LAB MANUAL

Course: CSC241-Object Oriented Programming

Department of Computer Science

Java Learning Procedure

1) Stage **J** (**Journey inside-out the concept**)
2) Stage **a_1** (**Apply the learned**)
3) Stage **V** (**Verify the accuracy**)
4) Stage **a_2** (**Assess your work**)

COMSATS Institute of Information Technology (CIIT)
Islamabad

Table of Contents

LAB # 01

Statement Purpose:

Objective of this lab is to make students understand the difference between object oriented and procedural approaches to programming

Activity Outcomes:

The student will understand the advantages of using OOP

The student will understand the difference between procedural and object oriented approaches

Instructor Note:

The Students should have knowledge about structured programming.

1) Stage J (Journey)

Introduction

Procedural programming uses a list of instructions to tell the computer what to do step-by-step. Procedural programming relies on procedures, also known as routines or subroutines. A procedure contains a series of computational steps to be carried out.
Procedural programming is intuitive in the sense that it is very similar to how you would expect a program to work. If you want a computer to do something, you should provide step-by-step instructions on how to do it. It is, therefore, no surprise that most of the early programming languages are all procedural. Examples of procedural languages include Fortran, COBOL and C, which have been around since the 1960s and 70s.

Object-oriented programming, or OOP, is an approach to problem-solving where all computations are carried out using objects. An object is a component of a program that knows how to perform certain actions and how to interact with other elements of the program. Objects are the basic units of object-oriented programming. A simple example of an object would be a person. Logically, you would expect a person to have a name. This would be considered a property of the person. You would also expect a person to be able to do something, such as walking. This would be considered a method of the person.
A method in object-oriented programming is like a procedure in procedural programming. The key difference here is that the method is part of an object. In object-oriented programming, you organize your code by creating objects, and then you can give those objects properties and you can make them do certain things.

One of the most important characteristics of procedural programming is that it relies on procedures that operate on data - these are two separate concepts. In object-oriented programming, these two concepts are bundled into objects. This makes it possible to create more complicated behavior with less code. The use of objects also makes it possible to reuse code. Once you have created an object with more complex behavior, you can use it anywhere in your code.

2) Stage a1 (apply)

Lab Activities:

Activity 1:

The example demonstrates the difference in approach if we want to find the circumference of circle.

Solution:

Procedural Approach	**Object Oriented Approach**

Public class **Circle**{
int radius;
Public void setRadius(int r)
{ radius = r;}
Public void showCircumference()
{
double c = 2*3.14*radius;
System.out.println(“Circumferenceis”+ c);
}
Public static void main()
{
setRadius(5);
showCircumference();
//output would be 31.4
setRadius(10);
showCircumference();
// output would be 62.8
}
}
	Public class **Circle**{
	Private int radius;
	Public void setRadius(int r)
	{ radius = r;}
	Public void showCircumference()
	{
	double c = 2*3.14* radius;
	System.out.println(“Circumference is”+ c);
	}
	}
	Public class **runner**
	{
	Public static void main()
	{
	Circle c1= new circle();
	c1.setRadius(5);
	c1.showCircumference();
	//output would be 31.4; it belongs to c1
	 Circle c2= new circle();
	c2.setRadius(10);
	c2.showCircumference();
	//output would be 62.8; it belongs to c2
	}
	}

Activity 2:

The example demonstrates the difference in approach if we want to model the concept of a Book. In object Oriented approach the concept can be defined once and then reused in form of different objects.

Procedural Approach	**Object Oriented Approach**

Public class **Book**{
string title;
double price;
int noOfPages;
 	
Public void setTitle(string t)
{ title = t;}
Public void setPrice (double p)
{ price = p;}
Public void setNoOfPages (int n)
{ noOfPages = n;}
	Public class **Book**{
	Private string title;
	Private double price;
	Private int noOfPages;
	
	Public void setTitle(string t)
	{ title = t;}
	Public void setPrice (double p)
	{ price = p;}
	Public void setNoOfPages (int n)
	{ noOfPages = n;}

Public void display()
{
System.out.println(“BookTitle”+ title + “ BookPrice “ + price + “BookPages” + noOfPages);
}
 	
Public static void main()
{
setTitle (“OOP”);
setPrice (200);
setNoOfPages (500);
display();
}
 	
}
	Public void display()
	{
	System.out.println(“BookTitle”+ title + “ BookPrice “ + price + “BookPages” + noOfPages);
	}
	}
	
	Public class **runner**
	{
	Public static void main()
	{
	Book b1= new Book();
	b1.setTitle (“OOP”);
	b1.setPrice (200);
	b1.setNoOfPages (“OOP”);
	b1.display();
	//output belongs to b1
	
	Book b2= new Book();
	b2.setTitle (“ICP”);
	b2..setPrice (150);
	b2.setNoOfPages (350);
	b2.display();
	
	//output belongs to b2
	}
	}

Activity 3:

The example demonstrates the difference in approach if we want to model the concept of an Account. Again we can see that in object Oriented approach the concept can be defined once and then reused uniquely by different objects.

Procedural Approach	**Object Oriented Approach**

Public class **Account**{
double balance;
Public void setBalance(double b)
{ balance = b;}
Public void showBalance()
{
System.out.println(“Balance is”+ balance);
}
Public static void main()
{
setBalance (5000);
showBalance (); // output would be 5000
}
}
	Public class **Account**{
	double balance;
	Public void setBalance(double b)
	{ balance = b;}
	Public void showBalance()
	{
	System.out.println(“Balance is”+ balance);
	}
	}
	Public class **runner**
	{
	Public static void main()
	{

	Account a1= new Account ();
	a1.setBalance(2500);
	a1.showBalance();
	//output would be2500; it belongs to a1
	
	Account a2= new Account ();
	a2.setBalance(5000);
	a2.showBalance();
	//output would be 5000; it belongs to a2
	
	}
	}

3) Stage v (verify)

<u>Home Activities:</u>

Activity 1:

Modify the last activity and include functions of withdraw and deposit.
Test these methods in main for procedural approach. For Object Oriented approach, modify the runner class and call withdraw and deposit functions for two objects.

Activity 2:

Write a program that has variables to store Car data like; CarModel, CarName, CarPrice and CarOwner. The program should include functions to assign user defined values to the above mentioned variable and a display function to show the values . Write a main that calls these functions

Now write another runner class that declares three Car objects and displays the data of all three.

.

4) Stage a2 (assess)

<u>Assignment:</u>

Write a program that contains variables to hold employee data like; employeeCode, employeeName and date Of Joining. Write a function that assigns the user defined values to these variables. Write another function that asks the user to enter current date and then checks if the employee tenure is more than three years or not. Call the functions in main.

Now write a runner class that declares two employee objects and check their tenure periods.

LAB # 02

Statement Purpose:

Objective of this lab is to understand the Object Oriented paradigm.

Activity Outcomes:

The student will be able to understand the Object oriented paradigm.

The student will be able to understand difference between class and object.

Instructor Note:

The students should brainstorm about the scenarios given in activities; in order to model them in terms of Objects.

1) Stage J (Journey)

Introduction

The world around us is made up of objects, such as people, automobiles, buildings, streets, and so forth. Each of these objects has the ability to perform certain actions, and each of these actions has some effect on some of the other objects in the world.

OOP is a programming methodology that views a program as similarly consisting of objects that interact with each other by means of actions.

Object-oriented programming has its own specialized terminology. The objects are called, appropriately enough, objects. The actions that an object can take are called methods. Objects of the same kind are said to have the same type or, more often, are said to be in the same class.

For example, in an airport simulation program, all the simulated airplanes might belong to the same class, probably called the Airplane class. All objects within a class have the same methods. Thus, in a simulation program, all airplanes have the same methods (or possible actions), such as taking off, flying to a specific location, landing, and so forth. However, all simulated airplanes are not identical. They can have different characteristics, which are indicated in the program by associating different data (that is, some different information) with each particular airplane object. For example, the data associated with an airplane object might be two numbers for its speed and altitude.

Things that are called procedures, methods, functions, or subprograms in other languages are all called methods in Java. In Java, all methods (and for that matter, any programming constructs whatsoever) are part of a class.

2) Stage a1 (apply)

Lab Activities:

Activity 1:

Consider the concept of a CourseResult. The CourseResult should have data members like the student name, course name and grade obtained in that course.
This concept can be represented in a class as follows:

Solution:

```
Public class CourseResult
{
        Public  String studentname;
        Public  String coursename;
        Public  String grade;
        public void display()
        {
        System.out.println("Student Name is: “ + studentname + "Course Name is: “ + coursename +
        "Grade is: “ + grade);
        }
}


Public class CourseResultRun
{
        public static void main(String[]args)
        {
                CourseResult c1=new CourseResult ();
                c1.studentName= “Ali”;
                c1.courseName= “OOP”;
                c1.grade= “A”;
                c1.display();

                CourseResult c2=new CourseResult ();
                c2.studentName= “Saba”;
                c2.courseName= “ICP”;
                c2.grade= “A+”;
                c2.display();

        }
}
```

Note that both objects; c1 and c2 have three data members, but each object has different values for their data members.

Activity 2:

The example below represents a Date class. As date is composed of three attributes, namely month, year and day; so the class contains three Data Members. Now every date object will have these three attributes, but each object can have different values for these three

Solution:

```
public class Date
{
                        public String month;
                        public int day;
                        public int year; //a four digit number.

                        public void displayDate()
                         {
                        System.out.println(month + " " + day + ", " + year);
                         }
 }
```

```
public class DateDemo
{
                         public static void main(String[] args)
                          {
                          Date date1, date2;
                          date1 = new Date();
                           date1.month = "December";
                          date1.day = 31;
                          date1.year = 2012;
                         System.out.println("date1:");
                         date1.display();


                         date2 = new Date();
                          date2.month = "July";
                          date2.day = 4;
                          date2.year = 1776;
                         System.out.println("date2:");
                         date2.display();
                          }
                          }
```

Activity 3:

Consider the concept of a Car Part. After analyzing this concept we may consider that it can be described by three data members: modelNumber**,** partNumber and cost.
The methods should facilitate the user to assign values to these data members and show the values for each object.
This concept can be represented in a class as follows:

Solution:

```
importjavax.swing.JOptionPane;

Public class CarPart
{
         private String modelNumber;
         private String partNumber;
         private String cost;

         public void setparameter(String x, String y,String z)
         {
                  modelNumber=x;
                  partNumber=y;
                  cost=z;
         }

         public static void display()
         {
         System.out.println("Model Number: “+modelNumber + “Part Number:  “+partNumber  +
         “Cost:  “ + cost);

         }

}
```

```
Public class CarPartRunner
{
        public static void main(String[]args)
        {
                CarPart car1=new CarPart ();
                String x=JOptionPane.showInputDialog("What is Model Number?" );
                String y=JOptionPane.showInputDialog("What is Part Number?" );
                String z=JOptionPane.showInputDialog("What is Cost?" );
                car1.setparameter(x,y,z);
                car1.display();
        }

}
```

3) Stage v (verify)

Home Activities:

Activity 1:

A Student is an object in a university management System. Analyze the concept and identify the data members that a Student class should have. Also analyze the behavior of student in a university management System and identify the methods that should be included in Student class.

Activity 2:

Time is an intangible concept. Analyze the concept and identify the data members and methods that should be included in Time class.

.

4) Stage a2 (assess)

Assignment 1:

Car is an object that helps us in transportation. Analyze the concept and identify the data members and methods that should be included in Car class.

Assignment 2:

Rectangle is an object that represents a specific shape. Analyze the concept and identify the data members and methods that should be included in Rectangle class.

LAB # 03

Statement Purpose:

Objective of this lab is to understand the importance of classes and construction of objects using constructors.

Activity Outcomes:

The student will be able to declare a classes and objects.

The student will be able to declare member functions and member variables of a class.

The student will be able to declare overloaded constructors.

Instructor Note:

The student should have understanding about object oriented paradigm.

1) Stage J (Journey)

Introduction

- **Data Abstraction**

  Abstraction is the process of recognizing and focusing on important characteristics of a situation or object and leaving/filtering out the un-wanted characteristics of that situation or object. For example a person will be viewed differently by a doctor and an employer.
  A doctor sees the person as patient. Thus he is interested in name, height, weight, age, blood group, previous or existing diseases etc of a person.
  An employer sees a person as an employee. Therefore, employer is interested in name, age, health, degree of study, work experience etc of a person.

- **Class and Object:**

  The fundamental idea behind object-oriented languages is to combine into a single unit both data and the functions that operate on that data. Such a unit is called an object. A class serves as a plan, or blueprint. It specifies what data and what functions will be included in objects of that class. An object is often called an “instance” of a class.

- **Instance Variables and Methods:**

  Instance variables represent the characteristics of the object and methods represent the behavior of the object. For example length & width are the instance variables of class Rectangle and Calculatearea() is a method.
  Instance variables and methods belong to some class, and are defined inside the class to which they belong.

  Syntax:

```
public class Class_Name
{
Instance_Variable_Declaration_1
Instance_Variable_Declaration_2
. . .
Instance_Variable_Declaration_Last

Method_Definition_1
Method_Definition_2
. . .
Method_Definition_Last
}
```

- **Constructors:**

It is a special function that is automatically executed when an object of that class is created. It has no return type and has the same name as that of the class. It is normally defined in classes to initialize data members. A constructor with no parameters is called a no-argument constructor. A constructor may contain arguments which can be used for initiation of data members.

Syntax:

```
class_name( )
{
        Public class_name()
        {
        //body
        }

        Public class_name(type var1, type var2)
        {
        //body
        }
}
```

If your class definition contains absolutely no constructor definitions, then Java will automatically create a no-argument constructor. If your class definition contains one or more constructor definitions, then Java does not automatically generate any constructor; in this case, what you define is what you get. Most of the classes you define should include a definition of a no-argument constructor.

2) Stage a1 (apply)

Lab Activities:

Activity 1:

The following example shows the declaration of class Rectangle. It has two data members that represent the length and width of rectangle. The method calculateArea will return the area of rectangle. The runner class will create an object of Rectangle class and area function will be called.

Solution:

```
public class Rectangle

        {
        Public int length, width;

        Public int Calculatearea ()
        {
        return (length*width);
        }
}
```

```
Public class runner
{
        Public static void mian ()
        {
        Rectangle rect = new Reactangle();
        rect.length= 10;
        rect.width = 5;
        System.out.println(rect.Calculatearea ( ));
        }
}
```

Activity 2:

The following example demonstrates the use of constructors

Solution:

```
public class Rectangle
{
         Public int length, width;

         public Rectangle()
         {
         length = 5;  width = 2;
         }

         public Rectangle(int l, int w)
         {
         length = l;  width = w;
         }

         Public intCalculatearea ()
         {
         return (length*width);
         }
}



Public class runner
{
         Public static void main ()
         {
         Rectangle rect = new Reactangle();

         System.out.println(rect.calculateArea( ));

         Rectangle rect = new Reactangle(10,20);
         System.out.println(rect. calculateArea ( ));
```

```
        }
}
```

Activity 3:

The following example shows the declaration of class Point. It has two data members that represent the x and y coordinate. Create two constructors and a function to move the point. The runner class will create an object of Point class and move function will be called.

Solution:

```
public class Point{

        private int  x;
        private int  y;
        public Point(){
        x=1;
        y=2;
        }
        public Point(int a, int b){
        x=a;
        y=b;
        }
        public void setX(int a){
        x=a;
        }
        public void setY(int b){
        y=b;
        }
        public void display(){
        System.out.println("x coordinate = "+x+" y coordinate = "+y);
        }
        public void movePoint(int a , int b){
        x=x+a;
        y=y+b;
        System.out.println("x coordinate after moving = "+x+" y coordinate after moving =
        "+y);
        }

}

Public class runner
{
         Public static void main ()
         {
         Point p1 = new Point();
         P1.move(2,3);
```

```
            P1.display();


            Point p2 = new Point();
            p2.move(2,3);
            p2.display();

            }

}
```

3) Stage v (verify)

Home Activities:

Activity 1:

Create a class circle class with radius as data member. Create two constructors (no argument, and two arguments) and a method to calculate Circumference.

