Chapter 19

Other Conditional and Iterative Statements

The if/else and while statements are flexible enough to implement the logic of any algorithm we might wish to implement, but Java provides some additional conditional and iterative statements that are more convenient to use in some circumstances. These additional statements include

- **switch**: an alternative to some multi-way if/else statements

- **conditional operator**: an expression that exhibits the behavior of an if/else statement

- **do/while**: a loop that checks its condition after its body is executed

- **for**: a loop convenient for counting

### 19.1 The switch Statement

The switch statement provides a convenient alternative for some multi-way if/else statements like the one in RestyledDigitToWord (§6.7). The general form of a switch is:
switch (integral expression) {
    case integral constant 1:
        statement(s)
        break;
    case integral constant 2:
        statement(s)
        break;
    case integral constant 3:
        statement(s)
        break;
    .
    .
    .
    case integral constant n:
        statement(s)
        break;
    default:
        statement(s)
        break;
}

- The reserved word switch identifies a switch statement.
- The expression, contained in parentheses, must evaluate to an integral value. Any integer type, characters, and Boolean expressions are acceptable. Floating point expressions are forbidden.
- The body of the switch is enclosed by curly braces, which are required.
- Each case label is followed by an integral constant. This constant can be either a literal value or a final symbolic value. In particular, non-final variables and other expressions are expressly forbidden. If the case label matches the switch's expression, then the statements that follow that label are executed. The statements and break statement that follow each case label are optional. One way to execute one set of statements for more than one case label is to provide empty statements for one or more of the labels, as in:

```java
switch (inKey) {
    case 'p':
    case 'P':
        System.out.println("Executing print");
        break;
    case 'q':
    case 'Q':
        done = true;
```
Here either an upper- or lowercase \( P \) result in the same action; either an upper- or lowercase \( Q \) sets the done Boolean variable. The \texttt{break} statement is optional. When a \texttt{case} label is matched, the statements that follow are executed until a \texttt{break} statement is encountered. The control flow then transfers out of the body of the \texttt{switch}. (In this way, the \texttt{break} within a \texttt{switch} works just like a \texttt{break} within a loop: the rest of the body of the statement is skipped and program execution resumes at the next statement following the body.) A missing \texttt{break} (a common error, when its omission is not intentional) causes the statements of the succeeding \texttt{case} label to be executed. The process continues until a \texttt{break} is encountered or the end of the body is reached.

- The \texttt{default} label is matched if none of the \texttt{case} labels match. It serves as a “catch all” like the final \texttt{else} in a multi-way \texttt{if/else} statement. The \texttt{default} label is optional. If it is missing and none of the \texttt{case} labels match the expression, then no statements in the \texttt{switch}’s body are executed.

\texttt{SwitchDigitToWord} \texttt{(19.1)} shows what \texttt{RestyledDigitToWord} \texttt{(6.7)} would look like with a \texttt{switch} statement instead of the multi-way \texttt{if/else} statement.

```java
import java.util.Scanner;

public class SwitchDigitToWord {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int value;
        System.out.println("Please enter an integer in the range 0...5: ");
        value = scanner.nextInt();
        if (value < 0) {
            System.out.println("Too small");
        } else {
            switch (value) {
                case 0:
                    System.out.println("zero");
                    break;
                case 1:
                    System.out.println("one");
                    break;
                case 2:
                    System.out.println("two");
                    break;
                case 3:
                    System.out.println("three");
                    break;
                case 4:
                    System.out.println("four");
                    break;
                case 5:
                    System.out.println("five");
                    break;
                default:
```
The `switch` statement has two restrictions that make it less general than the multi-way `if/else`:

1. The `switch` argument must be an integral expression.
2. Case labels must be constant integral values. Integral literals and constants are acceptable. Variables or expressions are not allowed.

