UNIT FOUR

INHERITANCE

Basics of Inheritance

- **Inheritance** is a mechanism that enables one class to inherit both the behavior and the attributes of another class

✓ The old class is known as "**base**" class, "**super**" class or "**parent**" class"; and the new class is known as "**sub**" class, "**derived**" class, or "**child**" class.

- Inheritance is a fundamental feature of **object oriented programming**.

✓ It enables the programmer to write a **class based on an already existing class.**

General Structure to define a Sub class

```
class SubClassName extends SuperClassName {
    data-type1 varName1;
   data-type2 varName2;
    data-typeN varNameN;
//Define Subclass Methods
        return-type methodName1(parameter list) {
                        //Body of methodName1
}
return-type methodName2(parameter list){
                        //Body of  methodName2
}
return-type methodNameN(parameter list){
}//End of  methodNameN()
}//End of Subclass
```

General Structure to define a Sub class--------

- The **keyword** **extends** is used to inherit a class from another class. Allows to extend from only one class.
- It **"extends"** signifies that the **properties of super class** are extended to the **subclass**.
- That means, **subclass contains its own members** as well as those of the **super class**.
- Java **does not support** **'multiple inheritance'**, that is you can only specify one **superclass** for any **subclass** that you create.
- **For example**, the following is **not possible in Java**.

```
class  Class1 extends Class2, Class3{
        //body of the subclass
        }
```

General Structure to define a Sub class--------

- You can create a **hierarchy of inheritance** in which a **subclass becomes a super class of another subclass**.
- However, **no class can be a super class of itself.**
- **For Example: Consider the following sub-class definition to demonstrate a hierarchy of inheritance.**

```
class Mammal extends Animal{
   //body of Mammal Class
   }
class Lion extends Mammal {
   //body of Lion Class
   }
```

General Structure to define a Sub class--------

- In the above example **Animal is a more general class** that contains **properties common** to all animals including **Mammals, Reptiles** and others.
- Since **Mammals are animals**, all **properties of Animals** are also **properties of Mammals**.

Activity 1

- Write simple Java program to demonstrate inheritance based on the following information:
  - Define a super class named A with instance variable of i and j and instance method named m1(), this method output the values of i and j when it is called through super and subclass objects.
  - Define a subclass named B that extends a superclass A with its own instance variable named k and method named m2() and sum() where m1() method output the values of k and sum() calculate the sum of i, j and k respectively when it is called through objects of a subclass.

Activity 1-------

- Define another class named ImplementSuClass to create objects of a superclass and subclass named superob and subob respectively.
- Assign separate values to superob's and subob's instance variable respectively.
- Call m1() through objects of super and subclass respectively and call m2() and sum() through objects of a subclass.

//Define super class named A

class A {

//Declare instance Variables of A

int i, j;

Activity 1-------

//Define a method named m1() and output the values of i and j

void m1() {

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

}//End of m1()

}//End of a super class A

// Define a subclass named B that extends a superclass A

class B extends A {

//Declare instance variable of sub class B

int k;

//Define a subclass method named m2()

void m2() {

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

Activity 1-------

}//End of M2()

//Define another sub-class method named sum()

void Sum() {

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

}//End of Sum()

}//End of sub-class B

//Define another class to create objects of A and B

class ImplementSuClass {

public static void main(String args[]) {

//Declare and create objects of A and B respectively

A superOb = new A();

B subOb = new B();

Activity 1-------

//Assign values to superob’s instance variable

superOb.i=10;

suprOb.j=20;

//Call a superclass method M1()

superOb.m1();

//Assign values to subOb’s instance variables

subOb.i = 7;

subOb.j = 8;

subOb.k = 9;

//Call a superclass method M1() throygh suOb object

subOb.m1();

Activity 1-------

//Call a sub-class Method m2()

subOb.Mm();

//Call a subclass method sum

subOb.sum();

}//End of main ()

}//End of Class

Member Access and Inheritance

- Although a **subclass includes all of the members of its superclass,** it **cannot access those members of the superclass that have been declared as private**.
- **Visibility modifiers** determine which **class members** are accessible and which do not.
- M**embers (variables and methods)** declared with **public** visibility are **accessible**.
- But those with **private visibility are not (**accessed only within the class definition in which it appears).

Member Access and Inheritance

- **How to make class/instance variables visible only to its subclasses?**
  - Java provides a **third visibility modifier- protected**
  - **Members (variables and methods) defined as protected are only visible to the sub-class.**
- The **protected visibility** modifier allows a **member** of a **base class to be accessed in the child**.
  - protected visibility provides more **encapsulation** than public does
  - protected visibility is not as tightly **encapsulated** as **private** visibility.

Activity 2

- **Example**: Write Java program to demonstrate private, protected and public access modifiers. Private members remain private to their class. This program contains an error and will not compile:

```
//Define a superclass by the name A
public class A {
//Declaration of instance variable of a class
int i; // public by default
// private to A
private int j;
//Allows to be accessed in a sub class
protected int m, n;
```

Activity 2-------

```
//Define parameterized Method of A
void M1(int x, int y) {
i=x;
j=y;
System.out.println("i in Super="+i);
System.out.println("j in Super="+j);
}//End of M1()
//Define protected instance method of a superclass M2()
protected void M2(int m, int n){
    this.m=m;
    this.n=n;
```

Activity 2------

System.out.println("m in Super="+m);

System.out.println("n in Super="+n);

}//End of M2()

}//End of A

//Define a subclass by the name B

class B extends A{

//Define Instance method of B sum()

int Sum(int a, int b) {

i=a;

 j=b;//**ERROR, j is not accessible here**

 //Declaration of instance Variable of class B

Activity 2------

```
int total;
total= (i+j); // ERROR, j is not accessible here
return (total);
}//End of Sum()
}//End of  class B
//Define another class to create objects of A and B
class AccessModifiers {
public static void main(String args[]) {
//Declare and create objects of A and B
A a1=new A();
B b1 = new B();
int sum;
```

Activity 2------

//call Super class method M1 through a1

a1.M1(40, 80);

//Call Super-class method M1() through b1

b1.M1(10, 12);

//Call a sub-class Name

sum= b1.Sum(30,40);

System.out.println("Total is " +sum);

//Call Super-class method M2() through b1

b1.M2(400, 600);

}//End of main ()

}//End of class

Activity 2------

- The above program will **not compile because the reference to j inside the sum( ) method of B causes an access violation**.
- Since **j is declared as private in a super class A, it is only accessible by other members of its own class**.
- Therefore Subclasses such as B have no access to it.

➢ **Note:**

➢ A **class member that has been declared as private** will remain **private to its class**.

- It is **not accessible by any code outside its class**, including **subclasses.**

➢ A **class member that has been declared as protected** will **not be accessed by only a super class** but also by a **sub class**.

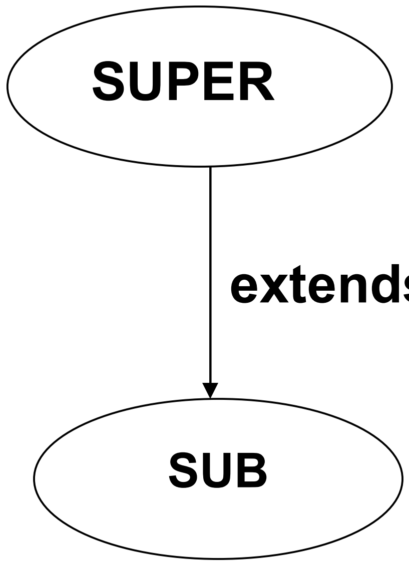
Exercise

1. Refer the previous java program in Activity 2 used to demonstrate private and public members. The private members j in A makes the program does not compile and produce error. To correct the program rewrite and use protected keyword for j and make the program error free.
2. Write Java Program to implement public and protected access modifier .(Use Area of Rectangle and Area of Triangle). The program is expected to define sub-class and super class and parameterized method and returns a value to the caller when they are invoked separately.

Types of inheritance

- Acquiring to the properties of an existing Object into newly creating Object to overcome the re-declaration of properties in deferent classes.
- These are **3 types:**

1. Single Inheritance



Single Inheritance-----

- **Single inheritance** enables a child class to inherit properties and behavior from a **single parent class**.
  - It allows a child class to **inherit** the properties and behavior of a base class, thus enabling code re-usability as well as adding new features to the existing code.
- **Example**: Write Java program to demonstrate single inheritance where **one sub-classes can extend only one super-class.**

```
public class Simple {
//Declaration of instance variable of class A
   int m, n;
 //Define instance method of class A
```

Single Inheritance-----

```
void dispA(){
    System.out.println("m="+m);
     System.out.println("n="+n);
  }//End of dispA()
}//End of class Simple
```

//Define a sub class by the name Sam by extending Simple

```
class Sam extends Simple{
```

//Define instance variable of a sub class

```
    int a,b, k;
```

//Define instance method of a sub class

```
    void dispB(){
```

Single Inheritance-----

```
 System.out.println("k="+k);
      System.out.println("a="+a);
       System.out.println("b="+b);
    }//End of dispB()
```

//Define another instance method of a sub class by the name Sum()

```
void Sum(){
System.out.println("m+n+k="+(m+n+k));
System.out.println("a+b+k="+(a+b+k));
}//End of Sum()
}//End of Sub class()
```

Single Inheritance-----

```
class TestSimple{
public static void main(String[] args) {
//Declare and create objects of class B
Sam b1=new Sam();
//Assign each instance variables of b1 accessed by b1
b1.m=20;
b1.n=30;
b1.a=40;
b1.b=50;
b1.k=60;
```

Single Inheritance-----

```
//CalldispA(), dispB() and Sum() method by a subclass
b1.dispA();
b1.dispB();
b1.Sum();
}//End of main ()
}//End of class
```

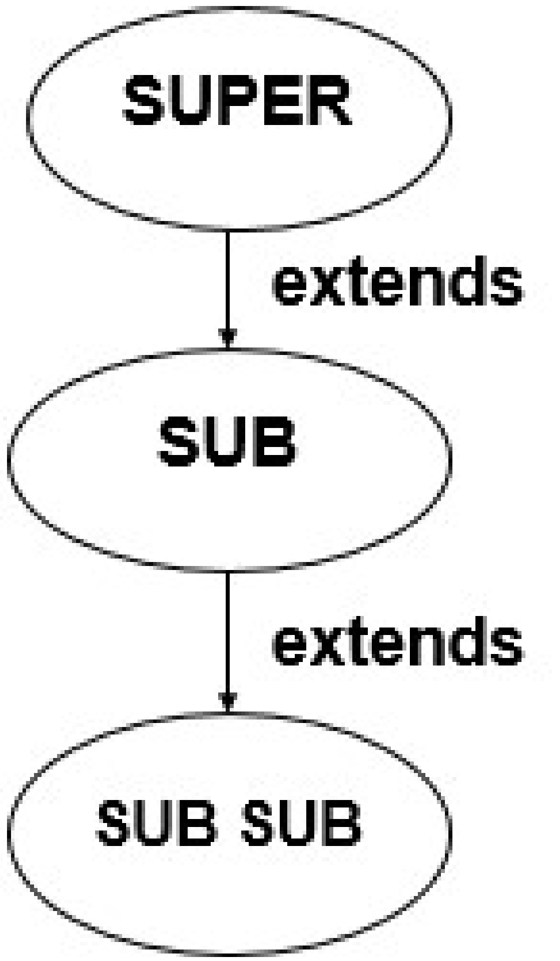
- **Exercise:**
  - Refer the previous Java program and rewrite the program based on the following information.
  - Use all of the instance variable of the super and subclass of the previous java program

Single Inheritance-----

- The superclass dispA() method is changed to parameterized method and return the product of the instance variable of a superclass
- The subclass dispB() method is defined as a parameterized method and return the average of the instance variables of the superclass and subclass.
- The subclass Sum() method is defined as a parameterized method and return the sum of all instance variables of a superclass and subclass members.