Activity 2:

Create a class Account class with balance as data member. Create two constructors (no argument, and two arguments) and methods to withdraw and deposit balance.

Activity 3:

Create a class "Distance" with two constructors (no argument, and two argument), two data members (feet and inches). Also create display function which displays all data members.
.

4) Stage a2 (assess)

Assignment 1:

Write a class Marks with three data members to store three marks. Create two constructors and a method to calculate and return the sum.
.

Assignment 2:

Write a class Time with three data members to store hr, min and seconds. Create two constructors and apply checks to set valid time. Also create display function which displays all data members.

LAB # 04

Statement Purpose:

The objective of this lab is to teach the students, concept of encapsulation and access modifiers.

Activity Outcomes:

At the end of this lab student will be familiar with the accessing rules of class data members and member functions

Instructor Note:

The Student must understand the procedure for creating a class.

Student should be able to call the methods of a class by using objects.

1) Stage J (Journey)

Introduction

Encapsulation

Information hiding means that you separate the description of how to use a class from the implementation details, such as how the class methods are defined. The programmer who uses the class can consider the implementation details as hidden, since he or she does not need to look at them. Information hiding is a way of avoiding information overloading. It keeps the information needed by a programmer using the class within reasonable bounds. Another term for information hiding is abstraction.

Encapsulation means grouping software into a unit in such a way that it is easy to use because there is a well-defined simple interface. So, encapsulation and information hiding are two sides of the same coin.

Access Modifiers

Java allows you to control access to classes, methods, and fields via access modifiers. The access to classes, constructors, methods and fields are regulated using access modifiers i.e. a class can control what information or data can be accessible by other classes. To take advantage of encapsulation, you should minimize access whenever possible.

Syntax:

```
classclass_name {

access_specifie type member1;

access_specifie type member1;
…….
}
```

The following table describes the access modifiers provided by JAVA.

Modifier	**Description**
(no modifier)	member is accessible within its package only
Public	member is accessible from any class of any package
Protected	member is accessible in its class package and by its subclasses
Private	member is accessible only from its class

The accessibility rules are depicted in the following table

Protection	Accessed inside Class	Accessed in subclass	Accessed in any Class
Private	Yes	No	No
Protected	Yes	Yes	No
Public	Yes	Yes	yes

Accessors and Mutators

We should always make all instance variables in a class private and should define public methods to provide access to these members. Accessor methods allow you to obtain the data. For example, the method getMonth() returns the number of the month. Mutator methods allow you to change the data in a class object.

2) Stage a1 (apply)

Lab Activities:

Activity 1:

The following example shows the declaration of class Circle. It has one data members radius. The data member is declared private and access is provided by declaring set and get methods.

Solution:

```
public class Circle
{
Private int radius;
public Circle()
{
radius=7;
}
public Circle(int r)
{
radius=r;
}

public void setRadius(int r)
{
radius=r;
}
Public int getRadius(){
        return radius;
}
public void display()
{
System.out.println("radius = "+radius);
}
public void CalculateCircumfrance(){
double a=3.14*radius*radius;
System.out.println("Circumfrance = "+a);
}
}
```

Public class **Runner**

```
{
Public static void main ()
{
Circle c = new Circle();
c1.setRadius(5);
System.out.println("circumference of Circle 1 is " + c1.area( ));

Int r = c1.getRadius();

Circle c2 = new Circle(r);
c2.setRadius(5);
System.out.println("circumference of Circle 2 is " + c2.area( ));

}

}
```

Activity 2:

The following example shows the declaration of class Rectangle. It has two data members that represent the length and width of rectangle. Both data member are declared private and access is provided by declaring set and get methods for both data members.

Solution:

public class **Rectangle**

```
{

        Private  int length, width;

        public Rectangle()
        {
        length = 5;  width = 2;
        }

        public Rectangle(int l, int w)
        {
        length = l;  width = w;
        }
        public void setLength(int l)   //sets the value of length
        {
        length = l;
        }

        public void setWidth(int w)   //sets the value of width
        {
        width = w;
        }

        public void getLength()   //gets the value of length
        {
        return length;
         }

        public void getWidth()   //gets the value of width
        {
        return width;
        }

        Public int area ()
        {
        return (length*width);
        }
        }



Public class Runner
{
        Public static void main ()
        {
        Rectangle rect = new Reactangle();
        Rect.setLength(5);
        Rect.setWidth(10);
        System.out.println("Area of Rectangle is " + rect.area( ));
        System.out.println("width of Rectangle is " + rect.getWidth( ));
        }

 }
```

Activity 3:

The following example shows the declaration of class Point. It has two data members that represent the x and y coordinate of a point. Both data member are declared private and access is provided by declaring set and get methods for both data members.

Solution:

```
public class Point{

        private int  x;
        private int  y;
        public Point(){
        x=0;
        y=0;
        }
        public Point(int a, int b){
        x=a;
        y=b;
        }
        public void setX(int a){
        x=a;
        }
        public void setY(int b){
        y=b;
        }

        Public int getX(){
        return x;
        }
        Public int getY(){
        return y;
        }
        public void display(){
        System.out.println("x coordinate = "+x+" y coordinate = "+y);
        }
        public void movePoint(int a , int b){
        x=x+a;
        y=y+b;
        System.out.println("x coordinate after moving = "+x+" y coordinate after moving =
        "+y);
        }

}

Public class runner
{
         Public static void main ()
         {
         Point p1 = new Point();
         P1.setX(10);
         P1.setY(7);
         P1.display();

         Point p2 = new Point(10,11);
         p2.move(2,3);
         p2.display();
         }

}
```

3) Stage v (verify)

Home Activities:

Activity 1:

Create an Encapsulated class Marks with three data members to store three marks. Create set and get methods for all data members. Test the class in runner

Activity 2:

Create an Encapsulated class Account class with balance as data member. Create two constructors and methods to withdraw and deposit balance. In the runner create two accounts. The second account should be created with the same balance as first account. (hint: use get function)
.

Activity 3:

Create an Encapsulated class Student with following characteristics:

Data Members:

String Name
Int [] Result_array[5] // Result array contains the marks for 5 subjects

Methods:
Student (String, int[])   // argument Constructor
Average ()   // it calculates and returns the average based on the marks in the array.

Runner:
Create two objects of type Student and call the Average method.
Compare the Average of both Students and display which student has higher average.
Create a third student with name as object 1 and result array as object 2

4) Stage a2 (assess)

Assignment:

Suppose you operate several hot dog stands distributed throughout town. Define an Encapsulated class named **HotDogStand** that has an instance variable for the hot dog stand's ID number and an instance variable for how many hot dogs the stand has sold that day.

Create a constructor that allows a user of the class to initialize both values. Also create a method named justSold that increments by one the number of hot dogs the stand has sold. The idea is that this method will be invoked each time the stand sells a hot dog so that you can track the total number of hot dogs sold by the stand.

Write a main method to test your class with at least three hot dog stands that each sell a variety of hot dogs. Use get function to display the hot dogs sold for each object.
.

LAB # 05

Statement Purpose:

The objective of this lab is to teach the students, how the objects can be passed to and returned from the functions.

Activity Outcomes:

After completion of this Lab students will be able to pass objects to methods

After completion of this Lab students will be able to return objects from methods

Instructor Note:

The Student must understand the procedure for creating a class.

Student should be able to call the methods of a class by using objects.

1) Stage J (Journey)

Introduction

In Java, all primitives are passed by value. This means a copy of the value is passed into the method. Objects can be passed natively, just like primitives. It is often misstated that Object parameters are passed by Reference. While it is true that the parameter is a reference to an Object, the reference itself is passed by Value.

2) Stage a1 (apply)

Lab Activities:

Activity 1:

Passing object as parameter and change value of its data member.

Solution:

```
public class ObjectPass
{
        public  int value;

        public static void increment(ObjectPass a)
        {
        a.value++;
        }
}
Public class ObjectPassTest
{
          public static void main(String[] args)
           {
            ObjectPass p = new ObjectPass();
             p.value = 5;
             System.out.println("Before calling: " + p.value);
             increment(p);
            System.out.println("After calling: " + p.value);
            }
}
```

<u>Output:</u>

```
Before calling: 5
 After calling: 6
```

Now it is like the pass was by reference! but the thing is what we pass exactly is a handle of an object, and in the called method a new handle created and pointed to the same object. Now when more than one handles tied to the same object, it is known as **aliasing**. This is the default way Java does when passing the handle to a called method, create alias.

Activity 2:

The following activity demonstrates the creation of a method that accepts and returns object.

Solution:

```
public class Complex
{
        Private double real;
        Private double imag;

        Public Complex()
        { real = 0.0;  imag = 0.0;  }

        Public Complex (double r, double im)
        { real = r;  imag = im; }

        Public Complex Add (Complex  b)
        {
        Complex  c_new = new Complex (real + b.real, imag+ b.imag);
        returnc_new;
        }

        Public Show ()
        {
        System.out,println( real + imag);
        }
}

Public class ComplexTest
{
        void main()
        {
        Complex C1 = new Complex(11, 2.3);
        Complex C2 = new Complex(9, 2.3);
        Complex C3 = new complex();
        C3 = C1.Add(C2);
        C3.show();
        }
}
```

Activity 3:

The following activity demonstrates the creation of a method that accepts two objects.

```
public class Point
{
        private int X;
        private int Y;
        public Point()
        {
                X = 5;
                Y = 6;
        }
        public Point(int a, int c)
        {
                X = a;
                Y = c;
        }
        public void setX (int a)
        {
                X = a;
        }
        public void setY (int c)
        {
                Y= c;
                }
        public int getX()
        {
                return X;
        }
        public int getY(){
                return Y;
        }
        public Point Add(Point Pa, Point Pb )
        {
                Point p_new = new Point(X+ Pa.X + Pb.X, Y+ Pa.Y + Pb.Y);
                return p_new;
        }
        public void display()
        {
                System.out.println(X);
                System.out.println(Y);
        }
        }
public class PointTest
{
        public static void main(String[] args)
        {
        Point p1 = new Point(10,20);
        Point p2 = new Point (30,40);
        Point p3 = new Point ();
        Point p4 = p1.Add(p2,p3);
        p4.display();
        }
}
```

3) Stage v (verify)

Home Activities:

Activity 1:

Create a class “ Distance” with two constructors (no argument, and two argument), two data members (feet and inches) . Create setter, getter and display method. Create a method that adds two Distance Objects and returns the added Distance Object.

Activity 2:

Create an Encapsulated class Book. Its data members are

- author (String)
- chapterNames[100]   (String[])

Create two overloaded constructors, one with no argument and one with two arguments.

Create a method compareBooks that compares the author of two Books and returns true if both books have same author and false otherwise. (This method must manipulate two Book objects)

Create a method compareChapterNames that compares the chapter names of two Books and returns the book with larger chapters. Display the author of the book with greater chapters in main.

Create a runner class that declares two objects of type Book. One object should be declared using no argument constructor and then the parameters should be set through the set() methods. The second object should be declared with argument constructor. Finally the CompareBooks()and compareChapterNames method should be called and the result should be displayed in the runner class.

4) Stage a2 (assess)

Assignment:

Define a class called Fraction. This class is used to represent a ratio of two integers. Create two constructors, set, get and display function. Include an additional method, **equals**, that takes as input another Fraction and returns true if the two fractions are identical and false if they are not.

LAB # 06

Statement Purpose:

This Lab will help students in understanding the concept static data and member function of a class.

Activity Outcomes:

After completion of this Lab students know in which scenario static data members, static functions and static objects can are used.

Instructor Note:

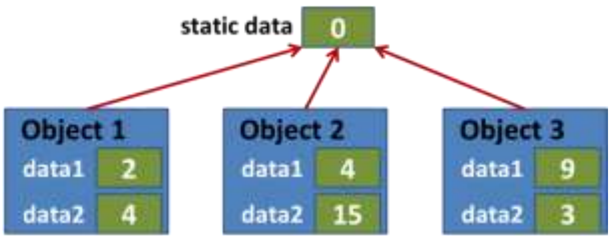
The student should have understanding about access specifiers.

The students should understand the concept of object’s address space in memory.

1) Stage J (Journey)

Introduction

There is an important exception to the rule that each object of a class has its own copy of all the data members of the class. In certain cases, only one copy of a variable should be shared by all objects of a class. A **static data member** is used for such propose.. Such a variable represents "class-wide" information (i.e., a property of the class shared by all instances, not a property of a specific object of the class). The declaration of a static member begins with keyword static.



In the above figure, consider Object1, Object2 and Object3 are three of a class. Each object has its own private data members. The class has a static data member that is share among all the object of that class. This data member has it own memory allocated that is separate from the memory allocated for each object of that class.

Static data member – access

A public static data member can be access using any object of the class or class name

- [Object Name].[name of static variable]
  - e1.count;
- [Class Name] .[name of static variable]
  - Employee .count;

A private static member can be access using a public member function

- [objectName].[public member function]
  - e1.getCount();
- [Class name].[public member function]
  - Employee.getCount();

A static method is a method that can only access static data member of class. It cannot access non static data member of that class

Static vs. non static functions

- Static function can only access static data member of class
  - className.static_ method ();
  - Object_of_class.static_method();
- Non static member function can access static and non static data member of class
  - Object_of_class.Non_static_ method ();

2) Stage a1 (apply)

Lab Activities:

Activity 1:

The following example demonstrates that static variables are common for all instances:

Solution:

```
Public class ABC
{
          static int Var1=77;              //Static integer variable
          String Var2;                     //non-static string variable
}
Public class ABCRunner
{

  public static void main(String args[])
  {
    ABC ob1 = new ABC ();
    ABC ob2 = new ABC ();
    ob1.Var1=88;
    ob1.Var2="I'm Object1";
    ob2.Var2="I'm Object2";
    System.out.println("ob1 integer:"+ob1.Var1);
    System.out.println("ob1 String:"+ob1.Var2);
    System.out.println("ob2 integer:"+ob2.Var1);
    System.out.println("ob2 STring:"+ob2.Var2);
  }
}
```

Output:

```
ob1 integer:88
ob1 String:I'm Object1
ob2 integer:88
ob2 String:I'm Object2
```

Activity 2:

The following example demonstrates counting of objects by using static variable.

Solution:

```
Public class NoOfObjects {

        Private static int objs=0;
        Private int a;

        Public NoOfObjects(){
                objs++;
        }

        Public NoOfObjects(intx){
                a=x;
                objs++;
        }

        Public static int getObjs (){
                return objs;
        }
}
Public class NoOfObjectsRunner
{

        Public static void main(String[] args){

        NoOfObjects o1=newNoOfObjects();
        NoOfObjects o2=newNoOfObjects(122);
        NoOfObjects o3=newNoOfObjects(150);

        System.out.println("Objects created:"+ NoOfObjects.getObjsCreated());
        System.out.println("Objects created:"+ o1.getObjsCreated());
        }
}
```

Activity 3:

The following example demonstrates static methods declaration.

```
Public class ABC
{
       Public static int i ;
       public String s;

       Public static void displayStatic()   //Static method
       {
         System.out.println("i:"+i);
        }
       Public void display()                //non static method
       {
         System.out.println("i:"+i);
         System.out.println("s:"+s);
       }
}
Public class ABCRunner
{
   public static void main(String args[]) //Its a Static Method
  {
     ABC a = new ABC();
     a.display();
    ABC.displayStatic();
  } }
```

3) Stage v (verify)

Home Activities:

Activity 1:

Create a SavingsAccount class. Use a static data member annualInterestRate to store the annual interest rate for each of the savers. Each member of the class contains a private data member savingsBalance indicating the amount the saver currently has on deposit. Provide member function calculateMonthlyInterest that calculates the monthly interest by multiplying the balance by annualInterestRate divided by 12; this interest should be added to savingsBalance.
Provide a static member function modifyInterestRate that sets the static annualInterestRate to a new value.
Write a driver program to test class SavingsAccount. Instantiate two different objects of class SavingsAccount, saver1 and saver2, with balances of $2000.00 and $3000.00, respectively. Set the annualInterestRate to 3 percent. Then calculate the monthly interest and print the new balances for each of the savers.
Then set the annualInterestRate to 4 percent, calculate the next month's interest and print the new balances for each of the savers.

