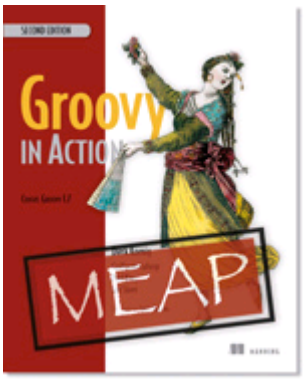
MANNING PUBLICATIONS

# *Declaring and Using Closures*

Article based on




## Groovy in Action, Second Edition

EARLY ACCESS EDITION

Dierk König, Paul King, Guillaume Laforge, Jon Skeet

MEAP Release: June 2009
Softbound print: Early 2011 | 700 pages
ISBN: 9781935182443

*This article is taken from the book* Groovy in Action, Second Edition. *The authors define closures as objects whose main purpose is their behavior. Then, they explain how to declare and invoke closures.*



Before diving into declaring and using closures, let's define closures. A *closure* is a piece of code wrapped up as an object. It acts like a method in that it can take parameters and it can return a value. It's a normal object in that you can pass a reference to it around just as you can a reference to any other object. Don't forget that the JVM has no idea you're running Groovy code, so there's nothing particularly odd that you *could* be doing with a closure object. It's just an object. Groovy provides a very easy way of creating closure objects and enables some very smart behavior.

If it helps you to think in terms of real-world analogies, consider an envelope with a piece of paper in it. For other objects, the paper might have the values of variables on it: "x=5, y=10" and so on. For a closure, the paper would have a list of instructions. You can give that envelope to someone, and that person might decide to follow the instructions on the piece of paper, or they might give the envelope to someone else. They might decide to follow the instructions lots of times, with a different context each time. For instance, the piece of paper might say, "Send a letter to the person you're thinking of," and the person might flip through the pages of their address book thinking of every person listed in it, following the instructions over and over again, once for each contact in that address book.

The Groovy equivalent of that example would be something like this:

```
Closure envelope = { person -> new Letter(person).send() }
addressBook.each (envelope)
```

That's a fairly long-winded way of going about it, but it shows the distinction between the closure itself (in this case, the value of the `envelope` variable) and its use (as a parameter to the `each` method). Part of what makes

closures unfamiliar when coming to them for the first time is that they're usually used in an abbreviated form. Groovy makes them very concise because they're so frequently used—but that brevity can be detrimental to the learning process. Just for the comparison, here's the previous code written using the shorthand Groovy provides. When you see this shorthand, it's often worth mentally separating it out into the longer form:

```
addressBook.each { new Letter(it).send() }
```

It's still a method call passing a closure as the single parameter, but that's all hidden—passing a closure to a method is sufficiently common in Groovy that there are special rules for it. Similarly, if the closure needs to take only a single parameter to work on, Groovy provides a default name—`it`—so that you don't need to declare it specifically. That's how our example ends up so short when we use all of the Groovy shortcuts.

Now, let's move on to how we declare closures.

## Declaring closures

So far, we have used the simple abbreviated syntax of closures: After a method call, put your code in curly braces with parameters delimited from the closure body by an arrow.

Let's start by adding to your knowledge about the simple abbreviated syntax, and then we'll look at two more ways to declare a closure: by using them in assignments and by referring to a method.

### The simple declaration

Listing 1 shows the simple closure syntax plus a new convenience feature. When there is only one parameter passed into the closure, its declaration is optional. The magic variable `it` can be used instead. Listing 1 shows two equivalent closure declarations.

**Listing 1 Simple abbreviated closure declaration**

```
    def log = ''
(1..10).each { counter -> log += counter }
assert log == '12345678910'

log = ''
(1..10).each { log += it }
    assert log == '12345678910'
```

Note that, unlike `counter`, the magic variable `it` needs no declaration.

This syntax is an abbreviation because the closure object as declared by the curly braces is the last parameter of the method and would normally appear within the method call's parentheses. As you will see, it is equally valid to put it inside parentheses like any other parameter, although it is hardly ever used this way[1]:

```
    def log = ''
(1..10).each({ log += it })
    assert log == '12345678910'
```

This syntax is simple because it uses only one parameter, the implicit parameter `it`. Multiple parameters can be declared in sequence, delimited by commas. A default value can optionally be assigned to parameters, in case no value is passed from the method to the closure.

**TIP**

Think of the arrow as an indication that parameters are passed from the method on the left into the closure body on the right.

