CHAPTER 7

METHODS

A *method* is a name given to a segment of code that can be executed or *called* one or more times in a program. Methods may also be given *parameters* that act as input values to the method call. Each method call may use different actual parameters that determine the effect of the method when it is executed.

Methods in Groovy partition large programs into smaller manageable units, thus simplifying the programming task. Each method is responsible for a particular functionality required in the application. One method can call or execute any other method. Thus, a task represented by one method can be partitioned into subtasks realized by other submethods. Further, methods developed in one program may be incorporated into other programs, avoiding the need to reprogram them.

Groovy methods as described in this chapter are synonymous with *functions*, *procedures*, or *subroutines* found in other programming languages.

7.1 METHODS

A method is defined using the keyword `def`. The simplest form of a method definition is one with no parameters as shown here:

```
def methodName() {
   // Method code goes here
}
```

Method names are presented as program identifiers (see Chapter 2). If a method takes no parameters that is signaled by `()`, which cannot be omitted.

EXAMPLE 01
Simple method definition

```
def greetings() {
    println 'Hello and welcome'
}

greetings()
```

Here, the method is named `greetings`. The method code involves printing a simple greeting to the user. The method is then invoked using the *method call* `greetings()`. The program output is:

```
Hello and welcome
```

◆

This method may also be written as in Example 02.

EXAMPLE 02
Method with three statements

```
def greetings() {
    print 'Hello'
    print ' and '
    println 'welcome'
}

greetings()
```

◆

On this occasion, method `greetings` has three statements. Each is a separate invocation of the print statement. Here, each statement is given on a separate line. This generally improves the readability of the code and is the style used throughout this textbook. If we want two or more statements on a line, we must use a semicolon separator, as in:

EXAMPLE 03
Multiple statements on a single line

```
def greetings() {
    print 'Hello'; print ' and '
    println 'welcome'
}

greetings()
```

◆

Consider now a method that includes some variables. The program is required to read two integer values and print them in reverse order. To achieve this effect, the method must have two variables as repositories for the data values.

EXAMPLE 04
Method variables

```
import console.*

def reverse() {
  print 'Enter the two integer values: '
  def first=Console.readInteger()
  def second=Console.readInteger()
  println "Reversed values: ${second} and ${first}"
}

reverse()        // now call it
```

Running the script might produce the following (with the user input shown as bold and italic):

```
Enter the two integer values: 12
34
Reversed values: 34 and 12
```

◆

The next example is similar to the last. It reads some data, processes it, and displays the results of its computation. The processing involves some arithmetic operations. The program reads three integer values representing a 24-hour clock time expressed as hours, minutes, and seconds. This time is converted to its total number of seconds.

EXAMPLE 05
Converting a clock time

```
import console.*

def processTime() {
  print 'Enter the time to be converted: '
  def hours=Console.readInteger()
  def minutes=Console.readInteger()
  def seconds=Console.readInteger()
  def totalSeconds=(60*hours+minutes)*60+seconds
  println "The original time of: ${hours} hours, ${minutes} minutes
  and ${seconds} seconds"
  println "Converts to: ${totalSeconds} seconds"
}

processTime()    // now call it
```

Running this program might produce:

```
Enter the time to be converted: 1
2
3
The original time of: 1 hours, 2 minutes and 3 seconds
Converts to: 3723 seconds
```

7.2 METHOD PARAMETERS

A method is more generally useful if its behavior is determined by the value of one or more parameters. We can transfer values to the called method using *method parameters.* A method with three parameters appears as:

```
def methodName(para1, para2, para3) {
  // Method code goes here
}
```

The method parameters appear as a list of *formal parameter* names enclosed in parentheses following the method name. The parameter names must differ from each other.

To illustrate, let us revisit our method `greetings`. The first version simply printed a fixed message. We can personalize its behavior if we provide a parameter representing the name of the person we wish to welcome. Here is the new version.

EXAMPLE 06
Method parameters

```
def greetings(name) {
  println "Hello and welcome, ${name}"
}

greetings('John')
```

Running the program produces the output:

```
Hello and welcome, John
```

◆

The actual parameter `'John'` initializes the formal parameter `name`, which the method then prints.

7.3 DEFAULT PARAMETERS

The formal parameters in a method definition can specify *default values.* Where default values are given, these values are used if the caller does not pass them explicitly. Default parameter values are shown as assignments. Where default parameters are introduced in a method definition, then they may only occur after nondefault parameters. That is, default parameters may only be used for parameters at the end of the formal parameter list. Default and nondefault parameters may not be intermixed. For example, in the method:

```
def someMethod(para1, para2=0, para3=0) {
  // Method code goes here
}
```

the second and third parameters have been given default values.

The someMethod may then be called with one, two, or three actual parameters. If only one actual parameter is supplied, the other two default to zero. If two actual parameters are used, the final parameter is zero. The method call must include at least one actual parameter and at most three actual parameters. An illustration of default parameters is shown in Example 07.