To illustrate these restrictions, consider the following `if/else` statement that translates easily to an equivalent `switch` statement:

```java
if ( x == 1 ) {
    // Do 1 stuff here . . .
} else if ( x == 2 ) {
    // Do 2 stuff here . . .
} else if ( x == 3 ) {
    // Do 3 stuff here . . .
}
```

The corresponding `switch` statement is:

```java
switch ( x ) {
    case 1:
        // Do 1 stuff here . . .
        break;
    case 2:
        // Do 2 stuff here . . .
        break;
    case 3:
        // Do 3 stuff here . . .
        break;
}
```

Now consider the following `if/else`:

```java
if ( x == y ) {
    // Do "y" stuff here . . .
} else if ( x > 2 ) {
    // Do "> 2" stuff here . . .
} else if ( x == 3 ) {
    // Do 3 stuff here . . .
}
```
This code cannot be easily translated into a switch statement. The variable y cannot be used as a case label. The second choice checks for an inequality instead of an exact match, so direct translation to a case label is impossible.

As a consequence of the switch statement’s restrictions, the compiler produces more efficient code for a switch than for an equivalent if/else. If a choice must be made from one of several or more options, and the switch statement can be used, then the switch statement will likely be faster than the corresponding multi-way if/else.

19.2 The Conditional Operator

As purely a syntactical convenience, Java provides an alternative to the if/else construct called the conditional operator. It has limited application but is convenient nonetheless. The following section of code assigns either the result of a division or a default value acceptable to the application if a division by zero would result:

```java
// Assign a value to x:
if ( z != 0 ) {
    x = y/z;
} else {
    x = 0;
}
```

This code has two assignment statements, but only one is executed at any given time. The conditional operator makes for a simpler statement:

```java
// Assign a value to x:
x = ( z != 0 ) ? y/z : 0;
```

The general form of a conditional expression is:

```
condition ? expression1 : expression2
```

- condition is a normal Boolean expression that might appear in an if statement. Parentheses around the condition are not required but should be used to improve the readability.
- expression1 the overall value of the conditional expression if the condition is true.
- expression2 the overall value of the conditional expression if the condition is false.

The conditional operator uses two symbols (?) and :) and three operands. Since it has three operands it is classified as a ternary operator (Java’s only one). Both expression1 and expression2 must be assignment compatible; for example, it would be illegal for one expression to be an int and the other to be boolean. The overall type of a conditional expression is the type of the more dominant of expression1 and expression2. The conditional expression can be used anywhere an expression can be used. It is not a statement itself; it is used within a statement.

As another example, the absolute value of a number is defined in mathematics by the following formula:

\[
|n| = \begin{cases} 
n, & \text{when } n \geq 0 \\
-n, & \text{when } n < 0 
\end{cases}
\]

In other words, the absolute value of a positive number or zero is the same as that number; the absolute value of a negative number is the additive inverse (negative of) of that number. The following Java expression represents the absolute value of the variable n:
Some argue that the conditional operator is cryptic, and thus its use reduces a program’s readability. To seasoned Java programmers it is quite understandable, but it is actually used sparingly because of its very specific nature.

19.3 The do/while Statement

The while statement (Section 17.1) checks its condition before its body is executed; thus, it is a top-checking loop. Its body is not executed if its condition is initially false. At times, this structure is inconvenient. Consider GoodInputOnly (19.2).

```java
import javax.swing.JOptionPane;

public class GoodInputOnly {
    public static void main() {
        int inValue = -1;
        while (inValue < 0 || inValue > 10) {
            // Insist on values in the range 0...10
            inValue = Integer.parseInt(
                JOptionPane.showInputDialog(
                    "Enter integer in range 0...10: ");
            )
        }
        // InValue at this point is guaranteed to be within range
        JOptionPane.showMessageDialog(null, "Legal value entered was " + inValue);
    }
}
```

Listing 19.2: GoodInputOnly—Insist the user enter a good value

The loop in GoodInputOnly traps the user until he provides a good value. Here’s how it works:

- The initialization of `inValue` to `-1` ensures the condition of the `while` will be true, and, thus, the body of the loop will be entered.
- The condition of the `while` specifies a set that includes all values that are not in the desired range. `inValue` is initially in this set, so the loop is entered.
- The user does not get a chance to enter a value until program execution is inside the loop.
- The only way the loop can be exited is if the user enters a value that violates the condition—precisely a value in the desired range.

