# UNIT-2
# CHAPTER -3
# DECISION MAKING AND BRANCHING

## 3.1 INTRODUCTION

A Java program is a set of statements, which are normally executed sequentially in the order in which they appear. This happens when options or repetitions of certain calculations are not necessary. However, in practice, we have a number of situations, where we may have to change the order of execution of statements based on certain conditions, or repeat a group of statements until certain specified conditions are met. This involves a kind of decision making to see whether a particular condition has occurred or not and then direct the computer to execute certain statements accordingly. When a program breaks the sequential flow and jumps to another part of the code, it is called *branching*. When the branching is based on a particular condition, it is known as *conditional branching*. If branching takes place without any decision, it is known as *unconditional branching*. Java language possesses such decision making capabilities and supports the following statements known as *control* or *decision making*statements.

1. ifstatement
2. switchstatement
3. conditional operatorstatement

## 3.2 DECISION MAKING WITH IFSTATEMENT

The „if" statement is a powerful decision making statement and is used to control the flow of execution of statements. It is basically a *two-way* decision statement and is used in conjunction with an expression. It takes the followingform:

It allows the computer to evaluate the *expression* first and then, depending on whether the value of the *expression* (relation or condition) is „true' or 'false', it transfers the control to a particular statement. This point of program has two paths to follow, one for the *true* condition and the other for the false condition

The if statement may be implemented in different forms depending on the complexity of conditions to be tested.

- Simple ifstatement
- if. . elsestatement
- Nested if. .elsestatement
- else ifladder

**3.2.1 SIMPLE IFSTATEMENT**

The general form of a simple if statement is

```
if(test expression)
{
   statement-block;
}
statement-x;
```

The 'statement-block' may be a single statement or a group of statements. If the *test expression* is true, the *statement-block* will be executed; otherwise the statement-block will be skipped and the execution will jump to the *statement-x.* It should be remembered that when the condition is true both the statement-block and the statement-x are executed in sequence.

Consider a case having two test conditions, one for weight and another for height. This is done using the compound relation
if (weight < 50 && height> 170) count = count +1;
This would have been equivalently done using two if statements as follows:
**if**(weight<50) **if**
(height>170)
count =count+l;

If the value of weight is less than **50,** then the following statement is executed. Which in turn is another if statement. This if statement tests height and if the height is greater than **170,** then the count is incremented by 1

**3.2.2 THE IF...ELSESTATEMENT**

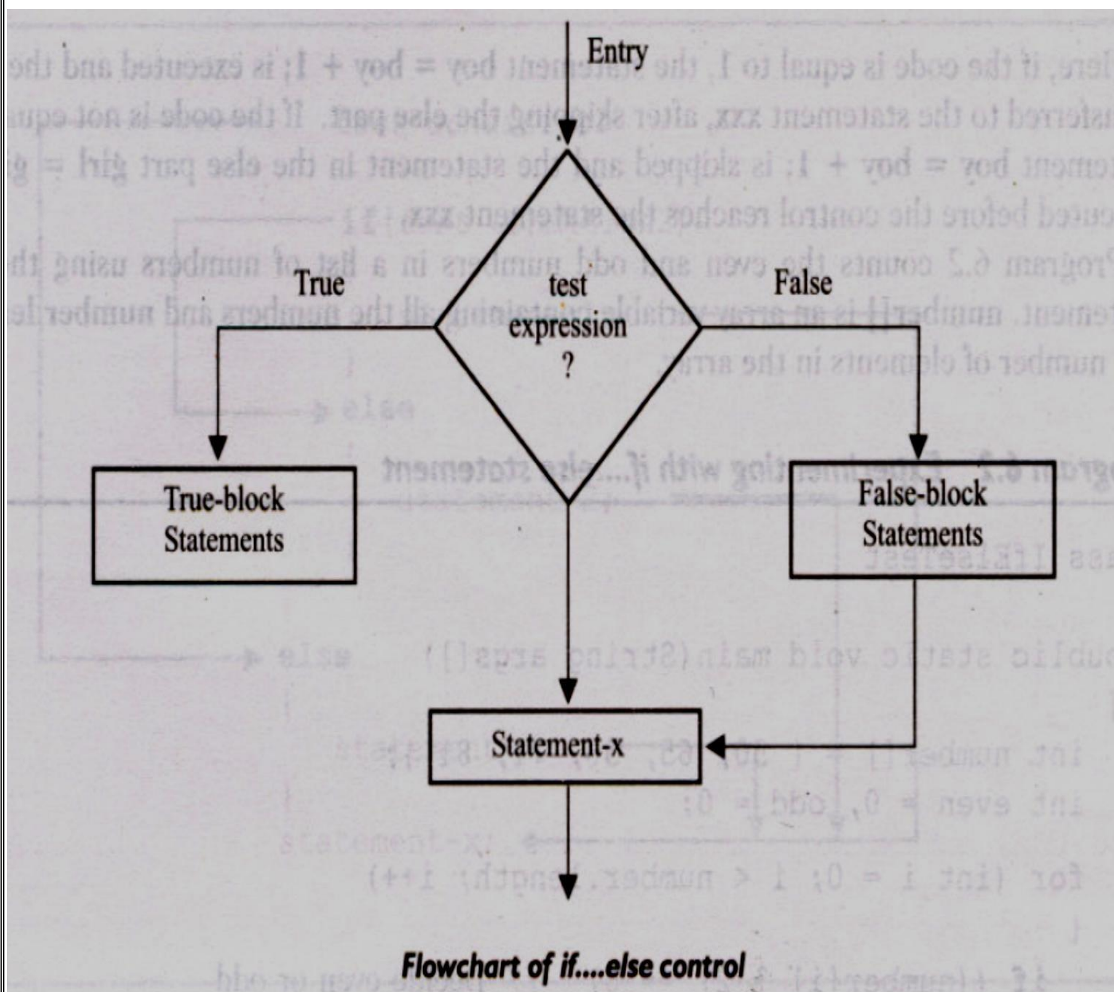
The if else statement is an extension of the simple if statement. The general form is

```
if(test expression)
{
   True-block statement(s)
}
else
{
   False-block statement(s)
}
statement-x
```

If the *test expression* is true, then *the true-block statement(s)* immediately following the if statement, are executed; otherwise, the *false-block statement(s)* are executed. In either case, either *true-block* or *false-block* will be executed, not both. In both the cases, the control is transferred subsequently to the *statement-x.*

