UNIT 3: CONTROL STATEMENTS

UNIT STRCUTURE

3.1 Learning Objectives

3.2 Introduction

3.3 Control Flow Statements

- 3.3.1 Decision Making Statements
- 3.3.2 Looping
- 3.3.3 Branching Statements

3.4 Let Us Sum Up

3.5 Further Readings

3.6 Answers to Check Your Progress

3.7 Model Questions

3.1 LEARNING OBJECTIVES

After going through this unit, you will be able to :

- describe different control statements of Java programming language
- use the control flow statements in Java programs
- describe different branching statements

3.2 INTRODUCTION

We have already learnt the basics of Java programming. In the previous unit, we have learnt about variables, constants, data types, operators and expressions. In this unit, we will discuss control flow statements in Java. Different control flow statements like if, if-else, while, for, break, continue etc. will also be discussed in this unit. In the next unit, we will discuss about class constructors and passing arguments to methods.

3.3 CONTROL FLOW STATEMENTS

When we write a program, we type statements into a file. Without control flow statements, the compiler executes these statements in the order they appear in the file from left to right, top to bottom in a sequence. We can use control flow statements in our programs to conditionally execute statements, to repeatedly execute a block of statements, and to otherwise change the normal, sequential flow of control.

Flow control in Java uses similar syntax as in C and C++. The Java programming language provides several control flow statements, as listed below :

Decision making : if, if-else, switch-case
Looping : while, do-while, for
Branching : break, continue, label, return

3.3.1 Decision Making Statements

While programing, we have a number of situations where we may have to change the order of execution of statements based on certain conditions or decisions. This involves a kind of decision making to see whether a particular condition has occured or not and then direct the computer to execute certain statements accordingly. The statements used to handle those situation are called decision making statement.

The *if* and *if-else* Statements : The ***if*** statement enables a program to selectively execute other statements, based on some criteria. The syntax of if statement is:

```
if (boolean_expression)
{
        statement-block ;
}
```

statement;

The *statement-block* may be a single statement or a group of statements. If the *boolean-expression* evaluates to true, then the block of code inside the *if* statement will be executed. If not the first set of code after the end of the *if* statement(after the closing curly brace) will be executed. For example,

```
if (percentage>=40)
{
        System.out.println("Pass");
}
```

In this case, if **percentage** contains a value that is greater than or equal to 40, the expression is true, and **println( )** will execute. If **percentage** contains a value less than 40, then the **println( )** method is bypassed. What if we want to perform a different set of statements if the expression is false? We use the *else* statement for that. The general syntax of ***if-else*** statement is:

```
if ( Boolean_expression )
        statement; //executes when the expression is true
else
        statement; //executes when the expression is false
```

Let us consider the same example with *Pass* or *Fail* depending on percentage of marks. i.e., if percentage is equal to or more than 40 then the output should be *Pass*; otherwise *Fail.* This can be done by using an *if* statement along with an *else* statement. Here is the segment of code :

```
if (percentage>=40)
        System.out.println("Pass");
else
        System.out.println("Fail");
```

When a series of decisions are involved, we may have to use more than one *if-else* statements in nested form.

The *switch* statement : We have seen that when one of the many alternatives is to be selected, we can design a program using if statements to control the selection. However, the program becomes difficult to read and follow when the number of alternatives increases. Like C/C++, JAVA has a built-in multiway decision statement known as a ***switch***.

The *switch* statement provides variable entry points to a block. It tests the value of a given *variable* or *expression* against a list of ***case*** values and when a match is found, a block of statements associated with that *case* is executed. The general form of *switch* statement is as follows :

```
switch( expression )
{
        case value : statements;
                     break;
        case value : statements
                     break;
        ...
        default : statements      //optional default section
                     break;
}
```

The *expression* is evaluated and compared in turn with each *value* prefaced by the *case* keyword. The values must be *constants* (i.e., determinable at compile-time) and may be of type byte, char, short, int, or long.

3.3.2 Looping

In looping, a sequence of statements are executed until some conditions for the termination of the loop are satisfied. The process of repeatedly executing a block of statements is known as *looping*. At this point, we should remember that Java does not support ***goto*** statement. Like C/C++, Java also provides the three different

statements for looping. These are:

- ***while***
- ***do-while***
- ***for***

The *while* and *do-while* statements : We use a *while* statement to continuously execute a block while the condition remains true. The general syntax of the *while* statement is:

```
while(expression)
{
        statement(s);
}
```

First, the *while* statement evaluates *expression* , which must return a boolean value. If the expression returns *true*, the *while* statement executes the statement(s) in the while block. The *while* statement continues testing the expression and executing its block until the expression returns *false*.

The Java programming language provides another statement that is similar to the *while* statement : the ***do-while*** statement. The general syntax of *do-while* is:

```
do
{
        statement(s);
}while (expression);
```

Statements within the block associated with a *do-while* are executed at least once. Instead of evaluating the expression at the top of the loop, *do-while* evaluates the expression at the bottom.
A program of *while* loop is shown below in program 3.1:

*//**Program 3.1 :** Fibo.java*

```
class Fibo
```

```
{
      public static void main(String args[])
      {
            System.out.println("0\n1");
            int  n0=0,n1=1,n2=1;
            while(n2<50)
            {
                  System.out.println(n2);
                  n0=n1;
                  n1=n2;
                  n2=n1+n0;
            }
            System.out.println(n2);
      }
}
```

The output of the above program will be the Fibonacci series between 0 to 50 (i.e., 0 1 1 2 3 5 8 13 21 34).

