Lecture 03: Control of Program Flow

This chapter contains:

3.1 Overview of Java Control Structures

As mentioned in chapter 1 (section 1.7), all programming languages implement control structures. Control structures are the mechanisms through which the programmer controls the logical flow of a computer program. You therefore cannot get very far into programming without using control structures. To review, the basic control structures are:

- Sequential Structures
- Selection Structures
- Iteration Structures
- Recursion

Sequential structures are self-evident; we therefore do not need to discuss this any further, except to emphasize that the order in which instructions are specified is of paramount importance. It is imperative that you gain mastery of how Java implements the other three control structures. This chapter focuses on selection structures and iteration structures. A discussion of recursion will be deferred until you have learned more about, and are more comfortable with Java.

3.2 Boolean Expressions

Boolean expressions were briefly mentioned in the previous chapter (section 2.7), but a full discussion was deferred until now. A Boolean expression is a relational expression that evaluates to **true** or **false**. It is also referred to as a condition. Figure 3.1 shows the Java syntax for Boolean expressions:

Figure 3.1: Java Syntax for Boolean Expressions

```
BooleanExpression ::=
 [!] <BooleanExpression> |
[!] <BooleanVariable> |
<Variable> <BooleanOperator> Variable> |
 <Variable> <BooleanOperator> <Expression> |
< Expression > <BooleanOperator> <Variable> |
< Expression > <BooleanOperator> < Expression> |
< Expression > <BooleanOperator> < Literal >  |
<Variable> <BooleanOperator> < Literal >
```

From the state syntax, please note the following:

1. The operator specified must be a valid Boolean operator. Valid Boolean operators and their meanings were provided in chapter 2 (section 2.6.2); in the interest of clarity, they are repeated in figure 3.2.
2. Expressions used to construct Boolean expressions may themselves be arithmetic expressions. The important thing to note is that the Boolean expression must evaluate to **true** or **false**.
3. Boolean operators observe the precedence shown in figure 3.3 (which is a subset of the operator precedence table shown in figure 2.17 of the previous chapter). The *bracket* (a pair of parentheses) may be used to force a different precedence, and therefore takes precedence over all other operators.
4. Boolean expressions do not occur on their own, but are usually used in selection or iteration statements. Figure 3.4 provides some examples of valid and invalid Boolean expressions.

Figure 3.2: Commonly Used Boolean Operators

Operator	Meaning
(…)	bracket
<	Less than
<=	Less than or equal to
==	Equal
!=	Not equal
>	Greater than
>=	Greater than or equal to
||	OR
&&	AND
!	NOT
&	Unconditional (bitwise) AND
|	Unconditional (bitwise) OR
?:	Conditional operator (this is a ternary operator, requiring three operands)

Figure 3.3: Boolean Operator Precedence

```
( )
!
<  <=  >  >=
==  !=
&
|
&&
||
?:
```

Figure 3.4: Examples of Boolean Expressions

// Assume the following declarations: int x, y, z; String StringA, StringB; boolean GreatDay;	
GreatDay	Valid
(x = y % x) <= (y*z)	Valid
x = y – 8 * x * x;	Invalid. This is an assignment statement.
StringA == StringB	Valid except that stings are best compared for equality via the **equals(...)** method
StringA <= (StringB + “the Great Pretender”)	Valid. The string on the left is compared to the string concatenation on the right.
x == StringB	Invalid. You may not mix operands of different types.
(((x * y – z) >= (z*z*z)) && (GreatDay))	Valid.

3.3 Selection Structures in Java

Recall from section 1.7.4 that two important selection structures in programming are the **If-Structure** and the **Case-Structure**. Java implements both structures via the **If-Statement** and the **Switch-Statement** respectively.

3.3.1 The If-Statement

Figure 3.5 shows the BNF syntax for the Java **If-Statement**. Essentially, the statement begins with the keyword **if**, followed by a parenthesized condition (i.e. a Boolean expression), and then the action to be taken. The action to be taken may be one of more valid Java statements. If multiple statements follow the condition, they must be embraced within a *block*, which is signaled by a left curly brace ({) and ended by a right curly brace (}). Another name for a block as described above is a *compound statement*. Optionally, the **If-Statement** may be followed by a corresponding **Else-Statement**. The **Else-Statement** is signaled by the keyword **else**, followed by a simple statement or a compound statement as described for the **If-Statement**. Incidentally, the Java If-Statement is almost identical to the corresponding statement in C++, from which it was derived.