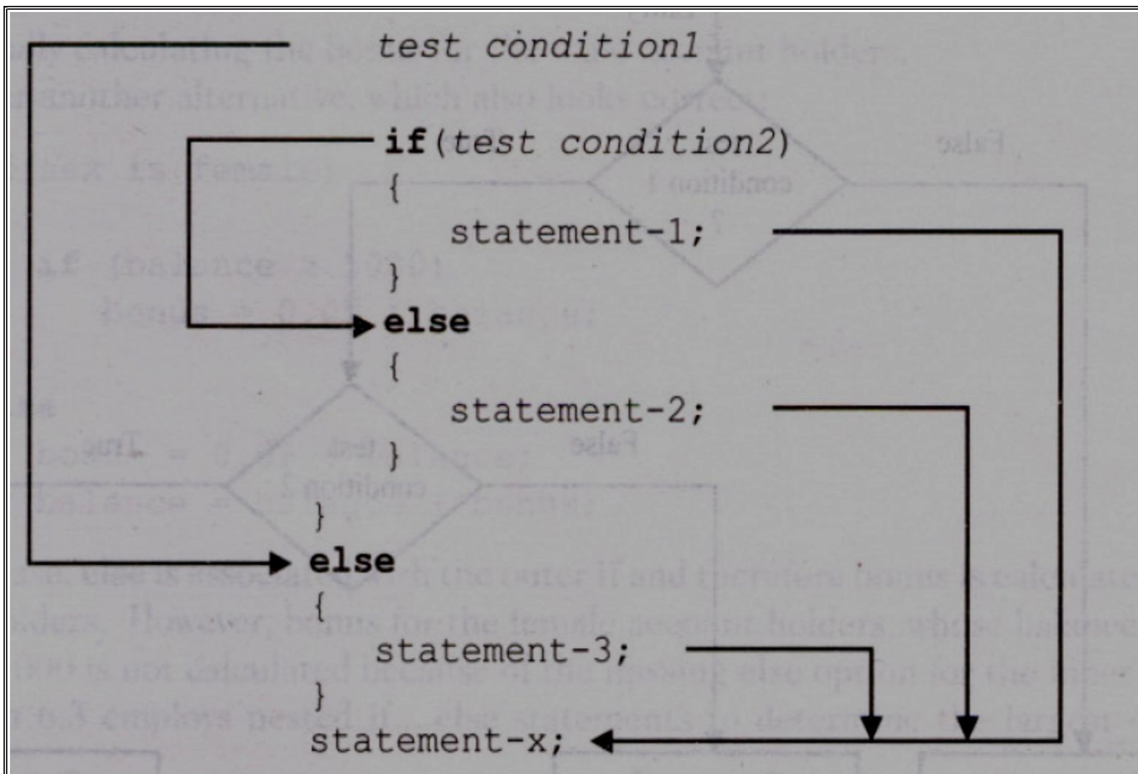
```
if(code == 1)
boy = boy + 1;
else
girl = girl + 1;
XXX;
```

if the code is equal to 1, the statement boy = boy + 1; is executed and the control is transferred to the statement xxx, after skipp,ing the else part. If the code is not equal to 1, the statement boy = boy + 1; is skipped and the statement in the else part girl = girl + 1; is executed before the control reaches the statementxxx



***Flowchart of if....else control***

**3.2.3 NESTING OF IF ELSESTATEMENTS**




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

If the *condition-1* is false, the statement-3 will be executed; otherwise it continues to perform the second test. If the *condition-2* is true, the statement-1 will be evaluated; otherwise the statement-2 will be evaluated and then the control is transferred to the statement-x.

A commercial bank has introduced an incentive policy of giving bonus to all its deposit holders. The policy is as follows: A bonus of 2 per cent of the *balance* held on 31st December is given to every one, irrespective of their balances, and 5 per cent is given to female account holders if their balance is more than Rs 5000. This logic can be coded as follows

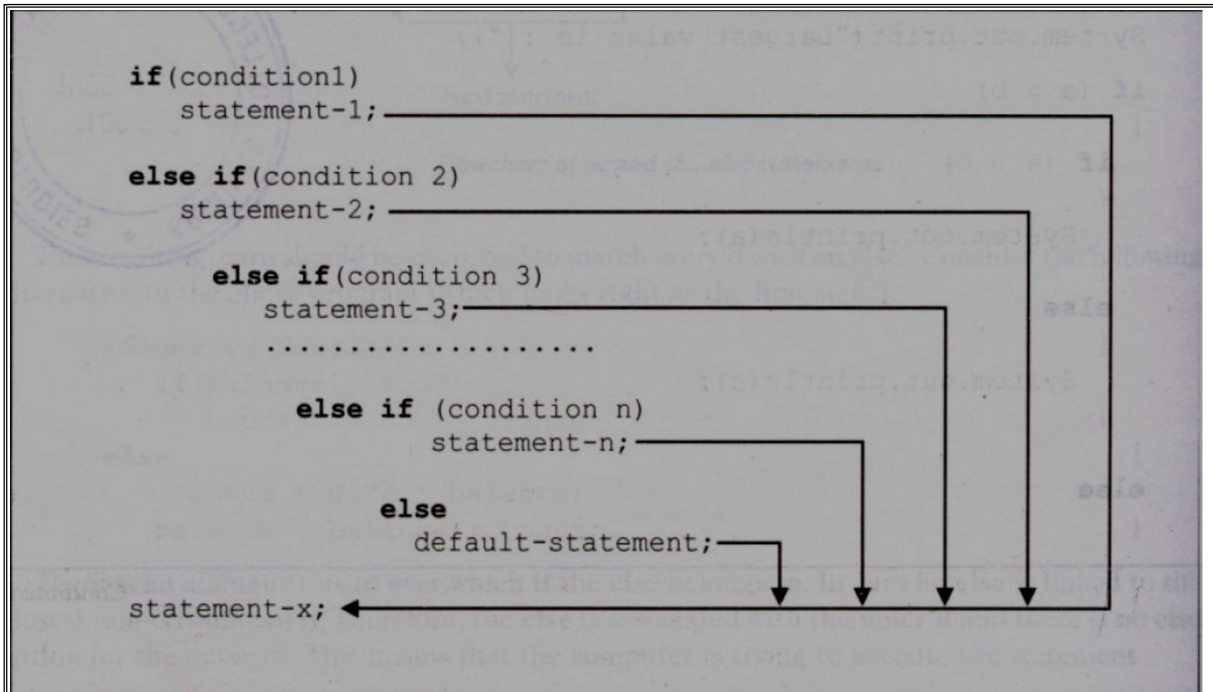
```
if (sex is female)
{
if(balance> 5000)
bonus = 0.05 * balance;
else
bonus = 0.02 * balance;

}
else
{
bonus = 0.02 * balance;
}

balance = balance + bonus;
```

**3.2.4 THE ELSE IFLADDER**

There is another way of putting ifs together when multipath decisions are involved. A multipath decision is a chain of ifs in which the statement associated with each else is an if. It takes the following general form




The previous page construct is known as the else if ladder. The conditions are evaluated from the top *(of* the ladder), downwards. As *soon* as the true condition is found, the statement associated with it is executed and the control is transferred *to* the *statement-x* (skipping the rest *of* the ladder).when all the n conditions become false, then the final else containing the *default-statement* will be executed.

Consider an example *of* grading the students in an academic institution. The grading is done according to the following rules:

Average marks Grade
80 to 100 Honours
60 to 79 First Division
SOto 59 Second Division
40 to 49 Third Division
0 to 39 Fail

This grading can be done using the else ***if*** ladder as follows~

**if**(marks>-79)
grade = "Honours";
**else** if (marks >59)
grade = "First Division";
**else**if (marks > 49)
grade = "Second Division";
**else**if (marks > 39) grade
= "Third Division"; **else**
grade = "Fail";
System.out.println("Grade: " + grade);

**3.3 THE SWITCHSTATEMENT**

When one of the many alternatives is to be selected, we can design a program using if statements to control the selection. However, the complexity of such a program increases dramatically when the number of alternatives increases. The program becomes difficult to read and follow. At times, it may confuse even the designer of the program. Fortunately, Java has a built-in multiway decision statement known as a switch. The switch statement 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 *expression* is an integer expression or characters. *value-l, value-2* .". are constants or constant expressions (evaluable to an integral constant) and are known as *case labels* Each of these values should be unique within a switch statement. *block-l, block-2* are statement lists and may contain zero or more statements. There is no need to put braces around these blocks but it is important to note that case labels end with a colon (:). When the switch is executed, the value of the expression is successively compared against the values *value-l, value-2,* If a case is found whose value matches with the value of the expression, then the block of statements that follows the case are executed. The *break* statement at the end of each block signals the end of a particular case and causes an exit from the switch statement, transferring the control to the *statement-x* following the switch. The *expression* is an integer expression or characters. *value-l, value-2* .". are constants or constant expressions (evaluable to an integral constant) and are known as *case labels* Each of these values should be unique within a switch statement. *block-l, block-2* are statement lists and may contain zero or more statements. There is no need to put braces around these blocks but it is important to note that case labels end with a colon (:).
The general form of the switch statement is as shown below:

```
switch(expression)
{
        case value-1:
                        block-1
                        break;
        case value-2:
                        block-2
                        break;
        ......
        ......
        default:
                        default-block
                        break;
}
statement-x;
```

When the switch is executed, the value of the expression is successively compared against the values *value-l, value-2,* If a case is found whose value matches with the value of the expression, then the block of statements that follows the case are executed. The break statement at the end of each block signals the end of a particular case and causes an exit from the switch statement, transferring the control to the *statement-x* following the switch.

