NRI INSTITUTE OF INFORMATION SCIENCE & TECHNOLOGY BHOPAL

DEPARTMENT OF INFORMATION TECHNOLOGY

LAB MANUAL

JAVA TECHNLOGY LAB

(IT-306)

INFORMATION TECHNOLOGY (IT)

EXPERIMENT NO.-1

Print “Hello World”

Objective:

To learn the fundamental of JAVA programming:

Underlying Concept/ procedure: **Java** is a programming language originally developed by James Gosling at Sun Microsystems (which has since merged into Oracle Corporation) and released in 1995 as a core component of Sun Microsystems' Java platform. The language derives much of its syntax from C and C++ but has a simpler object model and fewer low-level facilities than either C or C++. Java applications are typically compiled to bytecode (class file) that can run on any Java Virtual Machine (JVM) regardless of computer architecture. Java is a general-purpose, concurrent, class-based, object-oriented language that is specifically designed to have as few implementation dependencies as possible. It is intended to let application developers "write once, run anywhere" (WORA), meaning that code that runs on one platform does not need to be recompiled to run on another. Java is as of 2012 one of the most popular programming languages in use, particularly for client-server web applications, with a reported 10 million users.

Platform-Independent

The concept of Write-once-run-anywhere (known as the Platform independent) is one of the important key feature of java language that makes java as the most powerful language. Not even a single language is idle to this feature but java is more closer to this feature. The programs written on one platform can run on any platform provided the platform must have the JVM.

Simple

There are various features that makes the java as a simple language. Programs are easy to write and debug because java does not use the pointers explicitly. It is much harder to write the java programs that can crash the system but we can not say about the other programming languages. Java provides the bug free system due to the strong memory management. It also has the automatic memory allocation and deallocation system.

Object-Oriented

To be an Object Oriented language, any language must follow at least the four characteristics.

- Inheritance : It is the process of creating the new classes and using the behavior of the existing classes by extending them just to reuse the existing code and adding the additional features as needed.

- Encapsulation: : It is the mechanism of combining the information and providing the abstraction.
- Polymorphism: : As the name suggest one name multiple form, Polymorphism is the way of providing the different functionality by the functions having the same name based on the signatures of the methods.
- Dynamic binding : Sometimes we don't have the knowledge of objects about their specific types while writing our code. It is the way of providing the maximum functionality to a program about the specific type at runtime.

As the languages like Objective C, C++ fulfills the above four characteristics yet they are not fully object oriented languages because they are structured as well as object oriented languages. But in case of java, it is a fully Object Oriented language because object is at the outer most level of data structure in java. No stand alone methods, constants, and variables are there in java. Everything in java is object even the primitive data types can also be converted into object by using the wrapper class.

Robust

Java has the strong memory allocation and automatic garbage collection mechanism. It provides the powerful exception handling and type checking mechanism as compare to other programming languages. Compiler checks the program whether there any error and interpreter checks any run time error and makes the system secure from crash. All of the above features makes the java language robust.

Distributed

The widely used protocols like HTTP and FTP are developed in java. Internet programmers can call functions on these protocols and can get access the files from any remote machine on the internet rather than writing codes on their local system.

Portable

The feature Write-once-run-anywhere makes the java language portable provided that the system must have interpreter for the JVM. Java also have the standard data size irrespective of operating system or the processor. These features makes the java as a portable language.

Dynamic

While executing the java program the user can get the required files dynamically from a local drive or from a computer thousands of miles away from the user just by connecting with the Internet.

Secure

Java does not use memory pointers explicitly. All the programs in java are run under an area known as the sand box. Security manager determines the accessibility options of a class like reading and writing a file to the local disk. Java uses the public key encryption system to allow the java applications to transmit over the internet in the secure encrypted form. The bytecode Verifier checks the classes after loading.

Performance

Java uses native code usage, and lightweight process called threads. In the beginning interpretation of bytecode resulted the performance slow but the advance version of JVM uses the adaptive and just in time compilation technique that improves the performance.

Multithreaded

As we all know several features of Java like Secure, Robust, Portable, dynamic etc; you will be more delighted to know another feature of Java which is **Multithreaded.** Java is also a Multithreaded programming language. Multithreading means a single program having different threads executing independently at the same time. Multiple threads execute instructions according to the program code in a process or a program. Multithreading works the similar way as multiple processes run on one computer. Multithreading programming is a very interesting concept in Java. In multithreaded programs not even a single thread disturbs the execution of other thread. Threads are obtained from the pool of available ready to run threads and they run on the system CPUs. This is how Multithreading works in Java which you will soon come to know in details in later chapters.

Interpreted

We all know that Java is an interpreted language as well. With an interpreted language such as Java, programs run directly from the source code. The interpreter program reads the source code and translates it on the fly into computations. Thus, Java as an interpreted language depends on an interpreter program. The versatility of being **platform independent** makes Java to outshine from other languages. The source code to be written and distributed is platform independent. Another advantage of Java as an interpreted language is its error debugging quality. Due to this any error occurring in the program gets traced. This is how it is different to work with Java.