### Using assignments for declaration

A second way of declaring a closure is to directly assign it to a variable:

[1] Although, I have seen long-time Java programmers that preferred this style and kept it over a number of weeks before getting comfortable with more idiomatic Groovy.

```
Closure printer = { line -> println line }
```

The closure is declared inside the curly braces and assigned to the `printer` variable.

**TIP**

Whenever you see the curly braces of a closure, think: `new Closure(){}`.

There is also a natural kind of assignment to the return value of a method:

```
def Closure getPrinter() {
{ line -> println line }
}
```

Again, the curly braces denote the construction of a new closure object. This object is returned from the method call.

**TIP**

Curly braces can denote the construction of a new closure object or a Groovy *block*. Blocks can be class, interface, static or object initializers, or method bodies, or they can appear with the Groovy keywords `if`, `else`, `synchronized`, `for`, `while`, `switch`, `try`, `catch`, and `finally`. All other occurrences are closures.

As you can see, closures are objects. They can be stored in variables, they can be passed around, and, as you probably guessed, you can call methods on them. Since they're objects, you can also return a closure from a method.

### *Referring to methods as closures*

The third way of declaring a closure is to reuse something that is already declared: a method. Methods have a body and, optionally, return values, can take parameters, and can be called. The similarities with closures are obvious, so Groovy lets you reuse the code you already have in methods as a closure. Referencing a method as a closure is performed using the `reference.&` operator. The reference is used to specify which instance should be used when the closure is called, just like a normal method call to `reference.someMethod()`.

Listing 2 demonstrates method closures in action, showing two different instances used to give two different closures, even though the same method is invoked in both cases.

**Listing 2 Simple method closures in action**

```
class SizeFilter {
Integer limit

boolean filter (String value) {
    return value.length() <= limit
}
}

SizeFilter six  = new SizeFilter(limit:6)  //#1 GroovyBean constructor
SizeFilter five = new SizeFilter(limit:5)  //#1 calls

Closure smallerSix = six.&filter           //#2 Method closure assignment

def words = ['long string', 'medium', 'short', 'tiny']

assert 'medium' == words.find (smallerSix)     //#3 Calling with closure
assert 'short'  == words.find (five.&filter)   //#4 Passing a method closure directly
```

Each instance (created at #1) has a separate idea of how long a string it will deem to be valid in the `filter` method. We create a reference to that method with `six.&filter` at #2 and `five.&filter`, showing that the reference can be assigned to a variable that is then passed (at #3) or passed as a parameter to the `find` method at #4. We use a sample list of words to check that the closures are doing what we expect them to.

Method closures are limited to instance methods, but they do have another interesting feature—runtime overload resolution also known as multimethods. The same method name called with different parameters is used to call different implementations. Listing 3 gives you a taste.

**Listing 3 Multimethod closures**

```
        class MultiMethodSample {

       int mysteryMethod (String value) {
           return value.length()
       }
       int mysteryMethod (List list) {
           return list.size()
       }
       int mysteryMethod (int x, int y) {
           return x+y
       }
   }

   MultiMethodSample instance = new MultiMethodSample()
   Closure multi = instance.&mysteryMethod       //#1 Only a single closure is created

   assert 10 == multi ('string arg')             //#2 Different implementations
   assert  3 == multi (['list', 'of', 'values']) //#2 are called based on
        assert 14 == multi (6, 8)                    //#2 argument types
```

Here, a single instance is used and, indeed, a single closure (at #1)—but each time it's called, a different method implementation is invoked, at #2.

With the topic presented so far, you should be able to understand a construction that the Grails framework uses pervasively: properties of type Closure. The interesting effect of this construction is that you can change a property value at runtime and, consequently, you can assign a new closure to a property just as well. Listing 4 mimics a Grails controller where we change the list "action" at runtime.

**Listing 4 ClosureProperty that looks like a method but can be redefined at runtime**

```
        class MultiMethodSample {

       int mysteryMethod (String value) {
           return value.length()
       }
       int mysteryMethod (List list) {
           return list.size()
       }
       int mysteryMethod (int x, int y) {
           return x+y
       }
   }

   MultiMethodSample instance = new MultiMethodSample()
   Closure multi = instance.&mysteryMethod       //#1 Only a single closure is created

   assert 10 == multi ('string arg')             //#2 Different implementations
   assert  3 == multi (['list', 'of', 'values']) //#2 are called based on
        assert 14 == multi (6, 8)                    //#2 argument types
```

What makes ClosureProperties attractive is that calling the closure looks exactly like a method call but the ability to assign a new closure object yields possibilities that would otherwise only be available through metaprogramming—redefining execution logic at runtime on a per-instance basis.

