Experiment No.1

Title: Implementations of Classes and Objects in Java

Aim: Understanding the concepts of classes and objects in the Java environment.

Theory:

In object-oriented programming technique, we design a program using objects and classes.
An object in Java is the physical as well as a logical entity, whereas, a class in Java is a logical entity only.

What is an object in Java

An entity that has state and behavior is known as an object e.g., chair, bike, marker, pen, table, car, etc. It can be physical or logical (tangible and intangible). The example of an intangible object is the banking system.
An object has three characteristics:

- **state:** represents the data (value) of an object.
- **Behavior:** represents the behavior (functionality) of an object such as deposit, withdraw, etc.
- **Identity:** An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, it is used internally by the JVM to identify each object uniquely.

An object is an instance of a class. A class is a template or blueprint from which objects are created. So, an object is the instance(result) of a class.

Object Definitions:

- An object is *a real-world entity*.
- An object is *a runtime entity*.
- The object is *an entity which has state and behavior*.
- The object is *an instance of a class*.

What is a class in Java

A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical.
A class in Java can contain:

- **Fields**
- **Methods**
- **Constructors**
- **Blocks**
- **Nested class and interface**

Syntax to declare a class:

```
class <class_name>{
    field;
    method;
}
```

Scanner class:

- The Scanner class is a class in java.util, which allows the user to read values of various types. There are far more methods in class Scanner than you will need in this course. We only cover a small useful subset, ones that allow us to read in numeric values from either the keyboard or file without having to convert them from strings and determine if there are more values to be read.
- Class Constructors:
- There are two constructors that are particularly useful: one takes an InputStream object as a parameter and the other takes a FileReader object as a parameter.
- Scanner in = new Scanner(System.in); // System.in is an InputStream
- Scanner inFile = new Scanner(new FileReader("myFile"));
- Numeric and String Methods

Method	*Returns*
int nextInt()	Returns the next token as an int. If the next token is not an integer, InputMismatchException is thrown.
long nextLong()	Returns the next token as a long. If the next token is not an integer, InputMismatchException is thrown.
float nextFloat()	Returns the next token as a float. If the next token is not a float or is out of range, InputMismatchException is thrown.
double nextDouble()	Returns the next token as a long. If the next token is not a float or is out of range, InputMismatchException is thrown.
String next()	Finds and returns the next complete token from this scanner and returns it as a string; a token is usually ended by whitespace such as a blank or line break. If not token exists, NoSuchElementException is thrown.
String nextLine()	Returns the rest of the current line, excluding any line separator at the end.
void close()	Closes the scanner.

```
class Rectangle{
 int length;
 int width;
 void insert(int l, int w){
  length=l;
  width=w;
 }
 void calculateArea(){System.out.println(length*width);}
}
class TestRectangle1{
 public static void main(String args[]){
  Rectangle r1=new Rectangle();
  Rectangle r2=new Rectangle();
  r1.insert(11,5);
  r2.insert(3,15);
  r1.calculateArea();
  r2.calculateArea();
}
}
```

Problem Statement:

Create a class called Employee that includes three pieces of information as instance variables- first name, a last name and a monthly salary. Your class should have a constructor that initializes the three instance variables. Provide a set and a get method for each instance variable. If the monthly salary is not positive, set it to 0.0. Write a test application named EmployeeTest that demonstrates class Employee's capabilities. Create two Employee objects and display each object's yearly salary. Then give each Employee a 10% raise and display each Employee's yearly salary again.

```
import java.util.*;

class Employee
{
        String first_name;
        String last_name;
        double sal;
        public Employee()
        {
                first_name=null;
                last_name=null;
                sal=0.0;
         }

        public String getfirst_name()
        {
                return first_name;
        }

        public String getlast_name()
        {
                return last_name;
        }

        public double getsal()
        {
                return sal;
        }


        public void setfirst_name(String first)
        {
                first_name=first;
        }

        public void setlast_name(String last)
        {
                last_name=last;
        }

        public void setsal(double salary)
        {
                sal=salary;
        }
}

public class Employee_test
{
        public static void main(String[] args)
        {
                Employee e1=new Employee();
                Employee e2=new Employee();

                Scanner in=new Scanner(System.in);

                String first;
                String last;
                double salary;

                        System.out.println("Enter first name of first employee");
                        first=in.next();
                        e1.setfirst_name(first);

                        System.out.println("Enter last name of first employee");
                        last=in.next();
                        e1.setlast_name(last);

                        System.out.println("Enter monthly salary of first employee");
                        salary=in.nextDouble();
                        e1.setsal(salary);


                System.out.println("Enter first name of Second employee");
                        first=in.next();
                        e2.setfirst_name(first);
                System.out.println("Enter last name of second employee");
                        last=in.next();
                        e2.setlast_name(last);
                System.out.println("Enter monthly salary of second employee");
                        salary=in.nextDouble();
                        e2.setsal(salary);


        System.out.println("First Employee's Full name and Salary");
        System.out.println(e1.getfirst_name()+" "+e1.getlast_name()+ " "+e1.getsal()*12 +"\n");
        System.out.println("Second Employee's Full name and Salary");
                System.out.println(e2.getfirst_name()+" "+e2.getlast_name()+" "+e2.getsal()*12 +"\n");

        System.out.println("After incresing salary by 10%");
                System.out.println(e1.getfirst_name()+""+e1.getlast_name()+" "+e1.getsal()*12*1.10+"\n");
                System.out.println(e2.getfirst_name()+""+e2.getlast_name()+" "+e2.getsal()*12*1.10+"\n");
        }
}
```

Conclusion: Thus we implement the simple java program using Classes , Objects , constructor and Scanner class

Experiment No.2

Title:	Implementations the concept of static keyword.
Aim:	Understanding the concepts of static variable and static method.

Theory:

Java static keyword

The static keyword in Java is used for memory management mainly. We can apply static keyword with variables, methods, blocks and nested classes. The static keyword belongs to the class than an instance of the class.

The static can be:

1. Variable (also known as a class variable)
2. Method (also known as a class method)
3. Block
4. Nested class

1) Java static variable

If you declare any variable as static, it is known as a static variable.

- The static variable can be used to refer to the common property of all objects (which is not unique for each object), for example, the company name of employees, college name of students, etc.
- The static variable gets memory only once in the class area at the time of class loading.

Advantages of static variable

It makes your program **memory efficient** (i.e., it saves memory).

Understanding the problem without static variable

```
class Student{
    int rollno;
    String name;
    String college="ITS";
}
```

Suppose there are 500 students in my college, now all instance data members will get memory each time when the object is created. All students have its unique rollno and name, so instance data member is good in such case. Here, "college" refers to the common property of all objects. If we make it static, this field will get the memory only once.

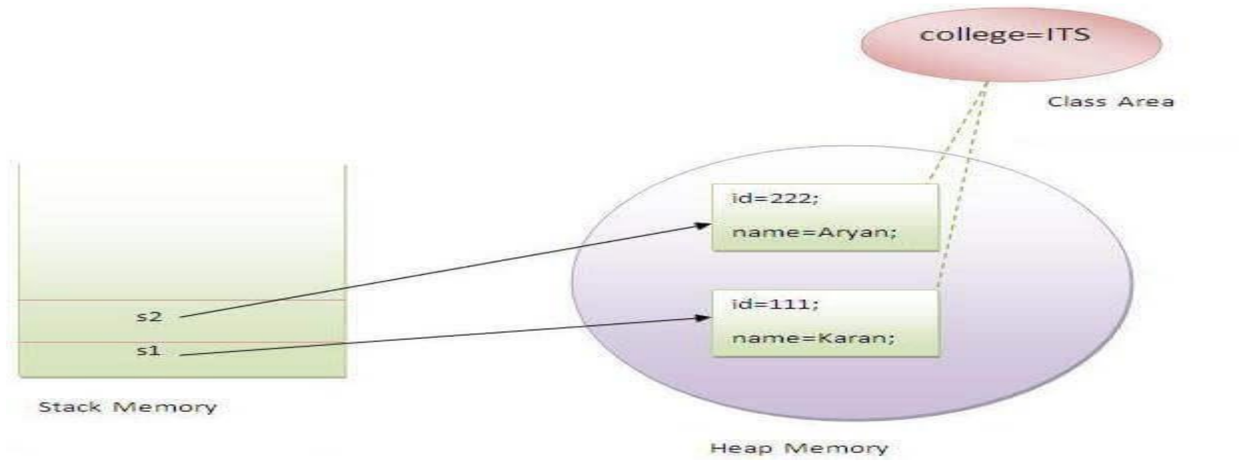
Java static property is shared to all objects.

Example of static variable

```
//Java Program to demonstrate the use of static variable
class Student{
   int rollno;//instance variable
   String name;
   static String college ="ITS";//static variable
   //constructor
   Student(int r, String n){
   rollno = r;
   name = n;
   }
   //method to display the values
   void display (){System.out.println(rollno+" "+name+" "+college);}
}
//Test class to show the values of objects
public class TestStaticVariable1{
 public static void main(String args[]){
 Student s1 = new Student(111,"Karan");
 Student s2 = new Student(222,"Aryan");
 //we can change the college of all objects by the single line of code
 //Student.college="BBDIT";
 s1.display();
 s2.display();
 }
}
```

Output:

```
111 Karan ITS
222 Aryan ITS
```



2) Java static method

If you apply static keyword with any method, it is known as static method.

- A static method belongs to the class rather than the object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- A static method can access static data member and can change the value of it.

Example of static method

```
//Java Program to demonstrate the use of a static method.
class Student{
    int rollno;
    String name;
    static String college = "ITS";
    //static method to change the value of static variable
    static void change(){
    college = "BBDIT";
    }
    //constructor to initialize the variable
    Student(int r, String n){
    rollno = r;
    name = n;
    }
    //method to display values
    void display(){System.out.println(rollno+" "+name+" "+college);}
}
//Test class to create and display the values of object
public class TestStaticMethod{
   public static void main(String args[]){
   Student.change();//calling change method
```

```
    //creating objects
    Student s1 = new Student(111,"Karan");
    Student s2 = new Student(222,"Aryan");
    Student s3 = new Student(333,"Sonoo");
    //calling display method
    s1.display();
    s2.display();
    s3.display();
    }
}
```

Output:

```
111 Karan BBDIT
222 Aryan BBDIT
333 Sonoo BBDIT
```

Statement: Create class SavingsAccount. Use a static variable annualInterestRate to store the annual interest rate for all account holders. Each object of the class contains a private instance variable savingsBalance indicating the amount the saver currently has on deposit. Provide method calculateMonthlyInterest to calculate the monthly interest by multiplying the savingsBalance by annualInterestRate divided by 12this interest should be added to savingsBalance. Provide a static method modifyInterestRate that sets the annualInterestRate to a new value.
Write a program to test class SavingsAccount. Instantiate two savingsAccount objects, saver1 and saver2, with balances of Rs 2000.00 and Rs 3000.00, respectively. Set annualInterestRate to 4%, then calculate the monthly interest and print the new balances for both savers. Then set the annualInterestRate to 5%, calculate the next month's interest and print the new balances for both savers.

Program:

```
import java.util.*;

class SavingsAccount
{
        private static double annualInterestRate;
        private double savingsBalance;

        SavingsAccount()
        {
                savingsBalance=0;
                annualInterestRate=0;
        }

        SavingsAccount(double balance)
        {
                savingsBalance=balance;
                annualInterestRate=0;
        }

        void calculateMonthlyInterest()
        {
                System.out.println("Current savings balance: "+savingsBalance);
                double monthlyInterest;

                monthlyInterest=(savingsBalance*annualInterestRate)/12;
                savingsBalance+=monthlyInterest;

                System.out.println("New savings balance: "+savingsBalance);

        }

        static void modifyInterestRate(double newInterestRate)
        {
                annualInterestRate=newInterestRate;
        }
}

class Saving_test
{
        public static void main(String[] args)
        {
                SavingsAccount saver1=new SavingsAccount(2000);
                SavingsAccount saver2=new SavingsAccount(3000);

                saver1.modifyInterestRate(.04);
                saver1.calculateMonthlyInterest();

                saver2.modifyInterestRate(.04);
                saver2.calculateMonthlyInterest();

                saver1.modifyInterestRate(.05);
                saver1.calculateMonthlyInterest();

                saver2.modifyInterestRate(.05);
                saver2.calculateMonthlyInterest();
        }
}
```

Conclusion: Thus we implement the simple java program using static keyword.