Figure 3.5: Syntax for the Java If-Statement

```
If-Statement ::=
if (<BooleanExpression>) <Statement>; | <CompoundStatement>
[else <Statement>; | <CompoundStatement>]

CompoundStatement ::=
{
  <Statement>;
  [*<Statement>;*]
}
```

From this definition, please note the following:

1. The statements appearing within an **if-loop** may be any valid Java statement, including other **If-Statement(s)**. Thus, we could have nested **If-Statement(s)** to several levels. Although Java allows several levels of nesting, it is prudent to restrict this to eight or less.
2. If you desire to specify more than one statement under a given **if-clause**, you must include a compound statement (called a block). When compound statements are used, statement termination of the block via the semicolon (;) is not required (the right curly brace takes the place of the semicolon). If only one statement is to follow the **if-clause** or **else-clause**, the compound statement block is optional.
3. Ambiguity can be avoided by using compound statements. In fact, it is good programming practice to always use them.
4. If nested loops are used, indent to improve the readability of your program.

3.3.1 The If-Statement (continued)

Example 1: This example illustrates an ambiguity.

```
/* The following is an example of ambiguity: which condition is the else related to? Typically, the
compiler will associate it with the latest if-condition. But based on the text of the message, this would
not coincide with the intent of the programmer. */

if (studentGPA <= 4.0)
   if (studentGPA < 2.5)
      System.out.println("Your performance is below the required level; you must withdraw.\n");
   else
       System.out.println ("Invalid GPA value.\n");

// This problem can be corrected by tightening the logic with the use of a compound statement block.
```

Example 2: This example avoids the ambiguity of Example 1.

```
if (studentGPA <= 4.0)
{
  if (studentGPA < 2.5)
     System.out.println ("Your performance is below the required level; you must withdraw.\n");
  else System.out.println ("You are doing fine.\n");
}
else System.out.println ("Invalid GPA value.\n");

// Note: The compiler treats the block as one statement.
```

Example 3: The following example performs a test for the occurrence of a leap year and prints a message.

```
int year, month, day;
boolean leapYear;
//…
if (year % 400) == 0) || ((year % 4) == 0) && (year % 100) != 0))
   leapYear = true;
else leapYear = false;
//…
if (leapYear && month ==2)
{
   System.out.println ("This is a leap year and it is February.\n");
   System.out.println ("There will be 29 days in the month. My best friend will have a birthday – " +
   " a rare event for her.\n");
}
// …
```

3.3.1 The If-Statement (continued)

Example 4: Here is another example that illustrates nested if-statements.

```
if (studentGPA <= 2.5)
   System.out.println (“Your performance is low; you must withdraw.\n”);
else if (studentGPA <= 3.5)
          System.out.println (“You are doing fairly well, but there is room for improvement.\n”);
       else if (studentGPA <= 4.0)
                 System.out.println (“Excellent!\n”);
               else System.out.println (“Invalid GPA value.\n”);
```

3.3.2 Using the Conditional Operator

The conditional operator is a ternary operator, which, though optional, is quite common. The syntax for its usage is depicted in figure 3.6.

Figure 3.6: BNF Syntax for use of the Conditional Operator

```
<Variable> = <Condition> ? <Expression1> : <Expression2>;
```

Is equivalent to:

```
if <Condition>
  <Variable> =  <Expression1>;
else
  <Variable> =  <Expression2>;
```

While use of this operator is not recommended by this course (due to its lack of clarity), you should recognize and understand it (in the event that you have to modify code that uses it). The equivalent **If-Statement** is much clearer, and is therefore preferred.

