Chapter V

Keyboard Input & Control Structures

Chapter V Topics

5.1 Keyboard Input
5.2 Introduction to Control Structures
5.3 Types of Control Structures
5.4 Relational Operators
5.5 One-Way Selection
5.6 Two-Way Selection
5.7 Multi-Way Selection
5.8 Fixed Repetition
5.9 Conditional Repetition
5.10 Control Structures and Graphics
5.11 Summary
5.1 Keyboard Input

Program input has seemed less than impressive so far. Frequently, you have executed programs multiple times with different values hard-coded in various program statements. Such an approach is hardly user-friendly and will sell little software. We are still a few chapters away from the attractive windows-style input that is provided in Java. At the same time, programs without input, especially when you know control structures that behave differently with different values, are quite tedious.

Java has many, many ways to do program input and output. There are about 50 or 60 classes available for this. What they all have in common is they are complex. In the future you will have a proper understanding of the mechanisms involved to manipulate keyboard input. It is not very practical to wait for some distant chapter to come around before we start entering data during program execution. What we will use again is the Expo class. There are special inputting methods in the Expo class which take care of the complicated stuff behind the scenes and allow you to simple enter the information you want.

Program Java0501.java, in figure 5.1, enters a name during program execution. Execute the program several times and experiment. You will note that two program statements are numbered to help explain how to use these features.

Figure 5.1

```
// Java0501.java
// This program (and the next few programs) demonstrate user keyboard input during program execution.
// This particular program demonstrates keyboard input of a String in a text window using the Expo class.
public class Java0501
{
    public static void main (String args[])
    {
        System.out.println("nJAVA0501.JAVA
        System.out.print("Enter name ===> ");   // Line 1
        String name = Expo.enterString();     // Line 2
        System.out.println("nName Entered: "+name);
        System.out.println();
    }
}
```
Please understand the explanations correctly that follow. You will learn which program statements are necessary to use keyboard input during program execution. You will also learn where to place these statements and how to use them.

```
System.out.print("Enter name ===>> ");   // Line 1
```

Line 1 is technically not necessary for the program to work, but it makes the program much more user friendly. This line is technically called a prompt. It asks or “prompts” the user for information. Without a prompt, the user would just see the cursor flashing on the screen and have no idea what to enter. Chances are the user would not even know he/she is supposed to enter something. With the prompt, the user knows not only that he/she is supposed to enter something; he/she will also know what they are supposed to enter.

It is very common to use a `System.out.print` statement for your prompt. With a `System.out.println` statement, the computer would perform a carriage-return after the prompt and the user would enter the information on the next line. Sometimes you might want this, but most of the time, we want to enter the information on the same line.

```
String name = Expo.enterString();  // Line 2
```

Line 2 is the action statement. It is here that the data entered at the keyboard is transferred to the computer memory. The `enterString` method "reads" in an entire string of characters from the keyboard until the <Enter> key is pressed. You can use this statement as many times as necessary in a program to get all the required program input during execution.

Program `Java0502.java`, in figure 5.2, demonstrates how to write a program with multiple lines of input entered from the keyboard during program execution. The `enterString` method is used three times, with three different variables.
Figure 5.2

public class Java0502 {
    public static void main (String args[])
    {
        System.out.println("nJAVA0502.JAVA n");
        System.out.print("Enter Line 1 ===&gt; ");
        String input1 = Expo.enterString();
        System.out.print("Enter Line 2 ===&gt; ");
        String input2 = Expo.enterString();
        System.out.print("Enter Line 3 ===&gt; ");
        String input3 = Expo.enterString();
        System.out.println();
        System.out.println(input1);
        System.out.println(input2);
        System.out.println(input3);
        System.out.println();
    }
}

There is no limit to how much data a program can enter. You just need to make sure you have a separate variable for each piece of data, and that the data is the correct type.

Speaking of types, it appears that keyboard input during program input happens with strings only. At least that has been the evidence during the last two program examples. Is it possible to enter numbers during program execution? Program Java0503.java, in figure 5.3, enters two integers and tries to display the sum of the two numbers.
The output of Program `Java0503.java`, in figure 5.3, may seem very strange. How did $100 + 200 = 100200$? The problem is that we still entered strings. Remember, when we use the plus (+) operator with strings, we do not get *addition* at all. We get *String Concatenation*. It simply joins the 2 strings together. In program `Java0503.java`, the 2 *numbers* were entered as strings. This is why the program simply joined the two numbers together and did not actually “add” them.
You were just getting excited that some means is introduced that allows some modest program input. Now you find that the input does not work for numbers and that just deflates your excitement. Well do not deflate too much. `enterString` is not the only input method of the `Expo` class. Program `Java0504.java`, in figure 5.4, makes a small, but very significant change by using `enterInt`.

**Figure 5.4**

```java
// Java0504.java
// This program uses the Expo.enterInt() method to enter integers from the keyboard.
// It is now possible to correctly add the two numbers.

public class Java0504 {
    public static void main (String args[]) {
        System.out.println("JAVA0504.JAVA");
        System.out.print("Enter 1st Number ===>>  ");
        int number1 = Expo.enterInt();
        System.out.print("Enter 2nd Number ===>>  ");
        int number2 = Expo.enterInt();
        int sum = number1 + number2;
        System.out.println();
        System.out.println(number1 + " + " + number2 + " = " + sum);
    }
}
```

You might be wondering about other data types now. What about real numbers and characters? Program `Java0505.java`, in figure 5.5, shows a program that displays the mean of three real numbers entered at the keyboard. Now we are using the `enterDouble` command.
Figure 5.5

// Java0505.java
// This program demonstrates how to use Expo.enterDouble() for three separate double keyboard inputs,
// which are used to display the mean.

public class Java0505
{
    public static void main (String args[])
    {
        System.out.println("JAVA0505.JAVA");
        System.out.print("Enter Number 1 ====> ");
        double n1 = Expo.enterDouble();
        System.out.print("Enter Number 2 ====> ");
        double n2 = Expo.enterDouble();
        System.out.print("Enter Number 3 ====> ");
        double n3 = Expo.enterDouble();
        System.out.println();
        System.out.println(n1);
        System.out.println(n2);
        System.out.println(n3);
        double mean = (n1 + n2 + n3) / 3;
        System.out.println();
        System.out.println("The mean is " + mean);
        System.out.println();
    }
}

The next program shows the enterChar command. This command might not be used as much as the other three, but it is perfect for things like entering someone’s middle initial. Another example would be if you created a computerized multiple choice test. The user needs to enter A, B, C, D or E. This command is designed for any time you want to enter just a single character.
Figure 5.6

// Java0506.java
// This program demonstrates how to use Expo.enterChar() which is ideal for entering a single letter.

public class Java0506
{
    public static void main (String args[])
    {
        System.out.println("nJAVA0506.JAVA");
        System.out.print("Enter First Name: ===>> ");
        String firstName = Expo.enterString();
        System.out.print("Enter Middle Initial: ===>> ");
        char middleInitial = Expo.enterChar();
        System.out.print("Enter Last Name: ===>> ");
        String lastName = Expo.enterString();
        System.out.println();
        System.out.println("Your full name is: " + firstName + " " + middleInitial + ". " + lastName);
        System.out.println();
    }
}

Expo Class Input Methods

Expo.enterInt() is used to enter an int from the text screen.
Expo.enterDouble() is used to enter a double from the text screen.
Expo.enterString() is used to enter a String from the text screen.
Expo.enterChar() is used to enter a char from the text screen.
5.2 Introduction to Control Structures

The previous chapter focused on using methods and parameters, which are an integral part of Java program organization. The intention in Exposure Java is to present program organization in stages. A thorough understanding of program organization and program design is vital in modern computer science. At the same time, you also need to learn the syntax of a programming language. You cannot write proper Java programs unless you know the Java sentence structure or syntax. A prerequisite for a creative writing course is knowledge of grammar.