Activity 2:

Write program to count the number of objects created and destroyed for a class using static data members and static member functions

4) Stage a2 (assess)

Assignment:

Create a Calculator class that has following methods:

sum, multiply, divide , modulus , sin , cos , tan

The user should be able to call these methods without creating an object of Calculator class.

LAB # 07

Statement Purpose:

The purpose of lab is to make students understand the concept of has-a relationship in object oriented programming**.**

Activity Outcomes:

Students will be able to understand that complex objects can be modeled as composition of other objects.

Students will be able to implement programs related to composition.

Instructor Note:

The student should be able to declare a class with primitive data members.

The student should be able to define methods; that can receive and return objects.

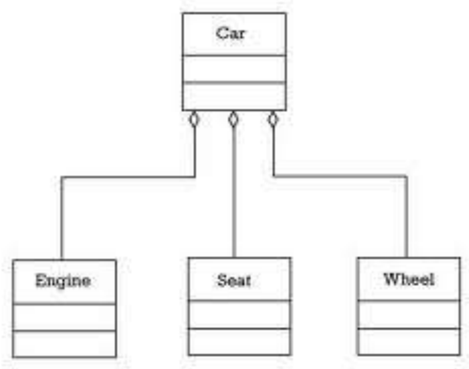
1) Stage J (Journey)

Introduction

In real-life, complex objects are often built from smaller, simpler objects. For example, a car is built using a metal frame, an engine, some tires, a transmission, a steering wheel, and a large number of other parts. A personal computer is built from a CPU, a motherboard, some memory, etc. Even you are built from smaller parts: you have a head, a body, some legs, arms, and so on. This process of building complex objects from simpler ones is called composition (also known as object containership).

More specifically, composition is used for objects that have a “has-a” relationship to each other. A car has-a metal frame, has-an engine, and has-a transmission. A personal computer has-a CPU, a motherboard, and other components. You have-a head, a body.

```
class wheel
{
}

class car
{
   wheel w1;
}
```



2)

Lab Activities:

Activity 1:

Composition is about expressing relationships between objects. Think about the example of a manager. A manager has the properties e.g Title and club dues. A manager has an employment record. And a manager has an educational record. The phrase "has a" implies a relationship where the manager owns, or at minimum, uses, another object. It is this "has a" relationship which is the basis for composition. This example can be programmed as follows:

Solution:

```
public class studentRecord
{
        Private String degree;

        public studentRecord()
        {}
        public void setDegree ( String deg)
        {
        degree = deg;
        }


        public void getDegree ()
        {
        return degree;
        }

}

class employeeRecord
{
        Private int emp_id;
        Private double salary;
        public employeeRecord ()
        {}
        public void setEmp_id ( int id)
        {
        emp_id = id;
        }
        public intgetEmp_id ()
        {
        return emp_id ;
        }
        public void setSalary ( intsal)
        {
        Salary  = sal;
        }
        public StringgetSalary ( intsal)
        {
        Salary  = sal;
        }

}

class Manager
{
        Private String title;
        Private double dues;
        Private employeeRecordemp;
        Private studentRecordstu;

        Public manager(String t,double d,employeeRecord e,studentRecord s)
        {
        titile = t;
        dues = d;
        emp = e;
        stu = s;  }

        Public void display()
        {
```

```
        System.out.println("Title is : " + title);
        System.out.println("Dues are : " + dues);

        System.out.println("Emplyoyee record is as under:" );
        System.out.println("EmployeeId is : " + emp. getEmp_id());
        System.out.println("EmployeeId is : " + emp. getSalary());

        System.out.println("Student record is as under:" );
        System.out.println("Degree is : " + stu.getDegree());
        }


}
Public class Runner
{
        voidmain()
        {
        studentRecord s = new studentRecord("MBA");
        employeeRecord e = new employeeRecord(111, 50000);
        Manager m1 = new Manager("financeManager " , 5000, e, s ) ;
        m1.display();
        }

}
```

Activity 2:

The program below represents an employee class which has two Date objects as data members.

Solution:

```
public class Date

{

        private int day;
        private int month;
        private int year;

        public Date (int theMonth, int theDay, int theYear )
        {
        day=checkday(theDay);
        month=checkmonth(theMonth);
        year=theYear;
        }

        Private intcheckmonth (int testMonth)
        {
        if (testMonth>0 &&testMonth<=12)
        returntestMonth;
        else
            {
        System.out.println("Invalid month" + testMonth + "set to 1");
        return 1;
            }
        }
        Private int checkday(int testDay)
        {
        intdaysofmonth[]={0,31,28,31,30,31,30,31,31,30,31,30,31};

        if (testDay>0 &&testDay<=daysofmonth[month])
        return testDay;

        else
        if (month==2 &&testDay==29 && (year%400==0 || (year%4==0 && year%100
        !=0)))
        return testDay;

        else
        System.out.println("Invalid date"+ testDay + "set to 1");
        return 1;
        }
        Public int getDay()
        {
        return day;
        }

        Public int getMonth()
        {
        return month;
        }

        Public int getYear()
        {
        return year;
        }

        public void display()
        {
              System.out.println(day +" "+ month+"  " + year);
        }


}
public class **employee**
 {
            private String name;
            private String fname;
            private Date birthdate;
            private Date hiredate;
```

```
    employe()
    {

    }
    employe(String x, String y, Date birthofDate, Date dateofHire)
    {
       name=x;
       fname=y;
       birthDate=birthofDate;
       hireDate=dateofHire;

    }
    public void setname(String x)
    {
       name=x;
    }
    public String getname()
    {
       return name;

    }
    public void setfname(String x)
    {
       fname=x;
    }
    public String getfname()
    {
       return fname;
    }


    public void setbirthdate(Date b)
    {
       birthdate= b;

    }
    public Date getbirthdate()
    {
       return birthdate;

    }

    public void sethiredate(Date h)
    {
       hiredate = h;
     }
 public Date gethiredate()
    {
       return hiredate;
     }
```

```
            public void display()
            {

            System.out.println("Name: "+ name +"  Father Name: "+ fname );
            birthdate.display();
            hiredate.display();

            }
}
public class Employrun {

   public static void main(String[] args)
{
         Date b = new Date (1,12,1990);
         Date h = new Date (5,6,2016);
         employee e1=new employe("xxx", "yyyy",b,h);
         e1.display();
   }

}
```

3) Stage v (verify)

Home Activities:

Activity 1:

Create an Address class, which contains street#, house#, city and code. Create another class Person that contains an address of type Address. Give appropriate get and set functions for both classes. Test class person in main.

Activity 2:

Create a class Book that contains an author of type Person (Note: Use the Person class created in the first exercise). Other data members are bookName and publisher. Modify the address of the author in runner class.

Activity 3:

Design a class Point with two data members x-cord and y-cord. This class should have an arguments constructor, setters, getters and a display function.
Now create another class "Line", which contains two Points "startPoint" and "endPoint". It should have a function that finds the length of the line.

Hint: formula is: $\text{sqrt}((x_2-x_1)^2 + (y_2-y_1)^2)$

Create two line objects in runner and display the length of each line.

4) Stage a2 (assess)

Assignment:

Create a class named Pizza that stores information about a single pizza. It should contain the following:

- Private instance variables to store the size of the pizza (either small, medium, or large), the number of cheese toppings, the number of pepperoni toppings, and the number of ham toppings.
- Constructor(s) that set all of the instance variables.
- Public methods to get and set the instance variables.
- A public method named calcCost( ) that returns a double that is the cost of the pizza. Pizza cost is determined by:

  Small: $10 + $2 per topping
  Medium: $12 + $2 per topping
  Large: $14 + $2 per topping

- public method named getDescription( ) that returns a String containing the pizza size, quantity of each topping.

Write test code to create several pizzas and output their descriptions. For example, a large pizza with one cheese, one pepperoni and two ham toppings should cost a total of $22.
Now Create a PizzaOrder class that allows up to three pizzas to be saved in an order. Each pizza saved should be a Pizza object. Create a method calcTotal() that returns the cost of order.
In the runner order two pizzas and return the total cost.

LAB # 08

Statement Purpose:

The objective of this lab is to familiarize the students with various concepts and terminologies of inheritance using Java.

Activity Outcomes:

This lab teaches you the following topics:

- Declaration of the derived classes along with the way to access of base class members.
- Protected Access modifier and working with derived class constructors.

Instructor Note:

As pre-lab activity, read Chapter 7 from the book, (Absolute Java, Savitch, W. & Mock, K., 5th Edition (2012), Pearson.), and also as given by your theory instructor.

1) Stage J (Journey)

Introduction

a. Inheritance

Inheritance is a way of creating a new class by starting with an existing class and adding new members. The new class can replace or extend the functionality of the existing class. The existing class is called the base class and the new class is called the derived class.

b. Protected Access Specifier

Protected members are directly accessible by derived classes but not by other users.

A class member labeled **protected** is accessible to member functions of derived classes as well as to member functions of the same class.

c. Derived class constructor

Constructors are not inherited, even though they have public visibility. However, the super reference can be used within the child's constructor to call the parent's constructor. In that case, the call to parent's constructor must be the first statement.

2) Stage a1 (apply)

Lab Activities:

Activity 1:

This example will explain the method to specify the derived class. It explains the syntax for writing the constructor of derived class.

Solution:

```
public class person {

                    protected String name  ;
                    protected String id ;
                    protected int phone ;

                    public person() {
                              name = "NaginaNazar" ;
                              id = "sp14bcs039" ;
                              phone = 12345 ;
                    }

                    public person(String a , String b , int c) {
                              name = a ;
                              id = b ;
                              phone = c ;
                    }
}

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

	public void setId(String j){
		id = j ;

	}

	public void setPhone(int a) {
		phone = a ;
	}

	public String getName() {
		return name ;
	}

	public String getid() {
		return id ;
	}

	Public int getPhone() {
		return phone ;
	}

	public void display( ) {
		System.out.println("Name : " + name + "ID : " + id + "Phone : " + phone ) ;

	}
}
public class student extends person {
	private String rollNo ;
	private int marks ;

	public student() {
		super() ;
		rollNo = "sp14bcs039" ;
		marks = 345 ;
	}

	public student(String a , String b , int c , String d , int e){
		super(a,b,c) ;
		rollNo = d ;
		marks = e ;
	}

	public void setRollNo(String a){
		rollNo = a ;
	}

	public void setMarks(int a ){
		marks = a ;
	}
```

```
                public String getRollNo() {
                        returnrollNo ;
                }

                publicintgetMarks() {
                        return marks ;
                }


        public void display( ) {
                super.display();
                System.out.println("Roll # : " + rollNo + "\nMarks : " + marks) ;

        }

        }
public class Runner
{
        public static void main(String []args)
        {
        Student s = new Student ("s-09",Ahmed","xyz","sp16-bcs-98,50);
        s.display();
        }
}
```

Activity 2:

This example demonstrates another scenario of inheritance. The super class can be extended by more than one class.

Solution:

```
public class Employee {

                protected String name;
                protected String phone;
                protected String address;
                protected int allowance;

                public Employee(String name, String phone, String address, int allowance)
                {
                this.name = name;
                this.phone = phone;
                this.address = address;
                this.allowance = allowance;
                }
}
 public class Regular extends Employee
{
        Private int basicPay;
        public Regular(String name, String phone, String address, int allowance, intbasicPay)
         {
        super(name, phone, address, allowance);
        this.basicPay = basicPay;
        }
        public void Display(){
        System.out.println("Name: " + name + "Phone Number: " + phone +"Address: " + address
        + "Allowance:  " + allowance + "Basic Pay:  " + basicPay);    }
}
public class Adhoc extends  Employee
{
        private  int numberOfWorkingDays;
        private int wage;

        public Adhoc(String name, String phone, String address, int allowance, int
        numberOfWorkingDays, int wage)
        {
        super(name, phone, address, allowance);
        this.numberOfWorkingDays = numberOfWorkingDays;
        this.wage = wage;
        }
        public void Display()
        {
        System.out.println("Name: " + name + "Phone Number: " + phone +"Address: " + address
                + "Allowance:   " + allowance + "Number Of Working Days:   " +
        numberOfWorkingDays +  "Wage:  " + wage);
         }
}
public class Runner
{
        public static void main(String []args){
        Regular regularObj = new Regular("Ahmed","090078601","Islamabad",15000,60000);
        regularObj.Display();
        AdhocadhocObj = new Adhoc("Ali","03333333333","Rawalpindi",500,23,1500);
        adhocObj.Display();
         }
}
```

3) Stage V (verify)

Home Activities:

Activity 1:

1. (The Person, Student, Employee, Faculty, and Staff classes)
   Design a class named **Person** and its two subclasses named **Student** and **Employee**. Design two more classes; **Faculty** and **Staff** and extend them from **Employee**. The detail of classes is as under:

   - A person has a name, address, phone number, and email address.
   - A student has a status (String)
   - An employee has an office, salary, and date hired. Use the Date class to create an object for date hired.
   - A faculty member has office hours and a rank.
   - A staff member has a title.
   - Create display method in each class

4) Stage a_2 (assess)

Lab Assignment and Viva voce

1. Imagine a publishing company that markets both book and audio-cassette versions of its works. Create a class publication that stores the title and price of a publication. From this class derive two classes:

   i. book, which adds a page count and

   ii. tape, which adds a playing time in minutes.

   Each of these three classes should have set() and get() functions and a display() function to display its data.

   Write a main() program to test the book and tape class by creating instances of them, asking the user to fill in their data and then displaying the data with display*()*.

2. Write a base class Computer that contains data members of wordsize(in bits), memorysize (in megabytes), storagesize (in megabytes) and speed (in megahertz). Derive a Laptop class that is a kind of computer but also specifies the object's length, width, height, and weight. Member functions for both classes should include a default constructor, a constructor to inialize all components and a function to display data members.

LAB # 09

Statement Purpose:

The objective of this lab is to familiarize the students with various concepts and terminologies of method overriding and concept of abstract classes.

Activity Outcomes:

This lab teaches you the following topics:

- Method overriding where a base class method version is redefined in the child class with exact method signatures.
- Abstract classes along with the access of base class members.

Instructor Note:

As pre-lab activity, read Chapter 7 from the book, (Absolute Java, Savitch, W. & Mock, K., 5th Edition (2012), Pearson.), and also as given by your theory instructor.

1) Stage J (Journey)

Introduction

Method Overriding

The definition of an inherited method can be changed in the definition of a derived class so that it has a meaning in the derived class that is different from what it is in the base class. This is called overriding the definition of the inherited method.

For example, the methods toString and equals are overridden (redefined) in the definition of the derived class HourlyEmployee. They are also overridden in the class SalariedEmployee. To override a method definition, simply give the new definition of the method in the class definition, just as you would with a method that is added in the derived class.

