Module-II (Part-I)

by:
Dr. Soumya Priyadarsini Panda
Sr. Assistant Professor
Dept. of CSE
Silicon Institute of Technology, Bhubaneswar

Contents

- Inheritance
- Use of ‘super’ keyword
- Method Overriding
- Static Vs. Dynamic Polymorphism
- Use of final Keyword
- Abstract class
- Interface

Inheritance

- Inheritance in java is a mechanism in which one class acquires all the properties and behaviours of another class.

- The inherited class is called a ***superclass / Parent class.***

- The class that inherits the properties of parentclass is called a ***subclass / Child class.***

Syntax :

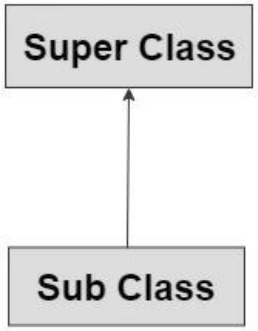
```
class Subclass-name extends Superclass-name
        {
        //methods and fields
        }
```

Example

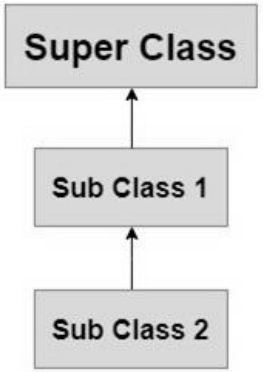
```
class A
        {
         //methods and fields of super class or parent class
        …..
        }


class B extends A
    {
   //methods and fields  of sub class or child class
    ……
    }
```

Types of Inheritance

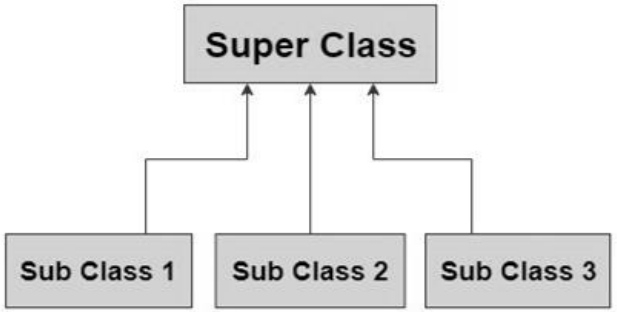
Single Inheritance



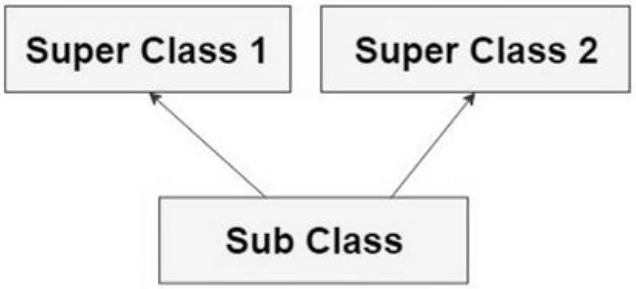
MultiLevel Inheritance



Hierarchial Inheritance



Multiple Inhertance



Multiple Inheritance is not supported by java through class concept

Can be achieved using **'interface'** in java

Types of Inheritance

Single Inheritance:

- In single inheritance there exists single base class and single derived class.

Multilevel Inheritance:

- In Multilevel inheritances there exists single base class, single derived class and multiple intermediate base classes.

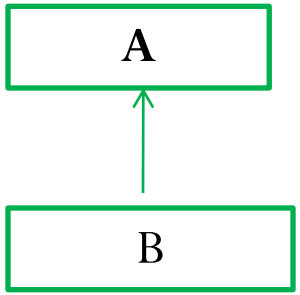
Hierarchical Inheritance:

- When a class has more than one child classes (sub classes) or in other words more than one child classes have the same parent class then this type of inheritance is known as hierarchical inheritance.

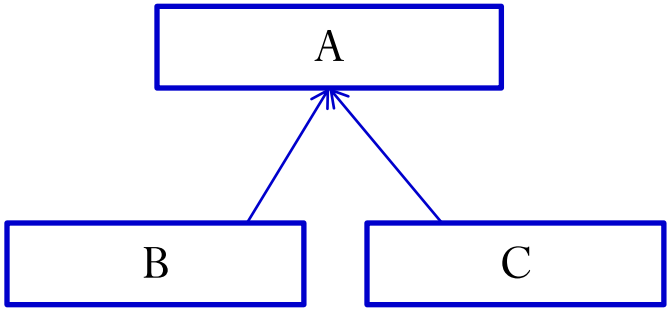
Multiple Inheritance:

- In multiple inheritance there exist multiple classes and single derived class. **The concept of multiple inheritance is not supported in java** through concept of classes but it can be supported through the concept of interface.