Frequently, it is mentioned that programming language syntax is trivial. The essence of programming is design, data structures and algorithms. This is very true, and very lovely, but language syntax tends to be trivial only if you know syntax. You will get very frustrated with your programs when they do not compile because of syntax problems.

In an earlier chapter we mentioned that program execution follows the exact sequence of program statements. That was a true, but also a rather incomplete explanation. There is a lot more to the program flow picture. We can expand on the exact program sequence by stating that program flow follows the sequence of program statements, unless directed otherwise by a Java control structure.

Program Flow

Program Flow follows the exact sequence of listed program statements, unless directed otherwise by a Java control structure.

Programs in any computer language require control structures. It may appear that all our program examples were written without the use of any control structures, but the control was subtle. As mentioned earlier, control was provided by a sequence of program statements. That type of control is called simple sequence.

Simple sequence alone is not very satisfactory for programs of any type of consequence. Programs constantly make decisions. A payroll program needs to change the amount paid, if the hours exceed 40 per week. The same payroll program can have many variations of tax deductions based on the number of dependents claimed. A payroll program also needs to have the ability to repeat the same actions for additional employees. In other words, a program needs to have the ability to repeat itself, or repeat certain program segments. The language features that allow that type of control will be introduced in this chapter.
5.3 Types of Control Structures

Program-execution-flow is controlled by three general types of control structures. They are simple sequence, selection, and repetition. Java provides syntax, and special keywords for each of these three control structures. Before we look at actual Java source code required to implement control, let us first take a look at diagrams that explain each control structure.

**Simple Sequence**

Simple sequence holds no surprises. A series of program statements are executed in the exact sequence that they are written. Altering the program execution logic requires altering the sequence of the program statements.

**Selection**

Frequently, programs cannot follow a single, simple sequence, path. Decisions need to be made like should the applicant be hired or not? Does the employee get overtime pay? Which tax bracket is the deduction to be computed from?
Selection is also called conditional branching or decision making. The program flow encounters a special condition. The value of the condition determines if the program flow will “branch off” from the main program sequence. There are 3 types of selection: one-way, two-way and multiple-way. Three diagrams, one for each type of selection, will be shown.

**One-Way Selection**

<table>
<thead>
<tr>
<th>Program Statement</th>
<th>Condition</th>
<th>Program Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>True</td>
<td></td>
</tr>
<tr>
<td></td>
<td>False</td>
<td>Program Statement</td>
</tr>
</tbody>
</table>

Selection control structures use a special conditional statement. If the condition is true, some action is performed, such as branching off to another sequence of program statements. In the case of one-way selection, the true condition branches off. If the condition is false, the program flow continues without change in program sequence.

Consider the analogy of driving South from Dallas to Austin. Along the way you check if your gas tank is low. If the tank is low, you stop for gas, and then continue to Austin. If the tank is not low you continue to drive south. Keep in mind that regardless of the tank condition, you are heading to Austin.
**Two-Way Selection**

The *two-way selection* control structure also checks to see if some special condition is *true*. But there is a significant difference in how the program flow is handled. With *one-way* selection, a *true* condition means to do something, like get off the road and get gas, before continuing. The *two-way selection* structure selects one direction, or the other direction, but never both.

The *one-way* analogy describes a trip traveling south from Dallas to Austin. Regardless of the gas tank situation, the trip travels to Austin in the same car. Now consider an analogy for *two-way selection*. You are now driving from Austin back to Dallas. The highway you would take is I35 (Interstate 35). When you are about 70 miles from Dallas, shortly after you pass the town of Hillsboro the highway *forks*. It splits in two. You need to decide between going left which means you will take I35W (Interstate 35 West) to Fort Worth or going right which means you will take I35E (Interstate 35 East) to Dallas.
Multiple-way (or *Multi-way*) selection is a little different from one-way and two-way selection. In this control structure the condition is not very explicit. There is a special *selection variable* that is used along with several *selection constants*. The *selection variable* is compared with each *selection constant* until a *match* is found. The *condition* that you do not really see is if the *selection variable* equals a particular *selection constant*. When it finds a match it executes the corresponding program statement.

Multi-way selection is a commonly used control structure that simulates many situations in real life. Many times there are more than 2 things to choose from. For example, do you want chocolate, vanilla, strawberry, pistachio, rocky road, cookies & cream, mint chocolate chip or moo-llennium crunch ice cream for dessert? Do you plan to go to Harvard, Yale, Stanford, Princeton, Texas Tech, UT, OU or some other university? In fact, any time you take a multiple choice test, you are experiencing real life multi-way selection.
Repetition

Another common application occurs when repetition is required. A grade book program needs to average grades for every student in a class of twenty-five students. A payroll program needs to process paychecks for many employees. Practically everything you can imagine is done multiple times. Nobody is interested in repeating program source code 500 times for some task that is to be performed 500 times. We want to create one program segment, and place this segment in some type of loop control structure that repeats 500 times.
5.4 Relational Operators

Both the selection control structure diagrams and the repetition diagram indicate a change of program flow occurring after some condition. Understanding conditional statements is the essence of understanding, and using, control structures. However, before we plunge into the syntax of the various conditional statements, you need to understand the relational operators that are used by Java in the conditional statements.

**Conditional Statement Definition**

A conditional statement is a program expression, which evaluates to **true** or **false**.

Most conditional statements require a relational operator.

All conditions must be placed inside parentheses.

Consider an expression, such as 5 + 4. Does that expression evaluate to **true** or **false**? Neither, it evaluates to 9. A relational operator is required to make an expression evaluate to **true** or **false**. Java has six relational operators: **Equals**, **Not equals**, **Greater than**, **Less than**, **Greater than or equal to**, and **Less than or equal to**.

The idea of conditional statements that are based on a relational operator can be considered in regular English statements:

*If we save more than $200.00 a month, we can go on a vacation*

*If your SAT score is high enough you will be admitted to college, otherwise you will not be able to go to college*

*Repeat calling established customers until you have 25 surveys*
## Java Relational Operators

<table>
<thead>
<tr>
<th>Name</th>
<th>Operator</th>
<th>Expression</th>
<th>Evaluates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equals</td>
<td>==</td>
<td>5 == 5</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5 == 10</td>
<td>false</td>
</tr>
<tr>
<td>Not equals</td>
<td>!=</td>
<td>50 != 25</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td></td>
<td>100 != 100</td>
<td>false</td>
</tr>
<tr>
<td>Less than</td>
<td>&lt;</td>
<td>100 &lt; 200</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td></td>
<td>200 &lt; 100</td>
<td>false</td>
</tr>
<tr>
<td>Greater than</td>
<td>&gt;</td>
<td>200 &gt; 100</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td></td>
<td>200 &gt; 200</td>
<td>false</td>
</tr>
<tr>
<td>Less than or equals</td>
<td>&lt;=</td>
<td>100 &lt;= 200</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td></td>
<td>200 &lt;= 200</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td></td>
<td>200 &lt;= 100</td>
<td>false</td>
</tr>
<tr>
<td>Greater than or equals</td>
<td>&gt;=</td>
<td>100 &gt;= 200</td>
<td>false</td>
</tr>
<tr>
<td></td>
<td></td>
<td>200 &gt;= 200</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td></td>
<td>200 &gt;= 100</td>
<td>true</td>
</tr>
</tbody>
</table>

The relational operators shown in this diagram will be used in the Java example programs that demonstrate the different control structures. Be careful not to confuse the equality operator (==) with the assignment operator (=).
5.5 One-Way Selection

The simplest control structure is one-way selection. Basically, one-way selection says to perform some indicated action if a condition is true. If the condition is false, continue to march as if the control structure did not exist.

