Chapter V

Control Structures I

Chapter VI Topics

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

The previous chapter focused on using methods and parameters, which is an integral part of Object Oriented Programming. The intention in Exposure Java is to present OOP in stages. A thorough understanding of Object Oriented Programming is vital in modern computer science. At the same time, you also need to learn the syntax of a program language. You cannot write proper Java OOP 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 program 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 I mentioned that program execution follows the exact sequence of program statements. That was 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 the 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.

You will note that the title of this chapter is *Control Structures I*. You have guessed correctly. There will be a second control structure chapter. The second chapter will explain a variety of advanced control structure features. At this stage, all we want to do is get our feet wet and feel the temperature of the water. You will be diving in the deep-end before you know it.

### 5.2 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 one 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**

```
Program Statement
    ↓
Program Statement
    ↓
Program Statement
    ↓
Program Statement
```

**Selection**

```
Program Statement
    ↓
Program Statement
```

**Repetition**

```
Program Statement
    ↓
Program Statement
    ↓
Program Statement
```

Chapter V   Control Structures I   5.3
Simple sequence holds no surprises. A series of program statements are executed in the exact sequence that they are written. Altering the program sequence requires altering the sequence of the program statements.

**Selection**

Frequently, programs cannot follow a single, simple sequence, path. Decisions need to be made. 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. Three diagrams will be shown. The first diagram shows **one-way** selection, the second diagram shows **two-way** selection, and the third one shows **multiple-way** selection.

**One-Way Selection**

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 not 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 in Las Vegas gambling in the casino. You are placing all your money on one large bet. If you win the bet, you will fly home first-class, otherwise you will hitchhike home. Perhaps you argue that in both cases the final destination is home. True, but there are two totally different travel modes getting there.
Multiple selection is a commonly used control structure that simulates many situations in real life. The program flow encounters a group of conditions, one after the other. If the condition is true, the program flow branches off to execute some other statement. The main program flow is rejoined below all the conditional statements.

Multiple selection in real life is encountered with any multiple choice test, an ATM machine or any other situation where you encounter more than two options that can be selected.
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.

5.3 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, and less than or equal.

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*

<table>
<thead>
<tr>
<th>Name</th>
<th>Operator</th>
<th>Expression</th>
<th>Evaluates</th>
</tr>
</thead>
</table>

5.8 Exposure Java, 2005 Edition 08-06-05
<table>
<thead>
<tr>
<th></th>
<th>Operator</th>
<th>Examples</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equals</td>
<td><code>==</code></td>
<td>5 == 5 K == 10</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>depends</td>
</tr>
<tr>
<td>Not equals</td>
<td><code>!=</code></td>
<td>50 != 25 100 != 100</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>false</td>
</tr>
<tr>
<td>Less than</td>
<td><code>&lt;</code></td>
<td>100 &lt; 200 P &lt; Q</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>depends</td>
</tr>
<tr>
<td>Greater than</td>
<td><code>&gt;</code></td>
<td>100 &gt; 200 P &gt; Q</td>
<td>false</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>depends</td>
</tr>
<tr>
<td>Less than or equals</td>
<td><code>&lt;=</code></td>
<td>25 &lt;= 26 25 &lt;= 25</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>true</td>
</tr>
<tr>
<td>Greater than or equals</td>
<td><code>&gt;=</code></td>
<td>1000 &gt;= 1000 K &gt; (P + Q)</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>depends</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 (`=`).

5.4 Keyboard Text Input
Program input has seemed less than impressive so far. Frequently, you have executed programs multiple times with different values hard-coded in the program. 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, is very tedious.

Program input in Java is not a simple matter. Java is a wonderful program language for many reasons, but in the area of program input Java is quite complex. You are several chapters away from properly understanding 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.

I propose a good solution to this problem. We can start using keyboard input right now, and you will learn what Java features are necessary to accommodate program input. At the same time do not expect any explanation on these features. Basically, you are told use this, it works and do not bother ask any why questions.

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

Figure 5.1

```
// Java0501.java
// This program demonstrates user keyboard input during program execution.
// Many program features will be used that will be explained later.

import java.util.*;           // Line 1
public class Java0501
{
    public static void main (String args[])
    {
        System.out.println("JAVA0501.JAVA
        Scanner input = new Scanner(System.in);   // Line 2
        System.out.print("Enter name  ===>  ");   // Line 3
        String name = input.nextLine();      // Line 4
        System.out.println("Name Entered: "+name);
        System.out.println();
    }
}
```

Figure 5.1 Continued
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. However, there will not be explanations to help you understand why these statements work as they do. That will come later.

```java
import java.util.Scanner; // Line 1

Scanner input = new Scanner(System.in); // Line 2
```

In the previous chapter you learned that many classes in Java are stored in standard libraries. Access to such classes requires an import statement at the head of the program. We need access to the `Scanner` Java class, which is located in the `java.util` library. Line 1 imports the necessary library for program input.

```java
Scanner input = new Scanner(System.in); // Line 2
```

What can I say about Line 2? Actually plenty, but this is neither the time nor the place. Line 2 creates a very important variable for you, called `input`. This statement must be placed in the `main` method before the `input` is used for any type of keyboard data entry.

```java
System.out.print("Enter name ===> "); // Line 3
```

Line 3 is the easiest statement to understand. It is a standard text output statement using `System.out.print`. This is known as the prompt. The next program statement will stop program execution and wait for the appropriate keyboard input. Without the prompt the program user has no clue what is happening and certainly does not know what type of input is required. Always use a prompt with any type of keyboard input during program execution.
Line 4 is the action statement. It is here that the data entered at the keyboard is transferred to the computer memory. The `nextLine` 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. In particular, pay close attention to the special statements necessary for keyboard input. Do you see that they are identical to the previous program? The only difference is that line 4, the statement with the `nextLine` method, is used three times for three sets of input.