2. Multilevel Inheritance

- In multilevel inheritance a **derived class** will be inheriting a **parent class** and as well as the derived class act as the parent class to other child class.



Activity 3

- Write java program to demonstrate multilevel inheritance based on the following information:
  - Use A, B and C as a class name where A is a parent class for B and B is a parent class for C and transitively A is parent class to C.
  - Define instance members named i, j under super class A and **dispA()** respectively
  - Define instance members named k and d**ispB()** under B sub class respectively.
  - Define instance members named m,n and **dispC()** and **sum()** under sub class C respectively.

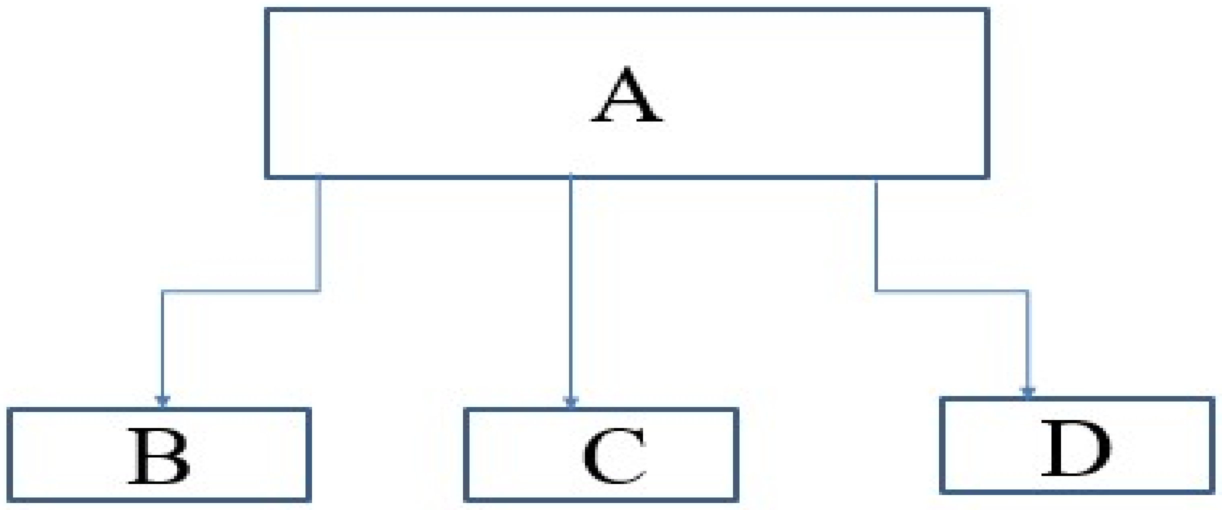
Activity 3=======

- The sum() method under C is expected to output the sum of (i+j+k+m+n) and dispC() outputs only the value of m and n.
- The dispA() method outputs the values of i and j when it is called through object of class C.
- The dispB() method outputs the value of k when it is called through object of class C.
- Define another class named Multiple to create separate objects named a1, b1 and b2 of class A, B and C respectively.

3. Hierarchical Inheritance

- In **hierarchical inheritance** one **parent class** will be **inherited** by many **sub-classes**.

- For example in the figure below shows class **A** will be inherited by **class B**, **C** and **D** respectively. **Class A** will be acting as a **parent class** for **B**, **C** and **D** and **class B**, **C** and **D** are acting as a **sub class or child class**.



Activity 4======

1. Write Java Program to demonstrate Hierarchical inheritance based on the following information.

- Define a super class named Calculate with width, height, depth and r as instance variables of a super class
- ✓ Define parameterized instance method named volume () to calculate volume of box at each time when it is called by objects of all sub classes.
- Define a subclass named RectArea that extends a super class Calculate with instance variable of width and height and parameterized instance method and rectArea().

Activity 4======

- ✓ The rectArea() calculate area of rectangle when it is called through objects of RectArea and returns a value to the caller.
- ▪ Define a subclass named TrigArea that extends a super class called Calculate with float type instance variable named width and height.
- ✓ Define parameterized instance method named trigArea() under TrigArea subclass. This method calculate area of triangle and returns a value to the caller when it is called through objects of TrigArea subclass.
- ▪ Define a subclass named CircleArea that extends a super class Calculate with instance variable of r.

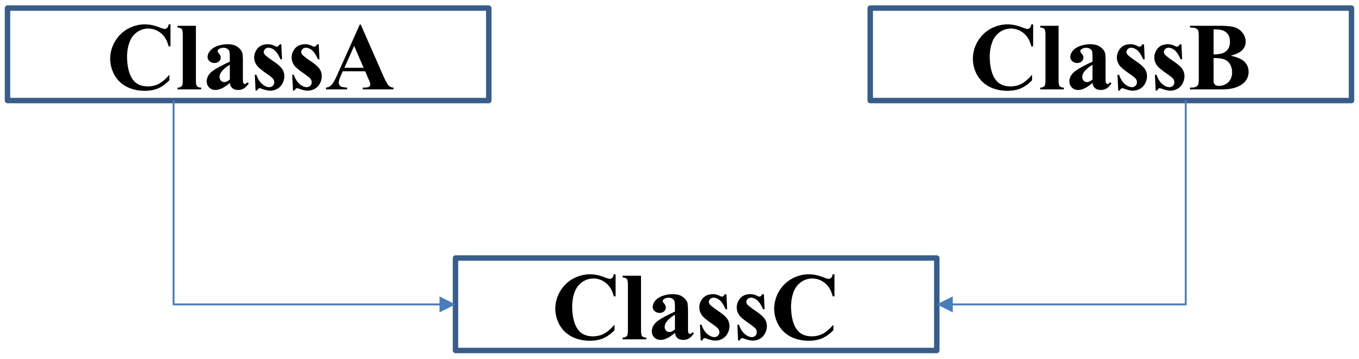
Activity 4=======

- ✓ Define parameterized method named circleArea() and this method calculate area of a circle and return a value to the caller when it is called through objects of CircleArea.
- Define another class named HierarchicalInheritance to create objects of a super and subclass
- Create objects of a class for both super class and subclasses you define.
- Call volume() through objects of each subclass
- Call rectArea() through objects of RectArea subclass
- Call trigArea() through objects of TrigArea subclass

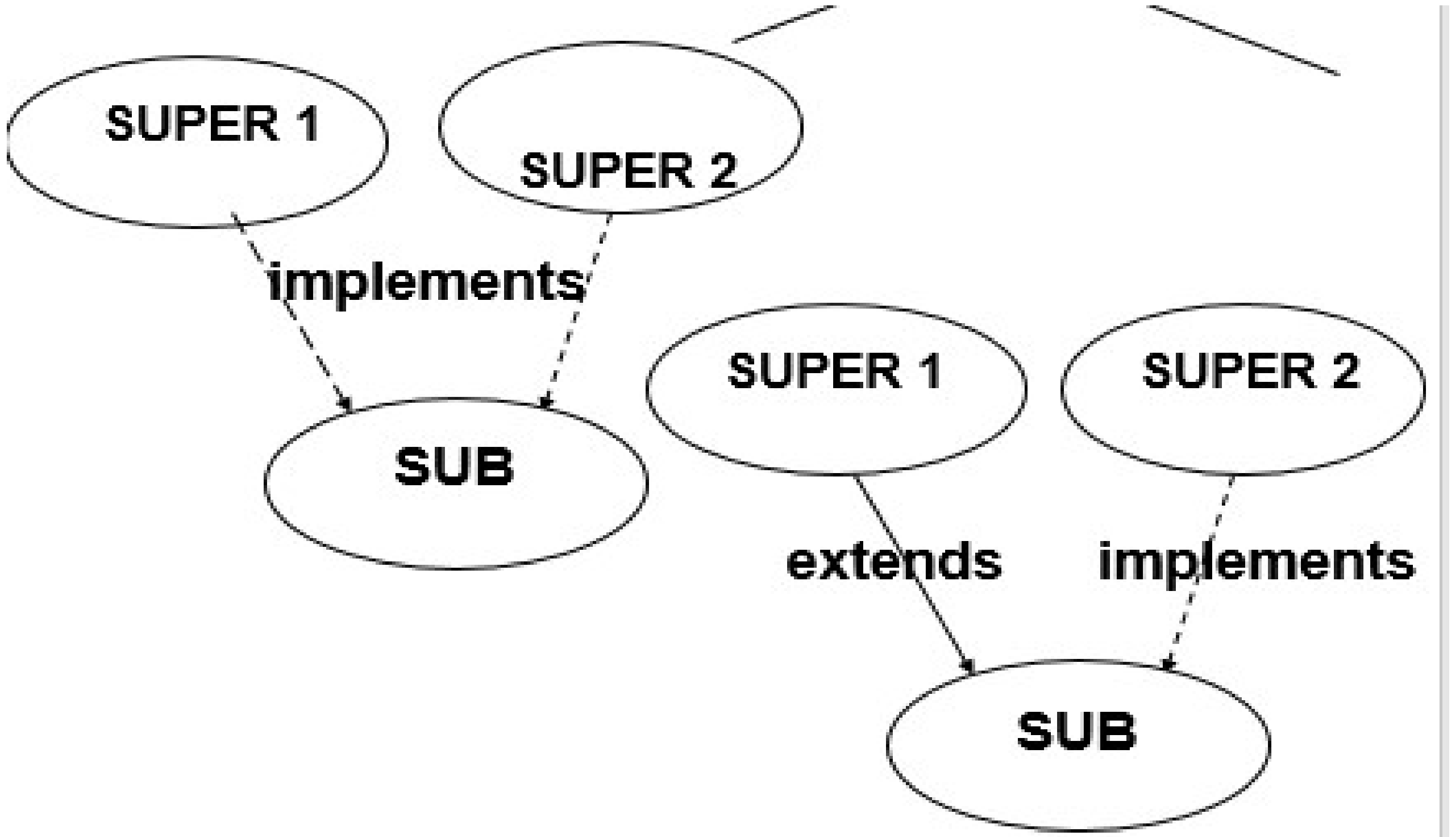
4. Multiple Inheritance in Java

- is the mechanism where ***one class extends*** more than ***one class.***

- ***Multiple inheritance*** basically is ***not supported*** by many ***Object Oriented Programming languages*** such as in ***Java***, ***C# and C++*** languages.
- As the ***child class*** has to ***manage*** the ***dependency*** of more than one parent class .
- But you can achieve ***multiple inheritance*** in ***Java*** using ***interfaces.***



~~. Multiple Inheritance~~



Reusing Codes

- Both procedural and Object Oriented Programming languages ***reuse codes*** that some one has already built and debugged.
- In Object Oriented Programming languages, there are ***two ways to reuse codes***.
- The first is, you simply ***create objects of your existing class*** inside the ***new class***.
- This is called ***composition***, because the ***new class*** is composed of ***objects of existing classes***.
- You are simply reusing the functionality of the code, not its form.