Program Java0507.java, in figure 5.7, computes the Christmas bonus for employees of some imaginary company. The company is generous, but only for those who work very hard. Employees will get a $1000 Christmas bonus only if they sell at least $500,000 in merchandise. You will see two outputs for this program, as will be the case with future program examples.

Figure 5.7

```
// Java0507.java
// This program demonstrates one-way selection with <if>.
// Run the program twice. First with Sales equals to 300,000
// and a second time with Sales equals 500,000.
public class Java0507
{
    public static void main (String args[])
    {
        System.out.println("JAVA0507.JAVA");
        System.out.print("Enter Sales ==>> ");
        double sales = Expo.enterDouble();
        double bonus = 0.0;
        System.out.println();
        if (sales >= 500000.0)
            bonus = 1000.0;
        System.out.println("Christmas bonus: "+bonus);
        System.out.println();
    }
}
```
Before we look at other control structure program examples, we need to first understand certain items that you will find in all the program examples. In particular, look at the conditional program statement. The conditional expression, \((sales >= 500000.0)\), is placed inside parentheses. This is not for looks. Java requires the use of parenthesis with ALL its conditional statements that are used in ALL of its control structures. The program statement: \(\text{bonus} = 1000.0;\) is placed below the conditional statement, and it is indented. This is not a Java requirement, but it is a common convention in program development. Both statements below are totally identical from the Java compiler’s point of view. However, you are expected to use the style of the first program segment.

\[
\text{if (sales >= 500000.0) bonus = 1000.0;}
\]

\[
\text{if (sales >= 500000.0) bonus = 1000.0;}
\]

The first example places the program statement that will be executed, \textit{if the condition is true}, indented on the next line. The second example follows the same program logic, but the entire statement is placed on one line. The first example is the preferred approach and will be followed in this book.

Using indentations with control structures helps to identify the program statement or statements that will only be executed \textit{if the condition is true}. You have seen the use of indentation in the main method. You can consider that the braces of the methods control the simple sequence of the program statements.

### Indentation Rule

Java syntax uses freeform program style. Program statements may be placed on multiple lines with or without indentation.

By convention, control structures and their conditional statements are placed on one line. The program statement that is executed, if the condition is true, is placed on the next line, and indented below the conditional statement.

The next program example is designed to prove to you that indentation is not connected at all with program control. Program \texttt{Java0508.java}, in figure 5.8, demonstrates that the program statement following the completed \texttt{if} statement is
executed regardless of the conditional statement. This is not such an odd idea. It must be possible to execute more than one statement when a conditional statement is true. Check the peculiar output of the next program, in figure 5.8 and make sure never to make this type of logic error.

Figure 5.8

```java
// Java0508.java
// This program demonstrates one-way selection with <if>.  
// It also shows that only one statement is controlled.
// Run the program twice. First with Sales equals to 300,000 
// and then a second time with Sales equals to 500,000.

public class Java0508
{
    public static void main (String args[])
    {
        System.out.println("JAVA0508.JAVA");
        System.out.print("Enter Sales ==> ");
        double sales = Expo.enterDouble();
        double bonus = 0.0;
        System.out.println();
        if (sales >= 500000.0)
            System.out.println("CONGRATULATIONS!");
            System.out.println("You sold half a million dollars in merchandise!");
            System.out.println("You will receive a $1000 Christmas Bonus!");
            System.out.println("Keep up the good work!");
            bonus = 1000.0;
            System.out.println();
            System.out.println("Christmas bonus: "+ bonus);
            System.out.println();
    }
}
```
The “multiple statements” dilemma is solved with **block structure**. You have been using block structure for some time, and you never realized it. With every program's main method you used braces `{ }`. Now look at program `Java0509.java`, in figure 5.9 and notice how the braces create a block that identifies the program statements that need to be “controlled” by the `if` condition.
The braces, and the program statements between the braces, form a block. Block structured languages allow the ability to identify a block of program code and use this with all control structures. The meaning of a block is consistent. Everything between the braces of the main program method forms the main program code. Likewise every statement between braces following a control structure will be controlled by the conditional statement of the control structure.
One-Way Selection

General Syntax:

if (condition true)
  execute program statement

Specific Examples:

if (counter > 100)
  System.out.println("Counter exceeds 100");

Use braces { } and block structure to control multiple program statements.

if (savings >= 10000)
{
  System.out.println("It’s skiing time");
  System.out.println("Let’s pack");
  System.out.println("Remember your skis");
}

5.6 Two-Way Selection

A slight variation of one-way selection is two-way selection. With two-way selection there are precisely two paths. The computer will either take one path, if the condition is true, or take the other path if the condition is false. It is not necessary to check a conditional statement twice. The reserved word else makes sure that the second path is chosen when the conditional statement evaluates to false. The program example that follows is very short. The program considers the value of a student’s SAT score. College admission is granted or denied based on the SAT score. Note the indentation style of the if ... else control structure that is shown in figure by Java0510.java, in figure 5.10.
Figure 5.10

// Java0510.java
// This program demonstrates two-way selection with <if..else>.
// Run the program twice: First with 1200, then with 1000.

public class Java0510
{
    public static void main (String args[])
    {
        System.out.println("JAVA0510.JAVA");
        System.out.print("Enter SAT = > > >   ");
        int sat = Expo.enterInt();
        System.out.println();

        if (sat >= 1100)
            System.out.println("You are admitted");
        else
            System.out.println("You are not admitted");
        System.out.println();
    }
}

Program Java0511.java, in figure 5.11, demonstrates block structure with two-way selection. The same type of SAT program is shown with a few statements added to help demonstrate the block structure. Pay attention to the style that is used with the block structure and the if ... else. You will notice different styles in different books. This is fine, as long as the style is consistent. At this stage, it is
wise to use the same style for your program assignments that you see displayed in this book. It will be less confusing for you. If you adopt a different style, or even worse, no style at all, and indent statements randomly, you will find it easy to get confused about the meaning of your program statements. It will also be very difficult for your teacher to help you if he/she cannot make heads or tails of your unreadable Java code.

Figure 5.11

```java
// Java0511.java
// This program demonstrates two-way selection with <if..else>.
// Multiple statements require the use of block structure.
// Run the program twice: First with 1100, then with 1099.

public class Java0511 {
    public static void main (String args[]) {
        System.out.println("nJAVA0511.JAVA
"));
        System.out.print("Enter SAT =\n");
        int sat = Expo.enterInt();
        System.out.println();
        if (sat >= 1100)
        {
            System.out.println("You are admitted");
            System.out.println("Orientation will start in June");
        }
        else
        {
            System.out.println("You are not admitted");
            System.out.println("Please try again when your SAT improves.");
        }
        System.out.println();
    }
}
```

![Screenshot of Java0511.java program output]

JAVA0511.JAVA
Enter SAT ==>>> 1100
You are admitted
Orientation will start in June
Press any key to continue...
Something else demonstrated by the 2 outputs in Figure 5.11 is that `if...else` has no mercy what so ever. In the first output, the student with exactly 1100 squeaks by and gets accepted. In the second output, the student with a 1099 misses the mark by one point. Too bad for him. He is not admitted no matter how close he was.

**Two-Way Selection**

**General Syntax:**

```
if (condition true)
    execute first program statement
else  // when condition is false
    execute second program statement
```

**Specific Example:**

```java
if (gpa >= 90.0)
    System.out.println ("You’re an honor graduate");
else
    System.out.println ("You’re not an honor graduate");
```
5.7 Multi-Way Selection

The final selection structure needs to be watched carefully. *Multiple-Way Selection* is very useful, but can cause some peculiar problems if you are not aware of certain peculiar quirks that lurk in Java.