Experiment No. 3

Title: Implementing the concept of Interface.	

Aim: To study

1. Defining interface
2. Implementing interface.
3. Implementing multiple interfaces.
4. Extending interface

Theory:

Introduction

Interfaces are used to encode similarities which classes of various types share, but do not necessarily constitute a class relationship.

An interface in Java is a blueprint of a **class**. It has static constants and abstract methods. The interface in Java is a mechanism to achieve abstraction. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritances in Java. In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.

For example:

```
 interface Bounceable
 {
            public abstract void setBounce();
 /*Interface methods are by default public and abstract and themethods in an interface ends
with a semicolon not with curly brace.*/
 }
```

Defining an interface

Interfaces are defined with the following syntax:

[*visibility*] interface ***InterfaceName*** [extends *other interfaces*] {

constant declarations

member type declarations

abstract method declarations}

The body of the interface contains abstract methods, but since all methods in an interface are, by definition, abstract, the abstract keyword is not required. Since the interface specifies a set of exposed behaviors, all methods are implicitly public.

Implementing an interface

The syntax for implementing an interface uses this formula:

... implements *InterfaceName*[, *another interface*, *another*, ...] ...

If a class implements an interface and is not abstract, and does not implement all its methods, it must be marked as abstract. If a class is abstract, one of its subclasses is expected to implement its unimplemented methods.

Classes can implement multiple interfaces

public class Frog **implements** Predator, Prey { ... }

Sample Program:

```
interface myinter
{
        void meth1();
        void meth2();
}
class myclass implements myinter
{
        public void meth1()
        {
                System.out.println("This is meth1");
        }

        public void meth2()
        {
                System.out.println("This is meth2");
        }
}
class DemoInter
{
        public static void main(String args[])
        {
                myclass  m  =new  myclass();
                m.meth1();
                m.meth2();

        }
}
```

Program for extending Interface

```
interface myinter1
{
        void meth1();
}

interface myinter2 extends myinter1
{
        void meth2();
}

class myclass implements myinter2
{
        public void meth1()
        {
                System.out.println("This is meth1");
        }

        public void meth2()
        {
                System.out.println("This is meth2");
        }
 }
class DemoInterextends
 {
        public static void main(String args[])
        {
                myclass m1 =new myclass();
                m1.meth1();
                m1.meth2();
        }
 }
```

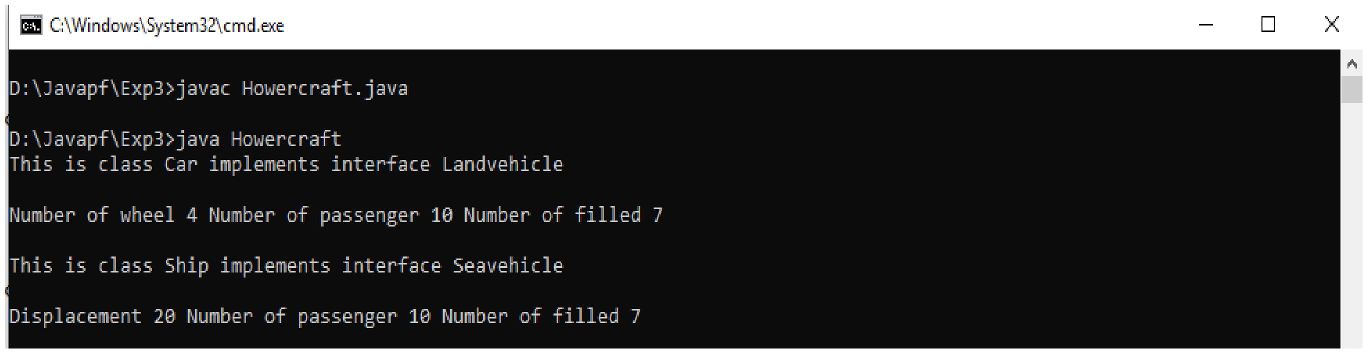
Statement: Create Vehicle Interface with name, maxPassanger, and maxSpeed variables. Create LandVehicle and SeaVehicle Inteface from Vehicle interface. LandVehicle has numWheels variable and drive method. SeaVehicle has displacement variable and launch method. Create Car class from LandVehicle, HoverCraft from LandVehicle and SeaVehicle interface. Also create Ship from SeaVehicle. Provide additional methods in HoverCraft as enterLand and enterSea. Similarly provide other methods for class Car and Ship. Demonstrate all classes in a application.

Program:

```
import java.util.*;

interface Vehicle
{

        int max_passenger=10;
        int max_filled=7;
}

interface Landvehicle extends Vehicle
{
        int num_wheel=4;
        public void drive();
}

interface Seavehicle extends Vehicle
{
        int displacement=20;
        public void launch();
}

abstract class Car implements Landvehicle
{
        abstract void display1();
}

class Howercraft implements Landvehicle, Seavehicle
{

        public void drive()
        {
                System.out.println("Number of wheel "+num_wheel+" Number of passenger
"+max_passenger+" Number of filled "+max_filled+"\n");
        }

        public void launch()
```

```
	{

		System.out.println("Displacement "+displacement+" Number of passenger "+max_passenger+" Number of filled "+max_filled);

	}
	public void display1()
	{
		System.out.println("This is class Car implements interface Landvehicle \n");
	}
	public void display2()
	{
		System.out.println("This is class Ship implements interface Seavehicle \n");
	}
	public static void main(String[] args)

	{
		Howercraft h=new Howercraft();
		h.display1();
		h.drive();
		h.display2();
		h.launch();
	}
}

abstract class Ship implements Seavehicle
{
	abstract  void display2();
}
```

Output:-



```
D:\Javapf\Exp3>javac Howercraft.java

D:\Javapf\Exp3>java Howercraft
This is class Car implements interface Landvehicle

Number of wheel 4 Number of passenger 10 Number of filled 7

This is class Ship implements interface Seavehicle

Displacement 20 Number of passenger 10 Number of filled 7
```

Conclusion: Thus we have studied how to create and implement interface and how to overcome problem of multiple inheritance using interface.

Experiment No. 04

Title: To study and implementing the concept of Inheritance.	
Aim: To study and implementing the concept of Inheritance.	

Theory:

Inheritance Basics

Inheritance is a major component of object-oriented programming. Inheritance will allow to you define a very general class, and then later define more specialized classes by simply adding some new details to the older more general class definition. This saves work, because the more specialized class *inherits* all the properties of the general class and you, the programmer, need onlyprogram the new features.

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object.

The syntax of Java Inheritance

```
class Subclass-name extends Superclass-name
{
  //methods and fields
}
```

The **extends keyword** indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

In the terminology of Java, a class which is inherited is called a parent or superclass, and the new class is called child or subclass.

//Inheritance

```
class A
{
	void meth1()
	{
		System.out.println("I am in Adcet");
	}
}
class B extends A
{
	void meth2()
```

```
        {
                System.out.println("I am in PL LAB");
        }
}
class demo
{
        public static void main(String ar[])
        {
                B
                     b=n
                ew B();
                b.meth1
                ();
                b.meth2();
        }
}
```

Types of inheritance in java

On the basis of class, there can be three types of inheritance in java: **single, multilevel and hierarchical**.

In java programming, multiple and hybrid inheritance is supported through interface only.

Single Inheritance :

When a class inherits another class, it is known as a ***single inheritance*.**

Multilevel Inheritance :

When there is a chain of inheritance, it is known as ***multilevel inheritance*.**

Hierarchical Inheritance :

When two or more classes inherit a single class, it is known as ***hierarchical inheritance*.**

Note: Multiple inheritance is not supported in Java through class.

Q) Why multiple inheritance is not supported in java?

To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.

Since compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error.

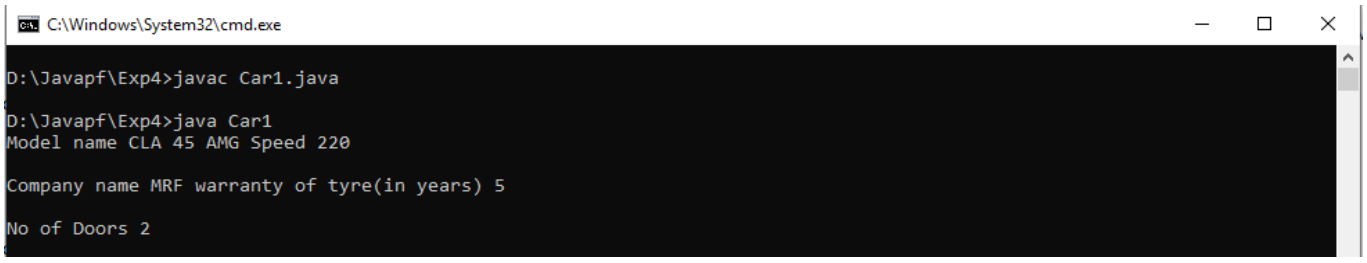
Statement: Create Separate Engine, Tyre, and Door Class. Create a Car class using these classes. Andshow functionality of each component in the car.

Program:

```
import java.util.*; class Engine
{
	int speed;
	String model_name;

	public void eng_Details(int e_speed,String m_name)
	{
		speed=e_speed; model_name=m_name;
		System.out.println("Model name "+model_name+" Speed "+speed);
	}
}
class Tyre extends Engine
{
	int warranty;
	String company_name;


	public void tyre_Details(int t_warranty,String c_name)
	{
		warranty=t_warranty; company_name=c_name;
		System.out.println("\nCompany name "+company_name+" warranty of tyre(in
years)"+warranty);
	}
}

class Door extends Tyre
{
	int door_no;

	public void door_Details(int n_door)
	{
		door_no=n_door;
		System.out.println("\n No of Doors "+door_no);
	}
}

class Car1
{
	public static void main(String[] args)
	{
		Engine e=new Engine();
		Tyre t=new Tyre();
		Door d=new Door();
		e.eng_Details(220,"CLA 45 AMG");
		t.tyre_Details(5,"MRF");
```

```
                    d.door_Details(2);
            }
}
```

Output:-



```
C:\Windows\System32\cmd.exe

D:\Javapf\Exp4>javac Car1.java

D:\Javapf\Exp4>java Car1
Model name CLA 45 AMG Speed 220

Company name MRF warranty of tyre(in years) 5

No of Doors 2
```

Conclusion: Thus we have studied the concept of inner class and inheritance and implement it.

Experiment No. 05

Title: Implementing the concept of package.
1. **Aim:** To study how to create own package. 2. How to import classes from the package 3. How to create hierarchy of classes in package.

Theory:

Packages

Java provides a mechanism for partitioning the class name space into more manageable chunks. This mechanism is the package. The package is both a naming and a visibility control mechanism. You can define classes inside a package that are not accessible by code outside that package. You can also define class members that are only exposed to other members of the same package.

Defining a Package

To create a package is quite easy: simply include a package command as the first statement in a Java source file. Any classes declared within that file will belong to the specified package. The package statement defines a name space in which classes are stored. If you omit the package statement, the class names are put into the default package, which has no name. While the default package is fine for short, sample programs, it is inadequate for real applications. Most of the time, you will define a package for your code.

This is the general form of the **package** statement:

package *pkg*;

Here, *pkg* is the name of the package.

More than one file can include the same **package** statement. The **package** statement simply specifies to which package the classes defined in a file belong. It does not exclude other classes in other files from being part of that same package. Most real-world packages are spread across many files.