Reusing Codes-----

- The second one is **inheritance**.
- Inheritance creates a new class as a type of an existing class.
- In **inheritance**, you literally take the **form** of the **existing class and add code** to it without modifying the existing class.
- The concept of inheritance greatly enhances the ability to reuse code as well as making design a much simpler and cleaner process.

- **READING ASSIGNMENT ABOUT:**

- **HYBRID INHERITANCE**
- **COMPOSITION**

A Superclass Variable Can Reference a Subclass Object

- A ***reference variable of a superclass*** can be assigned a ***reference to any subclass objects.***
- You will find this aspect of inheritance quite useful in a variety of situations.

➢ ***For example, consider the following java program:***

```
/*Program to implement A Superclass Variable Can Reference
to a Subclass Object */
class Box {
double width;
double height;
```

A Superclass Variable Can Reference a Subclass Object

```
double depth;
// constructor used when no dimensions specified
Box() {}
//constructor used when all dimensions specified
Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
// compute and return volume
double volume() {
```

A Superclass Variable Can Reference a Subclass Object

```
return width * height * depth;
}//End of volume()
}//End of Box class
// Here, Box is extended to include weight.
class BoxWeight extends Box {
// BoxWeight instance Variable
double weight;
// constructor for BoxWeight
BoxWeight(double w, double h, double d, double m) {
width = w;
height = h;
```

A Superclass Variable Can Reference a Subclass Object

```
depth = d;
weight = m;
}//End of Constructor BoxWeight
}//End of BoxWeight class
class RefDemo {
public static void main(String args[]) {
//Declare, create and initialize BoxWeight object
BoxWeight weightbox = new BoxWeight(3, 5, 7, 8);
//Declare and create Box object class
Box plainbox = new Box();
double vol;
```

A Superclass Variable Can Reference a Subclass Object

```
vol = weightbox.volume();
System.out.println("Volume of weightbox is " + vol);
System.out.println("Weight of box is " + weightbox.weight);
// assign BoxWeight reference to Box reference
plainbox = weightbox;
vol = plainbox.volume(); // OK, volume() defined in Box
System.out.println("Volume of plainbox is " + vol);
// The following statement is invalid
//Because plain box does not define a weight member
//System.out.println("Weight of plainbox is "+ plainbox.weight);
}    }
```

A Superclass Variable Can Reference a Subclass Object

- Here, **weightbox is a reference to BoxWeight objects**, and **plainbox is a reference to Box objects.**
- Since **BoxWeight is a subclass of Box**, it is permissible to **assign plainbox a reference to the weightbox object**.
- It is important to understand that it is the **type of the reference variable—not the type of the object** that it **refers to—that determines what members can be accessed.**
- That is, when a **reference to a subclass object is assigned to a superclass reference variable**, you will have **access only to those parts of the object defined by the superclass**.

A Superclass Variable Can Reference a Subclass Object

- This is why **plainbox can't access weight even when it refers to a BoxWeight object**.
- If you think about it, this makes sense, because the **superclass has no knowledge of what a subclass adds to i**t.
- This is why the **last line of code in the preceding fragment is commented out.**
- It is **not possible for a Box reference to access the weight field**, because it **does not define one**.

Using Super

- In the **preceding example**, **classes derived from Box were not implemented as efficiently.**
- **For example**, the **constructor for BoxWeight explicitly initializes the width, height, and depth fields of Box( ).**
- Not only does this **duplicate code found in its super class**, which is inefficient, but it **implies that a subclass must be granted access to these members**.
- However, there will be times when you will want to **create a super class** that **keeps the details of its implementation to itself** (that is, that keeps its data members **private**).

Using Super----

- In this case, there would be **no way for a subclass to directly access or initialize these variables on its own**.
- Since **encapsulation is a primary attribute of OOP**, it is **not surprising that Java provides a solution to this problem**.

➢ Whenever a **subclass needs to refer to its immediate superclass**, it can do so by use of the **keyword super.**

- Remember that the variable, **this**, is used in the source code of an **instance method to refer to the object that contains the method**.
- The intent of the name this, is refers to **“this object”** the one right here that this very method is in.

Using Super----

- Whenever the **compiler executes an instance method**, it **automatically sets the variable this, to refer to the object that contains the method.**

- **Super** is **like this** but it can only be used to refer to **methods and variables** in the **super class**.

- Super has **two general forms.**

1. The first calls the **'super class' constructor**.
2. The second is used to access a **member of the super class** that has been **hidden by a member of the subclass.**

1. The First use of Super

- A **constructor** is used to **construct** an **instance** of a **class**.
  - Unlike **properties** and **methods**, the **constructors** of a **superclass** are **not inherited** by a **subclass**.
  - They can only be **invoked** from the **constructors** of the **subclasses** using the **keyword** **super**.
- The **syntax** to call a **superclass's constructor** is:

  super (); or super(parameter-list);
  - Here, **parameter list** specifies any **parameters** needed by the **constructor** in the **super class**.
  - **Super()** must always be the **first statement executed inside the subclass constructors.**

1. The First use of Super

- Let see the following Java program to demonstrate the first use of super (calling the constructor of a super class). Use Class A as a parent class and B as a child class of A and C as a child class of B to write the program.

Activity

```
class A{
 protected int i;
 public A(int i){
this.i=i;
}//End of Constructor A
public void M1(){
System.out.println("This is from A");
    }//End of M1()
}//End of A
class B extends A{
protected int j; //variables of B class
```

Activity

//Initialize the super class members i using super()

```
public B(int i, int j){
super(i); //calling the constructor of the super class
this.j = j;
}//End of Constructor
public void M2(){
System.out.println("This is from B");
    }//End of M2()
 }//End of class B
class C extends B{
protected int k;
```

Activity

//Initialize the super class (B)members i & j using Super()

public C(int i, int j, int k){

super(i, j); ***//Calling the constructor of Class B***

this.k = k;

}//End of constructor C

public void M3(){

System.out.println("This is from C");

}//End of M3()

}//End of class C

//Define another class to create objects of classes

class UsingSuper{

Activity

//Main Method

```
public static void main(String args[]){
B b=new  B(20, 40);
C o = new C(3, 5, 9);
```

//Call M1() and M2() using objects of B

```
b.M1();
b.M2();
System.out.println("i in A= " + b.i);
System.out.println("j in B= " + b.j);
```

//Call all versions of Classes using objects of C

```
o.M1();
```

Activity

```
o.M2();
o.M3();
System.out.println("i in A= " + o.i);
System.out.println("j in B = " + o.j);
System.out.println("k in C = " + o.k);
 }//End of main ()
}//End of class
```

- The implementation code in the previous example describe the **constructor C()** calls **super()** with parameter **i and j**.
- This cause the **B(int i, int j) to be called**.
- Then **B() calls super()** with **parameter i** and **initialize** j.

Activity

- **Super(i)** calls the constructor of the super class of A with integer argument, that is **A(int i) which initializes i**.
- This avoids the **duplication of codes and makes our program efficient and robust.**

➢ In addition to this **super ():**

- **Enables subclasses to initialize private members of super classes** and
- **To reuse the implementation of constructors defined in super class as constructor are not inherited.**

2. The Second use of Super

- ***Super*** can be used by ***subclasses*** to refer to ***members*** of their ***immediate super classes explicitly***.
- The second form of super acts somewhat like this, except that it always ***refers to the super class of the subclass in which it is used.***
- This usage has the ***following general form***. ***super.member;***
- **Here, member can be *either a method or an instance variable.***
- This form of ***super*** is most applicable to situations in which ***member names of a subclass hide members by the same name in the superclass***.

➢ Consider the following simple class hierarchy:

Activity

```
/* Using super to overcome name hiding. /*Using super to access a
   superclass members that is hide by a subclass */
class A {
  int i;
}//End of A
 // Define a subclass by extending class A.
class B extends A {
 int i; // this i hides the i in A
//Parameterized constructor of B
 B(int a, int b) {
 //Using super to access the i in A
```

Activity

```
super.i= a; // i in A
 i=b; // This i is i in B
  } //End of constructor
//Define a sub class method dispB()
 void dispB(){
//Display i in A using super
 System.out.println("i in superclass: " + super.i);
 //Display i in B
 System.out.println("i in subclass: " + i);
  } //End of dispB()
 } //End of sub class B
```

Activity

```
//Define another class to create objects of B
public class SecondUseOfSuper {
//Main method()
public static void main(String args[]) {
 B b = new B(100, 200);
 //Call dispB()
 b.dispB();
  } //End of main ()
} //End of class
```

- Although the instance variable **i in B hides the i in A**, **super allows access to the i defined in the superclass**.

Activity

1. Write Java program to access instance variables of a super class hidden by a subclass using ***super*** keyword. Use the following information to write the program:

- Define a super class by the name A and instance variables of i and j in A.
- Define a subclass by the name B that extends A with instance variables of i, j that hides i and j in A and k of B.
- Define Parameterized constructor of B and access the A members hide by B through the keyword super and access its sub class members directly.
- Define instance method dispB() in B to display i and j in A, k in B when it is called through objects of B.

Overriding Methods and Hiding Fields

- A **field in a subclass** that has the **same name as an accessible field in its super class hides the super class's version** even if their types are different.
- **For example,** if a super class declares a **public field**, subclasses will either **inherit or hide it.**
- If a subclass **hides a field**, the **subclass doesn't inherit** the **super class's version of the field** and **methods in the subclass can access the super class's version of the field** only by qualifying the simple name with the **super keyword, as in super.fieldName**.
- They can access the subclass's version by its simple name. The **syntax is :** **super.fieldName;**

Activity

```
class C1{
protected float x;
protected int y;
//Parameterized Constructor
public C1(float x, int y){
this.x = x * x;
this.y =y;
    }//End of C1 constructor
}//End of C1 class
class C2 extends C1{
private float x;//This x hide x in C1
```

Activity

```
private float z;
//Parameterized Constructor
public C2(float x, int y, float z){
super(x,y);
this.x =x;
this.z = z;
}//End of C2 Constructor
public void M1(){
z = (super.x + y)/x;
System.out.println("Superclasses version of x: " +super.x);
System.out.println("Subclasses version of x:" +x);
```

```
class HideSuperClassField{
public static void main(String args[]){
C2 o = new C2(2, 4, 6);
o.M1();
System.out.println("y  = " +o.y);
}
}
```

- In the above example, C2 ***inherits y and hides x*** by defining another ***integer variable x***.
- In the expression ***z = (super.x + y)/x;*** super.x refers the ***superclass version of x*** while, ***x*** refers the ***subclass's version***.
- Java permits you to declare a field in a subclass with the same name as a field in a superclass so you can add fields to a class without warring about breaking compatibility with already existing subclasses.
- Java's willingness to tolerate hidden fields makes subclasses more accepting of changes in their super classes.

