CHAPTER 9

CLOSURES

Groovy *closures* are a powerful way of representing blocks of executable code. Since closures are objects, they can be passed around as method parameters, for example. Because closures are code blocks, they can also be executed when required. Like methods, closures can be defined in terms of one or more parameters. A significant characteristic of closures is that they can access state information. This means that any variables in scope when the closure is defined can be used and modified by the closure.

One of the most common uses for a closure is processing a collection. For example, we can iterate across the elements of a collection and apply the closure to them. Groovy's closures are one feature that make developing scripts so easy.

This chapter introduces the general concepts of closures. Appendix H explores further features of closures, and Appendix J considers some advanced topics.

9.1 CLOSURES

The syntax for defining a closure is:

```
{comma-separated-formal-parameter-list -> statement-list}
```

If no formal parameters are required, then the parameter `List` and the -> separator are omitted. Here is a simple example of a closure with no parameters.

```
def clos={println 'Hello world'}
clos.call()
```

EXAMPLE 01
A closure and its invocation

Here, the closure has no parameters and consists of a single `println` statement. The closure is referenced by the identifier `clos`. The code block referenced by this identifier can be executed with the *call statement*, as shown in the example. The result is to print the message:

```
Hello world
```

◆

By introducing formal parameters into closure definitions, we can make them more useful, as we did with methods. Here is the same closure with the name of the individual receiving the greeting now provided as a parameter:

EXAMPLE 02
Parameterized closure

```
def clos = {param -> println "Hello ${param}"}

clos.call('world')        // actual argument is 'world'
clos.call('again')        // actual argument is 'again'
clos('shortcut')          // abbreviated form
```

When we execute this script, the output produced is:

```
Hello world
Hello again
Hello shortcut
```

◆

Observe the third invocation in which the `call` has been omitted.

The next illustration repeats the previous example and produces the same result, but shows that an implicit single parameter referred to as `it` can be used.

EXAMPLE 03
Implicit single parameter

```
def clos = {println "Hello ${it}"}

clos.call('world')
clos.call('again')

clos('shortcut')
```

We noted in the introduction that state information could be accessed by closures. More formally, closures can refer to variables at the time the closure is defined. Consider Example 04. Here, the variable `greeting` defines the salutation. This variable is in scope before `clos` is defined and, hence, its value can be used when the closure is called. Initially, the variable greeting has the value 'Hello'.

EXAMPLE 04
Closures and enclosing scope

```
def greeting = 'Hello'
def clos = {param -> println "${greeting} ${param}"}
clos.call('world')

          // Now show that changes to this variable change the closure.
greeting = 'Welcome'
clos.call('world')
```

Before the second call to the closure, the value of the variable greeting is changed. This is reflected in the output produced by running the program:

```
Hello world
Welcome world
```

◆

In the next example, we augment the previous code with a method entitled demo. It is passed a single argument clo, representing a closure. The method calls this closure with the argument 'Ken'. The method also introduces a new scope in which another variable greeting is bound to the value 'Bonjour'. This additional call generates the output:

```
Welcome Ken
```

and reveals that the state accessible to a closure is that in existence at the time the closure is defined and not when it is called. Here is the illustration:

EXAMPLE 05
Effect of scope

```
def greeting = 'Hello'
def clos = {param -> println "${greeting} ${param}"}
clos.call('world')

// Now show that changes to this variable change the closure.
greeting = 'Welcome'
clos.call('world')

def demo(clo) {
    def greeting = 'Bonjour'              // does not affect closure
    clo.call('Ken')
}

demo(clos)
```

The resulting output is:

```
Hello world
Welcome world
Welcome Ken
```

When calling methods that take a closure as the final parameter, Groovy offers a simplification that makes the code somewhat easier to read. In the last example, the method demo was called and the actual parameter was the closure. Where the final parameter to a method call is a closure, then it can be removed from the list of actual parameters and placed immediately after the closing parenthesis. Hence, the call to method demo could appear as either of the following:

```
demo(clos)       // closure parameter within the parentheses
demo() clos      // parameter removed from parentheses
```

◆

This is demonstrated in the following example, where the closure is removed from the actual parameters in the call to demo and its empty parameter list is deleted.