The general form of a multileveledpackage statement is shown here:

package *pkg1*[.*pkg2*[.*pkg3*]];

For example, a package declared as package java.awt.image;

needs to be stored in **java/awt/image**, **java\awt\image**, or **java:awt:image** on your UNIX, Windows, or Macintosh file system, respectively. Be sure to choose your package names carefully. You cannot rename a package without renaming the directory in which the classes are stored.

Finding Packages and CLASSPATH

Packages are mirrored by directories.

How does the Java run-time system know where to look for packages that you create?

The answer has two parts. First, by default, the Java run-time system uses the current working directory as its starting point. Thus, if your package is in the current directory, or a subdirectory of the current directory, it will be found. Second, you can specify a directory path or paths by setting the **CLASSPATH** environmental variable. For example, consider the following package specification.

package MyPack;

In order for a program to find **MyPack**, one of two things must be true. Either the program is executed from a directory immediately above **MyPack**, or **CLASSPATH** must be set to include thepath to **MyPack**.

Program:

```
    Dell.java
    package
    mypack;
    public
    class
    Dell
    {
           public void disp()
           {
                   System.out.println("dell class");
           }
    }


     Hp.java
           package mypack;


public class Hp
{
           public void display()
           {
                   System.out.println("Hp class");
           }
}
```

Subpackage

```
Intel1.java
package
mypack.intel;
public class
Intel1
{
           public void view()
           {
                   System.out.println("Intel1 class");
           }
}
```

Main class

```
Laptopshows.java
import mypack.*;
import mypack.intel.*;class Laptopshows
{
        public static void main(String[]args)
        {
                Hp obj=new
                Hp(); Dell
                obj1=new
                Dell();
                Intel1 iobj=new
                Intel1();
                obj.display();
                obj1.disp();
                iobj.view();

        }
}
```

Statement:
Write a Java program to perform employee payroll processing using packages. In the java file, Emp.java creates a package employee and creates a class Emp. Declare the variables name,empid, category, bpay, hra, da, npay, pf, grosspay, incometax, and allowance. Calculate the values in methods. Create another java file Emppay.java. Create an object e to call the methods to perform and print values.

Program:

```
//SAVE AS Emp.java
package employee;
public class Emp{
 String name,empid, category;
 int bpay;
 double hra,da,npay,pf,grosspay,incometax,allowance;
 public Emp(String n, String id, String c, int b)
 {
   name = n;
   empid = id;
   category = c;
   bpay = b;
 }
 public void call()
 {

   da = bpay*0.05;
   hra = bpay*0.09;
   pf = bpay*0.11;
   allowance = bpay*0.10;
   grosspay = bpay+da+hra+allowance-pf;
   incometax = 0.75*grosspay;
   npay = grosspay- incometax;
  }

 public void display()
 {
   System.out.println("/n/n Employee Details");
   System.out.println("/n/n Name:"+name);
   System.out.println("/n/n Empid:"+empid);
   System.out.println("/n/n Category:"+category);
   System.out.println("/n/n bpay:"+bpay);
   System.out.println("/n/n da:"+da);
   System.out.println("/n/n hra:"+hra);
   System.out.println("/n/n pf:"+pf);
   System.out.println("/n/n all:"+allowance);
   System.out.println("/n/n gs:"+grosspay);
   System.out.println("/n/n Incometax:"+incometax);
   System.out.println("/n/n npay:"+npay);
   }
 }

//SAVE as Emppay.java
```

```
import java.io.*;
import employee.Emp;
class Emppay
{

 public static void main(String args[])
 {

   Emp e = new Emp("ANU","23","Female",12000);
   e.call();
   e.display();
   }
 }

/*OUTPUT:
Employee Details
Name: ANU
Empid: 23
Category: Female
bpay: 12000
da: 600.0
hra: 1080.0
pf: 1320.0
allowance: 1200.0
grosspay: 13560.0
Incometax: 10170.0
npay: 3390  */
```

Experiment No.6

Title: Implementing the concept of Exception Handling	

Aim: To Study

1. How to monitor code for Exception
2. How to Catch exception
3. How to use throws and finally clauses
4. how to create our own exception class

Theory:

- An *exception* is an abnormal condition that arises in a code sequence at run time.
- A Java exception is an object that describes an exceptional condition that has occurred in a piece of code
- When an exceptional condition arises, an object representing that exception is created and *thrown* in the method that caused the error
- An exception can be caught to handle it or pass it on
- Exceptions can be generated by the Java run-time system, or they can be manually generated by your code
- Java exception handling is managed by via five keywords: **try, catch, throw, throws,** and **Finally**
- Program statements to monitor are contained within a **try** block
- If an exception occurs within the **try** block, it is thrown
- Code within **catch** block catch the exception and handle it
- System generated exceptions are automatically thrown by the Java run-time system
- To manually throw an exception, use the keyword **throw**
- Any exception that is thrown out of a method must be specified as such by a **throws** clause

1. Any code that absolutely must be executed before a method returns is put in a **finally** block
2. General form of an exception-handling block

```
Try

{
    // block of code to monitor for errors
}
catch (ExceptionType1 exOb){
    // exception handler for ExceptionType1
}
catch (ExceptionType2 exOb){
    // exception handler for ExceptionType2
}
//…
```

```
finally{
      // block of code to be executed before try block ends
}
```

Exception Types

- All exception types are subclasses of the built-in class **Throwable**
- Throwable has two subclasses, they are
  - o Exception (to handle exceptional conditions that user programs should catch)
    - An important subclass of Exception is **RuntimeException**, that includes division by zero and invalid array indexing
  - o Error (to handle exceptional conditions that are not expected to be caught under normal circumstances). i.e. stack overflow
- finally
- It is used to handle premature execution of a method (i.e. a method open a file upon entry and closes it upon exit)
- **finally** creates a block of code that will be executed after **try/catch** block has completed and before the code following the **try/catch** block
- **finally** clause will execute whether or not an exception is thrown

Creating your Own Exception Classes

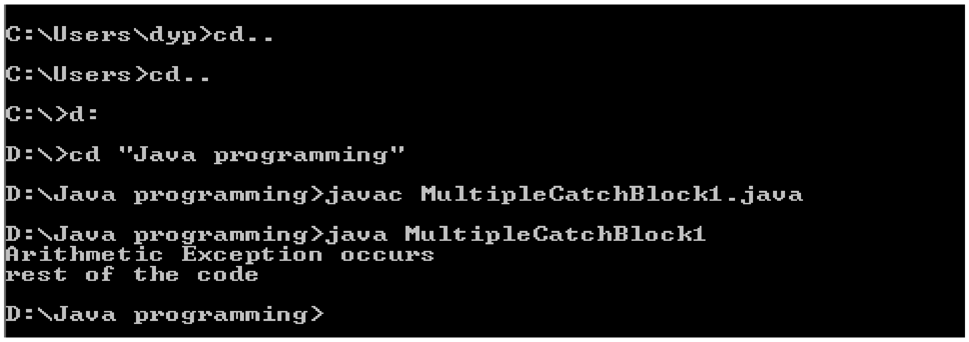
- You may not find a good existing exception class
- Can subclass Exception to create your own
- Give a default constructor and a constructor that takes a message

Example :

```
public class MultipleCatchBlock1 {


public static void main(String[] args) {


     try{
          int a[]=new int[5];
          a[5]=30/0;
         }
        catch(ArithmeticException e)
           {
            System.out.println("Arithmetic Exception occurs");
           }
        catch(ArrayIndexOutOfBoundsException e)
           {
            System.out.println("ArrayIndexOutOfBounds Exception occurs");
           }
```

```
                    catch(Exception e)
                      {
                       System.out.println("Parent Exception occurs");
                      }
                    System.out.println("reset of the code");
              }
          }
```

Output:



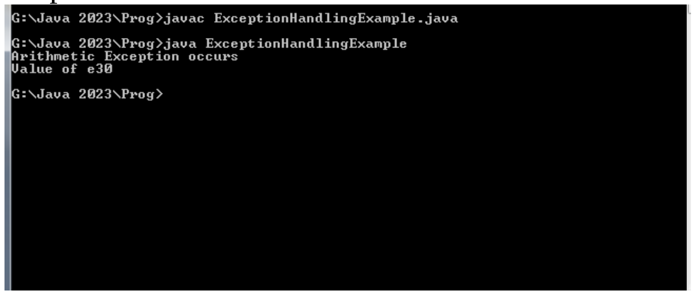
```
C:\Users\dyp>cd..

C:\Users>cd..

C:\>d:

D:\>cd "Java programming"

D:\Java programming>javac MultipleCatchBlock1.java

D:\Java programming>java MultipleCatchBlock1
Arithmetic Exception occurs
rest of the code

D:\Java programming>
```

Problem Statement:

Develop application which can handle any 5 combination of predefined compile time and runtime exceptions using multiple catch blocks. Use throws and finally keywords as well.

```
public class ExceptionHandlingExample
{
    public static void main(String[] args)
{
        try
            {
            // Code that may throw exceptions
                        int a[]=new int[5];
                        a[5]=30/0;
        }
            catch (NullPointerException e)
            {
            // Handle NullPointerException
                        System.out.println("NullPointerException Exception occurs");
        }
            catch (ArrayIndexOutOfBoundsException e)
            {
            // Handle ArrayIndexOutOfBoundsException
                        System.out.println("ArrayIndexOutOfBounds Exception occurs");
        }
            catch (NumberFormatException e)
            {
            // Handle NumberFormatException
        }
            catch (ArithmeticException e)
            {
            // Handle ArithmeticException
                        System.out.println("Arithmetic Exception occurs");
            }

            catch (Exception e)
            {
            // Handle any other exception
                        System.out.println("Exception occurs");
        }
            finally
            {
            // Code that should always run
            int c=10,d=20;
                        int e=d+c;
                        System.out.println("Value of e:\t"+e);
        }
    }
}
```

Output:



```
G:\Java 2023\Prog>javac ExceptionHandlingExample.java

G:\Java 2023\Prog>java ExceptionHandlingExample
Arithmetic Exception occurs
Value of e30

G:\Java 2023\Prog>
```

Conclusion: Thus we have studied Exception handling in different ways.

Experiment No.7

Title: Implementing the concept of I/O Programming
Aim: To Study 1. Byte Stream 2. Character stream 3. Buffered Stream 4. Input and Output from command line.

Theory:

I/O Streams

- Byte Streams handle I/O of raw binary data.
- Character Streams handle I/O of character data, automatically handling translation to and from the local character set.
- Buffered Streams optimize input and output by reducing the no. of calls to the native API.
- Scanning and Formatting allows a program to read and write formatted text.
- I/O from the Command Line describes the Standard Streams and the Console object.
- Data Streams handle binary I/O of primitive data type and String values.
- Object Streams handle binary I/O of objects.

File I/O

Java provides a number of classes and methods that allow you to read and write files. In Java, all files are byte-oriented, and Java provides methods to read and write bytes from and to a file. However, Java allows you to wrap a byte-oriented file stream within a character-based object. Two of the most often-used stream classes are FileInputStream and FileOutputStream, which create byte streams linked to files. To open a file, you simply create an object of one of these classes, specifying the name of the file as an argument to the constructor. While both classes support additional, overridden constructors, the following are the forms that we will be using:

```
FileInputStream(String  fileName)  throws  FileNotFoundException
FileOutputStream(String fileName) throws FileNotFoundException
```

Here, fileName specifies the name of the file that you want to open. When you create an input stream, if the file does not exist, then FileNotFoundException is thrown. For output streams, if

the file cannot be created, then FileNotFoundException is thrown.When an output file is opened, any preexisting file by the same name is destroyed.When you are done with a file, you should close it by calling close( ). It is defined by both FileInputStream and FileOutputStream, as shown here:

void close( ) throws IOException

To read from a file, you can use a version of read( ) that is defined within FileInputStream. The one that we will use is shown here: int read( ) throws IOException Each time that it is called, it reads a single byte from the file and returns the byte as an integer value. read( ) returns –1 when the end of the file is encountered. It can throw an IOException.
To write to a file, you will use the write( ) method defined by FileOutputStream. Its simplest form is shown here:

void write(int byteval) throws IOException

Basic input and output classes

The java.io package contains a fairly large number of classes that deal with Java input and output.