In a derived class, you can override (change) the definition of a method from the base class. As a general rule, when overriding a method definition, you may not change the type returned by the method, and you may not change a void method to a method that returns a value, nor a method that returns a value to a void method. The one exception to this rule is if the returned type is a class type, then you may change the returned type to that of any descendent class of the returned type. For example, if a function returns the type Employee, when you override the function definition in a derived class, you may change the returned type to HourlyEmployee, SalariedEmployee, or any other descendent class of the class Employee. This sort of changed return type is known as a covariant return type and is new in Java version 5.0; it was not allowed in earlier versions of Java.

Abstract Class

A class that has at least one abstract method is called an abstract class and, in Java, must have the modifier abstract added to the class heading. An abstract class can have any number of abstract methods. In addition, it can have, and typically does have, other regular (fully defined) methods. If a derived class of an abstract class does not give full definitions to all the abstract methods, or if the derived class adds an abstract method, then the derived class is also an abstract class and must include the modifier abstract in its heading.

In contrast with the term abstract class, a class with no abstract methods is called a concrete class.

2) Stage a1 (apply)

Lab Activities:

Activity 1:

The following activity demonstrates the creation of overridden methods**.**

Solution:

```
class A
 {
        int i, j;

        A(int a, int b) {
        i = a;
        j = b;
        }
        // display i and j
        void show() {
        System.out.println("i and j: " + i + " " + j);
        }
}
class B extends A
{
        int k;
        B(int a, int b, int c) {
        super(a, b);
        k = c;
        }
        // display k – this overrides show() in A
        void show() {
        System.out.println("k: " + k);
        }
}
Public class OverrideRunner
 {
        public static void main(String args[])
        {
        B subOb = new B(1, 2, 3);
        subOb.show(); // this calls show() in B
        }
}
```

Output:

The output produced by this program is shown here:

```
k: 3
```

Activity 2:

The following activity explains the use of overriding for customizing the method of super class.

The classes below include a CommisionEmployee class that has attributes of firstname, lastName, SSN, grossSales, CommisionRate. It has a constructor to initialize, set and get functions, and a function to display data members.

The other class BasePlusCommisionEmployee is inherited from CommisionEmployee. It has additional attributes of Salary. It also has set and get functions and display function.

The Earning method is overridden in this example.

Solution:

```
public class commissionEmployee
{
        protected String FirstName;
        protected String LastName;
        protected String SSN;
        protected double grossSales;
        protected double commonRate;
        public commissionEmployee()
        {
                FirstName="Nagina";
                LastName="Nazar";
                SSN="S003";
                grossSales=1234.1;
                commonRate=12.5;
        }
        Public commissionEmployee (String a,Stringe,String b, double c, double d){
                FirstName=a;
                LastName=e;
                SSN=b;
                grossSales=c;
                commonRate=d;
        }
        public void setFN(String a){
                FirstName=a;
        }
        public void setLN(String e){
                LastName=e;
        }
        public void setSSN(String b){
                SSN=b;
        }
        public void setGS(double c){
                grossSales=c;
        }
        public void setCR(double d){
                commonRate=d;
        }
        public String getFN(){
                returnFirstName;
        }

        public String getSSN(){
                return SSN;
        }
        public double getGS(){
                returngrossSales;
        }
        public double getCR(){
                returncommonRate;
        }
        public double earnings(){
                returngrossSales*commonRate;
```

```
                    }
                    public void display(){
                              System.out.println("first name:"+FirstName+",last name:"+LastName+"
,SSN:"+SSN+" Gross Sale:"+grossSales+" and commonRate:"+commonRate);
                    }
}
```

```
public class **BasePlusCommEmployee** extends **commissionEmployee**{
          private double salary;

          BasePlusCommEmployee(){
                    salary=48000;
          }
          BasePlusCommEmployee(String A,String E,String B, double C, double D, double S){
                    super(A,E,B,C,D);
                    salary=S;
          }
          //overridden method
          public double earnings(){
                    return super.earnings()+salary;
          }
          public void display(){
                    super.display();
                    System.out.println("Salary : "+salary);
          }
}
```

```
Public class OverrideRunner
 {
          public static void main(String args[])
          {
          BasePlusCommEmployee b = new BasePlusCommEmployee(“ali”, “ahmed”, “25-kkn”,
          100, 5.2, 25000);
          double earn = b.earnings();
          System.out.println("Earning of employee is “ + earn);

          }
}
```

Activity 3:

Abstract Class

```
          // A Simple demonstration of abstract.
          abstract class A {
          abstract void callme();
          // concrete methods are still allowed in abstract classes
          void callmetoo() {
          System.out.println("This is a concrete method.");
          }
          }
```

```
class B extends A {
void callme() {
System.out.println("B's implementation of callme.");
}
}
class AbstractDemo {
public static void main(String args[]) {
B b = new B();
b.callme();
b.callmetoo();
}
}
```

Output:

```
B’s implementation of Call me.
This is a concrete method
```

3) Stage V (verify)

<u>Home Activities:</u>

Activity 1:

Create a class named Movie that can be used with your video rental business. The Movie class should track the Motion Picture Association of America (MPAA) rating (e.g., Rated G, PG-13, R), ID Number, and movie title with appropriate accessor and mutator methods. Also create an equals() method that overrides Object ’s equals() method, where two movies are equal if their ID number is identical. Next, create three additional classes named Action , Comedy , and Drama that are derived from Movie . Finally, create an overridden method named calcLateFees that takes as input the number of days a movie is late and returns the late fee for that movie. The default late fee is $2/day. Action movies have a late fee of $3/day, comedies are $2.50/day, and dramas are $2/day. Test your classes from a main method.

Activity 2:

Write a program that declares two classes. The parent class is called Simple that has two data members num1 and num2 to store two numbers. It also has four member functions.

- The add() function adds two numbers and displays the result.
- The sub() function subtracts two numbers and displays the result.
- The mul() function multiplies two numbers and displays the result.
- The div() function divides two numbers and displays the result.

The child class is called VerifiedSimple that overrides all four functions. Each function in the child class checks the value of data members. It calls the corresponding member function in the parent class if the values are greater than 0. Otherwise it displays error message.

Activity 3:

Create an abstract class that stores data about the shapes e.g. Number of Lines in a Shape, Pen Color, Fill Color and an abstract method draw. Implement the method draw for Circle, Square and Triangle subclasses, the better approach is to draw these figures on screen, if you can't then just use a display message in the draw function.

4) Stage a_2 (assess)

Lab Assignment and Viva voce

Task 1:

Implement a Clock class that simulates time in hr:min:sec, derive a child class that overrides the display method and displays the time in both AM/PM and 24 hour format.

Task 2:

Override the built in method of StringTokenizer class countTokens so that the tokens that contain Numeric values cannot be counted.

Task 3:

Create an Abstract class Student that contains a method take exam, implement the method in the child classes PhdStudent and GradStudent in which PhdStudent takes exam by giving his final defense presentation while the graduate student gives a written paper.

LAB # 10

Statement Purpose:

Objective of this lab is to review concepts related to Inner Classes, we will describe one of the most useful applications of inner classes, namely, inner classes used as helping classes. An inner class is simply a class defined within another class. Because inner classes are local to the class that contains them, they can help make a class self-contained by allowing you to make helping classes inner classes

Activity Outcomes:

Student will learn to make inner class private and access the class through a method.

- How to access the private members of a class using inner class.
- How to use a method-local inner class.
- How to override the method of a class using anonymous inner class.
- How to pass an anonymous inner class as a method argument.
- How to use a static nested class.

Instructor Note:

As pre-lab activity, read Chapter 13 from the book, (Absolute Java, Savitch, W. & Mock, K., 5th Edition (2012), Pearson.), and also as given by your theory instructor.

1) Stage J (Journey)

Introduction

Inner Classes:

Inner classes are a security mechanism in Java. We know a class cannot be associated with the access modifier **private**, but if we have the class as a member of other class, then the inner class can be made private. And this is also used to access the private members of a class. Creating an inner class is quite simple. You just need to write a class within a class. Unlike a class, an inner class can be private and once you declare an inner class private, it cannot be accessed from an object outside the class.

Accessing the Private Members:

As mentioned earlier, inner classes are also used to access the private members of a class. Suppose a class is having private members to access them. Write an inner class in it, return the private members from a method within the inner class, say, **getValue()**, and finally from another class (from which you want to access the private members) call the getValue() method of the inner class.

Method-local Inner Class:

In Java, we can write a class within a method and this will be a local type. Like local variables, the scope of the inner class is restricted within the method. A method-local inner class can be instantiated only within the method where the inner class is defined.

Anonymous Inner Class:

An inner class declared without a class name is known as an **anonymous inner class**. In case of anonymous inner classes, we declare and instantiate them at the same time. Generally they are used whenever you need to override the method of a class or an interface. The syntax of an anonymous inner class is as follows:

Syntax:

```
AnonymousInner an_inner = new AnonymousInner()

{

   public void my_method()

{

   ........

   .......

 }

};
```

Anonymous Inner Class as Argument:

Generally if a method accepts an object of an interface, an abstract class, or a concrete class, then we can implement the interface, extend the abstract class, and pass the object to the method. If it is a class, then we can directly pass it to the method. But in all the three cases, you can pass an anonymous inner class to the method.

Syntax:

```
obj.my_Method(new My_Class(){

  public void Do(){

  .....

  .....  }

});
```

Static Nested Class

A static inner class is a nested class which is a static member of the outer class. It can be accessed without instantiating the outer class, using other static members. Just like static members, a static nested class does not have access to the instance variables and methods of the outer class. The syntax of static nested class is as follows:

Syntax:

```
class MyOuter {

  static class Nested_Demo{

  }

}
```

2) Stage a1 (apply)

Lab Activities:

Consult the case study of Bank Account as outer and Money as Inner class given in the recommended text book, implement the following tasks.

1: Make an inner class private and access the class through a method, check the access ability of the private inner class outside the outer class.

2: Run the code and check the class object code generated for both the inner and outer classes, verify the syntax of outer_class$innerclass is followed for the class files generated.

3: Convert the code implemented in Lab Task 1 into two separate classes and using the composition for inner class object.

Activity 1:

```
class Outer_Demo{

  int num;
  //inner class
  private class Inner_Demo{
    public void print(){
      System.out.println("This is an inner class");
    }
  }
  //Accessing he inner class from the method within
  void display_Inner(){
    Inner_Demo inner = new Inner_Demo();
    inner.print();
  }
}

public class My_class{
  public static void main(String args[]){
    //Instantiating the outer class
    Outer_Demo outer = new Outer_Demo();
    //Accessing the display_Inner() method.
    outer.display_Inner();
  }
}
```

Output:

```
This is an inner class.
```

Activity 2:

Access the private members of a class using inner class.

Code:

```
class Outer_Demo {
   //private variable of the outer class
   private int num= 175;
   //inner class
   public class Inner_Demo{
      public int getNum(){
         System.out.println("This is the getnum method of the inner class");
         return num;
      }
   }
}

public class My_class2{
   public static void main(String args[]){
      //Instantiating the outer class
      Outer_Demo outer=new Outer_Demo();
      //Instantiating the inner class
      Outer_Demo.Inner_Demo inner=outer.new Inner_Demo();
      System.out.println(inner.getNum());
   }
}
```

Output:

```
The value of num in the class Test is: 175
```

Activity 3:

Use a method-local inner class.

Code:

```
public class Outerclass{

   //instance method of the outer class
   void my_Method(){
      int num = 23;

      //method-local inner class
      class MethodInner_Demo{
         public void print(){
            System.out.println("This is method inner class "+num);
         }
      }//end of inner class
      //Accessing the inner class
      MethodInner_Demo inner = new MethodInner_Demo();
      inner.print();
   }

   public static void main(String args[]){
      Outerclass outer = new Outerclass();
      outer.my_Method();
   }
}
```

Output:

```
This is method inner class 23
```

3) Stage V (verify)

Home Activities:

Implement Vehicle as outer and owner as the inner class, the vehicle class contains vehicle name, engine cc, model as data members. The inner class data members are owners name, CNIC number and phone contact of the owner.
Write down proper setters/ getters and constructors for both the classes.

1: Override the method of a class using anonymous inner class.
2: Pass an anonymous inner class as a method argument.
3: Implement the inner class as static first and then as non static nested class.

4) Stage a2 (assess)

Lab Assignment and Viva voce

Task 1:

Redo the class Person from previous assignments so that the class Date is a private inner class of the class Person. Also, do a suitable test program.

Task 2:

Redo the class Employee and the class HourlyEmployee, so that the class Date is an inner class of the class Employee and an inherited inner class of the class HourlyEmployee. Also, do a suitable test program.

LAB # 11

Statement Purpose:

Objective of this lab is to learn how to define Interface and implement Interfaces. Students will also learn how to implement multiple interfaces in one class.

Activity Outcomes:

This lab teaches you the following topics:

- Interface class.
- Implementation of Interface classes and its methods.
- Implementation of multiple interfaces in class.
- Use of interfaces with abstract classes

Instructor Note:

As pre-lab activity, read Chapter 7 from the book, (Absolute Java, Savitch, W. & Mock, K., 5th Edition (2012), Pearson.), and also as given by your theory instructor.

1) Stage J (Journey)

Introduction

An interface is a type that groups together a number of different classes that all include method definitions for a common set of method headings.
Java interface specifies a set of methods that any class that implements the interface must have. An interface is itself a type, which allows you to define methods with parameters of an interface type and then have the code apply to all classes that implement the interface. One way to view an interface is as an extreme form of an abstract class. However, as you will see, an interface allows you to do more than an abstract class allows you to do. Interfaces are Java’s way of approximating multiple inheritance. You cannot have multiple base classes in Java, but interfaces allow you to approximate the power of multiple base classes.

Syntax:

How to define interface

```
public interface Ordered {

public boolean precedes(Object other);

}
```

How to implement interface and its method

```
public class Class_name implements Interface_Name{

public boolean precedes(Object other){

}}

public class implements SomeInterface, AnotherInterface{}
```

Abstract Classes Implementing Interfaces

A concrete class (that is, a regular class) must give definitions for all the method headings given in an interface in order to implement the interface. However, you can define an abstract class that implements an interface but gives only definitions for some of the method headings given in the interface. The method headings given in the interface that are not given definitions are made into abstract methods.

Derived Interfaces

You can derive an interface from a base interface. This is often called extending the interface. The details are similar to deriving a class.

The Comparable Interface

The Comparable interface is in the java.lang package and so is automatically available to your program. The Comparable interface has only the following method heading that must be given a definition for a class to implement the Comparable interface:

public int compareTo(Object other);

The method compareTo should return a negative number if the calling object “comes before” the parameter other, a zero if the calling object “equals” the parameter other, and a positive number if the calling object “comes after” the parameter other. The “comes before” ordering that underlies compareTo should be a total ordering. Most normal ordering, such as less-than ordering on numbers and lexicographic ordering on strings, is total ordering.