Now that you've seen all the ways of declaring a closure, it's worth pausing for a moment and seeing them all together performing the same function, just with different declaration styles.

### Comparing the available options

Listing 5 shows all of the ways of creating and using closures: through simple declaration, assignment to variables, and method closures. In each case, we call the `each` method on a simple map, providing a closure that doubles a single value. By the time we're finished, we've doubled each value three times.

**Listing 5 List of closure declaration examples**

```
     Map map = ['a':1, 'b':2]
map.each{ key, value -> map[key] = value * 2 }          //#1 Parameter sequence with commas
assert map == ['a':2, 'b':4]

Closure doubler = {key, value -> map[key] = value * 2 } //#2 Assign and then call
map.each(doubler)                                        //#2 a closure reference
assert map == ['a':4, 'b':8]

def doubleMethod (entry){                                //|#3 A usual method
    entry.value = entry.value * 2                        //|#3 declaration
}
doubler = this.&doubleMethod                             //#4 Reference and call
map.each(doubler)                                        //#4 a method as a closure
     assert map == ['a':8, 'b':16]
```

In #1, we pass the closure as the parameter directly. This is the form you've seen most commonly so far.

The declaration of the closure in #2 is disconnected from its immediate use. The curly braces are Groovy's way to declare a closure, so we assign a closure object to the variable doubler. Some people incorrectly interpret this line as assigning the result of a closure call to a variable. Don't fall into that trap! This line creates a Closure object, but doesn't call it. The `each` method calls it instead. You can see that passing the closure as an argument to the method via reference is exactly the same as declaring it in place, the style that we followed in all the previous examples.

The method declared in #3 is a perfectly ordinary method. There is no trace of our intention to use it as a closure.

In #4, the `reference.&` operator is used for referencing a method name as a closure. Again, the method is not immediately called; the execution of the method occurs as part of the next line. This is just like #2. The closure is passed to the `each` method, which calls it back for each entry in the map.

Typing[2] is optional in Groovy and, consequently, it is optional for closure parameters. Just like for methods, if you choose to specify explicit type markers for closure parameters, Groovy guarantees that those parameters will be the right time at runtime.

In order to fully understand how closures work and how to use them within your code, you need to find out how to invoke them. That is the topic of the next subtopic.

## Using closures

So far, you have seen how to create a closure for the purpose of passing it into a method such as `each`. But what happens inside the `each` method? How does it call your closure? If you knew this, you could come up with equally smart implementations. First, we'll look at how simple it is to call a closure and then move on to explore some advanced methods that the `Closure` type has to offer.

### Calling a closure

Suppose we have a reference `x` pointing to a closure; we can call it with `x.call()` or simply `x()`. You have probably guessed that any arguments to the closure call go between the parentheses.

We start with a simple example. Listing 6 shows the same closure being called both ways.

[2] The word typing has two meanings: declaring object types and typing keystrokes. Although Groovy provides optional typing, you still have to key in your program code.

**Listing 6 Calling closures**

```
def adder = { x, y -> return x+y }

assert adder(4, 3) == 7
assert adder.call(2, 6) == 8
```

We start off by declaring pretty much the simplest possible closure—a piece of code that returns the sum of two arguments. Then, we call the closure both directly and using the `call` method. Both ways of calling the closure achieve exactly the same effect.

Now, let's try something more intense. In Listing 7, we demonstrate calling a closure from within a method body and how the closure gets passed into that method in the first place. The example measures the execution time of the closure.

**Listing 7 Calling closures**

```
def benchmark(int repeat, Closure worker) {      //#1 Put closures last
    def start = System.nanoTime()                //#2 Some pre-work

    repeat.times { worker(it) }                  //#3 Call closure a given number of times

    def stop = System.nanoTime()                 //|#4 Some post-work
    return stop - start                          //|#4
}
def slow = benchmark(10000) { (int) it / 2 }     //|#5 Pass different closures
def fast = benchmark(10000) { it.intdiv(2) }     //|#5 for benchmarking
assert fast * 15 < slow                               //|#5
```

We needed to duplicate the benchmarking logic because we had no means to declare how to benchmark *something*. Now you know how. You can pass a closure into the `benchmark` method, where some pre- and post-work takes control of timing it appropriately.