Example 5: Here is an example that illustrates the use of the conditional operator.

```
String greetings;
boolean goodWeather;
//…
if (goodWeather)
{greetings = “This is a beautiful day!”;}
else
{greetings = “Oops! Today is going to be messy.”;};

// The equivalent statement using the ?: operator is as follows:
greetings = (goodWeather) ? “This is a beautiful day!” :  “Oops! Today is going to be messy.”;
```

3.3.3 The Switch-Statement

The **Switch-Statement** implements the **Case-Structure** of chapter 1 (section 1.7.4). It has the following syntax:

Figure 3.7: BNF Syntax for the Java Switch-Statement

```
Switch-Statement::=
switch (<Expression>)
{
  case <ConstantExpression1>:      {<Statement>;  [*<Statement>;* ] break;}
  …
  case <ConstantExpressionN>:      {<Statement>;  [*<Statement>;*] break;}
 [default:                         <Statement>; | <CompoundStatement>]
}

CompoundStatement ::=
{
  <Statement>;
  [*<Statement>;*]
}
```

Please note the following:

1. The **Switch-Statement** is the Java implementation of the **Case-Structure**, but is not as flexible (for example, the languages Pascal and RPG-400 both have more flexible implementations). It is used when an expression could have any of a finite set of values and the action required varies with each value.
2. The order in which the constant expressions are listed is unimportant.
3. Each constant expression may lead to one or several statements.
4. To avoid a fall-through case, it is good practice to include a **Break-Statement** after (to terminate) each case, as indicated.
5. The switch expression must evaluate to a **char**, **byte**, **short** or **int** value.
6. Nesting is facilitated — the statements inside a case option may be any valid statement, including another **Switch-Statement**.

Example 6: Switch statements are great for menus, as in the following example.

```
int Option;
// …
switch (Option)     {
case 1:        {LineDraw (); break;};
case 2:        {RectangleDraw (); break;};
case 3:        {SquareDraw (); break;};
case 4:        {TriangleDraw (); break;};
default:       System.out.println( “Invalid Option”);
}
// …
// Each selected option results in the invocation of a method. Methods will be discussed later in the course.
```

3.4 Iteration Structures in Java

Recall that in the discussion on algorithm development, three iteration structures were discussed in section 1.7.5, namely the **While-Structure**, the **Until-Structure** and the **For-Structure**. Java implements all three structures in the form of the **While-Statement**, the **Do-While-Statement** and the **For-Statement** respectively. We will examine each statement.

3.4.1 The While-Statement

The **While-Statement** is used for constructing a **while loop**. The syntax for usage appears in figure 3.8:

Figure 3.8: Syntax for the While-Statement

```
WhileStatement ::=
while (<BooleanExpression>)  <Statement>; | <CompoundStatement>

CompoundStatement ::=
{
  <Statement>;
  [*<Statement>;*]
}
```

From this definition, please observe the following:

1. The Boolean expression specified is a condition that evaluates to true or false.
2. The statement(s) specified could be any valid statement, including other iteration statement(s), thus resulting in nested iteration loops.
3. Where it is requested to have more than one statement within the while-loop, a compound statement is used. Remember that the right curly brace (}) takes the place of the semicolon for block termination.
4. If nested loops are used, indent to improve the readability of your program.
5. Your loop must have an exit strategy; otherwise it is an *infinite loop*. By exit strategy, we mean that a statement within the loop must change the condition that causes looping.

3.4.1 The While-Statement (continued)

Example 7: Figure 3.9 provides several examples of simple Java while-loops.

Figure 3.9: Simple Java While-Loops

Figure 3.9a: A Loop Controlled by a Boolean Flag:

```
boolean exitTime;
// …
exitTime = false;
while (!exitTime)
{
   /* Several statements one of which must change exitTime to true */
}
```

Figure 3.9b: A Counter Monitoring Loop:

```
int count, limit;
// …
count = 0; limit = 100;
while (count <= limit)
{
    //… One or more statements
    count++; // Increments count by 1
}
```

Figure 3.9c: A Time Delay Loop:

```
int count, limit;
// . . .
// A time delay loop
count = 0; limit = 60;
while (count <= limit) count ++;
```

OR

```
int count, limit;
// …
// An equivalent time delay loop
count = 0; limit = 60;
while (Count++ <= Limit);
```

3.4.2 The For-Statement

The **For-Statement** is the most flexible (and widely used) iterative statement in Java. Figure 3.10 shows its syntax:

Figure 3.10: Syntax for the For-Statement

```
ForStatement ::=
for (<Expression1>; <Condition1>; <Expression2>)  <Statement>; | <CompoundStatement>

CompoundStatement ::=
{
  <Statement>;
  [*<Statement>;*]
}
```