Defined Constants in Interfaces

The designers of Java often used the interface mechanism to take care of a number of miscellaneous details that do not really fit the spirit of what an interface is supposed to be. One example of this is the use of an interface as a way to name a group of defined constants. An interface can contain defined constants as well as method headings, or instead of method headings. When a method implements the interface, it automatically gets the defined constants. For example, the following interface defines constants for months:

```
public interface MonthNumbers {

public static final int JANUARY = 1,  FEBRUARY = 2,MARCH = 3, APRIL = 4,
MAY = 5,          JUNE = 6, JULY = 7, AUGUST = 8, SEPTEMBER = 9,
OCTOBER = 10, NOVEMBER = 11, DECEMBER = 12; }
```

Any class that implements the MonthNumbers interface will automatically have the 12 constants defined in the MonthNumbers interface. For example, consider the following toy class:

```
public class DemoMonthNumbers implements MonthNumbers {

 public static void main(String[] args)    {

        System.out.println( "The number for January is " + JANUARY);    } }
```

2) Stage a1 (apply)

Lab Activities:

Activity 1:

Declare a Interface, RegisterForExams that contains single method register, implements the interface in two different classes (a) Student (b) Employee. Write down a Test Application that contains at least a function that takes Interface type Parameter

Solution:

```
        public interface RegisterForExams {
          public void register();

        }
        public class InterfaceTestClass {

                    public InterfaceTestClass(RegisterForExams as)
                    {
                              as.register();
                    }
          }
        public class EmplayeeTask implements RegisterForExams{

          private String name;
          private String date;
          private int salary;

          public EmplayeeTask()
          {
                    name = null;
                    date = null;
                    salary = 0;
          }


          public EmplayeeTask(String name,String date,int salary)
          {
                    this.name = name;
                    this.date = date;
                    this.salary = salary;
          }

          @Override
          public void register() {
                    // TODO Auto-generated method stub
                    System.out.println("Name " + name + "\nsalary " + salary + "\n Employee
reistered on date " + date);
                    }
}
public class StudentTask implements RegisterForExams{
                    private String name;
                    private int age;
                    private double gpa;

                    public StudentTask()
                    {
                    name = null;
                    age = 0;
                    gpa = 0;
                    }
                    public StudentTask(String name,int age,double gpa)
                    {
                    this.name = name;
                    this.age = age;
                    this.gpa = gpa;

                    }
 @Override
public void register() {
                    // TODO Auto-generated method stub
    System.out.println("Student name " + name + " gpa " + gpa);
                    }
  }
```

Activity 2:

This example show how to implement interface to a class

```
        public class OrderedHourlyEmployee extends HourlyEmployee implements Ordered{

public boolean precedes(Object other) {
if (other ==  null)

return false;
  else if (!(other instanceof OrderedHourlyEmployee))
return false;

else {

  OrderedHourlyEmployee otherOrderedHourlyEmployee = (OrderedHourlyEmployee)other;
      return (getPay() < otherOrderedHourlyEmployee.getPay());
     }
}
public boolean follows(Object other)
{
if (other ==  null)
return false;
else if (!(other instanceof OrderedHourlyEmployee))

 return false;
else {
OrderedHourlyEmployee otherOrderedHourlyEmployee = (OrderedHourlyEmployee)other;
  return (otherOrderedHourlyEmployee.precedes( this));
  }    }
 }
```

Activity 3:

An Example that shows How to create your own interface and implement it in abstract class

```
interface I1 {

   void methodI1(); // public static by default
}

interface I2 extends I1 {

   void methodI2(); // public static by default
}

class A1 {
   public String methodA1() {
           String strA1 = "I am in methodC1 of class A1";
           return strA1;
   }
   public String toString() {
           return "toString() method of class A1";
   }
}

class B1 extends A1 implements I2 {

   public void methodI1() {
           System.out.println("I am in methodI1 of class B1");
   }
   public void methodI2() {
           System.out.println("I am in methodI2 of class B1");
   }
}

class C1 implements I2 {

   public void methodI1() {
           System.out.println("I am in methodI1 of class C1");
   }
   public void methodI2() {
           System.out.println("I am in methodI2 of class C1");
   }
}

// Note that the class is declared as abstract as it does not
// satisfy the interface contract
abstract class D1 implements I2 {

   public void methodI1() {
   }
   // This class does not implement methodI2() hence declared abstract.
}

public class InterFaceEx {

   public static void main(String[] args) {
           I1 i1 = new B1();
           i1.methodI1(); // OK as methodI1 is present in B1
           // i1.methodI2(); Compilation error as methodI2 not present in I1
           // Casting to convert the type of the reference from type I1 to type I2
           ((I2) i1).methodI2();
           I2 i2 = new B1();
           i2.methodI1(); // OK
           i2.methodI2(); // OK
           // Does not Compile as methodA1() not present in interface reference I1
           // String var = i1.methodA1();
           // Hence I1 requires a cast to invoke methodA1
           String var2 = ((A1) i1).methodA1();
           System.out.println("var2 : " + var2);
           String var3 = ((B1) i1).methodA1();
           System.out.println("var3 : " + var3);
```

```
                String var4 = i1.toString();
                System.out.println("var4 : " + var4);
                String var5 = i2.toString();
                System.out.println("var5 : " + var5);
                I1 i3 = new C1();
                String var6 = i3.toString();
                System.out.println("var6 : " + var6); // It prints the Object toString() method
                Object o1 = new B1();
                // o1.methodI1(); does not compile as Object class does not define
                // methodI1()
                // To solve the probelm we need to downcast o1 reference. We can do it
                // in the following 4 ways
                ((I1) o1).methodI1(); // 1
                ((I2) o1).methodI1(); // 2
                ((B1) o1).methodI1(); // 3
                /*
                 *
                 * B1 does not have any relationship with C1 except they are "siblings".
                 *
                 * Well, you can't cast siblings into one another.
                 *
                 */
                // ((C1)o1).methodI1(); Produces a ClassCastException
        }
    }
```

3) Stage V (verify)

Home Activities:

Activity 1:

Create an interface EnhancedShape that extends Shape and also requires a method public double perimeter( ) to be implemented by any class that uses the interface.

Activity 2:

Write down a test application that implements EnhancedShape interface, call the interface methods to verify the functionality of implemented methods.

Activity 1:

Code Conflicting interfaces i.e. containing same constants with different values or exactly same method signatures in different interfaces, note down what happens if you try to implement conflicting interfaces.

4) Stage a_2 (assess)

Lab Assignment and Viva voce

Task 1

Define an interface named Shape with a single method named area that calculates the area of the geometric shape: public double area();

Task 2

Implement the Shape interface for Rectangle, Circle and Triangle class.

Task 3

Implement a class CalculateAreas that has a function that takes shape type array of objects and builds an array of (double values) values for each corresponding shapes.

LAB # 12

Statement Purpose:

Objective of this lab is to explore the Generic enable types (classes and interfaces) to be parameters when defining classes, interfaces and methods along with ArrayList data structure.

Activity Outcomes:

Students will be able to use effectively Arraylists and Generic types.

Instructor Note:

Students should have understanding about basic operations of Arrays.

Students should have understanding about the concept of user defined types (Classes).

5) Stage J (Journey)

Introduction

At its core, the term generics means parameterized types. Parameterized types are important because they enable you to create classes, interfaces, and methods in which the type of data upon which they operate is specified as a parameter. Using generics, it is possible to create a single class; that automatically works with different types of data.

A class, interface, or method that operates on a parameterized type is called generic, as in generic class or generic method. It is important to understand that Java has always given you the ability to create generalized classes, interfaces, and methods by operating through references of type Object. Because Object is the super class of all other classes, an Object reference can refer to any type object. Thus, in pre-generics code, generalized classes, interfaces, and methods used Object references to operate on various types of objects. The problem was that they could not do so with type safety.

Generics add the type safety that was lacking. They also streamline the process, because it is no longer necessary to explicitly employ casts to translate between Object and the type of data that is actually being operated upon. With generics, all casts are automatic and implicit. Thus, generics expand your ability to reuse code and let you do so safely and easily.

Non-Generic Box Class

Begin by examining a non-generic Box class that operates on objects of any type. It needs only to provide two methods: set, which adds an object to the box, and get, which retrieves it:

```
public class Box {
private Object object;

public void set(Object object) { this.object = object; }
public Object get() { return object; }
}
```

Since its methods accept or return an Object, you are free to pass in whatever you want, provided that it is not one of the primitive types. There is no way to verify, at compile time, how the class is used. One part of the code may place an Integer in the box and expect to get Integers out of it, while another part of the code may mistakenly pass in a String, resulting in a runtime error.

A Generic Version of the Box Class

A *generic class* is defined with the following format:

```
class name<T1, T2, ..., Tn> { /* ... */ }
```

The type parameter section, delimited by angle brackets (<>), follows the class name. It specifies the *type parameters* (also called *type variables*) T1, T2, ..., and Tn.

To update the Box class to use generics, you create a *generic type declaration* by changing the code "public class Box" to "public class Box<T>". This introduces the type variable, T, that can be used anywhere inside the class.

With this change, the Box class becomes:

```
/**
 * Generic version of the Box class.
 * @param<T> the type of the value being boxed
 */
public class Box<T> {
   // T stands for "Type"
private T t;

public void set(T t) { this.t = t; }
public T get() { return t; }
}
```

As you can see, all occurrences of Object are replaced by T. A type variable can be any **non-primitive** type you specify: any class type, any interface type, any array type, or even another type variable.

This same technique can be applied to create generic interfaces.

Type Parameter Naming Conventions

By convention, type parameter names are single, uppercase letters. This stands in sharp contrast to the variable naming conventions that you already know about, and with good reason: Without this convention, it would be difficult to tell the difference between a type variable and an ordinary class or interface name.

The most commonly used type parameter names are:

E - Element (used extensively by the Java Collections Framework)
K - Key
N - Number
T - Type
V - Value
S,U,V etc. - 2nd, 3rd, 4th types

Invoking and Instantiating a Generic Type

To reference the generic Box class from within your code, you must perform a *generic type invocation*, which replaces T with some concrete value, such as Integer:

```
Box <Integer> integerBox;
```

You can think of a generic type invocation as being similar to an ordinary method invocation, but instead of passing an argument to a method, you are passing a *type argument* — Integer in this case — to the Box class itself.

Like any other variable declaration, this code does not actually create a new Box object. It simply declares that integerBox will hold a reference to a "Box of Integer", which is how Box<Integer> is read.

An invocation of a generic type is generally known as a *parameterized type*.

To instantiate this class, use the new keyword, as usual, but place <Integer> between the class name and the parenthesis:

```
Box <Integer> integerBox = new Box <Integer> ();
```

Diamond Operator

In Java SE 7 and later, you can replace the type arguments required to invoke the constructor of a generic class with an empty set of type arguments (<> - referred as diamond operator) as long as the compiler can determine, or infer, the type arguments from the context. This pair of angle brackets, <>, is informally called *the diamond*. For example, you can create an instance of Box<Integer> with the following statement:

```
Box <Integer> integerBox = new Box <> ();
```

Java Arraylist:

The Java ArrayList is a dynamic array-like data structure that can grow or shrink in size during the execution of a program as elements are added/deleted. An Array on the other hand, has a fixed size: once we declared it to be a particular size, that size cannot be changed. To use an ArrayList, you first have to import the class:

```
import java.util.ArrayList;
```

You can then create a new ArrayList object:

```
ArrayListlistTest = new ArrayList( );
```

The Java API has a list of all the methods provided by an ArrayList.

6) Stage a1 (apply)

Lab Activities:

Activity 1:

The following program shows a simple use of ArrayList. An array list is created, and then objects of type String are added to it. The list is then displayed. Some of the elements are removed and the list is displayed again.

Solution:

```
importjava.util.*;

classArrayListDemo {

public static void main(String args[]) {

// create an array list

ArrayList al = new ArrayList();

System.out.println("Initial size of al: " +

al.size());

// add elements to the array list

al.add("C");

al.add("A");

al.add("E");

al.add("B");

al.add("D");

al.add("F");

al.add(1, "A2");

System.out.println("Size of al after additions: " +

al.size());

// display the array list

System.out.println("Contents of al: " + al);

// Remove elements from the array list

al.remove("F");

al.remove(2);

System.out.println("Size of al after deletions: " +

al.size());
System.out.println("Contents of al: " + al);

}

}
```

The output from this program is shown here:

```
Initial size of al: 0
Size of al after additions: 7
Contents of al: [C, A2, A, E, B, D, F]
Size of al after deletions: 5
Contents of al: [C, A2, E, B, D]
```

Activity 2:

The following program defines two classes. The first is the generic class Gen, and the second is GenDemo, which uses Gen.

Solution:

```
// A simple generic class.
// Here, T is a type parameter that
// will be replaced by a real type
// when an object of type Gen is created.

class Gen<T> {
}
T ob; // declare an object of type T

// Pass the constructor a reference to
// an object of type T.
Gen(T o) {
ob = o;
}

// Return ob.
T getob()
{ returnob;
}

// Show type of T.
voidshowType() {
System.out.println("Type of T is " +
ob.getClass().getName());}
}
```

```
// Demonstrate the generic class.
classGenDemo {
public static void main(String args[]) {

// Create a Gen reference for Integers.
Gen<Integer>iOb;
// Create a Gen<Integer> object and assign its
// reference to iOb.  Notice the use of autoboxing
// to encapsulate the value 88 within an Integer object.
iOb = new Gen<Integer>(88);
// Show the type of data used by iOb.
iOb.showType();
// Get the value in iOb. Notice that
// no cast is needed.
int v = iOb.getob();
System.out.println("value: " + v);
System.out.println();
// Create a Gen object for Strings.
Gen<String>strOb = new Gen<String>("Generics Test");
// Show the type of data used by strOb.
strOb.showType();
// Get the value of strOb. Again, notice
// that no cast is needed.
String str = strOb.getob();
System.out.println("value: " + str);}
}
```

The output produced by the program is shown here:

```
Type of T is java.lang.Integer
value: 88
Type of T is java.lang.String
value: Generics Test
```

Activity 3:

You can declare more than one type parameter in a generic type. To specify two or more type parameters, simply use a comma-separated list. For example, the following TwoGenclass is a variation of the Gen class that has two type parameters:

```
// A simple generic class with two type
// parameters: T and V.
classTwoGen<T, V> {
T ob1;
V ob2;
// Pass the constructor a reference to
// an object of type T and an object of type V.
TwoGen(T o1, V o2) {
ob1 = o1;
ob2 = o2;}
// Show types of T and V.
voidshowTypes() {
System.out.println("Type of T is " +
ob1.getClass().getName());
System.out.println("Type of V is " +
ob2.getClass().getName());}
T getob1() {
return ob1;
}
V getob2() {
return ob2;
// Demonstrate TwoGen.
classSimpGen {
public static void main(String args[]) {
}
}
TwoGen<Integer, String>tgObj =
newTwoGen<Integer, String>(88, "Generics");
// Show the types.
tgObj.showTypes();
// Obtain and show values.
int v = tgObj.getob1();
System.out.println("value: " + v);
String str = tgObj.getob2();
System.out.println("value: " + str);
}
}
```

The output from this program is shown here:

```
Type of T is java.lang.Integer
Type of V is java.lang.String
value: 88
value: Generics
```

7) Stage V (verify)

Home Activities:

Activity 1:

Write a program that uses an ArrayList of parameter type Contact to store a database of contacts. The Contact class should store the contact's first and last name, phone number, and email address. Add appropriate accessor and mutator methods. Your database program should present a menu that allows the user to add a contact, display all contacts, search for a specific contact and display it, or search for a specific contact and give the user the option to delete it. The searches should find any contact where any instance variable contains a target search string. For example, if “elmore” is the search target, then any contact where the first name, last name, phone number, or email address contains “elmore” should be returned for display or deletion. Use the “for-each” loop to iterate through the ArrayList .

Activity 2:

Write a generic class, MyMathClass , with a type parameter T where T is a numeric object type (e.g., Integer, Double, or any class that extends java.lang.Number). Add a method named standardDeviation that takes an ArrayList of type T and returns as a double the standard deviation of the values in the ArrayList . Use the doubleValue () method in the Number class to retrieve the value of each number as a double. Refer to Programming Project 6.5 for a definition of computing the standard deviation. Test your method with suitable data. Your program should generate a compile-time error if your standard deviation method is invoked on an ArrayList that is defined for nonnumeric elements (e.g., Strings).

8) Stage a2 (assess)

Lab Assignment and Viva voce

Task 1
Create a generic class with a type parameter that simulates drawing an item at random out of a box. This class could be used for simulating a random drawing. For example, the box might contain Strings representing names written on a slip of paper, or the box might contain Integers representing a random drawing for a lottery based on numeric lottery picks.

Create an add method that allows the user of the class to add an object of the specified type along with an isEmpty method that determines whether or not the box is empty. Finally, your class should have a drawItem method that randomly selects an object from the box and returns it.
If the user attempts to drawn an item out of an empty box, return null . Write a main method that tests your class.