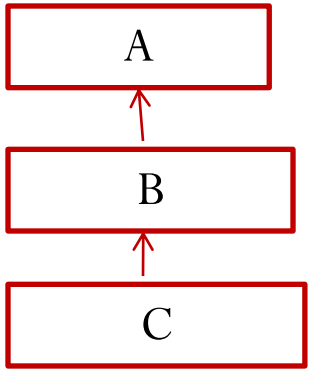
Example: Types of Inheritance



Single Inheritance



HierarchicalInheritance



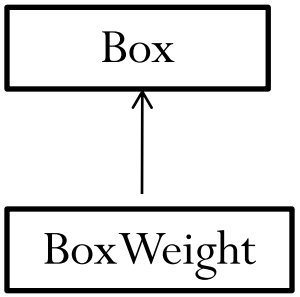
Multi-level Inheritance

```
class A
        {
        ……
        }
class B extends A
        {
        …..
        }
```

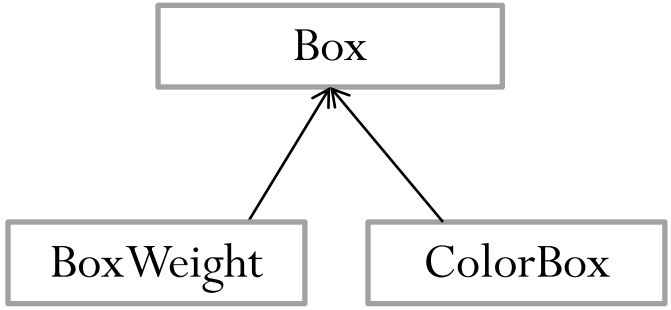
```
class A
        {
        ……
        }
class B extends A
        {
        …..
        }
class C extends A
        {
        …..
        }
```

```
class A
        {
        ……
        }
class B extends A
        {
        …..
        }
class C extends B
        {
        …..
        }
```

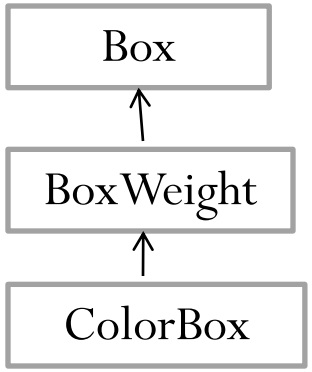
Example



Single Inheritance



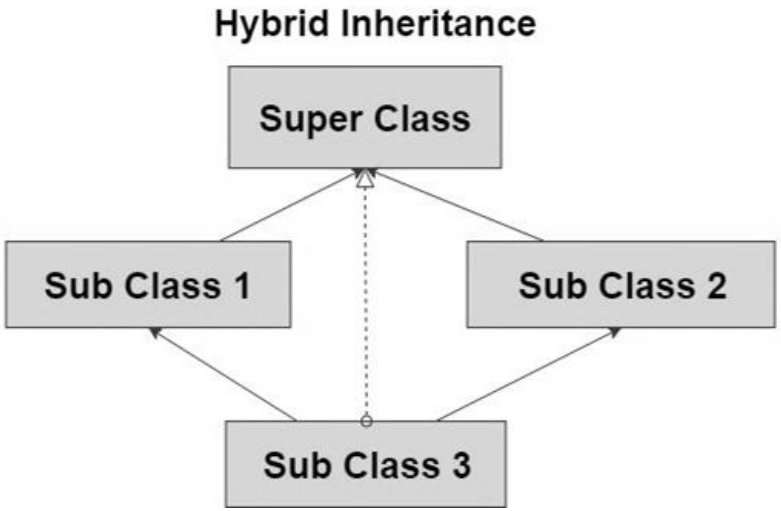
HierarchicalInheritance



Multi-level Inheritance

Hybrid Inheritance

• Combination of any inheritance type discussed so far. In the combination if one of the combination is multiple inheritance then **the inherited combination is not supported by java through the class** concept **but it can be supported through the concept of interface**.



Example

```
class Box {
    double width;
    double height;
    double depth;
    Box(double w, double h, double d) {
            width = w;
            height = h;
            depth = d;
            }
    Box() {
            width = -1;
            height = -1;
            depth = -1;
            }
    Box(double len) {
            width = height = depth = len;
            }
    double volume()
            {
            return width * height * depth;
            }
}  //end of class Box
```

Cont…

```
class BoxWeight extends Box
{
        double weight;

        // constructor for BoxWeight
        BoxWeight(double w, double h, double d, double m)
                {
                width = w;
                height = h;
                depth = d;
                weight = m;
                }
}
```

Cont…

```
class Driver
{
        public static void main(String args[])
                {
                BoxWeight b1 = new BoxWeight(10, 20, 15, 34.3);

                double vol;

                vol = b1.volume();


                System.out.println("Volume of box1 is " + vol);
                System.out.println("Weight of box1 is " + b1.weight);


        }
}
```