Based on this statement, please note:

1. **Expression1** is the initialization expression (typically an assignment); **Contition1** is a Boolean expression that determines the whether the loop iterates; **Expression2** is the increment expression (typically an assignment).
2. The **For-Statement** and the **While-Statement** are interchangeable. The above format may be representing using the following **while loop**:

```
<Expression1>;
while <Condition1>
{
   <Statement>;
  [*<Statement>;*]
   <Expression2>;
}
```

3. Three (sets of) arguments are specified within the parentheses: initialization expression(s), exit condition(s) and increment expression(s). For each category, more than one expression may be specified (separated via use of the comma). Expressions at each category are parallel, are evaluated left to right and should be of the same type.
4. The statements enclosed within the **for loop** could be any valid statement(s), including other iterative statements, thus resulting in nested loops.

Example 8: The following code is equivalent to the code in figure 3.9b:

```
int count, limit;
// …
for (count = 0;  count <= Limit; count++)
{
  //… One or more statements
}
```

3.4.2 The For-Statement (continued)

Example 9: The following code is equivalent to the code in figure 3.9c:

```
int count, limit;
// …
for (count = 0; count  <=limit; count++);
```

Example 10: The following is the code for a Java method that reverses a string. As an exercise, you are encouraged to test it by writing a Java program that invokes it.

```
public static String Reverse(String thisString)
{
    int y, z;
    char x;
    String revS = " ";
    for (z = thisString.length()-1; z >= 0; z--)
    { revS += thisString.charAt(z); }
    return revS.trim();
} // End of Reverse Method
```

Example 11: The following code will result in an infinite loop.

```
// An infinite for-loop
for (; ;);
```

3.4.3 The Do-While-Statement

In contrast to the **While-Statement** and the **For-Statement**, the **Do-While-Statement** sets up a loop where the condition is tested at the end of the loop. This is the Java implementation of the **Repeat-Until-Structure** of chapter 1 (section 1.7.5). The syntax for the usage is provided in figure 3.11.

Figure 3.11: Syntax for the Do-While-Statement

```
DoWhileStatement ::=
do <Statement>; |  <CompoundStatement>
while (<Condition>);

CompoundStatement ::=
{
  <Statement>;
  [*<Statement>;*]
}
```

3.4.3 The Do-While-Statement (continued)

Please observe:

1. The condition specified must be a Boolean expression; it determines whether the iteration will continue. When the condition becomes true, iteration stops.
2. The statement(s) specified within the loop may be any valid statement, including other iteration statements, thus resulting in nested loops.
3. Where it is required to have more than one statement within the loop, a compound statement must be specified.
4. If nested loops are used, remember to indent to improve readability.
5. As for all iterative loops, there must be an exit strategy, to avoid having an indefinite loop.

Example 12: The following code is equivalent to the code in figure 3.9b:

```
int count, limit;
// …
count = 0;
do
{
  // … One or more statements
  count++;
} while (Count <= Limit);
```

Example 13: The following code is equivalent to the code in figure 3.9c:

```
int count, limit;
// …
count = 0;
do count++;
while (count <=limit);
```

3.4.4 Labels, Break-Statement and Continue-Statement

Like several other languages (including C++ and Pascal), Java allows you to set up statement labels. To define a label, the syntax is simply:

```
Label ::=  <LabelName>:
```

Two of the simplest statements in Java are the **Break-Statement** and the **Continue-Statement**. The required syntax in either case is simply the keyword, and an optional label-name, followed by a semicolon, as depicted in figure 3.12.

Figure 3.12: The Break-Statement and Continue-Statement

```
BreakStatement ::=
break [<LabelName>];

ContinueStatement ::=
continue [<LabelName>];
```

The **Break-Statement** causes an immediate exit from a loop (and takes the place of the **Exit** verb of section 1.7.5). If a break-point label is specified, exit is made to that break-point. As mentioned earlier, break-statements often appear within a **Switch-Statement**. The **Continue-Statement** (not applicable to the **Switch-Statement**) causes the next iteration of the loop to begin (and takes the place of the **Iterate** verb of section 1.7.5). If a continuation-point label is specified, control is sent to the continuation-point. These two statements are often used within iterative loops.

