**Chapter V**

**Control Structures**

**Chapter V Topics**

5.1 Introduction

5.2 Types of Control Structures

5.3 Relational Operators

5.4 Keyboard User 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 Control Structures and Graphics

5.11 GridWorld and Control Structures

5.12 Summary

**5.1 Introduction**

The previous chapter focused on using *methods*and *parameters*, which are 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 programming 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 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 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.

**5.2 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**



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***

**Program Statement**

**Program Statement**

**Program Statement**

**Program Statement**

**Condition**

***True***

***False***



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***

**Program Statement**

**Program Statement**

**Program Statement**

**Condition**

***True***

***False***

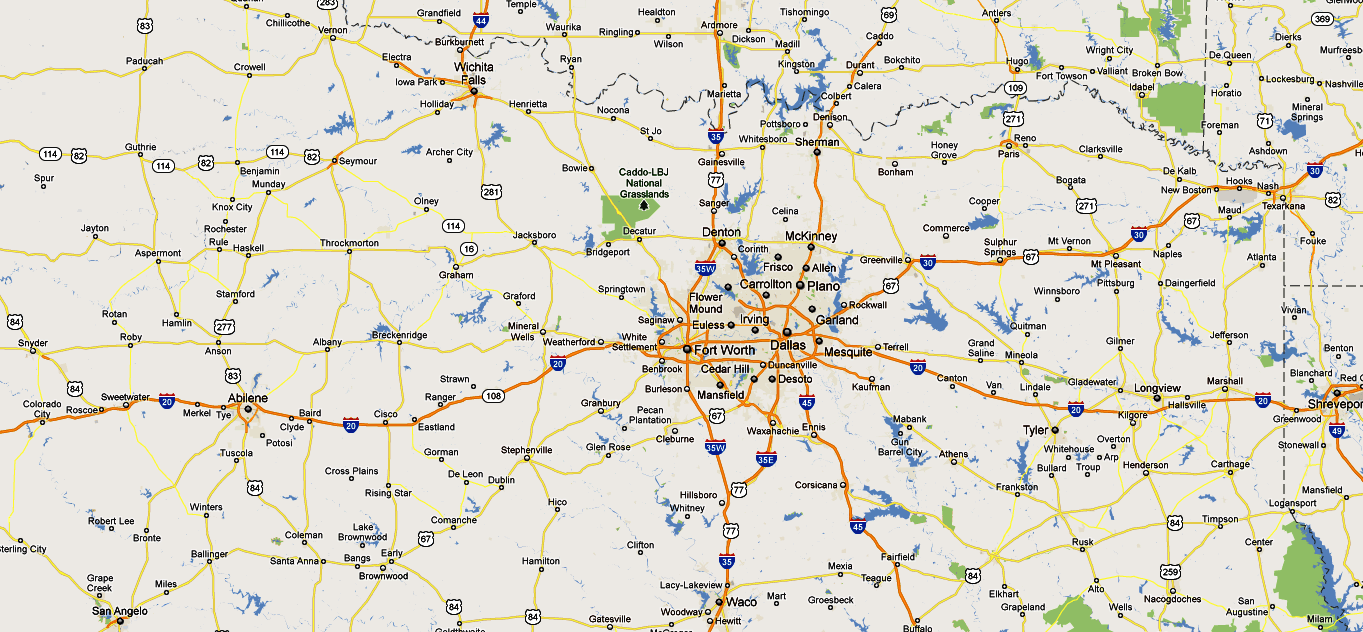
**Program Statement**

**Program Statement**

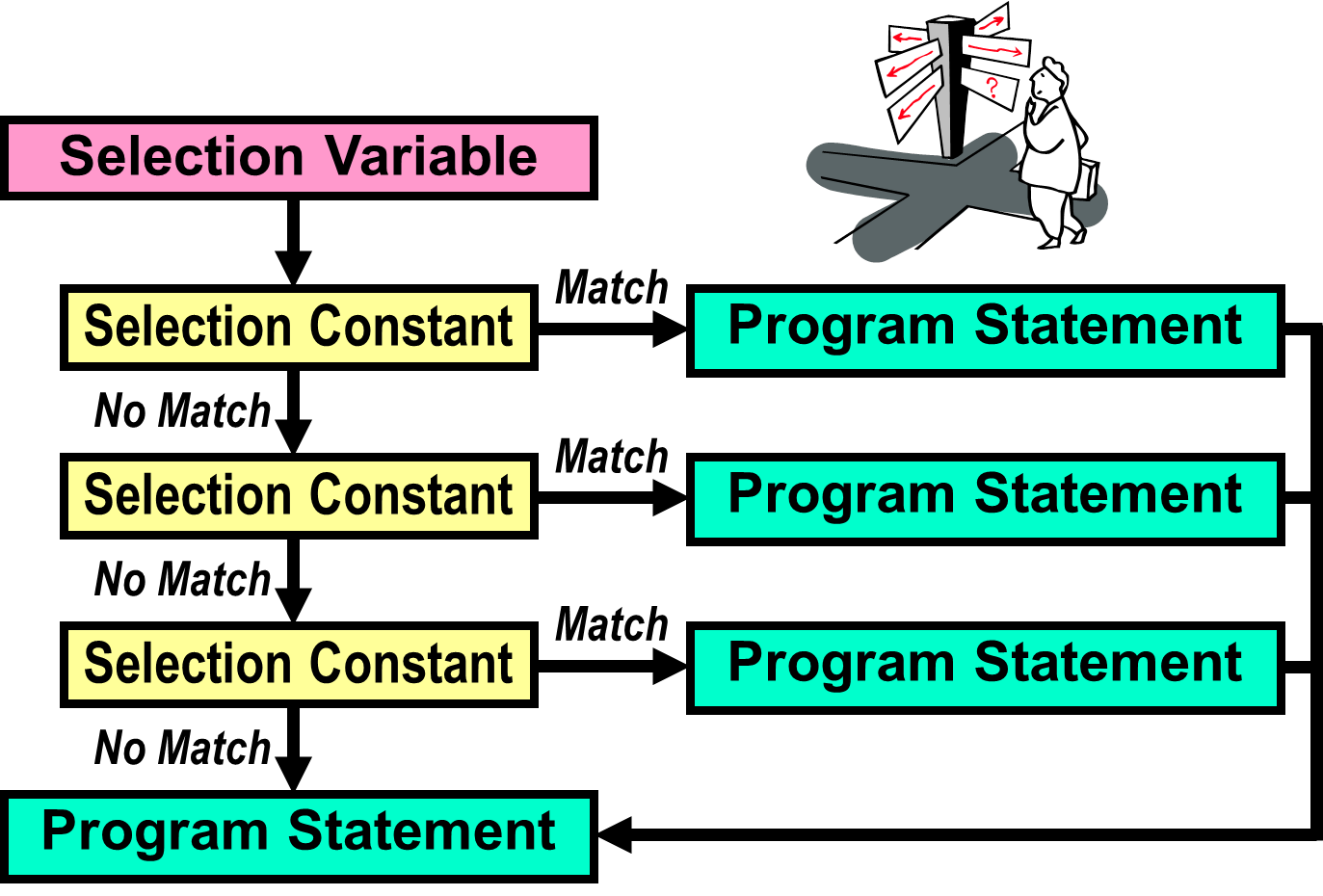


The *two-way*selection control structure also checks to see if some special condition is **true**. But there is a significant difference in how the program flow is handled. With *one-way* selection, a **true** condition means to do something, like get off the road and get gas, before continuing. The *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 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.

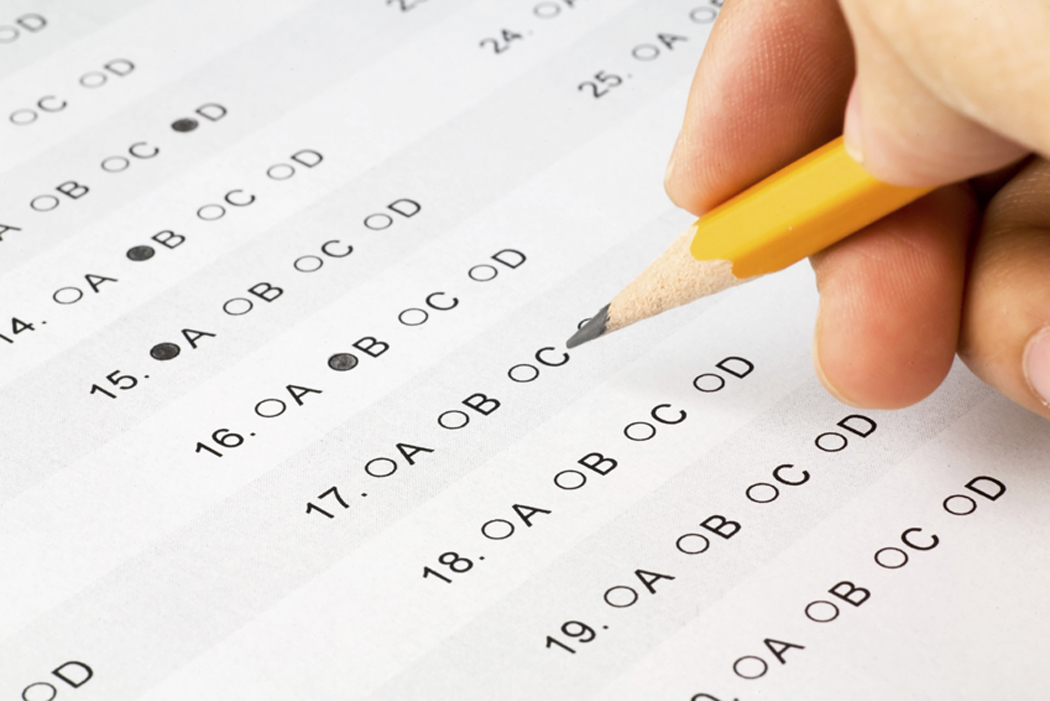


***Multiple-Way Selection***



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.