We put the closure parameter at the end of the parameter list in #1 to allow the simple abbreviated syntax when calling the method. In the example, we declare the type of the parameter to be a closure. This is only to make things more obvious—the Closure type is optional.

We effectively start timing the benchmark at #2. From a general point of view, this is arbitrary pre-work like opening a file or connecting to a database. It just so happens that our resource is time.

At #3, we call the given closure as many times as our repeat parameter demands. We pass the current count to the closure to make things more interesting. From a general point of view, a resource is passed to the closure.

We stop timing at #4 and calculate the time taken by the closure. This is where the post-work logic belongs: closing files, flushing buffers, returning connections to the pool, and so on.

The payoff comes at #5. We can now pass logic to the benchmark method. Note that we use the simple abbreviated syntax and use the magic `it` to refer to the current count. As a side effect, we learn that the general number division takes more than 15 times longer than the optimized `intdiv` method.

**BY THE WAY**

This kind of benchmarking should not be taken too seriously. There are all kinds of effects that can heavily influence such wall-clock based measurements: machine characteristics, operating system, current machine load, JDK version, Just-In-time compiler and Hotspot settings, and so on.

Figure 1 shows the UML sequence diagram for the general calling scheme of the declaring object that creates the closure, the `method` invocation on the caller, and the caller's callback to the given closure.

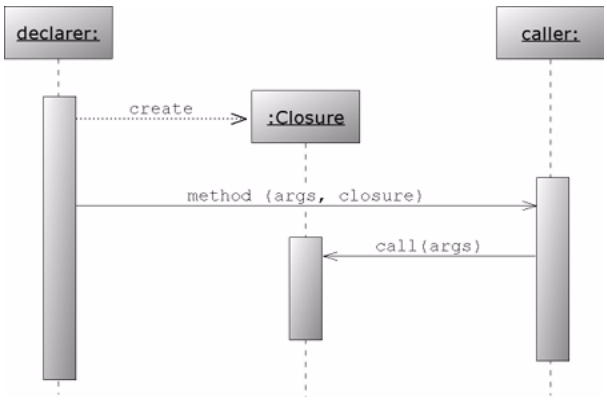



Figure 1 UML sequence diagram of the typical sequence of method calls when a declarer creates a closure and attaches it to a method call on the caller, which in turn calls that closure's `call` method

When calling a closure, you need to pass exactly as many arguments to the closure as it expects to receive, unless the closure defines default values for its parameters. A default value is used when you omit the corresponding argument. The following is a variant of the addition closure as used in Listing 6, with a default value for the second parameter and two calls—one that passes two arguments, and one that relies on the default:

```
def adder = { x, y=5 -> return x+y }

assert adder(4, 3) == 7
assert adder.call(7) == 12
```

The same rules apply for default parameters in closures as they do for methods. Also, closures can be used with a variable length argument list in the same way as methods.

At this point, you should be comfortable with passing closures to methods and have a solid understanding of how the callback is executed. Whenever you pass a closure to a method, you can be sure that a callback will be executed one way or the other (perhaps conditionally), depending on that method's logic. Closures are capable of more than just being called, though. In the next subtopic, you see what else they have to offer.

## *More closure methods*

The class `groovy.lang.Closure` is an ordinary class, albeit one with extraordinary power and extra language support. It has various methods available beyond `call`. We will present the most the important ones; even though you will usually just declare and call closures, it's nice to know there's some extra power available when you need it.

### REACTING ON THE PARAMETER COUNT

A simple example of how useful it is to change behavior based on the parameter count of a closure is map's `each` method. It passes either a `Map.Entry` object or key and value separately into the given closure, depending on whether the closure takes one argument or two. You can retrieve the information about the expected parameter count (and types, if declared) by calling the closure's `getParameterTypes` method:

```
def caller (Closure closure) {
    closure.getParameterTypes().size()
}

assert caller { one -> }      == 1
assert caller { one, two -> } == 2
```

As in the `Map.each` example, this allows for the luxury of supporting closures with different parameter styles, adapted to the caller's needs.

### HOW TO CURRY FAVOR WITH A CLOSURE

*Currying* is a technique invented by Moses Schönfinkel and Gottlob Frege and named after the logician Haskell Brooks Curry, a pioneer in *functional programming*. (Unsurprisingly, the functional language Haskell is also named after Curry.) The basic idea is to take a function with multiple parameters and transform it into a function with fewer parameters by fixing some of the values. A classic example is to choose some arbitrary value *n* and transform a function that sums two parameters into a function that takes a single parameter and adds *n* to it.