Architecture-Neutral

The term architectural neutral seems to be weird, but yes Java is an architectural neutral language

as well. The growing popularity of networks makes developers think distributed. In the world of network it is essential that the applications must be able to migrate easily to different computer systems. Not only to computer systems but to a wide variety of hardware architecture and Operating system architectures as well. The Java compiler does this by generating byte code instructions, to be easily interpreted on any machine and to be easily translated into native machine code on the fly. The compiler generates an architecture-neutral object file format to enable a Java application to execute anywhere on the network and then the compiled code is executed on many processors, given the presence of the Java runtime system. Hence Java was designed to support applications on network. This feature of Java has thrived the programming language.

Java is an Object Oriented Language. As a language that has the Object Oriented feature Java supports the following fundamental concepts:

- Polymorphism
- Inheritance
- Encapsulation
- Abstraction
- Classes
- Objects
- Instance
- Method
- Message Parsing

In this chapter we will look into the concepts Classes and Objects.

- **Object -** Objects have states and behaviors. Example: A dog has states-color, name, breed as well as behaviors -wagging, barking, eating. An object is an instance of a class.
- **Class -** A class can be defined as a template/ blue print that describe the behaviors/states that object of its type support.

Java Basic Data Types

Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.

Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals, or characters in these variables.

There are two data types available in Java:

1. Primitive Data Types

2. Reference/Object Data Types

Primitive Data Types:

There are eight primitive data types supported by Java. Primitive data types are predefined by the language and named by a key word. Let us now look into detail about the eight primitive data types.

byte:

- Byte data type is a 8-bit signed two's complement integer.
- Minimum value is -128 (-2^7)
- Maximum value is 127 (inclusive)(2^7 -1)
- Default value is 0
- Byte data type is used to save space in large arrays, mainly in place of integers, since a byte is four times smaller than an int.
- Example : byte a = 100 , byte b = -50

short:

- Short data type is a 16-bit signed two's complement integer.
- Minimum value is -32,768 (-2^15)
- Maximum value is 32,767(inclusive) (2^15 -1)
- Short data type can also be used to save memory as byte data type. A short is 2 times smaller than an int
- Default value is 0.
- Example : short s= 10000 , short r = -20000

int:

- Int data type is a 32-bit signed two's complement integer.
- Minimum value is - 2,147,483,648.(-2^31)
- Maximum value is 2,147,483,647(inclusive).(2^31 -1)
- Int is generally used as the default data type for integral values unless there is a concern about memory.
- The default value is 0.
- Example : int a = 100000, int b = -200000

long:

- Long data type is a 64-bit signed two's complement integer.
- Minimum value is -9,223,372,036,854,775,808.(-2^63)
- Maximum value is 9,223,372,036,854,775,807 (inclusive). (2^63 -1)
- This type is used when a wider range than int is needed.
- Default value is 0L.
- Example : int a = 100000L, int b = -200000L

float:

- Float data type is a single-precision 32-bit IEEE 754 floating point.
- Float is mainly used to save memory in large arrays of floating point numbers.
- Default value is 0.0f.
- Float data type is never used for precise values such as currency.
- Example : float f1 = 234.5f

double:

- double data type is a double-precision 64-bit IEEE 754 floating point.
- This data type is generally used as the default data type for decimal values. generally the default choice.
- Double data type should never be used for precise values such as currency.
- Default value is 0.0d.
- Example : double d1 = 123.4

boolean:

- boolean data type represents one bit of information.
- There are only two possible values : true and false.
- This data type is used for simple flags that track true/false conditions.
- Default value is false.
- Example : boolean one = true

char:

- char data type is a single 16-bit Unicode character.
- Minimum value is '\u0000' (or 0).
- Maximum value is '\uffff' (or 65,535 inclusive).
- Char data type is used to store any character.
- Example . char letterA ='A'

Reference Data Types:

- Reference variables are created using defined constructors of the classes. They are used to access objects. These variables are declared to be of a specific type that cannot be changed. For example, Employee, Puppy etc.
- Class objects, and various type of array variables come under reference data type.
- Default value of any reference variable is null.
- A reference variable can be used to refer to any object of the declared type or any compatible type.
- Example : Animal animal = new Animal("giraffe");

Java Literals:

A literal is a source code representation of a fixed value. They are represented directly in the code without any computation.

Literals can be assigned to any primitive type variable. For example:

```
byte a = 68;
char a = 'A'
```

byte, int, long, and short can be expressed in decimal(base 10),hexadecimal(base 16) or octal(base 8) number systems as well.