**Program Statement**

**Program Statement**

**Condition**

***True***

***False***

**Program Statement**

**Program Statement**

Description: C:\Users\JohnSchram\AppData\Local\Microsoft\Windows\Temporary Internet Files\Content.IE5\1NSA1V5K\MC900331509[1].wmf 

**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*

**Java Relational Operators**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Operator** | **Expression** | **Evaluates** |
| Equals | **==** | 5 == 5  5 == 10 | true  false |
| Not equals | **!=** | 50 != 25  100 != 100 | true  false |
| Less than | **<** | 100 < 200  200 < 100 | true  false |
| Greater than | **>** | 200 > 100  200 > 200 | true  false |
| Less than  or equals | **<=** | 100 <= 200  200 <= 200  200 <= 100 | true  true  false |
| Greater than  or equals | **>=** | 100 >= 200  200 >= 200  200 >= 100 | false  true  true |

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 User 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, 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.

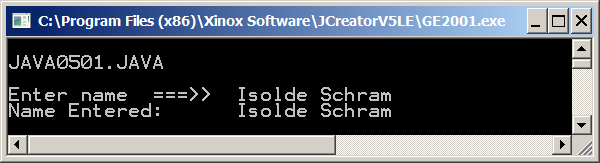
I propose a good solution to this problem. We can start using keyboard input right now, and you will learn what Java features are necessary to accommodate program input. At the same time do not expect any explanation on these features. Basically, you are told *use this, it works and do not bother ask 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.Scanner; // Line 1  public class Java0501  {  public static void main (String args[])  {  System.out.println("\nJAVA0501.JAVA\n");  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.

|  |
| --- |
| import java.util.Scanner; // Line 1 |

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.

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

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 keyboard data entry.

|  |
| --- |
| 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.

|  |
| --- |
| String name = input.nextLine(); // Line 4 |

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.

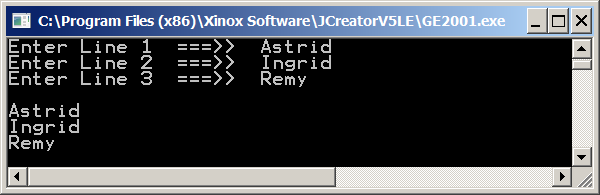
Program **Java0502.java**, in figure 5.2, demonstrates how to write a program with multiple lines of input entered from the keyboard during program execution. In particular, pay close attention to the special statements necessary for keyboard input. Do you see that they are identical to the previous 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.

**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("\nJava0502.JAVA\n");  Scanner input = new Scanner(System.in);  System.out.print("Enter Line 1 ===>> ");  String input1 = input.nextLine();  System.out.print("Enter Line 2 ===>> ");  String input2 = input.nextLine();  System.out.print("Enter Line 3 ===>> ");  String input3 = input.nextLine();  System.out.println();  System.out.println(input1);  System.out.println(input2);  System.out.println(input3);  System.out.println("\n\n");  }  } |

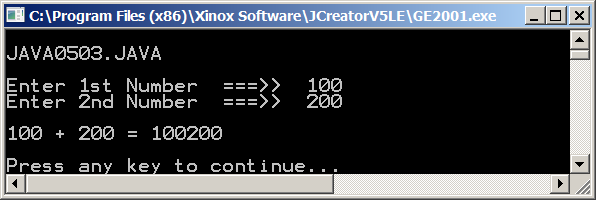
**Figure 5.2 Continued**



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**

|  |
| --- |
| // 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("\nJava0503.JAVA\n");  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);  System.out.println("\n\n");  }  } |

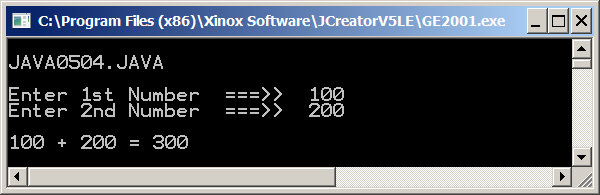


Program **Java0503.java**, in figure 5.3, provides ample proof that *string input it is*. Two perfectly good numbers were entered and the addition of the two numbers resulted in **concatenation** with output **100200**. Arithmetic numbers they are **not**.

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**

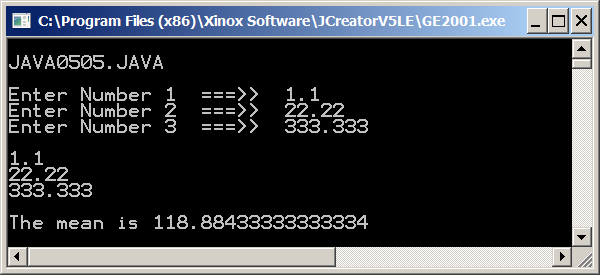
|  |
| --- |
| // Java0504.java  // This program uses the <nextInt> method to enter integers from the keyboard.  // It is now possible to correctly add the two numbers.  import java.util.Scanner;  public class Java0504  {  public static void main (String args[])  {  System.out.println("\nJava0504.J AVA\n");  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\n");  }  } |



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,  // which are used to display the mean.  import java.util.Scanner;  public class Java0505  {  public static void main (String args[])  {  System.out.println("\nJAVA0505.JAVA\n");  Scanner input = new Scanner(System.in);  System.out.print("Enter Number 1 ===>> ");  double n1 = input.nextDouble();  System.out.print("Enter Number 2 ===>> ");  double n2 = input.nextDouble();  System.out.print("Enter Number 3 ===>> ");  double n3 = input.nextDouble();  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("\n\n");  }  } |



|  |
| --- |
| **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. |

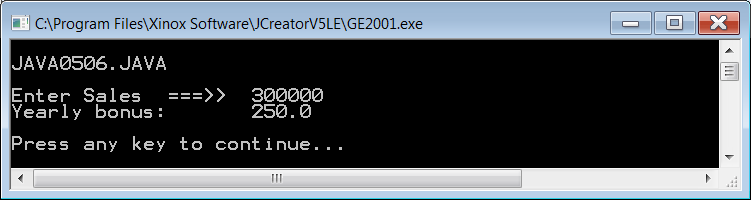
**5.5 One-Way Selection**

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.

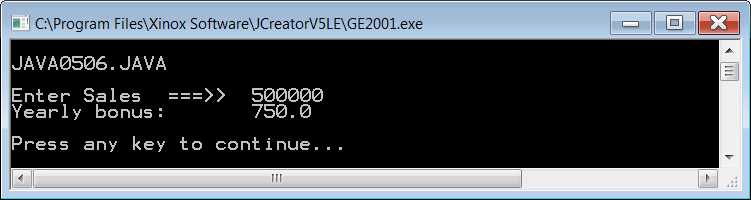
Program **Java0506.java**, in figure 5.6,computes the year-end bonus for employees of some imaginary company. Every employee automatically gets a $250.00 bonus. However, this company wants to reward its top sales people. The bonus is increased by an additional $500.00 if employee sales are equal to or 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 for data entry at the keyboard. The previous programs used the **input** identifier. It is not required to use **input**. Any identifier is fine with Java. However, you should use some name that makes sense, such as *input* or *keyboard*.

**Figure 5.6**

|  |
| --- |
| // 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("\nJAVA0506.JAVA\n");  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);  System.out.println();  }  } |



**Figure 5.6 Continued**



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: **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.

**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.

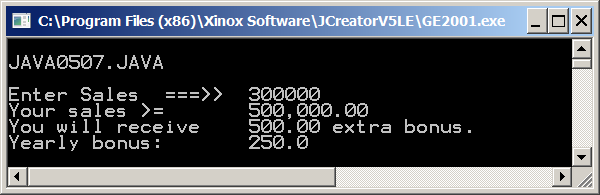
Using indentations with control structures helps to identify the program statement or statements that will only be executed if the condition is true. You have seen the use of indentation in the main method. You can consider that the braces of the methods control the simple sequence of the program statements.