Multiple-Way Selection occurs whenever you have more than 2 choices. The next program example uses a grade value in the range ['A'..'F']. Each grade number is associated with a number range. The *multi-way selection* structure does not use the conditional statement logic of the *if* structures. Multi-way selection is accomplished with the *switch* command which uses a selection variable with a value. The *selection variable* value is compared with a group of designated *case* values. When a match is found, the statements following the *case* value are executed. Look at the syntax of program *Java0512.java*, in figure 5.12, and observe two new Java keywords, *switch*, and *case*.

**Figure 5.12**

```java
// Java0512.java
// This program demonstrates multi-way selection with <switch> and <case>.
// This program compiles, but displays illogical output.

public class Java0512
{
    public static void main (String args[])
    {
        System.out.println("nJAVA0512.JAVA
n");
        System.out.print("Enter Letter Grade  ===>>  ");
        char grade = Expo.enterChar();
        System.out.println();

        switch (grade)
        {
            case 'A' :
                System.out.println("90 .. 100 Average");
            case 'B' :
                System.out.println("80 .. 89 Average");
            case 'C' :
                System.out.println("70 .. 79 Average");
            case 'D' :
                System.out.println("60 .. 69 Average");
            case 'F' :
                System.out.println("Below 60 Average");
        }
        System.out.println();
    }
}
```
The `switch` structure in the `Java0512.java` program example only works with certain data types. These are `int`, `char` and `String`. `switch` does not work with `double` or `boolean`. In this case `grade` is an `int` and it is also the `selection variable` that determines the matching process. Note that `grade` is placed between the parentheses of the `switch` structure.

With `switch` there are usually at least 3 possible outcomes. Each of the possible outcomes starts with the reserved word `case`, followed by a constant value that could be a possible match for the `selection variable`. A colon ( : ) is required to
separate the *selection constant* from the program statement that needs to be performed if a match is found.

How about the program output in figure 5.12, does it make sense? Please do realize that nothing is wrong. The *switch* statement behaves exactly the way that it was designed for Java. Be aware of that and make sure that you handle your programs accordingly. Program execution will compare each *case* with the *selection variable*, and the moment a match is made the program flow branches off. Now keep in mind that not only the program statement of the matching case statement executes, but every program statement in the entire *switch* block that follows.

Program *Java0513.java* cures the problem of the previous *switch* example. You will notice a new Java keyword has been added to every *case* statement. It is *break*, and it is placed between every *case* statement. *break* exits the current program block, which in this case means that program execution jumps to the end of the *main* method. Technically, you can use *break* in other situations to make various jumps to different program segments. For our purposes, we will only use *break* with *multiple selection*.

Program *Java0513.java*, shown in figure 5.13, has five possible matches or cases. If no match is found, the program flow arrives at a special *case*, called *default*. The program will execute the *default* statement when no match is found. In general, default is what a computer executes when nothing is specified. The *default* case does need a *break* statement since it is already at the end of the *switch* structure. The output examples are now logically correct.

Figure 5.13

```
// Java0513.java
// This program demonstrates multi-way selection with <switch> and <case>.
// The program adds <break> and <default>.
// The use of <break> is required for logical output.
// The <default> case occurs when no other case matches.
public class Java0513
{
    public static void main (String args[])
    {
        System.out.println("JAVA0513.JAVA\n");
        System.out.print("Enter Letter Grade  ===>>  ");
        char grade = Expo.enterChar();
        System.out.println();
        switch (grade)
        {
            case 'A' :
                System.out.println("90 .. 100 Average");
                break;
            case 'B' :
                System.out.println("80 .. 89 Average");
                break;
        }
    }
}
```
case 'C':
    System.out.println("70 .. 79 Average");
    break;
case 'D':
    System.out.println("60 .. 69 Average");
    break;
case 'F':
    System.out.println("Below 60 Average");
    break;
default :
    System.out.println("No Match Found");
}
System.out.println();
}

Figure 5.13 Continued
Program {

Java0514.java, shown in figure 5.14, demonstrates that each case in a switch can control multiple program statements just as easily as a single program statement. You can put as many program statements as you wish between the case and the break.

Figure 5.14

// Java0514.java
// This program demonstrates that multiple program statements
// can be placed between the <case> and the <break> statements.

public class Java0514
{
    public static void main (String args[])
    {
        System.out.println("nJAVA0514.JAVA
";
        System.out.print("Enter Letter Grade ==>>  ");
        char grade = Expo.enterChar();
        System.out.println();

        switch (grade)
        {
            case 'A' :
                System.out.println("90 .. 100 Average");
                System.out.println("Excellent!");
                break;
            case 'B' :
                System.out.println("80 .. 89 Average");
                System.out.println("Good");
                break;
            case 'C' :
                System.out.println("70 .. 79 Average");
                System.out.println("Fair");
                break;
            case 'D' :
                System.out.println("60 .. 69 Average");
        } 
    } 
}
System.out.println("Poor");
break;
case 'F' :
    System.out.println("Below 60 Average");
    System.out.println("Bad");
    break;
default :
    System.out.println("No Match Found");
}
System.out.println();

Figure 5.14 Continued

JAVA0514.JAVA
Enter Letter Grade ===> B
80 .. 89 Average
Good
Press any key to continue...

JAVA0514.JAVA
Enter Letter Grade ===> C
70 .. 79 Average
Fair
Press any key to continue...

JAVA0514.JAVA
Enter Letter Grade ===> d
No Match Found
Press any key to continue...
While the previous couple programs work, they only work when CAPITAL letters are entered. If you enter a lowercase letter, even a lowercase ‘a’, ‘b’, ‘c’, ‘d’ or ‘f’, the program will display “No Match Found”. This is precisely what happened in the last output shown in Figure 5.14. Program Java0515.java, shown in figure 5.15, shows how we can be a little more flexible and allow for both CAPITAL and lowercase letters. Essentially, multiple cases can yield the same result.

Figure 5.15

```java
// Java0515.java
// This program demonstrates how to allow for both capital and lowercase letters,

public class Java0515 {
    public static void main(String args[]) {
        System.out.println("JAVA0515.JAVA");
        System.out.print("Enter Letter Grade ===> ");
        char grade = Expo.enterChar();
        System.out.println();
        switch (grade) {
            case 'A' :
            case 'a' :
                System.out.println("90 .. 100 Average");
                System.out.println("Excellent!");
                break;
            case 'B' :
            case 'b' :
                System.out.println("80 .. 89 Average");
                System.out.println("Good");
                break;
            case 'C' :
            case 'c' :
                System.out.println("70 .. 79 Average");
                System.out.println("Fair");
                break;
            case 'D' :
            case 'd' :
                System.out.println("60 .. 69 Average");
                System.out.println("Poor");
                break;
            case 'F' :
            case 'f' :
                System.out.println("Below 60 Average");
                System.out.println("Bad");
                break;
            default :
                System.out.println("No Match Found");
        }
        System.out.println();
    }
}
```
Program **Java0516.java**, shown in figure 5.16, shows that a **switch** statement can also be controlled by an integer value. In this program a high school student needs to enter a **9**, **10**, **11** or **12**. The output will be **Freshman**, **Sophomore**, **Junior** or **Senior** respectively.