Output:

Volume of box1 is 3000.0

Weight of box1 is 34.3

Another Sub class of Box class

```
class ColorBox extends Box
{
        String color;

                // constructor for ColorBox
        ColorBox(double w, double h, double d, String c)
        {
        width = w;
        height = h;
        depth = d;
        color = c;
        }
}
```

Modified Driver class

```
class Driver {
        public static void main(String args[]) {
        BoxWeight b1 = new BoxWeight(10, 20, 15, 34.3);
        ColorBox b2 = new ColorBox(2, 3, 4, "Red");
        double vol;

        vol = b1.volume();
        System.out.println("Volume of box1 is " + vol);
        System.out.println("Weight of box1 is " + b1.weight);


        vol = b2.volume();
        System.out.println("Volume of box2 is " + vol);
        System.out.println("Color of box2 is " + b2.color);
        }
        }
```

Output:

Volume of box1 is 3000.0
Weight of box1 is 34.3
Volume of box2 is 24.0
Color of box2 is **Red**

Example: Assigning Boxweight refernce to Box reference

```
class Driver {
        public static void main(String args[])
                {
                BoxWeight b1 = new BoxWeight(10, 20, 15, 34.3);
                Box b2=new box();
                double vol;

                vol = b1.volume();
                System.out.println("Volume of box1 is " + vol);
                System.out.println("Weight of box1 is " + b1.weight);

                b2=b1;  /
                vol = b2.volume();
                System.out.println("Volume of box2 is " + vol);
        }
}
```

- When a reference to a sub class object is assigned to a super class reference variable, only those part can be accessed defined by the super class.

```
class Driver
{
        public static void main(String args[])
        {
        BoxWeight b1 = new BoxWeight(10, 20, 15, 34.3);
        Box b2=new box();
        double vol;

        vol = b1.volume();
        System.out.println("Volume of box1 is " + vol);
        System.out.println("Weight of box1 is " + b1.weight);

        b2=b1;  /assigns Boxweight refernce to Box reference
        vol = b2.volume();
        System.out.println("Volume of box2 is " + vol);
        System.out.println("Weight of box2 is " + b2.weight);
        //Invalid statement as weight is not defined in Box class
        }
}
```

Example

```
class A {
        int i;
        private int j;
        void setValue(int x, int y){
                i=x;
                j=y;
        }
   }
class B extends A{
        int total;
        void sum(){
                total=i+j;
                }
}
```

- ***A sub class can use all members of its super class except the members declared as private***

```
class Driver {
        public static void main(String args[])
        {
        B ob1=new B();
        ob1.setValue(10,20);
        ob1.sum();
   System.out.Println("Total is "+ ob1.total);
   }
```

Output: **Error**

Use of ‘super’ Keyword

- Whenever a sub class need to refer to its **immediate super class**, it can use the ‘super’ keyword.

- ‘super’ has 2 general use:
  - To call super class constructor
  - To access members of super class.

Using ‘super’ Keyword to call super class constructor

- A super class constructor can be called using the ‘super’ keyword from the sub class.

Example:

```
class BoxWeight extends Box
{
        double weight;
        BoxWeight(double w, double h, double d, double m) {
                super(w, h, d);
                weight = m;
                }
        BoxWeight() {
                super();
                weight = -1;
                }
BoxWeight(double len, double m) {
                super(len);
                weight = m;
                }}
```

Cont…

- When a subclass calls super( ), it is calling the constructor of its immediate super class.
  - Thus, super( ) always refers to the super class immediately above the calling class.
- Also, super( )must always be the first statement executed inside a subclass constructor.

Using ‘super’ Keyword to access super class members

- When members of super class are hidden from sub class

Syntax:

```
super.member
//member can be method name or instance variable.
```

Example:

```
super.width;

super.volume();
```

Example

```
class A{
        int i;
        }
class B extends A{
        int i;
        B(int a,int b) {
                super.i=a;          //i in class A
                i=b;                //i in class B
        void show() {
                System.out.println("i in super class: " +super.i);
                System.out.println("i in sub class:  " + i);
                }
        }
class Driver{
        public static void main(String args[ ]) {
        B ob1=new B(1,2);
        ob1.show();
        }
        }
```

Output:

i in super class: 1

i in sub class: 2

Example

```
class Box
{
private double width;
private double height;
private double depth;

        // construct clone of an object
        Box(Box ob) // pass object to constructor
        {
        width = ob.width;
        height = ob.height;
        depth = ob.depth;
        }
// constructor used when all dimensions specified
        Box(double w, double h, double d)
        {
        width = w;
        height = h;
        depth = d;
        }
```

Cont…

```
// constructor used when no dimensions specified
Box()
{
width = -1; // use -1 to indicate
height = -1; // an uninitialized
depth = -1; // box
}
// constructor used when cube is created
Box(double len)
        {
        width = height = depth = len;
        }
// compute and return volume
double volume()
        {
        return width * height * depth;
        }
}
```