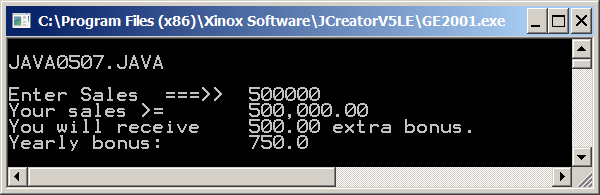
|  |
| --- |
| **Indentation Rule** |
| Java syntax uses free-form program style. Program statements may be placed on multiple lines with or without indentation.  By convention, control structures and their conditional statements are placed on one line. The program statement  that is executed, if the 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, however 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**

|  |
| --- |
| // 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("\nJAVA0507.JAVA\n");  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();  }  } |

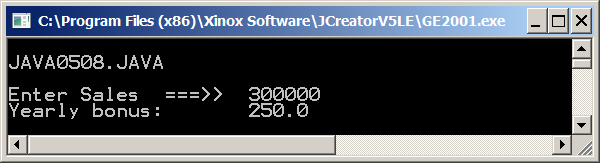


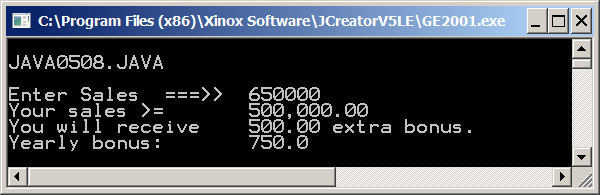


The “multiple statements” dilemma is solved with **block structure**. You have been using block structure for some time, and you never realized it. With every program's main method you used braces **{ }.** Now look at program **Java0508.java**, in figure 5.8 and notice how the braces create a block that 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("\nJAVA0508.JAVA\n");  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();  }  } |





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.

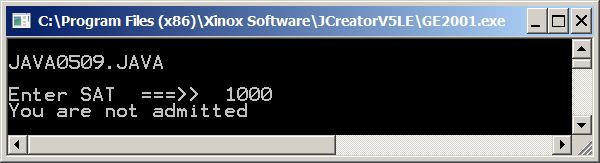
|  |
| --- |
| **One-Way Selection** |
| **General Syntax:**  if (condition true)  cexecute program statement  **Specific Examples:**  **if (counter > 100)**  **System.out.println("Counter exceeds 100");**  Use braces { } and block structure to control multiple  program statements.  **if (savings >= 10000)**  **{**  **System.out.println("It’s skiing time");**  **System.out.println("Let’s pack");**  **System.out.println("Remember your skis");**  **}** |

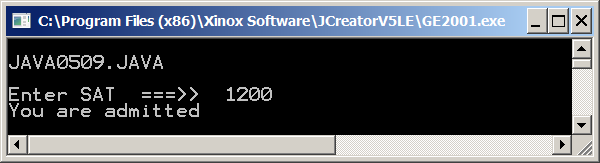
**5.5 Two-Way Selection**

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 **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("\nJAVA0509.JAVA\n");  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();**  }  } |

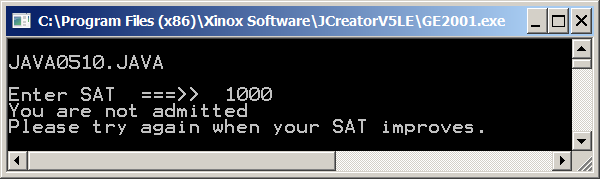


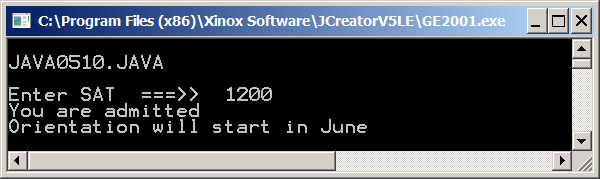


Program **Java0510.java**,in figure 5.10, demonstrates block structure with *two-way*selection. The same type of SAT program is shown with a few statements added to help demonstrate the block structure. Pay attention to the style that is used with the block structure and the **if ... else**. You will notice different styles in different books. This is fine, as long as the style is consistent.

**Figure 5.10**

|  |
| --- |
| // 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[])  {  System.out.println("\nJAVA0510.JAVA\n");  Scanner keyboard = new Scanner(System.in);  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.");**  **}**  System.out.println();  }  } |





|  |
| --- |
| **Two-Way Selection** |
| **General Syntax:**  if (condition true)  execute first program statement  else **// when condition is false**  execute second program statement  **Specific Example:**  **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-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 asks the user to enter an integer between **1** and **12**. Each number is associated with a particular month. 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 **Java0511.java**, in figure 5.11, and observe two new Java keywords, **switch**, and **case**.

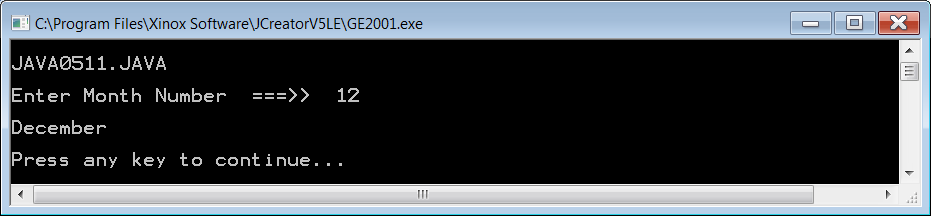
**Figure 5.11**

|  |
| --- |
| // 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("\nJAVA0511.JAVA\n");  Scanner keyboard = new Scanner(System.in);  System.out.print("Enter Month Number ===>> ");  int month = keyboard.nextInt();  System.out.println();  switch (month)  {  case 1 : System.out.println("January");  case 2 : System.out.println("February");  case 3 : System.out.println("March");  case 4 : System.out.println("April");  case 5 : System.out.println("May");  case 6 : System.out.println("June");  case 7 : System.out.println("July");  case 8 : System.out.println("August");  case 9 : System.out.println("September");  case 10 : System.out.println("October");  case 11 : System.out.println("November");  case 12 : System.out.println("December");  }  System.out.println();  }  } |





**Figure 5.11 Continued**



The **switch** structure in the **Java0511.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 **month** is an **int** and it is also the *selection variable* that determines the matching process. Note that **month** 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.

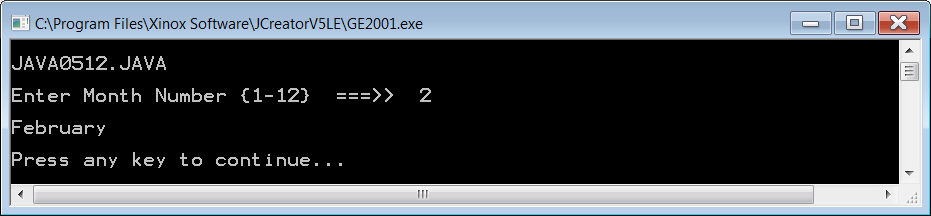
How about the program output in figure 5.11, does it make sense? Please do realize that nothing is wrong. The **switch** statement behaves exactly the way that it was designed for Java. Be aware of that and make sure that you handle your programs accordingly. Program execution will compare each **case** 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.

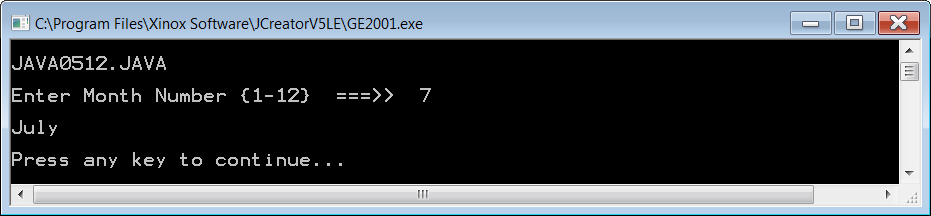
Program **Java0512.java** cures the problem of the previous **switch** example. You will notice a new Java keyword has been added to every **case** statement. It is **break**, and it is placed between every **case** statement. **break** exits the current program block, which in this case means that program execution jumps to the end of the **main** method. 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 **Java0512.java**, shown in figure 5.12, has the same twelve 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 not need a **break** statement since it is already at the end of the **switch** structure. The output examples are now logically correct.

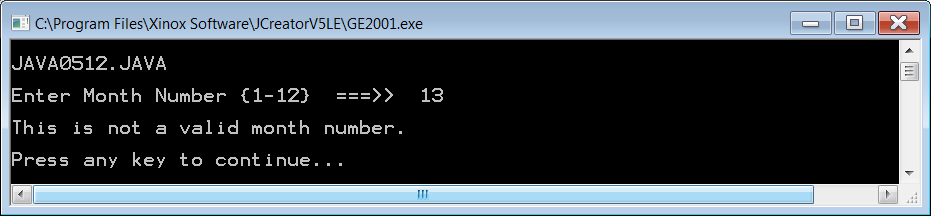
**Figure 5.12**

|  |
| --- |
| // 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("\nJAVA0512.JAVA\n");  Scanner keyboard = new Scanner(System.in);  System.out.print("Enter Month Number {1-12} ===>> ");  int month = keyboard.nextInt();  System.out.println();  switch (month)  {  case 1 : System.out.println("January"); **break;**  case 2 : System.out.println("February"); **break;**  case 3 : System.out.println("March"); **break;**  case 4 : System.out.println("April"); **break;**  case 5 : System.out.println("May"); **break;**  case 6 : System.out.println("June"); **break;**  case 7 : System.out.println("July"); **break;**  case 8 : System.out.println("August"); **break;**  case 9 : System.out.println("September"); **break;**  case 10 : System.out.println("October"); **break;**  case 11 : System.out.println("November"); **break;**  case 12 : System.out.println("December"); **break;**  **default : System.out.println("This is not a valid month number.");**  }  System.out.println();  }  } |





**Figure 5.12 Continued**



Earlier in this *Multi-Way Selection* section it was stated that **switch** works with the **int**, **char** and **String** data types. However, this assumes your JDK (Java Development Kit) is version 7.0 or newer. If you have an older version you will not be able to use **switch** with **String**.