The default is an optional case. When present, it will be execute a if the value of the expression does not match with any of the case values. If not present, no action takes place when all matches fail and the control goes to the *statement-x.*

The switch statement can be used to grade the students as

```
...........
...........
index = marks/10;
switch(index)
{
     case 10:
     case 9:
     case 8:
          grade = "Honours";
          break;
     case 7:
     case 6:
          grade = "First Division";
          break;
     case 5:
          grade = "Second Division";
          break;
     case 4:
          grade = "Third Division";
          break;
     default:
          grade = "Fail";
          break;
 }
 System.out.println(grade);
```

THE ?: OPERATOR

The Java language has an unusual operator, useful for making two-way decisions. This operator is a combination of? and : and takes three operands. This operator is popularly known as the conditional operator. The general form of use of the *conditional operator* is as follows.

```
conditional expression ? expression1 : expression2
```

The *conditional expression* is evaluated first. If the result is true, *expression]* is evaluated and is returned as the value of the conditional expression. Otherwise, *expression2* is evaluated and its value is returned.
For example, the segment
if (x < 0)
flag = 0;
else
flag = 1;
can be written as
flag = (x<O)? 0 : 1;
Consider the evaluation of the following function:

y = 1.5x + 3 for x <= 2
y = 2x + 5 for x > 2

This can be evaluated using .the conditional operator as follows:
y = (x>2) ? (2*x+5) (1.5*x+3);

**3.4 Decision making andLooping**

The process of repeatedly executing a block of statements is known as *looping*. The statements in the block may be executed any number of times, from zero to *infinite* number. If a loop continues forever, it is called an *infiniteloop.*
Java supports such looping features which enable us to develop concise programs containing repetitive processes without using unconditional branching statements like goto statement. Java does not define goto statement. In looping, sequences of statements are executed until some conditions for the termination of the loop are satisfied. A *program loop* therefore consists of two segments, one known as the *body of the loop* and the other known as the control statement. The *control statement* tests certain conditions and then directs the repeated execution of the statements contained in the body of the loop. Depending on the position of the control statement in the loop, a control structure may be classified either as the *entry-controlled* loop or as *exit-controlled* loop. In the entry-controlled loop, the control conditions are tested before the start of the loop execution. If the conditions are "not satisfied, then the body of the loop will not be executed. In the case of an exit-controlled loop, the test is performed at the end of the body of the loop and therefore the body is executed unconditionally for the first time.
The test conditions should be carefully stated in order to perform the desired number of loop executions. It is assumed that the test condition will eventually transfer the control out of the loop. In case, due to some reason it does not do so, the control sets up an *infinite* loop and the

body is executed over and over again. A looping process, in general, would include the following four steps:
1. Setting and initialization of acounter
2. Execution of the statements in theloop.
3. Test for a specified condition for execution of theloop.
4. Incrementing thecounter.
The test may either be to determine whether the loop has been repeated the specified number of times or to determine whether a particular condition has-been met with.
The Java language provides for three constructs for performing loop operations. They are:
1 The while statement
2 The do statement
3 The for statement

**3.4.1 THE WHILESTATEMENT**

The simplest of all the looping structures in Java is the while statement. The basic format of the while statement is

```
Initialization;
While (test condition)
{
    Body of the loop
}
```

The while is an *entry-controlled* loop statement. The *test condition* is evaluated and if the condition is true, then the body of the loop is executed. After execution of the body, the test condition is once again evaluated and if it is true, the body is executed once again. This process of repeated execution of the body continues until the test condition finally becomes false and the control is transferred out of the 1oop. On exit, the program continues with the statement immediately after the body of the loop.
The body of the loop may have one or more statements. The braces are needed only if the body contains two or more statements. However, it is a good practice to use braces even if the body has only one statement.
Consider the following code segment:

The body of the loop is executed 10 times for-n = 1,2, , 10 each time adding the square of the value of n, which.js incremented inside the loop. The test condition may also be written as n < 11; the result would be the same.

```
..........
..........
sum = 0;
n = 1;

while(n <= 10)
{
        sum = sum + n * n;
        n = n+1;
}
System.out.println("Sum = "+ sum);
..........
..........
```

**3.4.2 THE DO STATEMENT**

The while loop construct that we have discussed in the previous section makes a test condition before the loop is executed. Therefore, the body of the loop may not be executed at all if the condition is not satisfied at the very first attempt. On some occasions it might be necessary to execute the body of the loop before the test is performed. Such situations can be handled with the help of the do statement. This takes theform:

```
Initialization;
do
{
   Body of the loop
}
while (test condition)
```

On reaching the **do** statement, the program proceeds to evaluate the body of the loop first. At the end of the loop, the *test condition* in the while statement is evaluated. If the condition is true, the program continues to evaluate the body of the *loop* once again. This process continues as long as the *condition* is true. When the condition becomes false, the loop will be terminated and the control goes to the statement that appears immediately after the **while** statement.

Since the *test condition* is evaluated at the bottom of the loop, the do while construct provides an *exit-controlled* loop and therefore the body of the loop is always executed at least once.

Consider an example:

```
..........
..........
i = 1;
sum = 0;

do
{
    sum = sum + i;
    i = i+2;
}
while(sum < 40 || i < 10);
..........
..........
```

The loop will be executed as long as one of the two relations is true.

**3.4.3 THE FOR STATEMENT**

The for loop is another *entry-controlled* loop that provides a more concise loop control structure. The general form of the for loop is

```
for (intialization ; test condition ; increment)
{
    Body of the loop
}
```

The execution of the for statement is as follows:

1. *Initialization* of the *control variables* is done first, using assignment statements such as i = 1 and count = o. The variables i and count are known as loop-controlvariables.
2. The value of the control variable is tested using the *test condition. The* test condition is a relational expression, such as i <10 that determines when the loop will exit. If the condition is *true,* the body of the loop is executed; otherwise the loop is terminated and the execution continues with the statement that immediately follows theloop.
3. When the body of the loop is executed, the control is transferred back to the for statement after evaluating the last statement in the loop. Now, the control variable is *incremented* using an assignment statement such as i = i+ 1 and the new value of the control variable is again tested to see whether it satisfies the loop condition. If the condition is satisfied, the body of the loop is again executed. This process continues till the value of the control variable fails to satisfy the testcondition.