Cont…

```
// BoxWeight now fully implements all constructors.
class BoxWeight extends Box
        {
        double weight; // weight of box
        // construct clone of an object
        BoxWeight(BoxWeight ob) // pass object to constructor
                {
                super(ob);
                weight = ob.weight;
                }
                // constructor when all parameters are specified
        BoxWeight(double w, double h, double d, double m)
                {
                super(w, h, d); // call superclass constructor
                weight = m;
                }
```

Cont…

```
// default constructor
BoxWeight()
        {
        super();
        weight = -1;
        }

        // constructor used when cube is created
        BoxWeight(double len, double m)
                {
                super(len);
                weight = m;
                }
        }
```

Cont…

```
class DemoSuper
        {
        public static void main(String args[])
                {
        BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);

        BoxWeight mybox2 = new BoxWeight(); // default

        BoxWeight mycube = new BoxWeight(3, 2);

        BoxWeight myclone = new BoxWeight(mybox1);
        double vol;

        vol = mybox1.volume();
        System.out.println("Volume of mybox1 is " + vol);
        System.out.println("Weight of mybox1 is " + mybox1.weight);
        System.out.println();
```

Cont…

```
vol = mybox2.volume();
System.out.println("Volume of mybox2 is " + vol);
System.out.println("Weight of mybox2 is " + mybox2.weight);
System.out.println();

vol = myclone.volume();
System.out.println("Volume of myclone is " + vol);
System.out.println("Weight of myclone is " + myclone.weight);
System.out.println();

vol = mycube.volume();
System.out.println("Volume of mycube is " + vol);
System.out.println("Weight of mycube is " + mycube.weight);
System.out.println();
}
}
```

Output:

```
Volume of mybox1 is 3000.0
Weight of mybox1 is 34.3
Volume of mybox2 is -1.0
Weight of mybox2 is -1.0
Volume of myclone is 3000.0
Weight of myclone is 34.3
Volume of mycube is 27.0
Weight of mycube is 2.0
```

When constructors are executed

```
class A {
        A() {
        System.out.println("Inside A's constructor.");
                } }
class B extends A {
        B() {
                System.out.println("Inside B's constructor.");
                } }
class C extends B {
        C() {
        System.out.println("Inside C's constructor.");
        } }
class CallingCons {
        public static void main(String args[]) {
        C c = new C();
        }  }
```

Output:

Inside A's constructor
Inside B's constructor
Inside C's constructor

Method Overriding

- In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its super class, then the method in the subclass is said to override the method in the super class.

- When an overridden method is called from within its subclass, it will always refer to the version of that method defined by the subclass.

- The version of the method defined by the super class will be hidden.

Example

```
class A {
        int i, j;
        A(int a, int b) {
                i = a;
                j = b;
                }
        void show() {
                System.out.println("i and j: " + i + " " + j);
        }
   }
```

Cont…

```
class B extends A {
        int k;
        B(int a, int b, int c)
                {
                super(a, b);
                k = c;
                }
        void show()
                {
                System.out.println(“Hello”)
                }
        }
```

Cont…

```
class Override
    {
            public static void main(String args[])
            {
            B b1 = new B(1, 2, 3);
            b1.show(); // this calls show() in B
            }
    }
```

Example

```
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() {
                super.show();
                System.out.println("k: " + k);
    }
    }
```

Example

```
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() {
                super.show();   //calls super class show method
    System.out.println("k: " + k);
    }
    }
```