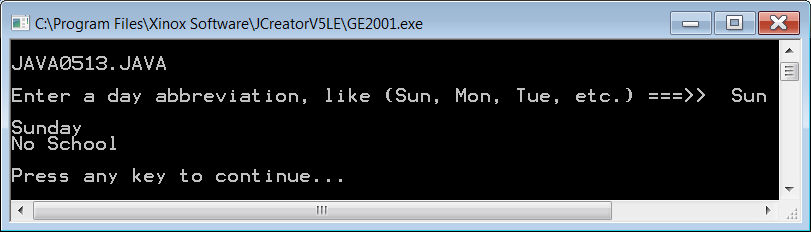
|  |
| --- |
| **Java 7.0 Warning** |
| The next 2 program examples will only compile and execute  if your JDK is version 7.0 or newer. |

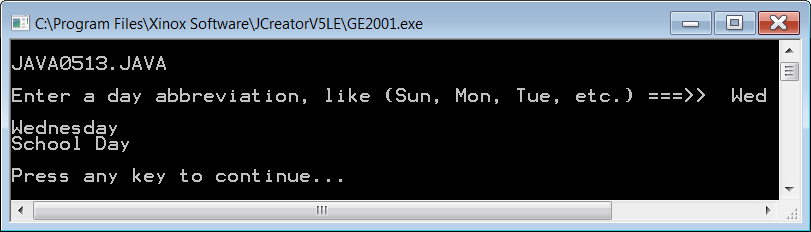
Program **Java0513.java**, shown in figure 5.13, demonstrates using **switch** with a **String** selection variable. It also demonstrates one way that **switch** is different from all other control structures. With any other control structure, if you want to control multiple lines of code, you need to put them inside { braces }. With **switch**, controlling multiple lines is done in the same way as controlling a single line because you can put as many programming statements as you wish between the **case** and **break** statements.

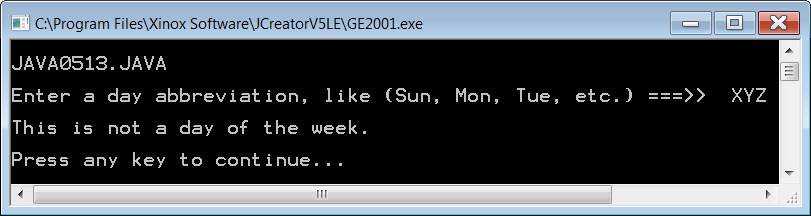
**Figure 5.13**

|  |
| --- |
| // Java0513.java  // This program demonstrates <switch> used with the <String> data type.  // This requires the uses of the Java 7.0 or later.  // It also shows that multiple program statements can be placed  // between the <case> and <break> commands.  // This is the one time {braces} are NOT used to control multiple statements.  import java.util.Scanner;  public class Java0513  {  public static void main (String args[])  {  System.out.println("\nJAVA0513.JAVA\n");  Scanner input = new Scanner(System.in);  System.out.print("Enter a day abbreviation, like (Sun, Mon, Tue, etc.) ===>> ");  String day = input.nextLine();  System.out.println();  switch (day)  {  case "Sun" :  System.out.println("Sunday");  System.out.println("No School");  break;  case "Mon" :  System.out.println("Monday");  System.out.println("School Day");  break;  case "Tue" :  System.out.println("Tuesday");  System.out.println("School Day");  break;  case "Wed" :  System.out.println("Wednesday");  System.out.println("School Day");  break;  case "Thu" :  System.out.println("Thursday");  System.out.println("School Day");  break;  case "Fri" :  System.out.println("Friday");  System.out.println("School Day");  break;  case "Sat" :  System.out.println("Saturday");  System.out.println("No School");  break;  default : System.out.println("This is not a day of the week.");  }  System.out.println();  }  } |

**Figure 5.13 Continued**



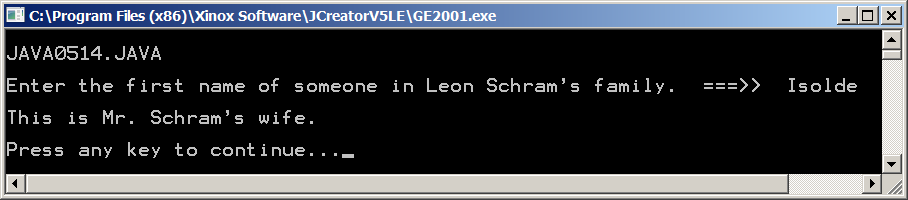




Program **Java0514.java**, shown in figure 5.14, shows a more complex program which uses a **String** selection variable with **switch**. In this program the user can enter the first name if anyone in Leon Schram’s family and it will display how that person is related to him. Consider this. Leon Schram has five grandsons: Anthony, Alec, Maddox, Jaxon and Braxton. In the program you will notice that there are five *cases*, one for each grandson; however, there is only one **println** statement that displays *This is one of Leon Schram’s grandsons*. This demonstrates another feature of **switch**. Multiple cases can yield the same result.

**Figure 5.14**

|  |
| --- |
| // Java0514.java  // This <String> example shows a more complex use of the <switch> structure,  // which can handle multiple matches for the same output.  import java.util.Scanner;  public class Java0514  {  public static void main (String args[])  {  System.out.println("\nJAVA0514.JAVA\n");  Scanner input = new Scanner(System.in);  System.out.print("Enter the first name of someone in Leon Schram's family. ===>> ");  String firstName = input.nextLine();  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 "Mari" :  case "Brenda" : 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 "Astrid" :  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.");  }  System.out.println();  }  } |



|  |
| --- |
| **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. |

**5.7 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 **Java0515.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 quickly 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 displays the same thing? You see the point here. There must be a better way. Program **Java0515.java**, in figure 5.15 demonstrates the inefficient repetition without using any loop structures. Please do not ever use this approach.

**Figure 5.15**

|  |
| --- |
| // Java0515.java  // This program displays 40 identical lines very inefficiently  // with 40 separate println statements.  public class Java0515  {  public static void main(String args[])  {  System.out.println("\nJAVA0515.JAVA\n");  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\n");  }  } |

**Figure 5.15 Continued**



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 **Java0516.java**, in figure 5.16, is considerably shorter than the previous program example.

**Figure 5.16**

|  |
| --- |
| // Java0516.java  // This program displays 40 identical lines efficiently  // with one println statement and a loop structure.  public class Java0516  {  public static void main(String args[])  {  System.out.println("\nJAVA0516.JAVA\n");  int k;  **for (k = 1; k <= 40; k++)**  **System.out.println("Eat at Joe's friendly diner for the best lunch value");**  System.out.println();  }  } |

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 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.

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

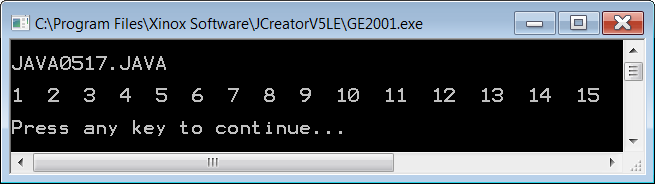
**Part 2** states the *Loop Exit Condition*. As long as the value of **k** is less than or equal to **40**, the program statement following the **for** parenthesis will be executed.

**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 5.17, demonstrates that the LCVcan 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 5.17

|  |
| --- |
| // Java0517.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 Java0517  {  public static void main(String args[])  {  System.out.println("\nJAVA0517.JAVA\n");  **for (int k = 1; k <= 15; k++)**  **System.out.print(k + " ");**  System.out.println();  }  } |

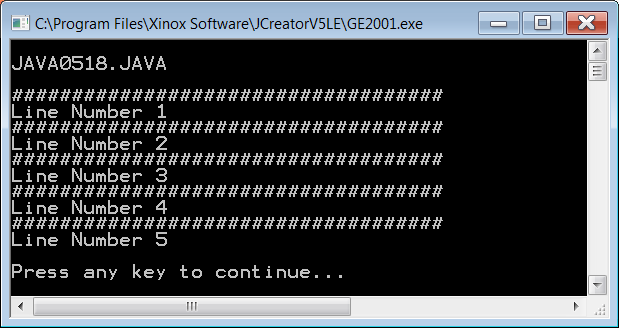


Program **Java0518.java**,in figure 5.18, is a slight variation on the previous program. This program demonstrates how to execute multiple statements in a loop control structure. Yes, you guessed it, your good friend *block structure* with braces **{ }** is back to save the day. Try this program out and comment out the braces to see the result.

**Figure 5.18**

|  |
| --- |
| // Java0518.java  // This program demonstrates how to use block structure  // with a <for> loop control structure.  public class Java0518  {  public static void main(String args[])  {  System.out.println("\nJAVA0518.JAVA\n");  **for (int k = 1; k <= 5; k++)**  **{**  **System.out.println("####################################");**  **System.out.println("Line Number " + k);**  **}**  System.out.println();  }  } |

**Figure 5.18 Continued**



You may get the impression that the **for** loop uses a LCV that always increments by **1**. This is not always true. Program **Java0519.java**,in figure 5.19, 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**.

