CHAPTER 8

FLOW OF CONTROL

The execution of a program statement causes an action to be performed. The programs we have developed execute one statement after another in a sequential manner. Because of this execution ordering of the statements, we describe the program logic as *sequential*. We can also create abstract actions with method definitions and then treat them as if they, likewise, were simple statements through their method calls. The statements we have explored include the assignment, input/output, and method calls.

Additionally, statements are provided in Groovy to alter the flow of control in a program's logic. They are then classified into one of three program *flow of control* structures:

- sequence
- selection
- iteration

8.1 WHILE STATEMENT

The fundamental iteration clause is the *while statement*. The syntax of the `while` statement is:

```
while(condition) {
  statement #1
  statement #2
  ...
}
```

The `while` statement is executed by first evaluating the *condition* expression (a `Boolean` value), and if the result is `true`, then the statements are executed. The entire process is repeated, starting once again with reevaluation of the condition. This loop continues until the condition evaluates to `false`. When the condition is `false`, the loop terminates. The program logic then continues with the statement immediately following the `while` statement. The group of statements is known as a *compound statement* or *block*.

Where only one statement is to be controlled by a `while` loop, the single statement may be presented as:

```
while(condition)
  statement
```

The program shown as Example 01 prints the values from 1 to 10 inclusive. Each iteration through the loop prints the current value of the variable count, and then increments it. The count is first set to the start value 1. The condition in the `while` statement specifies that the loop continues provided the count does not exceed the value of `LIMIT`.

EXAMPLE 01
`while` statement

```
          // Set limit and counter
def LIMIT=10
def count=1

println 'Start'

while(count<=LIMIT) {
  println "count: ${count}"
  count++
}

println 'Done'
```

The program's output is:

```
Start
count: 1
count: 2
count: 3
count: 4
count: 5
count: 6
count: 7
count: 8
count: 9
count: 10
Done
```

◆

Conventionally, we denote variables with fixed values by capitalization. They are generally known as *symbolic constants.* The value in defining such variables is that they document a given value with their name. Further, the definition occurs only once in the code, and only a single change is required to modify that value.

A typical use for a `while` statement is to loop over a series of statements an indeterminate number of times. A statement in the loop usually affects the condition that controls the looping. Example 02 demonstrates a program that reads an unknown number of positive integers, forming a running total for their values. The user enters any negative number to end the input loop.

EXAMPLE 02
Sum of a series of positive integers

```
import console.*

    // Running total
def sum=0

print 'Enter first value: '
def data=Console.readInteger()
while(data>=0) {
  sum+=data
  print 'Enter next value: '
  data=Console.readInteger()
}

println "The sum is: ${sum}"
```

A sample session with this program is:

```
Enter first value: 1
Enter next value: 2
Enter next value: 3
Enter next value: 4
Enter next value: -1
The sum is: 10
```

◆

Note that in this example, if the first input value is negative, then the loop will never be obeyed and the program will finish with a zero sum. Because of this, a `while` statement is often described as causing the statement(s) under its control to be obeyed *zero or more times.*

8.2 FOR STATEMENT

The *for statement* in Groovy can be used to iterate over a `Range`, a collection (`List`, `Map`, or array; see Chapter 4 and Appendix E) or a `String`.

```
for(variable in range) {
   statement #1
   statement #2
   ...
}
```

```
for(variable in collection) {
   statement #1
   statement #2
   ...
}
```

```
for(variable in string) {
   statement #1
   statement #2
   ...
}
```

Example 03 repeats the first example in this chapter. Using a for statement is a more appropriate looping construct to use when the number of times to repeat the logic is known.

EXAMPLE 03
for statement

```
def LIMIT=10

println 'Start'

for(count in 1..LIMIT)
  println "count: ${count}"

println 'Done'
```

◆

The next example demonstrates a for statement applied to a List.

EXAMPLE 04
Looping through a List

```
      // List
println 'Start'

for(count in [11, 12, 13, 14])
  println "count: ${count}"

println 'Done'
```

The output from this program is:

```
Start
count: 11
count: 12
count: 13
count: 14
Done#
```

◆

We can also iterate through the elements of a Map. In Example 05, the total age of the employees is recorded in a Map. It is worth noting that the loop variable

staffEntry. Since we are looping through all the entries in a Map, then every item is a Map.Entry (see JDK documentation) object that references both the key and value. Hence, in the loop, we refer to the staff member's age with staffEntry.value.

EXAMPLE 05
Looping through a Map

```
    // Staff name and age
def staff=['Ken' : 21, 'John' : 25, 'Sally' : 22]

def totalAge=0
for(staffEntry in staff)
   totalAge+=staffEntry.value

println "Total staff age: ${totalAge}"
```

The output produced is:

```
Total staff age: 68
```

◆

Finally, we show how we can also iterate through the characters that compose a String. In Example 06, name is processed character by character and inserted into a List.