EXAMPLE 07
Default parameters

```
def greetings(salutation, name='Ken') {
  println "${salutation} ${name}"
}
greetings('Hello', 'John')          // Hello John
greetings('Welcome')                // Welcome Ken
```

When we execute this script, we see that the second call to method greetings assumes that the name parameter has the default value 'Ken':

```
Hello John
Welcome Ken
```

◆

7.4 METHOD RETURN VALUES

A method can also return a value to its caller. This is achieved with the *return statement* of the form:

```
return expression
```

The statement indicates that control is to return immediately from the method to the caller, and that the value of the expression is to be made available to the caller. This value may be captured with an appropriate assignment.

The return statement is illustrated in Example 08. The method hmsToSeconds obtains a clock time through its parameters, and converts it into seconds. On this occasion, the method then returns the computed value to the caller. The calling code calls this method and prints the returned value.

EXAMPLE 08
Method return values

```
import console.*

def hmsToSeconds(h, m, s) {
  return (60*h+m)*60+s
}
       // Get the input from the user.
print 'Enter hours to convert: '
def hours=Console.readInteger()
print 'Enter minutes to convert: '
def minutes=Console.readInteger()
print 'Enter seconds to convert: '
def seconds=Console.readInteger()

     // Now call the method.
def total=hmsToSeconds(hours, minutes, seconds)
println "Total number of seconds=${total}"
```

A session running this program could produce:

```
Enter hours to convert: 1
Enter minutes to convert: 2
Enter seconds to convert: 3
Total number of seconds=3723
```

◆

Finally, we note that the return keyword is optional. If it is omitted, then the value of the final statement is the value returned. Example 09 repeats the previous example, with method hmsToSeconds revised.

EXAMPLE 09
Implicit returns

```
import console.*

def hmsToSeconds(h, m, s) {
  def totalSeconds=(60*h+m)*60+s
  totalSeconds
}

     // Get the input from the user.
print 'Enter hours to convert: '
def hours=Console.readInteger()
print 'Enter minutes to convert: '
def minutes=Console.readInteger()
print 'Enter seconds to convert: '
def seconds=Console.readInteger()
```

```
    // Now call the method.
def total = hmsToSeconds(hours, minutes, seconds)
println "Total number of seconds = ${total}"
```

◆

7.5 PARAMETER PASSING

Method parameters in Groovy use a parameter passing strategy known as *pass by value*. This means that the value of the actual parameter is used to initialize the value of the formal parameter. For example, in the previous program the actual parameter hours is used to initialize the formal parameter h in the call to the method hmsToSeconds. A similar arrangement applies to the other two formal parameters.

In Chapter 2 (Section 2.6), we discussed how variables are object references. The variable refers to that part of memory occupied by the object. Figures 2.1 through 2.4 illustrated these concepts. This means that when a method formal parameter is initialized with its corresponding actual parameter, it is actually aliased with it. Figure 2.2 describes this effect. Hence, in Example 09, at the point of call of the hmsToSeconds method, the formal parameter h is an alias for the actual parameter hours.

An implication of this arrangement is that any assignment to a formal parameter within a method body establishes a new object for the formal parameter to refer. Consequently, the corresponding actual parameter is unaffected by this. We demonstrate this in Example 10.

EXAMPLE 10
Parameter aliasing

```
def printName(name) {
  println "Name (at entry): ${name}"
  name = 'John'
  println "Name (after assignment): ${name}"
}

def tutor = 'Ken'
printName(tutor)

println "Tutor: ${tutor}"
```

When we run this program, the output produced is:

```
Name (at entry): Ken
Name (after assignment): John
Tutor: Ken
```

◆

The method printName is defined in terms of a formal parameter name. First, the method prints this value at its point of entry into the method. It then assigns a new String object to this formal parameter variable. Following the consequence of Figure 2.2, the name parameter now references a new String object with the value 'John'. The final print statement in the method reveals that it does indeed have this new value. In the code, the printName method is called with the object ('Ken') referenced by the variable tutor as the actual value. Because this is also referenced by the name parameter at the point of entry to the method, then this is why Ken is the first line printed. After return from the printName method, the program finishes by printing the value of tutor. We see that this is unaffected by the change to the formal parameter.

A consequence of this aliasing of formal and actual parameters is that the swap method, as defined in Example 11, does not produce the effect that we might expect. The definition for method swap suggests that the parameters x and y have their values interchanged. This occurs during the execution of the method but, as has been explained, these changes are not reflected in the corresponding actual parameters. The execution of this program reveals what happens.

```
Enter the first value: 12
Enter the second value: 34
First: 12
Second: 34
```

EXAMPLE 11
Interchange method

```
import console.*

def swap(x, y) {
  def temp = x
  x = y
  y = temp
}

print 'Enter the first value: '
def first = Console.readInteger()
print 'Enter the second value: '
def second = Console.readInteger()

     // Now call the swap method
swap(first, second)
println "First: ${first}"
println "Second: ${second}"
```

◆

7.6 SCOPE

The method `processTime` in Example 05 has four variables: `hours`, `minutes`, `seconds`, and `totalSeconds`. These are referred to as *local variables* since they are introduced in this method. Local variables have the method body in which they are defined as their *scope*. This means that they can only be referenced in their scope and have no existence outside of this scope. Hence, elsewhere in the code, these variables have no meaning.