**Figure 5.16**

```java
// Java0516.java
// This program demonstrates multi-way selection can also be controlled with an <int> variable.

public class Java0516
{
    public static void main (String args[])
    {
        System.out.println("JAVA0516.JAVA");
        System.out.println("Enter Letter Grade ==>> a 90 .. 100 Average");
        System.out.println("Excellent! Press any key to continue...");
        int grade = Expo.enterInt();
        System.out.println();

        switch (grade)
        {
            case 9  : System.out.println("Freshman"); break;
            case 10 : System.out.println("Sophomore"); break;
            case 11 : System.out.println("Junior"); break;
            case 12 : System.out.println("Senior"); break;
            default : System.out.println("You are not in high school.");
        }
    }
}
```
Figure 5.16 Continued

```java
System.out.println();
}
```

JAVA0516.JAVA
What grade are you in? ==> 9
Freshman
Press any key to continue...

JAVA0516.JAVA
What grade are you in? ==> 10
Sophomore
Press any key to continue...

JAVA0516.JAVA
What grade are you in? ==> 11
Junior
Press any key to continue...

JAVA0516.JAVA
What grade are you in? ==> 12
Senior
Press any key to continue...
Program **Java0517.java**, shown in figure 5.17, is like a souped-up version of the previous program. Now the person can be in any grade from *Kindergarten* (grade 0) through *Graduate School* (any grade above 16). As with characters, multiple integer *cases* can give you the same outcome.

**Figure 5.17**

```java
// Java0517.java
// This is a more complicated program where <int> is being used
// to control multi-way selection.
// In this example multiple cases can yield the same result.

public class Java0517
{
  public static void main (String args[])
  {
    System.out.println("JAVA0517.JAVA");
    System.out.print("What grade are you in? ===>> ");
    int grade = Expo.enterInt();
    System.out.println();

    switch (grade)
    {
      case 0 :
      case 1 :
      case 2 :
      case 3 :
      case 4 :
      case 5 :
        System.out.println("Elementary School");
        break;
      case 6 :
      case 7 :
      case 8 :
        System.out.println("Middle School");
        break;
      case 9 :
      case 10 :
      case 11 :
      case 12 :
        System.out.println("High School");
        break;
      case 13 :
      case 14 :
      case 15 :
      case 16 :
        System.out.println("College");
        break;
      default :
        System.out.println("Graduate School");
    }
    System.out.println();
  }
}
```
Figure 5.17 Continued

JAVA0517.JAVA
What grade are you in? ===>>  7
Middle School
Press any key to continue...

JAVA0517.JAVA
What grade are you in? ===>>  11
High School
Press any key to continue...

JAVA0517.JAVA
What grade are you in? ===>>  2
Elementary School
Press any key to continue...

JAVA0517.JAVA
What grade are you in? ===>>  13
College
Press any key to continue...
In 2011, *Java Version 7* (jdk 1.7.0) was released with several new features. One of these new features is that the `switch` now works with strings. Program *Java0518.java*, shown in figure 5.18, asks you to enter the first name of one of Leon Schram’s children. You then need to enter *John, Greg, Maria* or *Heidi*. The program will then give you information specific to the child you entered.

**Figure 5.18**

```java
// Java0518.java
// This program demonstrates multi-way selection can
// also be controlled with a <String> variable.
// This is a new feature with Java Version 7 (jdk 1.7.0).

public class Java0518 {
    public static void main (String args[])
    {
        System.out.println("nJAVA0518.JAVA");
        System.out.println("Enter the first name of one of Leon Schram’s children. ==>> ");
        String firstName = Expo.enterString();
        System.out.println();

        switch (firstName)
        {
            case "John" : 
                System.out.println("This is Mr. Schram's older son.");
                break;
            case "Greg" :
                System.out.println("This is Mr. Schram's younger son.");
                break;
            case "Maria"
                System.out.println("This is Mr. Schram's older daughter.");
                break;
            case "Heidi"
                System.out.println("This is Mr. Schram's younger daughter.");
                break;
            default
                System.out.println("This is not one of Mr. Schram's children.");
        }
        System.out.println();
    }
}
```
Program **Java0519.java**, shown in figure 5.19, is again like a souped-up version of the previous program. Now you can enter the first name of any member of Leon Schram’s immediate family. To improve readability a line is skipped between each group of cases which yield the same outcome.

**Figure 5.19**

```java
// This is a more complicated program where <String> is being used to control multi-way selection.
// In this example multiple cases can yield the same result.

public class Java0519 {
    public static void main(String[] args) {
        System.out.println("
JAVA0519.JAVA
");
        System.out.print("Enter the first name of someone in Leon Schram's family. ===>> ");
        String firstName = Expo.enterString();
        System.out.println();
        switch (firstName) {
            case "Isolde":
                System.out.println("This is Mr. Schram's wife.");
                break;
            case "John":
            case "Greg":
                System.out.println("This is one of Mr. Schram's sons.");
                break;
            case "Maria":
            case "Heidi":
                System.out.println("This is one of Mr. Schram's daughters.");
                break;
            case "Mike":
            case "David":
                System.out.println("This is one of Mr. Schram's sons-in-law.");
                break;
            case "Diana":
                System.out.println("This is Mr. Schram's daughter-in-law.");
                break;
            case "Jessica":
            case "Haley":
            case "Brenda":
            case "Mari":
                System.out.println("This is one of Mr. Schram's granddaughters.");
                break;
            case "Anthony":
            case "Alec":
            case "Maddox":
            case "Jaxon":
            case "Braxton":
                System.out.println("This is one of Mr. Schram's grandsons.");
        }
    }
}
```
break;

case "Austrid":
case "Ingrid":
    System.out.println("This is one of Mr. Schram's sisters.");
    break;

case "Remy":
    System.out.println("This is Mr. Schram's brother.");
    break;

case "Darlene":
case "Kassi":
case "Holli":
    System.out.println("This is one of Mr. Schram's nieces.");
    break;

case "Gene":
case "Sean":
case "Blake":
    System.out.println("This is one of Mr. Schram's nephews.");
    break;

default:
    System.out.println("This is not someone in Mr. Schram's immediate family.");
    System.out.println("Make sure you spell the name correctly and only capitalize the first letter.");
    System.out.println();
}

Figure 5.19 Continued
Multiple-Way Selection

General Syntax:

switch(selectionVariable)
{
    case selectionConstant :
        program statement;
        program statement;
        :    :    :
        break;

    case selectionConstant :
        program statement;
        program statement;
        :    :    :
        break;

    default
        program statement;
        program statement;
        :    :    :
}

Specific Example:

switch(courseGrade)
{
    case 'A'  :   points = 4;    break;
    case 'B'  :   points = 3;    break;
    case 'C'  :   points = 2;    break;
    case 'D'  :   points = 1;    break;
    case 'F'  :   points = 0;    break;
    default   :   points = 0;
}

The default statement is used to handle the situation when a proper match is not found. Frequently an error message is used to indicate that no match was found.
5.8 Fixed Repetition

We now make a major jump in the control structure department. We have just looked at two control structures. Both control structures were different but they each were some type of selection control. Now you will learn how to repeat program segments.

The title of this section is repetition. That is not the only word. You can also say looping, and the formal computer science word is iteration. The name does not matter, as long as you realize that this is a control structure that executes certain program segments repeatedly.

The Java language has three different types of iterative control structures. In this section we will explore fixed iteration with the for loop structure. Fixed means that you know – ahead of time – how many times a certain program segment will repeat. You can determine this just by looking at the code of the loop structure. It is not necessary to run the program to find this out.

Before we actually look at a Java loop structure, let us first look at a program that helps to motivate the need for efficiency. Program Java0520.java displays 40 advertising lines for Joe’s Friendly Diner. Now I do not care, and hopefully you do not care why Joe wants to display the following message 40 times:

*Eat at Joe’s friendly diner for the best lunch value*

The point is that we need to write a program that will generate that message. Joe is paying us good money, and we are just lowly programmers following the orders of a customer.