The aim of this program is to enter three names. You will see that there are three separate prompts that request the appropriate information from the keyboard. You might try and remove the prompts from the program and see what happens. It is still possible to compile and execute the program, but it is not user-friendly.

```java
String name = input.nextLine(); // Line 4

Figure 5.2
// Java0502.java
// This program demonstrates how to use <nextLine> for three separate String keyboard inputs.
import java.util.Scanner;
public class Java0502
{
    public static void main (String args[])
    {
        System.out.println("JJava0502.JAVA");
        Scanner input = new Scanner(System.in);
        System.out.println("Enter Line 1 ===>> ");
        String input1 = input.nextLine();
        System.out.println("Enter Line 2 ===>> ");
        String input2 = input.nextLine();
        System.out.println("Enter Line 3 ===>> ");
        String input3 = input.nextLine();
        System.out.println(input1);
        System.out.println(input2);
        System.out.println(input3);
    }
}

Figure 5.2 Continued
```
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.

**Figure 5.3**

```java
// Java0503.java
// This program demonstrates <String> objects concatenation with keyboard entered data.

import java.util.Scanner;

public class Java0503 {
    public static void main (String args[]) {
        System.out.println("Java0503.JAVA");
        Scanner input = new Scanner(System.in);
        System.out.print("Enter 1st Number ===>> ");
        String number1 = input.nextLine();
        System.out.print("Enter 2nd Number ===>> ");
        String number2 = input.nextLine();
        String sum = number1 + number2;
        System.out.println();
        System.out.println(number1 + " + " + number2 + " = " + sum);
    }
}
```

**Java0503.java Output**

```
Java0503.JAVA
Enter 1st Number ===>> 100
Enter 2nd Number ===>> 200
100 + 200 = 100200
```

Program **Java0503.java**, in figure 5.3, provides ample proof that *string input it is*. Two perfectly good numbers were entered and the addition of the two numbers resulted in **concatenation** with output **100200**. Numbers they are **not**.
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. With the input.nextLine() statement only string input is possible. Program Java0504.java, in figure 5.4, makes a small, but very significant change to input.nextInt() and now you can enter integers.

Figure 5.4

// Java0504.java
// This program uses the <nextInt> method to enter integers from the keyboard.
// It is now possible to correctly add the two numbers.

import java.util.Scanner;

public class Java0504
{
    public static void main (String args[])
    {
        System.out.println("nJava0504.JAVA
        ");
        Scanner input = new Scanner(System.in);
        System.out.print("Enter 1st Number ===>> ");
        int number1 = input.nextInt();
        System.out.print("Enter 2nd Number ===>> ");
        int number2 = input.nextInt();
        int sum = number1 + number2;
        System.out.println();
        System.out.println(number1 + " + " + number2 + " = " + sum);
        System.out.println("n
        ");
    }
}

Java0504.java Output

Java0504.JAVA
Enter 1st Number ===>> 100
Enter 2nd Number ===>> 200
100 + 200 = 300

There remains one more program example in this exciting keyboard input section. Is it possible to enter real numbers? Program Java0505.java, in figure 5.5, shows a program that displays the mean of three real numbers entered at the keyboard. This time the nextInt() method is changed to nextDouble().

Figure 5.5

// Java0505.java
// This program demonstrates how to use <nextDouble> for three separate double keyboard inputs.

5.14 Exposure Java, 2005 Edition 08-06-05
import java.util.Scanner;

public class Java0505
{
    public static void main (String args[])
    {
        Scanner input = new Scanner(System.in);
        System.out.println("JAVA0505.JAVA");
        double n1 = input.nextDouble();
        System.out.println("Enter Number 1 ===>> ");
        double n2 = input.nextDouble();
        System.out.println("Enter Number 2 ===>> ");
        double n3 = input.nextDouble();
        System.out.println("Enter Number 3 ===>> ");
        double mean = (n1+n2+n3)/3;
        System.out.println();
        System.out.println(n1);
        System.out.println(n2);
        System.out.println(n3);
        System.out.println("The mean is "+mean);
        System.out.println("\n\n");
    }
}

JAVA0505.JAVA

Enter Number 1 ===>> 1.1
Enter Number 2 ===>> 22.22
Enter Number 3 ===>> 333.333

1.1
22.22
333.333

The mean is 118.88433333333334

Scanner Class Input Methods

nextLine() is used to enter string information.

nextInt() is used to enter integer information.

nextDouble() is used to enter real number information.

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 Java0505.java, in figure 5.6, computes the year-end bonus for employees of some imaginary company. Every employee automatically gets a $250.00 bonus. However, this company wants to reward its top sales people. The bonus is increased by an additional $500.00 if employee sales are equal to or exceed the half million dollar mark for the year. You will see two outputs in figure 5.6, as will be the case with future program examples. Additionally, note the keyboard.nextDouble() statement, which is used to indicate that input and keyboard are identifiers. It is not required to use input.