Output:
i and j: 1 2
k: 3

Method Overriding Cont…

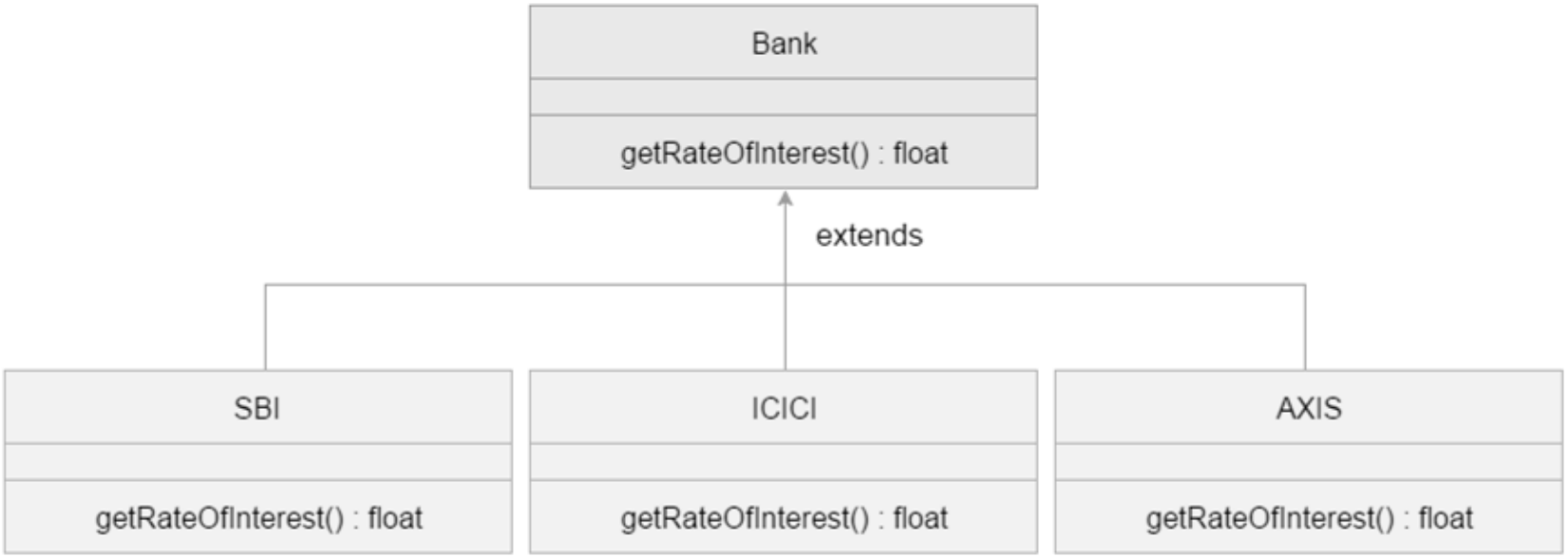
- Method overriding occurs only when the names and the type signatures of the two methods are identical. If they are not, then the two methods are simply overloaded.

Usage of Java Method Overriding

- Method overriding is used to provide specific implementation of a method that is already provided by its super class.

- Method overriding is used for runtime polymorphism

Example


Bank
getRateOfInterest() : float
extends
SBI
getRateOfInterest() : float
ICICI
getRateOfInterest() : float
AXIS
getRateOfInterest() : float

```
class Bank
	{
	int getRateOfInterest(){return 0;}
	}
class SBI extends Bank
	{
	int getRateOfInterest(){return 8;}
	}
class ICICI extends Bank
	{
	int getRateOfInterest(){return 7;}
	}
class AXIS extends Bank
	{
	int getRateOfInterest(){return 9;}
	}
class Test{
	public static void main(String args[]) {
			SBI s=new SBI();
			ICICI i=new ICICI();
			AXIS a=new AXIS();
	System.out.println("SBI Rate of Interest: "+s.getRateOfInterest());
	System.out.println("ICICI Rate of Interest: "+i.getRateOfInterest());
	System.out.println("AXIS Rate of Interest: "+a.getRateOfInterest());
	}
}
```

Output:

```
SBI Rate of Interest: 8
ICICI Rate of Interest: 7
AXIS Rate of Interest: 9
```

Polymorphism

- There are two types of polymorphism in java:
  - **Compile time polymorphism**
  - **Runtime polymorphism**

Compile time polymorphism:

- The type of polymorphism that is implemented when the compiler compiles a program is called compile-time polymorphism.
- i.e. the method is to be invoked is decided at compile time.
- This type of polymorphism is also called as **static polymorphism** or **early binding**.
- Java supports compiletime polymorphism **through method overloading.**

Runtime polymorphism:

- The type of polymorphism that is implemented dynamically when a program being executed is called run-time polymorphism.
- i.e. the method which has to be invoked is decided during the run time.
- The run-time polymorphism is also called **dynamic polymorphism** or **late binding**.
- Run-time polymorphism is achieved through **method overriding** by the dynamically method dispatch.

Dynamic Method Dispatch:

- Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time.
- Through Dynamic method dispatch Java implements run-time polymorphism
- A super class reference variable can refer to a subclass object.
  - Java uses this fact to resolve calls to overridden methods at runtime.

Example:

```
class A {
    void show() {
            System.out.println("Inside A's method");
            }
        }

class B extends A {
    void show() {
            System.out.println("Inside B's method");
            }
    }
```

```
class Dispatch {
    public static void main(String args[]) {
            A   a = new  A();
            B   b = new  B();



            A   r;      //creating reference variable of super class


            r = a;
            r. show();                      // calls A's version of show()

            r = b;
            r. show();                       // calls B's version of show()

            }
}
```

Output:

Inside A's show method

Inside B's show method

Difference between method overloading and method overriding

Method Overloading	Method Overriding
Method overloading is used to increase the readability of the program.	Method overriding is used to provide the specific implementation of the method that is already provided by its super class.
Method overloading is performed within class.	Method overriding occurs in two classes that have IS-A (inheritance) relationship.
In case of method overloading, parameter must be different.	In case of method overriding, parameter must be same.
Method overloading is the example of compile time polymorphism.	Method overriding is the example of run time polymorphism.
In java, method overloading can't be performed by changing return type of the method only. Return type can be same or different in method overloading. But you must have to change the parameter.	Return type must be same in method overriding.

