an `Int`. The **by-name** argument is much the same only when you call the function you do not have to explicitly make a function. To start off, we will call this function in the simplest way we possibly could.

```
scala> incr(5)
About to increment.
res0: Int = 6

scala> incrByName(5)
About to increment.
res1: Int = 6
```

No surprises here. **They appear to both do the same thing**. Both print the statement and give us back 6. **However, the two are not really doing the same thing**. To make that clear, we will call them again and have the argument be **a block that includes a print**.

```
scala> incr({println("Eval"); 5})
Eval
About to increment.
res2: Int = 6

scala> incrByName({println("Eval"); 5})
About to increment.
Eval
res3: Int = 6
```

Now it is clear that **they are not doing the same thing**. **When you pass an argument by value, it is evaluated before the function starts so that you can have the value to pass through. When you pass by name, the evaluation happens when the parameter is used.** That is why the line "Eval" printed out after "About to increment." in the second case. The `println` in the function happens before the value of `n` is ever accessed.



Mark Lewis
@DrMarkCLewis

Using **pass-by-name** gives you the ability to do some rather interesting things when mutations of values comes into play. To see that, we can write a slightly different function.

```
scala> def thriceMultiplied(n : => Int):Int = n*n*n
thriceMultiplied: (n: => Int)Int
```

It might seem that this method should be called cubed, but as we will see, this might not always apply for it because it uses pass-by-name. To start with, let us call it with an expression that gives a simple value, but prints something first.

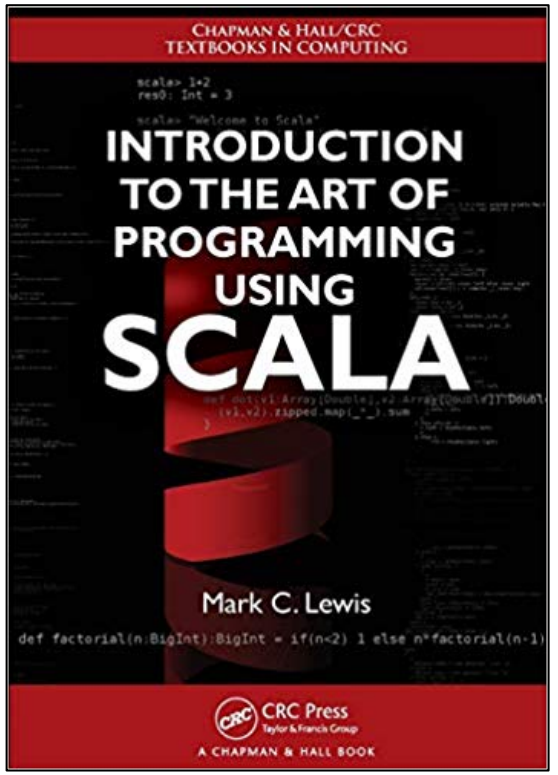
```
scala> thriceMultiplied({println("Get value."); 5})
Get value.
Get value.
Get value.
res4: Int = 125
```

Note that the `println` statement happened three times. This is because the value n was used three times in the function. **Had the parameter not been passed by-name, that would not have happened**. It would have printed once, when the function was called. Try this for yourself.

This particular call still gave us a valid cube though. So why was the function not simply called cube? The reason is that if the code in the argument does the right thing, the result won't be a cube. Here is an example.

```
scala> var i=5
i: Int = 5

scala> thriceMultiplied({i+=1; i})
res5: Int = 336
```

336 is not a perfect cube. So how did we get this value? In this example we introduced a var. The code in the **pass-by-name** argument alters this var. As a result, every time that n is used, the value given by n is different. The first time we use n the value is 6 because the original 5 was incremented and then returned. The next time it is 7 because the 6 that it was set to the previous time is incremented again. The last time it is 8. So the answer is 6 * 7 * 8 = 336.