Task 2
In the sport of diving, seven judges award a score between 0 and 10, where each score may be a floating-point value. The highest and lowest scores are thrown out and the remaining scores are added together. The sum is then multiplied by the degree of difficulty for that dive. The degree of difficulty ranges from 1.2 to 3.8 points. The total is then multiplied by 0.6 to determine the diver’s score.

Write a computer program that inputs a degree of difficulty and seven judges’ scores and outputs the overall score for that dive. The program should use an ArrayList of type Double to store the scores.

LAB # 13

Statement Purpose:

1. Understanding exception-throwing methods.
2. Using try-catch to handle exceptions.
3. Understanding and writing custom exceptions.

Activity Outcomes:

This lab teaches you the following topics:

- Exception handling
- (try, catch, finally, throw and throws) to handle exceptions.

Instructor Note:

As pre-lab activity, read Chapter 9 from the book, (Absolute Java, Savitch, W. & Mock, K., 5th Edition (2012), Pearson.), and also as given by your theory instructor.

1) Stage J (Journey)

Introduction

A Java exception is an object that describes an exceptional (that is, error) 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. That method may choose to handle the exception itself, or pass it on. Either way, at some point, the exception is *caught* and processed. Exceptions can be generated by the Java run-time system, or they can be manually generated by your code. Exceptions thrown by Java relate to fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment. Manually generated exceptions are typically used to report some error condition to the caller of a method. Java exception handling is managed via five keywords: **try**, **catch**, **throw**, **throws**, and **finally.**

Exception-Throwing Methods

Runtime errors appear in Java as exceptions, exception is a special type of classes that could be thrown to indicate a runtime error and provide additional data about that error. If a method is declared to throw an exception, any other method calls it should deal with this exception by throwing it (if it appears) or handle it. Recalling reading from user using BufferedReader class, we used to declare main method from which we call readLine() using throws IOException, this because readLine() is declared to throw that exception.

```
import java.io.*;
class ReadFromUser{
  public static void main(String[] args) throws IOException{
    .
    .
    .
  }
}
```

If we wish to write a method that simplifies reading from user, you may want to declare it to throw IOException.

```
import java.io.*;
class ReadFromUser{
  public static void main(String[] args) throws IOException{
    String in = read("Enter your name: ");
  }
  public static String read(String message) throws IOException{
    System.out.print(message);
    BufferedReader in = new BufferedReader(   new InputStreamReader(System.in));
    String input = in.readLine();
    return input;
  }
}
```

In the previous code, if an exception occurs in readLine() method, this exception is thrown as it is to read method, which originally called it, because this last one is also declared to throw that exception, it also throws it as it is to main method, which originally called it, finally, the exception is throws to JVM which prints it out on the screen so the user can know there was error.

try-catch **Exception Handling**

Another technique for handling runtime errors is to use try-catch block to handle different types of exceptions and take appropriate action instead of throwing them to the user. The format of try-catch block is the following.

```
try{
  //Statement(s) that may throw exceptions
}catch(Exception e){
  //Statement(s) to handle exception.
}
```

For example, we can place readLine() method which throws IOException in a try-catch block as the following.

```
BufferedReader in = new BufferedReader new InputStreamReader(System.in));
String input;
try{
  input = in.readLine();
}catch(IOException e){
  System.out.println("Error occurred");
}
```

When we are expecting more than one exception, we can use several catch blocks, each one for different type of exceptions. For example, when reading a string using BufferedReader and converting it to integer, you can expect two different exceptions: IOException and NumberFormatException which occurs when the provided string cannot be converted to integer.

```
BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
String input;
try{
  input = in.readLine();
  int x = Integer.parseInt(input);
}catch(IOException e){
  System.out.println("Error reading input");
}catch(NumberFormatException err){
  System.out.println("This is not a valid number");
}
```

Finally

You can attach a finally-clause to a try-catch block. The code inside the finally clause will always be executed, even if an exception is thrown from within the try or catch block. If your code has a return statement inside the try or catch block, the code inside the finally-block will get executed before returning from the method. Here is how a finally clause looks:

```
public void openFile(){
      FileReader reader = null;
```

```
        try {
            reader = new FileReader("someFile");
            int i=0;
            while(i != -1){
                i = reader.read();
                System.out.println((char) i );
            }
        } catch (IOException e) {
            //do something clever with the exception
        } finally {
            if(reader != null){
                try {
                    reader.close();
                } catch (IOException e) {
                    //do something clever with the exception
                }
            }
            System.out.println("--- File End ---");
        }
    }
```

No matter whether an exception is thrown or not inside the try or catch block the code inside the finally-block is executed. The example above shows how the file reader is always closed, regardless of the program flow inside the try or catch block.

Note: If an exception is thrown inside a finally block, and it is not caught, then that finally block is interrupted just like the try-block and catch-block is. That is why the previous example had the reader.close() method call in the finally block wrapped in a try-catch block:

```
    finally {
            if(reader != null){
                try {
                    reader.close();
                } catch (IOException e) {
                    //do something clever with the exception
                }
            }
            System.out.println("--- File End ---");
        }
```

2) Stage a1 (apply)

Lab Activities:

Activity 1:

The example below shows the procedure for catching IndexOutOfBounds and InputMismatch exception.

.

Solution:

```
 importjava.util.InputMismatchException;
 importjava.util.Scanner;
 public class Marks {
 public static void main(String args[]){
        Scanner s=new Scanner(System.in);
        try{
        int[] marks = new int[5];
        for(inti=0;i<=5;i++){
        System.out.println("Enter marks for "+i+"st subjects :");
        marks[i]=s.nextInt();
        }
        }

        catch(InputMismatchException h){
                System.out.println("Please enter correct number!");
        }
 catch(ArrayIndexOutOfBoundsException e){
        System.out.println("The error is"+e);

 }
 }
 }
```

Activity 2:

The example below shows the procedure for using throw keyword.

Solution:

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

                throw new Exception();
        }
        catch(Exception e){
                System.out.println("Error!!!");
        }

}

}
```

Activity 3:

In some cases while developing our own applications, we need to specify custom types of exceptions to handle custom errors that may occur during program execution. A custom exception is a class that *inherits* Exception class or any of its subclasses, since inheritance is beyond the scope of this course, we will define it as using extends Exception in class declaration.

Solution:

```
class MyCustomException extends Exception{

  private String message;
  public MyCustomException(String message){
    this.message = message;
  }
  public String toString(){
    return message;
  }
}
```

To use your custom exception, declare an object of it and throw that object using throw keyword. It is optional to declare the method containing throw statement with throws keyword. In the following example, the program reads student id, this id should be of length 7 and consists only of digits, otherwise it throws an exception.

```
class InvalidIDException extends Exception{

  private String message;
  public InvalidIDException(String message){
    this.message = message;
  }
  public String toString(){
    return message;
  }
}

Import javax.swing.*;
class StudentsData{
  public static void main(String[] args){
    String id, name;
    name = JOptionPane.showInputDialog("Enter student name");
    id = JOptionPane.showInputDialog("Enter student ID");

    try{
      verfyID(id);
    }
    catch(InvalidIDException e){
      JoptionPane.showMessageDialog(null, e.toString());
    }
}
  public static void verifyID(String id)
                   throws InvalidIDException{
    if(id.length() != 7){
      throw new InvalidIDException("Check ID length");
    }
    try{
      Long.parseLong(id);
    }
    catch(Exception err){
      throw
        new InvalidIDException("ID can contain only digits");
    }
  }
}
```

Activity 4:

The example below shows the procedure for using throws keyword.

Solution:

```
//The following class represents the exception to be thrown:
public class illegalamount extends Exception {

        public illegalamount(){
                super();
        }

        public illegalamount(String a){
                super(a);
        }

}
```

//Account Class' methods deposit and withdraw will throw an exception of type illegalamount if the amount variable is not in the valid range.

```
public class Account {

        private double balance;

        public Account()
        {
        balance = 0;
        }
        public Account( double initialDeposit)
        {
        balance = initialDeposit;
        }
        public double getBalance()
        {
        return balance;
        }



        public double deposit( double amount)throws illegalamount{
```

```
        if (amount > 0){
        balance += amount;}

        else{
                throw new illegalamount("error in deposit");
        }
        return balance;
        }


        public double withdraw(double amount) throws illegalamount
        {
        if  ((amount > balance) || (amount < 0)){
                balance -= amount;}
        else{
                throw new illegalamount("error in withdraw");
        }
        return balance;
        }

}
```

3) Stage V (verify)

Home Activities:

Activity 1:
Create a new Java file and:

1. Create an exception class named NegativeNumberException extended from the class Exception. This class should contain a parameterless constructor to call the Exception class constructor with the message "You should enter a positive number".
2. Create a class named exercise3 that contains:
    a) A method named M1 that accepts a double variable as argument and returns its square root. The method should throw the NegativeNumberException exception defined above.
    b) The main method to prompt the user to enter a number and then to call the method M1 (insert a try-catch block to handle the NegativeNumberException exception)

Activity 2:
Write a program that calculates the average of N integers. The program should prompt the user to enter the value for *N* and then afterward must enter all N numbers. If the user enters a negative value for *N*, then an exception should be thrown (and caught) with the message " *N* must be positive." If there is any exception as the user is entering the *N* numbers, an error message should be displayed, and the user prompted to enter the number again.

4) Stage a_2 (assess)

Lab Assignment and Viva voce

Task 1:

Write a Java program to:

1. Read a number from the keyboard.
2. Write a try-throw-catch bloc to calculate the square root of the entered number. An exception should be handled if the entered number is negative. Use the Exception class.

Task 2:

Write a program that:

1. Create an exception class named AgeOutOfRangeException extended from the class Exception. This class should contain a constructor with no parameter to call the Exception class constructor with the message "You are older than the requested age (25 years)".
2. Create an exception class named LowGpaException extended from the class Exception. This class should contain a constructor with no parameter to call the Exception class constructor with the message "Your GPA is not sufficient to apply for this job (2.5)".
3. Create a main class to prompt the user to enter his/her age and his GPA. The user application for a job will be rejected either if his age is greater than 25 years or his GPA is less than 2.5. You should declare two nested try-throw-catch blocks; one to handle the AgeOutOfRangeException and the other to handle the LowGpaException. If the user enters acceptable age and GPA, display the message "Your application is accepted and is under study".

LAB # 14

Statement Purpose:

The purpose of lab is to make students understand ways of getting information in and out of your Java programs, using files.

Activity Outcomes:

Students will be able to retrieve create file.

Students will be able to write and read data to and from a file.

Students will learn about Object Serialization.

Instructor Note:

The student should be able to declare a class.

The student should have understanding about Exception Handling.

.

1) Stage J (Journey)

Introduction

To read an entire object from or write an entire object to a file, Java provides object serialization. A serialized object is represented as a sequence of bytes that includes the object's data and its type information. After a serialized object has been written into a file, it can be read from the file and deserialized to recreate the object in memory.

A class that implements the Serializable interface is said to be a **serializable** class. To use objects of a class with writeObject() and readObject() , that class must be serializable. But to make the class serializable, we change nothing in the class. All we do is add the phrase implements Serializable . This phrase tells the run-time system that it is OK to treat objects of the class in a particular way when doing file I/O.

Classes ObjectInputStream and ObjectOutputStream, which respectively implement the ObjectInput and ObjectOutput interfaces, enable entire objects to be read from or written to a stream.

To use serialization with files, initialize ObjectInputStream and ObjectOutputStream objects with FileInputStream and FileOutputStream objects

ObjectOutput interface method writeObject takes an Object as an argument and writes its information to an OutputStream.

A class that implements ObjectOuput (such as ObjectOutputStream) declares this method and ensures that the object being output implements Serializable.

ObjectInput interface method readObject reads and returns a reference to an Object from an InputStream. After an object has been read, its reference can be cast to the object's actual type.

```
.  /**
   Demonstrates binary file I/O of serializable class objects.
  */
 public class ObjectIODemo
  {
 public static void main(String[] args)
    {
 try
      {
ObjectOutputStreamoutputStream =
 new ObjectOutputStream(new FileOutputStream("datafile"));
SomeClassoneObject = new SomeClass(1, 'A');
SomeClassanotherObject = new SomeClass(42, 'Z');
outputStream.writeObject(oneObject);
outputStream.writeObject(anotherObject);
outputStream.close();
System.out.println("Data sent to file.");
  }
 catch(IOException e)
```

```
  {
System.out.println("Problem with file output.");
   }
System.out.println(
                "Now let's reopen the file and display the data.");

try
        {
ObjectInputStreaminputStream =
 new ObjectInputStream(new FileInputStream("datafile"));
Notice the type casts.
SomeClassreadOne = (SomeClass)inputStream.readObject( );
SomeClassreadTwo = (SomeClass)inputStream.readObject( );
System.out.println("The following were read from the file:");
System.out.println(readOne);
System.out.println(readTwo);
  }
 catch(FileNotFoundException e)
 {
System.out.println("Cannot find datafile.");
  }
 catch(ClassNotFoundException e)
        {
System.out.println("Problems with file input.");
        }

catch(IOException e)
        {
System.out.println("Problems with file input.");
        }
System.out.println("End of program.");
      }
  }
```

2) Stage a1 (apply)

Lab Activities:

Activity 1:

The following example demonstrates writing of objects to a file.

Solution:

```
 import java.io.*;


   public static class Person implements Serializable
 {
       public String name = null;
       public int    age  =   0;
   }
```

```
public class ObjectOutputStreamExample {

    public void writeObject() {

try
{
        ObjectOutputStream objectOutputStream =
            new ObjectOutputStream(new FileOutputStream("filename"));

        Person p = new Person();
        person.name = "Jakob Jenkov";
        person.age  = 40;

        objectOutputStream.writeObject(p);

    }
         catch (ClassNotFoundException ex) {
            ex.printStackTrace();
        } catch (FileNotFoundException ex) {
            ex.printStackTrace();
        } catch (IOException ex) {
            ex.printStackTrace();
        }
finally
{
            //Close the OutputStream
            try {
                if (objectOutputStream!= null) {
                    objectOutputStream.close();
                }
            } catch (IOException ex) {
                ex.printStackTrace();
            }
 }

}
}
```

Activity 2:

The following example demonstrates reading of all objects from a file.

Solution:

```
import java.io.*;


public class ObjectInputStreamExample {

public void readObjects()
{

try
{
    ObjectInputStream objectInputStream =
            new ObjectInputStream(new FileInputStream("filename"));

while (true)
{


        Person personRead = (Person) objectInputStream.readObject();
       System.out.println(personRead.name);
        System.out.println(personRead.age);

}

 }
catch (EOFException ex) {  //This exception will be caught when EOF is reached
            System.out.println("End of file reached.");
        } catch (ClassNotFoundException ex) {
            ex.printStackTrace();
        } catch (FileNotFoundException ex) {
            ex.printStackTrace();
        } catch (IOException ex) {
            ex.printStackTrace();
        }
 finally
{
            //Close the ObjectInputStream
            try {
                if (objectInputStream!= null) {
                    objectInputStream.close();
                }
            } catch (IOException ex) {
                ex.printStackTrace();
            }
 }
    }

}
```

Activity 3:

The following example demonstrates searching of an object from a file.

Solution:

```
import java.io.*;


public class ObjectInputStreamExample {

public void searchObject(String name)
{

try
{
   ObjectInputStream objectInputStream =
         new ObjectInputStream(new FileInputStream("filename"));

while (true)
{

      Person personRead = (Person) objectInputStream.readObject();

If (personRead.name.equals(name)
            {
             System.out.println(personRead.name);
             System.out.println(personRead.age);
            break;
            }