Figure 5.6

```java
// Java0506.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.

import java.util.Scanner;
public class Java0506
{
    public static void main (String args[])
    {
        System.out.println("JAVA0506.JAVA");
        Scanner keyboard = new Scanner(System.in);
        System.out.print("Enter Sales ===>>  ");
        double sales  = keyboard.nextDouble();
        double bonus = 250.00;
        if (sales >= 500000.0)
            bonus += 500.0;
        System.out.println("Yearly bonus:       "+bonus);
    }
}
```

Java0506.java Output #1

JAVA0506.JAVA

Enter Sales ===>>  300000
Yearly bonus:  250.0

Figure 5.6 Continued

Java0506.java Output #2

JAVA0506.JAVA
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, \( (\text{Sales} \geq 500000.0) \), 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: `Bonus += 500;` 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.

```java
if (Sales >= 500000)
    Bonus += 500;

if (Sales >= 500000) Bonus += 500;
```

The first example places the program statement that will be executed, *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 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 Java0507.java, in figure 5.7, demonstrates that the program statement following the completed 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 5.7 and make sure never to make this type of logic error.

Figure 5.7

```java
// Java0507.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.
import java.util.Scanner;
public class Java0507 {
    public static void main (String args[])
    {
        System.out.println("JAVA0507.JAVA
        Scanner keyboard = new Scanner(System.in);
        System.out.print("Enter Sales  ===>>  ");
        Double sales  = keyboard.nextDouble();
        double bonus = 250.00;
        if (sales >= 500000.0)
        bonus += 500.0;
        System.out.println("Your sales >=       500,000.00");
        System.out.println("You will receive    500.00 extra bonus.");
        System.out.println("Yearly bonus:       " + bonus);
        System.out.println();
    }
}
```

Figure 5.7 Continued

Java0507.java Output #1

JAVA0507.JAVA

5.18 Exposure Java, 2005 Edition 08-06-05
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 **Java0508.java**, in figure 5.8 and notice how the braces create a block that identifies the program statements that need to be “controlled” by the `if` condition.

**Figure 5.8**

```
// Java0508.java
// This program demonstrates one-way selection with <if>.
// It fixes the logic problem of the previous program
// with block structure by using braces.

import java.util.Scanner;
public class Java0508
{
    public static void main (String args[])
    {
        System.out.println("JAVA0508.JAVA");
        Scanner keyboard = new Scanner(System.in);
        System.out.print("Enter Sales  ===>>  ");
        double sales  = keyboard.nextDouble();
        double bonus = 250.00;
        if (sales >= 500000.0)
        {
            bonus += 500.0;
            System.out.println("Your sales >= 500,000.00");
            System.out.println("You will receive 500.00 extra bonus.");
        }
        System.out.println("Yearly bonus: "+ bonus);
        System.out.println();
    }
}
```

**Figure 5.8 Continued**

```
Java0508.java Output #1 with 300000
Yearly Bonus: 250.0
```
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. Notice the two outputs of program Java0508.java. Now does the output make sense?

### One-Way Selection Syntax

One-Way selection general syntax:

```java
if (condition true)
    execute program statement

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

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

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

### 5.5 Two-Way Selection

5.20 Exposure Java, 2005 Edition 08-06-05
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 Java0509.java, in figure 5.9.

Figure 5.9

```
// Java0509.java
// This program demonstrates two-way selection with <if..else>.

import java.util.Scanner;

public class Java0509
{
    public static void main (String args[])
    {
        System.out.println("JAVA0509.JAVA
");
        Scanner keyboard = new Scanner(System.in);
        System.out.print("Enter SAT ===>> ");
        int sat = keyboard.nextInt();
        if (sat >= 1100)
        System.out.println("You are admitted");
        else
        System.out.println("You are not admitted");
        System.out.println();
    }
}
```

Java0509.java Output #1 with 1000

JAVA0509.JAVA
Enter SAT ===>> 1000
You are not admitted

Java0509.java Output #2 with 1200

JAVA0509.JAVA
Enter SAT ===>> 1200
You are admitted

Program Java0510.java, in figure 5.10, 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.

**Figure 5.10**

```java
// Java0510.java
// This program demonstrates two-way selection with <if..else>. Multiple statements require the use of block structure.