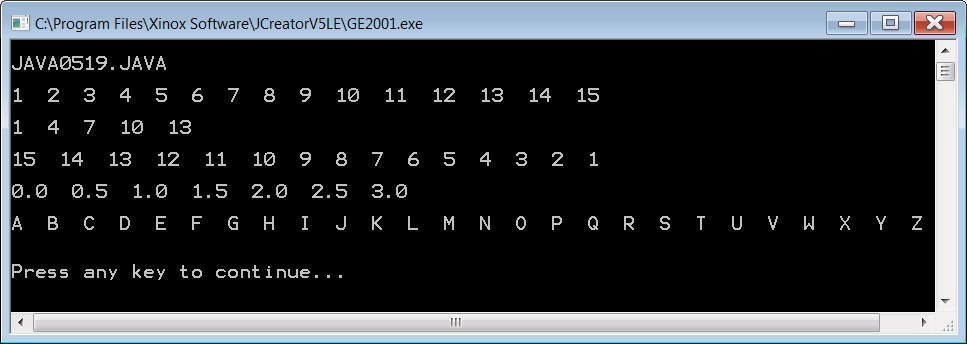
The third loop uses **r--** and this loop does not increment at all; it decrements the LCV by **1** each time. This control structure displays the numbers from **15** to **1.**

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 5.19**

|  |
| --- |
| // Java0519.java  // This program displays various counting schemes.  // It also demonstrates the versatility of the <for> loop.  public class Java0519  {  public static void main(String args[])  {  System.out.println("\nJAVA0519.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\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( );**  **}** |

**5.8 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 left. I think you get the picture. Computer life is full of situations with unknown repetitions. Java is well prepared for the task with two conditional loop structures: The precondition **while** loop and the postcondition **do...while** loop.

|  |
| --- |
| **Note to Teachers and Students with Advanced Knowledge** |
| It is possible to treat the **for** loop structure like a conditional loop that is not fixed. In fact, a **for** loop can be designed to behave exactly like a **while** loop.  It is my intention to use and treat a **for** looplike a *fixed* iteration  loop and use the **while** loop and **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, do not worry.  Ignore this little summary box, and move on. |

**The Precondition *while* Loop**

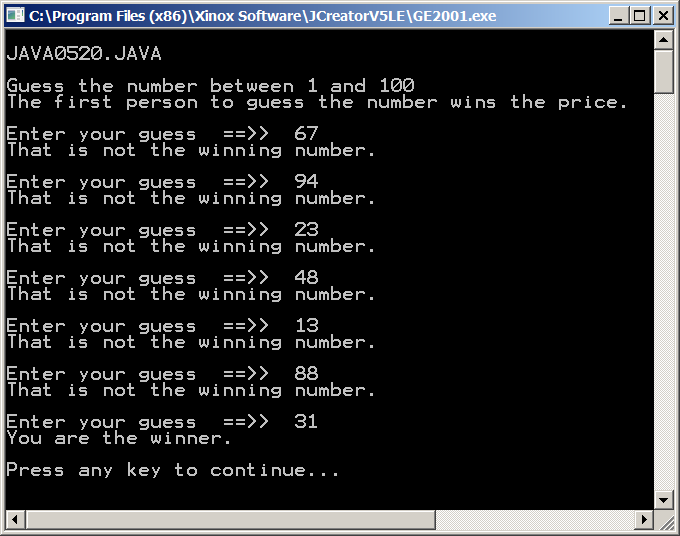
Program **Java0520.java**,in figure 5.20,introduces the **while** loop. This loop is called a *precondition*loop. The condition, which determines when the loop exits, is tested before the loop body is started. Like earlier control structures, you will note that the condition is placed inside parentheses.

The **while** loop structure requires a statement inside the loop body that was not necessary with the **for** loop. The purpose of this statement is to give an opportunity for the loop control variable to change. The **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 **while** loop. Only the *loop condition* is checked.

**Figure 5.20**

|  |
| --- |
| // Java0520.java  // This program demonstrates the precondition <while> loop.  // This loop will continue until the winning number is entered.  // This loop does not repeat in a fixed number of times.  import java.util.Scanner;  public class Java0520  {  public static void main(String args[])  {  System.out.println("\nJAVA0520.JAVA\n");  Scanner input = new Scanner(System.in);  System.out.println("Guess the number between 1 and 100");  System.out.println("The first person to guess the number wins the price.");  System.out.println();  int guess = 0;  **while (guess != 31)**  **{**  **System.out.print("Enter your guess ==>> ");**  **guess = input.nextInt();**  **if (guess == 31)**  **System.out.println("You are the winner.");**  **else**  **System.out.println("That is not the winning number.");**  **System.out.println();**  **}**  }  } |

**Figure 5.20 Continued**



|  |
| --- |
| **Pre-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:**  **int pin = 0;** // initialize condition variable  **while(pin != 5678)**  **{**  **System.out.print("Enter pin. ===>> ");**  **pin = input.nextInt();** // 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 5.21 and 5.22, in your brain.

**Figure 5.21**

|  |  |
| --- | --- |
| **Program Segment NoNo #1** | **Program Segment YesYes #1** |
| **int x = 0;**  **while(x < 10)**  **System.out.println(x);**  The loop condition variable, **x**, never changes.  The loop will not exit. | **int x = 0;**  **while(x < 10)**  **{**  **x++;**  **System.out.println(x);**  **}**  The loop condition variable, **x**, changes.  The loop exits when **x** reaches 10. |

**Figure 5.22**

|  |  |
| --- | --- |
| **Program Segment NoNo #2** | **Program Segment YesYes #2** |
| **int x;**  **while(x < 10)**  **{**  **x++;**  **System.out.println(x);**  **}**  The loop condition variable, **x**, is never initialized.  This program will not compile. | **int x = 0;**  **while(x < 10)**  **{**  **x++;**  **System.out.println(x);**  **}**  The loop condition variable, **x**,  is initialized. The program will compile and execute correctly. |

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

Program **Java0521.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 **Java0521.java**,in figure 5.23, checks to see if the PIN(Personal Identification Number) at an ATM(Automatic Teller Machine) is entered correctly.

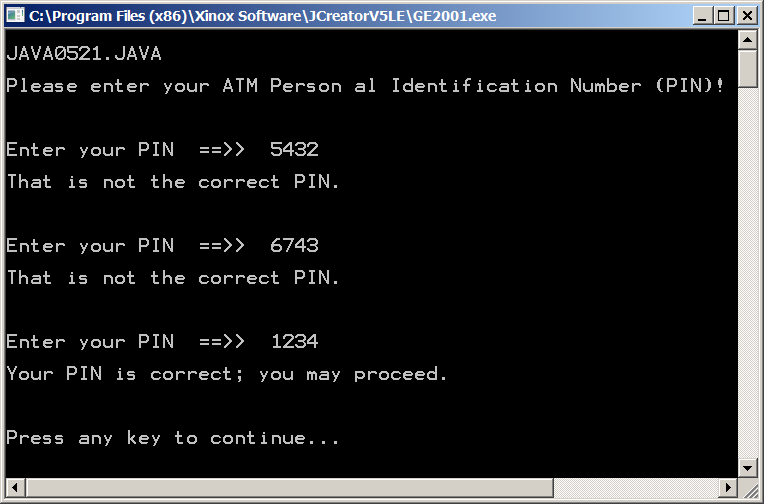
The conditional loop in this program allows an unending number of tries to enter the correct **PIN** number. This is not realistic in the real world. After three or four attempts the card would be confiscated and the account locked.

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. The later chapter will also handle the issue of entering too many guesses.

**Figure 5.23**

|  |
| --- |
| // Java0521.java  // This program demonstrates the postcondition <do..while> loop.  // This loop structure guarantees at least one repetition of  // the loop body. Like the <while> loop this is not a  // "fixed iteration" loop.    import java.util.Scanner;  public class Java0521  {  public static void main(String args[])  {  System.out.println("\nJAVA0521.JAVA\n");  Scanner input = new Scanner(System.in);  System.out.println("Please enter your ATM Person al Identification Number (PIN)!");  System.out.println("\n\n");  int PIN = 0;  **do**  **{**  **System.out.print("Enter your PIN ==>> ");**  **PIN = input.nextInt();**  **System.out.println();**  **if (PIN == 1234)**  **System.out.println("Your PIN is correct; you may proceed.");**  **else**  **System.out.println("That is not the correct PIN.");**  **System.out.println("\n\n");**  **}**  **while (PIN != 1234);**  }  } |