Most of the classes consist of:

1. Byte streams that are subclasses of InputStream or OutputStream
2. Character streams that are subclasses of Reader and Writer

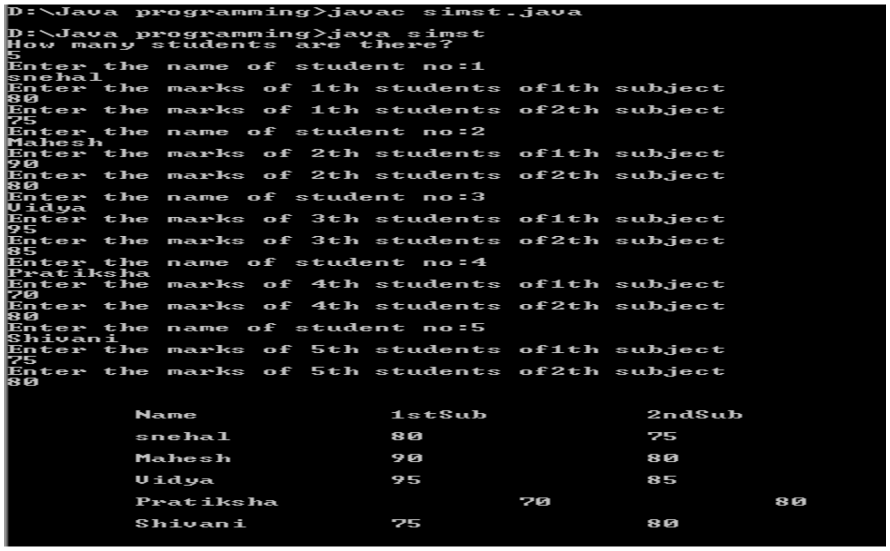
The Reader and Writer classes read and write 16-bit Unicode characters. InputStream reads 8-bit bytes, while OutputStream writes 8-bit bytes. As their class name suggests, ObjectInputStream and ObjectOutputStream transmit entire objects. ObjectInputStream reads objects; ObjectOutputStream writes objects.

Sample Program:

```
import    java.io.*;
  class simst
  {
          public static void main(String[] args) throws IOException
          {
                  BufferedReader br= new BufferedReader(new InputStreamReader(System.in));
                  String snm[]=new String[100];
                  String a;
                  int n;
                    Int m[][]=new int[10][20];
                    System.out.println("How many students are there?");
                    a=br.readLine();
                   n=Integer.parseInt(a);
                  for(int i=1;i<=n;i++)
                  {
                  System.out.println("Enter      the      name      of      student      no:"+i);
                  snm[i]=br.readLine();
                  for(int j=1;j<=2;j++)
```

```
                {
                System.out.println("Enter the marks of "+i+"th students of"+j+"th subject");
                 a=br.readLine();
                m[i][j]=Integer.parseInt(a);

                }
        }
                System.out.println("\n-------------------------------------------------------------------");
                System.out.println("\tName\t\t1stSub\t\t2ndSub");
                System.out.println("-------------------------------------------------------");
                for(int i=1;i<=n;i++)
                {
                    System.out.print("\t"+snm[i]);
                    for(int j=1;j<=2;j++)
                  {
                         System.out.print("\t\t"+m[i][j]);
                  }
                         System.out.println("\n");
                }
                System.out.println("--------------------------------------");
            }
        }
```

Output:



```
D:\Java programming>javac simst.java

D:\Java programming>java simst
How many students are there?
5
Enter the name of student no:1
snehal
Enter the marks of 1th students of1th subject
80
Enter the marks of 1th students of2th subject
75
Enter the name of student no:2
Mahesh
Enter the marks of 2th students of1th subject
90
Enter the marks of 2th students of2th subject
80
Enter the name of student no:3
Vidya
Enter the marks of 3th students of1th subject
95
Enter the marks of 3th students of2th subject
85
Enter the name of student no:4
Pratiksha
Enter the marks of 4th students of1th subject
70
Enter the marks of 4th students of2th subject
80
Enter the name of student no:5
Shivani
Enter the marks of 5th students of1th subject
75
Enter the marks of 5th students of2th subject
80

        Name            1stSub          2ndSub

        snehal          80              75

        Mahesh          90              80

        Vidya           95              85

        Pratiksha               70              80

        Shivani         75              80
```

Problem Statement:
Take file name as input to your program through command line, if file exists the open and display contents of the file. After displaying contents of file ask user – 1.do you want to add the data at the end of file or 2.replace specified text in file by other text. Based on user’s response, then accept data from user and append it to file. If file in not existing then create a fresh new-file and store user data into it. Also. User should type exit on new line to stop the program. Do this program using Character stream classes.

Conclusion: Thus we studied different Input and output stream.

Experiment No.08

Title : Implementation of GUI using AWT	
Aim : Implementation of GUI using AWT	

Theory :

The Java *Abstract Windowing Toolkit* (AWT) provides numerous classes that support window program development. These classes are used to create and organize windows, implement GUI components, handle events, draw text and graphics, perform image processing, and obtain access to the native Windows implementation.

The Component class is the superclass of the set of AWT classes that implement graphical user interface controls. These components include windows, dialog boxes, buttons, labels, text fields, and other common GUI components. The Component class provides a common set of methods that are used by all these subclasses. These methods include methods for handling events and working with images, fonts, and colors. More than 70 methods are implemented by this class. It is a good idea to browse the API pages of the Component class to get a feel for the kinds of methods that are available. You don't have to worry about learning them now.

The Container Class

The Container class is a subclass of the Component class that is used to define components that have the capability to contain other components. It provides methods for adding, retrieving, displaying, counting, and removing the components that it contains. It provides the deliverEvent() method for forwarding events to its components. The Container class also provides methods for working with layouts. The layout classes control the layout of components within a container. The Container class has two major subclasses: Window and Panel. Window provides a common superclass for application main windows (Frame objects) and Dialog windows. The Panel class is a generic container that can be displayed within a window. It is subclassed by the java.applet.Applet class
as the base class for all Java applets.

The Window Class

The Window class provides an encapsulation of a generic Window object. It is subclassed by Frame and Dialog to provide the capabilities needed to support application main windows and dialog boxes. The Window class contains a single constructor that creates a window that has a frame window as its parent. The parent frame window is necessary because only objects of the Frame class or its subclasses contain the functionality needed to support the implementation of an independent application window.

A Window object does not have a border or a menu bar when it is created. In this state it may be used to implement a pop-up window. The default layout for a Window object is BorderLayout.

Frame

The Frame class is used to provide the main window of an application. It is a subclass of Window that supports the capabilities to specify the icon, cursor, menu bar, and title. Because it implements the MenuContainer interface, it is capable of working with MenuBar objects.

Frame provides two constructors: a default parameterless constructor that creates an untitled frame window and a constructor that accepts a string argument to be used as the frame window's title. The second constructor is typically used.

The GridLayout Class

The GridLayout class is used to lay out the components of a Container object in a grid where all components are the same size. The GridLayout constructor is used to specify the number of rows and columns of the grid.

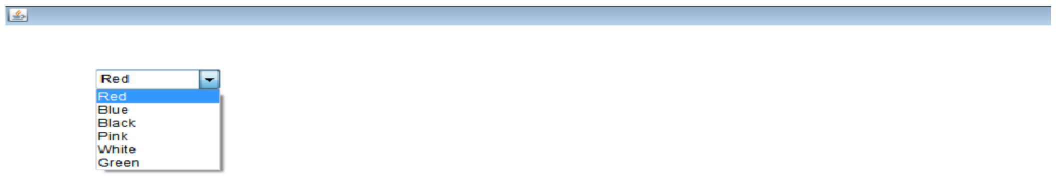
The GridBagLayout Class

The GridBagLayout class lays out the components of a Container object in a grid-like fashion, where some components may occupy more than one row or column. The GridBagConstraints class is used to identify the positioning parameters of a component that is contained within an object that is laid out using GridBagLayout. The Insets class is used to specify the margins associated with an object that is laid out using a GridBagLayout object. Refer to the API description of the GridBagLayout class for more information on how to use this layout.
The user communicates with window programs by performing actions such as clicking a mouse button or pressing a key on the keyboard. These actions result in the generation of Event objects. The process of responding to the occurrence of an event is known as event handling. Window programs are said to be event driven because they operate by performing actions in response to events

Sample Program:

```
import java.awt.*;
public class ChoiceDemo
{
  ChoiceDemo()
  {
    Frame choice_f= new Frame();
    Choice obj=new Choice();
    obj.setBounds(80,80, 100,100);
    obj.add("Red");
    obj.add("Blue");
    obj.add("Black");
    obj.add("Pink");
    obj.add("White");
    obj.add("Green");
    choice_f.add(obj);

 choice_f.setSize(400,400);

 choice_f.setLayout(null);
    choice_f.setVisible(true);
  }
  public static void main(String args[])
  {
    new ChoiceDemo();
  }
}
```

Output:



Problem Statement:

Write a GUI based program to create a User registration using AWT.

Conclusion: Thus we have studied and implement GUI using AWT.

Experiment No.9

Title: Implementation of GUI using SWING	
Aim: Implementation of GUI using SWING.	

Theory:

Difference between AWT and Swing

There are many differences between java awt and swing that are given below.

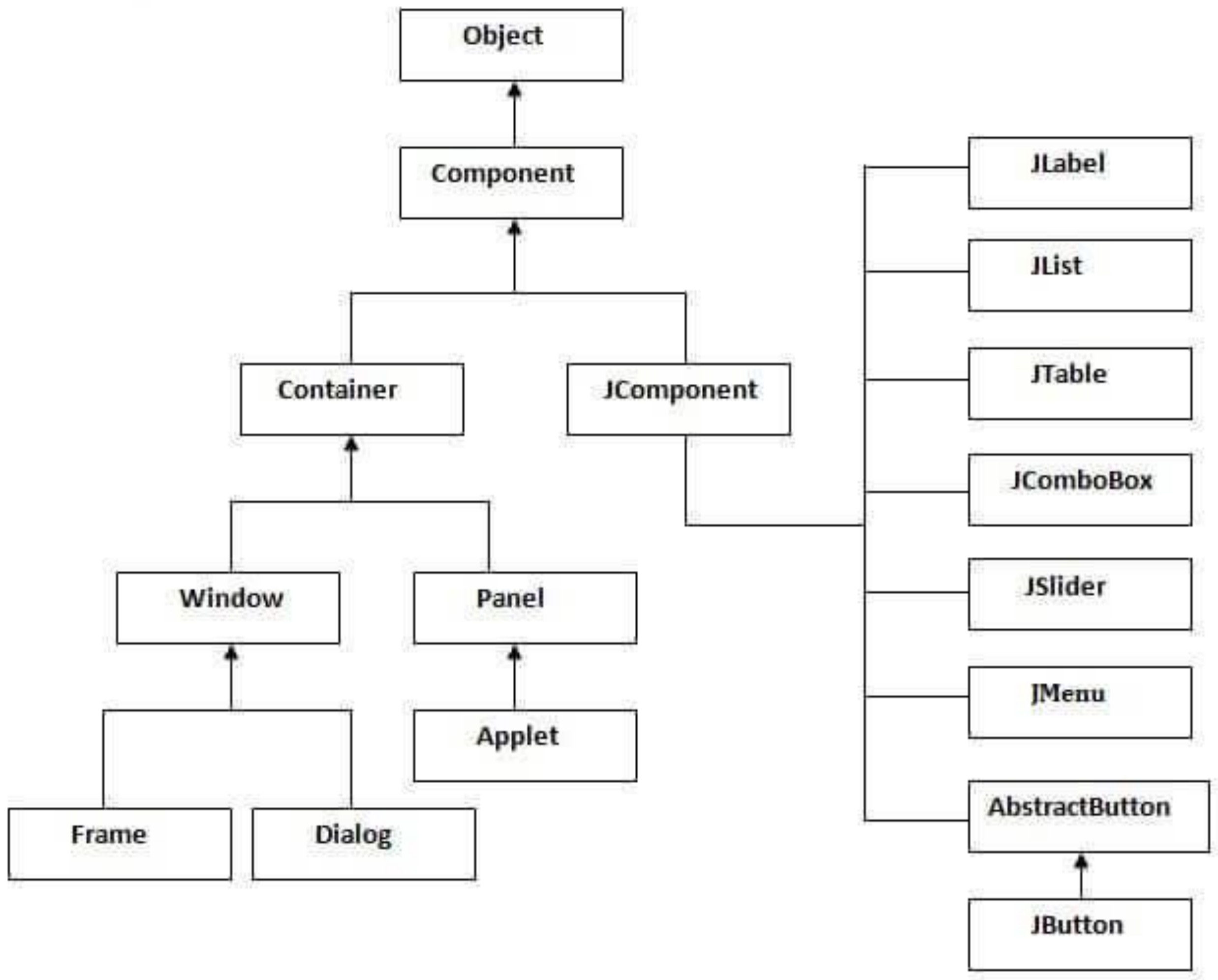
No.	Java AWT	Java Swing
1)	AWT components are **platform-dependent**.	Java swing components are **platform-independent**.
2)	AWT components are **heavyweight**.	Swing components are **lightweight**.
3)	AWT **doesn't support pluggable look and feel**.	Swing **supports pluggable look and feel**.
4)	AWT provides **less components** than Swing.	Swing provides **more powerful components** such as tables, lists, scrollpanes, colorchooser, tabbedpane etc.
5)	AWT **doesn't follows MVC**(Model View Controller) where model represents data, view represents presentation and controller acts as an interface between model and view.	Swing **follows MVC**.