Consider the following segment of a program

```
for (x  = 0 ; x < = 9 ; x = x+1)
{
      System.out.println(x);
}
```

This for loop is executed 10 times and prints the digits 0 to 9 in one line. The three sections enclosed within parentheses must be separated by semicolons. Note that there is no semicolon at the end of the *increment* section, x = x+ 1. The lor statement allows for negative *increments. For* example, the loop discussed above can be written as follows

```
for ( x = 9 ; x > = 0 ; x = x-1)
    System.out.println(x);
```

This loop is also executed 10 times, but the output would be from 9 to 0 instead of 0 to 9. Braces are optional when the body of the loop contains only one statement. Since the conditional test is always performed at the beginning of the loop, the body of the loop may not be executed at all, if the condition fails at the start. Forexample,

```
for (x = 9; x < 9;x = x-1)
{
..........
..........
}
```

Will never be executed because the test condition fails at the very beginning itself

One of the important points about for loop is that all the three actions, namely *initialization, testing* and *incrementing,* are placed in the for statement itself, thus making them visible to the programmers and users, in oneplace.

Comparison of the Three Loops

| for | while | do |
| --- | --- | --- |
| for (n=1;n<=10;++n)<br>{<br>..........<br>..........<br>} | n = 1<br>while (n<=10)<br>{<br>..........<br>..........<br>n = n+1;<br>} | n = 1<br>do<br>{<br>..........<br>..........<br>n = n+1<br>}<br>while (n<=10); |

**Additional Features of for Loop**

The for loop has several capabilities that are not found in other loop constructs. For example, more than one variable can be initialized at a time in the for statement. The statements

```
p = 1;
for (n=0;  n<17; ++n)
```

can be rewritten as

```
for (p=1, n=0; n<17; ++n)
```

initialization section has two parts p = 1 and n = 1 separated by a *comma.* Like the initialization section, the increment section may also have more than one part. For example, the loop

```
for (n=1, m=50; n<=m; n=n+1, m=m-1
{
        ..........
        ..........
}
```

is perfectly valid. The multiple arguments in the increment section are separated by *commas.* The third feature is that the test condition may have any compound relation and the testing need not be limited only to the loop control variable. Consider the example that follows:

```
sum = 0 ;
for (i = 1, i < 20 && sum < 100; ++i)
{
        ..........
        ..........
}
```

The loop uses a compound test condition with the control variable i and external variable sum. The loop is executed as long as both the conditions i < 20 and sum < 100 are true. The sum is evaluated inside the loop. It is also permissible to use expressions in the assignment statements of initialization and incrementsections.

For example, a statement of the type

**for**(x = (m+n)/2; x > 0; x = x/2) is perfectly valid.

Another unique aspect of for loop is that one or more sections can be omitted, if necessary. Consider the following statements:

```
..........
..........
m = 5;
for ( ;m != 100 ; )
{
      System.out.println(m);
      m = m+5;
}
..........
..........
```

Both the initialization and increment sections are omitted in for statement. The initialization has been done before t for statement and the control variable is incremented inside the loop. In such cases, the sections are left blank. However, the semicolons separating the sections mustremain.

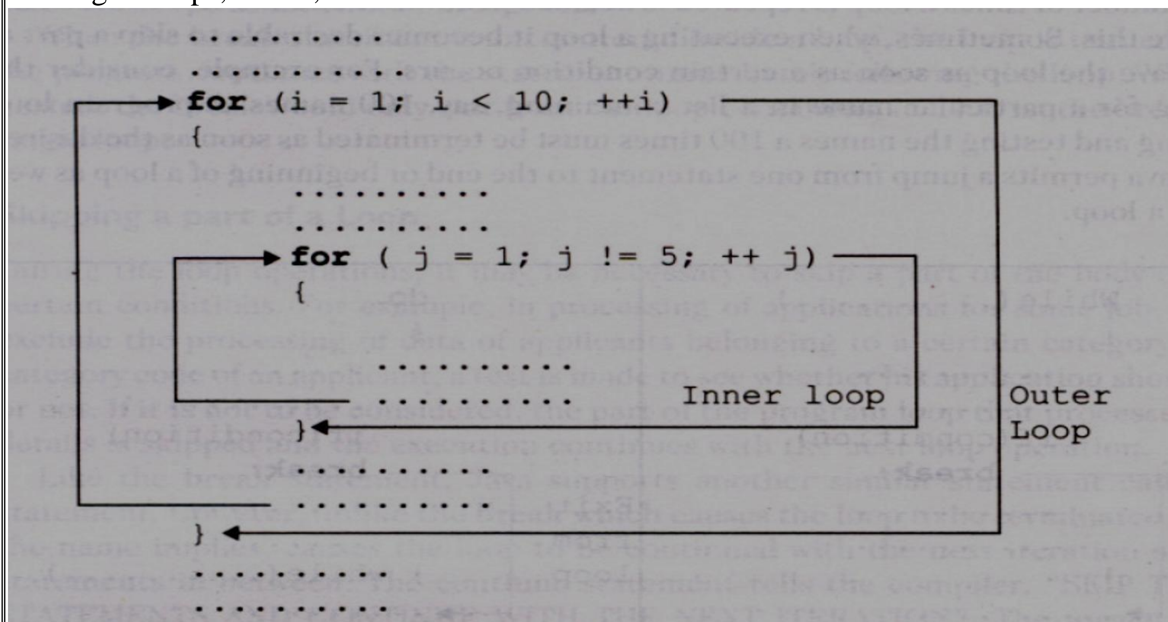
Notice that the body of the loop contains only a semicolon, known as a*empty* statement. This can also be written as

for (j=1000; j > 0; j = j-1);

This implies that the compiler will not give an error message if we place a semicolon by mistake at the end of for statement. The semicolon will be considered as an *empty* statement and the program may produce some nonsense.

**Nesting of for Loops**

Nesting of loops, that is, one for statement within another for statement is allowed in Java.