import java.util.Scanner;
public class Java0510 {
    public static void main (String args[])
    {
        Scanner keyboard = new Scanner(System.in);
        System.out.println("JAVA0510.JAVA
        System.out.print("Enter SAT ===>> ");
        int sat = keyboard.nextInt();
        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.");
    }
}
```

**Java0510.java Output #1**

JAVA0510.JAVA
Enter SAT ===>> 1000
You are not admitted
Please try again when your SAT improves.

**Java0510.java Output #2**

JAVA0510.JAVA
Enter SAT ===>> 1200
You are admitted
Orientation will start in June

**Two-Way Selection Syntax**

Two-Way selection general syntax:

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

if (GPA >= 90.0)
    System.out.println ( "You’re an honor graduate");
else
    System.out.println ("You’re not an honor graduate");

5.6 Multiple-Way Selection

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

Multiple-Selection occurs in menu selections, such as an Automated Teller
Machine (ATM), any computerized multiple choice test, and a host of other
applications. In the real world it is usually not a situation of do A or do B.
Normally, there are many different options.

The next program example uses a grade value in the range ['A'..'F']. Each grade
number is associated with a number range. The multiple-selection structure does
not use the conditional statement logic of the if structures. We now have 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 statement following
the case value is executed. Look at the syntax of program Java0511.java, in
figure 5.11, and observe two new Java keywords, switch, and case.

Figure 5.11

// Java0511.java
// This program demonstrates multi-way selection with <switch> and <case>.
// This program compiles, but displays illogical output.

import java.util.Scanner;
public class Java0511
{
    public static void main (String args[])
    {
        System.out.println("inJAVA0511.JAVAin");
        Scanner keyboard = new Scanner(System.in);
        System.out.print("Enter Letter Grade ===>> ");
        String temp = keyboard.nextLine();
        char grade = temp.charAt(0);
        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();
    }
}

Java0511.java Output #1
JAVA0511.JAVA
Enter Letter Grade ===>> C
70 .. 79 Average
60 .. 69 Average
Below 60 Average

Java0511.java Output #2
JAVA0511.JAVA
Enter Letter Grade ===>> A
90 .. 100 Average
80 .. 89 Average
70 .. 79 Average
60 .. 69 Average
Below 60 Average

Java0511.java Output #3
JAVA0511.JAVA
Enter Letter Grade ===>> F
Below 60 Average

The switch structure in the Java0511.java program example requires the use of a simple data type, like a single character. The charAt method is used to convert the String variable to a char variable. Then comes the statement switch(grade) and an opening brace, indicating the start of the control structure program block. grade is placed between parentheses. grade is the selection variable that determines the matching process. Each one of the possible branches starts with

5.24 Exposure Java, 2005 Edition 08-06-05
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 5.18, 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 `Java0512.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. You can use `break` in other situations to make various
jumps to different program segments. Use `break` only with `multiple selection`.

In the program example, shown in figure 5.12, are five possible matches or cases.
If no match is found, the program flow arrives at a special type of `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
three output examples are now logically correct.

---

**Figure 5.12**

```java
// Java0512.java
// This program demonstrates multi-way selection with <switch> and <case>.
// This program adds <break> and <default>.
// The use of <break> is required for logical output.

import java.util.Scanner;
```
public class Java0512
{
    public static void main (String args[])
    {
        System.out.println("JAVA0512.JAVA\n");
        Scanner keyboard = new Scanner(System.in);
        System.out.print("Enter Letter Grade ==>> ");
        String temp = keyboard.nextLine();
        char grade = temp.charAt(0);
        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();
    }
}
```java
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.

### 5.7 Fixed Repetition

We now make a major jump in the control structure department. We have just looked at three control structures. All three 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 the number of times that a program segment repeats, is fixed. You can look
at the code of the loop structure and determine how many items it will repeat. It is not necessary to run the program to determine that.

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 Java0513.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 say 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. That approach is not really so bad, and with a little clever block copying, you are really done in no time. This is true, but Joe only wants 40 lines. What about a program that needs to print 10,000 flyers for Joe’s friendly diner? Are you then going to generate 10,000 program segments that each say the same thing? You see the point here. There must be a better way. Program Java0513.java, in figure 5.13 demonstrates the inefficient repetition without using any loop structures. Please do not ever use this approach.

Figure 5.13

// Java0513.java
// This program displays 60 identical lines very inefficiently
// with 60 separate println statements.

import java.lang.System;

public class Java0513
{
    public static void main(String args[])
    {
        System.out.println("JAVA0513.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");
        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");
        System.out.println("Eat at Joe's friendly diner for the best lunch value");
    }
}

5.28 Exposure Java, 2005 Edition  08-06-05
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");
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");
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");
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");
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");
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");
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");
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");
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");
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");
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");
System.out.println("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
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
I apologize for that last horrendous program. It was necessary since I 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 Java0514.java, in figure 5.14, is considerably shorter than the previous program example.

Figure 5.14

```java
// Java0514.java
// This program displays 60 identical lines efficiently
// with one println statement and a loop structure.

class Java0514 {
    public static void main(String args[]) {
        System.out.println("JAVA0514.JAVA");
        int k;
        for (k = 1; k <= 60; k++)
            System.out.println("EAT AT JOE’S FRIENDLY DINER FOR THE BEST LUNCH VALUE");
    }
}
```

Please take my word for it that program Java0514.java generates the exact same output as the previous, tedious version. Load the program, or type in the code to satisfy yourself. How is the repetition accomplished? Ah . . . with the special magic of the for loop structure. The only new keyword in the repetition program is the for statement. Right now focus strictly on that statement. It is repeated below to make your life simpler. The only part that looks familiar is a set of parentheses, which you have noticed used by previous control structures. Otherwise, the statement looks pretty odd with a bunch of small statements all tossed together.

```
for (k = 1; k <= 60; k++)
```
The **for** loop structure consists of three distinct parts that work together to control the program flow.

**Part 1** initializes the *Loop Control Variable (LCV)*, which in this case means that k starts with the value **1**.

**Part 2** states the *Loop Exit Condition*. As long as the value of k is less than or equal to **60**, the program statement following the **for** parenthesis will be executed.

**Part 3** indicates the manner in which the *Loop Control Variable* is altered. In this case the value of k is incremented by **1** each time through the loop.

The next program example, in figure 5.15, 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 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 5.23**

```java
// Java0515.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 Java0515
{
    public static void main(String args[])
    {
        System.out.println("JAVA0515.JAVA");
        for (int k = 1; k <= 15; k++)
        {
            System.out.print(k + " ");
            System.out.println();
        }
    }
}
```

**Java0515.java Output**

```
1  2  3  4  5  6  7  8  9  10  11  12  13  14  15
```

Program **Java0616.java**, in figure 5.16, is a slight variation on the previous program. This program demonstrates how to execute multiple statements in a loop control structure. Yes, you guessed it, your good friend *block structure* with braces **{ }** is back to save the day. Try this program out and comment out the braces to see the result.
You may get the impression that the `for` loop uses a LCV that always increments by 1. This is not always true. Program `Java0516.java`, in figure 5.17, has five separate loop control structures. Each loop uses a different incremeneter.