EXAMPLE 06
Looping through a String

```
def name='Kenneth'
def listOfCharacters=[]

for(letter in name)
   listOfCharacters<<letter

println "listOfCharacters: ${listOfCharacters}"
```

The output is:

```
listOfCharacters: ["K", "e", "n", "n", "e", "t", "h"]
```

◆

8.3 IF STATEMENT

The general form of the *if statement* is:

```
if(condition) {
   statement #1a
   statement #1b
   ...
} else {
```

```
        statement #2a
        statement #2b
        ...
    }
```

where `if` and `else` are reserved words. If the *condition* evaluates to the `Boolean` value `true`, then the compound statement starting with `statement #1a` is executed and control is then passed to the statement following the `if` statement. If the value of the condition is `false`, then the compound statement starting with `statement #2a` is executed and again control continues with the statement after the `if` statement. As earlier, a single statement may replace either of the compound statements.

An if statement offers a means of selecting one of two distinct logical paths through a program. Sometimes, we wish to select whether to execute some program code. We achieve this through a shortened version of the `if` statement:

```
if(condition) {
    statement #1
    statement #2
    ...
}
```

If the condition evaluates to `true`, then the compound statement is executed and the program continues with the statement following the `if` statement. If the condition evaluates to `false`, then the compound statement is ignored and the program continues with the next statement. As before, a single statement may replace the compound statement.

In Example 07, the program reads two integers and prints them in ascending order. This is achieved by using an `if-else` statement to select the correct `print` statement:

EXAMPLE 07
A simple `if` statement

```
import console.*

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

if(first<second)
    println "${first} and ${second}"
else
    println "${second} and ${first}"
```

An interactive session with this program might produce:

```
Enter first value: 34
Enter second value: 12
12 and 34
```

◆

Example 08 repeats this exercise. This time, the program employs the shortened version of the `if` statement. If the condition determines that the first value is greater than the second, then the values are interchanged.

EXAMPLE 08
Interchange two values

```
import console.*

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

     // Exchange the order
if(first>second) {
   def temp=first
   first=second
   second=temp
}

println "${first} and ${second}"
```

An execution of this program produces:

```
Enter first value: 34
Enter second value: 12
12 and 34
```

◆

Various combinations of `if` statements are allowed. For example, the statement associated with the `else` clause may be another `if` statement. This can be repeated any number of times. Such a construct is used to select from among a number of logical paths through the code. To illustrate this, consider a program fragment to read an examination score (any value from 0 to 100, inclusive) and assign a letter grade. The grading scheme that applies is shown by:

Score	*Grade*
70–100	A
60–69	B
50–59	C
40–49	D
0–39	E

A chain of `if–else` statements can then describe the necessary processing:

```
if(score>=70)
    grade='A'
else if(score>=60)
    grade='B'
else if(score>=50)
    grade='C'
else if(score>=40)
    grade='D'
else
    grade='E'
```

8.4 SWITCH STATEMENT

The `if–else` statement chain in the last section occurs so frequently that a special statement exists for this purpose. This is called the *switch statement* and its form is:

```
switch(expression) {
  case expression #1:
    statement #1a
    statement #1b
    ...
  case expression #2:
    statement #2a
    statement #2b
    ...

  ...
  case expression #N:
    statement #Na
    statement #Nb
    ...
  default:
    statement #Da
    statement #Db
    ...

}
```

where `switch`, `case`, and `default` are Groovy keywords. The `default` clause and its statements are optional. The control expression enclosed in parentheses is evaluated. This value is then compared, in turn, against each of the *case expressions*. If a match is made against one of the case expressions, then all statements from that case clause through to the end of the `switch` are executed. If no match is made, then the default statements are obeyed if a default clause is present. Example 09 illustrates the basic behavior of a `switch` statement.

EXAMPLE 09
Basic switch behavior

```
def n=2
switch(n) {
    case 1: println 'One'
    case 2: println 'Two'
    case 3: println 'Three'
    case 4: println 'Four'
    default: println 'Default'
}
println 'End of switch'
```

The control expression is simply the value of the variable `n`. When evaluated, it is compared, in turn, to the value of the case expressions. A match is found at `case 2` and the output from the code is:

```
Two
Three
Four
Default
End of switch
```

◆

Normally, the statements of a case label are intended to be mutually exclusive. Having selected the matching case expression, we normally wish for only the corresponding statements to be obeyed, and then control passed to the statement following the `switch` statement. We achieve this with a *break statement* that, in the context of a `switch` statement, immediately terminates it and continues with the statement after the `switch`. Example 10 illustrates.

EXAMPLE 10
`switch` and `break` statement

```
def n=2
switch(n) {
  case 1:
    println 'One'
    break
  case 2:
    println 'Two'
    break
```

```
    case 3:
      println 'Three'
      break
    case 4:
      println 'Four'
      break
    default:
      println 'Default'
      break
}
println 'End of switch'
```

Running this program produces:

```
Two
End of switch
```

◆

A switch statement can be used as a replacement for the chain of if statements shown at the end of the previous section. The code in Example 11 shows a switch statement based on the value of the examination score. Each case clause matches against a range representing the grade. This time no default has been used.