Prefix 0 is used to indicates octal and prefix 0x indicates hexadecimal when using these number systems for literals. For example:

```
int decimal = 100;
int octal = 0144;
int hexa =  0x64;
```

String literals in Java are specified like they are in most other languages by enclosing a sequence of characters between a pair of double quotes. Examples of string literals are:

```
"Hello World"
"two\nlines"
"\"This is in quotes\""
```

String and char types of literals can contain any Unicode characters. For example:

```
char a = '\u0001';
String a = "\u0001";
```

Java language supports few special escape sequences for String and char literals as well. They are:

Notation	Character represented
\n	Newline (0x0a)
\r	Carriage return (0x0d)
\f	Formfeed (0x0c)
\b	Backspace (0x08)
\s	Space (0x20)

\t	tab
\"	Double quote
\'	Single quote
\\	backslash
\ddd	Octal character (ddd)
\uxxxx	Hexadecimal UNICODE character (xxxx)

Code:

```
class hello
{
public static void main(String args[ ])
{
System.out.println("Hello World");
}
}
```

Output: Hello World

Conclusion:

EXPERIMENT NO.-2

Java program with multiple statements (square root, multiplication, division etc.)

Objective:

To learn the fundamental of math functions in JAVA programming:

Underlying Concept/ procedure: Class Math

```
java.lang.Object
  └ java.lang.Math
```

---

public final class **Math**

extends Object

The class `Math` contains methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions.

Unlike some of the numeric methods of class `StrictMath`, all implementations of the equivalent functions of class `Math` are not defined to return the bit-for-bit same results. This relaxation permits better-performing implementations where strict reproducibility is not required.

By default many of the `Math` methods simply call the equivalent method in `StrictMath` for their implementation. Code generators are encouraged to use platform-specific native libraries or microprocessor instructions, where available, to provide higher-performance implementations of `Math` methods. Such higher-performance implementations still must conform to the specification for `Math`.

sqrt

```
public static double sqrt(double a)
```

Returns the correctly rounded positive square root of a `double` value. Special cases:

- If the argument is NaN or less than zero, then the result is NaN.
- If the argument is positive infinity, then the result is positive infinity.
- If the argument is positive zero or negative zero, then the result is the same as the argument.

Otherwise, the result is the `double` value closest to the true mathematical square root of the argument value.

Parameters:

`a` - a value.

Returns:

the positive square root of `a`. If the argument is NaN or less than zero, the result is NaN.

Code:

```
import java.lang.Math;

class squareroot

{

public static void main(String args[ ])

{

double x=5;

double y;

y = Math.sqrt(x);

System.out.println("y=" + y);

}

}
```

Output: y = 2.23607

Conclusion:

EXPERIMENT NO.-3

Typecasting

Objective:

To learn the fundamental of typecasting in JAVA programming:

Underlying Concept/ procedure: **Type Casting** refers to changing an entity of one datatype into another. This is important for the type conversion in developing any application. If you will store a int value into a byte variable directly, this will be illegal operation. For storing your calculated int value in a byte variable you will have to change the type of resultant data which has to be stored.

Code:

```
class typewrap

{

public static void main(string args[])

{

char c = 'x';

byte b = 50;

short s=1996;

int i=123456789;

long l =1234567654321L;

float f1 =3.142F;

float f2=1.2e-5F;
```

```
System.out.println("c="+c);

System.out.println("b="+b);

System.out.println("s="+s);

System.out.println("i="+i);

System.out.println("l="+l);

System.out.println("f1="+f1);

System.out.println("f2="+f2);

System.out.println("d2="+d2);


System.out.println(" ");

System.out.println("types converted");


short s1=(short)b;

short s2=(short)i;

float n2=(float)i;

int m1=(int)f1;


System.out.println("(short)b="+s1);

System.out.println("(short)i="+n2);

System.out.println("(float)l="+n2);
```

```
System.out.println("(int)f1="+m1);

}

}
```

Output:

Conclusion:

EXPERIMENT NO.-4

Loops concept

Objective:

To learn the fundamental of loops in JAVA programming:

Underlying Concept/ procedure:

There may be a sitution when we need to execute a block of code several number of times, and is often referred to as a loop.

Java has very flexible three looping mechanisms. You can use one of the following three loops:

- while Loop
- do...while Loop
- for Loop

As of java 5 the enhanced for loop was introduced. This is mainly used for Arrays.

The while Loop:

A while loop is a control structure that allows you to repeat a task a certain number of times.

Syntax:

The syntax of a while loop is:

```
while(Boolean_expression)

{

  //Statements

}
```

When executing, if the boolean_expression result is true then the actions inside the loop will be executed. This will continue as long as the expression result is true.

Here key point of the while loop is that the loop might not ever run. When the expression is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Example:

```
public class Test {

   public static void main(String args[]){

      int x= 10;



      while( x < 20 ){

         System.out.print("value of x : " + x );

         x++;

         System.out.print("\n");

      }

   }

}
```

This would produce following result:

```
value of x : 10

value of x : 11

value of x : 12

value of x : 13

value of x : 14

value of x : 15

value of x : 16

value of x : 17

value of x : 18

value of x : 19
```

The do...while Loop:

A do...while loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time.

