Lab Manual

**OOP**

**Laboratory 12:**

**Statement Purpose:**

At the end of this lab, the students should be able to:

* Understand Method Overriding
* Understand Abstract Classes
* Understand Polymorphism

**Method Overriding:**

In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to override the method in the superclass. When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden.

**NOTE:** Don't confuse the term overriding with the term overloading, overriding means inheriting a

method from the super class and altering it's functionality, while overloading means having

constructors and methods with the same name but with different signatures.

**Example:**

//Method overriding

**public** **class** MOverride {

**int** i,j;

MOverride(**int** a, **int** b)

{

i=a;

j=b;

}

//display i and j

**void** show()

{

System.*out*.println("i and j: " + i + " " + j);

}

}

**class** B2 **extends** MOverride

{

**int** k;

B2(**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);

}

}

**class** Override

{

**public** **static** **void** main(String args[])

{

B2 subOb = **new** B2(1, 2, 3);

subOb.show();//this calls show() in B

}

}

**Output:**

K : 3

If you want to access the superclass version of an overridden method, you can do so by using super. If the method in B is changed as shown below, the superclass version of show() is called within the subclass version. This allows all instance variables to be displayed.

**void** show(){

**super**.show(); //this calls A's show()

System.*out*.println("K : " + k);

}}

**Output:**

i and j: 1 2

K : 3

**Abstract Class:**

//Using abstract methods and classes.

**abstract** **class** Figure {

**double** dim1;

**double** dim2;

Figure(**double** a, **double** b){

dim1=a;

dim2=b;

}

//area is now an abstract method

**abstract** **double** area();

}

**class** Rectangle **extends** Figure{

Rectangle(**double** a, **double** b)

{

**super**(a, b);

}

//override area for rectangle

**double** area(){

System.*out*.println("Inside Area for Rectangle.");

**return** dim1 \* dim2;

}

}

**class** Triangle **extends** Figure{

Triangle(**double** a, **double** b){

**super**(a, b);

}

//override area for right triangle

**double** area(){

System.*out*.println("Inside Area for Triangle.");

**return** dim1 \* dim2 /2;

}

}

**class** AbstractAreas{

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

//Figure f=new Figure(10, 10); //illegal now

Rectangle r=**new** Rectangle(10, 8);

Triangle t=**new** Triangle(10.0, 8.0);

Figure figref; //this is OK, no object is created

figref=r;

System.*out*.println("Area is " + figref.area());

figref=t;

System.*out*.println("Area is " +figref.area());

}

}

**Output:**

inside Area for Rectangle.

Area is 80.0

Inside Area for Triangle.

Area is 40.0

**Polymorphism:**

Polymorphism is the ability to create a variable, a function, or an object that has more than one form.

Method overriding is a feature which you get when you implement inheritance in your program.

In java language, polymorphism is essentially considered into two versions.

1. Compile time polymorphism (static binding or method overloading)
2. Runtime polymorphism (dynamic binding or method overriding)

A simple example can be from real world e.g. Animal. An application can have Animal class, and its specialized sub classes like Cat and Dog. These subclasses will override the default behavior provided by Animal class + some of its own specific behavior.

**class** Animal {

**public** **void** makeNoise()

{

System.*out*.println("Some sound");

}

}

**class** Dog **extends** Animal{

**public** **void** makeNoise()

{

System.*out*.println("Bark");

}

}

**class** Cat **extends** Animal{

**public** **void** makeNoise()

{

System.*out*.println("Meawoo");

}

}

Now which makeNoise() method will be called, depends on type of actual instance created on runtime e.g.

**public** **class** Lab12 {

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

Animal a1 = **new** Cat();

a1.makeNoise(); //Prints Meowoo

Animal a2 = **new** Dog();

a2.makeNoise(); //Prints Bark

}

}

Java uses a technique for method invocation called "dynamic dispatch". If I have

**class** A {

**public** **void** draw() {System.*out*.println("draw in A class"); }

**public** **void** spin() { System.*out*.println("spin in A class"); }

}

**class** B **extends** A {

**public** **void** draw() {

System.*out*.println("draw in B class");

}

**public** **void** bad() {

System.*out*.println("bad in B class");

}

}

**public** **class** Lab12 {

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

A testObject = **new** B();

testObject.draw(); // calls B's draw, polymorphic

testObject.spin(); // calls A's spin, inherited by B

testObject.bad(); // compiler error, you are manipulating this as an A

}

}

**Output:**



Then we see that B inherits spin from A. However, when we try to manipulate the object as if it were a type A, we still get B's behavior for draw. The draw behavior is polymorphic

**Abstract Method:** Abstract method does not have any body. It is always ends with (;) semicolon. Abstract method must be overridden. It must be in an abstract class. It can never be static and final. Abstract methods are those which need to be implemented in subclass. If class has one abstract method then whole class is declared as abstract. Private method cannot be abstract.