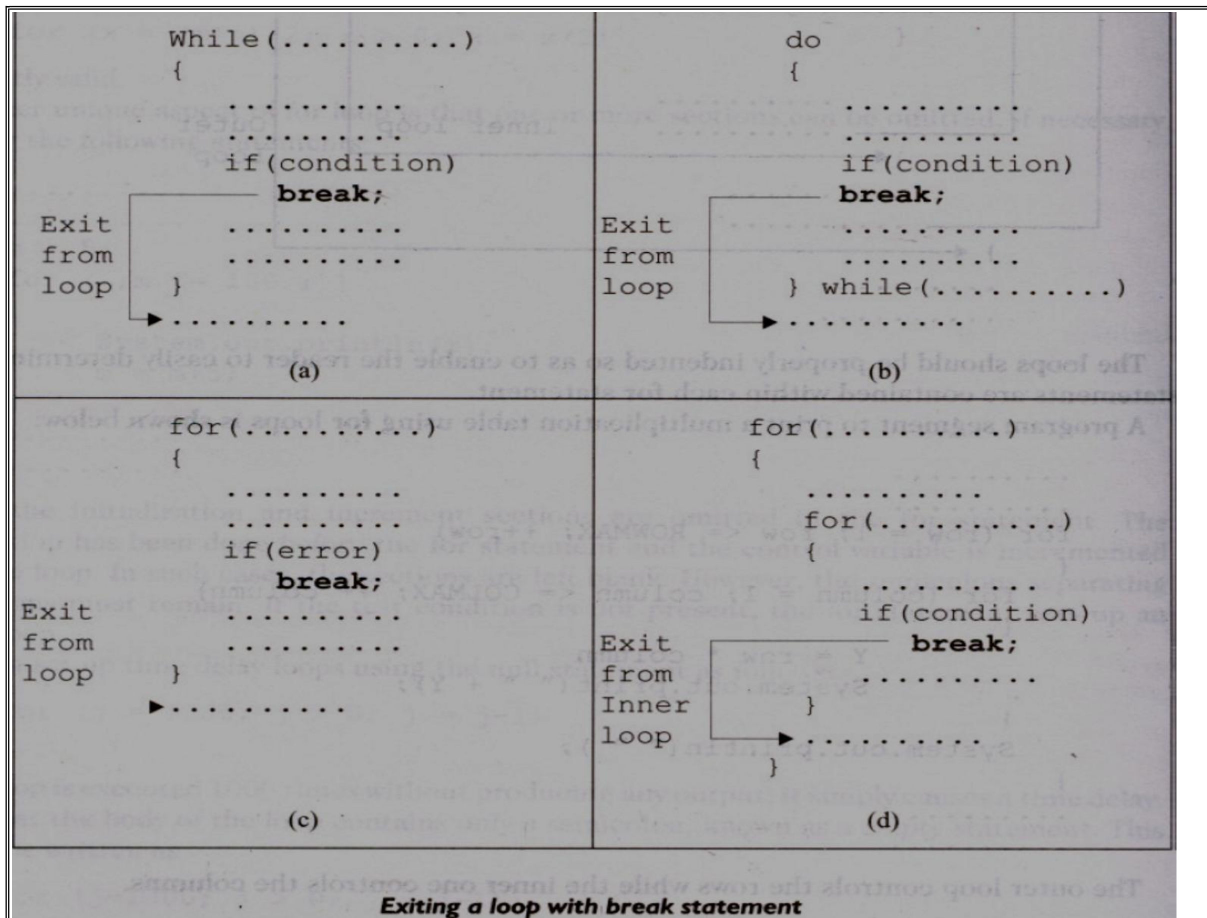



The loops should be properly indented so as to enable the reader to easily determine which statements are contained within each for statement. A program segment to print a multiplication table using for loops is shown below

```
for (row = 1; row <= ROWMAX; ++row)
{
    for (column = 1; column <= COLMAX; ++ column)
    {
        Y = row * column
        System.out.print(" " + Y);
    }
    System.out.println(" " );
}
```

**Jumps in Loops**

Loops perform a set of operations repeatedly until the control variable fails to satisfy the test condition. The number of times a loop is repeated is decided in advance and the test condition is written to achieve this. Sometimes, when executing a loop it becomes desirable to skip a part of the loop or to leave the loop as soon as a certain condition occurs. For example, consider the case of searching for a particular name in a list containing, say, *100* names. A program loop written for reading and testing the names of *100 persons* must be terminated as soon as the desired name is found. Java permits a jump from one statement to the end or beginning of a loop as we as a jump out of a loop.



**Exiting a loop with break statement**

**Jumping Out of a Loop**

An early exit from a loop can be accomplished by using the break statement. This statement can also be used within while, do or for loops. 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 loops are nested, the break would only exit from the loop containing it. That is, the break will exit only a singleloop.

**Skipping a part of a Loop**

Like the break statement, Java supports another similar statement called the continue statement. However, unlike the break which causes the loop to be terminated, the continue, as the name implies, causes the loop to be continued with the next iteration after skipping any statements in between. The continue statement tells the compiler. "SKIP THE FOLLOWING STATEMENTS AND CONTINUE WITH THE NEXT ITERATION". The format of the continue statement is:

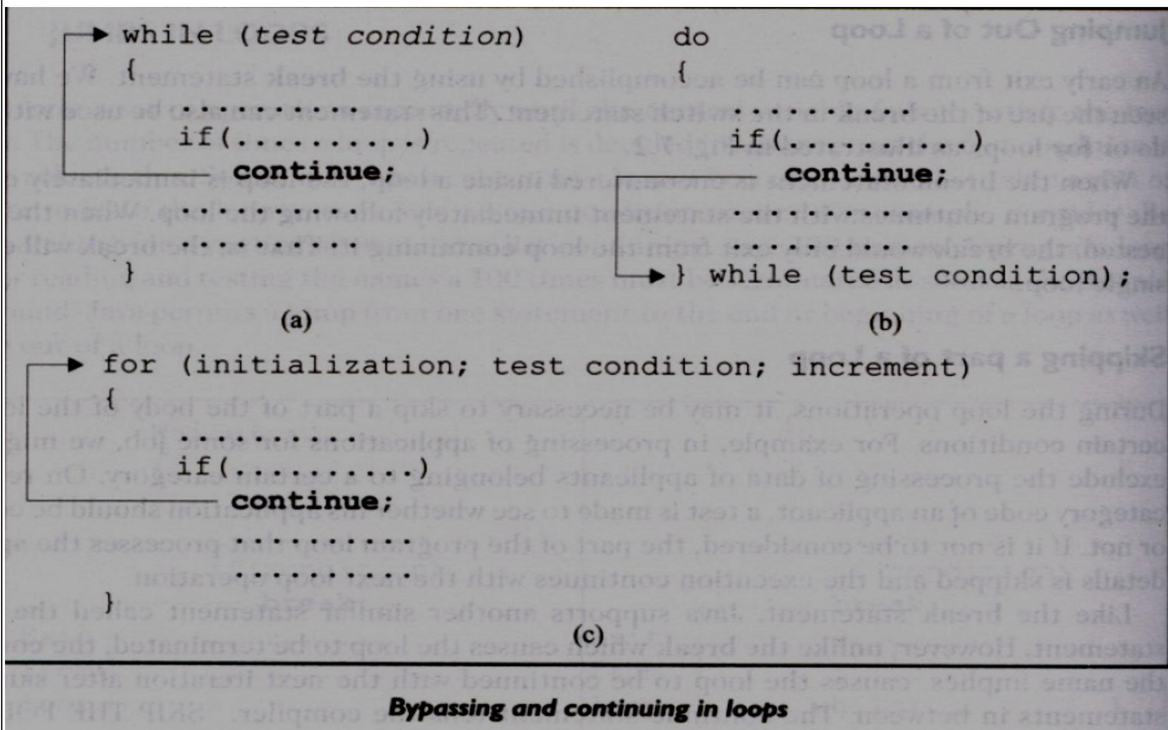
```
Continue;
```

In while and do loops, continue causes the control to go directly to the *test condition* and then to continue the iteration process. In the case of for loop, the *increment* section of the loop is executed before the *test condition* is evaluated.

**3.4.4 LABELLEDLOOPS**

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, place it before the loop with a colon at the end. Example:

```
loop1:      for (..........)
            {
                  ..........
                  ..........
            }
            ..........
```



*Bypassing and continuing in loops*

A block of statements can be labelled as shown below:

```
block1:         {
                   ..........
                   ..........
     block2:       {
                        ..........
                        ..........
                   }
                   ..........
                   ..........
                }
```

Simple break statement causes the control to jump outside the nearest loop and a simple continue statement restarts the current loop. If we want to jump outside a nested loops or to continue a loop that is outside the current one, then we may have to use the labelled break and labelled continuestatements.

```
class ContinueBreak
{
   public static void main(String args[ ])
   {
      LOOP1 :  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 LOOP1;
                  }
               }
               System.out.println("Termination by BREAK");
   }
}
```

The above program produces following output:

```
*
*  *
*  *  *
*  *  *  *
*  *  *  *  *
*  *  *  *  *  *
*  *  *  *  *  *  *
*  *  *  *  *  *  *  *
*  *  *  *  *  *  *  *  *
Termination by BREAK
```

# Chapter-4

## CLASSES, OBJECTS AND METHODS

### 4.1 INTRODUCTION

Java is a true object-oriented language and therefore the underlying structure of all Java programs is classes. Anything we wish to represent in a Java program must be encapsulated in a class that defines the *state* and *behaviour*of the basic program components known as *objects.*

Classes create objects and objects use methods to communicate between them. That is all about object-oriented programming.