Syntax:

The syntax of a do...while loop is:

```
do

{

   //Statements

}while(Boolean_expression);
```

Notice that the Boolean expression appears at the end of the loop, so the statements in the loop execute once before the Boolean is tested.

If the Boolean expression is true, the flow of control jumps back up to do, and the statements in the loop execute again. This process repeats until the Boolean expression is false.

Example:

```
public class Test {

   public static void main(String args[]){

      int x= 10;



      do{

         System.out.print("value of x : " + x );

         x++;

         System.out.print("\n");

      }while( x < 20 );

   }

}
```

This would produce following result:

```
value of x : 10

value of x : 11

value of x : 12

value of x : 13

value of x : 14

value of x : 15

value of x : 16

value of x : 17

value of x : 18

value of x : 19
```

The for Loop:

A for loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

A for loop is useful when you know how many times a task is to be repeated.

Syntax:

The syntax of a for loop is:

```
for(initialization; Boolean_expression; update)

{

   //Statements

}
```

Here is the flow of control in a for loop:

The initialization step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.

Next, the Boolean expression is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and flow of control jumps to the next statement past the for loop.

After the body of the for loop executes, the flow of control jumps back up to the update statement. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the Boolean expression.

The Boolean expression is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then update step,then Boolean expression). After the Boolean expression is false, the for loop terminates.

Example:

```
public class Test {

  public static void main(String args[]){



    for(int x = 10; x < 20; x = x+1){

      System.out.print("value of x : " + x );

      System.out.print("\n");

    }

  }

}
```

This would produce following result:

```
value of x : 10

value of x : 11

value of x : 12
```

```
value of x : 13

value of x : 14

value of x : 15

value of x : 16

value of x : 17

value of x : 18

value of x : 19
```

Enhanced for loop in Java:

As of java 5 the enhanced for loop was introduced. This is mainly used for Arrays.

Syntax:

The syntax of enhanced for loop is:

```
for(declaration : expression)

{

   //Statements

}
```

Declaration . The newly declared block variable, which is of a type compatible with the elements of the array you are accessing. The variable will be available within the for block and its value would be the same as the current array element.

Expression . This evaluate to the array you need to loop through. The expression can be an array variable or method call that returns an array.

Example:

```
public class Test {

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

```
    int [] numbers = {10, 20, 30, 40, 50};

    for(int x : numbers ){
      System.out.print( x );
      System.out.print(",");
    }
    System.out.print("\n");
    String [] names ={"James", "Larry", "Tom", "Lacy"};
    for( String name : names ) {
      System.out.print( name );
      System.out.print(",");
    }
  }
}
```

This would produce following result:

```
10,20,30,40,50,
James,Larry,Tom,Lacy,
```

The break Keyword:

The break keyword is used to stop the entire loop. The break keyword must be used inside any loop or a switch statement.

The break keyword will stop the execution of the innermost loop and start executing the next line of code after the block.

Syntax:

The syntax of a break is a single statement inside any loop:

```
break;
```

Example:

```
public class Test {

  public static void main(String args[]){

    int [] numbers = {10, 20, 30, 40, 50};



    for(int x : numbers ){

      if( x == 30 ){

           break;

      }

      System.out.print( x );

      System.out.print("\n");

    }

  }

}
```

This would produce following result:

```
10

20
```

The continue Keyword:

The continue keyword can be used in any of the loop control structures. It causes the loop to immediately jump to the next iteration of the loop.

In a for loop, the continue keyword causes flow of control to immediately jump to the update statement.

In a while loop or do/while loop, flow of control immediately jumps to the Boolean expression.

Syntax:

The syntax of a continue is a single statement inside any loop:

```
continue;
```

Example:

```
public class Test {

  public static void main(String args[]){

    int [] numbers = {10, 20, 30, 40, 50};



    for(int x : numbers ){

      if( x == 30 ){

            continue;

      }

      System.out.print( x );

      System.out.print("\n");

    }

  }

}
```

This would produce following result:

```
10

20

40
```

```
50
```

Code:

```
class factorial

{

public static void main(String args[])

{

int fact=1,i;

for(i=4;i>=1;i--)

{

fact=fact*i;

}

System.out.println("factorial of 4:"+fact);

}

}
```

Output:

Conclusion:

EXPERIMENT NO.-5

Multiple classes

Objective:

To learn the fundamental of multiple classes in JAVA programming:

Underlying Concept/ procedure:

You've now added extra methods to the basic program structure, but it is also possible to add extra classes too. In the basic program structure you had one class, which was declared public. Since a public class must always have the same name as the file in which it is contained, if your program is to include an extra public class, it should be in a separate file of the same name. Additional classes do not contain a main method.

One advantage of using additional classes is if you've written some methods for use in one program that you think may be useful in another. By writing these methods in a separate public class (in a separate file) any program may make use of them in exactly the same way as usual just by adding the name of the class containing the method, to the beginning of the method name, i.e.

ClassName.methodName(argument);

Code:

```
class room

{

float length;

float breadth;

void getdata(float a, float b)

{

length = a;

breadth = b;

}

}// This class save as room.java
```

```
class roomarea // This class save as roomarea.java and compile, run.