Overriding Instance Methods

- An ***instance method*** in a subclass with the same signature(***name plus the number and the type of its parameters***) and ***return type*** as an instance method in the super class ***overrides*** **the** ***super class's method*****.**
- The ability of a subclass to ***override a method*** allows a class to ***inherit from a super class*** **whose behavior is "close enough" and then to modify behavior as needed.**
- The ***overriding method*** has the ***same name, number and type of parameters, and return type*** as the method it ***overrides***.
- 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***.

Activity

1. Write Java Program to demonstrate overriding method based on the following information.

- Define a super class by the name A having m, n instance variable and M1() instance methods. The M1() displays the values of m and n when it is called through objects of A.
- Define Parameterized Constructor of a super class A and this super class constructor is called using super from the subclass.
- Define a sub class by the name B by extending A with instance variable of k.
- Define parameterized constructor of B and call the super class constructor through super and initialize k.
- Define M1() method in subclass B to override the superclass version. When this method is called it is the sub class version of the method. Call the super class version using super.M1() and this method displays the values of m and n in A.

```
//Java Program to demonstrate Method overriding.
//Define a super class by the name A1
class A1  {
//Instance variable of super class
int i, j;
//Parameterized Constructor
A1(int a, int b) {
i=a;
j=b;
} //End of Constructor A1
/*Define dispA() to display i and j when it is called using
   super within a sub class */
void dispA(){
 System.out.println("i and j: " + i + "  " + j);
  } //End of dispA()
} //End of class A1
```

```
//Define subclass B1 that extends superclass A1
class B1 extends A1 {
int k;
//Define Parameterized constructor B1
B1(int a, int b, int c) {
//Calling constructor of A1 using super
super(i, j);
k=c;
 } //End of constructor B1
//This method overrides the method in A1
  void dispA(){
 //Call the superclass version of dispA() using super
  super.dispA();
System.out.println("k: " + k);
  }//End of dispA()
}//End of Class B1
```

```
//Define another class to create objects of sub class
class MethodOverriding {
//Main method()
 public static void main(String args[]) {
 //Declare, create and initialize objects of A1
 A1 a=new A1(50,90);
 //Declare, create and initialize objects of B1
 B1 ob = new B1(10, 20, 30);
 //CalldispA()-the super class version of A1
 a.dispA();
 // This calls dispA() in B1 not in A1
ob.dispA();
}//End of main ()
} //End of Class
```

- When ***dispA( ) is invoked on an object of type B***, the version of ***dispA( ) defined within B is used***.
- That is, the version of ***dispA( ) inside B overrides the version declared in A***.
- But to ***access the superclass version*** of an ***overridden*** method, you can do so by ***using super***.
- For example, in this ***version of B***, the superclass version of ***dispA( )*** is ***invoked*** within the ***subclass***' version.
- This allows all instance variables of the superclass version to be displayed.

- ***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***. For example, consider this modified version of the preceding example to demonstrate method overloading:

```
// Methods with differing type of signatures
//are overloaded – not  overridden.
//Define a super class
class AO {
//Instance variables of a super class
int i, j;
//Parameterized constructor of a super class
AO(int a, int b) {
i=a;
j=b;
} //End of constructor AO
```

```
// display i and j  in AO
 void dispA() {
System.out.println("i and j: " + i + " " + j);
} //End of dispA()
}//End of super class AO
// Define a subclass by extending class AO.
class BO extends AO {
 //instance variable of BO
int k;
BO(int a, int b, int c) {
//Call the super class Constructor using super
super(a, b);
 k=c;
}//End of BO
// overload dispA() method defined in AO
```

```
void dispA(String msg) {
System.out.println(msg + k);
  } //End of dispA()
}//End of BO Class
//Define another class to create objects of a sub class
class NotOverriding {
public static void main(String args[]) {
BO ob = new BO(10, 20, 30);
ob.dispA("This is k in B"); // this calls dispA() in B
ob.dispA();// this calls dispA() in A
  } //End of main ()
} //End of class
```

- The version of dispA( ) in B takes a string parameter.
- This makes its ***type signature different from the one in A***, which ***takes no parameters***.
- Therefore, ***no overriding (or name hiding)*** takes place.

Overloading vs. Overriding

▪ Overloaded methods can be either in the same class or different classes related by inheritance ▪ Overloaded methods have the same name but a different signature.	▪ Overridden methods are in different classes related by inheritance ▪ Overridden methods have the same signature and return type;

Activity

- Write Java program to create a class Shape with methods getArea() and getPerimeter(). Create three subclasses: Circle, Rectangle, and Triangle. Override the getArea() and getPerimeter() methods in each subclass to calculate and return the area and perimeter of the respective Shapes

Hiding Class Methods

- If a subclass defines a ***class method with the same signature*** as a class method in the ***superclass***, the ***method in the subclass hides*** the one in the superclass.
- The distinction between hiding and overriding has important implication.
- The version of the ***overridden method*** that gets ***invoked is the one in the subclass***.
- The version of the ***hidden method*** that gets ***invoked depends*** on whether it is ***invoked from the super class or the subclass.***

Example 10

- Look at the following example that contains two classes, ***Animal and Cat***.
- Animal contains one instance method and one class method.
- The Cat class overrides the instance method in Animal and hides the class method in Animal.

```
class Animal{
      public static void classMethod(){
      System.out.println("The class method in Animal.");
}
      public void instanceMethod(){
System.out.println("The instance method in Animal.");
      }
}
```

```
   class Cat extends Animal{
        public static void classMethod(){
        System.out.println(“The class method in Cat.”);
        }
        public void instanceMethod(){
        System.out.println(“The instance method in Cat.”);
        }
}
   class Test{
        public static void main(String args[]){
        Animal a1 = new  Cat();
        a1.classMethod();
        a1.instanceMethod();
        }
}
```

- The main method in Test class ***creates an instance of Cat*** which is referenced by a ***reference variable of type Animal*** and calls classMethod() and instanceMethod() on it.
- The output from the above program is as follows.

```
The class method in Animal
The instance method in Cat
```

- As promised, the version of the ***hidden method that gets invoked is the one in the superclass***, and the ***version of the overriden method that gets invoked is the one in the subclass***.

Activity

1. Execute the above example
2. Add the following lines of codes to the main method execute it and justify the output.

```
Cat myCat = new Cat();
Animal a2 = new Animal();
a2.classMethod();
myCat.classMethod();
a2.instanceMethod();
```

3. By writing a simple Java program check the following.

a) The access specifier for an overriding method can allow more, but not less, access than the overridden method. For example, a protected instance method in the superclass can be made public, but not private, in the subclass.

b) You will get a compile-time error if you attempt to change an instance method in the superclass to a class method in the subclass, and viceversa.

c) In a subclass, you can overload the methods inherited from the superclass. Such overloaded methods hide nor override the superclass methods- they are new methods, unique to the subclass.

Overview of Polymorphism

- Another essential feature of ***object-oriented programming*** is ***polymorphism***
- Using the ***same method name*** to indicate ***different implementations***
- ***allows one interface*** to be used for a ***general class of actions***
- is often expressed by the phrase ***"one interface, multiple methods."***
- This means that it is possible to design a generic interface to a group of related activities.
- Shortly , polymorphism means ***"many forms"***

upcasting

- When you create a **superclass** and one or more **subclasses**, each ***object*** of the ***subclass “is a” superclass object***

- Because every ***subclass “is a” superclass member.***
- ***Therefore, you can convert subclass objects to superclass objects***. This is called **up casting.**
- An ***object*** can be treated as ***its own type*** or as an ***object of its base type***.
- *Casting* from a ***derived type*** to a ***base type*** moves up on the ***inheritance*** diagram, so it’s commonly referred to as ***up casting***.
- **Up casting** is always **safe** because you’re going from a more ***specific type*** to a more ***general type***.

- That is, the ***derived class is a superset of the base class.***
- It might ***contain more methods*** than the ***base class***, but it must contain at ***least*** the ***methods*** in the ***base class.***
- The only thing that can occur to the ***class interface during the upcast*** is that it can ***lose methods, not gain*** them.
- This is why the compiler allows ***upcasting*** without any explicit casts or other special notation.

➢ Remember that a ***subclass and its base class has an 'is a' relationship***.

- That is the ***new(derived) class*** is a ***type of the existing(base) class***.

```
/*Java program to demonstrate up casting-to assign objects
   of  a sub class to reference variable of a super class */
class Box1{
protected double width;
protected double height;
protected double depth;
//Parameterized constructor
public Box1(double w, double h, double d){
width = w;
height = h;
depth = d;
}//End of constructor
public double volume(){
return width * height * depth;
   }//End of volume()
}//End of super class
```

```
//Define a sub class by extending a super class
class MatchBox extends Box1{
private double weight;
//Parameterized constructor of a sub class
MatchBox(double w, double h, double d, double weight){
//Call the super class constructor using super
super(w, h, d);
this.weight = weight;
}//End of sub class Constructor
void dispB(){
 System.out.println(“Value of weight MatchBox:"+weight);
 }//End of dispB()
}//End of class
//A  class defined to create objects of a super and sub class
class UpcastingExample{
//Main Method
public static void main(String args []){
//Declare Reference variable of a super class Box1
Box1 b1, b2;
//Declare, create and initialize objects of a sub class
MatchBox mb1 = new MatchBox(20, 4,10,3);
//Assign obj of sub class to reference variable of super class
b1 = new MatchBox(10, 5, 6,9); //Up casting
//Assign mb1 object to reference variable of a super class
b2 = mb1; //Upcasting
System.out.println("Volume of b1= "+b1.volume());
System.out.println("Volume of b2= "+b2.volume());
System.out.println( "Volume of mb1 = "+ mb1.volume());
//Calling a sub class method dispB()
mb1.dispB();
   }//End of main ()
}//End of class
```

- A new ***object*** of type ***MatchBox*** class is assigned to ***reference variable*** of type ***Box 1*** class, ***b1***.
- In the next line another ***MatchBox*** ***object referenced*** by ***mb1*** is assigned to the ***reference variable b2*** which of type ***Box 1 class***.
- This means we trust ***mb1 as type of (object of) Box 1 class***.
- That is we **upcast mb1 to its base type**.