**Figure 5.23 Continued**



|  |
| --- |
| **Post-Conditional Repetition** |
| **General Syntax:**  initialize condition variable before the do..while loop  do  {  loop body  alter condition variable in the loop body  }  while(condition is true)  **Specific Example:**  **int pin = 0;** // initialize condition variable  **do**  **{**  **System.out.print("Enter pin. ===>> ");**  **pin = input.nextInt();** // alter condition variable  **}**  **while(pin != 5678);**  // Not a heading, Semicolon is required.  **System.out.println("Welcome.");** |
| **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 pre-conditional repetition is **while**.  The commands for post-conditional repetition are **do..while**. |

|  |
| --- |
| **AP Examination Alert for Selection Control Constructures** |
| The one-way selection **if** is tested on the AP Exam.  if (sales >= 500000)  bonus = bonus + 5000.0;  The two-way selection **if..else** is tested on the AP Exam.  if (sat >= 1200)  System.out.println("You're admitted");  else  System.out.println("You're not admitted");  The multi-way selection **switch..case..break** is not tested  on the AP Exam.  switch (grade)  {  case 'A' : gpaPoints = 4; break;  case 'B' : gpaPoints = 3; break;  case 'C' : gpaPoints = 2; break;  case 'D' : gpaPoints = 1; break;  case 'F' : gpaPoints = 0;  } |

|  |
| --- |
| **AP Examination Alert for Repetition Control Constructures** |
| The fixed-repetition **for** loop is tested on the AP Exam.  for (int k = 1; k <= max; k++)  sum += k;  The pre-condition **while** loop is tested on the AP Exam.  while (k < max)  {  sum += k;  k++;  }  The post-condition **do..while** loop is not tested on the AP Exam.  do  {  sum += k;  k++;  }  while (k < max); |

**5.10 Control Structures and Graphics**

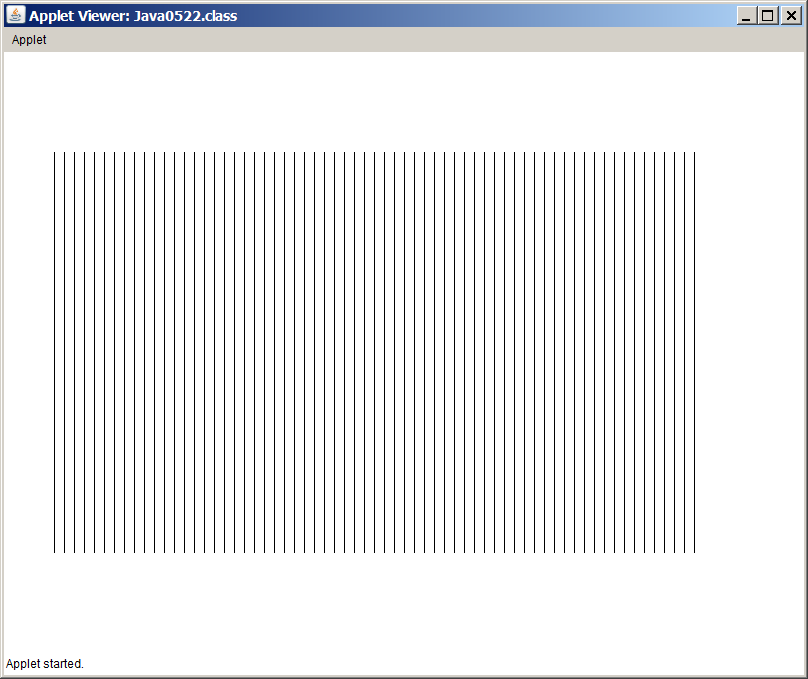
Java has many very nice graphics features. You learned some of the graphics capabilities in the last chapter. In the last section of this chapter you will investigate some more graphics program capabilities. However, there is a major difference. In the last chapter you learned how to use many methods that were part of the **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 a simple introduction to animation. Moving graphic images relies heavily on the use of control structures.

Program **Java0522.java**, in figure 5.24, 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 5.24**

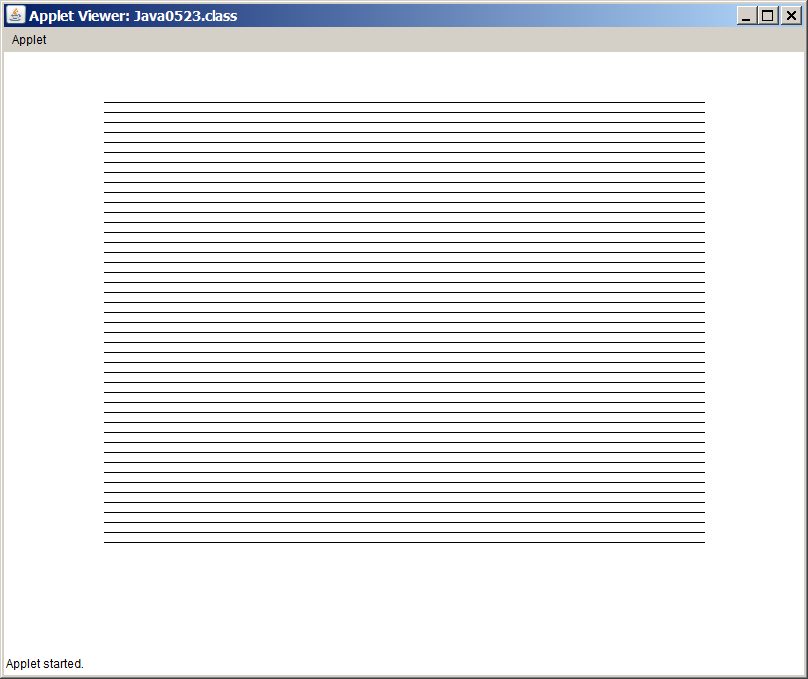
|  |
| --- |
| // Java0522.java  // This program shows how a control structure can be used with graphics.  // This program draws vertical lines, because x1 and x2 have the same value.  import java.awt.\*;  import java.applet.\*;  public class Java0522 extends Applet  {  public void paint(Graphics g)  {  int y1 = 100;  int y2 = 500;  for (int x = 50; x < 700; x +=10)  g.drawLine(x,y1,x,y2);  }  } |



Program **Java 0523.java**, in figure 5.25, makes a few small changes to the previous program to draw horizontal lines. This time **x1** and **x2** are fixed values.

**Figure 5.25**

|  |
| --- |
| // Java0523.java  // This program shows how a control structure can be used with graphics.  // This program draws horizontal lines, because y1 and y2 have the same value.  import java.awt.\*;  import java.applet.\*;  public class Java0523 extends Applet  {  public void paint(Graphics g)  {  int x1 = 100;  int x2 = 700;  for (int y = 50; y < 500; y +=10)  g.drawLine(x1,y,x2,y);  }  } |



In the graphics department where control structures are combined with graphics commands, horizontal lines and vertical lines are pretty comfortable. Some steps higher on the complexity ladder are programs with diagonal lines. Diagonal lines will require that both the **x** values and the **y** values are altered.

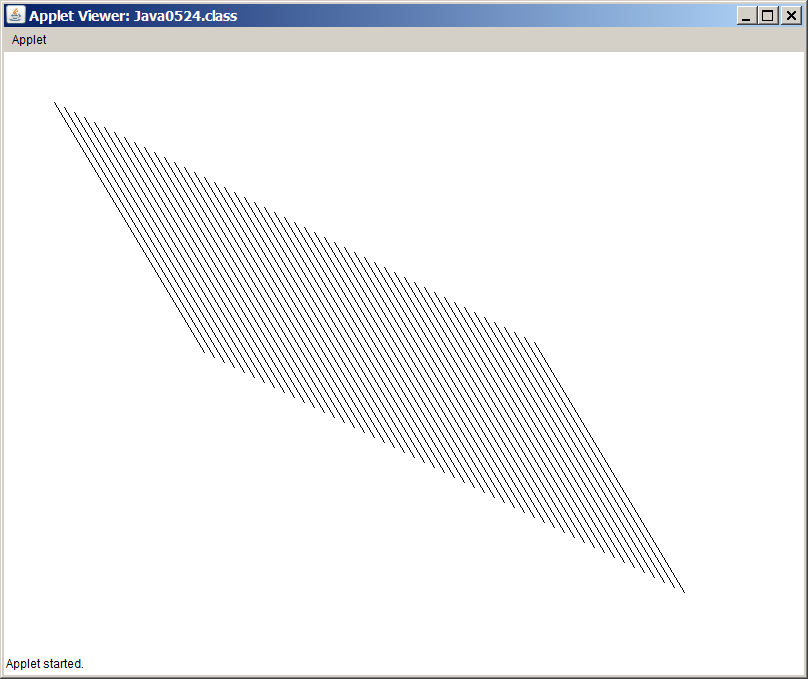
Program **Java0524.java**, in figure 5.26, shows a different approach from the two previous programs. In this case the **LCV** (Loop Control Variable) is strictly used to determine how many lines will be drawn. Previously the LCV value was also used to alter the **x** values or the **y** values.