Features of the Java Foundation Classes	
Feature	**Description**
Swing GUI Components	Includes everything from buttons to split panes to tables.
Pluggable Look-and-Feel Support	Gives any program that uses Swing components a choice of look and feel. For example, the same program can use either the Java or the Windows look and feel. Many more look-and-feel packages are available from various sources. As of v1.4.2, the Java platform supports the GTK+ look and feel, which makes hundreds of existing look and feels available to Swing programs.
Accessibility API	Enables assistive technologies, such as screen readers and Braille displays, to get information from the user interface.
Java 2D API	Enables developers to easily incorporate high-quality 2D graphics, text, and images in applications and applets. Java 2D includes extensive APIs for generating and sending high-quality output to printing devices.
Drag-and-Drop Support	Provides the ability to drag and drop between Java applications and native applications.
Internationalization	Allows developers to build applications that can interact with users worldwide in their own languages and cultural conventions. With the input method framework developers can build applications that accept text in languages that use thousands of different characters, such asJapanese, Chinese, or Korean.

JFC features apply to Swing components.
The Swing API is powerful, flexible--and immense. In release 1.4 of the Java platform, the SwingAPI has 17 public packages:

javax.accessibility	javax.swing.plaf	javax.swing.text.htm l
javax.swing	javax.swing.plaf.basic	javax.swing.text.pars er
javax.swing.border	javax.swing.plaf.metal	javax.swing.text.rtf
javax.swing.colorchooser	javax.swing.plaf.multi	javax.swing.tree
javax.swing.event	javax.swing.table	javax.swing.undo
javax.swing,filechooser	javax.swing.text	

The hierarchy of java swing API is given below.



Commonly used Methods of Component class

The methods of Component class are widely used in java swing that are given below.

Method	**Description**
public void add(Component c)	add a component on another component.
public void setSize(int width,int height)	sets size of the component.
public void setLayout(LayoutManager m)	sets the layout manager for the component.

public void setVisible(boolean b)	sets the visibility of the component. It is by default false.

Event and Listener (Java Event Handling)

Changing the state of an object is known as an event. For example, click on button, dragging mouse etc. event classes and Listener interfaces for event handling.

Java Event classes and Listener interfaces

Event Classes	**Listener Interfaces**
ActionEvent	ActionListener
MouseEvent	MouseListener and MouseMotionListener
MouseWheelEvent	MouseWheelListener
KeyEvent	KeyListener
ItemEvent	ItemListener
TextEvent	TextListener
AdjustmentEvent	AdjustmentListener
WindowEvent	WindowListener
ComponentEvent	ComponentListener
ContainerEvent	ContainerListener
FocusEvent	FocusListener

Code:

```
import      java.awt.*;import javax.swing.*;

/*<applet code="swingdemo" width=200 height=200>

</applet>*/

public class swingdemo extends JApplet

{         public void init()

          { JLabel l1=new JLabel("My swing program"); ImageIcon i1=new ImageIcon("images.gif");
                    JButton b1=new JButton(i1);
                    add(l1);
                    add(b1);
           }
          public void paint(Graphics g)
          {
                    g.drawString("my swing program",30,30);
          }
    }
```

Problem Statement:
Develop a Swing GUI based standard calculator program. Use event handling, Layout of swing package.

Conclusion: Thus we have studied and implemented GUI with SWING.

Experiment No.10

Title: Implementing the concept of Mouse Event	
Aim: To Study how to handle different mouse event with AWT controls.	

Theory:

The MouseEvent Class

There are eight types of mouse events. The **MouseEvent** class defines the following integer constants that can be used to identify them:

MOUSE_CLICKED The user clicked the mouse.
MOUSE_DRAGGED The user dragged the mouse.
MOUSE_ENTERED The mouse entered a component.
MOUSE_EXITED The mouse exited from a component.
MOUSE_MOVED The mouse moved.
MOUSE_PRESSED The mouse was pressed.
MOUSE_RELEASED The mouse was released.
MOUSE_WHEEL The mouse wheel was moved (Java 2, v1.4).

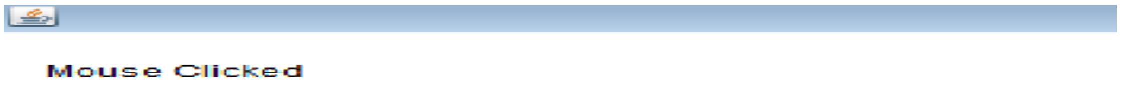
MouseEvent is a subclass of **InputEvent**. Here is one of its constructors.
MouseEvent(Component *src*, int *type*, long *when*, int *modifiers*,int *x*, int *y*, int *clicks*, boolean *triggersPopup*)

Here, *src* is a reference to the component that generated the event. The type of the event is specified by *type*.
The most commonly used methods in this class are **getX( )** and **getY( )**. These return the X and Y coordinates of the mouse when the event occurred. Their forms are shown here:
int getX( )
int getY( )

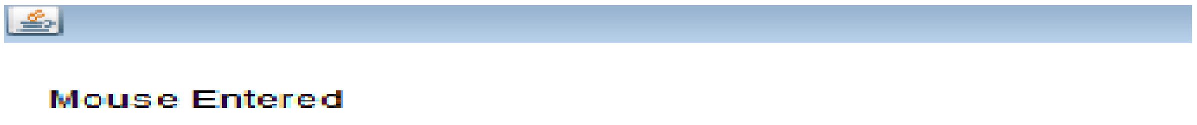
//program for the mouselistener

```
import          java.awt.*;
import      java.applet.*;
import java.awt.event.*;


/*<applet code="ms" height=20 width=20>
</applet>*/


 public class ms extends Applet implements MouseMotionListener, MouseListener
{
        String   msg;
        int x,y;
        public void init()
        {

        addMouseMotionListener(this);
                addMouseListener(this);
        }
        public void mouseClicked(MouseEvent m)
        {       x=0;
                y=40;
                msg="Mouse Clicked";
                showStatus("mouse clicked at "+m.getX()+","+m.getY());
                repaint();
        }
        public void mouseEntered(MouseEvent m)
        {
                x=0;
                y=40;
                msg="Mouse  Entered";
                showStatus("mouse clicked at "+m.getX()+","+m.getY());
                //showStatus("");
                repaint();
        }

        public void mouseDragged(MouseEvent m)
        {
                x=m.getX();
                y=m.getY();
                msg="#";
                showStatus("Dragging mouse at "+x+","+y);
                        repaint();
        }
        public void mouseExited(MouseEvent m)
        {
                x=0;
                y=20;

                msg="mouse    Exited";
                showStatus("");



                repaint();
        }
        public void mouseReleased(MouseEvent m)
        {
                x=0;
                y=20;
                msg="mouse released ";
                showStatus("");
                repaint();
        }
        public void mousePressed(MouseEvent m)
        {


                x=0;
                y=40;
                msg="mouse pressed.";
                showStatus("Mouse     pressed     at     "+m.getX()+","+m.getY());
                repaint();
        }
        public void mouseMoved(MouseEvent m)
        {
                x=0;
                y=40;

                msg="Mouse  Moved";
                showStatus("");
                repaint();
        }
        public void paint(Graphics g)
        {
                g.drawString(msg,x,y);
        }
}
```

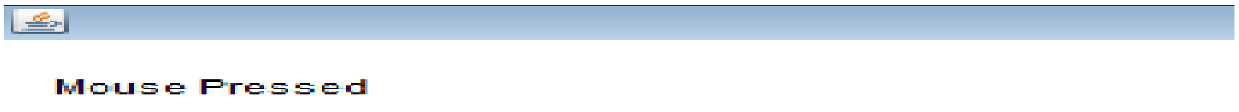
Output:



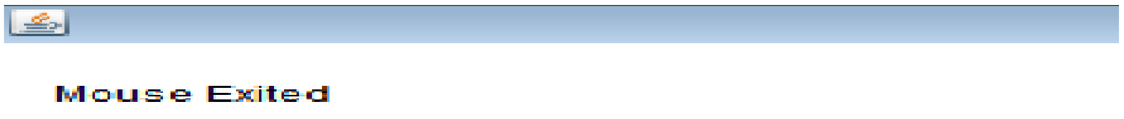
```
Mouse Clicked
```



```
Mouse Entered
```



```
Mouse Pressed
```



```
Mouse Exited
```

Conclusion: Thus we studied how to handle mouse event generated from AWT controls

Experiment No.11

Title: Implementing the concept of Multithreading using Thread class	
Aim: To study what is multithreading and how to handle multiple tasks simultaneously.	

Theory:

A ***thread*** is the flow of execution of a single set of program statements. ***Multithreading*** consists of multiple sets of statements which can be run in parallel. With a single processor only one thread can run at a time but strategies are used to make it appear as if the threads are running simultaneously. Depending on the operating system ***scheduling method***, either timeslicing or interrupt methods will move the processing from one thread to another.

Serialization is the process of writing the state of an object to a byte stream. It can be used to save state variables or to communicate through network connections.

The Thread Class

The ***Thread Class*** allows ***multitasking*** (ie running several tasks at the same time) by instantiating (ie creating) many threaded objects, each with their own run time characteristics. Tasks that slow the processor can be isolated to prevent apparent loss of GUI response. One way to create threads is to ***extend*** the Thread class and override the ***run()*** method such as:

```
class HelloThread extends Thread
{
  public void run()
  {
    for        int        x=0;x<100;        ++x)
      System.out.print(" Hello ");
  }
}
```

Multithreading:

Multithreading is a technique that allows a program or a process to execute many tasks concurrently (at the same time and parallel). It allows a process to run its tasks in parallel mode on a single processor system

In the multithreading concept, several multiple lightweight processes are run in a single process/ task or program by a single processor. For Example, When you use a **word processor** you perform a many different tasks such as **printing, spell checking** and so on. Multithreaded software treats each process as a separate program.

Advantages of multithreading over multitasking:

- Reduces the computation time.
- Improves performance of an application.

- Threads share the same address space so it saves the memory.
- Context switching between threads is usually less expensive than between processes.
- Cost of communication between threads is relatively low.