Example

```
class Figure
{
    double dim1, dim2;
    Figure(double a, double b)
    {
            dim1 = a;
            dim2 = b;
    }
    double area()
    {
    System.out.println("Area for Figure is undefined.");
    return 0;
    }
}
```

Cont…

```
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;
    }
}
```

Cont…

```
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;
    }
}
```

Cont…

```
class FindAreas
{
    public static void main(String args[])
    {
            Figure f = new Figure(10, 10);
            Rectangle r = new Rectangle(9, 5);
            Triangle t = new Triangle(10, 8);

            Figure figref;
            figref = r;
            System.out.println("Area is " + figref.area());
            figref = t;
            System.out.println("Area is " + figref.area());
            figref = f;
            System.out.println("Area is " + figref.area());
    }
}
```

Output:

Inside Area for Rectangle.
Area is 45.0
Inside Area for Triangle.
Area is 40.0
Area for Figure is undefined.
Area is 0.0

final keyword in Java

- The ***final*** keyword is used in different contexts:
  - **final variables**
  - **final methods**
  - **final classes**
- final is a **non-access modifier**

final variables:

- Used to create constants
  - If a variable is declared as final, the value of the variable can’t be changed.

- final variables must be used only for the values that remain constant throughout the execution of program.

Example:

```
final int THRESHOLD = 5;
```

final methods:

- To prevent method overriding
  - A final method cannot be overridden.

Example:

```
final void show()
        {
        System.out.println(“Final method");
        }
```

Example

```
class Bike{
final void run(){
System.out.println("running");
}
}
 class Honda extends Bike {
    void run()          // error since final methods can not be overridden
    {
    System.out.println("running safely with 100kmph");
    }
 public static void main(String args[]) {
                Honda h1= new Honda();
                h1.run();
    }
}
```

final class:

- To Prevent Inheritance
  - When a class is declared as final then it cannot be sub classed
  - i.e. No other class can extend it.

Example:

```
final class A
        {
                // methods and fields
        }
```

Example

```
final class A
{
    // methods and fields
}

class B extends A
{
    // ERROR! Can't subclass A
}
```

Example

```
class XYZ{
        protected void func(){
              System.out.println("Hello java");
        }
   }
class Simple extends XYZ {
       void func()
            {
            System.out.println("Hello java");
            }
}
   class Test {
        public static void main(String args[])
        {
             Simple obj=new Simple();
             obj.func();
        }
   }
```

Output: error

Example

```
class A{
        void func(){
           System.out.println("base");
        }
    }
    class BTest extends A{
        void func(){
           System.out.println("derived");
        }
    }
class Test{
        public static void main(String ss[])
        {
           BTest b1;
           b1=new A();
           b1.func();
        }
}
```

Output: error

Data Abstraction

- Abstraction is a process of hiding the implementation details and showing only functionality to the user
- It shows only important things to the user and hides the internal details.
- There are two ways to achieve abstraction in java
  - **Abstract class**
  - **Interface**

Abstract class

- A class that is declared as abstract is known as **abstract class**

- **Example:**

```
abstract class A
        {
        …….
        }
```

Abstract class Cont…

- An abstract class cannot be directly instantiated with the new operator.
- We cannot declare abstract constructors, or abstract static methods.
- Any subclass of an abstract class must either implement all of the abstract methods in the super class, or be itself declared abstract.

Abstract Method

- An abstract method is a method with only signature (i.e., the method name, the list of arguments and the return type) without implementation (i.e., the method's body).

Example:

```
abstract void printStatus();
    //no body
```

- An abstract class provides a template for further development

Example

```
abstract class A {
        abstract void show();
        void showconcrete() {
                System.out.println("This is a concrete method");
                }
        }
class B extends A {
                void show() {
                System.out.println("B's implementation of show");
                    }
   }
```

Cont…

```
class Demo  {
        public static void main(String args[]) {
                B   b = new  B();
                b. show();
                b. showconcrete ();
                }
        }
```

Output:

B's implementation of show

This is a concrete method

Example

- Define an abstract class named “Figure”, having data members dim1 and dim2. Extend this class to create two concrete classes named Rectangle and Triangle. Override the getArea() method in the sub classes. Invoke the getArea() method in the main method of another Driver class through the abstract class reference variable.