There is one way to accomplish this task with the knowledge you have acquired so far. You can write 40 output statements. Program Java020.java, in figure 5.20 demonstrates the inefficient repetition without using any loop structures. That approach is not really so bad, and with a little clever block copying, you are quickly done in no time. This is true, but Joe only wants 40 lines. What about a program that needs to print 100 or 1000 or 1,000,000 flyers for Joe’s friendly diner? Are you then going to generate 1,000,000 program statements that each displays the same thing? You see the point here. There must be a better way. Please do not ever use this approach.
public class Java0520
{
    public static void main(String args[])
    {
        System.out.println("JAVA0520.JAVA
");
        System.out.println("Eat at Joe's friendly diner for the best lunch value");
        System.out.println("Eat at Joe's friendly diner for the best lunch value");
        System.out.println("Eat at Joe's friendly diner for the best lunch value");
        System.out.println("Eat at Joe's friendly diner for the best lunch value");
    }
}
Figure 5.20 Continued

```
JAVA0520.JAVA
Eat at Joe's friendly diner for the best lunch value
Eat at Joe's friendly diner for the best lunch value
Eat at Joe's friendly diner for the best lunch value
Eat at Joe's friendly diner for the best lunch value
Eat at Joe's friendly diner for the best lunch value
Eat at Joe's friendly diner for the best lunch value
Eat at Joe's friendly diner for the best lunch value
Eat at Joe's friendly diner for the best lunch value
Eat at Joe's friendly diner for the best lunch value
Eat at Joe's friendly diner for the best lunch value
Eat at Joe's friendly diner for the best lunch value
Eat at Joe's friendly diner for the best lunch value
Eat at Joe's friendly diner for the best lunch value
Eat at Joe's friendly diner for the best lunch value
Eat at Joe's friendly diner for the best lunch value
Eat at Joe's friendly diner for the best lunch value
Eat at Joe's friendly diner for the best lunch value
Eat at Joe's friendly diner for the best lunch value
Eat at Joe's friendly diner for the best lunch value
Eat at Joe's friendly diner for the best lunch value
Eat at Joe's friendly diner for the best lunch value
Eat at Joe's friendly diner for the best lunch value
Eat at Joe's friendly diner for the best lunch value
Eat at Joe's friendly diner for the best lunch value
Eat at Joe's friendly diner for the best lunch value
Eat at Joe's friendly diner for the best lunch value
Press any key to continue...
```
We apologize for that last horrendous program. It was necessary since we want to make a strong point why loop structures are very beneficial in computer science. The same *Joe’s Diner* program is going to be demonstrated again. However, this time the program uses a *for* loop control structure to assist with the repetition process. Program *Java0521.java*, in figure 5.21, has the exact same output as the previous program example, but is considerably shorter.

**Figure 5.21**

```
// Java0521.java
// This program displays 40 identical lines efficiently
// with one println statement and a loop structure.

public class Java0521 {
    public static void main(String args[]) {
        int k;
        for (k = 1; k <= 40; k++)
            System.out.println("Eat at Joe’s friendly diner for the best lunch value");
    }
}
```

How is the repetition accomplished in this program? Ah . . . with the special magic of the *for* loop structure. The only new keyword in the repetition program is the *for* statement. Right now let us focus strictly on that statement. It is repeated below and on the next page to make your life simpler. The only part that looks familiar is a set of parentheses, which you have already seen in use by previous control structures. Otherwise, the statement looks pretty odd with a bunch of small statements all tossed together.

```
for (k = 1; k <= 40; k++)
```

The *for* loop structure consists of three distinct parts that work together to control the program flow.

To understand this you need to understand something very important about the *for* loop. It is designed to repeat some process a certain number of times. For example, if you were told to do something exactly 10 times, what would you need to do? You would need to count each time you did it. *for* loops are the same. They need a *counter*. A variable is used for this counting. The technical name for this *counting* variable is the *Loop Control Variable* or *LCV*. Now let us look at the 3 parts of the *for* loop.
for (k = 1; k <= 40; k++)

Part 1:  \( k = 1; \)
This initializes the Loop Control Variable or counter, which in this case means that \( k \) starts with the value 1.

Part 2:  \( k <= 40; \)
The middle part states the Loop Entry Condition. As long as the value of \( k \) is less than or equal to 40, the program statement in the line following the \texttt{for} statement will be executed, again and again if necessary. As \( k \) counts it eventually gets to 38, 39, 40… 41… and when \( k = 41 \) the loop will stop. It will not execute a 41st time. It stops because the \texttt{boolean} expression \( k <= 40; \) is no longer \texttt{true}. Once the Loop Entry Condition becomes \texttt{false}, the loop stops. It is very common to think of this middle part as an Exit Condition. Either way, just remember the loop body is entered as long as the loop condition equals \texttt{true}, and it is exited when the loop condition becomes \texttt{false}.

Part 3:  \( k++ \)
This third part tells the computer how the counter is going to count. \( k++ \) tells the computer to count forward by ones. You might wonder why the computer needs to be told how to count. Keep in mind that you do not have to count by 1s. When small children play hide-and-seek, you can frequently hear them counting by 5s. “85, 90, 95, 100! Ready or not, here I come!”

Program \texttt{Java0522.java}, in figure 5.22, demonstrates that the LCV can handle two jobs at the same time. In this program the LCV checks that the program statement is repeated 15 times. In addition, the LCV also serves as a visual counter because the value of the LCV is displayed each time the loop structure statement is executed. This program also shows a common Java convention to define a variable at the same place where the variable is initialized.

Figure 5.22

```java
// Java0522.java
// This program displays consecutive numbers 1 through 15.
// It also shows how the loop control variable may be
// defined inside the <for> program statement.
public class Java0522
{
    public static void main(String args[])
    {
        System.out.println("nJAVA0522.JAVA\n");
        \texttt{for (int k = 1; k <= 15; k++)}
            System.out.print(k + " ");
        System.out.println("n");
    }
}
```
If we look at the previous program it is important to understand what is happening. \( k \) is initialized to 1. That is its value the first time through the loop. So the statement: \( \text{System.out.print}(k + " "); \) will print a 1 followed by a couple spaces. The computer then does the \( k++; \) to count for the next time through. \( k \) is now 2. The computer then checks to see if \( k \leq 40; \) ? It still is, so it goes through the loop again. This process repeats until the 15 is displayed. After the computer displays the 15, the \( k++; \) makes the counter store 16. Now the \textbf{boolean} expression \( k \leq 40; \) is \textbf{false} so the loop will stop and not execute a 16\textsuperscript{th} time.

Program \texttt{Java0520.java}, in figure 5.20, is a slight variation on the previous program. This program demonstrates how to repeat multiple statements in a \texttt{for} loop. Yes, you guessed it, your good friend \textit{block structure} with braces \{ \} is back to save the day. Try this program out and comment out the braces to see the result.

**Figure 5.20**

// Java0520.java
// This program demonstrates how to use block structure
// with a <for> loop control structure.

public class Java0520
{
    public static void main(String args[])
    {
        System.out.println("nJAVA0520.JAVA\n");
        int k;
        for (k = 1; k <= 10; k++)
        {
            System.out.println("####################################");
            System.out.println("Line Number " + k);
        }
    System.out.println();
    }
}
We mentioned earlier that the for loop does not always have to count by 1. The next program will demonstrate this. Program Java0524.java, in figure 5.24, has five separate loop control structures. Each loop uses a different incrementer.

The first loop uses p++, and increments by 1. This is old news and the loop structure displays numbers from 1 to 15.

The second loop uses q+=3, and increments by 3. This loop structure displays numbers 1, 4, 7, 10, 13.