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 LCV by 1 each time. 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 from 0 to 3.0.

The fourth loop uses `t++`, with `t` as a `char` and increments one character letter each time. This loop structure displays letters `A` to `Z`.

---

**Figure 5.16**

```java
// Java0516.java
// This program demonstrates how to use block structure
// with a <for> loop control structure.

public class Java0516 {
    public static void main(String args[])
    {
        System.out.println("JAVA0516.JAVA");
        for (int k = 1; k <= 5; k++)
        {
            System.out.println("####################################");
            System.out.println("Line Number " + k);
        }
        System.out.println();
    }
}
```

**Java0516.java Output**

JAVA0516.JAVA

####################################
Line Number 1
####################################
Line Number 2
####################################
Line Number 3
####################################
Line Number 4
####################################
Line Number 5
// Java0517.java
// This program displays various counting schemes.
// It also demonstrates the versatility of the <for> loop.

public class Java0517
{
    public static void main(String args[])
    {
        System.out.println("JAVA0517.JAVA\n");
        for (int p = 1; p <= 15; p++)
            System.out.print(p + "  ");
        System.out.println();
        for (int q = 1; q <= 15; q+=3)
            System.out.print(q + "  ");
        System.out.println();
        for (int r = 15; r >= 1; r--)
            System.out.print(r + "  ");
        System.out.println();
        for (double s = 0; s <= 3; s+=0.5)
            System.out.print(s + "  ");
        System.out.println();
        for (char t = 'A'; t <= 'Z'; t++)
            System.out.print(t + "  ");
        System.out.println("\n\n");
    }
}
### Repetition Control Structures with for

Java has a variety of control structures for **repetition**. Other computer science terms for repetition are **looping** and **iteration**.

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

**for loop syntax:**

```java
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.

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

### 5.8 Conditional Repetition

Fixed iteration is not always possible. There are too many situations where it is established that repetition will take place, but it is not known how frequently something will be repeated. 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.
Computer life is full of situations with unknown repetition. Java is well prepared for the task with two conditional loop structures: The precondition \texttt{while} loop and the postcondition \texttt{do...while} loop.

\begin{center}
\textbf{Note to Teachers and Students with Advanced Knowledge}
\end{center}

It is possible to treat the \texttt{for} loop structure like a conditional loop that is not fixed. In fact, a \texttt{for} loop can be designed to behave exactly like a \texttt{while} loop.

It is my intention to use and treat a \texttt{for} loop like a \textit{fixed} iteration loop and use the \texttt{while} loop and \texttt{do...while} loop for other repetition situations.

This approach is less likely to cause confusion. At some later date, when you are comfortable with all the control structures, you can use them in any appropriate manner.

If this does not make sense to you, good, ignore this little summary box, and move on.

\begin{center}
\textbf{The Precondition \texttt{while} Loop}
\end{center}

Program \texttt{Java0518.java}, in figure 5.26, introduces the \texttt{while} loop. This loop is called a \texttt{precondition} loop. The condition, which determines when the loop exits, is tested before the loop body is started. Like earlier decision structures, and the \texttt{for} loop structure, you will note that the condition is placed inside parentheses.

The \texttt{while} loop structure requires a statement inside the loop body that was not necessary with the \texttt{for} loop. This is an opportunity for the loop condition variable to change. The \texttt{for} loop included a component in the loop heading that incremented the loop counter by some specified amount. There is no equivalent component in the \texttt{while} loop. Only the \textit{loop condition} is checked.

\begin{figure}
\centering
\scalebox{0.8}{Program Java0518.java, in figure 5.26, introduces the \texttt{while} loop. This loop is called a \texttt{precondition} loop. The condition, which determines when the loop exits, is tested before the loop body is started. Like earlier decision structures, and the \texttt{for} loop structure, you will note that the condition is placed inside parentheses. The \texttt{while} loop structure requires a statement inside the loop body that was not necessary with the \texttt{for} loop. This is an opportunity for the loop condition variable to change. The \texttt{for} loop included a component in the loop heading that incremented the loop counter by some specified amount. There is no equivalent component in the \texttt{while} loop. Only the \textit{loop condition} is checked.}
\caption{Figure 5.18}
\end{figure}
public class Java0518 {
    public static void main(String args[]) {
        System.out.println("JAVA0518.JAVA");
        int p = 1;
        int q = 1;
        int r = 15;
        double s = 0;
        char t = 'A';

        while (p <= 15) {
            System.out.print(p + " ");
            p++;
        }
        System.out.println();

        while (q <= 15) {
            q++;
            System.out.print(q + " ");
        }
        System.out.println();

        while (r >= 0) {
            System.out.print(r + " ");
            r--;
        }
        System.out.println();

        while (s < 3) {
            System.out.print(s + " ");
            s += 0.5;
        }
        System.out.println();

        while (t <= 'Z') {
            System.out.print(t + " ");
            t++;
        }
        System.out.println("\n\n");
    }
}
while loop syntax:
initialize condition variable
while(condition is true)
  loop body
  alter condition variable in loop body

x = 0;  // initialize condition variable
while(x < 10)
{
  x++;  // alter condition variable
  System.out.println("x = " + x);
}

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 5.19 and 5.20, in your brain.

Figure 5.19

<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.
The loop condition variable, X, is never initialized. This program will not compile.

The loop condition variable, X, is initialized. The program will compile and execute correctly.

**The Postcondition do...while Loop**

Program [Java0519.java](#) adds a small change to the conditional loop business with the keyword `do`. The body of the loop is placed between the starting `do` keyword, and the closing `while` condition. Since the condition of the loop exit is tested at the conclusion of the loop body, the loop is called postcondition. Program [Java0519.java](#), in figure 5.21, produces the same output as the two previous examples. Once again, your first mission is to understand the syntax of this new loop structure. There are many similarities with the precondition `while` loop, but there are also some very significant differences.

In the later *Control Structures II* chapter, we will discuss when to use which loop structure. Can you think, right now, why you would prefer one loop rather than another loop? Maybe that question is unfair. You have barely gotten a minor taste of these new loop structures. You probably are not in a position to do any kind of choosing. However, you can certainly think about the choices.
public class Java0519
{
    public static void main(String args[])
    {
        System.out.println("JAVA0519.JAVA
");
        int p = 1;
        int q = 1;
        int r = 15;
        double s = 0;
        char t = 'A';
        do
        {
            System.out.print(p + " ");
            p++;
        } while (p <= 15);
        System.out.println();
        do
        {
            System.out.print(q + " ");
            q*3;
        } while (q <= 15);
        System.out.println();
        do
        {
            System.out.print(r + " ");
            r-;
        } while (r >= 0);
        System.out.println();
        do
        {
            System.out.print(s + " ");
            s += 0.5;
        } while (s < 3);
        System.out.println();
        do
        {
            System.out.print(t + " ");
            t++;
        } while (t <= 'Z');
        System.out.println("\n\n");
    }
}
The next section in this chapter will present a variety of small programs. Each program has a control structure that was introduced in this chapter. Our concern will be with the output of each program, and more importantly, develop some methods to determine program output correctly, which involves control structures.

You can expect that on quizzes, tests and the APCS Examination only a program segment or a method is shown. In this chapter, at least, you will be shown a complete program. This may be beneficial if you want to check out the program output for yourself. Each one of the following exercises follows the same format, which is to determine the output of the program execution.

Determining the program output involves creating a small chart that present a tracing of the variable values for each repetition through the control structure. Many students will be able to determine the output of the simpler examples without writing down anything. The trace is shown for each example, simple or not, to get you used to the method. In future chapters you will find that these traces are not quite as simple. Exercise 01 will start of nice and easy.

**Exercise 01**