EXAMPLE 06
Leave the closure outside of the actual argument list

```
def greeting='Hello'
def clos={param -> println "${greeting} ${param}"}

def demo(clo) {
    def greeting='Bonjour'                         // does not affect closure
    clo.call('Ken')
}

//demo() clos                                      // 1: closure reference; include parentheses
demo() {param -> println "Welcome ${param}"}       // 2: closure literal; include parentheses

demo clos                                          // 3: closure reference; omit parentheses
demo {param -> println "Welcome ${param}"}         // 4: closure literal; omit parentheses
```

The output is:

```
Welcome Ken
Hello Ken
Welcome Ken
```

◆

The final two program statements (numbered 3 and 4 in the comments) call the method `demo` and pass a closure as the actual parameter. The first of these two method calls uses a reference to a closure object while the second uses a closure literal. In both illustrations, the parentheses for the method call are omitted.

Observe also the statements numbered 1 and 2 in the comments. The second uses a closure literal and is acceptable to Groovy. However, the first uses a closure reference and is not successfully identified as part of the statement. This causes an execution error that reports that the method call is passed a `null` parameter.

More usually, closures are applied to collections (see Section 9.2). Effectively, we iterate over the elements in the collection and apply a closure to each element. For example, all numeric types support a method entitled `upto`. The signature for this method is:

```
void upto(Number to, Closure closure)
```

The programmer might call the method as in:

```
1.upto(10) {...}
```

This would call the closure literal 10 times. If the closure has a formal parameter `p`:

```
1.upto(10) {p -> ...}
```

then on each iteration, the parameter takes the value 1, 2, ..., 10.

The method `upto` iterates from the numeric value of the recipient number (1) to the given parameter value (10), calling the `closure` on each occasion. We can usefully employ this method to provide a way to compute the factorial of some value. We use `upto` to generate the series of integers 1, 2, 3, ... to a given limit. For each value, we compute the partial factorial until the series is complete. Here is the code:

EXAMPLE 07
Factorial with closures

```
def factorial = 1
1.upto(5) {num -> factorial *= num}
println "Factorial(5): ${factorial}"
```

Running the script produces the output:

```
Factorial(5): 120
```

◆

9.2 CLOSURES, COLLECTIONS, AND STRINGS

Several List, Map, and String methods accept a closure as an argument (see also Appendix H). It is this combination of closures and collections that provides Groovy with some elegant solutions to common programming problems. For example, the each method with the signature:

```
void each(Closure closure)
```

is used to iterate through a List, Map, or String and apply the closure on every element. Example 08 presents a number of simple examples of the each method and a closure.

EXAMPLE 08
Illustrations of the method each and a closure

```
[1, 2, 3, 4].each {println it}

['Ken' : 21, 'John' : 22, 'Sally' : 25].each {println it}
['Ken' : 21, 'John' : 22, 'Sally' : 25].each {println "${it.key} maps to: ${it.value}"}

'Ken'.each {println it}
```

The first example prints the values 1, 2, 3, and 4 on separate lines. The final example prints each letter of the name on a separate line. In the second illustration, the keys and values from the Map are displayed in the style Ken=21. The third demonstration separately accesses the key and value from the Map element and prints them as Ken maps to: 21. The output is:

```
1
2
3
4
Sally=25
John=22
Ken=21
Sally maps to: 25
John maps to: 22
Ken maps to: 21
K
e
n
```

Often, we may wish to iterate across the members of a collection and apply some logic only when the element meets some criterion. This is readily handled with a conditional statement in the closure.