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

**Figure 5.26**

|  |
| --- |
| // Java0524.java  // This program shows how a control structure can be used with graphics.  // This program draws diagonal lines, because x1, y1, y2, y2, all four change.  import java.awt.\*;  import java.applet.\*;  public class Java0524 extends Applet  {  public void paint(Graphics g)  {  int x1 = 50;  int x2 = 200;  int y1 = 50;  int y2 = 300;  for (int k = 1; k < 50; k++)  {  g.drawLine(x1,y1,x2,y2);  x1 += 10;  x2 += 10;  y1 += 5;  y2 += 5;  }  }  } |

**Figure 5.26 Continued**

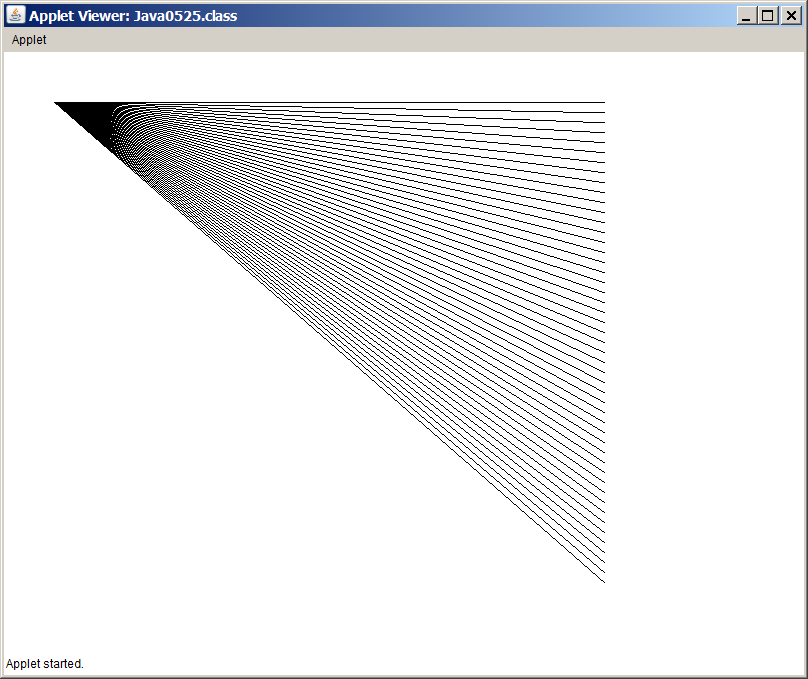


Program **Java0525.java**, in figure 5.27, plays another twist with straight lines. This time it is not a set of **x** values or a set of **y** values that change, but one coordinate point changes with each repetition.

**Figure 5.27**

|  |
| --- |
| // Java0525.java  // This program demonstrates how to rotate a line around a point.  // In this case the (x1,y1) coordinate stays fixed and the (x2,y2) point changes.  import java.awt.\*;  import java.applet.\*;  public class Java0525 extends Applet  {  public void paint(Graphics g)  {  int x1 = 50;  int y1 = 50;  int x2 = 600;  int y2 = 50;  for (int k = 1; k < 50; k++)  {  g.drawLine(x1,y1,x2,y2);;  y2 += 10;  }  }  } |

**Figure 5.27 Continued**

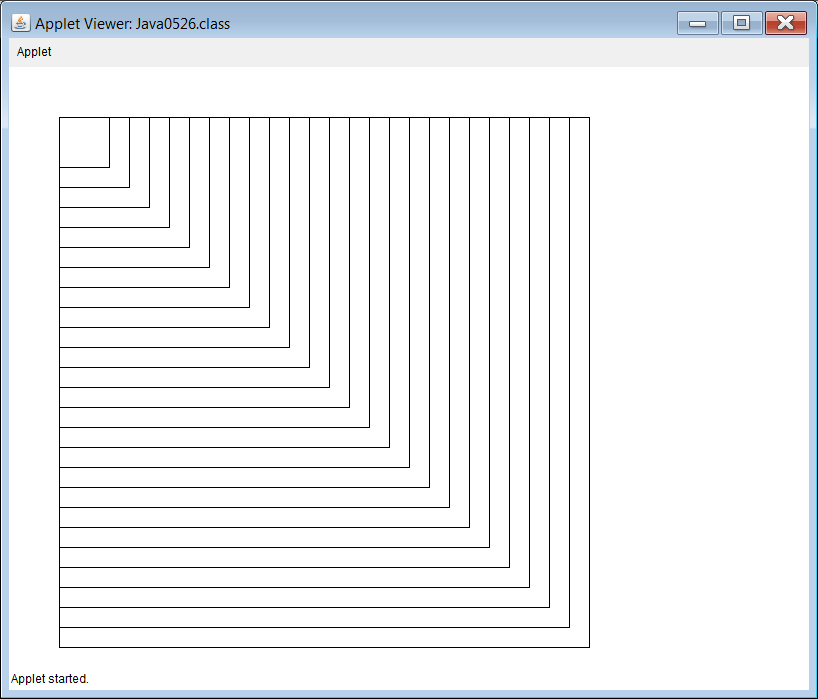


Program **Java0526.java**, in figure 5.28, is a very small program. If you look at the program example, you will see that the logic is quite straight forward. Each method call to **drawRect** uses the same (50,50) top-left coordinate value. The value of the width and height of the rectangle is constantly changing. Variable **side** starts with value **50** and increments by 10 pixels with each repetition. The output display demonstrates that a modest program can be interesting.

**Figure 5.28**

|  |
| --- |
| // Java0526.java  // This program is example of displaying multiple graphics rectangles  // using a loop control structure.  // Note how all rectangle share the same top-left corner.  import java.awt.\*;  import java.applet.\*;  public class Java0526 extends Applet  {  public void paint(Graphics g)  {  for (int side = 50; side < 500; side +=10)  g.drawRect(50,50,side,side);  }  } |

**Figure 5.28 Continued**

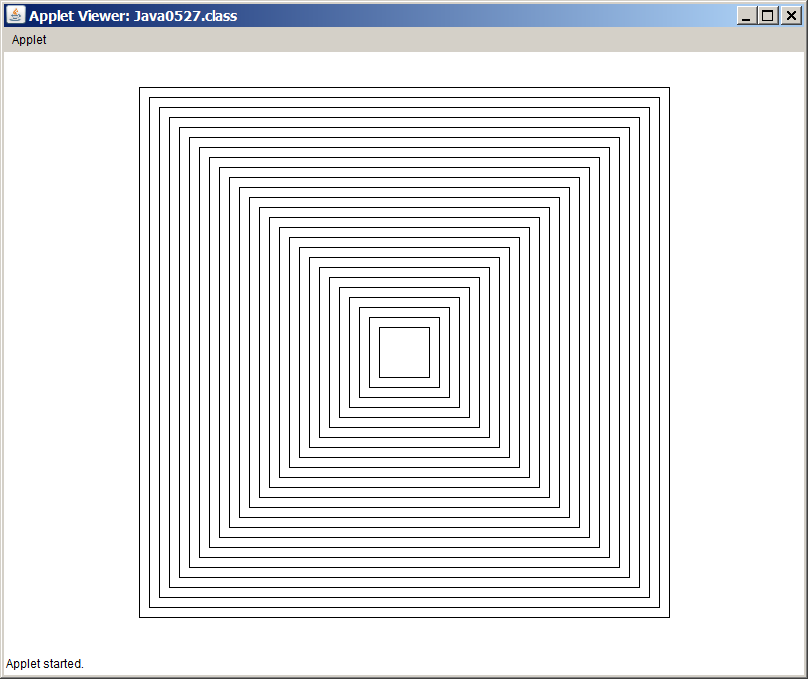


You can create a different effect by changing the top-left coordinate values along with the size of the rectangles. **Java0527.java**, in figure 5.29, uses a **k** loop control variable to repeat twenty-five calls to **drawRect**. Inside the **for** loop, the values for **x**, **y** and **side** change to create the growing squares.

**Figure 5.29**

|  |
| --- |
| // Java0527.java  // This program is another example of displaying multiple graphics rectangles using a loop control structure.  import java.awt.\*;  import java.applet.\*;  public class Java0527 extends Applet  {  public void paint(Graphics g)  {  int x = 375;  int y = 225;  int side = 50;  for (int k = 1; k <= 20; k++)  {  g.drawRect(x,y,side,side);  x -= 10;  y -= 10;  side += 20;  }  }  } |

**Figure 5.29 Continued**

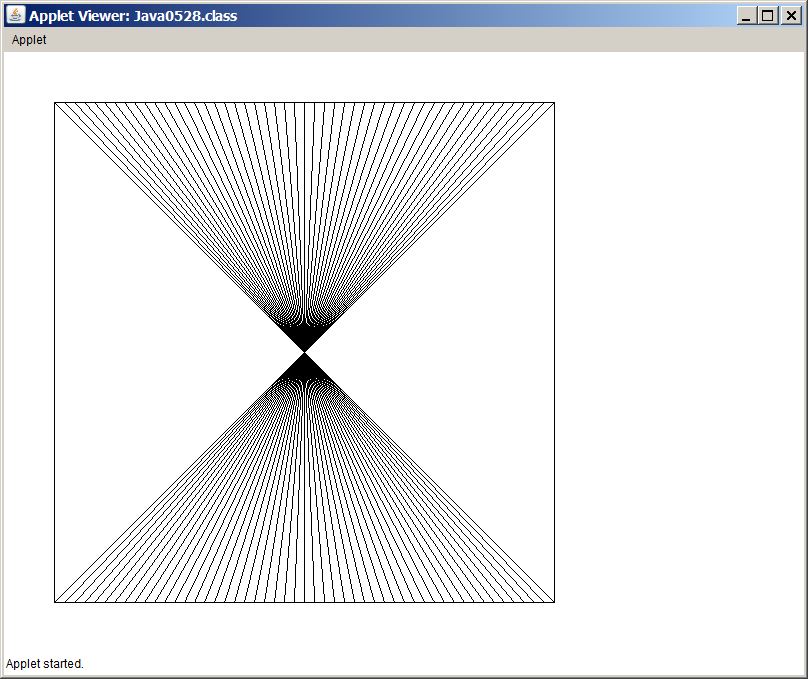


Program **Java0526.java**, in figure 5.30, draws a series of lines that connect the top and bottom of a square to a fixed point.