The third loop uses r-- and this loop does not increment at all; it decrements the counter by 1 each time. The loop is counting backwards. This control structure displays the numbers from 15 to 1.

The fourth loop uses s+=0.5 and increments in fractional amounts. This loop structure displays numbers 0 to 3.0 counting by 0.5 or ½.

The fifth loop uses t++, with t as a char and increments one character letter each time. This loop structure displays letters A to Z. It might seem strange to count with letters, but that is actually possible, and sometimes useful.
Figure 5.24

// Java0524.java
// This program displays various counting schemes.
// It also demonstrates the versatility of the <for> loop.

public class Java0524 {
    public static void main(String args[]) {
        System.out.println("JAVA0524.JAVA\n");
        for (int p = 1; p <= 15; p++)
            System.out.print(p + " ");
        System.out.println("\n");
        for (int q = 1; q <= 15; q+=3)
            System.out.print(q + " ");
        System.out.println("\n");
        for (int r = 15; r >= 1; r--)
            System.out.print(r + " ");
        System.out.println("\n");
        for (double s = 0; s <= 3; s+=0.5)
            System.out.print(s + " ");
        System.out.println("\n");
        for (char t = 'A'; t <= 'Z'; t++)
            System.out.print(t + " ");
        System.out.println("\n");
    }
}
Fixed Repetition

Java has a variety of control structures for repetition.

Other computer science terms for repetition are *looping* and *iteration*.

*Fixed Repetition* is done with the *for* loop structure.

**General Syntax:**

```
for (Part1; Part2; Part3)
    loop body;
```

The *for* loop has three distinct parts:

*Part1* initializes the Loop Control Variable (LCV).

*Part2* sets the exit condition for the loop.

*Part3* determines how the LCV changes.

**Specific Examples:**

```
for (k = 1; k <= 10; k++)
    System.out.println("Java is 10 times more fun");
```

Just like the if statement, use braces { } and block structure to control multiple program statements.

```
for (k = 20; k >= 0; k--)
{
    System.out.println("Rocket Launch Countdown");
    System.out.println();
    System.out.println("T minus " + k + " seconds.");
    System.out.println();
}
```
5.9 Conditional Repetition

The `for` loop is designed for fixed repetition. Fixed repetition means you know – ahead of time – how many times you want the loop to repeat. The next program example, in figure 5.22, is designed to show you that fixed repetition is not always possible. The program wants a user to enter a PIN (Personal Identification Number). A user should not be given access to the program unless they know the correct PIN. In this program, the PIN is 5678. When the program is compiled and executed, you should notice the output is… a little strange.

**Figure 5.22**

```java
// Java0525.java
// This program is supposed to keep repeating until a correct PIN# of 5678 is entered.
// The program does not work because the <for> loop is used in an inappropriate manner.
// The <for> loop is meant for "fixed" repetition.
// Entering a PIN# is an example of "conditional" repetition.

public class Java0525 {
    public static void main(String args[]) {
        System.out.println("JAVA0525.JAVA\n");
        for (int j = 1; j <= 10; j++) {
            System.out.println();
            System.out.print("Enter 4 digit PIN#.  -->  ");
            int pin = Expo.enterInt();
            if (pin != 5678)
                System.out.println("Incorrect Password.  Try Again.");
            else
                System.out.println("\nYou are now logged in.  Welcome to the program.");
        }
    }
}
```

**NOTE:** The output of this program might cause a little confusion.
OK, this output should look seriously messed up. At first it is fine. The first 4 attempts at entering the password are incorrect. The user is told they did not enter the correct pin and to try again. On the 5th try something weird happens. The user enters the correct PIN of 5678. Since the password is correct, there is no message saying, “This is not the correct PIN.” The strange part is the user has to enter the PIN again, and again and again and again and AGAIN! Even though the PIN was entered correctly 5 times, the user is still not permitted access. What is even weirder is that on the 10th attempt, when the user enters an incorrect password, the user is given access.
What causes this problem is the fact that the wrong type of loop was used. We used a for loop. This particular fixed repetition had the statement:

\[
\text{for (int } j = 1; j <= 10; j++)
\]

This is a loop that will count from 1 to 10, and therefore repeat 10 times no matter what. It did not matter what PINs were entered. This program will ask for a PIN 10 times, and then give you access to the program, regardless of what PINs are entered. The purpose of the previous program was to demonstrate that fixed repetition is not always appropriate.

There are many situations where you know that something has to be repeated, but you do not know how many times. The entry of a password can be correct the first time or it may take many repeated attempts before the correct password is entered. Another situation may be based on time, which means that somebody who uses the computer slowly gets few repetitions and a fast operator gets many repetitions. Video games frequently repeat the same scene, but this repetition is usually not based on a fixed amount. Video games rely more on number of lives left. I think you get the picture. Life is full of situations where something is repeated, but it is not repeated a known number of times. Instead, it is repeated until something happens. This is called conditional repetition.

The while Loop

Program Java0526.java, in figure 5.26, introduces the while loop. This loop is a conditional loop. Like earlier control structures, you will note that the condition is placed inside parentheses. Remember the for loop has 3 parts to its heading. The middle part is the condition we just mentioned. The for loop also takes care of initializing the Loop Control Variable and also altering it in the heading. The while loop only has the loop entry condition in its heading. Initializing the Loop Control Variable must be done before the loop. Altering the LCV must be done inside the loop. In program Java0526.java the LCV is pin. It is defined and initialized before the while loop. Inside the while loop we make the user reenter the pin again and again until he/she enters the correct pin of 5678.

Figure 5.26

```java
// Java0526.java
// This program fixes the problem of the previous program by using a while command.
// Now the loop will stop when the correct PIN# of 5678 is entered.

public class Java0526
{
    public static void main(String args[])
    {
```

Page 194   Exposure Java 2012, PreAPCS Edition   04-25-12
```java
{ System.out.println("\nJAVA0523.JAVA\n");
    int pin = 0;

    while (pin != 5678)
    {
        System.out.println();
        System.out.print("Enter 4 digit PIN#.  -->  ");
        pin = Expo.enterInt();
        if (pin != 5678)
            System.out.println("Incorrect Password.  Try Again.");
    }

    System.out.println("\nYou are now logged in.  Welcome to the program.";)
}
```

Figure 5.26 Continued

Now the output makes sense. On the first 4 attempts, when the **pin** is entered incorrectly, the user is asked to “Try Again.” On the 5\textsuperscript{th} attempt, when the correct **pin** of 5678 is entered, the loop stops as it should and the user is granted access.
**Flags**

Computer Science is full of funny terms. *Encapsulation, Polymorphism, Concatenation*, just to mention a few. When dealing with *conditional* loops, it is important to understand what the *flag* is. The *flag* is the special value that makes the loop stop. Maybe this value is entered by the user. Maybe it is computed in the loop. Whatever special value makes the loop stop is the *flag*. A good way to remember that is to think of a race track. Cars go in a loop again and again. The race keeps going until the checkered *flag* is waved and the race is over.

### Conditional Repetition

**General Syntax:**

initialize condition variable before the while loop
while(condition is true)
{
    loop body
    alter condition variable in the loop body
}

**Specific Example:**

```java
int pin = 0;  // initialize condition variable
while(pin != 5678)
{
    System.out.print("Enter pin. ===>> ");
    pin = Expo.enterInt();  // alter condition variable
}
System.out.println("Welcome.");
```
Failure to handle the loop condition variable can bring about bizarre results, including an infinite loop program execution. Carefully hard code the following **nono** and **yesyes** program segments, of figures 5.27 and 5.28, in your brain.