In Groovy, Closure's `curry` method returns a clone of the current closure, having bound one or more parameters to a given value. Parameters are bound to curry's arguments from left to right. Listing 8 gives an implementation of the addition example.

**Listing 8 A simple currying example**

```
     def adder  = { x, y -> return x+y }
def addOne = adder.curry(1)
     assert addOne(5) == 6
```

We reuse the same closure you've seen a couple of times now for general summation. We call the `curry` method on it to create a new closure, which acts like a simple adder, but with the value of the first parameter always fixed as 1. Finally, we check our results.

If you're new to closures or currying, now might be a good time to take a break. It's a deceptively simple concept to describe mechanically, but it can be quite difficult to internalize. Just take it slowly, and you'll be fine.

The real power of currying comes when the closure's parameters are themselves closures. This is a common construction in functional programming, but it does take a little getting used to.

For example, suppose you are implementing a logging facility. It should support customized formatting and appending to an output device. The idea is to provide a single closure for a customized version of each activity, while still allowing you to implement the overall pattern of when to do the formatting and appending in one place. Listing 9 uses currying to inject the customized activities into that pattern:

**Listing 9 Using the curry method to configure formatting and appending of log entries**

```
     def logger = { formatter, appender, line ->         //#1 General logging device
    appender(formatter(line))
}
def rightFormatter = { line -> line.padLeft(25) + "n" }    //|#2 Tool
def out = new StringBuilder()                               //|#2 details
def stringAppender = { line -> out << line }                //|#2

def myLog  = logger.curry(rightFormatter, stringAppender)    //#3 Custom device

myLog 'here is some log message'

     assert out.toString().contains('log message')
```

Closure (#1) is like a recipe: given any output formatter, appender, and a line to log, hat line must first be formatted and then appended—easy. The short closures in #2 are the specific ingredients in the recipe. They could be specified every time, but we're always going to use the same ingredients. Currying (at #3) allows us to remember just one object rather than each of the individual parts. To continue the recipe analogy, we've put all of the ingredients together, and the result needs to be put in the oven whenever we want to do some logging.

Currying is a strategy that the functional programming community needed to develop since, in a true functional world, you have no objects to carry mutable state. But in Groovy, we do have objects and can use them together with closure properties as a curry surrogate. Listing 10 puts the formatting and appending logic into closures that are held by the `GeneralLogger` as properties.

**Listing 10 Property closures as a curry surrogate**

```
        class GeneralLogger {
      Closure formatter = { line -> line }     //|#1 Closure properties
      Closure appender  = { }                  //|#1
      Closure logger    = { line ->
          appender(formatter(line))
      }
  }
  def out = new StringBuilder()
  def myLog = new GeneralLogger(
      formatter : { line -> line.padLeft(25) + "n" },
      appender  : { line -> out << line }
  ).logger

  myLog 'here is some log message'

       assert out.toString().contains('log message')
```

This is another compelling example where closure properties demonstrate their flexibility over methods with fixed implementations. Whatever style you prefer, Groovy gives you the freedom of choice.

#### CLASSIFICATION VIA THE ISCASE METHOD

Closures implement the `isCase` method to make closures work as classifiers with `ingrep` and `switch`. In that case, the respective argument is passed into the closure. Calling the closure needs to evaluate to a Groovy Boolean value. For example, consider this code:

```
        def odd = { it % 2 == 1}

  assert [1,2,3].grep(odd) == [1, 3]

  switch(10) {
      case odd : assert false
  }

       if (2 in odd) assert false
```

You can see that the switch statement allows us to classify values with arbitrary logic. Again, this is only possible because closures are objects.

#### REMAINING METHODS

For the sake of completeness, it needs to be said that closures support the `clone` method in the usual Java sense.

The `asWriteable` method returns a clone of the current closure that has an additional `writeTo(Writer)` method to write the result of a closure call directly into the given Writer.

## *Summary*

You have seen that closures follow our theme of *everything is an object*. They capture a piece of logic, making it possible to pass it around for execution, return it from a method call, or store it for later usage.

Closures encourage centralized resource handling, thus making your code more reliable. This doesn't come at any expense. In fact, the codebase is relieved from structural duplication, enhancing expressiveness and maintainability.

Defining and using closures is surprisingly simple because all the difficult tasks such as keeping track of references and relaying method calls back to the delegating owner are done transparently.