```java
public class Ex0501
{
    public static void main (String args[])
    {
        for (int x = 1; x < 8; x++)
            System.out.println("x = "+x);
    }
}
```

<table>
<thead>
<tr>
<th>x</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x = 1</td>
</tr>
<tr>
<td>2</td>
<td>x = 2</td>
</tr>
<tr>
<td>3</td>
<td>x = 3</td>
</tr>
<tr>
<td>4</td>
<td>x = 4</td>
</tr>
<tr>
<td>5</td>
<td>x = 5</td>
</tr>
<tr>
<td>6</td>
<td>x = 6</td>
</tr>
<tr>
<td>7</td>
<td>x = 7</td>
</tr>
</tbody>
</table>

**Exercise 02**
public class Ex0502
{
    public static void main (String args[])
    {
        for (int x = 1; x <= 8; x++)
            System.out.println("x = " + x);
    }
}

<table>
<thead>
<tr>
<th>x</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x = 1</td>
</tr>
<tr>
<td>2</td>
<td>x = 2</td>
</tr>
<tr>
<td>3</td>
<td>x = 3</td>
</tr>
<tr>
<td>4</td>
<td>x = 4</td>
</tr>
<tr>
<td>5</td>
<td>x = 5</td>
</tr>
<tr>
<td>6</td>
<td>x = 6</td>
</tr>
<tr>
<td>7</td>
<td>x = 7</td>
</tr>
<tr>
<td>8</td>
<td>x = 8</td>
</tr>
</tbody>
</table>

Exercise 03

public class Ex0503
{
    public static void main (String args[])
    {
        for (int x = 0; x <= 8; x+=2)
            System.out.println("x = " + x);
    }
}

<table>
<thead>
<tr>
<th>x</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x = 0</td>
</tr>
<tr>
<td>2</td>
<td>x = 2</td>
</tr>
<tr>
<td>4</td>
<td>x = 4</td>
</tr>
<tr>
<td>6</td>
<td>x = 6</td>
</tr>
<tr>
<td>8</td>
<td>x = 8</td>
</tr>
</tbody>
</table>

Exercise 04
public class Ex0504
{
    public static void main (String args[])
    {
        int x = 0;
        int y = 0;
        for (y = 1; y <= 25; y++)
        {
            y+=5;
            System.out.println(y);
        }
    }
}

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>0</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>0</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>0</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td>0</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>30</td>
<td>30</td>
</tr>
</tbody>
</table>

Exercise 05

public class Ex0505
{
    public static void main (String args[])
    {
        int x = 0;
        int y = 0;
        for (x = 1; x > 1; x--)
        {
            y++;
            System.out.println("y = " + y);
        }
    }
}

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>y = 0</td>
</tr>
</tbody>
</table>

The for loop condition is never met. The y++ statement is never executed and the output statement can only display zero.

Exercise 06
public class Ex0506
{
    public static void main (String args[])
    {
        int x = 0;
        int y = 0;
        while (x < 5)
        {
            y++;
            x = y;
        }
        System.out.println("y = "+ y);
    }
}

<table>
<thead>
<tr>
<th>y</th>
<th>x</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>y = 5</td>
</tr>
</tbody>
</table>

Exercise 07

public class Ex0507
{
    public static void main (String args[])
    {
        int x = 0;
        int y = 0;
        while (x < 10)
        {
            y = x + 2;
            x = y + 3;
        }
        System.out.println("y = "+ y);
    }
}

<table>
<thead>
<tr>
<th>y</th>
<th>x</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>10</td>
<td>y = 7</td>
</tr>
</tbody>
</table>

Exercise 08
public class Ex0508
{
    public static void main (String args[])
    {
        int x = 0;
        int y = 0;
        while (x < 10)
        {
            y = x * 2;
            x++;
        }
        System.out.println("x = "+ x);
        System.out.println("y = "+ y);
    }
}

<table>
<thead>
<tr>
<th>y</th>
<th>x</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>10</td>
<td>x = 10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>y = 18</td>
</tr>
</tbody>
</table>
public class Ex0509
{
    public static void main (String args[])
    {
        int x = 2;
        while (x < 10)
        {
            if (x % 2 == 0)
                x+=2;
            else
                x++;  
        }
        System.out.println("x = " + x);
    }
}

<table>
<thead>
<tr>
<th>x</th>
<th>x % 2 == 0</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>true</td>
<td>x = 10</td>
</tr>
</tbody>
</table>

**Exercise 10**
public class Ex0510
{
    public static void main (String args[])
    {
        int x = 2;
        do
            { 
                if (x % 2 == 0)
                    x+=2;
                else
                    x++; 
            }
        while (x < 10);
        System.out.println("x = " + x);
    }
}

<table>
<thead>
<tr>
<th>x</th>
<th>x % 2 == 0</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>true</td>
<td>x = 10</td>
</tr>
</tbody>
</table>

Exercise 11
public class Ex0511
{
    public static void main (String args[])
    {
        int x = 10;
        int y = 20;
        do
            { 
                x = y + 2;
                y = x - 2;
            }
        while (x < y);
        System.out.println("x = " + x);
    }
}

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>20</td>
<td>x = 22</td>
</tr>
</tbody>
</table>
**Exercise 11 Continued**

You may be tempted to think that this loop never finishes. It appears that x and y never alter their values from 22 and 20 respectively. This is true until you realize that repetition only continues as long as x is smaller than y. That condition does not last long at all.

**Exercise 12**