Mark Lewis
@DrMarkCLewis

These calls with both parentheses and curly braces might seem odd. The creators of **Scala** thought so. As a result, they made it so that **if a function or method takes an argument list with only one value in it, that value can be in curly braces instead of parentheses**. **This allows a shorter syntax for the call that looks like this**.

```
scala> thriceMultiplied {i+=1; i}
res6: Int = 990
```

You might be tempted to think that leaving off the parentheses changed what happened because the result is different. It was not the parentheses though. Remember that after the last call `i` had been incremented up to 8. So this time the result was 9 * 10 * 11 = 990.

Mark Lewis
@DrMarkCLewis

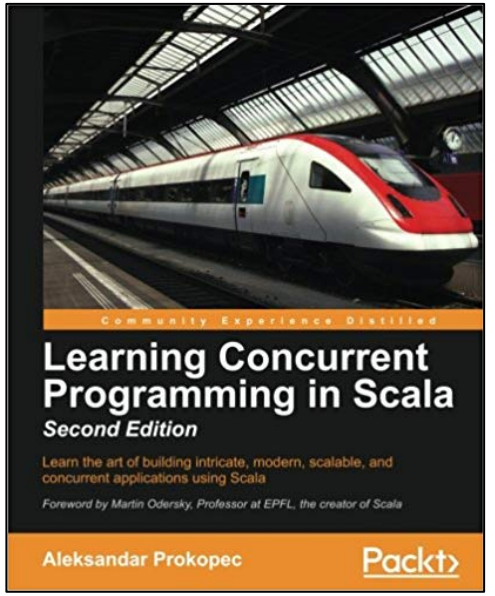
In the following example, we use **byname parameters** to declare a **runTwice** method, which **runs the specified block of code body twice**:

```
def runTwice(body: =>Unit) = {
  body
  body
}
```

A **byname parameter** is formed by putting the **=>** annotation before the type. **Whenever the runTwice method references the body argument, the expression is re-evaluated**, as shown in the following snippet:

```
runTwice { // this will print Hello twice
  println("Hello")
}
```

```
scala>   def runTwice(body: =>Unit) = {
     |     body
     |     body
     |   }
runTwice: (body: => Unit)Unit

scala>

scala>   runTwice { // this will print Hello twice
     |     println("Hello")
     |   }
Hello
Hello

scala>
```



Aleksandar Prokopec
@alexprokopec

In the examples that follow, we will rely on the global ExecutionContext object. To make the code more concise, we will introduce the **execute convenience method** in the package object of this chapter, **which executes a block of code on the global ExecutionContext object**:

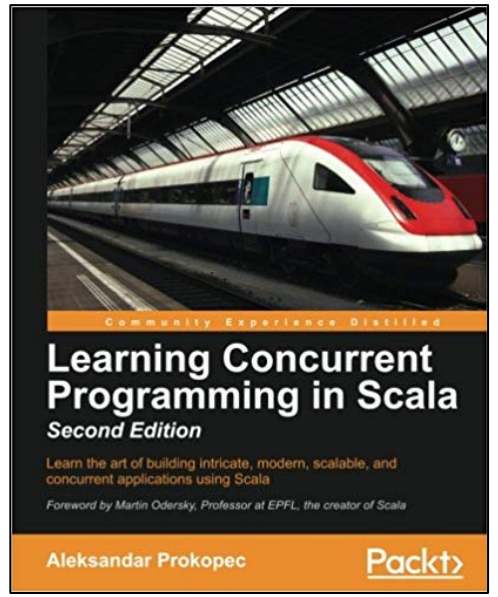
by-name parameter

```
def execute(body: =>Unit) = ExecutionContext.global.execute(
  new Runnable { def run() = body }
)
```