The initialization before the loop check is somewhat artificial. It is there only to ensure entry into the loop. It seems unnatural to check for a valid value before the user gets a chance to enter it. A loop that checks its condition after its body is executed at least once would be more appropriate. The `do/while` is bottom-checking loop that behaves exactly in this manner. Its flowchart is shown in Figure 19.1.

The `do/while` statement has the general form:
19.3. THE **DO/WHILE** STATEMENT

![Flowchart of a do/while statement](image)

Figure 19.1: Execution flow in a do/while statement

```
do
    body
while (condition);
```

- The reserved words `do` and `while` identify a `do/while` statement. The `do` and `while` keywords delimit the loop's body, but curly braces are still required if the body consists of more than one statement.

- The condition is associated with the `while` at the end of the loop. The condition must be enclosed within parentheses.

- The body is like the body of the `while` loop.

`BetterInputOnly (19.3)` uses a `do/while` to erase the criticisms of `GoodInputOnly (19.2)`.

```java
import javax.swing.JOptionPane;

public class BetterInputOnly {
    public static void main() {
        int inValue;
        do {
            // Insist on values in the range 0...10
            inValue = Integer.parseInt(JOptionPane.showInputDialog("Enter integer in range 0...10: "));
        } while (inValue < 0 || inValue > 10);
        // inValue at this point is guaranteed to be within range
        JOptionPane.showMessageDialog(null, "Legal value entered was " + inValue);
    }
}
```

13 March 2008 Draft © 2008 Richard L. Halterman
The body of a do/while statement, unlike the while statement, is guaranteed to execute at least once. This behavior is convenient at times as BetterInputOnly shows.

We can use BetterInputOnly as a starting point for a general-purpose reusable class, IntRange (19.4).

```
import javax.swing.JOptionPane;

// Restricts the user to entering a restricted range of integer
// values
public class IntRange {
    public static int get(int low, int high) {
        int inValue;
        do {
            // Insist on values in the range low...high
            inValue = Integer.parseInt(JOptionPane.showInputDialog
                ("Enter integer in range " + low + "..." + high));
        } while (inValue < low || inValue > high);
        // inValue at this point is guaranteed to be within range
        return inValue;
    }
}
```

The break and continue statements can be used in the body of a do/while statement. Like with the while statement, break causes immediate loop termination (any remaining statements within the body are skipped), and continue causes the remainder of the body to be skipped and the condition is immediately checked to see if the loop should continue or be terminated.

## 19.4 The for Statement

Recall IterativeCountToFive (17.2) from Section 17.1, reproduced here.

It simply counts from one to five. Counting is a frequent activity performed by computer programs. Certain program elements are required in order for any program to count:

- A variable must be used to keep track of the count; count is the aptly named counter variable.
- The counter variable must be given an initial value, 1.
- The variable must be modified (usually incremented) as the program counts. The statement
  \[
  \text{count} = \text{count} + 1;
  \]
increments count.

• A way must be provided to determine if the count has completed. The condition of the while controls the extent of the count.

Java provides a specialized loop that packages these four programming elements into one convenient statement. Called the for statement, its general form is

\[
\text{for ( initialization ; condition ; modification )} \\
\text{body}
\]

• The reserved word for identifies a for statement.

• The header, contained in parentheses, contains three parts, each separated by semicolons:

  1. **Initialization.** The initialization part assigns an initial value to the loop variable. The loop variable may be declared here as well; if it is declared here, then its scope is limited to the for statement. The initialization part is performed one time.