}

 }
catch (EOFException ex) {  //This exception will be caught when EOF is reached
         System.out.println("End of file reached.");
      } catch (ClassNotFoundException ex) {
         ex.printStackTrace();
      } catch (FileNotFoundException ex) {
         ex.printStackTrace();
      } catch (IOException ex) {
         ex.printStackTrace();
      }
 finally
{
         //Close the ObjectInputStream
         try {
            if (objectInputStream!= null) {
               objectInputStream.close();
             }
         } catch (IOException ex) {
            ex.printStackTrace();
         }
 }
   }
}
```

3) Stage v (verify)

<u>Home Activities:</u>

Activity 1:

Create a class Book that has name(String), publisher (String) and an author (Person). Write five objects of Book Class in a file named “BookStore”.

Activity 2:

Consider the Book class of Activity 1 and write a function that displays all Books present in file “BookStore”.

Activity 3:

Consider the Book class of Activity 1 and write a function that asks the user for the name of a Book and searches the record against that book in the file “BookStore”.

4) Stage a2 (assess)

<u>Assignment:</u>

Create an ATM System with **Account** as the Serializable class. Write ten objects of **Account** in a file. Now write functions for withdraw, deposit, transfer and balance inquiry.

Note:

a. Each function should ask for the account number on which specific operation should be made.
b. All changes in Account object should be effectively represented in the file.

LAB # 15

Statement Purpose:

The purpose of lab is to make students understand basic concepts of Layouts of GUI in Java. The students will learn about the three basic types of layouts and understand the difference between them.

Activity Outcomes:

Students will be able to create frames with different layouts.

Students will be able to create simple to medium level complex GUI

Instructor Note:

The student should have understanding related to basic GUI components

1) Stage J (Journey)

Introduction

In many other window systems, the user-interface components are arranged by using hardcoded pixel measurements. For example, put a button at location (10, 10) in the window using hard-coded pixel measurements, the user interface might look fine on one system but be unusable on another. Java's layout managers provide a level of abstraction that automatically maps your user interface on all window systems.

The Java GUI components are placed in containers, where they are arranged by the container's layout manager. In the preceding program, you did not specify where to place the OK button in the frame, but Java knows where to place it, because the layout manager works behind the scenes to place components in the correct locations. A layout manager is created using a layout manager class.

Layout managers are set in containers using the SetLayout(aLayoutManager) method. For example, you can use the following statements to create an instance of XLayout and set it in a container:

LayoutManagerlayoutManager = new XLayout();

container.setLayout(layoutManager);

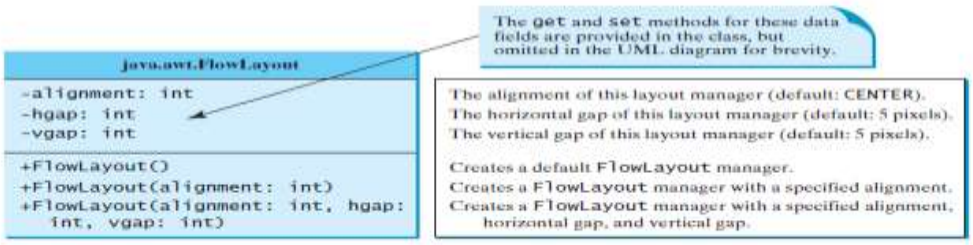
2) Stage a1 (apply)

Lab Activities:

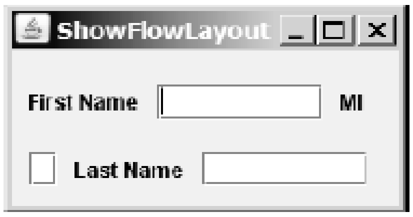
Activity 1:

FlowLayout

FlowLayout is the simplest layout manager. The components are arranged in the container from left to right in the order in which they were added. When one row is filled, a new row is started. You can specify the way the components are aligned by using one of three constants: FlowLayout.RIGHT, FlowLayout.CENTER, or FlowLayout.LEFT. You can also specify the gap between components in pixels. The class diagram for FlowLayout is shown in Figure below


java.awt.FlowLayout
-alignment: int
-hgap: int
-vgap: int
+FlowLayout()
+FlowLayout(alignment: int)
+FlowLayout(alignment: int, hgap: int, vgap: int)
The get and set methods for these data fields are provided in the class, but omitted in the UML diagram for brevity.
The alignment of this layout manager (default: CENTER).
The horizontal gap of this layout manager (default: 5 pixels).
The vertical gap of this layout manager (default: 5 pixels).
Creates a default FlowLayout manager.
Creates a FlowLayout manager with a specified alignment.
Creates a FlowLayout manager with a specified alignment, horizontal gap, and vertical gap.

Create the following frame using Flow Layout.



(a)

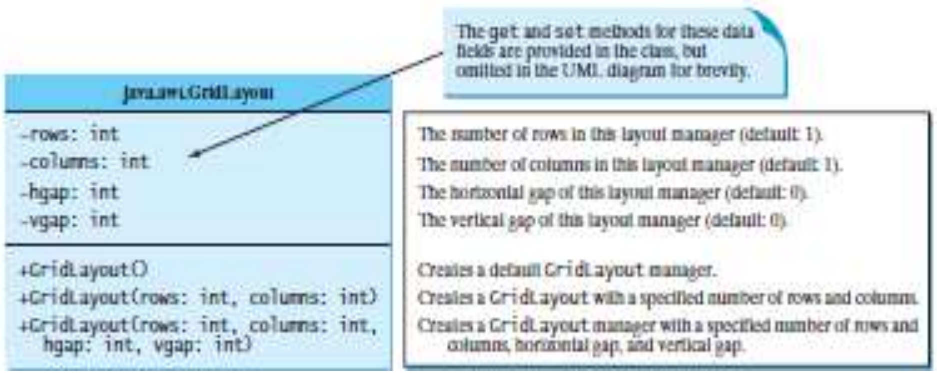
Solution:

```java
import javax.swing.JLabel;
import javax.swing.JTextField;
import javax.swing.JFrame;


import java.awt.FlowLayout;

public class ShowFlowLayout extends JFrame {
  public ShowFlowLayout() {
    // Set FlowLayout, aligned left with horizontal gap 10
    // and vertical gap 20 between components
    setLayout(new FlowLayout(FlowLayout.LEFT, 10, 20));

    // Add labels and text fields to the frame
    add(new JLabel("First Name"));
    add(new JTextField(8));
    add(new JLabel("MI"));
    add(new JTextField(1));
    add(new JLabel("Last Name"));
    add(new JTextField(8));
  }

  /** Main method */
  public static void main(String[] args) {
    ShowFlowLayout frame = new ShowFlowLayout();
    frame.setTitle("ShowFlowLayout");
    frame.setSize(200, 200);
    frame.setLocationRelativeTo(null); // Center the frame
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setVisible(true);
  }
}
```

Activity 2:

GridLayout

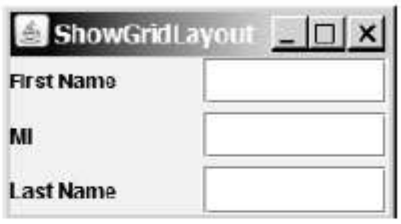
The GridLayout manager arranges components in a grid (matrix) formation. The components are placed in the grid from left to right, starting with the first row, then the second, and so on, in the order in which they are added. The class diagram for GridLayout is shown in Figure below.



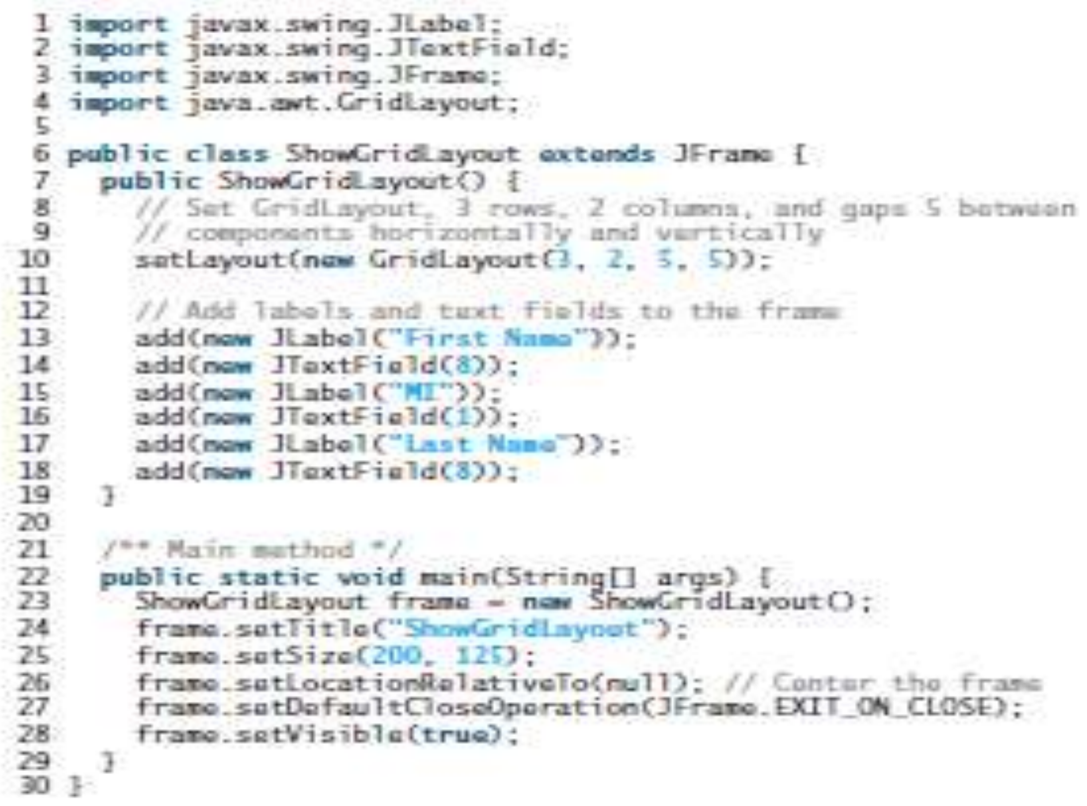
The get and set methods for these data fields are provided in the class, but omitted in the UML diagram for brevity.

java.awt.GridLayout	
-rows: int	The number of rows in this layout manager (default: 1).
-columns: int	The number of columns in this layout manager (default: 1).
-hgap: int	The horizontal gap of this layout manager (default: 0).
-vgap: int	The vertical gap of this layout manager (default: 0).
+GridLayout()	Creates a default GridLayout manager.
+GridLayout(rows: int, columns: int)	Creates a GridLayout with a specified number of rows and columns.
+GridLayout(rows: int, columns: int, hgap: int, vgap: int)	Creates a GridLayout manager with a specified number of rows and columns, horizontal gap, and vertical gap.

Create the following frame using Grid Layout



Solution:



```java
import javax.swing.JLabel;
import javax.swing.JTextField;
import javax.swing.JFrame;
import java.awt.GridLayout;

public class ShowGridLayout extends JFrame {
  public ShowGridLayout() {
    // Set GridLayout, 3 rows, 2 columns, and gaps 5 between
    // components horizontally and vertically
    setLayout(new GridLayout(3, 2, 5, 5));

    // Add labels and text fields to the frame
    add(new JLabel("First Name"));
    add(new JTextField(8));
    add(new JLabel("MI"));
    add(new JTextField(1));
    add(new JLabel("Last Name"));
    add(new JTextField(8));
  }

  /** Main method */
  public static void main(String[] args) {
    ShowGridLayout frame = new ShowGridLayout();
    frame.setTitle("ShowGridLayout");
    frame.setSize(200, 125);
    frame.setLocationRelativeTo(null); // Center the frame
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setVisible(true);
  }
}
```

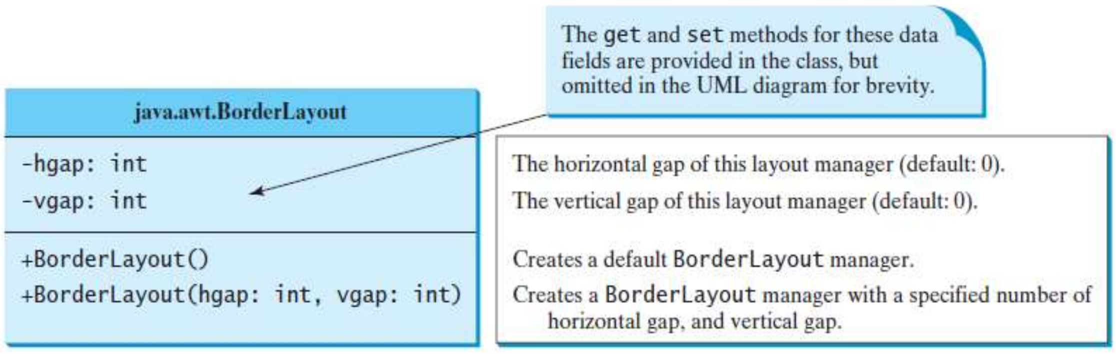
Activity 3:

BorderLayout

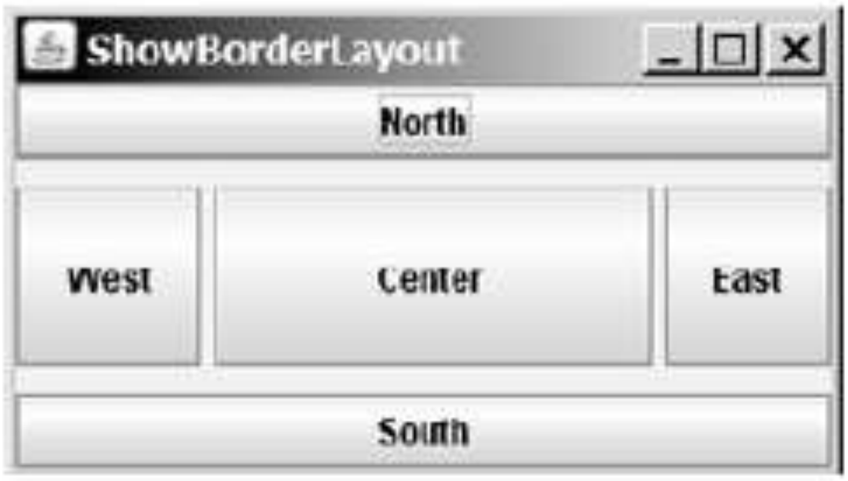
The BorderLayout manager divides a container into five areas: East, South, West, North, and Center. Components are added to a BorderLayout by using add(Component, index), where index is a constant as mentioned below:

- BorderLayout.EAST,
- BorderLayout.SOUTH,
- BorderLayout.WEST,
- BorderLayout.NORTH,
- BorderLayout.CENTER.

The class diagram for BorderLayout is shown in Figure below:


The get and set methods for these data fields are provided in the class, but omitted in the UML diagram for brevity.
java.awt.BorderLayout
-hgap: int
-vgap: int
+BorderLayout()
+BorderLayout(hgap: int, vgap: int)
The horizontal gap of this layout manager (default: 0).
The vertical gap of this layout manager (default: 0).
Creates a default BorderLayout manager.
Creates a BorderLayout manager with a specified number of horizontal gap, and vertical gap.

Run the below code and ensure that the output matches the UI below the code.