EXAMPLE 09
Conditional elements

```
          // even values only
[1, 2, 3, 4].each {num -> if(num % 2==0) println num}

          // staff at least 25 years old
['Ken' : 21, 'John' : 22, 'Sally' : 25].each {staff ->
    if(staff.value>=25) println staff.key
}
['Ken' : 21, 'John' : 22, 'Sally' : 25].each {staffName, staffAge ->
    if(staffAge>=25) println staffName
}
          // only lowercase letters
'Ken'.each {letter -> if(letter>='a' && letter<='z') println letter}
```

The output from the script is:

```
2
4
Sally
Sally
e
n
```

Observe the two examples to find those staff members who are at least 25 years old. In both cases, we iterate over a `Map` and apply a closure to each member of the `Map`. In the first, the closure parameter `staff` is a `Map.Entry` that includes the key and the value pair. Hence, to check the age, we use `staff.value` in the `Boolean` expression. In the second example, the closure has two parameters representing the two `Map.Entry` elements, namely, the key (`staffName`) and the value (`staffAge`).

The find method finds the first value in a collection that matches some criterion. The condition to be met by the collection element is specified in the closure that must be some Boolean expression. The find method returns the first value found or null if no such element exists. The signature for this method is:

```
Object find(Closure closure)
```

EXAMPLE 10
Illustrations of the find method and closures

```
          // locate the value 7
def value=[1, 3, 5, 7, 9].find {element->element>6}
println "Found: ${value}"

          // locate no value (null)
value=[1, 3, 5, 7, 9].find {element->element>10}
println "Found: ${value}"

          // first staff member over 21
value=['Ken' : 21, 'John' : 22, 'Sally' : 25].find {staff->staff.value>21}
println "Found: ${value}"
```

Output from this script is:

```
Found: 7
Found: null
Found: Sally=25
```

Notice that when we apply find to a Map, the return object is a Map.Entry. It would not, in this case, be appropriate to use a pair of parameters for the key and the value, as in:

```
value=['Ken' : 21, 'John' : 22, 'Sally' : 25].find {key, value -> value > 21}
```

since we are then not able to specify what is returned, the key or the value.

Whereas the find method locates the first item (if any) in a collection that meets some criterion, the method findAll finds all the elements, returning them as a List. The signature for this method is:

```
List findAll(Closure closure)
```

It finds all values in the receiving object matching the closure condition. Example 11 gives some examples of using findAll. The second illustration reveals how simple closures can be combined to implement more complex algorithms. The merit of this approach is that each closure is relatively simple to express.

EXAMPLE 11
Illustrations of the method findAll and closures

```
        // Find all items that exceed the value 6
def values=[1, 3, 5, 7, 9].findAll {element->element>6}
values.each {println it}

        // Combine closures by piping the result of findAll
        // through to each
[1, 3, 5, 7, 9].findAll {element->element>6}.each {println it}

        // Apply a findAll to a Map finding all staff over the age of 24
values=['Ken' : 21, 'John' : 22, 'Sally' : 25].findAll {staff->staff.value>24}
values.each {println it}
```

Again, applying findAll to a Map delivers a List of Map.Entry elements. This is shown by the final line of output from the script:

```
7
9
7
9
Sally=25
```

Two other related methods that take a closure argument are any and every. Method any iterates through each element of a collection checking whether a Boolean predicate is valid for at least one element. The predicate is provided by the closure. Method every checks whether a predicate (a closure that returns a true or false value) is valid for all the elements of a collection, returning true if they do so and false otherwise. The signatures for these methods are:

```
boolean any(Closure closure)
boolean every(Closure closure)
```

◆

Example 12 shows some representative examples.

EXAMPLE 12
Examples of methods any and every

```
        // Any number over 12?
def anyElement=[11, 12, 13, 14].any {element -> element>12}
println "anyElement: ${anyElement}"

        // Are all values over 10?
def allElements=[11, 12, 13, 14].every {element -> element>10}
println "allElements: ${allElements}"

        // Any staff member over the age of 30?
def anyStaff=['Ken' : 21, 'John' : 22, 'Sally' : 25].any {staff -> staff.value>30}
println "anyStaff: ${anyStaff}"
```

When we run this script, we get the output:

```
anyElement: true
allElements: true
anyStaff: false
```

◆

Two further methods that we wish to consider are `collect` and `inject`. Again, both have a closure as a parameter. The method `collect` iterates through a collection, converting each element into a new value using the closure as the transformer. The method also returns a new `List` of the transformed values. It has the signature:

```
List collect(Closure closure)
```

Example 13 shows simple uses for this method.