Example 14: In the following code, the **break** statement forces a complete exit from the loop, while the **continue** statement re-loops to the **ThisPoint** label.

```
// Assume that moreRequired and noMore are Boolean variables
// . . .
Again:
while (moreRequired)
{
  //  …
  ThisPoint:
  //  …
  if (noMore) break Again;           // exits the loop completely
  if (partial) continue ThisPoint;   // loops back to ThisPoint
}
```

It must be noted that the use of labels in this way is completely optional. In fact, you are encouraged to avoid such constructs during the formative learning stage of your programming, and stick to more traditional iterative loops.

3.5 Static Methods and the Driver Class

As mentioned in earlier discussions, Java is an object-oriented programming language (OOPL). Java programs are made up of applications, which are comprised of classes; classes are made up of data items and methods; and methods are comprised of data items and statements.

Also, it is imperative that you remember at all times that a class is an implementation of an object type, and Java methods are implementations of subroutines/subprograms (review section 1.7 and chapter 2).

Since Java methods are the building blocks for Java classes, you cannot get very far in Java programming without working with these program components — classes, data items, method, and statements. We have had introductory discussions about these components, and have indeed been using them. However, at this point, it becomes necessary to provide some more clarifications on Java methods.

You are accustomed to using methods of certain classes that are shipped with Java (for example, **System.out.print**(…); **JOptionPane.showMessageDialog**(…), **Double.parseDouble**(…); etc.). These are all examples of static methods. By *static*, we mean that an instance of the parent class does not need to be created in order to use the method. Not only can you make use of static Java methods, you may also create your own static methods within your *driver class*. By *driver class*, we mean the program class with your Java **main (…)** method. It is called your driver class because it controls your Java program. Every Java program contains at least a driver class. However as you will learn later in the course, a program may also contain other classes working together. For now, let's focus on static methods in the driver class.

To further refresh your memory, a method has the same definition as a subroutine; it is a component of a program that carries out a set of related activities. Figure 3.13 employs the BNF notation to show the required syntax for specifying a method in Java. Essentially, a method consists of a *method heading* and a *method body*. The method heading has the *method signature*, which consists of the method qualifier(s), return type, method name, and any parameter(s) used; the body consists of zero or more statements, and is enclosed between a left curly brace({) and a right curly brace (}).

Figure 3.13: BNF Notation Showing Syntax for Java Method

```
Method ::=
<Qualifier(s)><ReturnType><MethodName>([<Parameter(s)>])
{
  // Body of the Method
  [*<Statement>;*]
  [<ReturnStatement>;]
}

Qualifier ::=
[final] public | private | protected [static] [abstract]

ReturnStatement ::=
return <Variable>; | <Expression>;
```

3.5 Static Methods and Driver Class (continued)

From this definition, please note the following:

- The qualifiers were introduced in section 2.5.2 of the previous chapter; moreover, they will be revisited from time to time (whenever necessary) as we proceed through the course.
- The qualifiers typically used for static methods in driver classes are **public** and **static**. The keyword **public** means that the method is accessible from anywhere within the Java program; and **static** means that it will not be necessary to create an instance of the class in order to use the method.
- The return type is the data-type for any data item that the method will return. Any valid Java data type (primitive or *advanced data type* to which you will be introduced later in the course) may be specified. Alternately, if the method will not return any data item, then the keyword **void** must be specified for the return type.
- If the return type is not **void**, then the method must include at least one **return-statement**, which returns a value belonging to the stated return type.
- The method must have a valid identifier as its name; please review 2.5.2 on identifiers.
- The method may have parameter(s) (recall that this concept was discussed in section 1.7.2 of chapter 1). A Java parameter is like a variable declaration, but without the terminating semicolon. If the method contains multiple parameters, they are separated by use of the comma. Parameters are place holders for arguments that must be supplied when the method is called. On the call (invocation) of the method, each argument is copied to its corresponding parameter (first argument to first parameter, second argument to second parameter, and so on). Naturally and importantly, the argument and its corresponding parameter must be defined on the same data type. This feature is often referred to as *passing parameters by value*.