Statement:

Write a program that bounces a blue ball inside a JPanel. The ball should begin moving with a mousePressed event. When the ball hits the edge of the JPanel, it should bounce off the edge and continue in the opposite direction. The ball should be updated using a Runnable.

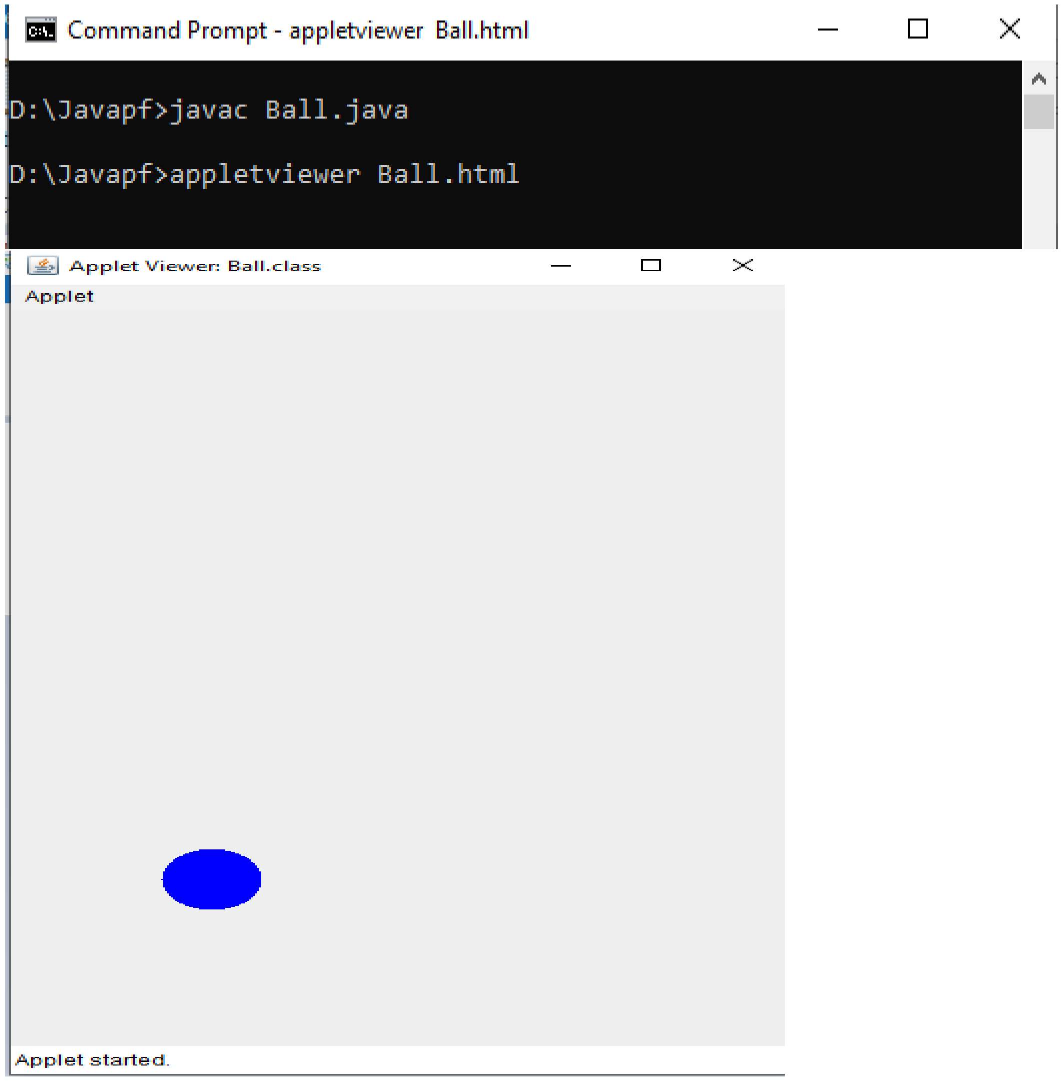
Program:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Ball extends JApplet implements Runnable
{
private Thread blueBall;
private boolean xUp, yUp, bouncing;
private int x, y, xDx, yDy;
private final int MAX_X = 200, MAX_Y = 200;

public void init()
{
//initialize values
xUp  = false;
yUp  = false;
xDx = 1;
yDy = 1;
bouncing = false;

//let ball applet be its own MouseListener
addMouseListener(

new MouseListener()
{
public void mousePressed( MouseEvent event )
{
createBall( event );
//delegate call to ball starter
}
public void mouseExited( MouseEvent event ) {}
public void mouseClicked( MouseEvent event ) {}
public void mouseReleased( MouseEvent event ) {}
public void mouseEntered( MouseEvent event ) {}
}
);
setSize( MAX_X, MAX_Y ); // set size of Applet
}

//creates a ball and sets it in motion if
//no ball exists
private void createBall( MouseEvent event )
{
if ( blueBall == null )
{
x = event.getX();
y = event.getY();
blueBall = new Thread( this );

bouncing = true; // start ball's bouncing
blueBall.start();
}
}
//called if applet is closed. by setting blueBall to null, threads will be ended.
public void stop()
{
blueBall = null;
}

// draws ball at current position
public void paint( Graphics g )
{
super.paint( g );

if ( bouncing )
{
g.setColor( Color.blue );
g.fillOval( x, y, 50, 50 );
}
}
// action to perform on execution, bounces ball perpetually until applet is closed.
public void run()
{

while ( true )
{
//sleep for a random interval
try
{
```

```
blueBall.sleep( 2000 );
}

//process InterrupedException during sleep
catch( InterruptedException exception )
{
System.err.println( exception.toString() );
}

// determine new x position
if ( xUp == true )
x += xDx;
else
x -= xDx;

//determine new y position
if ( yUp == true )
y += yDy;
else
y -= yDy;

// randomize variables for creating next move
if ( y <= 0 )
{
yUp = true;
yDy = ( int ) ( Math.random() * 5 + 2 );
yUp = false;
}
if ( x <= 0 )
{
xUp = true;
xDx = ( int ) ( Math.random() * 5 + 2 );
}
else if ( x >= MAX_X - 10 )
{
xUp = false;
xDx = ( int ) ( Math.random() * 5 + 2 );
}
repaint();
}
}
}
```

```
Applet code:-
<html>
<body>
<applet code="Ball.class" width="300" height="300">

</applet>
</body>
</html>
```

Output:-



```
D:\Javapf>javac Ball.java

D:\Javapf>appletviewer Ball.html
```

Command Prompt - appletviewer Ball.html
Applet Viewer: Ball.class
Applet
Applet started.

Conclusion: Thus we studied implementing the concept of Multithreading using Thread class.

Experiment No.12

Title: Implementing the concept of Multithreading using Runnable interface
Aim: To study what is multithreading and how to handle multiple tasks simultaneously.

Theory:

Implementing java.lang.Runnable :

Implementing the Runnable interface gives you a way to extend from any class you like, but still define behavior that will be run by a separate thread. It looks like this:

```
class MyRunnable implements Runnable
   {
         public void run()
          {
                System.out.println("Important job running in MyRunnable");
          }
   }
```

Regardless of which mechanism you choose, you've now got yourself some code that can be run by a thread of execution. So now let's take a look at *instantiating* your thread-capable class, and then we'll figure out how to actually get the thing *running*.

However if you need to inherit from another class as well, you can ***implement*** a Runnable interface instead and write the required ***run()*** method.

Thread object methods are used on instantiated thread objects to control the thread appropriately. These methods include ***currentThread(), getName(), getPriority(), sAlive(), join(), run(), setName(string), setPriority(int), sleep(longInt)*** and ***start()***.

Synchronization

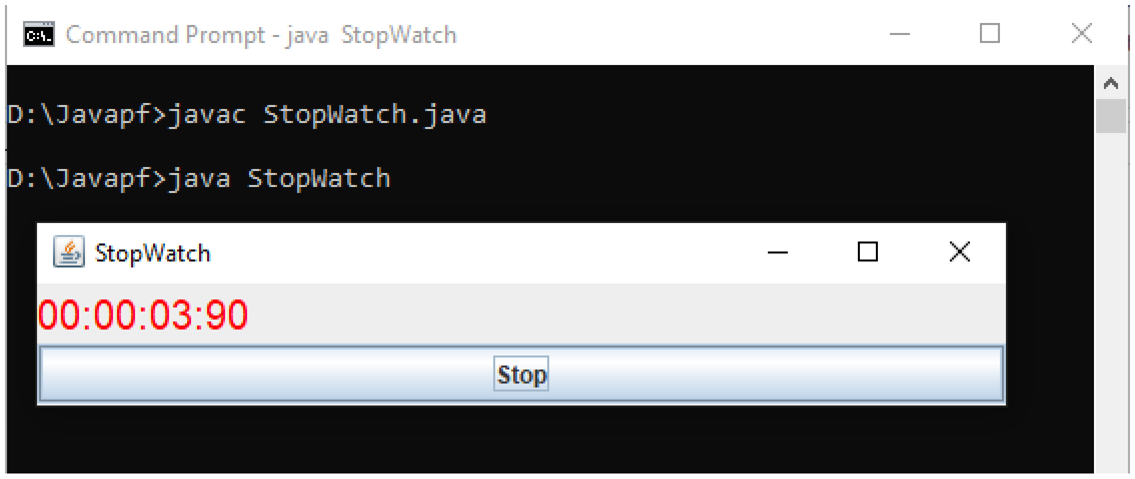
Thread synchronization is required when two or more threads need to share a resource. A ***monitor*** (aka semaphore) is an object that provides a muually exclusive lock (mutex). Java provides the ***synchronized*** keyword as the key that locks/unlocks an object. It can be used as a class or method modifier or as a statement (very localized). Any long running method should not be synchonized as it would become a traffic bottleneck. To guarantee that a variable is threadsafe (ie. not shared between threads) it can be marked as ***volatile***.

Statement: Create Stop Watch with Swing GUI and Multithreading.

Program:

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
import java.text.DecimalFormat;
import java.text.NumberFormat;

public class StopWatch extends JFrame implements ActionListener,Runnable
{
JLabel disp; JButton btn; boolean stop=false; int i,j,k,l;
public StopWatch()
{
disp=new JLabel();
btn=new JButton("Start");
disp.setFont(new Font("Helvetica",Font.PLAIN,20));
disp.setBackground(Color.cyan);
disp.setForeground(Color.red);
Container c=getContentPane();
c.setLayout(new GridLayout(2,1));
c.add(disp);
c.add(btn);
btn.addActionListener(this);
}
public void run()
{
for(i=0;;i++)
{
for(j=0;j< 60;j++)
{
for(k=0;k< 60;k++)
{
for(l=0;l< 100;l++)
{

if(stop)
{
break;
}
NumberFormat nf = new DecimalFormat("00");
disp.setText(nf.format(i)+":"+nf.format(j)+":"+nf.format(k)+":"+nf.format(l));
try
{
Thread.sleep(10);
}
catch(Exception e){}
}         }
}
}
}
public void actionPerformed(ActionEvent ae)
{
Thread t=new Thread(this);
if(ae.getActionCommand().equals("Start"))
{
t.start(); btn.setText("Stop");


}
else
{
stop=true;
}
}


public static void main(String[] args)
{
StopWatch s=new StopWatch();
s.setSize(500,100);
s.setVisible(true);
s.setTitle("StopWatch");
s.setDefaultCloseOperation(EXIT_ON_CLOSE);
}
}
```

Output:-



```
D:\Javapf>javac StopWatch.java

D:\Javapf>java StopWatch
```

Conclusion: Thus we implementing the concept of Multithreading using Runnable interface.

Experiment No.13

Title: Implementation Socket Programming Client -Server	
Aim: Implementation of Socket Programming-Iterative Server Implementation.	