**Figure 5.30**

|  |
| --- |
| // Java0528.java  // This program demonstrates how to draw multiple lines easily with a loop structure  // inside a rectangle to form a pattern.  import java.awt.\*;  import java.applet.\*;  public class Java0526 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);    }  } |

**Figure 5.30 Continued**

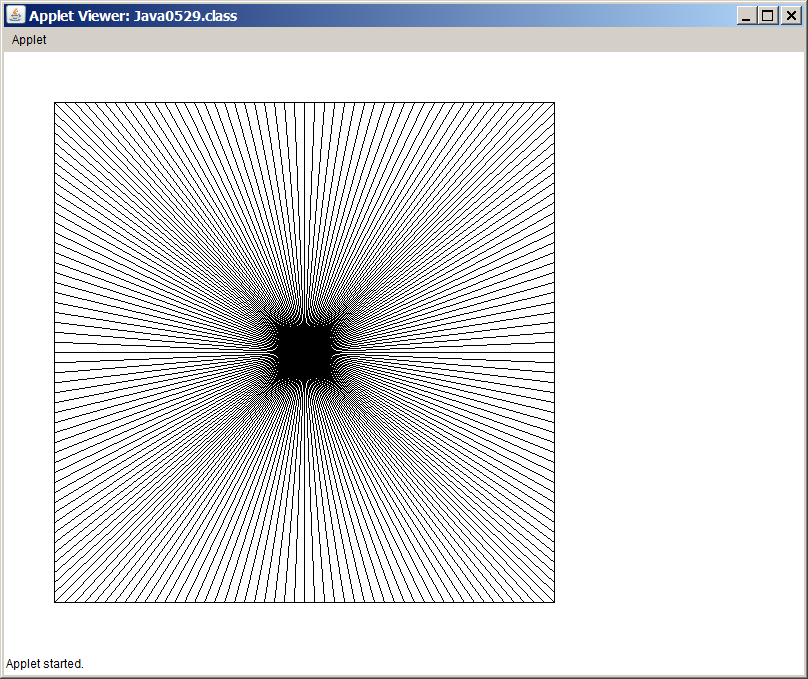


Program **Java0528.java** started an interesting pattern. This pattern will be continued by a second loop structure that connects the left and right side of the square with straight lines. **Java0529.java**, in figure 5.31, demonstrates a square filled with straight lines. Each line travels through the center of the square in such a manner that all lines intersect through the same point.

**Figure 5.31**

|  |
| --- |
| // Java0529.java  // This program continues the pattern started in Java0528.java to create an interesting pattern.  import java.awt.\*;  import java.applet.\*;  public class Java0529 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);  }  } |

**Figure 5.31 Continued**



**5.11 GridWorld and Control Structures**

The GridWorld Case Study is an impressive and complex program. One of its nice features is the ability to control graphics program output with little effort. In this section three programs follow that make the trusty GridWorld bug drop flowers in different patterns. Please realize that it will take quite some time before you will start to feel comfortable with the many capabilities of the GridWorld program.

In this section you are going to look at two of the common **Bug** methods, **move** and **turn** and use these methods together with a variety of control structures to create some flower designs on the grid.

|  |
| --- |
| **Lab Experiment 0530**  The detailed step-by-step instructions, provided in the previous chapter, to create a project, compile the project and then execute the project with many details will not be repeated in this chapter.  You will receive the necessary steps once followed by the final picture result. If this is not sufficient return to Chapter IV for details.  You will do many Lab Experiments with the GWCS. It is very important that you can quickly create a GridWorld project of any specified folder without assistance. |

|  |
| --- |
| **01a. Create a GridWorld Project with Folder Java0530**  **Start JCreator.**  **Click File, New and Project**.  **Click Empty Project and Next.**  **Click the Location Browse button and navigate to folder Java0530.**  **Click OK.**  **Click Finish twice.** (This is not the same as a double click)  **Compile (Build) the project.**  **Execute (Run) the project.**  Everybody should see the exact same grid display as shown in step 01b. |

|  |
| --- |
| **01b. Create a GridWorld Project with Folder Java0530**    In the last chapter the GridWorld objects were placed at random locations on the grid. This project places the **Bug** object at grid location (8,1), which means Row-8, Col-1. It may appear to you that the location is actually Row-9, Col-2. In computer graphics the first row and the first col are both at **0**. This means that the top, left cell is at GridWorld location (0,0).  Program **Java0530.java** is shown below and you see the statement **Location(8,1)**, which places the new object at that location. The other statements you see used here will be explained in future chapters. |

|  |
| --- |
| **02a. Bug Method move and turn**  **Click on the Bug object.** (You will see all the available Bug methods)  **Click the move method.**  **Click on the Bug object then move again.** |

|  |
| --- |
| **02b. Bug Method move and turn**  **Click on the Bug object.**  **Click the turn method.** (The bug turns 45 degrees) |

|  |
| --- |
| **03. Make a small square**  On your own create a small (3 X 3) square. |

|  |
| --- |
| **Lab Experiment 0531**  The previous lab experiment showed that you can create patterns by instructing the GridWorld program to perform one **move** method call or one **turn** method call. In this lab experiment you will see that loops are used to create the pattern of an hourglass. |

|  |
| --- |
| **01. Create Project Java0531**  You should be able to create a GridWorld project efficiently without the need to see any more steps. If you cannot create a project without looking up the steps, you need to devote extra practice time. The GWCS will be used frequently throughout the APCS curriculum and it is required that you can get to a GWCS example project in 30 seconds..  . |

|  |
| --- |
| **02. Execute Project Java0531**  The bug creates a pattern of flowers in the design of an hourglass. |

|  |
| --- |
| **03. Study the program code of Java0531.java**  The start of this program is the same as the previous project.  After the **barry Bug** object is created, multiple loops are used to make **barry** turn and move efficiently until the hourglass pattern is finished. |
| // Java0531.java  // This program displays an hour glass of flowers.  import info.gridworld.actor.ActorWorld;  import info.gridworld.actor.Bug;  import info.gridworld.grid.Location;  public class Java0531  {  public static void main(String[] args)  {  ActorWorld world = new ActorWorld();  Bug barry = new Bug();  world.add(new Location(8,1), barry);  world.show();  barry.turn();  barry.turn();  for (int k = 1; k <= 7; k++)  barry.move();  for (int k = 1; k <= 5; k++)  barry.turn();  for (int k = 1; k <= 7; k++)  barry.move();  for (int k = 1; k <= 3; k++)  barry.turn();  for (int k = 1; k <= 7; k++)  barry.move();  for (int k = 1; k <= 3; k++)  barry.turn();  for (int k = 1; k <= 7; k++)  barry.move();  }  } |

|  |
| --- |
| **Lab Experiment 0532**  This is a second example of a GridWorld program that uses loop control structures to create a pattern. This time the pattern makes a spiral of flowers. |

|  |
| --- |
| **01. Create Project Java0532**  Are you a member of the 30 second club yet? If not practice and practice some more.  . |

|  |
| --- |
| **02. Execute Project Java0532**  The bug creates a pattern of flowers in the design of a spiral. |

|  |
| --- |
| **03. Study the program code of Java0532.java** |
| // Java0532.java  // This program displays a spiral of flowers.  import info.gridworld.actor.ActorWorld;  import info.gridworld.actor.Bug;  import info.gridworld.grid.Location;  public class Java0532  {  public static void main(String[] args)  {  ActorWorld world = new ActorWorld();  Bug barry = new Bug();  world.add(new Location(5,4),barry);  world.show();  for (int k = 1; k <= 2; k++)  barry.move();  barry.turn();  barry.turn();  for (int k = 1; k <= 3; k++)  barry.move();  barry.turn();  barry.turn();  for (int k = 1; k <= 4; k++)  barry.move();  barry.turn();  barry.turn();  for (int k = 1; k <= 5; k++)  barry.move();  barry.turn();  barry.turn();  for (int k = 1; k <= 6; k++)  barry.move();  barry.turn();  barry.turn();  for (int k = 1; k <= 7; k++)  barry.move();  barry.turn();  barry.turn();  for (int k = 1; k <= 8; k++)  barry.move();  barry.turn();  barry.turn();  for (int k = 1; k <= 9; k++)  barry.move();  barry.turn();  barry.turn();  }  } |

|  |
| --- |
| **04. Lab Experiment Java0532 Challenge**  The spiral does use loop structures for efficiency, but there are really too many loop structures. With a combination of loop structures and decision structures, the program code can be much shorter and create the same exact spiral. Take some time now and try to alter the program code. |

**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 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 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**. With the introduction of Java 7.0, it is now possible to create multiple selection using **String** valuesfor **case** values.

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.

|  |
| --- |
| **AP Examination Alert** |
| **These 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** |

|  |
| --- |
| **AP Examination Alert** |
| **These control structures will not be tested on the APCS Examination:**  **switch..case multiple-way selection**  **do..while post-condition iteration** |

Control structures make many programs simpler, especially graphics program that display many pixels and many lines.

The GridWorld Case Study can be used to study the use of control structures. The **Bug** class is excellent for this purpose. Each **move** call of a **Bug** object not only moves the object, but also drops a flower. With this capability the **Bug** object, combined with various control structures can display an interesting patterns on the GridWorld of flowers.