  2. **Condition.** The condition part is a Boolean expression, just like the condition of while and do/while. The condition is checked each time before the body is executed.

  3. **Modification.** The modification part changes the loop variable. The change should be such that the condition will eventually become false so the loop will terminate. The modification is performed during each iteration after the body is executed.

• The body is like the body of any other loop.

With a while loop, these four counting components (variable declaration, initialization, condition, and modification) can be scattered throughout the method. With a for loop, the programmer can determine all the important information about the loop’s control by looking at one statement. Figure 19.2 shows the control flow within a for statement.

ForCounter (19.5) uses a for loop to do the work of IterativeCountToFive (17.2).

```java
public class ForCounter {
    public static void main(String[] args) {
        for (int count = 1; count <= 5; count++) {
            System.out.println(count);
        }
    }
}
```

Listing 19.5: ForCounter—IterativeCountToFive (17.2) using a for statement in place of the while

TimesTable (17.5) that prints a multiplication table is better written with nested for statements as BetterTimesTable (19.6) shows.
19.4. THE FOR STATEMENT

Figure 19.2: Execution flow in a for statement

```java
public class BetterTimesTable {
    public static void main(String[] args) {
        // Print a multiplication table to 10 x 10
        // Print column heading
        System.out.println("1 2 3 4 5 6 7 8 9 10");
        System.out.println("+----------------------------------------");
        for (int row = 1; row <= 10; row++) {
            // Table has ten rows.
            System.out.printf("%3d", row);
            // Print heading for this row.
            for (int column = 1; column <= 10; column++) {
                System.out.printf("%4d", row*column);
            }
            System.out.println(); // Move cursor to next row
        }
    }
}
```

Listing 19.6: BetterTimesTable—Prints a multiplication table using for statements

13 March 2008 Draft © 2008 Richard L. Halterman
A for loop is ideal for stepping through the rows and columns. The information about the control of both loops is now packaged in the respective for statements instead of being spread out in various places in main(). In the while version, it is easy for the programmer to forget to update one or both of the counter variables (row and/or column). The for makes it harder for the programmer to forget the loop variable update, since it is done right up front in the for statement header.

It is considered bad programming practice to do either of the following in a for statement:

- Modify the loop control variable within the body of the loop. If the loop variable is modified within the body, then the logic of the loop’s control is no longer completely isolated to the for statement’s header. The programmer must look elsewhere within the statement to understand completely how the loop works.

- Prematurely exit the loop with a break. This action also violates the concept of keeping all the loop control logic in one place (the for’s header).

The language allows both of these practices, but experience shows that it is best to avoid them. If it seems necessary to violate this advice, consider using a different kind of loop (while or do/while) that does not imply the same degree of control regularity implied by the for loop.

ForPrintPrimes (19.7) is a rewrite of PrintPrimes (18.2) that replaces its while loops with for loops.

```java
import java.util.Scanner;

public class ForPrintPrimes {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        int maxValue;
        System.out.print("Display primes up to what value? ");
        maxValue = scan.nextInt();
        for (int value = 2; value <= maxValue; value++) {
            // See if value is prime
            int trialFactor = 2;
            boolean isPrime = true; // Assume no factors unless we find one
            for (trialFactor = 2; isPrime && trialFactor < value - 1;
                 trialFactor++) {
                if (value % trialFactor == 0) { // Is trialFactor a factor?
                    isPrime = false;
                    break; // No need to continue, we found a factor
                }
            }
            if (isPrime) {
                System.out.print(value + " "); // Display the prime number
            }
        }
        System.out.println(); // Move cursor down to next line
    }
}
```

Listing 19.7: ForPrintPrimes—PrintPrimes using for loops
The conditional in the for loop can be any legal Boolean expression. The logical and (\&\&), or (\|\|), and not (!) operators can be used to create complex Boolean expressions, if necessary. The modification part of the for loop is not limited to simple arithmetic and can be quite elaborate. For example:

```java
for ( double d = 1000; d >= 1; d = Math.sqrt(d) ) {
    /* Body goes here */
}
```