**Figure 5.27**

<table>
<thead>
<tr>
<th>Program Segment NoNo #1</th>
<th>Program Segment YesYes #1</th>
</tr>
</thead>
<tbody>
<tr>
<td>int x = 0;</td>
<td>int x = 0;</td>
</tr>
<tr>
<td>while(x &lt; 10)</td>
<td>while(x &lt; 10)</td>
</tr>
<tr>
<td>System.out.println(x);</td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>x++;</td>
</tr>
<tr>
<td></td>
<td>System.out.println(x);</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>

The loop condition variable, x, never changes.
The loop will not exit.

The loop condition variable, x, changes.
The loop exits when x reaches 10.

**Figure 5.28**

<table>
<thead>
<tr>
<th>Program Segment NoNo #2</th>
<th>Program Segment YesYes #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>int x;</td>
<td>int x = 0;</td>
</tr>
<tr>
<td>while(x &lt; 10)</td>
<td>while(x &lt; 10)</td>
</tr>
<tr>
<td>{</td>
<td>{</td>
</tr>
<tr>
<td>x++;</td>
<td>x++;</td>
</tr>
<tr>
<td>System.out.println(x);</td>
<td>System.out.println(x);</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>

The loop condition variable, x, is never initialized.
The program will not compile.

The loop condition variable, x, is initialized. The program will compile and execute correctly.
Fixed Repetition vs. Conditional Repetition

*Fixed Repetition* describes a situation where you know – ahead of time – how many times you want the loop to repeat.

An example would be drawing exactly 100 circles on the screen.

The command for fixed repetition is `for`.

*Conditional Repetition* describes a situation where you do NOT know how many times the loop will repeat.

The loop has to repeat until some *condition* is met.

An example would be entering a password.

The command for conditional repetition is `while`.

---

5.10 Control Structures and Graphics

Java has many very nice graphics features. You learned some of the graphics capabilities in the last chapter. In the last section of this chapter you will investigate some more graphics program capabilities. However, there is a major difference. In the last chapter you learned how to use many methods that were part of the *Expo* class. Graphic displays are not simply a matter of calling the right method for the right purpose. You can also perform some neat tricks by using control structures. Control structures combined with graphic method calls can have some very nice visual effects.

You will see shortly that interesting patterns can be created with simple straight lines that are repeatedly drawn in slightly different locations. As a matter of fact,
straight lines can create the illusion of forming curves. You will also see a simple introduction to animation. Moving graphic images relies heavily on the use of control structures.

Program **Java0527.java**, in figure 5.29, starts nice and simple by drawing vertical lines that are 10 pixels apart. Note that **y1** and **y2** values, which control the top-down values, are fixed at **100** and **500**. With the help of a control structure you can draw many lines with a single **drawLine** statement.

**Figure 5.29**

```java
import java.awt.*;
import java.applet.*;

public class Java0527 extends Applet
{
    public void paint(Graphics g)
    {
        int x1 = 100;
        int y1 = 100;
        int x2 = 100;
        int y2 = 500;
        for (int k = 1; k <= 81; k++)
        {
            Exps.drawLine(g,x1,y1,x2,y2);
            x1 += 10;
            x2 += 10;
        }
    }
}
```
Program **Java0528.java**, in figure 5.30, makes a few small changes to the previous program to draw horizontal lines. This time \(x_1\) and \(x_2\) are fixed values.

**Figure 5.30**

```java
// Java0528.java
// This program shows how a control structure can be used with graphics.
// This program draws horizontal lines, because the starting y and ending y values are the same.
import java.awt.*;
import java.applet.*;

public class Java0528 extends Applet
{
    public void paint(Graphics g)
    {
        int x1 = 100;
        int y1 = 50;
        int x2 = 900;
        int y2 = 50;
        for (int k = 1; k <= 50; k++)
        {
            Expo.drawLine(g,x1,y1,x2,y2);
            y1 += 10;
            y2 += 10;
        }
    }
}
```

![Applet Viewer: Java0528 class](image.png)
In the graphics department where control structures are combined with graphics commands, horizontal lines and vertical lines are pretty comfortable. One notch higher on the complexity ladder is programs with diagonal lines. Diagonal lines will require that both the \( x \) values and the \( y \) values are altered.

Program **Java0529.java**, in figure 5.31, shows a different approach from the two previous programs. Before either the \( x \) value were altered or the \( y \) values were altered. Now both will be.

The four coordinate values required to draw the diagonal lines are first initialized before the **for** loop. After each line is drawn, each of the four values is changed in preparation for the next line.

**Figure 5.31**

```
// Java0529.java
// This program shows how a control structure can be used with graphics.
// This program draws diagonal lines, because all 4 variables -- x1, y1, y2, y2 -- change.

import java.awt.*;
import java.applet.*;

public class Java0529 extends Applet
{
    public void paint(Graphics g)
    {
        int x1 = 50;
        int x2 = 200;
        int y1 = 50;
        int y2 = 300;
        for (int k = 1; k <= 60; k++)
        {
            Expo.drawLine(g,x1,y1,x2,y2);
            x1 += 10;
            x2 += 10;
            y1 += 5;
            y2 += 5;
        }
    }
}
```

Figure 5.31 Continued

Figure 5.32

Program **Java0530.java**, in figure 5.32, plays another twist with straight lines. This time it is not a set of \( x \) values or a set of \( y \) values that change, but one coordinate point changes with each repetition.
Straight lines are nice, but repetition can also be done with ovals, stars, rectangles, or any other graphics object. The only limit is your imagination. Program **Java0533.java**, in figure 5.33, is another very small program. This one draws ovals. Like the previous programs, it sets up all of the variables before the **for** loop. The loop makes sure 40 ovals are drawn. If you look at the output on the next page, you will see the ovals. The first one is the one in the middle. Every time the loop is repeated only one thing is changed. The variable **hr** (for *horizontal radius*) has 10 added to it. Nothing else is changed. This means every oval has the same center *x*,*y* coordinate, and the same *vertical radius*. Since the *horizontal radius* increase by 10 each time, each oval get 20 pixels fatter. If you are wondering, “Why 20, and not 10?” Remember that the *diameter* of a circle/oval is *twice* its *radius*. If the radius increases by 10, the diameter increases by 20.
Figure 5.33

// Java0531.java
// This program demonstrates several ovals. All of the ovals have the same center and vertical radius.
// The horizontal radius keeps changing.

import java.awt.*;
import java.applet.*;

class Java0531 extends Applet
{
    public void paint(Graphics g)
    {
        int x = 500;
        int y = 325;
        int hr = 50;
        int vr = 100;
        for (int k = 1; k <= 40; k++)
        {
            Expo.drawOval(g, x, y, hr, vr);
            hr += 10;
        }
    }
}
5.11 Summary

This chapter introduced controls structures. Prior to this chapter, programs were strictly controlled by the sequence of program statements. Java, like most programming languages, provides three types of control structures. There are simple sequence, selection and repetition control structures. Simple sequence provides control by the order in which program statements are placed in a program. This is very important, but very limited control. It is neither possible to make any decisions nor is it possible to repeat any block of program statements.

Input from the keyboard is with a variety of methods from the Expo class. You do need to realize that method enterString enters strings, method enterInt enters integers, enterDouble enters real numbers and enterChar enters characters.

Selection provides three types of decision control structures. There is one-way selection with if, two-way selection with if..else, and multi-way selection with switch.

Repetition provides two types of loop control structures. There is fixed repetition with for and conditional repetition with while.

This chapter is not the complete story. Every program example uses only a single condition with selection and repetition. It is possible to use multiple conditions with control structures and it is also possible to nest one control structure inside another control structure. These features will be shown in later chapters.

This chapter concluded by returning to graphics programming. You did not learn any new methods to create graphics display. However, you did see examples of how control structures can enhance the appearance of graphic displays.