Example 12

```
class A3{
protected int i;
public A3(int i){
this.i = i;
}//End of Constructor
//Define a method as object of a class is a parameter
public void Increment(A3 base){
base.i++;
System.out.println("Incremented I = "+base.i);
   }//End of Increment()
}//End of class A3
//Define a sub class by the name A
class B3 extends A3{
//Parameterized Constructor
public B3(int i){
```

```
//Calling a super class Constructor
super(i);
}//End of Constructor
//Define Parameterized method
//The parameter is its object of a class
public void Decrement(B3 sub){
sub.i--;
System.out.println("Decremented I = "+sub.i);
   }    //End of Decrement()
}//End of Sub class B3
//Define a class to create objects of super and sub class
class UpcastingExample2{
public static void main (String args[]){
//Declare Reference Variable of super class
A3 a1;
//Declare Reference Variable of a sub class
```

```
B3 b1, b2;
//create and initialize  objects of b1
b1=new B3(400);
//Create and initialize objects of a sub class
b2 = new B3(22);
//Assign Objects of a sub class to reference variable of A3
a1=b2;
a1=b1;
//Create and Initialize objects of A3
a1 = new A3(110);
//Call through a1 and pass subclass object b1
a1.Increment(b1);
//Call through b1 and pass sub class object b2
b1.Decrement(b2);
```

//Call through a1 and pass sub class object b2

a1.Increment(b2);

//Call through b2 and pass sub objects of b2

b2.Decrement(b1);

//Call through b1 and pass objects of a1

b1.Increment(a1);

}//End of main ()

}//End of class

➢ ***Note***

- Here, the ***Increment method expects an object of class*** as an argument.
- But ***an object of class B3 that is b1 and b2 is passed to it***.
- This means ***upcast b1 and b2 to the type of A3.***

Implementing Polymorphism

- When an ***overridden method*** is called from within a ***subclass***, the ***subclass version of that method is referred***.
- To call the ***super class version of the overridden method*** from with in ***a subclass "super" should be used***.
- Because of ***up casting***, a ***super class reference variable can refer to a subclass object***.

➢ When an ***overridden method is called through a super class reference***:

- ***Java determines*** which version of that ***method to execute*** based upon the ***type of the object being referred*** to at the time the ***call occurs i.e at run time***.

- In other words, it is the ***type of the object*** being ***referred*** to(***not the type of the reference variable***) that ***determines which version*** of an ***overridden method will be executed***.

➢ **To *implement polymorphism*:**
  - ***Create a superclass reference variable that refers to subclass objects,***
  - ***Call overridden methods through this reference variable***

➢ ***Example:***

- Write Java Program to implement polymorphism. Define a super class A and define dispA() method. Define another sub class B and override dispA() method under B. The program determine which version of overridden method is called through the object not through the reference variable.

```
//Define a super class A
class A9{
//Define Super class Method dispA()
public void dispA(){
System.out.println("This is from the Superclass.");
    }//End of dispA()
}//End of super class
//Define a sub class B by extending A
class B9 extends A9{
//Sub class override a super class method dispA()
public void dispA(){
System.out.println("This is from sub class.");
}//End of dispA()in sub class
}//End of sub class B
//Define another class to create objects of a sub class
//and reference variable of a super class
```

```
class ImplementingPolymorphism {
//Main Method
public static void main(String args[]){
//Declare and create objects of B
B9 b = new B9();
//A super class reference variable refers to a sub class object
A9 a = b;
//Call a sub class version of dispA()
a.dispA();
//Create objects of a super class
a = new A9();
//Call a super class version of dispA() through objects of A
a.dispA();
    }//End of main ()
}//End of class
```

- The first call to ***dispA() calls the subclasses version*** because the ***super class reference variable "a" refers an object of the subclass, B***.
- Where as the second call to ***dispA() calls the super class version***; because this time "***a" refers an object of the superclass, A***.
- This shows that ***which version of the overridden method to be called*** is determined by the ***object being referenced***.

➢ ***Java determines the version of an overridden method to be called at run time***.

- This is called ***dynamic method dispatch*** or ***dynamic binding***.
- **Dynamic method dispatch** is the mechanism by which a ***call to an overridden method*** is resolved ***at run time, rather than compile time***.

Example

1. Write Java Program to demonstrate dynamic dispatch method based on the following information:

- Define a super class by the name Shapes and area() method.
- Define three sub classes by the name **Rect, Circle and Trig.**
- All sub classes define its own private members of instance variables and parameterized constructor.
- All of the subclasses **override the area() method** defined in their superclass and perform its operation called through reference variables of a super class.

```
class Shapes{
public void area(){
System.out.println("Difficult to find area of general
    Shape.");
    }//End of area()
}//End of Shapes class
//Define a sub class Rect by extending Shapes
class Rect extends Shapes{
//Define its own private members of Rect
private double width;
private double height;
//Parameterized Constructor
public Rect(double w, double h){
width=w;
height=h;
}//End of constructor
//Override area()by Rect Sub class
public void area(){
System.out.println("Area of Rectangle is:" +(width * height));
   }//End of area() under Rect class
}//End of Rect Class
//Define a sub class Circle by extending Shapes
class Circle extends Shapes{
//Define its own private members of Circle class
private double radius;
public Circle(double r){
radius  = r;
}//End of constructor
//Circle class Override area() of a super class
public void area(){
System.out.println("Area Circle:"+Math.PI * radius * radius);
   }//End of area() of Circle sub class
}//End of Circle sub class
```

```
//Define Trig sub class by extending Shapes super class
class Trig extends Shapes{
private double base;
private double height;
//Parameterized Constructor
public Trig(double b, double h){
base= b;
height =h;
}//End of constructor
public void area(){
System.out.println("Area of triangle: "+ 0.5 *(base* height));
   }//End of area()
}//End of Trig Class
```

```
class DynamicDispatch {
public static void main(String args[]){
//Declare and create objects of a Super class shapes
Shapes s = new Shapes();
//Call the Shapes version of area()
s.area();
//Reference variable s refers to  an object of type Rect
s = new Rect(3, 4);
//Call the Rect version of area()
s.area();
//Reference variable s refers to  an object of type Circle
s = new Circle(5);
//Call the Circle version of area()
s.area();
//Reference variable s refers to  an object of type Trig
s = new Trig(3, 4);
s.area(); //Call the Trig version of area()
    }//End of main ()
}//End of class
```

- In main(), a ***reference variable of type Shapes***, called ***s is declared and four calls to area() method are made through s.area().***
- As the output shows, the ***first call to area() invokes the super class' version of area().***
  - This is because at that time, ***s refers an object of type Shapes.***
  - The ***second call, s.area(),*** invokes the ***Rect's area() method as s refers an object of type Rect at that time***.
  - With the same reason, the third and fourth calls to area() invokes the Circle's and Trig version of area().
  - Therefore, if a ***super class contains a method*** that is ***overridden by a subclass***, then when different types of ***objects are referred*** to through ***super class reference variable***, different versions of the ***method are executed***.

- Here one ***interface, s.area()*** is used to call different ***implementation of area() method***.
  - This means there is one ***interface***, ***s.area() for multiple methods.***
  - ***This is one way of implementing polymorphism***.
  - Even more methods that ***overrides area()*** can be added without ***affecting the interface***.
- ***Polymorphism*** allows a ***general class*** to specify ***methods*** that will be ***common*** to all of its derivatives, while allowing ***subclasses*** to define the ***specific implementation*** of some or all of those methods.
  - The ***super class provides all elements that a subclass can use directly.***
  - Thus, by ***combining inheritance with overridden methods***, a ***super class can define the general form of the methods that will be used by all of its subclasses.***

Abstract classes and Methods

- In the previous ***Example***, Shapes class is ***unable to create a meaningful implementation for area() method***.
- The definition of ***area()*** is simply **a *placeholder***.
- It will ***not compute and display the area of any type of shape*** as ***shape is too general***.
- The ***intent of defining area() in the super class*** is to create a ***common interface for all the classes derived from it***.
- ***Shapes defines a generalized form of area()*** that will be ***shared by all of its subclasses.***
- In such type of situation the ***implementation of area() can be left,*** that is it can be an ***abstract method***.
- An ***abstract method*** is an ***incomplete method*** that has ***only a declaration and no method body***.

- S*yntax to declare an abstract method:*

  abstract return_type methodName(parameter-list);

- ***For example to make the area method in the Shapes class*** as an ***abstract method***, we define as: ***abstract void area();***
- When we define a class to be ***“final”,*** it cannot be ***extended*** or ***inherited*** or not to be ***overridden or overloaded***.
- But it is possible to ***define*** a ***super class to be abstract*** if we want the ***properties of the super classes to be always extended and used by sub classes.***
- ***Abstract classes*** provides a ***common root for a group of classes***, nicely ***tied together in a package***.
- A class containing ***abstract method*** is called an ***abstract class***.
- If a ***class contains one or more abstract methods***, the ***class itself must be qualified as abstract as shown below.***

```
abstract class Shapes{
        abstract void area();
        }
```

- An ***Abstract class is a conceptual class***.
- An ***Abstract class cannot be instantiated – objects cannot be created.***
- ***For example***, if the class is defined as ***abstract***, the statement:

  Shapes s = new Shapes(); //is not allowed and is an error
- If you ***inherit from an abstract class*** and need to ***make objects of the new type***, you must provide ***method definitions for all the abstract methods in the base class***.
- If you ***don't then the derived class is also abstract***, and the compiler will force you to qualify that class with the **abstract keyword**.

Abstract class Syntax

```
abstract class ClassName {
    ...
    abstract  returnType MethodName1();
    …
    //Non Abstract Method
    returnType Method2(){
    // Body of non-abstract method
    }
}
```

- When a ***class contains one or more abstract methods***, it should be declared as ***abstract class***.
- The ***abstract methods of an abstract class*** must be **defined in its subclass.**

Abstract class Example

```
public abstract class Shapes {
        //Abstract method
        public abstract double area();
        // Define non-abstract method
        public void move() {
        // Body of non-abstract method
        }
}
```

Abstract Classes Properties

- A class with ***one or more abstract methods*** is automatically ***abstract and it cannot be instantiated.***
  - A ***class declared as abstract***, even with ***no abstract methods*** **can** ***not be instantiated***.
  - A ***subclass of an abstract class*** can be ***instantiated*** if it ***overrides all abstract methods*** by ***implementing them***.
  - A ***subclass that does not implement*** all of the ***superclass abstract methods*** is ***itself abstract; and it cannot be instantiated.***
- ***Example***: Write Java program to demonstrate abstract class and abstract method based on the following information:
  - Define abstract class by the name Staff and its instance variables of name, age and salary.
  - Define Parameterized constructor of Staff class

- Define abstract method by the name income() and two non-abstract method by the name tax() and pension() respectively. The tax () return(salary*.35)-1500 and the pension() returns(salary*.07)
- Define a sub class by the name AcademicStaff, houseAllownace as an instance variable, parameterized constructor and implement abstract income() of abstract class and the method return values to the caller as follows:

return ((salary+houseAllowance)-((tax())+(pension())));

- Define a sub class by the name AdminStaff, parameterized constructor and implement abstract income() of abstract class and the method return values to the caller as follows:

return salary-((tax())+(pension()));

- Define Reference variable of Abstract class by the name s and s refers to objects of each sub class.
- The program displays the name and income of employees in each class separately.

```
class TestAbstract{
 //Main method
  public static void main(String[] args) {
 //Declare reference variable of abstract class
    Staff s;
  //Abstract class reverence variable s refers to objects of
AcedemicStaff
 s= new AcademicStaff("Zelalem", 34, 40000, 400);
 //Call non abstract method
 System.out.println("Income Tax of Employee="+s.tax());
 System.out.println("Pension of an employee="+s.pension());
//Call ovrrided method and display Name, age, and salary
System.out.print("Name of Employee is"+ " "+s.name+"
"+",his age is"+ " "+s.age);
System.out.println("and his income is"+" "+s.income());
//Abstract class reverence variable s refers to objects of
AdminStaff
s=new AdminStaff("Zinash", 34, 20000);
//Call non abstract method
     System.out.println("Income Tax of Employee="+s.tax());
    System.out.println("Pension of an
employee="+s.pension());
//Call ovrrided method and display Name, age, and salary
System.out.print("Name of Admin Staff is"+" "+s.name+" "+"
his age is="+s.age);
System.out.println("and his salary="+s.income());
   }
}
```

```
double pension(){
   return (salary*.07);
}//End of pension ()
}//End of abstract class
//Define sub class AcademicStaff that extends  abstract class
class AcademicStaff extends Staff{
//Define its own private instance variables
private double houseAllowance;
//Define Parameterized Constructor
public AcademicStaff (String str, int a, double s, double ha){
//Calling super class constructor using super key word
super(str, a, s);
houseAllowance = ha;
}//End of constructor
//A sub class implement a super class abstract method
public double income(){
```

```
return ((salary+houseAllowance)-((tax())+(pension())));
    }//End of income()
}//End of AcdemicStaff class
//Define a sub class by extending abstract class
class AdminStaff extends Staff{
//Parameterized constructor
public AdminStaff (String str, int a, double s){
//Calling a super class abstract constructor
super(str, a, s);
}//End of constructor
//AdminStaf Subclass implement abstract income() method
public double income(){
return salary-((tax())+(pension()));
    }//End of income ()
}//End of AdminStfaa sub class
```

```
class AbstractEx {
public static void main (String args[]){
//Declaration of reference variable of abstract class
Staff s;
//Abstract class reverence variable s refers to objects of AcedemicStaff
s= new AcademicStaff("Zelalem", 34, 4000, 400);
System.out.print("Employee Name is"+" "+s.name+" ");
System.out.println("and his Income  =: "+" "+s.income());
//Abstract class reverence variable s refers to objects of AdminStaff
s = new AdminStaff("Ahmed", 36, 2000);
System.out.print("Employee Name is"+" "+s.name+" ");
System.out.println("and his Income  = :"+s.income());
//s=new Staff();Staff is abstract class it cannot be instantiated
    }//End of main()
}//End of AbstractExample class
```

- The ***Staff class*** declares the structure of a given ***abstraction without providing a complete implementation of every method.***
- That is, it defines the ***member variables and the implementation of tax() method***, leaving the ***income() method to each subclass*** to fill in the details as it is calculated differently in different types of staffs.
- Such a ***class determines the nature of the methods*** that the ***subclasses must implement***.
- Both subclasses AcademicStaff and AdminStaff , ***implement (override) the abstract method income().***
- Objects of these classes are assigned to a ***reference variable, s of type Staff by up casting***.
- This ***reference variable, s is used to call both implementations***.
- ➢ Notice that ***one interface is used to call the two implementations***.
- This is ***what polymorphism is***.

Using final keyword with Inheritance

- In Java, the ***final keyword*** can be used in three places: for ***data, methods and classes***.
- ➢ The ***keyword final has three uses.***
- First, it can be used to create the equivalent of a ***named constant***. The other ***two uses of final apply to inheritance.***
- ➢ ***Generally the following are the three uses of final keyword:***
- To ***declare constants which can not change its value of definition***.
- ***Used in inheritance for final methods not to be Overridden or Over Loaded***
- ***Used in inheritance for final Classes not to be extended or inherited***

Using final to Prevent Overriding

- While ***method overriding*** is one of Java's most powerful features, there will be times when you will want to ***prevent it from occurring***.
- To ***disallow a method from being overridden***, specify ***final as a modifier*** at the start of its declaration.
- Methods declared as ***final cannot be overridden***. The following fragment illustrates final:

```
class A {
  final void meth() {
    System.out.println("This is a final method.");
  }
}
 class B extends A {
  void meth() { // ERROR! Can't override.
    System.out.println("Illegal!");
  }                    }
```

- Because ***meth( ) is declared as final***, it cannot be overridden in B.

If you attempt to do so, a ***compile-time error will result***.

- Methods declared as ***final*** can sometimes provide a ***performance enhancement***:
- The compiler is free to inline calls to them because it "knows" they will not be overridden by a subclass.
- When a small final function is called, often the Java compiler can copy the bytecode for the subroutine directly inline with the compiled code of the calling method, thus eliminating the costly overhead associated with a method call.
- Inlining is only an option with final methods.
- Normally, Java resolves ***calls to methods dynamically***, at run time. This is called ***late binding***.
- However, since ***final methods cannot be overridden***, a call to one can be resolved at compile time. This is called ***early binding***.

Using final to Prevent Inheritance

- Sometimes you will want to prevent a ***class from being inherited***.
- To do this, precede the ***class declaration with final***.
- ***Declaring a class as final implicitly declares all of its methods as final, too.***
- As you might expect, it is illegal to declare a class as both ***abstract and final*** since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations.
- Here is an example of a final class:

```
final class A {
 }
 // The following class is illegal.
class B extends A { // ERROR! Can't subclass A
 }
```

- As the comments imply, it is illegal for B to inherit A since A is declared as final.

INTERFACES AND INNER CLASSES

Interface Overview

- The **interface keyword** takes the **abstract concept** one step further.

➢ You could think of it as a **"pure" abstract class**.

- It allows the creator to establish the form for a class: **method names, parameter lists,** and **return types,** but no **method bodies.**

- An **interface** can also contain **fields**, but these are implicitly **static** and **final**.

➢ An **interface** is a mechanism that provides only a **form**, but **no implementation**.

Interface continued-----

- An interface specifies **what** a **class must do**, but **not how** it **does** it.
- Interfaces **lack instance variables** and their **methods** are **declared without** any **body**.

➢ An interface says: "This is what all **classes** that **implement** this **particular interface** will look like."

- Thus, **any code** that **uses** a particular **interface knows what methods** might be **called** for that **interface,** and that's all.

✓ So the **interface** is used to establish a **"protocol"** between **classes.**

Differences between Classes and Interfaces

INTERFACES	CLASSES
▪ **Lack instance variables**, and their **methods are declared without any body.**	▪ Has its **own instance variables and their methods also declared** with **body**
▪ Provides one class would implement **any number of interfaces.**	▪ **It requires an interface for a class to implement any number of interfaces**
▪ Designed to support **dynamic method resolution at run time**	▪ **Designed to support method declaration immediately when it is**

Continued--

Interface is not instantiated since it lacks instance variables.	**Classes are instantiated since it has its own instance variables.**
Interfaces are developed to support multiple inheritance.	**Class alone unable to implement multiple inheritance**
Methods present in interfaces are **pure abstract**	Methods present in classes are **pure abstract only when the class is defined as abstract**
Use only **public** specifier in **interface**	Use **public**, **final** and **abstract** specifier in **classes**

Defining an Interface

- To **create an interface**, use the **interface keyword**

✓ ***The general Syntax of interfaces is shown below:***

```
Access modifier interface_ keyword  interface_name {

return-type method-name1(parameter-list);

return-type method-name2(parameter-list);

type final varname1 = value;

type final varname2 = value;

return-type method-nameN(parameter-list);

type final varnameN = value;

}
```

Defining an Interface continued

- **Where**
  - The access modifier to be used must be either **public or default**
  - **InterfaceName**-name of the interface and can be any valid identifier.
  - The ***methods are abstract methods***.
  - ✓ They have ***no bodies and end with a semi colon*** after the parameter list.
  - ✓ They are, essentially, **abstract methods**; there can be **no default implementation** of any method specified within an interface

Defining an Interface continued

- ✓ Each **class** that includes an **interface** **must implement all of the methods.**
- ✓ You can declare the methods in an **interface** as **public**, but they are **public** even if you don't say it.
- ➢ **Variables** can be **declared inside** of **interface** declarations.
- ▪ The **variables** are implicitly **static** and **final**, meaning they **cannot** be **changed** by the **implementing class**.
- ▪ They **must** also be **initialized with a constant value**.
- ▪ All **methods** and **variables** are **implicitly public** if the **interface,** itself, is **declared** as **public**.

Defining an Interface continued

- Here is an **example of an interface definition**.
  - It declares a **simple interface which contains one method called callback( )** that takes a **single integer parameter**.

```
interface Callback {
void callback(int param);
}
```

Implementing Interfaces

- Once an **interface** has been **defined**, **one** or **more classes** can **implement** that **interface**.
- To **implement** an **interface**,

i) **Include** the **implements clause** in a **class definition.**

ii) **And** then **create** the **methods defined** by the **interface.**

- The **general form** of a **class** that **includes** the **implements** clause looks like this:

Access-Modifier class-keyword Class-name implements interface1, interface2, …, interfaceN {

…//body of the class

}

Implementing Interfaces continued

- The type **signature** of the **implementing method** must match exactly the **type signature specified** in the **interface definition**.
- Reducing the **accessibility** of a **method** during **inheritance** is **not allowed**.
- So when you **implement** an **interface**, the methods from the **interface** must be **defined** as **public**.

Uses of Interface

- An interface can not be **instantiated or not create objects**.
- But you can declare **reference variables** of **interface types**.
- By **up casting,** any **instance** of any **class** that **implements** the **declared interface** can be stored in such reference variable.
- This enables us to **access a class object** through **interface reference**.
- When you call a **method** through one of these reference, the correct **version** will be **called based** on the **actual instance** of the **interface being referred to**.

Uses of interface continued

- ✓ This is one of the key **features of interfaces**.
- The **method** to be **executed** is **looked up dynamically** at **run time**, allowing classes to be **created later than** the **code** which **calls methods**.

Activity on Interface

1. Write Java program to demonstrate an interface that implemented by classes based on the following information:

- Define an interface named Shape and abstract method of interface named area(). This abstract method is implemented by each class that implements Shape interface.
- Define a class named CircleArea that implements the interface Shape.

✓ Define private instance variable named r, parameterized constructor under this class. This class implement the abstract method area() and the method calculate area of a circle and returns a value to the caller.

Activity on Interface continued

- Define a class named RectArea that implements the interface Shape.

✓ Define its own private instance variable under this class named width, height and parameterized constructor. This class implement the abstract method area() and the method calculate area of a rectangle and returns a value to the caller.

- Define a class named TrigArea that implements the interface Shape.

✓ Define its own private instance variable under this class named width, height and parameterized constructor. This class implement the abstract method area() and the method calculate area of a triangle and returns a value to the caller.

Activity on Interface continued

- Define another class named ImplementInterface that contains the main method.
- Under the main method, define reference variable of an interface named s, and this reference variable refers to objects of each class that implements the interface.
- Call each version of area() through reference variable of the interface and print the output of the returned value.

Activity continued

```
interface Shape{
double area();
}//End of interface
//Define a class named CircleArea that implements Shape interface
class CircleArea implements Shape{
//Define its own private instance variable
private double r;
//Define parameterized constructor
public CircleArea(double r){
rad=r;
}//End of constructor
```

Activity continued

```
//The CircleArea implement area() of an interface
public double area(){
return Math.PI*rad*rad;
    }//End of income()
}//End of CircleArea class
//Define a class named RectArea that implements Shape
class RectArea implements Shape{
//Define its own private instance variables
private double width;
private double height;
```

Activity continued

```
//Parameterized Constructor
public RectArea(double w, double h){
width=w;
height=h;
}//End of Constructor
//The RectArea implement area() of an interface
public double area(){
return (width *height);
    } //End of income()
}//End of RectArea classs
```

Activity continued

```
//Define a class TirgArea to implement Shape interface
class TirgArea implements Shape{
//Define its own private instance variables
private double width;
private double height;
//Define parameterized constructor
public TirgArea(double w, double h){
width=w;
height=h;
}//End of constructor
```

Activity continued

```
//The TrigArea class implement area() of an interface
public double area(){
return (0.5*base*height);
    }//End of area()
}//End of TirgArea class
class ImplementInterface {
public static void main(String args[]){
```

//Declare reference variable of an interface by the name s

```
Shape s;
```

//Reference variable s refers to objects of CircArea

```
s= new CircleArea(5);
```

Activity continued

```
//Call area() in CircleArea and display output
System.out.println("Area of a circle is "+s.area());
//Reference variable s refers to objects of RecArea class
s=new RectArea(3,4);
System.out.println("Area of a Rect is "+s.area());
//Reference variable s refers to objects of TirArea class
s= new TirgArea(3,4);
//Call area() in TirArea and display output
System.out.println("Area of Right Triangleis  is "+s.area());
   }//End of main ()
}//End of class
```

Activity continued

- In the above example, a **reference variable, s** of **interface** Shapes is declared.

- By **upcasting** this reference variable stores an **object of a CircArea class s.area(),** calls area() method of the CircleArea class.
- In the next statement, s stores an object of a RectArea class and s.area() calls the area() method of the RectArea class.
- Similarly the next s.area() calls the area() method of TirgArea class
- This shows that the **version of the method** that will be **called is determined based** on the **object being referred by the reference variable s**.

“Multiple Inheritance” in Java

- In Java **“multiple inheritance "**is not allowed, that is, ***java does not allow a class to extend two or more classes.***
- So every class can have ***only one direct superclass***.
- However, ***interface*** can be used to accomplish many of the same goals as ***multiple inheritance***.

Implementing Multiple interfaces

- While a **class can extend only one other class**, ***it can implement any number of interfaces.***
- In fact, a class can both extend one other class and implement one or more interfaces.

✓ **So, we can have things like :**

```
class subClass extends Superclass implements interface1,
    interface2,……interfaceN{
    //body of the sub class that extends super class and
    //implements and interface
}
```

Implementing Multiple interfaces continued

- Although **interfaces are not classes**, they are very much like **abstract classes**.

➢ That is, **interfaces can never be used for constructing objects**, but can be used as a **basis for making subclasses**.

- When a class **implements an interface**, the **interface is the base of the class.**
- That is, a **class that implements more than one interface has multiple bases**.
- This enables us to able to **upcast to more than one base type**.

Activity

- Write java program to demonstrate a subclass that extends a superclass and implements two interfaces named Staff and Student.
  - Define an interface named Staff and abstract method netIncome()
  - Define another interface named Student having abstract method named cGpa()
- Define a super class named Allstaff with instance variable of salary, parameterized constructor and two instance method named tax() and pension() as follows:

Activity------

```
double tax(){
          return((salary*.35)-1500)
          }
double pension()
          return(salary*.07)
          }
```

- Define a subclass named PgStudent that extends AllStaff superclass and implements Staff and Student interface and the subclass implements the abstract method of both interfaces .

Activity------

- Use the following information to define a sub class
  - private instance variables named totalcredit and totalpoint
  - Parameterized constructor and use super() to call the superclass constructor
  - Implement the abstract method netIncome() of Staff interface and return values to the caller as follows

    ```
    return (salary-tax()-pension());
    ```
  - Implement the abstract method cGpa() of Student interface and return values to the caller as follows

    ```
    return(totalPoint/totalCredit);
    ```

Activity------

- Define another class to create objects of super and subclass and to declare reference variables of an interface
- Declare, create and initialize objects of a subclass that extends a superclass and implements Staff and Student interface.
- Declare reference variables of Staff and its reference variable of Staff interface refers to objects of a subclass and call netIncome() through reference variable
- Declare reference variable of Student interface, and reference variable of Student interface refers to objects of a subclass and call cGpa () through reference variable of a subclass

Activity------

```
//Define an interface named Staff
interface Staff{
//Define abstract method named netIncome()
    double netIncome();
}//End of netIncome()
//Define an interface named Student
interface Student{
//Define abstract method named cGpa()
    double cGpa();
}//End of cGpa()
//Define a superclass named AllStaff
```

Activity------

```
class AllStaff{
//Define protected variable named salary
protected double salary;
//Define parameterized constructor
public AllStaff(double s){
    salary=s;
    }//End of constructor
//Define a method named tax()
public double tax(){
    return((salary*.35)-1500)
    }//End of tax()
```

Activity------

```
//Define a method named pension ()
public double pension(){
    return (salary*0.07);
    }//End of pension ()
}//End of AllStaff superclass
/*Define a subclass named PgStudent. This class extends AllStaff
    superclass and implements the Staff and Student interface*/
class PgStudent extends AllStaff implements Staff, Student{
//Define private instance variables in this subclass
    private double totalCredit;
    private double totalPoint;
```

Activity------

```
//Define parameterized constructor
public  PgStudent(int credit, int point, double s){
//use super () to call superclass constructor
super(s);
totalCredit=credit;
totalPoint= point;
}//End of constructor
/*The subclass implements the abstract method of Staff interface
   named netIncome() and this method calculate the net income and
   return a value to the caller */
public double netIncome(){
```

Activity------

```
return (salary-tax()-pension());
    }//End of netIncome()
/*The subclass implements the abstract method of Student interface
    named cGpa() and this method calculate the cumulative gpa and
    return a value to the caller */
public double cGpa(){
return (totalPoint/totalCredit);
    }//End of cGpa()
}//End of a subclass
//Define another class named MultipleInheritance to create objects
    of  a subclass and declare reference variable of an interface*/
```

Activity------

```
//Define another class to create objects and reference variables
class MultipleInterfaceExample {
public static void main(String args[]){
//Declare, create and initialize objects of a subclass
PgStudent pgStudent = new PgStudent(12, 40, 5000);
/*Declare reference variable of Staff interface named staff and this
   variable refers to objects of a subclass*/
Staff staff=pgStudent;
/*Declare reference variable of Student interface named stdent and
   this variable refers to objects of a subclass*/
Student stdent= pgStudent;
```

Activity------

```
//Call tax() through objects of a subclass
System.out.println("Income Tax=" +pgStudent.tax());
//Call pension() through objects of a subclass
System.out.println("Pension of All staff="+pgStudent.pension());
//Call netIncome() through staff reference variable
System.out.println("Income of Staff is:" +staff.netIncome());
//Call cGpa() through stdent reference variable
System.out.println("CGPA of the student is:"+stdent.cGpa());
}//End of main ()
}//End of class
```

Extending an interface with Inheritance

- You can easily add new **method declarations** to an **interface by using inheritance**, and you can also **combine several interfaces into a new interface with inheritance**. In both cases you get a new interface.
- To **extend interfaces we use the following general form:**

```
interface Interface2 extends Interface1{
    //body of Interface2
}
```

- When a **class implements an interface that inherits another interface**, it must provide **implementations for all methods defined within the interface inheritance chain**.

Implementing an interface with a class

- To **implement an interfaces,** **we use the following general form:**

```
class class_Name implements interface1, interface2{
      //body of a class
}
```

- When a **class implements an interface that inherits another interface**, it must provide **implementations for all methods defined within the interface inheritance chain**.

Activity

- Write java program a class implements an interface that extends another interface based on the following information:
  - Define an interface named A and two abstract methods named m1() and m2() respectively.
  - Define another interface named B that extends the interface A and its own abstract method named m3(). This interface includes the abstract method of interface A.
  - Define a class named MyClass that implements interface A and B.
  - Define another class to create objects of a class that implements interface A and B.

Activity------

```
//Define an interface named A
interface A {
//Define abstract method named m1() and m2()
public void m1();
public void m2();
}//End of interface A
//Define another interface named B that extends another interface A
interface B extends A {
//Define abstract method named m3()
public void m3();
}//End of interface B
```

Activity------

```
/*Define a class named MyClass that implements
implements interface A and B */
class MyClass implements B {
//Implement m1()
public void m1() {
System.out.println("Implement m1() of interface A.");
}//End of m1()
//Implement m1() of interface A
public void m2() {
System.out.println("Implement m2()of interface A.");
}//End of m2()
```

Activity------

```
//Implement m3() of interface B
public void m3() {
System.out.println("Implement m3() of interface B.");
    }//End of m3()
}//End of MyClass
/*Define another class to create objects of a class
that implements interface A and B */
class ExtendingInterface{
//main ()
public static void main(String arg[]) {
//Declare and create objects of a class named ob
```

Activity------

```
MyClass ob = new MyClass();
//Call m1(), m2() and m3() through objects of ob
ob.m1();
ob.m2();
ob.m3();
    }//End of main()
}//End of a class
```

- In the above example, **interface B extends interface A to have a new interface with inheritance**.
- Similarly, **interface B adds new method declaration m3() in addition to m1() and m2() of interface A**.

Activity------

- As an **experiment you might want to try removing** the implementation for **m1( ) in MyClass**.
  - This will cause a **compile-time error**.
  - As stated earlier, any **class that implements an interface must implement all methods defined by that interface**, including any that are **inherited from other interfaces**.

Introducing Nested Classes and Inner Classes

- It is possible to **define** a **class within another class**; such classes are known as **nested classes**.
- The **scope** of a **nested class** is bounded by the **scope** of its **enclosing class.**
- Thus, **if class B is defined within class A, then B is known to A, but not outside of A.**

➢ A **nested class** has access to the **members,** including **private members**, of the class in which it is nested.

- However, the **enclosing class does not have access** to the **members** of the **nested class.**

➢ There are **two types of nested classes**: **static and non-static.**

- A **static nested class** is one which has the **static modifier** applied.

Introducing Nested Classes and Inner Classes-----

- Because it is **static**, it must **access** the **members** of its **enclosing class** through an **object**.
- That is, it cannot **refer** to **members** of its **enclosing class directly.**
- Because of this restriction, static nested classes are **seldom used**.

➢ **Nested class allows to**:

- **Group classes that logically belong together**
- **Control the visibility of one class within the other**.

➢ **Nested classes** can be either **member** or **local class**.

- **Member classes** can be either **static** or **non-static**.
- **Local classes** can be either **named** or **anonymous**.

1. Static Nested Classes

- Static nested classes are declared by preceding the class declaration with the **static modifier**.
  - A **static nested class** acts just like any top-level class except that **its name and accessibility** are defined by its **enclosing type.**
  - The **name of a nested type is expressed** as **EnclosingName.NestedName**.
  - It access the **members of its enclosing class through an object**.
  - That is, it **cannot refer to members** of its **enclosing class directly.**
- If you don't need a connection between the nested class object and the outer class object, then you can make the nested class **static**.