{
public static void main(String args[ ])
{
float area;
room room1 = new room();
room1.getdata(20,10);
area = room1.length*room1.breadth;
System.out.println("Area=" + area);
}
}
```

Output:

Conclusion:

EXPERIMENT NO.-6

If else(decision making statements)

Objective:

To learn the fundamental of decision making statements in JAVA programming:

Underlying Concept/ procedure:

There are two types of decision making statements in Java. They are:

- if statements
- switch statements

The if Statement:

An if statement consists of a Boolean expression followed by one or more statements.

Syntax:

The syntax of an if statement is:

```
if(Boolean_expression)
{
  //Statements will execute if the Boolean expression is true
}
```

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.

Example:

```
public class Test {
  public static void main(String args[]){
    int x = 10;

    if( x < 20 ){
      System.out.print("This is if statement");
    }
  }
}
```

This would produce following result:

```
This is if statement
```

The if...else Statement:

An if statement can be followed by an optional *else* statement, which executes when the Boolean expression is false.

Syntax:

The syntax of a if...else is:

```
if(Boolean_expression){
   //Executes when the Boolean expression is true
}else{
   //Executes when the Boolean expression is false
}
```

Example:

```
public class Test {
   public static void main(String args[]){
      int x = 30;

      if( x < 20 ){
         System.out.print("This is if statement");
      }else{
         System.out.print("This is else statement");
      }
   }
}
```

This would produce following result:

```
This is else statement
```

The if...else if...else Statement:

An if statement can be followed by an optional *else if...else* statement, which is very usefull to test various conditions using single if...else if statement.

When using if , else if , else statements there are few points to keep in mind.

- An if can have zero or one else's and it must come after any else if's.
- An if can have zero to many else if's and they must come before the else.
- Once an else if succeeds, none of he remaining else if's or else's will be tested.

Syntax:

The syntax of a if...else is:

```
if(Boolean_expression 1){
   //Executes when the Boolean expression 1 is true
}else if(Boolean_expression 2){
   //Executes when the Boolean expression 2 is true
}else if(Boolean_expression 3){
   //Executes when the Boolean expression 3 is true
}else {
   //Executes when the none of the above condition is true.
}
```

Example:

```
public class Test {
   public static void main(String args[]){
      int x = 30;

      if( x == 10 ){
         System.out.print("Value of X is 10");
      }else if( x == 20 ){
         System.out.print("Value of X is 20");
      }else if( x == 30 ){
         System.out.print("Value of X is 30");
      }else{
         System.out.print("This is else statement");
      }
   }
}
```

This would produce following result:

```
Value of X is 30
```

Nested if...else Statement:

It is always legal to nest if-else statements, which means you can use one if or else if statement inside another if or else if statement.

Syntax:

The syntax for a nested if...else is as follows:

```
if(Boolean_expression 1){
   //Executes when the Boolean expression 1 is true
   if(Boolean_expression 2){
      //Executes when the Boolean expression 2 is true
   }
}
```

You can nest *else if...else* in the similar way as we have nested *if* statement.

Example:

```
public class Test {
   public static void main(String args[]){
      int x = 30;
                 int y = 10;

      if( x == 30 ){
         if( y == 10 ){
         System.out.print("X = 30 and Y = 10");
      }
   }
}
```

This would produce following result:

```
X = 30 and Y = 10
```

The switch Statement:

A *switch* statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case.

Syntax:

The syntax of enhanced for loop is:

```
switch(expression){
    case value :
       //Statements
       break; //optional
    case value :
       //Statements
       break; //optional
    //You can have any number of case statements.
```

```
    default : //Optional
       //Statements
}
```

The following rules apply to a switch statement:

- The variable used in a switch statement can only be a byte, short, int, or char.
- You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
- The value for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.
- When the variable being switched on is equal to a case, the statements following that case will execute until a *break* statement is reached.
- When a *break* statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
- Not every case needs to contain a break. If no break appears, the flow of control will *fall through* to subsequent cases until a break is reached.
- A *switch* statement can have an optional default case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

Example:

```
public class Test {
   public static void main(String args[]){
      char grade = args[0].charAt(0);

      switch(grade)
      {
      case 'A' :
         System.out.println("Excellent!");
         break;
      case 'B' :
      case 'C' :
         System.out.println("Well done");
         break;
      case 'D' :
         System.out.println("You passed");
      case 'F' :
         System.out.println("Better try again");
         break;
      default :
         System.out.println("Invalid grade");
      }
      System.out.println("Your grade is " + grade);
   }
```

```
}
```

Compile and run above program using various command line arguments. This would produce following result:

```
$ java Test a
Invalid grade
Your grade is a a
$ java Test A
Excellent!
Your grade is a A
$ java Test C
Well done
Your grade is a C
$
```

Code:

```
class test

{

public static void main(String args[ ])

{

int num[ ] = {50,65,56,71,81};

int even = 0, odd = 0;

for(int i=0;i<num.length;i++)

{

If((num[i]%2==0))

{

even += 1;

}

else

{
```

```
odd += 1;
}
}
System.out.println("Even numbers:" + even + "Odd numbers:" + odd);
}
}
```

Output:

Conclusion:

EXPERIMENT NO.-7

Inheritance

Objective:

To learn the fundamental of inheritance in JAVA programming:

Underlying Concept/ procedure:

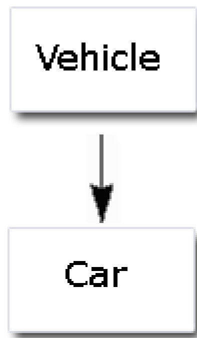
It is one of the most important features of Object Oriented Programming. It is the concept that is used for reusability purpose. Inheritance is the mechanism through which we can derive classes from other classes. The derived class is called as child class or the subclass or we can say the extended class and the class from which we are deriving the subclass is called the base class or the parent class. To derive a class in java the keyword extends is used. To clearly understand the concept of inheritance you must go through the following example.

The concept of inheritance is used to make the things from general to more specific e.g. When we hear the word vehicle then we got an image in our mind that it moves from one place to another place it is used for traveling or carrying goods but the word vehicle does not specify whether it is two or three or four wheeler because it is a general word. But the word car makes a more specific image in mind than vehicle, that the car has four wheels . It concludes from the example that car is a specific word and vehicle is the general word. If we think technically to this example then vehicle is the super class (or base class or parent class) and car is the subclass or child class because every car has the features of it's parent (in this case vehicle) class.

The following kinds of inheritance are there in java.

- **Simple Inheritance**
- **Multilevel Inheritance**

Pictorial Representation of Simple and Multilevel Inheritance



Simple Inheritance **Multilevel Inheritance**

Simple Inheritance

When a subclass is derived simply from it's parent class then this mechanism is known as simple inheritance. In case of simple inheritance there is only a sub class and it's parent class. It is also called single inheritance or one level inheritance.

eg.

```
class A {
 int x;
 int y;
 int get(int p, int q){
 x=p; y=q; return(0);
 }
 void Show(){
 System.out.println(x);
 }
}

class B extends A{
 public static void main(String args[]){
 A a = new A();
 a.get(5,6);
 a.Show();
 }
 void display(){
 System.out.println("B");
 }
}
```

Multilevel Inheritance

It is the enhancement of the concept of inheritance. When a subclass is derived from a derived class then this mechanism is known as the multilevel inheritance. The derived class is called the subclass or child class for it's parent class and this parent class works as the child class for it's just above (parent) class. Multilevel inheritance can go up to any number of level. e.g.

```
class A {
  int x;
  int y;
  int get(int p, int q){
  x=p; y=q; return(0);
  }
  void Show(){
  System.out.println(x);
  }
}
class B extends A{
  void Showb(){
  System.out.println("B");
  }
}

class C extends B{
  void display(){
  System.out.println("C");
  }
  public static void main(String args[]){
  A a = new A();
  a.get(5,6);
  a.Show();
  }
}
```

Java does not support multiple Inheritance

Multiple Inheritance

The mechanism of inheriting the features of more than one base class into a single class is known as multiple inheritances. Java does not support multiple inheritances but the multiple inheritances can be achieved by using the interface.

In Java Multiple Inheritance can be achieved through use of Interfaces by implementing more than one interface in a class.

Code:

```
class room

{

int l;

int b;

room(int x, int y)

{

l = x;

b =y;

}

int area()

{

return(l*b);

}

}

class bedroom extends room

{

int h;

bedroom(int x, int y,int z)

{

super(x,y);
```

```
h = z;
}
int volume( )
{
return(l*b*h);
}
}
class inheritance
{
public static void main(String args[ ])
{
bedroom room1 = new bedroom(14,12,10);
int area1 = room1.area( );
int volume1 = room1.volume( );
System.out.println("Area1 =" + area1);
System.out.println("Volume1 =" + volume1);
}
}
```

Output:

Conclusion:

EXPERIMENT NO.-8

Polymorphism

Objective:

To learn the fundamental of polymorphism in JAVA programming:

Underlying Concept/ procedure:

Polymorphism is the ability of an object to take on many forms. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.

Any java object that can pass more than one IS-A test is considered to be polymorphic. In Java, all java objects are polymorphic since any object will pass the IS-A test for their own type and for the class Object.

It is important to know that the only possible way to access an object is through a reference variable. A reference variable can be of only one type. Once declared the type of a reference variable cannot be changed.

The reference variable can be reassigned to other objects provided that it is not declared final. The type of the reference variable would determine the methods that it can invoke on the object.

A reference variable can refer to any object of its declared type or any subtype of its declared type. A reference variable can be declared as a class or interface type.

Example:

Let us look at an example.