Once you have defined a static method in your Java program (specifically, your driver class), you may invoke it from anywhere in the program. If the method returns a value, it is typically called by including it in an expression (that will use the value returned), or in an **assignment-statement** that retrieves the returned value into a specified variable. If the method does not return a value, then it may be invoked by simply specifying its name. Irrespective of how the method is called, if it has parameters, be sure to supply corresponding arguments on the call statement. The upcoming section discusses an example that should clarify these principles.

Now let us revisit and refine a concept that was introduced to you in chapter 1, namely algorithm development (review section 1.7): When you construct your program plan, you should start with a UML diagram of each class comprising the program (for now, just one, but soon you will be writing programs with multiple classes). Your UML class diagram identifies the class-name, principal data items, and the *method signatures* (i.e. method headings) for each method comprising the class. This should be followed with an algorithm for each method specified in the UML class diagram.

3.6 A Programming Example

To reinforce what we have covered in this chapter, let us develop a program that will allow the user to enter a string. The program will then reverse that string and return the result to the user. The program will make use of the method illustrated in Example 10. Figure 3.14 provides an algorithm for the problem. The first portion the figure shows the UML class diagram. Figure 3.15 shows the Java program. Notice that the Java code corresponds nicely to the stated UML class diagram in figure 3.14.

Figure 3.14: Algorithm for String Reversal Program

StringReversal // The class-name
// The principal data items for this program String inputString, reversedString
// Headings of the principal methods public static void **main**(String[] args) public static String **reverseS(**String thisString**)**

The main Routine:

```
START
  Let more be Boolean;
  Let exitKey be a character;
  more := true;
  While (more) do the following:
    Prompt for and accept inputString;
    reversedString := reverseS(inputString);
    Display ("This is your original string: " + inputString);
    Display ("This is the reversed string: " + reversedString);

    // Find out if user wants to continue and take appropriate action
    Prompt for and accept exitKey;
    If (exitKey = 'X' or 'x') more := false; End-If;
  End-While-More;
STOP
*********************************************************************************
```

Subroutine: reverseS(thisString) Returns a string

```
// Assume thisString is a string;
START
  // Switch the first character with the last, the second with the second-to-last, and so
  Let revString be a string, initialized to null;
  Let z and lim be integer  ;
  Set lim to the length of thisString;
  For (z := lim to 1 with increments of -1) do the following:
        Append the character at position z in thisString to revString;
  End-For;
  Return revString;
STOP
```

Figure 3.15: Java Code for the String Reversal Program

```
/* StringReversal.java :     Accepts a string from the user and reverses it. This continues until user quits.*/
/* Author Elvis Foster                                                                                     */
// *************************************************************************************
package Application3;
import javax.swing.JOptionPane; // This object facilitates dialog boxes, etc.

public class StringReversal
{
  // Global Declarations
  static String inputString, reversedString, outputString;
  final static String HEADING = "String Reversal Program";

 // No constructor necessary
 // public StringReversal() { }

 // Main Method
    public static void main(String[] args)
    {
     // Declare Variables
     char exitKey = ' ';
     String continueString;
     boolean more = true;

     while (more) // While user wishes to continue
     {
        // Accept the string, reverse it, and then display the result
        inputString = JOptionPane.showInputDialog(null, "Input String: ", HEADING,      +
              JOptionPane.QUESTION_MESSAGE);
        reversedString = reverseS(inputString);

        outputString =  "Input String:    " + inputString + "\n" +
         "Reversed String: " + reversedString + "\n";
        JOptionPane.showMessageDialog(null, outputString, HEADING,JOptionPane.INFORMATION_MESSAGE);

        // Find out whether user wants to continue
        continueString = JOptionPane.showInputDialog(null, "Press X to exit, or any other key: ", HEADING,  +
           JOptionPane.QUESTION_MESSAGE);
        exitKey = continueString.charAt(0);
        if (exitKey == 'X' || exitKey == 'x') more = false;
     } // End-while user wishes to continue

    } // End main

    // String Reversal Method
    public static String reverseS(String thisString)
    {
     int y, z;
     char x;
     String revString = " ";
     for (z = thisString.length()-1; z >= 0; z--)
     { revString += thisString.charAt(z); }
     return revString.trim();
    } // End of Reverse Method
} // End of class StringReversal
```

3.7 Summary and Concluding Remarks

By now you should be feeling really excited about the power of your newly discovered skill of programming. Take some time to review the main points covered in this chapter. The summary following summary and review questions should help:

Boolean expressions are conditions that evaluate to true or false. They are constructed by combining variables/literals with other expressions and Boolean operators.