Example

```
abstract class Figure
    {
    double dim1, dim2;
    Figure (double a, double b)
                        {
                dim1 = a;
                dim2 = b;
                        }
     abstract double getArea();
     // getArea is now an abstract method
}
```

Cont…

```
class Rectangle extends Figure
        {
        Rectangle(double a, double b)
                {
                super(a, b);
                }
        // override area for rectangle
        double getArea()
                {
                System.out.println("Inside Area for Rectangle.");
                return dim1 * dim2;
                }
        }
```

Cont…

```
class Triangle extends Figure   {
    Triangle(double a, double b)
            {
            super(a, b);
            }

    // override getArea for right triangle
    double getArea()
            {
            System.out.println("Inside Area for Triangle.");
            return dim1 * dim2 / 2;
            }
}
```

Cont…

```
class Driver {
    public static void main(String args[])  {
            Rectangle r = new Rectangle(9, 5);
            Triangle t = new Triangle(10, 8);

            Figure figref;
            figref = r;
            System.out.println("Area is " + figref.getArea());
            figref = t;
            System.out.println("Area is " + figref.getArea());
    }
}
```

Output:

```
Inside Area for Rectangle.
Area is 45.0

Inside Area for Triangle.
Area is 40.0
```

Interfaces

- A Java interface is a 100% abstract superclass which define a set of methods its subclasses must support.
- An interface contains only public abstract methods (methods with signature and no implementation) and possibly constants (public static final variables).
- It is used to achieve abstraction and multiple inheritances in Java.

Interfaces Cont…

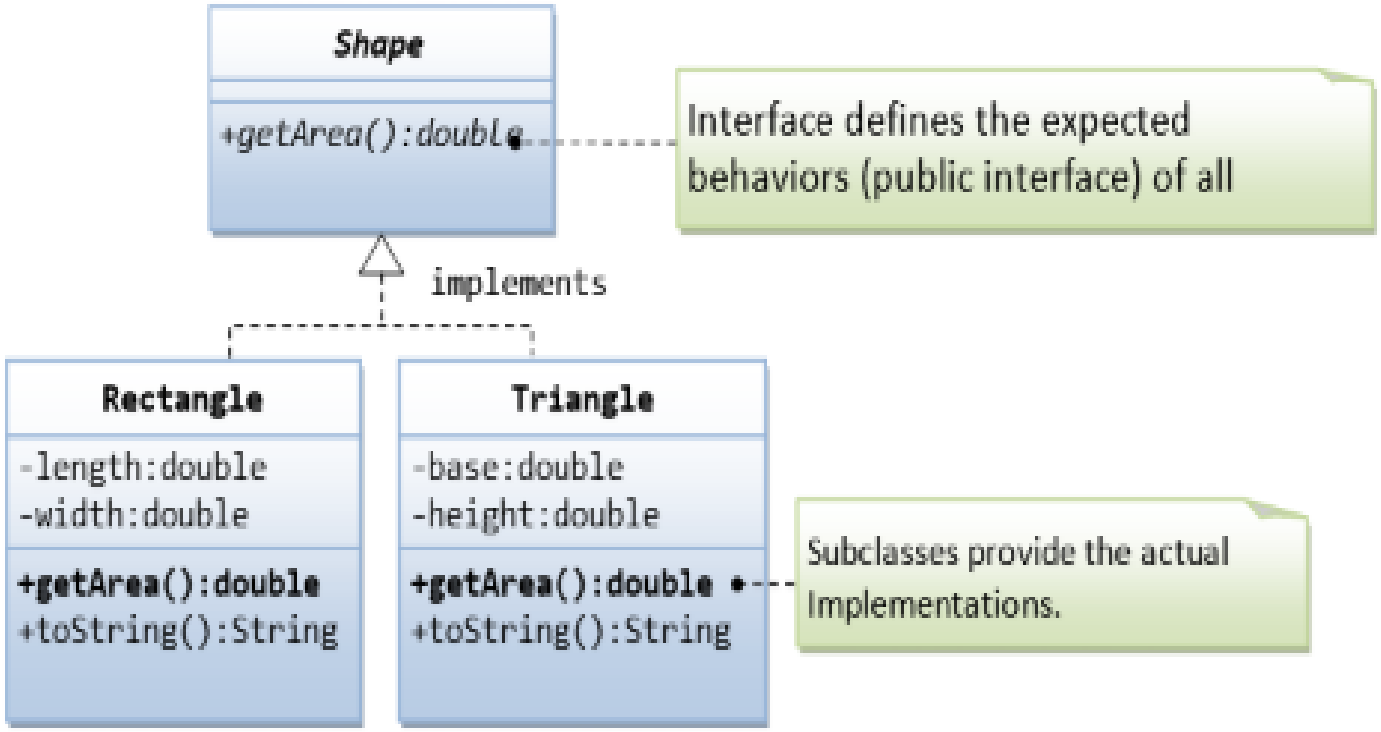
- It is used to achieve total abstraction.
- Since java does not support multiple inheritance in case of class, but by using interface it can achieve multiple inheritance.
  - A subclass, however, can implement more than one interfaces.
  - ***The methods that implement an interface must be declared public.***

Example

```
interface Callback
        {
                void callback(int par);
        }
class Client implements Callback
        {
        public void callback(int p)
                {
                System.out.println("callback called with " + p);
                }
        }
```

NOTE: When you implement an interface method, it must be declared as ***public.***

Example


Shape
+getArea():double
Interface defines the expected behaviors (public interface) of all
implements
Rectangle
-length:double
-width:double
+getArea():double
+toString():String
Triangle
-base:double
-height:double
+getArea():double
+toString():String
Subclasses provide the actual Implementations.