```java
import javax.swing.JButton;
import javax.swing.JFrame;
import java.awt.BorderLayout;

public class ShowBorderLayout extends JFrame {
  public ShowBorderLayout() {
    // Set BorderLayout with horizontal gap 5 and vertical gap 10
    setLayout(new BorderLayout(5, 10));

    // Add buttons to the frame
    add(new JButton("East"), BorderLayout.EAST);
    add(new JButton("South"), BorderLayout.SOUTH);
    add(new JButton("West"), BorderLayout.WEST);
    add(new JButton("North"), BorderLayout.NORTH);
    add(new JButton("Center"), BorderLayout.CENTER);
  }

  /** Main method */
  public static void main(String[] args) {
    ShowBorderLayout frame = new ShowBorderLayout();
    frame.setTitle("ShowBorderLayout");
    frame.setSize(300, 200);
    frame.setLocationRelativeTo(null); // Center the frame
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setVisible(true);
  }
}
```

Panels as SubContainers

Suppose that you want to place ten buttons and a text field in a frame. The buttons are placed in grid formation, but the text field is placed on a separate row. It is difficult to achieve the desired look by placing all the components in a single container. With Java GUI programming, you can divide a window into panels. Panels act as subcontainers to group user-interface components.

You add the buttons in one panel, then add the panel into the frame. The Swing version of panel is JPanel. You can use new JPanel() to create a panel with a default FlowLayout manager or new JPanel(LayoutManager) to create a panel with the specified layout manager.

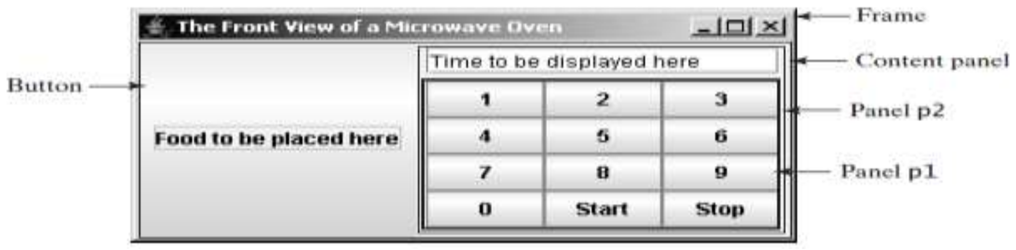
Use the add(Component) method to add a component to the panel. For example, the following code creates a panel and adds a button to it:

```java
JPanel p = new JPanel();

p.add(new JButton("OK"));
```

Panels can be placed inside a frame or inside another panel. The following statement places panel p into frame f:

```java
f.add(p);
```

Run the below code and ensure that the output is similar to below:



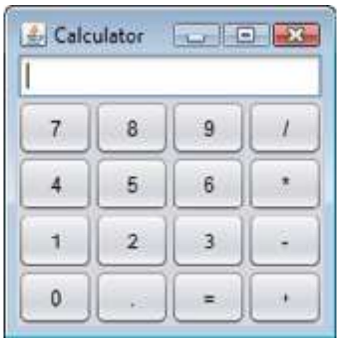
```java
import java.awt.*;
import javax.swing.*;

public class TestPanels extends JFrame {
  public TestPanels() {
    // Create panel p1 for the buttons and set GridLayout
    JPanel p1 = new JPanel();
    p1.setLayout(new GridLayout(4, 3));


    // Add buttons to the panel
    for (int i = 1; i <= 9; i++) {
      p1.add(new JButton("" + i));
    }

    p1.add(new JButton("" + 0));
    p1.add(new JButton("Start"));
    p1.add(new JButton("Stop"));

    // Create panel p2 to hold a text field and p1
    JPanel p2 = new JPanel(new BorderLayout());
    p2.add(new JTextField("Time to be displayed here"),
      BorderLayout.NORTH);
    p2.add(p1, BorderLayout.CENTER);

    // add contents into the frame
    add(p2, BorderLayout.EAST);
    add(new JButton("Food to be placed here"),
      BorderLayout.CENTER);
  }

  /** Main method */
  public static void main(String[] args) {
    TestPanels frame = new TestPanels();
    frame.setTitle("The Front View of a Microwave Oven");
    frame.setSize(400, 250);
    frame.setLocationRelativeTo(null); // Center the frame
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setVisible(true);
  }
}
```

3) Stage v (verify)

Home Activities:

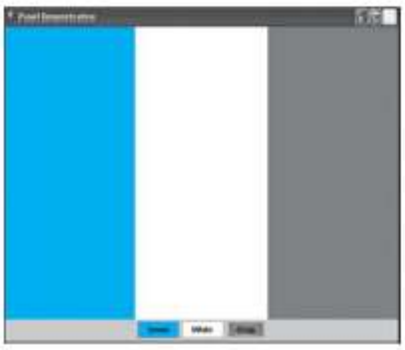
Activity 1:

Create the following GUI. You do not have to provide any functionality.



Activity 2:

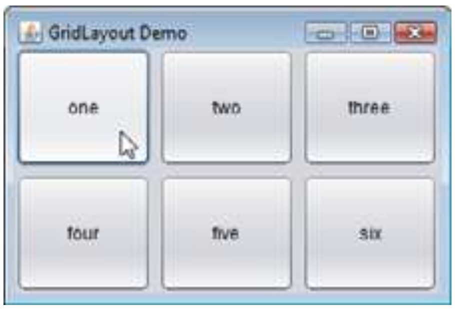
Create the following GUI. You do not have to provide any functionality.



4) Stage a2 (assess)

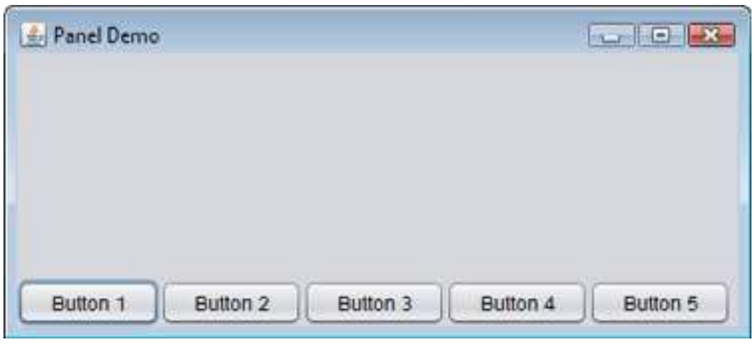
Assignment 1:

Create the following GUI. You do not have to provide any functionality.



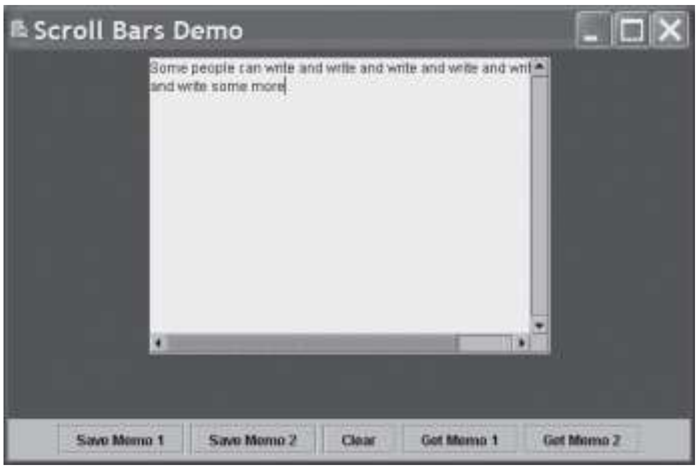
Assignment 2:

Create the following GUI. You do not have to provide any functionality.



Assignment 3:

Create the following GUI. You do not have to provide any functionality.



Some people can write and write and write and write and wri
and write some more

Note: Use **JScrollPane** class for the creating the scroll pane.

LAB # 16

Statement Purpose:

In this lab, student will learn and practice the basic concepts of events based programming in GUI based interfaces in Java. They will learn event generation and event handling.

Activity Outcomes:

After completing this lesson, you should be able to do the following:

- Understand why events are needed in GUI
- Understand the mechanics of event generation and event handling
- Practice simple event based programming
- Create a simple but useful GUI based program

Instructor Note:

The student should have understanding related to basic GUI components and Layouts.

1) Stage J (Journey)

Introduction

Any program that uses GUI (graphical user interface) such as Java application written for windows, is event driven. Event describes the change of state of any object.

Example : Pressing a button, Entering a character in Textbox

Event handling has three main components,

- **Events :** An event is a change of state of an object.
- **Events Source :** Event source is an object that generates an event.
- **Listeners :** A listener is an object that listens to the event. A listener gets notified when an event occurs.

A source generates an Event and sends it to one or more listeners registered with the source. Once event is received by the listener, they processe the event and then return. Events are supported by a number of Java packages, like **java.util**, **java.awt** and **java.awt.event**.

Important Event Classes and Interface

Event Classe	**Description**	**Listener Interface**
ActionEvent	generated when button is pressed, menu-item is selected, list-item is double clicked	ActionListener
MouseEvent	generated when mouse is dragged, moved,clicked,pressed or released also when the enters or exit a component	MouseListener
KeyEvent	generated when input is received from keyboard	KeyListener
ItemEvent	generated when check-box or list item is clicked	ItemListener

TextEvent	generated when value of textarea or textfield is changed	TextListener
MouseWheelEvent	generated when mouse wheel is moved	MouseWheelListener
WindowEvent	generated when window is activated, deactivated, deiconified, iconified, opened or closed	WindowListener
ComponentEvent	generated when component is hidden, moved, resized or set visible	ComponentEventListener
ContainerEvent	generated when component is added or removed from container	ContainerListener
AdjustmentEvent	generated when scroll bar is manipulated	AdjustmentListener
FocusEvent	generated when component gains or loses keyboard focus	FocusListener

2) Stage a1 (apply)

Lab Activities:

Activity 1:

Run the below code. It should create a label and a button. The label should have text "Hello" but when the button the pressed the text changes to "Bye"

```
Import java.awt.FlowLayout;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

import javax.swing.*;

public class test extends JFrame {

        privateJLabel label;

        privateJButton b;
```

```
          public test(){

                    this.setLayout(newFlowLayout(FlowLayout.LEFT,10,20));

                    label=newJLabel("Hello");

                    this.add(label);

                    b=newJButton("Toggle");

                    b.addActionListener(newmyHandler());

                    add(b);

                    }

Clas smyHandler implements ActionListener{


                    Public void actionPerformed(ActionEvent e){

                              label.setText("Bye");

                    }

          }

Public static void main(String[] args) {

                    // TODO Auto-generated method stub

                    test f=new test();

                    f.setTitle("Hi and Bye");

                    f.setSize(400, 150);

                    f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

                    f.setLocationRelativeTo(null);;

                    f.setVisible(true);

          }

}
```

Activity 2:

Run and understand the below code. Basically this code sets the text in label on button press event. Any text entered in the textfield is copied into the label. Ensure that it is so and understand how it works.

```
Import java.awt.FlowLayout;

Import java.awt.event.ActionEvent;

Import java.awt.event.ActionListener;

Import javax.swing.*;


Public class test extendsJFrame {

        Private JTextField tf1;

        Private JLabel label;

        Private JButton b;


        public test(){

                this.setLayout(newFlowLayout(FlowLayout.LEFT,10,20));

                tf1=newJTextField(8);

                this.add(tf1);

                label=newJLabel("New Text");

                this.add(label);

                b=newJButton("Change");

                b.addActionListener(newmyHandler());

                add(b);

                }

Class myHandler implements ActionListener{

                Public void actionPerformed(ActionEvent e){

                        String s=tf1.getText();

                        label.setText(s);
```

```
                    tf1.setText("");
                }
            }
Public static void main(String[] args) {
                // TODO Auto-generated method stub
                test f=new test();
                f.setTitle("Copy Text");
                f.setSize(400, 150);
                f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                f.setLocationRelativeTo(null);;
                f.setVisible(true);
            }
}
```

Activity 3:

Run and understand the below code. We now first see which button triggered the event through the getSource event and then either disapper one button or copy text in the TextField into the label.

```
Import java.awt.FlowLayout;
Import java.awt.event.ActionEvent;
Import java.awt.event.ActionListener;
Import javax.swing.*;


Public class test extends JFrame {
            Private JTextField tf1;
            Private JLabel label;
            Private JButton b;
            Private JButton b1;
```

```
        public test(){
                this.setLayout(newFlowLayout(FlowLayout.LEFT,10,20));
                tf1=newJTextField(8);
                this.add(tf1);
                label=newJLabel("New Text");
                this.add(label);
                b=newJButton("Change");
                b.addActionListener(newmyHandler());
                add(b);

                b1=newJButton("Disappear");
                b1.addActionListener(newmyHandler());
                add(b1);
                }
    Class myHandler implements ActionListener{

                Public void actionPerformed(ActionEvent e){
                        if(e.getSource()==b)
                        {
                        String s=tf1.getText();
                        label.setText(s);
                        tf1.setText("");
                        }

                        if(e.getSource()==b1)
                                b.setVisible(false);
                }
        }
```

```
Public static void main(String[] args) {
                // TODO Auto-generated method stub
                test f=new test();
                f.setTitle("Copy Text");
                f.setSize(400, 150);
                f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                f.setLocationRelativeTo(null);;
                f.setVisible(true);
        }
}
```

3) Stage v (verify)

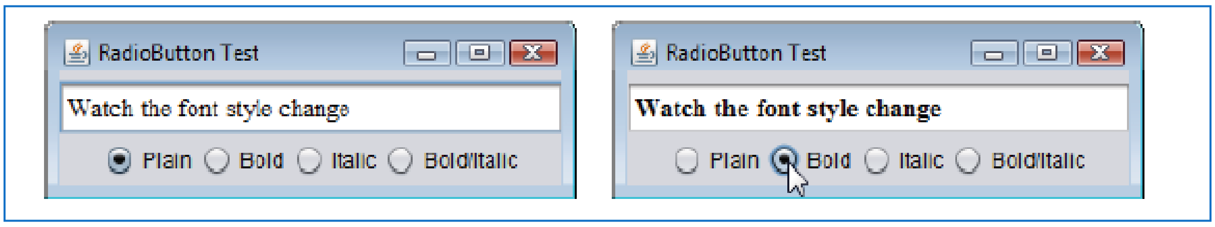
Home Activities:

Activity 1:

Create a frame with one label, one textbox and a button. Display the information entered in textbox on button click.

Activity 2:

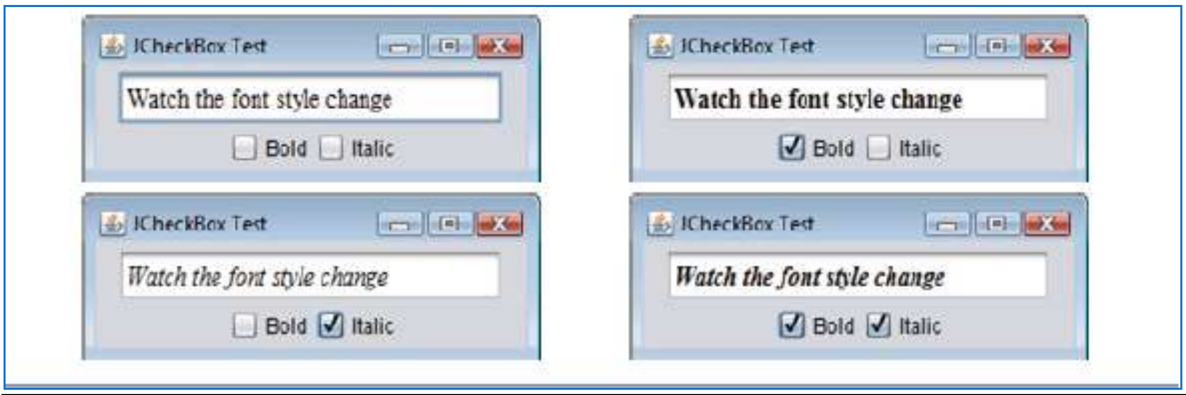
Create frames as follows:



4) Stage a2 (assess)

Assignment 1:

Create Frames as follows:


JCheckBox Test
Watch the font style change
Bold
Italic
JCheckBox Test
Watch the font style change
Bold
Italic
JCheckBox Test
Watch the font style change
Bold
Italic
JCheckBox Test
Watch the font style change
Bold
Italic

Assignment 2:

Make a functional non scientific calculator. Recall the last task of the previous lab .