A compound statement is a block of statements, enclosed within curly braces ({ … }).

The Java **If-Statement** implements the **If-Structure** in programming. The statement tests for a condition, and caries out the statements specified if the condition is true. Optionally, the statement may have a corresponding **Else-Statement**, in the event that the tested condition is false. In such circumstance, the statements in the **else-clause** are executed. The statement following the **if-condition** may be a simple statement or a compound statement.

The Java **Switch-Statement** implements the **Case-Structure** in programming. The statement works with expressions that evaluate to a **char, byte, short *or* int** value. On each specific value tested for, a single action may be specified via a simple statement, or a combination of actions may be specified via a compound statement. **Switch-Statements** are perfect for manipulating menus.

The Java **While-Statement** implements the **While-Structure** of programming. A condition is specified at the beginning of the **while-loop**. Iteration occurs as long as that condition evaluates to true, and stops when the condition becomes false. The **while-condition** may be followed by a simple statement or a compound statement. To avoid an infinite loop, at least one statement within the **while-loop** must force the tested looping condition to change from true to false.

The Java **For-Statement** implements the **For-Structure** of programming. An initialization statement sets a set of values to stated variables. The **for-condition** performs a test to ensure that looping continues. The increment statement of the **for-loop** ensures that variable(s) initialized in the initialization statement will gradually advance toward the condition tested. Iteration occurs as long as that condition evaluates to true, and stops when the condition becomes false. The **for-loop** may contain a simple statement or a compound statement.

The Java **Do-While-Statement** implements the **Repeat-Until-Structure** of programming. A condition is specified at the end of the loop. Iteration continues until that condition evaluates to true. The **do-while-loop** may contain a simple statement or a compound statement. To avoid an infinite loop, at least one statement within the **do-while-loop** must force the tested condition to change from false to true.

The Java **Break-Statement** and **Continue-Statement** are used to prematurely exit an iteration loop.

A Java method is a section of a Java class that carries out a specific set of related activities. The method consists of a qualifier, return type, method-name, and any parameters it may have.

Practice writing out the syntax for the statements covered in this chapter, as well as examples using each statement. The next chapter builds on the information covered here, while introducing some additional features of Java programming.

3.8 Review Questions

1. When would you use a **Java If-Statement** and when would you use a **Switch-Statement**?

2. Explain the difference between a Java **While-Statement** and a **For-Statement**. Describe a scenario that most appropriately fits each statement.

3. Explain the difference between a Java **While-Statement** and a **Do-While-Statement**. Describe a scenario that most appropriately fits each statement.

4. Telephone numbers in several Western countries are of the form **999-999-9999**, representing the area code and a seven-digit number. Develop the algorithm for a program that will prompt the user for telephone number, accepted as a 12-byte string, and then perform data validation to ensure that the string received represents a valid telephone number. On each entry, the program will examine the entry and send a message to the user to inform whether or not the entry is a valid telephone number. This should happen until the user quits. Write a Java program to implement your algorithm.

5. A prime number is a positive integer that is perfectly divisible (without a remainder) only by itself and 1. Develop an algorithm that prompts the user for any positive inter, and performs a test to determine whether it is a prime number or not. On each entry, the algorithm must inform the user of the result of the evaluation. This should continue until the user quits. Write a Java program to implement your algorithm.

3.9 Recommended Reading

[Bell & Parr 2010] Bell, Douglas and Mike Parr. 2010. *Java for Students* 6th Ed. New York: Pearson. See chapters 7 & 8.

[Liang 2014] Liang, Y. Daniel. 2014. *Introduction to Java Programming — Comprehensive Version*, 10h ed. Boston, MA: Pearson Education. See chapter 5.

[Savitch & Carrano 2008] Savitch, Walter and Frank M. Carrano. 2008. *Java: An Introduction to Problem Solving & Programming* 5th ed. Upper Saddle River, NJ: Prentice Hall. See chapters 3 & 4.

[Oracle 2015]. Oracle Corporation. 2015. “Java™ Platform, Standard Edition 8API Specification.” Accessed January 19, 2015. http://docs.oracle.com/javase/8/docs/api/