Example

```
public interface Shape
        {
        double getArea();
        }
public class Rectangle implements Shape
        {
                .......
                public double getArea() {
                        return length * width;
                        }
        }
public class Triangle implements Shape {
        …..
        public double getArea() {
                        return 0.5 * base * height;
        }
   }
```

```
public class TestShape {
public static void main(String[] args)
{
    Shape s1 = new Rectangle(1, 2);
    System.out.println(s1);
    System.out.println("Area is " + s1.getArea());
    Shape s2 = new Triangle(3, 4);
    System.out.println(s2);
    System.out.println("Area is " + s2.getArea());
        }
}
```

Difference between Abstract class & Interface

- The abstract keyword is used to declare abstract class.
- An abstract class can be extended using the keyword "extends".
- Abstract class can have abstract and non-abstract methods.
- Abstract class doesn't support multiple inheritance.
- A Java abstract class can have class members like private, protected, etc.

Example:

```
public abstract class Shape
                    {
    public abstract void draw();
                    }
```

- The interface keyword is used to declare interface.
- An interface can be implemented using the keyword "implements".
- Interface can have only abstract methods.
- Interface supports multiple inheritance.
- Members of a Java interface are public by default.

Example:

```
public interface Drawable
                    {
    void draw();
                    }
```

Home work Questions

Q1. Write a program to create a class named Shape. It should contain two methods, draw() and erase() that prints “Drawing Shape” and “Erasing Shape” respectively. For this class, create three sub classes, Circle, Triangle and Square and each class should override the parent class functions - draw () and erase (). The draw() method should print “Drawing Circle”, “Drawing Triangle” and “Drawing Square” respectively. The erase() method should print “Erasing Circle”, “Erasing Triangle” and “Erasing Square” respectively. Create objects of Circle, Triangle and Square, assign each to Shape variable(reference) and call draw() and erase() method using each object.

Q2. Define a class Employee having basic data members empName, empID and empSal, with necessary member functions and constructors. Define a class Manager which is inherited from Employee class and having a data member bonus. Define the driver class that create object of the class Manager and access Manager details.

Q3. Define an interface **Calculator** which has the basic methods **add(), sub(), mul() and div().** Define a concrete class named as **DemoCalculator** that implements the interface. Define the driver class, which create object reference of the interface Calculator and perform all basic operation of the calculator.

Example: Figure class using Inheritance

```
class Figure {
        double dim1, dim2;
        Figure (double a, double b)
                {
                dim1 = a;
                dim2 = b;
                }
        double getArea()
                {
                System.out.println("Area  is: ");
                return dim1 * dim2;
                }
        }
class Rectangle extends Figure {
                Rectangle(double a, double b)
                        {
                        super(a, b);
                        }
                void show()
                        {
                        System.out.println("Rectangle  Area:");
                        }
                }
```

```
class Driver {
        public static void main(String  args[])
                {
                Rectangle r = new Rectangle(9, 5);
                r.show();
                double a=r.getArea();
                System.out.println("Rect  Area.="+a);
        }
        }
```

Example: Figure class using abstract class

```
abstract class Figure
        {
        double dim1, dim2;
        Figure (double a, double b)
                {
                dim1 = a;
                dim2 = b;
                }
        abstract double getArea();
}

class Rectangle extends Figure
                {
                Rectangle(double a, double b)
                        {
                        super(a, b);
                        }
                double getArea()
                        {
                        System.out.println("Area  of Rectangle:");
                        return dim1 * dim2;
                        }

                }
```

```
class Driver
        {
        public static void main(String  args[])
                {
                Rectangle r = new Rectangle(9, 5);
                double a=r.getArea();
                System.out.println("Rect  Area.="+a);
        }
}
```

Example: Figure class using interface

```
interface Figure
        {
        double dim1=9;
        double dim2=5;
        double getArea();
        }
class Rectangle implements  Figure
        {
        public double getArea()
                {
                System.out.println("Area  of Rectangle:");
                return dim1  *  dim2;
                }
        }
class Driver
        {
        public static void main(String  args[])
                {
                Rectangle r = new Rectangle();
                double a=r.getArea();
                System.out.println("Rect  Area.="+a);
                }
        }
```

Multiple Inheritance using interface to be cont…