```java
class Ex0512 {
    public static void main(String args[]) {
        int x = 10;
        int y = 1;
        do {
            if (x % 2 == 0) {
                x += 5;
            } else {
                y += 2;
            }
        } while (y < x);
        System.out.println("x = " + x);
    }
}
```

<table>
<thead>
<tr>
<th>x % 2 == 0</th>
<th>x</th>
<th>y</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>10</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>false</td>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>false</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>false</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>false</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>false</td>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>false</td>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>false</td>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>false</td>
<td>15</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This is an example where the loop continues for some time. If you observe the pattern you will find that it is not necessary to trace the entire set of repetitions. The value of x reaches 15 and then x is no longer altered. At this stage the value of y is no longer of any consequence, as long as the pattern allows an exit. Regardless of the y value, when the loop exits the value of x will still be 15.
Exercise 13

public class Ex0513
{
    public static void main (String args[]) {
        int x = 1;
        int y = 3;
        int z = 5;
        while (z > x + y) {
            x = y + z;
            y = x + z;
            z = x - y;
        }
        System.out.println("x = " + x);
        System.out.println("y = " + y);
        System.out.println("z = " + z);
    }
}

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>13</td>
<td>-5</td>
<td>x = 8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>y = 13</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>z = -5</td>
</tr>
</tbody>
</table>
This program intentionally repeats ten times. It is a classic example of a repetition that produces cyclical results. Every time that the k loop counter is odd, the x, y, z values are 5, 8 and -3, and whenever k is even the x, y, z values are 5, 2 and 3.
public class Ex0515 {
    public static void main (String args[])
    {
        int x = 168;
        int y = 90;
        int z = 0;
        do
        {
            z = x % y;
            if (z == 0)
                System.out.println("y = " + y);
            else
            {
                x = y;
                y = z;
            }
        }
        while (z != 0);
    }
}

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>168</td>
<td>90</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>90</td>
<td>78</td>
<td>12</td>
<td>y = 6</td>
</tr>
<tr>
<td>78</td>
<td>12</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>6</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
5.11 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 Graphics 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 an simple introduction to animation. Moving graphic images relies heavily on the use of control structures.

Program Java0520.java, in figure 5.22, is a very small program. There appears to be only one call to the drawRect method. Yes, it is only one call, but this single method call is inside a loop structure that repeats fifty times. The result is an interesting set of growing rectangles that all start at the same top-left corner.

If you look at the program example, you will see that the logic is quite straightforward. Each method call to drawRect uses the same (50,50) top-left coordinate value. On the other hand, the value of the width and height of the rectangle is constantly changing. Variable side starts with value 50 and increments by 10 pixels with each repetition. The output display demonstrates that even a modest program can already create an interesting effect.

Do remember that each one of the graphics program examples is an applet. You will need to load each program, compile the program and then load and execute the corresponding .html file. Also remember that html files will appear into the JCreator browse window until you change the file type.
import java.awt.*;
import java.applet.*;

public class Java0520 extends Applet
{
   public void paint(Graphics g)
   {
      for (int side = 50; side < 500; side +=10)
         g.drawRect(50,50,side,side);
   
   
}
You can create a different effect by changing the top-left coordinate values along with the size of the rectangles. **Java0521.java**, in figure 5.23, uses a `k` loop control variable to repeat twenty-five calls to `drawRect`. Inside the `for` loop the values for `x`, `y` and `side` change to create the growing squares.

**Figure 5.23**