EXAMPLE 13
Sample uses of `collect` method

```
            // Square of the values
def list=[1, 2, 3, 4].collect {element->return element * element}
println "list: ${list}"

            // Square of the values (no explicit return)
list=[1, 2, 3, 4].collect {element->element * element}
println "list: ${list}"

            // Double of the values (no explicit return)
list=(0..<5).collect {element->2 * element}
println "list: ${list}"

            // Age by one year
def staff=['Ken' : 21, 'John' : 22, 'Sally' : 25]
list=staff.collect {entry -> ++entry.value}
def olderStaff=staff.collect {entry -> ++entry.value; return entry}
println "staff: ${staff}"
println "list: ${list}"
println "olderStaff: ${olderStaff}"
```

Running this, we get the output:

```
list: [1, 4, 9, 16]
list: [1, 4, 9, 16]
list: [0, 2, 4, 6, 8]
staff: [Sally:27, John:24, Ken:23]
list: [26, 23, 22]
olderStaff: [Sally=27, John=24, Ken=23]
```

The third example of method collect is applied to a Range. This is permissible since the Range interface extends the List interface and can, therefore, be used in place of Lists. Observe also the illustration that iterates across the staff collection, increasing the age by 1. The returned value is a List of the new age values from the Map. The recipient Map object referred to as staff is also modified by the closure. The final example that assigns to oldStaff builds a List of Map.Entrys, with the age increased again.

◆

Example 14 further illustrates the collect method. Note the method map, which applies the closure parameter to the collect method over the list parameter. The map method is used for doubling, tripling, and for finding those that are even-valued elements of a list of integers. We shall find further uses for this map algorithm (see Appendix J).

EXAMPLE 14
Further examples of collect

```
          // A series of closures
def doubles={item -> 2 * item}
def triples={item -> 3 * item}
def isEven={item -> (item % 2==0)}

          // A method to apply a closure to a list
def map(clos, list) {
    return list.collect(clos)
}
          // Uses:
println "Doubling: ${map(doubles, [1, 2, 3, 4])}"
println "Tripling: ${map(triples, [1, 2, 3, 4])}"
println "Evens: ${map(isEven, [1, 2, 3, 4])}"
```

The output from the script is:

```
Doubling: [2, 4, 6, 8]
Tripling: [3, 6, 9, 12]
Evens: [false, true, false, true]
```

◆

The final method that we explore in this section is entitled inject. This method iterates through a List, passing the initial value to the closure together with the first element, and then passing into the next iteration the computed value from the previous closure and the next element of the collection, and so on. Here is its signature:

```
Object inject(Object value, Closure closure)
```

Here are three examples of finding the factorial of 5.

EXAMPLE 15
Factorial of 5

```
        // Direct usage
def factorial=[2, 3, 4, 5].inject(1) {previous, element -> previous * element}
println "Factorial(5): ${factorial}"

        // Equivalence
def fact=1
[2, 3, 4, 5].each {number->fact*=number}
println "fact: ${fact}"

        // Named list
def list=[2, 3, 4, 5]
factorial=list.inject(1) {previous, element->previous * element}
println "Factorial(5): ${factorial}"

        // Named list and closure
list = [2, 3, 4, 5]
def closure = {previous, element -> previous * element}
factorial = list.inject(1, closure)
println "Factorial(5): ${factorial}"
```

The output is:

```
Factorial(5): 120
Fact: 120
Factorial(5): 120
Factorial(5): 120
```

◆

The segment of code that uses the variable `fact` aims to show that the result of method `inject` can be achieved using an `each` iterator method. First, the variable `fact` is assigned the value of the first parameter to inject (here, 1). Then, we iterate through each element of the `List`. For the first value (`number` = 2), the closure evaluates `fact*=number`, that is, `fact=1*2=2`. For the second value (`number` = 3), the closure again evaluates `fact*=number`, that is, `fact=2*3=6`, and so on.

9.3 OTHER CLOSURE FEATURES

Since a closure is an `Object`, it can be a parameter to a method. In Example 16, the simple `filter` method expects two parameters, a `List` and a `closure`. The method finds all those elements of the list that satisfy the condition specified by the `closure` using, of course, method `findAll`. The program then demonstrates two uses for the method.