EXAMPLE 11 switch and a range

```
import console.*

print 'Enter examination score: '
def score=Console.readInteger()
def grade

switch(score) {
  case 70..100:
    grade='A'
    break
  case 60..69:
    grade='B'
    break
  case 50..59:
    grade='C'
    break
  case 40..49:
    grade='D'
    break
```

```
  case 0..39:
    grade = 'E'
    break
}

println "Score: ${score}; grade: ${grade}"
```

Running this program produces:

```
Enter examination score: 50
Score: 50; grade: C
```

◆

The case expressions have been shown as an integer literal or a Range of integer values. In fact, the case expression might be a String, List, regular expression, or object of some class (see Chapter 12). Example 12 shows a switch statement in which the case expressions are Lists. A match is found if the value of the control expression is a member of the collection.

EXAMPLE 12
List case expressions

```
def number = 32

switch(number) {
  case [21, 22, 23, 24] :
    println 'number is a twenty something'
    break
  case [31, 32, 33, 34] :
    println 'number is a thirty something'
    break
  default :
    println 'number type is unknown'
    break
}
```

The output is:

```
number is a thirty something
```

◆

In Example 13, we show a switch statement in which the case expressions are regular expressions. Again, a match is made against the given patterns.

EXAMPLE 13
Regular expressions for `case` labels

```
def number='1234'

switch(number) {
  case ~'[0-9]{3}-[0-9]{4}' :
    println 'number is a telephone number'
    break
  case ~'[0-9]{4}' :
    println 'number is a 4-digit sequence'
    break
  default :
    println 'number type is unknown'
    break
}
```

The output is:

```
number is a 4-digit sequence
```

◆

8.5 BREAK STATEMENT

The *break statement* is used to alter the flow of control inside loops and `switch` statements. We have already seen the `break` statement in action in conjunction with the `switch` statement. The `break` statement can also be used with `while` and `for` statements. Executing a `break` statement with any of these looping constructs causes immediate termination of the innermost enclosing loop.

Example 14 illustrates this idea. The program forms the sum of at most 100 positive integer values. The user provides the values as input. If, at any point, a negative value is entered, then the `for` loop immediately terminates and the value of the summation is printed.

EXAMPLE 14
`for` loop and `break` statement

```
import console.*

def MAX=100
def sum=0

for(k in 1..MAX) {
    print 'Enter next value: '
    def value=Console.readInteger()
    if(value<0)
        break
```

```
    sum+=value
}
println "sum: ${sum}"
```

Running the program produces:

```
Enter next value: 11
Enter next value: 12
Enter next value: 13
Enter next value: 14
Enter next value: -1
sum: 50
```

◆

8.6 CONTINUE STATEMENT

The *continue statement* complements the `break` statement. Its use is restricted to `while` and `for` loops. When a `continue` statement is executed, control is immediately passed to the test condition of the nearest enclosing loop to determine whether the loop should continue. All subsequent statements in the body of the loop are ignored for that particular loop iteration.

In Example 15, the program finds the sum of 10 integers input by the user. If any negative value is entered, it is not included as part of the sum. It does, however, count as an input value.

EXAMPLE 15 `for` loop and `continue` statement

```
import console.*

def MAX=10
def sum=0

for(k in 1..MAX) {
    print 'Enter next value: '
    def value=Console.readInteger()
    if(value<0)
        continue
    sum+=value
}

println "sum: ${sum}"
```

An execution of the program is:

```
Enter next value: 1
Enter next value: 2
Enter next value: 3
Enter next value: 4
Enter next value: -5
Enter next value: -6
Enter next value: -7
Enter next value: 8
Enter next value: 9
Enter next value: 10
sum: 37
```

◆

8.7 EXERCISES

1. Write a method entitled `quotient` that finds the quotient of two positive integers using only the operations of additions and subtraction.
2. Write a program that reads a single positive integer data value and displays each individual digit from that value as a word. For example, the input value 932 should display:

```
932: nine three two
```

3. Write a program that accepts a (24-hour) clock time expressed in hours, minutes, and seconds, and verbalizes the time as suggested by the following values:

```
09:10:00      ten past nine
10:45:00      quarter to eleven
11:15:00      quarter past eleven
17:30:00      half past five
19:50:00      ten to eight
06:12:29      just after ten past six
06:12:30      just before quarter past six
00:17:29      just after quarter past midnight
```

4. The Fibonacci sequence is defined by the following rule: The first two values in the sequence are both 1. Every subsequent value is the sum of the two preceding values. If `fib(n)` denotes the nth value in the sequence, then:

```
fib(1) = 1
fib(2) = 1
fib(n) = fib(n-1) + fib(n-2)
```