Here \(d\) is reassigned by a method call. The following loop is controlled entirely by user input (\(\text{scan}\) is a \(\text{Scanner}\) object):

```java
for ( int i = scan.nextInt(); i != 999; i = scan.nextInt() ) {
    /* Body goes here */
}
```

While the for statement supports such complex headers, simpler is usually better. Ordinarily the for loop should manage just one control variable, and the initialization, condition, and modification parts should be straightforward. If a particular programming situation warrants a particularly complicated for construction, consider using another kind of loop.

Any or all of the parts of the for statement (initialization, condition, modification, and body) may be omitted:

- **Initialization.** If the initialization is missing, as in

  ```java
  for (; i < 10; i++ ) {
      /* Body goes here */
  }
  ```
  then no initialization is performed by the for loop, and it must be done elsewhere.

- **Condition.** If the condition is missing, as in

  ```java
  for ( int i = 0; ; i++ ) {
      /* Body goes here */
  }
  ```
  then the condition is true by default. A break must appear in the body unless an infinite loop is intended.

- **Modification.** If the modification is missing, as in

  ```java
  for ( int i = 0; i < 10; ) {
      /* Body goes here */
  }
  ```
  then the for performs no automatic modification; the modification must be done by a statement in the body to avoid an infinite loop.

- **Body.** If the body is missing, as in

  ```java
  for ( int i = 0; i < 10; i++ ) {}%0A
  ```

or
for ( int i = 0; i < 10; i++ );

then an empty loop results. This can be used for a nonportable delay (slower computers will delay longer than faster computers), but some compilers may detect that such code has no functional effect and “optimize” away such an empty loop.

While the for statement supports the omission of parts of its header, such constructs should be avoided. The for loop’s strength lies in the ability for the programmer to see all the aspects of the loop’s control in one place. If some of these control responsibilities are to be handled elsewhere (not in the for’s header) then consider using another kind of loop.

Programmers usually select a simple name for the control variable of a for statement. Recall that variable names should be well chosen to reflect the meaning of their use within the program. It may come as a surprise that \( i \) is probably the most common name used for an integer control variable. This has its roots in mathematics where variables such as \( i, j, \) and \( k \) are commonly used to index vectors and matrices. Computer programmers make considerable use of for loops in array processing, so programmers have universally adopted this convention of short control variable names. Thus, it generally is acceptable to use simple identifiers like \( i \) as loop control variables.

The break and continue statements can be used in the body of a for statement. Like with the while and do/while statements, break causes immediate loop termination, and continue causes the condition to be immediately checked to determine if the iteration should continue. As previously mentioned, for loop control should be restricted to its header, and the use of break and continue should be avoided.

Any for loop can be rewritten with a while loop and behave identically. For example, consider the for loop

```java
for ( int i = 1; i <= 10; i++ ) {
    System.out.println(i);
}
```

and next consider the while loop that behaves exactly the same way:

```java
int i = 1;
while ( i <= 10 ) {
    System.out.println(i);
    i++;
}
```

Which is better? The for loop conveniently packages the loop control information in its header, but in the while loop this information is distributed throughout the small section of code. The for loop thus provides a better organization of the loop control code. Does one loop outperform the other? No. These two sections of code are compiled into exactly the same bytecode:

```
0 iconst_1   // ---+
1 istore_0   // -------> i = 1
2 goto 15    // --------> go to line 15
5 getstatic #2 <Field java.io.PrintStream out> // --+
8 iload_0    // ------------------------+
9 invokevirtual #3 <Method void println(int)> // ---+--> print i
12 iinc 0 1   // ---+ i++
15 iload_0    // ----
16 bipush 10   // ----
18 if_icmple 5 // -------+ if i <= 10 go to line 5
```

Thus, the for loop is preferred in this example.
19.5 Summary

• Add summary items here.

19.6 Exercises