Earlier, we noted that method parameters appear as a list of formal parameter names enclosed in parentheses following the method name. The parameter names must differ from each other and they too represent names that are local to the method. When the method is called, these formal parameters are initialized with the values of the corresponding actual parameters. The formal parameters also behave as local variables with the method body as their scope.

This same mechanism is used for variables defined outside of a method, such as the variables `first` and `second` used in Example 11. Appendix B describes how a Groovy script is compiled into a Java class with a `run` method. Groovy variables defined outside a method using `def` are effectively local to the generated `run` method and cannot be referenced by any of our Groovy methods (see Example 12).

Example 12 includes the method `printName` and the defined variable `tutor`. From the preceding paragraph, we know that the variable `tutor` is local to the generated `run` method and cannot, therefore, be referenced in the method `printName` (see commented line in the method body).

EXAMPLE 12
Variable scope

```
def printName(name) {
  println "Name (at entry): ${name}"
  //name = tutor
  name = 'Ken'
  println "Name (after assignment): ${name}"
}

def tutor = 'Ken'

printName('John')

//println "Name: ${name}"          // ERROR: No such property
```

When we run this program, we have the output:

```
Name (at entry): John
Name (after assignment): Ken
```

◆

The two lines of output show that the formal parameter `name` is first initialized with the actual parameter value `'John'`. Then it is changed by assignment to the `String` literal `'Ken'`.

Note also the commented line at the end of the listing. The parameter `name`, like any variables defined within the body of the `printName` method, has the method body as its scope. Hence, these variables can only be referenced within the method. Any attempt to reference the `name` variable elsewhere in the code will produce an error as shown.

This scoping rule comes up again in Example 13. At the point at which a variable is defined is irrelevant, they still cannot be referenced in the body of a method.

EXAMPLE 13
Variables and methods in same scope

```
def tutor='Ken'

def printName(name) {
  println "Name: ${name}"
  //println "Tutor: ${tutor}"
}

printName('John')
```

The program output is, as we would expect, with the `tutor` variable inaccessible to the method `printName` (see commented line in the method body).

```
Name: John
```

◆

7.7 COLLECTIONS AS METHOD PARAMETERS AND RETURN VALUES

A Groovy method can accept a collection parameter, such as a `List`, and return a collection value. In Example 14, the method `sort` is used to order a `List` of values. If the second parameter is the `Boolean` value `true`, then the `List` is sorted into ascending order. If the `Boolean` value is `false`, then the `List` is ordered in descending order.

EXAMPLE 14
`List` parameter and return

```
def sort(list, ascending=true) {
  list.sort()
  if(ascending==false)
     list=list.reverse()
  return list
}
```

```
def numbers = [10, 5, 3, 6]

assert(sort(numbers, false) == [10, 6, 5, 3])
```

◆

Here, rather than display the result, we have shown the `assert` keyword by which we make an assertion about the value returned from the `sort` method. Since the method does indeed produce the list `[10, 6, 5, 3]`, the assertion is true and the program produces no output. Had the assertion been false, then an `AssertionError` would be raised and reported. A detailed discussion of assertions is given in Chapter 15.

Further aspects of methods, such as recursive methods and statically typed method parameters and return values, are examined in Appendix G.

7.8 EXERCISES

The reader should consult the supporting Appendix G before completing these exercises.

1. Prepare and test a method entitled `square` that returns the square of its single parameter.
2. Pre-decimal coinage in Great Britain had 12 pence in a shilling and 20 shillings in a pound. Write methods to add and subtract two of these monetary amounts. Both methods will require six parameters. The first three represent the first monetary amount and the remaining three the other amount. Each method should return the value expressed as pence.
3. Write and test a method to determine whether a given time of day is before another. Each time is represented by a triple of the form 11, 59, AM, or 1, 15, PM.
4. The values 1, 2, 4, 8, 16, ... are powers of the value 2. First, develop a method `isEven` that determines whether its single integer parameter is an even value. Then, using `isEven`, develop a recursive method `isPowerOfTwo` that determines if its single parameter is a power of 2.
5. Using (only) the methods `head` and `tail` (see Appendix G, Example 02), develop a method `length` that determines the number of elements in a list given as its parameter.
6. Using (only) the methods `head`, `tail`, and `cons`, develop a method `reverse` that reverses the elements of a `List` given as its parameter.