The *for* statement : The *for* statement provides a compact way to iterate over a range of values. The general form of the *for* statement can be expressed like this:

```
for (initialization; termination_condition; increment)
{
      statement(s);
}
```

The *initialization* is an expression that initializes the loop. It is executed once at the beginning of the loop. The *termination_condition* determines when to terminate the loop. This condition is evaluated at the top of each iteration of the loop. When the condition evaluates to *false*, the loop terminates. Finally, *increment* is an expression that gets invoked after each iteration through the loop. All these components are optional. In fact, to write an infinite loop, we can

omit all three expressions:

```
for ( ; ; )
    {
        // infinite loop
    }
```

Often, *for* loops are used to iterate over the elements in an array or the characters in a string. The following program segment uses a *for* loop to calculate the summation of 1 to 50:

```
for (i =1; i <= 50 ; i + +)
    {
        sum = sum + i ;
    }
```

3.3.3 Branching Statements

The Java programming language supports three branching statements:

- The ***break*** statement
- The ***continue*** statement
- The ***return*** statement

The *break* statement : In Java, the *break* statements has two forms: *unlabelled* and *labelled*. We have seen the unlabelled form of the *break* statement used with *switch* earlier. As noted there, an unlabelled *break* terminates the enclosing *switch* statement, and the flow of control transfers to the statement immediately following the *switch*. It can be used to terminate a *for*, *while*, or *do-while* loop. A ***break*** (unlabelled form) statement, causes an immediate jump out of a loop to the first statement after its end. When the *break* statement is encountered inside a loop, the loop is immediately exited and the program continues with the statement immediately following the loop. When the loop is nested, the *break* would only exit from the loop containing it. This means, the *break* will exit only a single loop.

The *continue* statement : The ***continue*** statement causes an immediate branch to the end of the innermost loop that encloses it, skipping over any intervening statements. It is written as:

continue ;

A continue does not cause an exit from the loop. Instead, it immediately initiates the next iteration. We can use the continue statement to skip the current iteration of a *for*, *while*, or *do-while* loop.

In Java, we can give a label to a block of statements. A label is any valid Java variable name. To give a label to a loop, we have to place the label name before the loop with a colon at the end. For example,

loop1: for(.....)

```
        {
                .....
                .....
        }
                .....
```

We have seen that a simple break statement causes the control to jump outside the nearest loop and a simple continue statement returns the current loop. If we want to jump outside a nested loop or to continue a loop that is outside the current one, then we may have to use the *labelled* ***break*** and *labelled* ***continue*** statement. The labelled form of *break* and *continue* can be used as follows:

*//**Program 3.2 :*** breakDemo.java

```
class breakDemo
{
    public static void main(String[] args)
    {
        outer:   for(int i=1;i<100;i++)
             {
                 System.out.println(" ");
                 if(i>=10)
```

```
                    break;
                for( int j=1;j<100;j++)
                {
                    System.out.print("* ");
                    if(j==i)
                    continue outer;//labelled continue
                }
            }
        }
    }
```

The output of the above program will be like this:

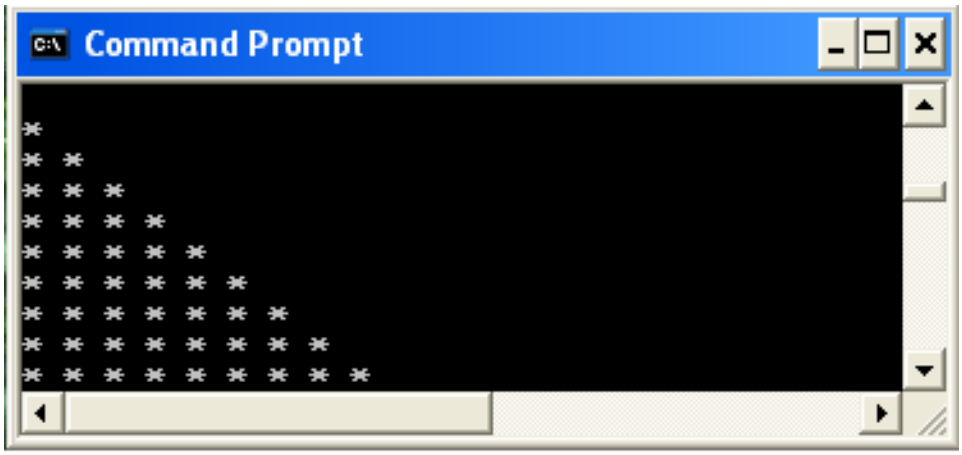


Figure 3.1: Output of program 3.2

The *return* statement : The last of the branching statements is the *return* statement. We can use *return* to exit from the current method. The flow of control returns to the statement that follows the original method call. The *return* statement has two forms: one that returns a value and one that doesnot. To return a value, simply put the value (or an expression that calculates the value) after the return keyword:

return sum;

The data type of the value returned by return must match the type of the method's declared return value. When a method is declared void, use the form of *return* that doesnot return a value:

return;