```
public interface Vegetarian{}
public class Animal{}
public class Deer extends Animal implements Vegetarian{}
```

Now the Deer class is considered to be polymorphic since this has multiple inheritance. Following are true for the above example:

- A Deer IS-A Animal
- A Deer IS-A Vegetarian
- A Deer IS-A Deer
- A Deer IS-A Object

When we apply the reference variable facts to a Deer object reference, the following declarations are legal:

```
Deer d = new Deer();
Animal a = d;
Vegetarian v = d;
Object o = d;
```

All the reference variables d,a,v,o refer to the same Deer object in the heap.

Virtual Methods:

In this section, I will show you how the behavior of overridden methods in Java allows you to take advantage of polymorphism when designing your classes.

We already have discussed method overriding, where a child class can override a method in its parent. An overridden method is essentially hidden in the parent class, and is not invoked unless the child class uses the super keyword within the overriding method.

Code:

```
/* File name : Employee.java */
public class Employee
{
   private String name;
   private String address;
   private int number;
   public Employee(String name, String address, int number)
   {
      System.out.println("Constructing an Employee");
      this.name = name;
      this.address = address;
      this.number = number;
   }
   public void mailCheck()
   {
      System.out.println("Mailing a check to " + this.name
       + " " + this.address);
   }
   public String toString()
   {
      return name + " " + address + " " + number;
   }
   public String getName()
   {
```

```
    return name;
  }
  public String getAddress()
  {
    return address;
  }
  public void setAddress(String newAddress)
 {
    address = newAddress;
 }
 public int getNumber()
 {
   return number;
 }
}
```

Now suppose we extend Employee class as follows:

```
/* File name : Salary.java */
public class Salary extends Employee
{
  private double salary; //Annual salary
  public Salary(String name, String address, int number, double
    salary)
  {
     super(name, address, number);
     setSalary(salary);
  }
  public void mailCheck()
 {
     System.out.println("Within mailCheck of Salary class ");
     System.out.println("Mailing check to " + getName()
     + " with salary " + salary);
  }
  public double getSalary()
  {
     return salary;
  }
  public void setSalary(double newSalary)
  {
     if(newSalary >= 0.0)
     {
       salary = newSalary;
     }
  }
```

```
  public double computePay()
  {
    System.out.println("Computing salary pay for " + getName());
    return salary/52;
  }
}
```

Now you study the following program carefully and try to determine its output:

```
/* File name : VirtualDemo.java */
public class VirtualDemo
{
  public static void main(String [] args)
  {
    Salary s = new Salary("Mohd Mohtashim", "Ambehta,  UP",
                          3, 3600.00);
    Employee e = new Salary("John Adams", "Boston, MA",
                          2, 2400.00);
    System.out.println("Call mailCheck using
                          Salary reference --");
    s.mailCheck();
    System.out.println("\n Call mailCheck using
                          Employee reference--");
    e.mailCheck();
   }
}
```

This would produce following result:

```
Constructing an Employee
Constructing an Employee
Call mailCheck using Salary reference --
Within mailCheck of Salary class
Mailing check to Mohd Mohtashim with salary 3600.0

Call mailCheck using Employee reference--
Within mailCheck of Salary class
Mailing check to John Adams with salary 2400.0
```

Output:

Conclusion:

EXPERIMENT NO.-9

Array

Objective:

To learn the fundamental of array in JAVA programming:

Underlying Concept/ procedure:

Java provides a data structure, the **array**, which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables.

This tutorial introduces how to declare array variables, create arrays, and process arrays using indexed variables.

Declaring Array Variables:

To use an array in a program, you must declare a variable to reference the array, and you must specify the type of array the variable can reference. Here is the syntax for declaring an array variable:

```
dataType[] arrayRefVar;   // preferred way.

or

dataType arrayRefVar[];  //  works but not preferred way.
```

Note: The style **dataType[] arrayRefVar** is preferred. The style **dataType arrayRefVar[]**comes from the C/C++ language and was adopted in Java to accommodate C/C++ programmers.

Example:

The following code snippets are examples of this syntax:

```
double[] myList;         // preferred way.

or
```

```
double myList[];         //  works but not preferred way.
```

Creating Arrays:

You can create an array by using the new operator with the following syntax:

```
arrayRefVar = new dataType[arraySize];
```

The above statement does two things:

- It creates an array using new dataType[arraySize];
- It assigns the reference of the newly created array to the variable arrayRefVar.

Declaring an array variable, creating an array, and assigning the reference of the array to the variable can be combined in one statement, as shown below:

```
dataType[] arrayRefVar = new dataType[arraySize];
```

Alternatively you can create arrays as follows:

```
dataType[] arrayRefVar = {value0, value1, ..., valuek};
```

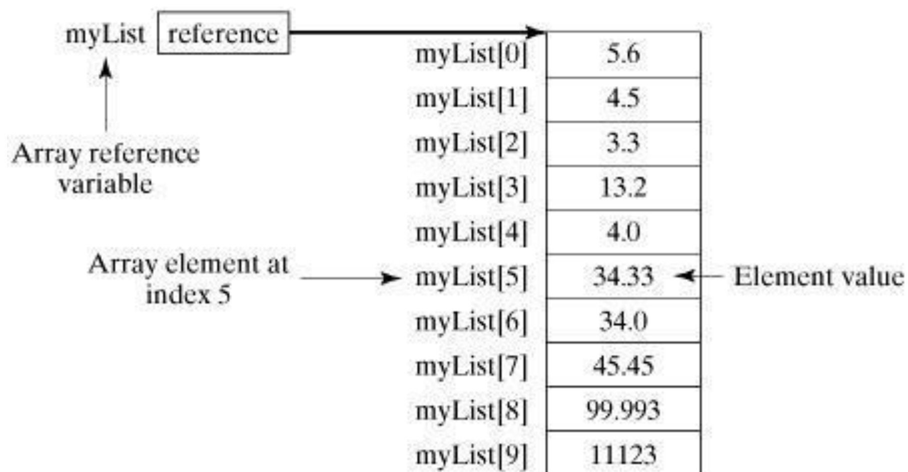
The array elements are accessed through the **index**. Array indices are 0-based; that is, they start from 0 to **arrayRefVar.length-1**.

Example:

Following statement declares an array variable, myList, creates an array of 10 elements of double type, and assigns its reference to myList.:

```
double[] myList = new double[10];
```

Following picture represents array myList. Here myList holds ten double values and the indices are from 0 to 9.



Processing Arrays:

When processing array elements, we often use either for loop or foreach loop because all of the elements in an array are of the same type and the size of the array is known.

Code:

Here is a complete example of showing how to create, initialize and process arrays:

```
public class TestArray {
  public static void main(String[] args) {
    double[] myList = {1.9, 2.9, 3.4, 3.5};

    // Print all the array elements
    for (int i = 0; i < myList.length; i++) {
      System.out.println(myList[i] + " ");
    }
    // Summing all elements
    double total = 0;
    for (int i = 0; i < myList.length; i++) {
      total += myList[i];
    }
    System.out.println("Total is " + total);
    // Finding the largest element
    double max = myList[0];
    for (int i = 1; i < myList.length; i++) {
      if (myList[i] > max) max = myList[i];
    }
    System.out.println("Max is " + max);
  }
}
```

Output:

```
1.9
2.9
3.4
3.5
Total is 11.7
Max is 3.5
```

Conclusion:

EXPERIMENT NO.-10

Encapsulation

Objective:

To learn the fundamental of Encapsulation in JAVA programming:

Underlying Concept/ procedure:

Encapsulation is one of the four fundamental OOP concepts. The other three are inheritance, polymorphism, and abstraction.

Encapsulation is the technique of making the fields in a class private and providing access to the fields via public methods. If a field is declared private, it cannot be accessed by anyone outside the class, thereby hiding the fields within the class. For this reason, encapsulation is also referred to as data hiding.

Encapsulation can be described as a protective barrier that prevents the code and data being randomly accessed by other code defined outside the class. Access to the data and code is tightly controlled by an interface.

The main benefit of encapsulation is the ability to modify our implemented code without breaking the code of others who use our code. With this feature Encapsulation gives maintainability, flexibility and extensibility to our code.

Benefits of Encapsulation:

- The fields of a class can be made read-only or write-only.
- A class can have total control over what is stored in its fields.
- The users of a class do not know how the class stores its data. A class can change the data type of a field, and users of the class do not need to change any of their code.

Code:

Let us look at an example that depicts encapsulation:

```
/* File name : EncapTest.java */
public class EncapTest{

   private String name;
   private String idNum;
   private int age;

   public int getAge(){
```

```
    return age;
  }

  public String getName(){
    return name;
  }

  public String getIdNum(){
    return idNum;
  }

  public void setAge( int newAge){
    age = newAge;
  }

  public void setName(String newName){
    name = newName;
  }

  public void setIdNum( String newId){
    idNum = newId;
  }
}
```

The public methods are the access points to this class's fields from the outside java world. Normally these methods are referred as getters and setters. Therefore any class that wants to access the variables should access them through these getters and setters.

The variables of the EncapTest class can be access as below::

```
/* File name : RunEncap.java */
public class RunEncap{

  public static void main(String args[]){
    EncapTest encap = new EncapTest();
    encap.setName("James");
    encap.setAge(20);
    encap.setIdNum("12343ms");

    System.out.print("Name : " + encap.getName()+
                     " Age : "+ encap.getAge());
   }
}
```

Output: This would produce following result:

```
Name : James Age : 20
```

Conclusion: