UNIT 1 INHERITANCE

Structure **Page no.**

1.0 INTRODUCTION

In the last unit of the previous block, you learned about class and objects. This unit takes you further in learning java programming and will explain one important concept of object-oriented programming, known as inheritance. The inheritance can be known as one of the essential tools in object-oriented programming language because it enables the reusability of properties of other classes. It takes advantage of the similarities that exist between various classes. It helps in data generalization and code reduction. If we look at the history of Inheritance, it was invented in 1969 for Simula (a programming language) and is now being used throughout many object-oriented programming languages such as Java, C++ , C# , and Python, etc.

1.1 OBJECTIVES

After going through this unit, you will be able to:

- Explain basic concepts of inheritance,
- Write programs using the constructors along with inheritance,
- Apply access specifier concepts in hierarchical inheritance, and
- Use inheritance in solving complex programming problems.

1.2 BASICS OF INHERITANCE

The basic idea behind Inheritance in Java programming is to enable the programmer to create multiple new classes that are built upon existing classes. When these new classes inherit the existing classes, then you can reuse the methods and fields of the parent class. Moreover, as per requirement, you can add the new methods/functionalities and attributes to your new class(inherited class). Inheritance represents the IS-A relationship which is also known as the parent-child relationship.

The syntax of Java Inheritance

```
class child_class extends parent_class
        {
            //fields
            //methods
        }
```

In the syntax, you can see the child_class is created using the keyword class, and it inherits the properties of the parent class with the help of the *extends keyword*. And other additional methods and fields also can be created as shown in the syntax.
In general, the child class is known as sub-class, and the parent class is known as super-class.

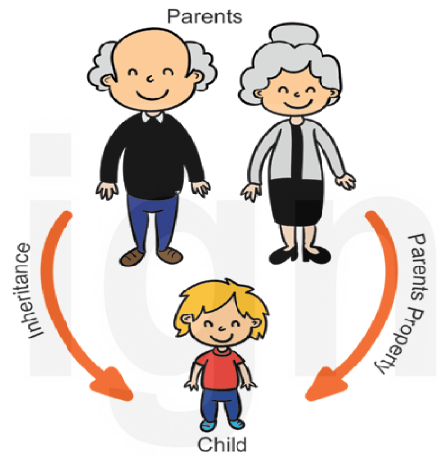


Figure 1: Inheritance- parents to child

Figure 1, shows a real-life example in which the child inherits the properties from their parents. A child will have the feature of its parents as well as its own features. In programming context, you may understand it in a way where code written in the parent class is available to the child class, and there is no need to write that code again. This reusability of code not only minimise the effort in applications development but also save a lot of time and resources.

The objective behind Inheritance in Java is to create new classes built upon existing classes, i.e. new classes will be able to use the attributes and methods of already exiting classes. Inheritance facilitates reuse of the methods and fields of the parent class. Moreover, you can also create new methods and fields in the new class as per requirement.

It also facilitates the programmer to achieve *run time polymorphism* and code reusability. This indicates that there is no need to write the same code again and again while using it in other classes. About polymorphism, you will learn in the next unit of this block.

For showing the Inheritance, there must be at least two classes : The class which inherits the methods and data members/fields from parent class is known as a subclass(child class) of the class from which it inherits, and the class from which subclass is inherited is known as a base class or parent class.

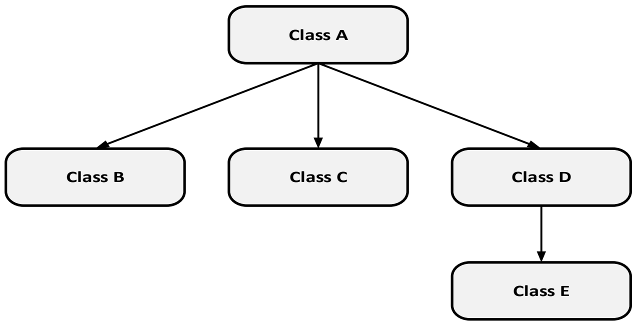


Figure 2: Class hierarchy

In figure 2, class B, class C, and class D (which are known as “sibling classes”) inherit the properties of class A. If a class, for example, class B is a subclass of class A then we say that class A is the superclass of class B. Sometimes, we also use the term derived class and base class instead of subclass and superclass respectively.

Inheritance can be carried to several generations of the classes, as shown in figure 2. Class E is a subclass of class D, which is also a subclass of class A. In this case, we can also consider like class E is a subclass of class A, even though it is not a direct subclass. This whole phenomenon of classes representation is known as a class hierarchy.

The fundamental idea behind Inheritance in Java is to enhance code reusability by inheriting the methods and fields of parent classes. In figure 3, it is shown that the Electronics class is the parent class for the child classes “Phone” and “Sound System” which indicates that these two classes will have the properties of Electronics class.

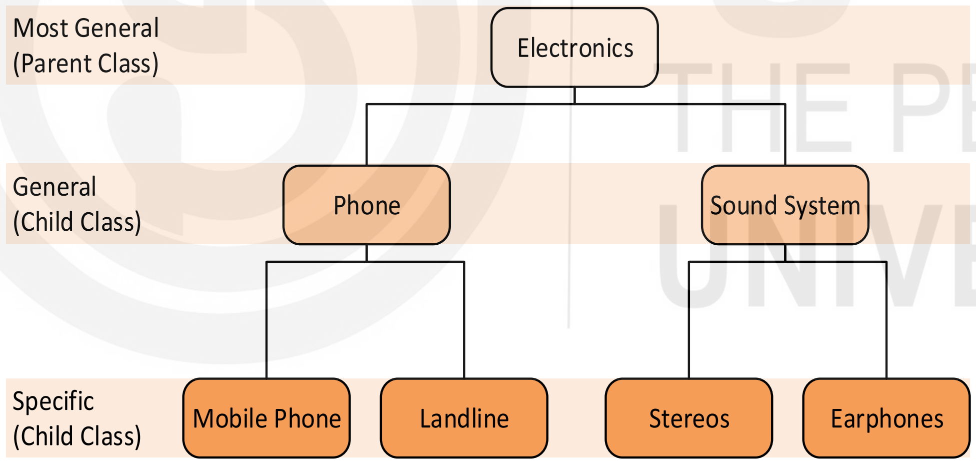


Figure 3: Hierarchy and generality of classes

Similarly, Mobile Phone and Landline classes are deriving the properties of the “Phone” class, which is treated as a parent class. And Stereos and Earphones are treated as a child class of the class “Sound System”. This class hierarchy is moving from general to specific. In figure 3, Electronics is the most general class while the classes at the bottom are the most specific.

Let us discuss the programming aspects of Inheritance of Java with the help of a simple example in which “Vehicle” is a parent class, and “four-wheeler” is a child class. The child class inherits the properties of the parent class with the help of the **extends** keyword.

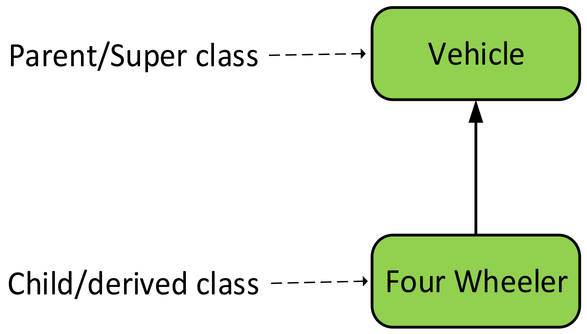
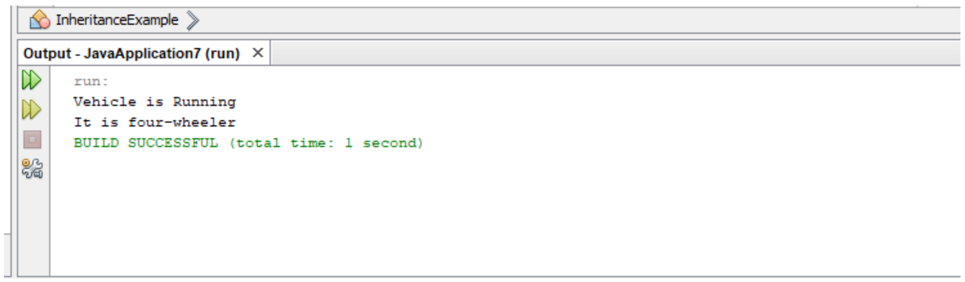


Figure 4: Super and Sub-class relation

Programming Example 1:

```
class Vehicle
{
  /* Vehicle is a parent class */
  void run()
  {
    System.out.println("Vehicle is Running");
  }
}

class FourWheeler extends Vehicle
{
  /* FourWheeler is a child class */
  void wheel()
  {
    System.out.println("It is four-wheeler");
  }
}

class InheritancesExamples
{
  public static void main(String args[])
  {
    FourWheeler four_wheeler = new FourWheeler ();  /*creating object*/
    four_wheeler.run();
    four_wheeler.wheel();
  }
}
```

Output:



```
run:
Vehicle is Running
It is four-wheeler
BUILD SUCCESSFUL (total time: 1 second)
```

In example 1, class FourWheeler (child class) inherits the properties of the class Vehicle (parent class) with the help of the **extends** keyword. in the example, run() is a method of the superclass, and wheel() is the method of FourWheeler class. Now you can see the object “four_wheeler” which is an object of the child class, can inherit its parent class vehicle's method run().

1.3 ADVANTAGES OF INHERITANCE

One of the biggest advantages of Inheritance is that the code already mentioned in the parent class does not need to be rewritten in the child class. Because child class may use the code without rewriting it.

Another advantage of Inheritance is like it can save time and effort because of code reusability. It also minimizes the development and maintenance costs because of the clear model structure, which is quite easy to understand. It is also possible to keep some data private in base class, which derived classes cannot use.

In addition to the points mentioned above, you may consider the whole application development in context of - Inheritance represents the *IS-A relationship*, also known as the parent-child relationship. When you are designing a system, you may consider some features that will be inherited/accessed from the parent's classes, and some features which need some special implementation may directly be implemented in a derived class. More details about the implementation of this philosophy will be discussed in the later section of this unit and also in the next unit of this block.

1.4 MEMBER ACCESS AND INHERITANCE

Access modifiers are simply keywords in Java programming that provides accessibility of a class and its members. In Java there are four types of access specifiers that is public, default, protected and private. These access specifiers are used to control what parts of a program can be accessed by whom. Well defined set of permissions can prevent unauthorized access and misuse of data.

Out of these access modifiers, protected is accessible within package and outside the package but with the help of implementing Inheritance feature only. While public and private are like, when a member of a class is declared private, it can be accessed by other members of the class only. The public access specifier makes data members and member functions accessible outside of the class. That is why the public modifier has always preceded the main( ) method because it is called by code that is outside the program.

In case when no access modifier is used, then a member of the class will be public within its own package but cannot be accessed outside of its own package. Examples for declaration using access specifiers are given below.

public int numb;

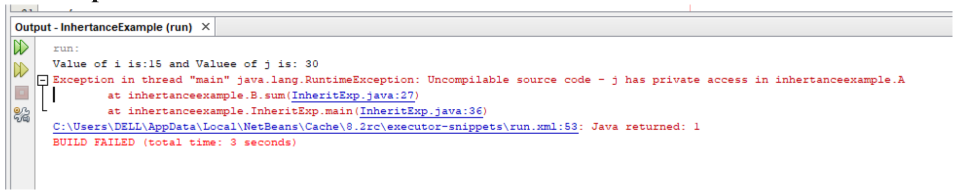
private int age;

private int mymethod (int a , char b) {…………….}

Further, if we look at in terms of Inheritance, a subclass includes all of the members of its superclass, but it cannot access the superclass members that are declared as private. Let have a look at the below given example.

```
/*--In a class hierarchy, private members remain private to their class*/

// It is a superclass
class A
{
    int i;
    private int j;
    void setij(int x, int y)
   {
        i = x;
        j = y;
       System.out.println("Value of i is:"+i+" and Valuee of j is: "+j);
   }
}
/*---A's j is not accessible here in this class------*/
class B extends A
{
    int total;
    void sum()
   {
        total = i + j;   // Error, j is not accessible here as  it is declared private
   }
}
class Main
{
   public static void main (String[] args)
   {
        B Bobj = new B();
        Bobj.setij(15, 30);
        Bobj.sum();
        System.out.println("Total is:" +Bobj.total);
    }
}
```

The Output:



```
run:
Value of i is:15 and Valuee of j is: 30
Exception in thread "main" java.lang.RuntimeException: Uncompilable source code - j has private access in inhertanceexample.A
        at inhertanceexample.B.sum(InheritExp.java:27)
        at inhertanceexample.InheritExp.main(InheritExp.java:36)
C:\Users\DELL\AppData\Local\NetBeans\Cache\8.2rc\executor-snippets\run.xml:53: Java returned: 1
BUILD FAILED (total time: 3 seconds)
```

This program will not compile because access of j inside the sum( ) method of class B causes an access violation. Since j is declared as private, it is only accessible to other members of its own class only.

1.5 TYPES OF ACCESS SPECIFIERS

If you have declared anything public, then it can be accessed in any other class and package, but it is declared as private, then it can be accessed in the defined class only.

Whenever a member does not have an explicit access specification, it can be accessed in a subclass as well as in other classes of the same package. Protected members are the members which can be seen in the direct subclass of other packages. A summary of various access specifiers is shown in table 1.

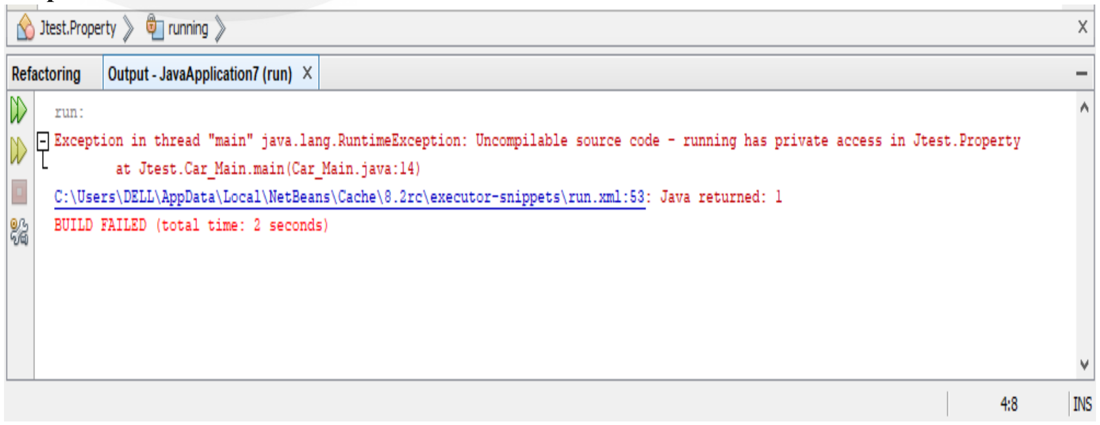
Table 1 Summary of Access Specifier

	High restriction ----→ Low restriction			
	Private	Default	Protected	Public
Inside Class	YES	YES	YES	YES
Inside Package	NO	YES	YES	YES
Outside Package subclass	NO	NO	YES	YES
Outside Package	NO	NO	NO	YES

Let's have a look at the example for private access specifier; in this example, you can see that there are two classes "Property" and "Car_Main" inside the package Jtest. In the Property class, private member running is defined, and we are trying to use this method in another class, Car_Main.

```
package Jtest;
public class Property
{
  private String running;
}

package Jtest;

public class Car_Main
{
  public static void main(String[] args)
 {
    Property obj = new Property();
    obj.running="I is a fast running car";
    System.out.println("Property of the car is: "+obj.running);
  }
}
```

Output:



```
run:
Exception in thread "main" java.lang.RuntimeException: Uncompilable source code - running has private access in Jtest.Property
        at Jtest.Car_Main.main(Car_Main.java:14)
C:\Users\DELL\AppData\Local\NetBeans\Cache\8.2rc\executor-snippets\run.xml:53: Java returned: 1
BUILD FAILED (total time: 2 seconds)
```

The program's output shows that private member "running" can not be accessed in another class Car_Main in the same package. It concludes as the private member is defined.

Now, if you want to access the private member of the class into another class of the same package, then it can be accessed with the help of the public method of the same class in which the private member is defined.

For example, Setproperty and Getproperty are these two public methods; with the help of these two methods, private members can be accessed into Car_Main class which is a different class.

```
package Jtest;
public class Property
{
  private String running;

  public void Setproperty(String running)
  {
    this.running=running;
  }
  public String Getproperty()
  {
    return this.running;
  }
}

package Jtest;

public class Car_Main
{
  public static void main(String[] args)
  {
    Property obj = new Property();
    obj.Setproperty("It is a fast running Car");
    System.out.println("Property of the car is: "+obj.Getproperty());
  }
}
```

The output of the program:

```
Property of the car is: It is a fast running Car
```

In this example, you can see that Setproperty method is used to set the value of the private member in the same class and then, with the help of Getproperty method; it is accessed in the Car_Main class. And you can see that the program executes successfully and shows the outcome.

The example given below explains the concept of protected and public access specifiers.

```
package Jtest;

public class Animal
{
  public int legcount;
  protected void Display()
  {
    System.out.println("I am a protected Animal");
  }
  public void Display2()
  {
    System.out.println("I am a Public Animal");
    System.out.println("I have "+legcount+" legs");
  }
}

package Jtest;
public class PetDog extends Animal
{
  public static void main(String[] args)
  {
    PetDog pd = new PetDog();
    pd.Display();
  }
}
```

Output of the program:

```
I am a protected Animal
```

Another example program for further clarification is given below:

```
package Jtest;

public class Main
{
  public static void main(String[] args)
  {
    Animal ani = new Animal();
    ani.legcount=4;
    ani.Display2();
  }
}
```

Output of the program:

```
I am a Public Animal
I have 4 legs
```

CHECK YOUR PROGRESS-1

Q1. Ram has written the code given below, but it is showing compile-time error. Can you find out the mistake he has made?

```
class Ram
{
   //Class Ram Members
}

class Shyam
{
   //Class Shyam Members
}

class Reet extends Ram, Shyam
{
   //Class Reeta Members
}
```

Q2. Find out the output of the program given below.

```
class A
{
   int methodA(int i)
   {
      i /= 10;

      return i;
   }
}

class B extends A
{
   int methodB(int i)
   {
      i *= 20;

      return methodA(i);
   }
}

public class MainClass
{
   public static void main(String[] args)
   {
      B b = new B();

      System.out.println(b.methodB(100));
   }
}
```

Q3. Why you can not instantiate the Class Ram in the below code outside the package even though it has public constructor?

```
package package1;

class Ram
{
   public Ram()
```

```
    {
       //public constructor
    }
}

package package2;

import package1.*;

class Ravi
{
   Ram a = new Ram();
}
```

....

....

....

....

....

Q4. Is the code given below written correctly? If yes then what will its output?

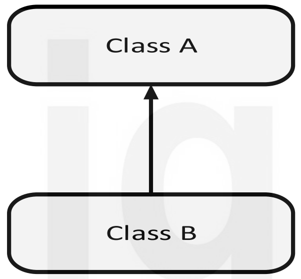
```
package pack1;

class A
{
   protected static String s = "A";
}

class B extends A
{

}

class C extends B
{
   static void methodOfC()
   {
      System.out.println(s);
   }
}

public class MainClass
{
   public static void main(String[] args)
   {
      C.methodOfC();
   }
}
```

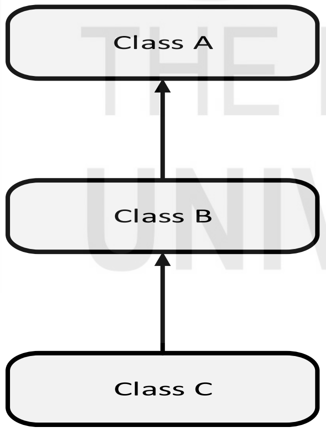
1.6 TYPES OF INHERITANCE

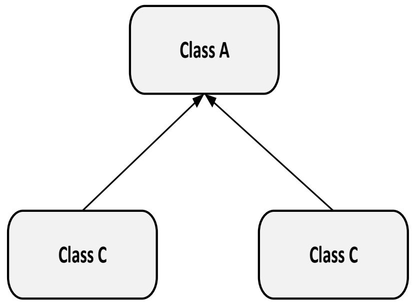
Java supports many types of Inheritance. The table given below give a summarised presentation of the inheritance supported by Java.

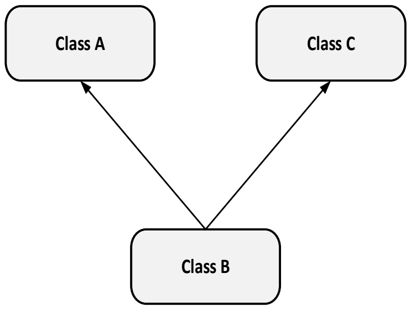
Table 2 Types of Inheritance

Name of Inheritance	**Block Diagram**	**Syntax**

Single Inheritance		public class A
		{
		-------- 	
		}
		public class B **extends** A
		{
		--------- 	
		}

Multi-Level Inheritance		public class A
		{
		-------- 	
		}
		public class B **extends** A
		{
		--------- 	
		}
		public class C **extends** B
		{
		--------- 	
		}

Hierarchical Inheritance		public class A{
		-------- 	
		}
		public class B **extends** A{
		--------- 	
		}
		public class C **extends** A{
		--------- 	
		}

Multiple Inheritance		public class A
		{
		-------- 	
		}
		public class B
		{
		--------- 	
		}
		public class C **extends** A, B
		{
		--------- 	
		}//**Java does not support multiple inheritances**

Java supports single inheritance, hierarchical inheritance and multi level inheritance. But multiple inheritance is not supported by java. Java provides the feature known as Interface, which fills the gap of non supporting of multiple inheritances. You will learn about Interface in unit 4 of this block.

1.7 USE OF SUPER KEYWORDS

Java defines a special variable named "super" . This keyword is used to refer to the objects of the immediate parent class. In general, super refers to the object that contains the method. It forgets about the objects of the class in which you are writing, and it remembers about the objects that belong to the superclass in that class in which you are writing.

The **super** keyword does not know about the functionalities of methods and variables of the superclass, it can only be used to call to methods and variables in the superclass. The **super** has two general forms/uses, the first one is to call the superclass constructor, and the second is to access a member of the superclass that a member of a subclass has hidden.

1.7.1 Calling the superclass constructor

A subclass can call a constructor defined by its superclass by using the following syntax of super:

super(arg-list); here arg-list indicates the arguments required by the superclass constructor. To see the use of super(), consider the following example given below. In the given example BoxWeight class inherits the properties of Box class.

```
/* --------------Definition of the Box class-----------------*/
class Box
{
   private double width;
   private double height;
   private double depth;

   /*--------------constructor with all the specified dimensions-----------*/
   Box(double w, double h, double d)
    {
      width = w;
```

```
            height = h;
            depth = d;
        }
/*--------------constructor when dimensions are not specified------------ */
         Box()
         {
            width = -1;
            height = -1;
            depth = -1;
        }
/*--------------constructor when box is in the form of cube. ---------------*/
         Box (double len)
         {
            width=height=depth=len;
        }
/*-------------method to compute the volume--------------------------------*/
        double Volume()
         {
            return width*height*depth;
        }
}
class BoxWeight extends Box
{
        double weight;
/*-------------constructor with all the specified parameters---------------*/
        BoxWeight (double w, double h, double d, double m)
         {
            super(w, h, d); /*--------call superclass constructor---------------- */
            weight=m;
        }
        BoxWeight ()
         {
            super( );
            weight=-1;
        }
        BoxWeight (double len, double m)
         {
            super(len);
            weight = m;
        }
}
/*----------------the main class implementation--------------------------*/
public class SuperDemo
{
        public static void main(String args[ ])
            {
            BoxWeight box1 = new BoxWeight (20, 30, 25, 54.5);
            BoxWeight box2 = new BoxWeight ();
            BoxWeight cube = new BoxWeight (5, 7);
            double vol;
            vol = box1.Volume();
            System.out.println ("Volume of the box1 is: "+ vol);
            System.out.println ("Weight of the box1 is:"+box1.weight);
        System.out.println ();

         vol = box2.Volume();
        System.out.println ("Volume of the box2 is:"+ vol);
        System.out.println ("Weight of the box2 is:"+ box2.weight);
        System.out.println ();

         vol = cube.Volume();
        System.out.println ("Volume of the cube is: "+vol);
        System.out.println ("Weight of the cube is: "+cube.weight);
        System.out.println ();
    }
}
```

Output of theprogram:

```
Volume of the box1 is: 15000.0
Weight of the box1 is: 54.5

Volume of the box2 is: -1.0
Weight of the box2 is: -1.0

Volume of the cube is: 125.0
Weight of the cube is: 7.0
```

Explanation:
In the child class BoxWeight, super call the constructor of the parent class as shown below.
super(w, h, d)-------------> Box(double w, double h, double d)
super( )---------------------->Box( )
super(len)------------------>Box (double len)
When, we review the key concept behind super( ) is like when a subclass calls super ( ), it calls the constructor of the immediate superclass. The super( ) always refers to the superclass just above the calling class. This concept is applicable even in a multileveled hierarchy also.

1.7.2 To access the member of the superclass

Another use of **super** keyword is similar to **this,** except that it always refers to the superclass of the subclass in which it is used. The general syntax to access the members of the superclass is as follows.

super.*member*

here, *member* can be either a method or an instance variable. Let's have a look on superclass Animal and child class Dog, in Dog class, the sound method calls the sound( ) method of the superclass with the help of super keyword.

```
class Animal
{
   public void sound()
    {
       System.out.println("The Animal makes sound");
   }
}
```

```
class Dog extends Animal
{
   public void sound()
    {
      super.sound();
      System.out.println("The dog barks: bow wow");
   }
}
public class Mainclass
{
   public static void main(String args[])
    {
      Animal mydog = new Dog();
      mydog.sound();
   }
}
```

Output of the program is as follows.

```
The Animal makes sound
The dog barks: bow wow
```

The above example was based on to access the superclass method, now let's have an example based on to access the instance variable of the superclass.

```
/*----------Using super to overcome the name hiding--------*/
class A
{
   int i;
}
/*--------Create a subclass by extending class A-------------*/
class B extends A
{
  int i;  // this i hides the i in defined class A
  B(int a, int b)
  {
     super.i=a;  // i defined in class A
     i=b;        // i defined in class B
  }
  void show()
  {
     System.out.println("i is superclass: "+super.i);
     System.out.println("i in subclass: "+i);
  }
}
class Main
{
  public static void main(String args[ ])
  {
     B subobj = new B (5, 10);
     subobj.show( );
  }
}
```

Output of the program:

```
i in superclass: 5
i in subclass: 10
```

So, you can see the instance variable i in B hides the variable i defined in class A, super keyword allows access to the variable i defined in the superclass A. Also, you have already seen that super can be used to invoke the methods hidden by subclass.

1.8 CREATING MULTI-LEVEL CLASS HIERARCHY

Till now, we have seen a few simple examples of class hierarchies that consist of using a superclass and a subclass only. However, you can build hierarchies that can contain as many layers as you want. For example, given three classes Student, Marks, and Percentage, as shown in figure 4. Percentage can be a subclass of Marks, which is further a subclass of Student.

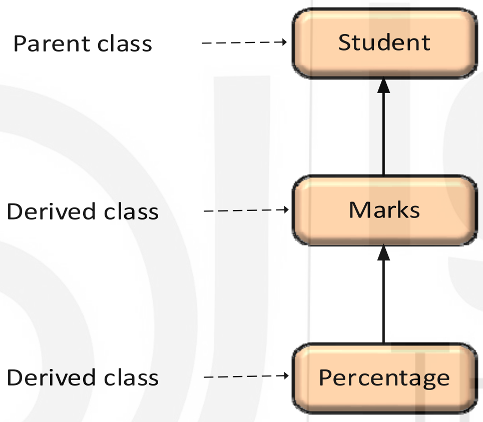


Figure 5 Multi-level class hierarchy

In multi-level hierarchy, each subclass inherits all of the properties found in its superclass.

```
class Student
{
  int rollnumb;
  String name;
  /*r and n are used to set the roll number and name*/
  Student (int r, String n)
  {
    rollnumb = r;
    name = n;
  }
  void show()
  {
    System.out.println ("Student Roll Number is: "+rollnumb);
    System.out.println ("Student Name is: "+name);
  }
}
class Marks extends Student
```

```
{
   int s1, s2, s3, s4, s5, sum;
   Marks(int r, String n, int m1, int m2, int m3, int m4, int m5)
   {
      super(r,n);
      s1 = m1;
      s2 = m2;
      s3 = m3;
      s4 = m4;
      s5 = m5;
   }
   void showmarks()
   {
      show();
      sum=s1 + s2 + s3 + s4 + s5;
      System.out.println ("Total marks: "+sum);
   }
}
class Percentage extends Marks
{
   float percent;
   Percentage (int r, String n, int m1, int m2, int m3, int m4, int m5, float p)
   {
      super(r, n, m1, m2, m3, m4, m5);
      percent = p;
   }
   void showpercent( )
   {
      showmarks();
      percent = sum/5;
      System.out.println ("Percentage: "+percent+"%");
   }
}
class Main
{
   public static void main(String args[ ])
   {
      Percentage p = new Percentage (1025, "Sunil", 90, 85, 80, 88, 75, 0);
      p.showpercent ( );
   }
}
```

Output of the program:

```
Student Roll number is: 1025
Student Name: Sunil
Total marks: 418
Percentage: 83.0%
```

In the above program, the three classes Student, Marks, and Percentage forming a multi-level hierarchy. The Student class serves as the parent class for the derived class Marks, which in turn serves as a parent class for the derived class Percentage.

CHECK YOUR PROGRESS-2

Q1. What will be the output of the program given below?

```
class A
{
   {
      System.out.println("You are in class A.");
   }
}

class B extends A
{
   {
      System.out.println("You are in class B");
   }
}

class C extends B
{
   {
      System.out.println("Here, Now in class C");
   }
}

public class MainClass
{
   public static void main(String[] args)
   {
      C c = new C();
   }
}
```

....

....

....

....

Q2. What will be the output of the program given below?

```
public class mainclass
{
   public static void main(String s[])
   {
      A a = new A();
      a.i = 21;
      B b = new B();
      b.i = 32; // LINE X
      b.j = 25;
      printI(a);
      printI(b); // LINE Y
      printJ(b);
   }
```

```
    public static void printI(A a1)
    {
        System.out.println(a1.i);
    }

    public static void printJ(B b1)
    {
        System.out.println(b1.j);
    }
}

class A
{
    int i;
}

class B extends A
{
    int j;
}
```

....

....

....

....

Q3. What do you mean by Generalized and Specialized classes in Java? Explain it with the help of a program.

....

....

....

....

1.9 DEMONSTRATING ORDER OF CONSTRUCTORS EXECUTION

Constructors in Java are similar to methods that are invoked while creating an object in the class. In general, the order of constructor execution depends on the order in which the class hierarchy of the classes is created. In a class hierarchy, constructors complete their execution in order of derivation, from superclass to subclass. For example, given a subclass Parents and a superclass Grandparents, Grandparents constructor executed before the Parents.

```
class Grandparents
{
   Grandparents()
   {
      System.out.println("Inside Grandparents constructor");
   }
}
class Parents extends Grandparents
{
   Parents()
```

```
  {
  System.out.println("Inside Parents constructor");
  }
}
class Child extends Parents
{
  Child()
  {
    System.out.println("Inside Child constructor");
  }
}
class CallingConstructor
{
  public static void main (String[] args)
  {
    Child c = new Child();
  }
}
```

Output of the program:

```
Inside Grandparents constructor
Inside Parents constructor
Inside Child constructor
```

It can be seen in the above example that the constructor execution is in the order of derivation. If you think that this order makes sense, it is because a superclass has no knowledge of any subclass. Therefore any initialization it needs to perform is separate from and possibly prerequisite to any initialization performed by the subclass. Therefore super class constructor must complete its execution first.

1.10 USE OF FINAL KEYWORD

The final keyword in Java is to **restrict the access of its contents from being modified**. If any field is declared as final, it means that the field can not be modified and will become **essentially a constant**. This declaration can be made in one of two ways: First, you can give it a value during declaration, and second, you can assign a value within a constructor.

Assigning value while the declaration is one of the most common practices. For example

```
final int NUMB = 1;
```

The final keyword is quite useful when you want a variable to **always store the same value throughout the program**, for example

```
final double PI = 3.14;
```

The final keyword can be used along with the variables, methods, and classes like- final variable, final method, and final class respectively.

```
class Bike
{
  final int speedlimit=90; /*-------final variable--------*/
  void run()
  {
    speedlimit = 200;
    System.out.println("Speed limit is:" +speedlimit);
  }
  public static void main(String args[ ])
  {
    Bike bike = new Bike( );
    bike.run( );
  }
}
```

Output of the program:

```
Error: can not assign a value to final variable speedlimit.
```

There can be a case of the final blank variable when it is not initialized during declaration. Then it is a must to initialize in the constructor of the class; otherwise, it will throw a compile-time error, for example.

```
class Blank_Test
{
  final int i;
  Blank_Test(int x)
  {
    i = x;       /*---initialization of final variable inside the constructor---*/
  }
}
class Mainclass
{
  public static void main (String[] args)
  {
    Blank_Test bt1 = new Blank_Test(15);
    System.out.println("Value of i is:"+bt1.i);
  }
}
```

Output of the program:

```
Value of i is: 15
```

Use of final keyword with the variables are stated above, now other two uses, i.e., with method and class apply to Inheritance.

Using final to Prevent Overriding

Overriding in Java is one of the most powerful features; there will be times when you will want to prevent it from occurring. If you want to disallow a method to be overridden, mention the final as a modifier at the beginning of its declaration. The method declared as final can not be overridden. Look at the example given below.

```
class Ram
{
  final void method1( )
  {
    System.out.println("This is a final method.");
  }
}
class Ravi extends Ram
{
  void method1( )
  {
    /*---Error, Can't override----*/
    System.out.println("Illegal, method cand not be overridden");
  }
}
```

Output of the program:

```
Error: method1( ) in Ravi cannot override method1( ) in Ram
```

Because method1( ) is declared as final that's why it can not be overridden in Ravi. If you will try to do so, it will give a compile-time error.

Using final to Prevent Inheritance
final keyword can also be used to prevent a class from being overridden. To do this, start the class declaration with the final keyword; doing this, it implicitly declares all of its methods as final.

```
final class Ram
{
  /*-------class Ram is declared as final---------------- */
  void method()
  {
    System.out.println("This is a final class.");
  }
}
class Ravi extends Ram
{
  void method()
  {
    System.out.println("Ravi can not inherit the class Ram");
  }
}
```

Output of the program:

```
Error: cannot inherit from final Ram
```

CHECK YOUR PROGRESS-3

Q1. What do you mean by method hiding in Java?