The Executor and ExecutionContext objects are a nifty concurrent programming abstraction, but they are not a silver bullets. They can improve throughput by reusing the same set of threads for different tasks, but they are unable to execute tasks if those threads become unavailable, because all the threads are busy with running other tasks. In the following example, **we declare 32 independent executions, each of which lasts two seconds, and wait 10 seconds for their completion**:

```
object ExecutionContextSleep extends App {
  for (i<- 0 until 32) execute {
    Thread.sleep(2000)
    log(s"Task $i completed.")
  }
  Thread.sleep(10000)
}
```

You would expect that all the executions terminate after two seconds, but this is not the case. Instead, on our quad-core CPU with hyper threading, the global ExecutionContext object has eight threads in the thread pool, so it executes work tasks in batches of eight. After two seconds, a batch of eight tasks print that they are completed, after two more seconds another batch prints, and so on. This is because the global ExecutionContext object internally maintains a pool of eight worker threads, and calling sleep puts all of them into a timed waiting state. Only once the sleep method call in these worker threads is completed can another batch of eight tasks be executed. Things can be much worse. We could start eight tasks that execute the guarded block idiom seen in Chapter 2, *Concurrency on the JVM and the Java Memory Model*, and another task that calls the notify method to wake them up. As the ExecutionContext object can execute only eight tasks concurrently, the worker threads would, in this case, be blocked forever. We say that executing blocking operations on ExecutionContext objects can cause starvation.



Aleksandar Prokopec
@alexprokopec

“we declare 32 independent executions, each of which lasts two seconds, and wait 10 seconds for their completion”

```
scala> ExecutionContextSleep.main(Array())
```

2 seconds elapse

1st batch of 8

```
Task 5 completed.
Task 3 completed.
Task 0 completed.
Task 4 completed.
Task 7 completed.
Task 1 completed.
Task 2 completed.
Task 6 completed.
```

2 seconds elapse

2nd batch of 8

```
Task 8 completed.
Task 14 completed.
Task 15 completed.
Task 9 completed.
Task 13 completed.
Task 12 completed.
Task 10 completed.
Task 11 completed.
```

2 seconds elapse

3rd batch of 8

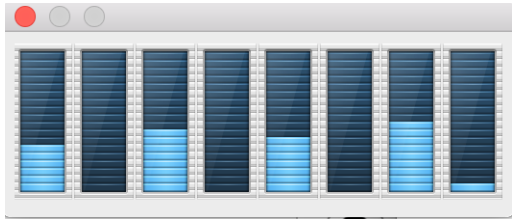
```
Task 17 completed.
Task 16 completed.
Task 18 completed.
Task 23 completed.
Task 19 completed.
Task 21 completed.
Task 22 completed.
Task 20 completed.
```

2 seconds elapse

4th batch of 8

```
Task 24 completed.
Task 25 completed.
Task 27 completed.
Task 30 completed.
Task 29 completed.
Task 28 completed.
Task 26 completed.
Task 31 completed.

scala>
```

CPU usage



Note
The `Future[T]` type encodes latency in the program; use it to encode values that will become available later during execution.

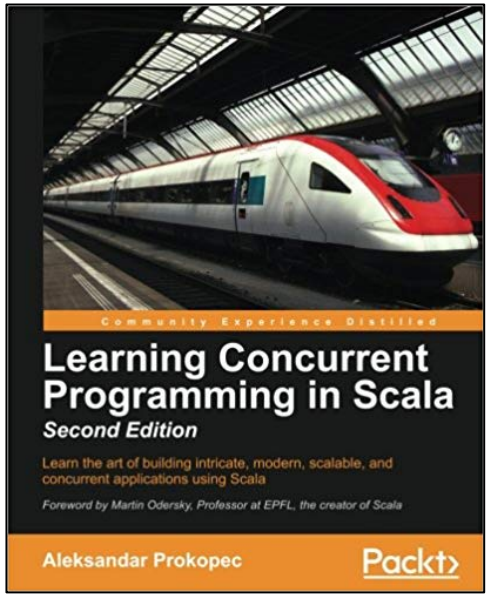
When programming with futures in **Scala**, we need to distinguish between **future values** and **future computations**. A future value of the type **Future[T]** denotes some value of type **T** in the program that might not be currently available, but could become available later. **Usually, when we say a future, we really mean a future value**. In the scala.concurrent package, futures are represented with the **Future[T]** trait:

```
trait Future[T]
```