Theory:

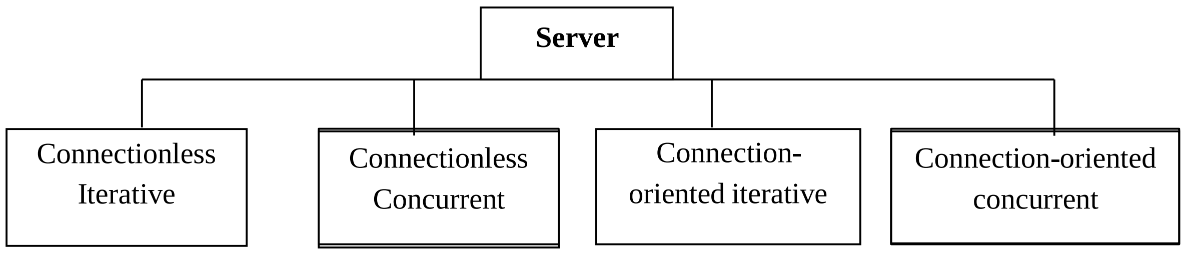
Both clients & servers can run in concurrent mode:

- **Concurrent in Clients** :-

Clients can run on a machine either iteratively on concurrently. Running clients iteratively means running them one by one, one client must start, run & terminate before the m/c can start another client. Most computers today, however, allow concurrent clients, i.e two or more clients can run at the same time.

- **Concurrency in servers** :-

An iterative server can process only one request at a time, it receive a request, processes it, & sends the response to the response to the requestor before it handles another request. A concurrent server, on the other hand, can process many requests at the same time & thus can share its time between many respects. The servers use either UDP, a connectionless transport layer protocol or TCP, a connection-oriented transport layer protocol & the service method.
Theoretically, we can have four types of servers: Connectionless iterative, Connectionless Concurrent, Connection-oriented iterative & connection-oriented concurrent.


Server
Connectionless Iterative
Connectionless Concurrent
Connection-oriented iterative
Connection-oriented concurrent

- **Connectionless Iterative Server** :-

The servers that use UDP are normally iterative, which as we have said, means that the server processes one request at a time. A server gets the request in a datagram from UDP, processes the request, & gives the response to UDP to send to the client.
The server plays no attention to the other datagram's. Theses datagram's are stored in a queue, waiting for service. They could all be from many clients. In either case they are processed one by one in order of arrival.

Statement: Implement Socket programming of client-sever.

Program:

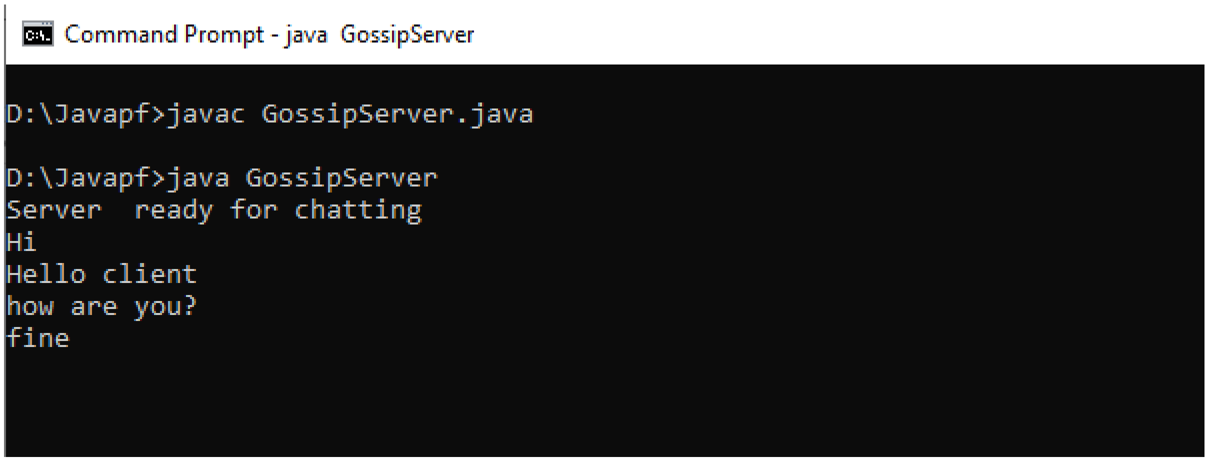
For Server:

```
import java.io.*;
import java.net.*;
public class GossipServer
{
  public static void main(String[] args) throws Exception
  {


 ServerSocket sersock = new ServerSocket(3000);
      System.out.println("Server  ready for chatting");
      Socket sock = sersock.accept( );
                              // reading from keyboard (keyRead object)
      BufferedReader keyRead = new BufferedReader(new InputStreamReader(System.in));
                                                              // sending to client (pwrite object)
      OutputStream ostream = sock.getOutputStream();
      PrintWriter pwrite = new PrintWriter(ostream, true);

                              // receiving from server ( receiveRead  object)
      InputStream istream = sock.getInputStream();
      BufferedReader receiveRead = new BufferedReader(new InputStreamReader(istream));

      String receiveMessage, sendMessage;
      while(true)
      {
        if((receiveMessage = receiveRead.readLine()) != null)
        {
           System.out.println(receiveMessage);
        }
        sendMessage = keyRead.readLine();
        pwrite.println(sendMessage);
        pwrite.flush();
      }
    }
}
```

For Client:

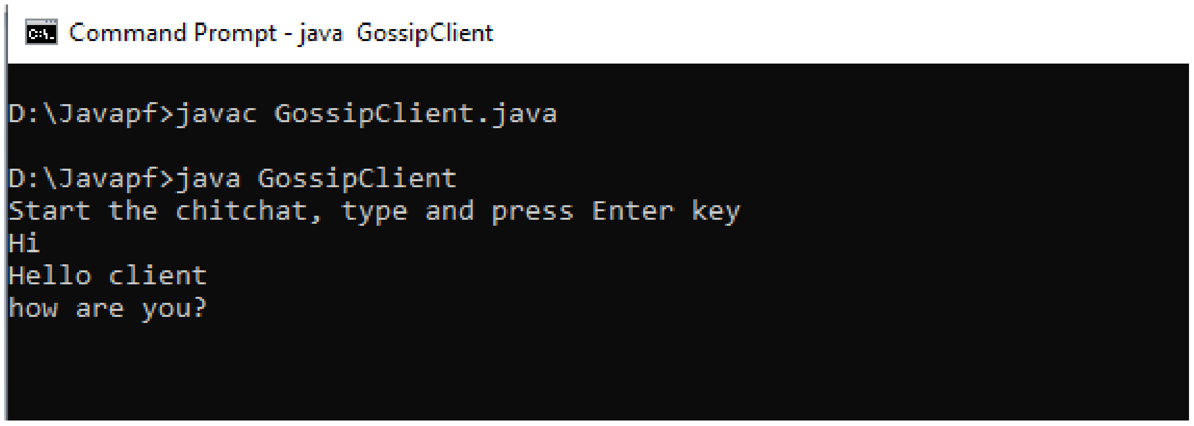
```
        import java.io.*;
import java.net.*;
public class GossipClient
{
  public static void main(String[] args) throws Exception
  {
     Socket sock = new Socket("127.0.0.1", 3000);
                              // reading from keyboard (keyRead object)
     BufferedReader keyRead = new BufferedReader(new InputStreamReader(System.in));
                             // sending to client (pwrite object)
     OutputStream ostream = sock.getOutputStream();
     PrintWriter pwrite = new PrintWriter(ostream, true);

                             // receiving from server ( receiveRead  object)
     InputStream istream = sock.getInputStream();
     BufferedReader receiveRead = new BufferedReader(new InputStreamReader(istream));

     System.out.println("Start the chitchat, type and press Enter key");

     String receiveMessage, sendMessage;
     while(true)
     {
        sendMessage = keyRead.readLine();  // keyboard reading
        pwrite.println(sendMessage);       // sending to server
        pwrite.flush();                    // flush the data
        if((receiveMessage = receiveRead.readLine()) != null) //receive from server
        {
            System.out.println(receiveMessage); // displaying at DOS prompt
        }
      }
    }
}
```

Output:-



Command Prompt - java GossipServer

```
D:\Javapf>javac GossipServer.java

D:\Javapf>java GossipServer
Server  ready for chatting
Hi
Hello client
how are you?
fine
```



Command Prompt - java GossipClient

```
D:\Javapf>javac GossipClient.java

D:\Javapf>java GossipClient
Start the chitchat, type and press Enter key
Hi
Hello client
how are you?
```

Conclusion: Thus we have studied and implemented the socket programming.

Experiment No.14

Title: Implementing the concept of Database Programming
Aim: To study JDBC and ODBC connectivity in Java.

Theory:

JDBC Introduction

The JDBC API is a Java API that can access any kind of tabular data, especially data stored in a Relational Database.

JDBC helps you to write java applications that manage these three programming activities:

1. Connect to a data source, like a database
2. Send queries and update statements to the database
3. Retrieve and process the results received from the database in answer to your query

   JDBC includes four components:

1. The JDBC API provides programmatic access to relational data from the Java™ programming language. Using the JDBC API, applications can execute SQL statements, retrieve results, and propagate changes back to an underlying data source. The JDBC API canalso interact with multiple data sources in a distributed, heterogeneous environment.

   The JDBC API is part of the Java platform, which includes the *Java Standard Edition* (Java SE) and the *Java Enterprise Edition* (Java EE). The JDBC 4.0 API is divided into two packages: java.sql and javax.sql. Both packages are included in the Java SE and Java EE platforms.
2. **JDBC Driver Manager —**
   The JDBC DriverManager class defines objects which can connect Java applications to a JDBC driver. DriverManager has traditionally been the backbone of the JDBC architecture. It is quite small and simple.
   JDBC Test Suite —
   The JDBC driver test suite helps you to determine that JDBC drivers will run your program. These tests are not comprehensive or exhaustive, but they do exercise many of the important features in the JDBC API.
3. **JDBC-ODBC Bridge —**
   The Java Software bridge provides JDBC access via ODBC drivers. Note that you need to load ODBC binary code onto each client machine that uses this driver. As a result, the ODBC driver is most appropriate on a corporate network where client installations are not a major problem, or for application server code written in Java in a three-tier architecture.

SELECT Statements

SQL is a language designed to be used with relational databases. There is a set of basic SQL commands that is considered standard and is used by all RDBMSs. For example, all RDBMSs use the SELECT statement.