1. Static Nested Classes----

- The object of an **ordinary inner class implicitly** keeps a **reference to the object of the enclosing class** that created it.
- This is not true, however, when you say an inner class is static.
- **A nested class means:**

1. You don't need an **outer-class object** in order to create an **object of a nested class**.
2. You can't access a ***non-static outer-class object*** from an ***object of a nested class***.

- Nested classes are different from ordinary inner classes in another way, as well.

- [illegible]

Example 4

```
abstract class Staff{
        protected String name;
        protected float salary;
        public Staff(String s, float f){
                name=s;
                salary=f;
        }
        abstract public float netIncome();
}
class Outer{
        public static class AcademicStaff extends Staff{
                private float allowance;
                public AcademicStaff(String s, float a, float b){
                        super(s,a);
                        allowance=a;
                }
        public float netIncome(){
                return (salary*0.8f)+allowance;
        }
}
public static class AdminStaff extends Staff{
        public AdminStaff(String s, float a){
                super(s,a);
        }
```

```
            public float netIncome(){
                        return salary * 0.8f;
            }
}
public static AcademicStaff acadStaff(String s, float f1, float f2){
            return new AcademicStaff(s, f1, f2);
            }
public static AdminStaff adminStaff(String s, float a){
            return new AdminStaff(s,a);
            }
}
class Test{
     public static void main(String args[]){
            Staff s1=Outer.acadStaff(“Nigussie”, 5000, 400);
            System.out.println(“Income of “+s1.name’+”is”+s1.netIncome());
             Staff s2=Outer.adminStaff(“Zelalem”, 5000);
             System.out.println(“Income of “+s2.name’+”is”+s2.netIncome());
            Outer.AdminStaff as=new Outer.AdminStaff(“Getachew”, 690);
             System.out.println(“Income of “+as.name’+”is”+as.netIncome());
            }
}
```

- When you observe the previous example, in main(), ***no object of Outer class is necessary***; instead, you use the normal syntax for selecting a **static** member to call the methods that return references to **AcademicStaff** and **AdminStaff** classes.
- An ***object non-static inner*** class should be linked to the ***Outer class object***.
- But here all the three objects have no connection with Outer class object.
- The full name of AdminStaff class is Outer.AdminStaff.
- This full name clearly indicates that the ***class exists as part of the Outer class***, not as a stand-alone type.
- Code outside the Outer class must use this full name.

- Once AdminStaff is a ***member of Outer***, the AdminStaff class can ***access all other members of Outer***, including all **inherited members**.
- In this sense the **nested class** is seen as part of the ***implementation of the enclosing class*** and so is completely trusted.
- There is no restriction on how a static nested class can be extended.
- It can be extended by any class to which it is accessible.
- Of course, the extended class does not inherit the privileged access that the nested class has to the enclosing class.

Inner Classes

- **Non-static nested classes** are called **inner classes.**
- An inner class has access to all of the **variables and methods of its outer class** and refers to them directly in the same way that other non-static members of the outer class do.
- Thus, an **inner class** is fully within the ***scope of its enclosing class***.
- An instance of an inner class is associated with an instance of its enclosing class.
- You often need to closely tie a nested class object to a particular object of the enclosing class.
- In general ***inner class objects can be created outside the enclosing class as:***

OuterClassName.InnerClassName=new OuterClassName().new InnerClassName()

Or

OuterClassName.InnerClassName=OuterClassObject.new InnerClassName()

- Consider the following example.

Example 5

```
class Bankaccount{
        private long number; //account number
        private long balance; // current balance
        private Action lastAct;//last action performed
        public BankAccount(long n, long b){
                number = n;
                balance = b;
        }
        public class Action{
                private String act;
                private long amount;
                Action(String act, long  amount){
                        this.act=act;
                        this.amount=amount;
                }
        public String tostring(){
        //identify our enclosing account
        return number + “:” +act+” ” + amount;
        }
}
```

```
    public void deposit(long amount){
          balance+=amount;
          lastAct=new Action(‘deposit’, amount);
          }
    public void withdraw(long amount){
          balance-=amount;
          lastAct=new Action(“withdraw”, amount);
          }
    public void action(){
          System.out.println(lastAct.toString());
          }
}
class Test{
    public static void main(String args[]){
          BankAccount bal= new BankAccount(1122, 500){
          bal.deposit(500);
          bal.action();
          bal.withdraw(300);
          bal.action();
    }
}
```

- When you observe the previous example, the class Action records a single action on the account.
- Action class objects are created inside instance methods of the enclosing class, as in **deposit and withdraw.**
- The current object **this** is associated with the inner object by default.
- The creation code in deposit is the same as the more explicit

  lastAct=this.new Action(‘deposit’, amount);

  Any BankAccount object could be substituted for this.

- An inner class declaration is just like a top-level class declaration except for one restriction ***inner classes cannot have static members***(including static nested types), except for ***final static fields that are initialized to constants or expressions*** built up from constants.
- The rationale for allowing constants to be declared in an inner class is the same as that for allowing them in interfaces it can be convenient to define constants within the type that uses them.
- As with top-level classes, ***inner classes can extend any other class including its enclosing class implement any interface and be extended by any other class.***
- ***An inner class can be declared final or abstract.***

A) Local Inner Classes

- You can define inner classes in code blocks, such as ***a method body, constructor, or initialization block***.
- These local inner classes are ***not members of the class of which the code is a part*** but are ***local to that block, just as a local variable is***.
- Such classes are completely **inaccessible outside the block** in which they are defined there is simply no way refer to them.
- But instances of such classes are normal objects that can be **passed as arguments and returned from methods**, and they exist until they are no longer referenced.
- Because ***local inner classes are inaccessible***, they can't have access modifiers, nor can they be declared **static** because, well, they are local inner classes.

- As **local inner classes** can access all the variables that are in scope where the class is defined local ***variables, method parameters, instance variables***(assuming it is a non-static block), and ***static variables.***
- The only restriction is that a ***local variable or method parameter can be accessed*** only if it is ***declared final***.
- **There are two reasons for doing this:**

1. As shown previously, you are implementing an interface of some kind so that you can create and return a reference.
2. You are solving a complicated problem and you want to create a class to aid in your solution, but you don't want it publicly available.

- Consider the following simple example:

Example 6. 1

```
interface 11{
        void m();
}
   class C{
        public11 m2(){
                class Inner implements 11{
                        public void m(){
                System.out.println(“From Local inner Class”);
                }//End of Method m
                return new Inner();
        }
}
class Test{
   public static void main(String args[]){
        11i;
        C o =new C();
        i=o.m2();
   }
}
```

- The Inner class is local to the m2 method; it is not a member of the enclosing class C.
- Because **Inner** is local to the method m2, it cannot be accessible outside of method m2().
- Notice the upcasting that occurs in the return statement-noting comes out of m2() except a referenceto11, the base class. In the main method, m2() is invoked.
- This method returns an object of type 11 which like any other object once it is returned.
- Method m(), the implementation of the abstract method, method of interface 11, m() is invoked through this method.
- Consider the interface Shape as a common “interface” to its derived classes in the following example:

Example 6.2

```
interface Shapes{
            double area();
}
class Outer{
            public Shapes circ(double d){
                        class Circle implements Shapes{
                        private double radius;
                        public Circle(double r){
                                    radius= r;
                        }
                        public double area(){
                        return Math.PI*radius*radius;
                        }
            }//End of a class Circle
            return new Circle(d);
    }
classTest{
    public static void main(String args[]){
            Shapes shape;
            outer obj=new Outer();
            shape=obj.circ(5);
            system.out.println("Area= " +shape.area());
    }
}
```

Anonymous Inner Classes

- Anonymous inner class is an inner class that has ***no name.***
- Anonymous Inner classes ***extend a class or implement an interface.***
- These classes are defined at the same time they are ***instantiated with new***.
- For example, consider the program in 6.1 above, the class Inner is fairly lightweight and is not needed outside the method m2().
- The name Inner doesn't much value to the code what is important is that it is an I1 object.
- The m2() method could use an anonymous inner class instead.
- Consider the following simple example to demonstrate anonymous inner class:

Example 7.1

```
interface 11{
        void m();
}
class C{
        public 11 m2(){
                return new 11(){
                public void m(){
                System.out.println(“From Local Inner Class”);
                }//End of method m
        }
   };
}
class Test{
        public static void main(String args[]){
                11 I;
                C o= new C();
                i=o.m2();
                i.m();
        }
}
```

- **Anonymous classes** are defined in the new expression itself, as part of a statement.
- The type specified to new is the supertype of the anonymous class.
- Because 11 is an interface, the anonymous class in m2()implicitly extends Object and implements 11.
- An anonymous class cannot have an explicit extends for implements clause, nor can it have any modifiers.
- The m2() method combines the creation of the return value with the definition of the class that represents that return value.
- This looks like you are starting out to create an 11 object:

  return new 11()
- But then, before you get to the semicolon, you say, "But wait, I think I will slip in a class definition";

```
return new 11(){
        public void m(){
        System.out.println("From Local Inner Class");
        }//End of Method m
}
```

- This means: "Create an object of an anonymous class that it implements interface 11."
- The reference returned by the new expression is automatically upcast to a 11 reference.
- The anonymous inner-class syntax is shorthand for:

```
   classMy11 implements 11{
        public void m(){
        System.out.println("From Local Inner Class");
        }//End of Method m
}
   return new My11();
```

Example 7.2

- The following code shows what to do if your base class needs a constructor with an argument:

```
abstract class Base{
        int x;
        public Base(int x){
                this.x=x;
        }
        void m();
}
class Outer{
        public Base m2(0{
                return new Base(12){
                public void m(){
                System.out.println(“From Local Inner Class”);
                }//End of method m
        };
}
}
class Test{
        public static void main(String args[]){
                Base b;
                Outer o = new Outer();
                b=o.m2();
                b.m();
        }
}
```

- As you observed the previous example, you simply pass the appropriate argument to the base-class constructor, seen here as the x passed in

  new Base(x);
- The semicolon at the end of the anonymous inner class doesn’t mark the end of the class body.
- Instead, it marks the end of the expression that happens to contain the anonymous class.
- Thus, it is identical to the use of the semicolon everywhere else.

- **Anonymous inner classes** cannot have explicit constructor declared because they have no name to give the constructor.
- If anonymous inner class is complex enough that it needs explicit constructors then it should probably be a local inner class.
- In practice, many anonymous inner classes need **little or no initialization.**
- In either case, an anonymous inner class can have an initializes and initialization blocks that can access the values that would logically have been passed as a constructor argument as shown in the following example:

Example 7.3

```
interface Shapes{
        double area();
}
class Outer{
        public Shapes circ(final double d){
                return new Shapes(){
                private double radius;
                {
                        radius =d;
                }
                public double area(){
                return Math.PI*radius*radius;
                }
        };//End class Circle
   }
}
class Test{
   public static void main(String args[]){
        Shapes shape;
        Outer obj=new Outer();
        shape=obj.circ(5);
        System.out.println("Area ="+shape.area());
   }
}
```

- As shown in the above example, an instance initializer is the constructor for an anonymous inner class.
- Of course, it’s limited; you can’t overload instance initializers, so that you can have only one of these constructors.