Classes provide a convenient method for packing together a group of logically related data items and functions that work on them. In Java, the data items are called fields and the functions are called *methods.* Calling a specific method in an object is described as sending the object a message.
A class is essentially a description of how to make an object that contains fields and methods.

It provides a sort of *template* for an object and behaves like a basic data type such as into it is therefore important to understand how the fields and methods are defined in a class and how they are used to build a Java program that incorporates the basic OOP concepts such as *encapsulation, inheritance* and *polymorphism.*

### 4.2 DEFINING ACLASS

A class is a user-defined data type with a template that serves to define its properties. Once the class type has been defined, we can create "variables" of that type. In Java, these variables are termed as *instances* of classes, which are the actual *objects.* The basic form of a class definition is:

```
class   classname  [extends superclassname]
{
        [ variable declaration; ]
        [ methods declaration; ]
}
```

Everything inside the square brackets is optional. This means that the following would be a valid class definition:

```
class Empty
  {

  }
```

Because the body is empty, this class does not contain any properties and therefore cannot do anything. We can, however, compile it and even create objects using it.

Classname and superclassname are any valid Java identifiers. The keyword extends indicates that the properties of the superclassname class are extended to the classname class. This concept is known as inheritance.

ADDING VARIABLES

Data is encapsulated in a class by placing data fields inside the body of the class definition. These variables are called instance variables because they are created whenever an object of the class is instantiated. We can declare the instance variables exactly the same way as we declare local variables. Example:

```
class Rectangle
  {
        int width;
        intlength;
   }
```

The class **Rectangle** contains two integer type instance variables. It is allowed to declare them in one lineas

```
        int length, width;
```

These variables are only declared and therefore no storage space has been created in the memory.
Instance variables are also known as member variables.

ADDING METHODS

A class with only data fields (and without methods that operate on that data) has no life. The objects created by such a class cannot respond to, any, messages. We must therefore add methods that are necessary for manipulating the data contained in the class. The general form of a method declaration is

```
type methodname (parameter-list)
{
     method-body;
}
```

Method declarations have four basic parts:

- The name of the method (methodname)
- The type of the value the method returns(type)
- A list of parameters(parameter-list)
- The body of themethod

The type specifies the type of value the method would return. This could be a simple data type such as int as well as any class type. It could even be void type, if the method does not return any value. The method name is a valid identifier. The parameter list is always enclosed inparentheses.

This list contains variable names and types of all the values we want to give to the method as input. The variables in the list are separated by commas. In the case where no input data are required, the declaration must retain the empty parentheses.

Examples:

(int m, float x,floaty)          / / Three parameters
()                              / / Emptylist

The body actually describes the operations to be performed on the data. Let us consider the Rectangle class again and add a method getData ( ) to it.

```
class Rectangle
{
        int length;
        int width;

        void getData(int x , int y )
        {
           length  = x ;
           width   = y ;
        }
}
```

Note that the method has a return type of void because it does not return any value. We pass two integer values to the methods which are then assigned to the instance variables length and width. The getData() method is basically added to provide values to the instance variables.

Let us add some more properties to the class. Assume that we want to compute the area of the rectangle defined by the class. This can be done as follows:

```
class Rectangle
{
        int length, width;  // Combind declaration

        void getData(int x , int y)
        {
                length = x ;
                width  = y ;
        }
        int rectArea( )
        {
                int area = length * width;
                return(area);
        }
}
```

The new method rectArea() computes area of the rectangle and returns the result. Since the result would be an integer, the return type of the method has been specified as into Also note that the parameter list is empty.

Declaration of instance variables (and also local variables) can be combined as

int length, width;

The parameter list used in the method header should always be declared independently separated by commas. That is,

voidgetData (int x, y) // Incorrect is illegal.

Now, our class Rectangle contains two instance variables and two methods. We can add more variables and methods, if necessary.

Most of the times when we use classes, we will have many methods and variables within the class. Instance variables and methods in classes are accessible by all the methods in the class but a method cannot access the variables declared in other methods. Example:

```
class Access
{
        int x ;
        void method1( )
        {
                int y ;
                x = 10 ;    // legal
                y =  x  ;   // legal
        }
```

CREATING OBJECTS

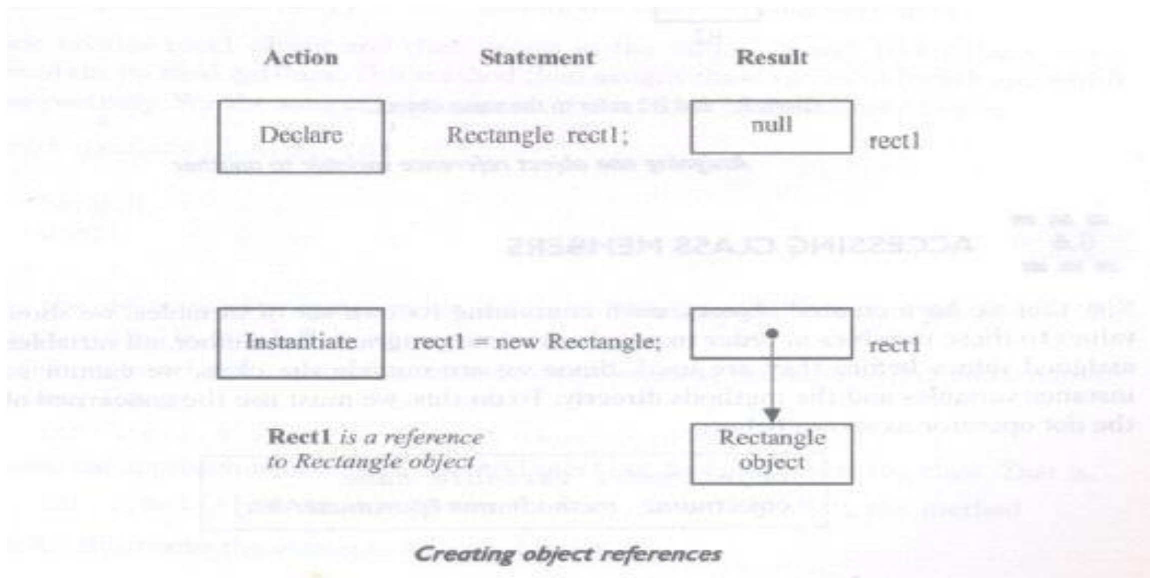
An object in Java is essentially a block of memory that contains space to store all the instance variables. Creating an object is also referred to as instantiating an object.