EXAMPLE 16
Closures as method parameters

```
        // Find those items that qualify
def filter(list, predicate) {
    return list.findAll(predicate)
}

        // Two predicate closure
def isEven={x -> return (x % 2==0)}
def isOdd={x -> return ! isEven(x)}

def table=[11, 12, 13, 14]

        // Apply filter
def evens=filter(table, isEven)
println "evens: ${evens}"

def odds=filter(table, isOdd)
println "odds: ${odds}"
```

The output reveals that the variable evens is a `List` of all the even-valued integers from the `table`.

```
evens: [12, 14]
odds: [11, 13]
```

Closures can also be parameters to other closures. Example 17 introduces a closure `takeWhile` that delivers those elements from the beginning of a `List` that meets some criteria defined by the closure parameter named `predicate`.

EXAMPLE 17
Closures as parameters to closures

```
        // Find initial list that conforms to predicate
def takeWhile={predicate, list ->
    def result=[]
    for(element in list) {
        if(predicate(element)) {
            result<<element
        } else
            return result
    }
    return result
}

        // Two predicate closures
def isEven={x -> return (x % 2==0)}
def isOdd={x -> return ! isEven(x)}

def table1=[12, 14, 15, 18]
def table2=[11, 13, 15, 16, 18]

        // Apply takeWhile
def evens=takeWhile.call(isEven, table1)
println "evens: ${evens}"
```

```
def odds=takeWhile(isOdd, table2)
println "odds: ${odds}"
```

The variable evens has the even-valued integer prefix from table1. This is shown by the program output:

```
evens: [12, 14]
odds: [11, 13, 15]
```

◆

In Example 18, the method multiply is defined. It accepts a single parameter and returns a closure. This closure multiplies two values, one of which is pre-set to the value of the method parameter. The variable twice is now a closure that returns double the value of its single parameter. In a similar manner, the closure multiplication accepts a single parameter and returns a closure. Like method multiply, the closure it returns multiplies its parameter by some predefined value. The closure quadruple multiplies its single parameter by the value 4.

EXAMPLE 18
Closures as return values

```
        // Method returning a closure
def multiply(x) {
    return {y -> return x * y}
}

def twice=multiply(2)

println "twice(4): ${twice(4)}"

        // Closure returning a closure
def multiplication={x -> return {y -> return x * y}}

def quadruple=multiplication(4)

println "quadruple(3): ${quadruple(3)}"
```

The output demonstrates that the closure twice does indeed double its parameter while the closure quadruple multiplies its parameter by 4:

```
twice(4): 8
quadruple(3): 12
```

The final example we consider demonstrates that a closure may contain other nested closure definitions. In Example 19, we define the closure selectionSort, which sorts a list of integers into ascending order. To implement this closure, we are required to locate the smallest item of the unsorted tail region of the list and move it to the front. Moving the item to the front of the tail region actually involves swapping the front item with the smallest item. Hence, we implement the closure `selectionSort` with two local closures, `minimumPosition` and `swap`. The latter does the exchange we require, and the former finds the smallest item in the tail region of the `List`.

EXAMPLE 19
Selection sort

```
def selectionSort = {list ->

    def swap = {sList, p, q ->
        def temp = sList[p]
        sList[p] = sList[q]
        sList[q] = temp
    }
    def minimumPosition = {pList, from ->
        def mPos = from
        def nextFrom = 1 + from
        for(j in nextFrom..<pList.size()) {
            if(pList[j] < pList[mPos])
                mPos = j
        }
        return mPos
    }

    def size = list.size() -1
    for(k in 0..<size) {
        def minPos = minimumPosition(list, k)
        swap(list, minPos, k)
    }

    return list
}

def table = [13, 14, 12, 11, 14]

def sorted = selectionSort(table)

println "sorted: ${sorted}"
```

Running the program produces the desired result:

```
sorted: [11, 12, 13, 14, 14]
```

◆

We have to be especially diligent about the scope rules of variables and parameters when working with closures. See Appendices H and J for further discussion on these and other aspects of closures.