A SELECT statement, also called a query, is used to get information from a table. It specifies one or more column headings, one or more tables from which to select, and some criteria for selection. The

RDBMS returns rows of the column entries that satisfy the stated requirements. A SELECT statement such as the following will fetch the first and last names of employees who have company cars:

```
SELECT First_Name, Last_Name
FROM Employees
WHERE Car_Number IS NOT NULL
```

The result set (the set of rows that satisfy the requirement of not having null in the Car_Number column) follows. The first name and last name are printed for each row that satisfiesthe requirement because the SELECT statement (the first line) specifies the columns First_Name andLast_Name. The FROM clause (the second line) gives the table from which the columns will be selected.

```
FIRST_NAME            LAST_NAME
------------            -------------

Axel                   Washington
Florence               Wojokowski
```

The following code produces a result set that includes the whole table because it asks for all of the columns in the table Employees with no restrictions (no WHERE clause). Note that SELECT * means "SELECT all columns."

```
SELECT *
FROM Employees
```

WHERE Clauses

The WHERE clause in a SELECT statement provides the criteria for selecting values. For example, in the following code fragment, values will be selected only if they occur in a row in which the column Last_Name begins with the string 'Washington'.

```
SELECT First_Name, Last_Name
FROM Employees
WHERE Last_Name LIKE 'Washington%'
```

The keyword LIKE is used to compare strings, and it offers the feature that patterns containing wildcards can be used. For example, in the code fragment above, there is a percent sign (%) at the end of 'Washington', which signifies that any value containing the string 'Washington' pluszero or more additional characters will satisfy this selection criterion. So 'Washington' or 'Washingtonian' would be matches, but 'Washing' would not be. The other wildcard used in LIKE clauses is an underbar (_), which stands for any one character. For example,

```
WHERE Last_Name LIKE 'Ba_man'
```

would match 'Batman', 'Barman', 'Badman', 'Balman', 'Bagman', 'Bamman', and so on.
The code fragment below has a WHERE clause that uses the equal sign (=) to compare numbers. It selects the first and last name of the employee who is assigned car 12.

```
SELECT First_Name, Last_Name

FROM            Employees
WHERE Car_Number = 12
```

The next code fragment selects the first and last names of employees whose employee number is greater than 10005:

```
SELECT First_Name, Last_Name
FROM Employees
WHERE Employee_Number > 10005
```

WHERE clauses can get rather elaborate, with multiple conditions and, in some DBMSs, nested conditions. This overview will not cover complicated WHERE clauses, but the following code fragment has a WHERE clause with two conditions; this query selects the first and last names of employees whose employee number is less than 10100 and who do not have a company car.

```
SELECT First_Name, Last_Name
FROM Employees
WHERE Employee_Number < 10100 and Car_Number IS NULL
```

Common SQL Commands

SQL commands are divided into categories, the two main ones being Data Manipulation Language (DML) commands and Data Definition Language (DDL) commands. DML commands deal with data, either retrieving it or modifying it to keep it up-to-date. DDL commands create or change tables and other database objects such as views and indexes.
A list of the more common DML commands follows:

- SELECT — used to query and display data from a database. The SELECT statement specifies which columns to include in the result set. The vast majority of the SQL commands used in applications are SELECT statements.
- INSERT — adds new rows to a table. INSERT is used to populate a newly created table or to add a new row (or rows) to an already-existing table.
- DELETE — removes a specified row or set of rows from a table
- UPDATE — changes an existing value in a column or group of columns in a table

The more common DDL commands follow:

- CREATE TABLE — creates a table with the column names the user provides. The user also needs to specify a type for the data in each column. Data types vary from one RDBMS to another, so a user might need to use metadata to establish the data types used by a particular database. CREATE TABLE is normally used less often than the data manipulation commands because a table is created only once, whereas adding or deleting rows or changingindividual values generally occurs more frequently.
- DROP TABLE — deletes all rows and removes the table definition from the database. A JDBC API implementation is required to support the DROP TABLE command as specified

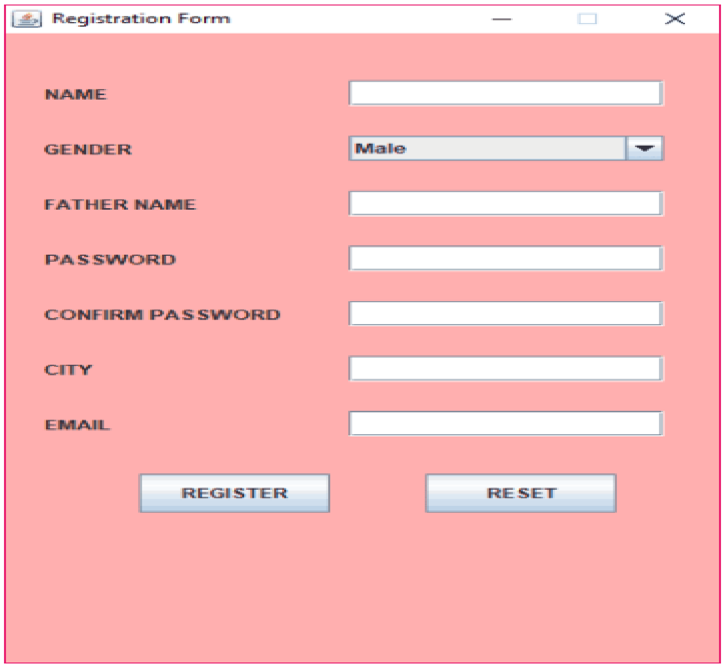
by SQL92, Transitional Level. However, support for the CASCADE and RESTRICT options

of DROP TABLE is optional. In addition, the behavior of DROP TABLE is implementation-defined when there are views or integrity constraints defined that reference the table being dropped.

- ALTER TABLE — adds or removes a column from a table. It also adds or drops table constraints and alters column attributes

Problem Statement:

Write a GUI based program to create a student registration and Login. Store Registration data in Database and take Login information from Database.



Conclusion: Hence we have studied the program is used to deal with the database.

Experiment No.15

Title: Implementing the concept of Map Classes
Aim: To study different Map Classes in java.

Theory:
A *map* is an object that stores associations between keys and values, or *key/value pairs*.

Given a key, you can find its value. Both keys and values are objects. The keys must be unique, but the values may be duplicated. Some maps can accept a **null** key and **null** values, others cannot.

Several classes provide implementations of the map interfaces. The classes that can be used for maps are summarized here:

Class	**Description**
AbstractMap	Implements most of the **Map** interface.
HashMap	Extends **AbstractMap** to use a hash table.
TreeMap	Extends **AbstractMap** to use a tree.
WeakHashMap	Extends **AbstractMap** to use a hash table with weak keys.
LinkedHashMap	Extends **HashMap** to allow insertion-order iterations.
IdentityHashMap	Extends **AbstractMap** and uses reference equality when comparing documents.

The HashMap Class:
The **HashMap** class uses a hash table to implement the **Map** interface. This allows the execution time of basic operations, such as **get( )** and **put( )**, to remain constant even for large sets.

The following constructors are defined:
HashMap( )
HashMap(Map *m*)
HashMap(int *capacity*)
HashMap(int *capacity*, float *fillRatio*)

The first form constructs a default hash map. The second form initializes the hash map by using the elements of *m*. The third form initializes the capacity of the hash map to *capacity*. The fourth form initializes both the capacity and fill ratio of the hash map by using its arguments. The meaning of capacity and fill ratio is the same as for **HashSet**, described earlier.

HashMap implements **Map** and extends **AbstractMap**. It does not add any methods of its own

You should note that a hash map does *not* guarantee the order of its elements.
Therefore, the order in which elements are added to a hash map is not necessarily the order in which they are read by an iterator. The following program illustrates **HashMap**. It maps names to account balances.

Sample Program:

```
import java.util.*;
class HashMapDemo {
public static void main(String args[]) {
// Create a hash map
HashMap hm = new HashMap();
// Put elements to the map
hm.put("John Doe", new Double(3434.34));
hm.put("Tom Smith", new Double(123.22));
hm.put("Jane Baker", new Double(1378.00));
hm.put("Todd Hall", new Double(99.22));
hm.put("Ralph Smith", new Double(-19.08));
// Get a set of the entries
Set set = hm.entrySet();
// Get an iterator
Iterator i = set.iterator();
// Display elements
while(i.hasNext()) {
Map.Entry me = (Map.Entry)i.next();
System.out.print(me.getKey() + ": ");
System.out.println(me.getValue());
}
System.out.println();
// Deposit 1000 into John Doe's account
double balance = ((Double)hm.get("John Doe")).doubleValue();
hm.put("John Doe", new Double(balance + 1000));
System.out.println("John Doe's new balance: " +
hm.get("John Doe"));
}
}
```

The LinkedHashMap Class:
Java 2, version 1.4 adds the **LinkedHashMap** class. This class extends **HashMap**. **LinkedHashMap** maintains a linked list of the entries in the map, in the order in which they were inserted. This allows insertion-order iteration over the map. That is, when iterating a **LinkedHashMap**, the elements will be returned in the order in which they were inserted. You can also create a **LinkedHashMap** that returns its elements in the order in which they were last accessed.

LinkedHashMap defines the following constructors.

LinkedHashMap( )
LinkedHashMap(Map *m*)
LinkedHashMap(int *capacity*)
LinkedHashMap(int *capacity*, float *fillRatio*)
LinkedHashMap(int *capacity*, float *fillRatio*, boolean *Order*)

The first form constructs a default **LinkedHashMap**. The second form initializes the **LinkedHashMap** with the elements from *m*. The third form initializes the capacity.
The fourth form initializes both capacity and fill ratio. The meaning of capacity and fill ratio are the same as for **HashMap**. The last form allows you to specify whether the elements will be stored in the linked list by insertion order, or by order of last access. If *Order* is **true**, then access order is used. If *Order* is **false**, then insertion order is used.

LinkedHashMap adds only one method to those defined by **HashMap**.
This method is **removeEldestEntry( )** and it is shown here.

protected boolean removeEldestEntry(Map.Entry *e*)

This method is called by **put( )** and **putAll( )**. The oldest entry is passed in *e*. By default, this method returns **false** and does nothing. However, if you override this method, then you can have the **LinkedHashMap** remove the oldest entry in the map. To do this, have your override return **true**.

Statement:

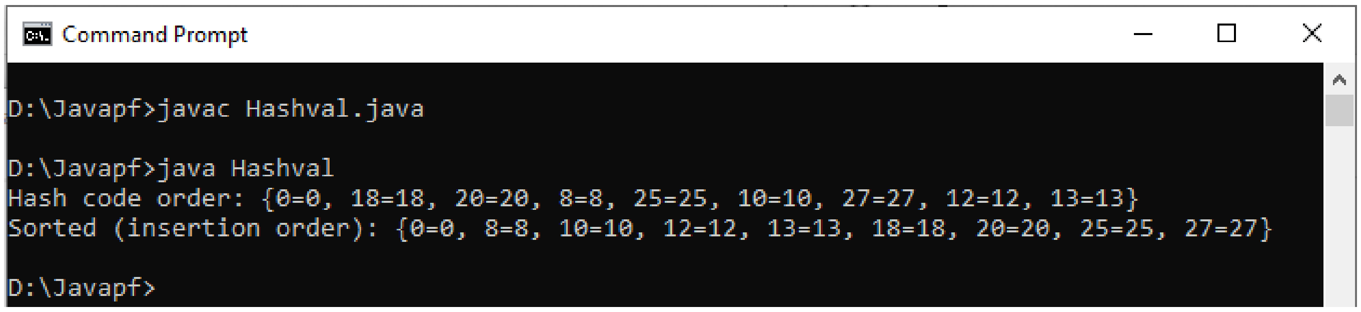
Fill a HashMap with key-value pairs. Print the results to show ordering by hash code. Extract the pairs, sort by key, and display the result.

Program:

```
import java.util.HashMap;
import java.util.Iterator;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.ListIterator;
import java.util.Map;
import java.util.Random;
import java.util.Set;

class HashMapper
{
    HashMap<Integer, String> map = new HashMap<Integer, String>();
    LinkedHashMap<Integer, String> linkedmap = new LinkedHashMap<Integer, String>();

    public void fillMap()
            {
            Random rand = new Random(42);
            int k;
            for (int i=0; i<10; i++)

{
                    k = rand.nextInt(i+20);
                    map.put(k, Integer.toString(k));
            }
            System.out.println("Hash code order: " + map);
    }

    public void remap()
            {
            Set<Integer> keyset = map.keySet();
            Iterator<Integer> it;
            int  temp;
            int smallest;
            int iterations = keyset.size();
            for (int i = 0; i < iterations; i++)
            {
                    it = keyset.iterator();
                    smallest = it.next();
                    it = keyset.iterator();
                    while(it.hasNext())
                    {
```

```
                    temp = it.next();
                    if (temp < smallest) smallest = temp;
                }
                linkedmap.put(smallest, map.get(smallest));
                keyset.remove(smallest);
            }
            System.out.println("Sorted (insertion order): " + linkedmap);
    }
}


public class Hashval
{
    public static void main(String[] args)
    {
            HashMapper hm = new HashMapper();
            hm.fillMap();
            hm.remap();
    }
}
```

Output:-



```
D:\Javapf>javac Hashval.java

D:\Javapf>java Hashval
Hash code order: {0=0, 18=18, 20=20, 8=8, 25=25, 10=10, 27=27, 12=12, 13=13}
Sorted (insertion order): {0=0, 8=8, 10=10, 12=12, 13=13, 18=18, 20=20, 25=25, 27=27}

D:\Javapf>
```

Conclusion: hence we have studied different Map Classes and implement it.