**Example: This example shows how abstract method is used in a class.**

**public** **class** Lab12 {

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

Addition a = **new** Addition (5, 8);

Subtraction s = **new** Subtraction (32, 16);

Multiplication m = **new** Multiplication (4, 2);

System.*out*.println ("Sum of Addition: " + a.sum ()); System.*out*.println ("Sum of Subtraction: " + s.sum ()); System.*out*.println ("Sum of Multiplication: " + m.sum ());

}

}

**abstract** **class** Calculator

{

//define 2 integers

**protected** **int** no1;

**protected** **int** no2;

//declare abstract method

**abstract** **int** sum();

}

**class** Addition **extends** Calculator //extends with Superclass

{

Addition (**int** n1, **int** n2)

{

no1 = n1;

no2 = n2;

}

**int** sum()//define method

{

**return** no1 + no2;//return Addition

}

}

**class** Subtraction **extends** Calculator//extends with Superclass

{

Subtraction (**int** n1, **int** n2)

{

no1 = n1;

no2 = n2;

}

**int** sum()//define method

{

**return** no1 - no2; //return Subtraction

}

}

**class** Multiplication **extends** Calculator //extends with Superclass

{

Multiplication (**int** n1, **int** n2)

{

no1 = n1;

no2 = n2;

}

**int** sum() //define method

{

**return** no1 \* no2; //return Multiplication

}

}

**Output:**



**Example: (Polymorphism)**

Below is the code to draw the happy-face figure using a “polymorphic array” with an abstract base class type.

**import** java.awt.\*;

**public** **class** Lab12\_Poly\_figure **extends** java.applet.Applet {

**public** **void** paint(Graphics g)

{

BaseClass [] refArray = **new** BaseClass[5];

refArray[0] = **new** Head();

refArray[1] = **new** LeftEye();

refArray[2] = **new** RightEye();

refArray[3] = **new** Nose();

refArray[4] = **new** Mouth();

**for** (BaseClass e: refArray)

e.draw(g);

}// end paint

}// end class PolyArray

**abstract** **class** BaseClass

{

**abstract** **void** draw(Graphics g);

}

**class** Head **extends** BaseClass

{

**public** **void** draw(Graphics g)

{

g.drawOval(75, 75, 400, 400);

}

}// end class Head

**class** RightEye **extends** BaseClass

{

**public** **void** draw(Graphics g)

{

g.fillOval(150, 200, 60, 40);

}

}// end class RightEye

**class** LeftEye **extends** BaseClass

{

**public** **void** draw(Graphics g)

{

g.drawOval(300, 200, 60, 40);

}

}// end class LeftEye

**class** Nose **extends** BaseClass

{

**public** **void** draw(Graphics g)

{

g.drawOval(250, 300, 30, 30);

}

}// end class Nose

**class** Mouth **extends** BaseClass

{

**public** **void** draw(Graphics g)

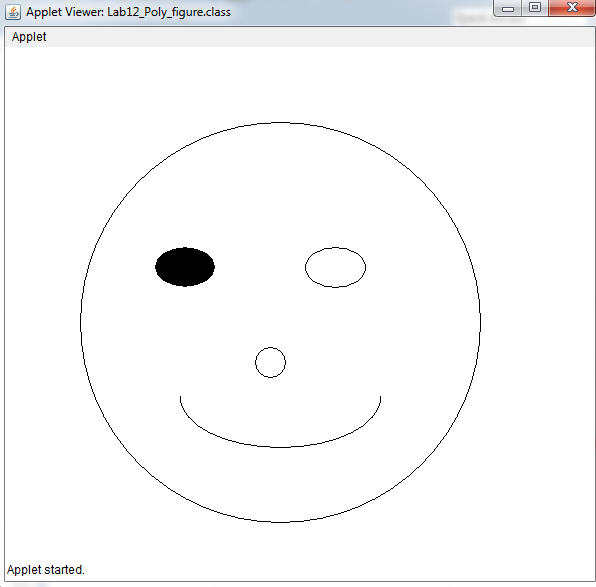
{

g.drawArc(175, 300, 200, 100, 180, 180);

}

}// end class Mouth

**Output:**



**Lab Task: Marks: 10**

**Question 1: Marks: 2**

Make an abstract class named Shape with two abstract methods area and perimeter. Implement a subclass “EquilateralTriangle” having a double variable side denoting the three sides of the equilateral triangle [Note that since all the 3 sides are equal, the constructor will have only one parameter]. The area and perimeter of the equilateral triangle are given as follows:

Area = ¼\*\*(side)2

Perimeter = 3\*side

Provide accessor method for the sides. Test your class using the main class TestShapes.

**Question 2:** **Marks: 8**

Using a polymorphic Array and Abstract base class, draw the house as shown below.