Objects in Java are created using the new operator. The new operator creates an object of the specified class and returns a reference to that object. Here is an example of creating an object of type Rectangle.

```
        void method2( )
        {
                int z ;
                x = 5 ;     // legal
                z = 10;     // legal
                y = 1 ;     // illegal
        }
}
```

Rectangle rect1 / / declare
rect1 = new Rectangle ( ) / / instantiate

The first statement declares a variable to hold the object reference and the second one actually assigns the object reference to the variable. The variable rect1 is now an object of the Rectangle class. Following figure shows the example

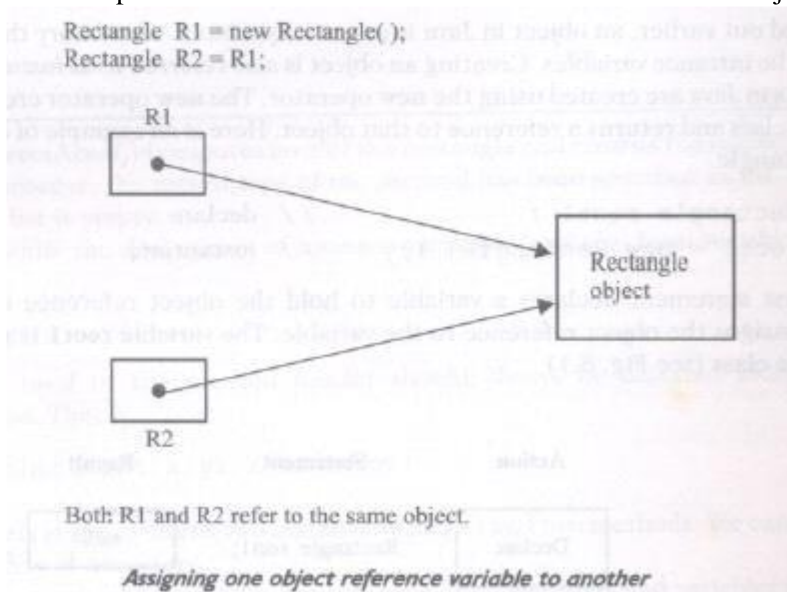


*Creating object references*

Both statements can be combined into one as shown below:
Rectangle rect1 = new Rectangle ( );
The method Rectangle ( ) is the default constructor of the class. We can create any number of objects of Rectangle.

```
Rectangle  rect1 = new  Rectangle(  );
Rectangle  rect2 = new  Rectangle(  );
```

It is important to understand that each object has its own copy of the instance variables of its class. This means that any changes to the variables of one object have no effect on the variables of another. It is also possible to create two or more references to the same object



*Assigning one object reference variable to another*

ACCESSING CLASS MEMBERS

All variables must be assigned values before they are used. Since we are outside the class, "we cannot access the instance variables and the methods directly. To do this, we must use the concerned object and the dot operator as shownbelow:

```
objectname. variable name
objectname . methodname (parameter-list) ;
```

Here object name is the name of the object, variable name is the name of the instance variable inside the object that we wish to access, methodname is the method that we wish to call, and parameter-list is a comma separated list of "actual values" (or expressions) that must match in type and number with the parameter list of the methodname declared in the class. The instance variables of the Rectangle class may be accessed and assigned values asfollows:

```
rect1.length  = 15;
rect1.width   = 10;
rect2.length  = 20;
rect2.width   = 12;
```

Note that the two objects rect1 and rect2 store different values as shown below:

| | rec1 | | rec2 |
|---|---|---|---|
| rect1.length | 15 | rec2.length | 20 |
| rect1.width | 10 | rec2.width | 12 |

This is one way of assigning values to the variables in the objects. Another way and more convenient way of assigning values to the instance variables are to use a method that is declared inside theclass.

In our case, the method getData can be used to do this work. We can call the getData method on any Rectangle object to set the values of both length and width. Here is the code segment to achievethis.

```
Rectangle rect1 = new Rectangle( ); // Creating an object
rect1.getData(15,10);     //  Calling the method using the object
```

This code creates reed object and then passes in the values 15 and 10 for the x and y parameters of the method getData. This method then assigns these values to length and width variables respectively. For the sake of convenience, the method is again shownbelow:

Object rect1 contains values for its variables. We can compute the area of the rectangle represented by rect1. This again can be done in two ways.

- The first approach is to access the instance variables using the dot operator and compute the area . That is,

```
int area1 = rect1.length * rect1.width ;
```

- The second approach is to call the method **rectArea** declared inside the class. That is,

```
int area1 = rect1.rectArea( );          //  Calling the method
```

```
class Rectangle
{
  int length, width;        //    Declaration of variables

  void getData(int x, int y)  //  Definition of method
  {
    length = x;
    width  = y;
  }

  int rectArea()              //  Definition of another method
  {
    int area = length * width;
    return (area);
  }
}
class RectArea                //  Class with main method
{
  public static void main(String args[ ])
  {
    int area1,area2;
    Rectangle rect1 = new Rectangle();     // Creating objects
    Rectangle rect2 = new Rectangle();

    rect1.length = 15;                     // Accessing variables
    rect1.width  = 10;
    area1 = rect1.length * rect1.width;

    rect2.getData(20,12);                  // Accessing methods
    area2 = rect2.rectArea();

    System.out.println("Area1 = " + area1);
    System.out.println("Area2 = " + area2);
  }
}
```

Output:

```
Area1 = 150
Area2 = 240
```

## 4.3 CONSTRUCTORS

All objects that are created must be given initial values. We can do these using two approaches. The first approach uses the dot operator to access the instance variables and then assigns values to them individually. It can be a tedious approach to initialize all the variables of all the objects.

The second approach takes the help of a method like getData to initialize each object individually using statements like,

rect1.getData (15, lO);

It would be simpler and more concise to initialize an object when it is first created. Java supports a special type of method, called a constructor that enables an object to initialize itself when it is created.

Constructors have the same name as the class itself. They do not specify a return type, not even void. This is because they return the instance of the class itself.

Let us consider our Rectangle class again. We can now replace the getData method by a constructor method as shown below:

```
class Rectangle
{
    int length ;
    int width ;

    Rectangle(int x, int y) // Constructor method
    {
        length = x ;
        width  = y ;
    }
    int rectArea( )
    {
        return(length * width);
    }
}
```

Application of constructors

```
class Rectangle
{
    int length, width ;
    Rectangle(int x , int y)                    // Defining constructor
    {
        length = x ;
        width  = y ;
    }
    int  rectArea( )
    {
      return (length * width);
    }
}
class RectangleArea
{
    public static void main(string args[ ])
    {
        Rectangle rect1 = new Rectangle(15,10); // Calling constructor
        int area1 = rect1.rectArea( ) ;
        System.out.println("Area1 = "+ area1) ;
    }
}
```

Output:

```
Area1 = 150
```

**4.4 METHODSOVERLOADING**

In Java it is possible to create methods that have the same name, but different parameter lists and different definitions. This is called method overloading. Method overloading is used when objects are required to perform similar tasks but using different input parameters. When we call a method in an object, Java matches up the method name first and then the number and type of Parameters to decide which one of the definitions to execute. This process is known as polymorphism

To create an overloaded method, all we have to do is to provide several different method definitions in the class, all with the same name, but with different parameter lists. The difference may either be in the number or type of arguments. That is, each parameter list should be unique. Method's return type does not play any role in this. Here is an example of creating an overloaded method.

```
class Room
{
    float length ;
    float breadth ;

    Room(float x, float y)        // constructor1
    {
        length   = x ;
        breadth  = y ;
    }
    Room(float x)                 // constructor2
    {
        length = breadth = x ;
    }
    int area( )
    {
        return (length * breadth) ;
    }
}
```

Here, we are overloading the constructor method Room (). An object representing a rectangular room will be created as

Room room1 = new Room (25.0, 15.0); //using constructor

On the other hand, if the room is square, then we may create the corresponding object as

Room room2 = new Room (20.0); / / using constructor2

**STATIC MEMBERS**

A class basically contains two sections. One declares variables and the other declares methods. These variables and methods are called instance variables and instance methods. This is because every time the class is instantiated, a new copy of each of them is created. They are accessed using the objects (with dotoperator).

Let us assume that we want to define a member that is common to all the objects and accessed without using a particular object. That is, the member belongs to the class as a whole rather than the objects created from the class. Such members can be defined as follows:

staticint count;

static intmax(intx, int y); ')

The members that are declared static as shown above are called static members. Since these members are associated with the class itself rather than individual objects, the static variables and static methods are often referred to as class variables and class methods in order to distinguish them from their counterparts, instance variables and instance methods.