```
// Java0521.java
// This program is another example of displaying multiple graphics rectangles using a loop control structure.
import java.awt.*;
import java.applet.*;
public class Java0521 extends Applet
{
    public void paint(Graphics g)
    {
        int x = 375;
        int y = 275;
        int side = 50;
        for (int k = 1; k <= 25; k++)
        {
            g.drawRect(x,y,side,side);
            x -= 10;
            y -= 10;
            side += 20;
            
        }
    }
}
```
Program **Java0522.java**, in figure 5.24, draws a series of lines that connect the top and bottom of a square.

**Figure 5.24**

// Java0522.java
// This program demonstrates how to draw multiple lines easily with a loop structure.

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

public class Java0522 extends Applet
{
    public void paint(Graphics g)
    {
        g.drawRect(50,50,500,500);
        for (int x = 50; x <= 550; x += 10)
            g.drawLine(x,50,600-x,550);
    }
}
Program **Java0522.java** started an interesting pattern. This pattern will be continued by a second loop structure that connects the left and right side of the square with straight lines. **Java0523.java**, in figure 5.25, demonstrates a square filled with straight lines. Each line travels through the center of the square in such a manner that all lines intersect through the same point.

**Figure 5.25**

```java
// Java0523.java
// This program continues the pattern started in Java0522.java to create an interesting pattern.
import java.awt.*;
import java.applet.*;

public class Java0523 extends Applet
{
    public void paint(Graphics g)
    {
        g.drawRect(50,50,500,500);
        for (int x = 50; x <= 550; x += 10)
            g.drawLine(x,50,600-x,550);
        for (int y = 50; y <= 550; y += 10)
            g.drawLine(50,y,550,600-y);
    }
}
```
Some of the most sophisticated computer science skills are used for computer animation. The creation of computer animation is a multi-billion dollar industry. Movies like *Jurassic Park*, *Toy Story* and *The Incredibles*, to name just a few, involve a level of computer science software development and computer hardware power that is at the cutting edge of human computer knowledge.

Program *Java0524.java*, in figure 5.26, introduces a very modest animation program. A circle will move on the screen. First the circle is drawn and then after some delay the circle is erased. This creates an illusion of movement.

**Figure 5.26**

```
// Java0524.java
// This program introduces animation using the "draw and erase" method.

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

public class Java0524 extends Applet
{
    public void paint(Graphics g)
    {
        for (int x = 0; x < 780; x += 5)
        {
            g.setColor(Color.black);
            g.fillOval(x,300,20,20);
        }
    }
```
This graphics section will conclude with a program that simulates a snowman moving across the screen. Program **Java0525.java**, in figure 5.27, uses the same *draw and erase* approach of the previous program to move a snowman.

**Figure 5.27**

// Java0525.java
// This program demonstrates an animated snowman.

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

public class Java0525 extends Applet
{
    public void paint(Graphics g)
    {
        g.setColor(Color.black);
        g.fillRect(0,0,800,600);
        for (int x = 0; x < 780; x += 5)
        {
            g.setColor(Color.white);
            g.fillOval(x,300,20,20) ;
            g.fillOval(x-6,315,32,32);
            g.fillOval(x-15,340,50,50);
            for (long delay = 1; delay < 10000000; delay++);
            g.setColor(Color.black);
            g.fillOval(x,300,20,20) ;
            g.fillOval(x-6,315,32,32);
            g.fillOval(x-15,340,50,50);
        }
    }
}
Many computer age teenagers are hardly impressed by the modest display of a moving circle and a pitiful-looking snowman flickering across the monitor. You were brought up on sophisticated video games and computer special effects. Who knows, this computer course may be the first small step into a future career of computer animation? The reality is that you can only start modestly. In the area of music there are very few Mozarts who wrote symphonies as a small child. Most future professional musicians start awkwardly to play an instrument with few people in attendance as they practice. Computer animation is no different.

5.12 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 is
simple sequence, selection and repetition. 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 not possible to make any decisions nor is it possible to repeat any block of program statements.

Input at from the keyboard is accomplished with a variety of Java statements that will be explained in greater detail in a future chapter. You do need to realize that method `nextLine` enters strings, method `nextInt` enters integers and method `nextDouble` enters double data types.

Selection provides three types of decision control structures. There is one-way selection with `if`, two-way selection with `if..else` and multiple-way selection with `case..break`.

Iteration also provides three types of loop control structures. There is fixed iteration with `for`, pre-condition iteration with `while` and postcondition iteration with `do..while`.

This chapter is not the complete story. Every program example uses only a single condition with selection and iteration. 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 a later chapter.

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. The last two program examples introduced some simple computer animation with the `draw and erase` technique. This is not an impressive-looking animation technique, but it is a start that can be comprehended at this early stage.

**AP Examination Alert**

In this chapter you have learned all the available control structures in Java, but only the following control structures will be tested on the APCS Examination:

```
if    one-way selection
if..else  two-way selection
for    fixed iteration
while   pre-condition iteration
```
These control structures will not be tested:

- `switch..case` multiple-way selection
- `do..while` post-condition iteration