By contrast, **a future computation is an asynchronous computation that produces a future value**. A future computation can be started by calling the apply method on the **Future** companion object. This method has the following signature in the scala.concurrent package:

```
def apply[T](b: =>T)(implicit e: ExecutionContext): Future[T]
```

This method takes a by-name parameter of the type T. This is the body of the asynchronous computation that results in some value of type T. It also takes an implicit ExecutionContext parameter, which abstracts over where and when the thread gets executed, as we learned in Chapter 3, *Traditional Building Blocks of Concurrency*. Recall that **Scala**'s implicit parameters can either be specified when calling a method, in the same way as normal parameters, or they can be left out-in this case, the **Scala** compiler searches for a value of the ExecutionContext type in the surrounding scope. Most **Future** methods take an implicit execution context. Finally, the **Future.apply** method returns a future of the type T. This future is completed with the value resulting from the asynchronous computation, **b**.



Aleksandar Prokopec
@alexprokopec

https://www.scala-lang.org/api/2.13.0/scala/util/Using$.html

Scala Standard Library 2.13.0 Search

scala.util

Using

```
object Using
```

A utility for performing automatic resource management. It can be used to perform an operation using resources, after which it releases the resources in reverse order of their creation.

Usage

There are multiple ways to automatically manage resources with `Using`. If you only need to manage a single resource, the `apply` method is easiest; it wraps the resource opening, operation, and resource releasing in a `Try`.

Example:

```
val lines: Try[Seq[String]] =
  Using(new BufferedReader(new FileReader("file.txt"))) { reader =>
    Iterator.unfold(())(_ => Option(reader.readLine()).map(_ -> ())).toList
  }
```

If you need to manage multiple resources, `Using.Manager` should be used. It allows the managing of arbitrarily many resources, whose creation, use, and release are all wrapped in a `Try`.

```scala
object Using {
  /** Performs an operation using a resource, and then releases the resource,
    * even if the operation throws an exception.
    *
    * $suppressionBehavior
    *
    * @return a [[Try]] containing an exception if one or more were thrown,
    *         or the result of the operation if no exceptions were thrown
    */
  def apply[R: Releasable, A](resource: => R)(f: R => A): Try[A] =
    Try { Using.resource(resource)(f) }
```

```scala
/** Performs an operation using a resource, and then releases the resource,
  * even if the operation throws an exception. This method behaves similarly
  * to Java's try-with-resources.
  *
  * $suppressionBehavior
  *
  * @param resource the resource
  * @param body     the operation to perform with the resource
  * @tparam R the type of the resource
  * @tparam A the return type of the operation
  * @return the result of the operation, if neither the operation nor
  *         releasing the resource throws
  */
def resource[R, A](resource: R)(body: R => A)(implicit releasable: Releasable[R]): A = { …
```

```
scala> import scala.util.{Success, Try, Using}
import scala.util.{Success, Try, Using}

scala> import java.io.{File, FileInputStream, FileWriter}
import java.io.{File, FileInputStream, FileWriter}

scala>   val file = new File("/Users/philipschwarz/tmp/tmp.txt")
file: java.io.File = /Users/philipschwarz/tmp/tmp.txt

scala>   Using( new FileWriter(file) ){ _.write("xyz") }
res0: scala.util.Try[Unit] = Success(())

scala>   val tryChar: Try[Int] = Using( new FileInputStream(file) ){ _.read }
tryChar: scala.util.Try[Int] = Success(120)

scala>   assert( tryChar == Success('x'))

scala>   val char: Int = Using.resource( new FileInputStream(file) ){ _.read }
char: Int = 120

scala>   assert( char == 'x')

scala>
```