Static variables are used when we want to have a variable common to all instances of a class. One of the most common examples is to have a variable that could keep a count of how many objects

of a class have been created. Remember, Java creates only one copy for a static variable which can be used even if the class is never actually instantiated.

Like static variables, static methods can be called without using the objects. They are also available for use by other classes. Methods that are of general utility but do not directly affect an instance of that class are usually declared as class methods. Java class libraries contain a large number of class methods. For example, the Math class of Java library defines many static methods to perform math operations that can be used in any program. For example,

float x = Math.sqrt (25.0);

The method sqrt is a class method (or static method) defined in Math class.

Note that the static methods are called using class names. In fact, no objects have been created for use. Static methods have severalrestrictions:

1. They can only call other staticmethods.
2. They can only access staticdata.
3. They cannot refer to this or super in anyway

Program given bellow Defining and using static members class Math operation

```
class Mathoperation
{
    static float mul(float x, float y)
    {
        return x*y;
    }
    static float divide(float x, float y)
    {
        return x/y ;
    }
}
class MathApplication
{
    public void static main(string args[ ])
    {
        float a = MathOperation.mul(4.0,5.0) ;
        float b = MathOperation.divide(a,2.0) ;
        System.out.println("b = "+ b) ;
    }
}
```

Output of Above Program is:

b = 10.0

**NESTING OF METHODS**

A method of a class can be called only by an object of that class (or class itself, in the case of static methods) using the dot operator. However, there is an exception to this. A method can be called by using only its name by another method of the same class. This is known as nesting of methods.

Program given bellow illustrates the nesting of methods inside a class. The class Nesting defines one constructor and two methods, namely largest ( ) and display ( ). The method display ( ) calls the method largest ( ) to determine the largest of the two numbers and then displays the result.

Program given bellow illustrates Nesting of methods

```
class Nesting
{
    int m, n;
    Nesting(int x, int y)         //  constructor method
    {
        m = x;
        n = y;
    }
    int largest( )
    {
        if(m >= n)
            return(m);
        else
            return(n);
    }
    void display( )
    {
        int large = largest( );   //  calling a method
        System.out.println("Largest value = " +large);
    }
}
class NestingTest
{
    public static void main(String args[ ])
    {
        Nesting nest = new Nesting(50, 40);
        nest.display( );
    }
}
```

Output of Program given Above would be:
Largest value = 50
A method can call any number of methods. It is also possible for a called method to call another method. That is, method1 may call method2, which in turn may call method3.

**Assignment 2**

1. **Which of the following for loops will be an infiniteloop?**
   A. For(;;)
   B. For(i=0;i<1;i--)
   C. For(i=0;;i++)
   D. All of the above
   **Answer: Option D**
2. **In java--- can only test for equality, where as --- can evaluate any type of Boolean expression.**
   A. Switch,if
   B. If,switch
   C. If, break
   D. Continue,if
   **Answer: Option A**
3. **Which of the following class definitions defines a legal abstractclass?**
   A. Class A{abstract voidunfinished(){}}
   B. Class A {abstract voidunfinished();}
   C. Abstract class A{abstract voidunfinished();}
   D. Public class abstract A(abstract voidunfinished();}
   **Answer: Option C**
4. **Which of the following declares an abstract method in an abstract javaclass?**
   A. Public abstractmethod();
   B. Public abstract voidmethod();
   C. Public void abstractmethod();
   D. Public voidmethod(){}
   E. Public abstract voidmethod(){}
   **Answer: Option B**
5. **Which of the following is a correctinterface?**
   A. Interface A(voidprint(){}}
   B. Abstract interfaceA{print():}
   C. Abstract interface A{abstract voidprint();{}}
   D. Interface A{voidprint();}
   **Answer: Option D**
6. **Runnable isa**
   A. Class
   B. Abstractclass
   C. Interface
   D. Variable
   E. Method
   **Answer: Option C**
7. **Which method compares the given object to thisobject?**
   A. public boolean equals(Objectobj)
   B. public final voidnotifyAll()
   C. public final voidnotify()
   D. public finalClassgetClass()
   **Answer: Option A**
8. **The object cloning is a way to create exact copy of anobject?**
   A. True
   B. False
   **Answer: Option A**
9. **The clone() method is definedin?**
   A. Abstractclass
   B. ObjectClass
   C. ArrayListclass
   D. None of theabove
   **Answer: Option B**

**10. Which method of object class can clone anobject?**

A. copy()
B. Objectcopy()
C. Objectclone()
D. Clone()

**Answer: Option C**

**11. Generally string is a sequence of characters, But in java, string isan**

A. Object
B. Class
C. Package
D. None of theabove

**Answer: Option A**

**12. String class in encapsulated under whichpackage?**

A. java.lang
B. java.util
C. java.io
D. java.awt

**Answer: Option A**

**13. Java defines a peer class of String,called?**

A. StringBuffer
B. StringBuilder
C. Both A &B
D. None of theabove

**Answer: Option A**

**14. Which concept is used to make Java more memoryefficient**

A. Stringliteral
B. By newkeyword
C. Both A &B
D. None of theabove

**Answer: Option A**

**15. Which method of string class in java is used to converts the boolean intoString?**

A. public static String valueOf(doublei)
B. public static String valueOf(booleani)
C. public boolean equals(ObjectanObject)
D. public static String valueOf(Objectobj)

**Answer: Option B**

**16. Which class is thread-safe i.e. multiple threads cannot access it simultaneously,So it is safe and will result in anorder?**

A. StringBufferclass
B. StringBuilderclass
C. Both A &B
D. None of theabove

**Answer: Option A**

**17. Which constructor creates an empty string buffer with the specified capacity aslength.**

A. StringBuffer()
B. StringBuffer(Stringstr)
C. StringBuffer(intcapacity)
D. None of theabove

**Answer: Option C**

**18. The Object class is not a parent class of all the classes in java bydefault?**

A. True
B. False

**Answer: Option B**

**19. The wrapper classes are part of the which package, that is imported by default intoall Javaprograms?**

A. java.lang

B. java.awt
C. java.io
D. java.util

**Answer: Option A**

20. **Which package does java provides in which it acts as an object -oriented wrapper around most commondatabases?**
    A. JDBC
    B. ODBC
    C. None of theabove

    **Answer: Option A**
21. **Wrapper classes are not used to convert any data type into anobject?**
    A. True
    B. False

    **Answer: False**
22. **Void it is not a wrapperclass?**
    A. True
    B. False

    **Answer: Option A**
23. **The primitive data type values will be stored in?**
    A. HeapMemory
    B. StackMemory
    C. Both A &B
    D. None of theabove

    **Answer: Option B**
24. **Which can be used as raw data for operations such as arithmetic, logical,etc?**
    A. Primitive datatypes
    B. Wrapperclasses

    **Answer: OptionA**
25. **The following ways specifies to load the class filesin**
    **By setting the classpath in the commandprompt**
    **By -classpath switch**
    A. Temporary
    B. Permanent
    C. Both A &B
    D. None of theabove

    **Answer: Option A**

**Long Answer Question**

1. Explain simple if statement in Java with an example.
2. Explain if-else statement in Java with an example.
3. Explain else if ladder in Java with an example.
4. Explain the switch statement in Java with an example
5. Explain the for loop in Java with an example.
6. Explain the while loop in Java with an example.
7. Explain the do-while loop in Java with an example.
8. What are objects? How are they created in Java? Explain with an example.
9. Explain the concept of constructors in Java.
10. Explain the concept of method overloading in Java with an example.
11. Explain the concept of static members in Java. List their limitations.
12. How do you define a class in Java? Explain with an example.