....

....

....

....

Q2. The code given below shows a compile-time error. Can you suggest the appropriate corrections?

```
class X
{
   public X(int i)
   {
      System.out.println(“Parent Class.”);
   }
}

class Y extends X
{
   public Y()
   {
      System.out.println(“Child Class.”);
   }
}
```

Q3. What will be the output of the program given below?

```
class Base
 {
   public void Print()
   {
      System.out.println("Welcome to Base class");
   }
}

class Derived extends Base {
   public void Print() {
      System.out.println("Now, it is Derived class");
   }
}

class Main
{
   public static void DoPrint( Base o ) {
      o.Print();
   }
   public static void main(String[] args) {
      Base x = new Base();
      Base y = new Derived();
      Derived z = new Derived();
      DoPrint(x);
      DoPrint(y);
      DoPrint(z);
   }
}
```

....

....

Q4. Can you guess? How many public classes a Java program can have?

....

....

....

....

......

1.11 SUMMARY

In this unit, we have discussed the concept of Inheritance, which is one of the importent feature of object oriented programming. This unit explained how to derive the properties of one class into another class. We also discussed the access specifiers of data members along with a comparative summary Also in the unit multi-level class hierarchy has been discussed. A few key terms like "Super", "Final" and "This" are also discussed. Also in this unit, demonstration of constructors calling has been discussed.

1.12 SOLUTIONS/ANSWER TO CHECK YOUR PROGRESS

Check your progress 1:

Answer 1: In Java programming, a class can not extend more than one class; here in this question, Reeta extends more than Ram and Shayam two classes.

Java does not support Multiple Inheritance. If we try to perform, then it may suffer from collision of the methods with the same name because multiple classes may have the method with the same name.

For example, if a Class $Child$ extends the properties of the class $Parent_1$ and class $Parent_2$ which have a method with the same name, then Class $Child$ will have two methods with the same name. This will create ambiguity and confusion about which one to use. Therefore, to avoid this situation, java does not support multiple Inheritance.

Answer 2: The output of the program is: 200

Answer 3:

In the given code, class Ram itself has been defined with default access modifier, which indicates that class Ram can be instantiated within the package in which it is defined but it can not be instantiated outside the package, even though if it has a public constructor.

Answer 4:

Yes, it is written correctly. Its output will be A

Check your progress 2:

Answer 1. The output of the program is given below.

You are in class A.

You are in class B

Here, Now in class C

Answer 2

The output of the program is:

21

32

25

Answer 3

The top-level class or superclass is known as “Generalized class”. It contains common data and common behaviour. And the low-level or sub-level classes are known as “specialized classes”. In general, it contains more specific data.

Example:

```
class Person
{
int name;
int age;
}
Class Student extends Person
{
int roll_Number;
int marks;
}
class Employee extends Person
{
int employee_Id;
float emp_Salary;

}
```

In this class hierarchy, Person is generalized class and Student and Employee are the specialized classes.

Check your progress 3:

Answer 1.

If a subclass contains a static method with the same signature as a static method defined in the superclass, then the method defined in the subclass hides the one defined in the superclass.

There is a difference between hiding a static method and overriding an instance method:

- Invoked overridden instance method will be from subclass.
- And hidden static method that gets invoked depends on whether it is invoked from superclass or subclass.

For example, suppose that there are two classes; a superclass Animal and subclass Cat, which contains one instance method and one static method.

```
/*---------------Animal Class-----------------------------*/
package Jtest;
public class Animal
{
   public static void Eating_habit()
   {
       System.out.println("Generally, Animals eat grass.");
   }
   public void InstanceEating_habit()
   {
      System.out.println("Instance Method: Animals eat grass");
   }
}
/*----------------------------Cat class----------------------------*/
package Jtest;
public class Cat extends Animal
{
  public static void Eating_habit()
   {
     System.out.println("Static: Cat drinks the Milk.");
   }
  public void InstanceEating_habit()
   {
     System.out.println("Instance Method: Cat drinks the Milk.");
   }
  public static void main(String[] args)
   {
     Cat mCat = new Cat();
     Animal mAnimal = mCat;
     Animal.Eating_habit();
     mAnimal.InstanceEating_habit();
   }
}
```

Output of the program:

```
Generally, Animals eat grass.

Instance Method: Cat drinks the Milk.
```

In the program, you can see that the Cat class overrides the instance method in Animal and hides the static method in Animal. The main method creates an instance of Cat and invokes Eating_habit, and InstancEating_habit.

Answer 2. Write explicit calling statement to super class constructor in Class Y constructor.

```
class X
```

```
{
   public X(int i)
   {
      System.out.println("Parent Class.");
   }
}

class Y extends X
{
   public Y()
   {
      super(15)                      /* correction */
      System.out.println("Child Class.");
   }
}
```

Answer 3:

```
Welcome to Base class
Now it is Derived class
Now it is Derived class
```

Answer 4: A Java program can have maximum only one public class.

1.13 REFERENCES/FURTHER READING

- Herbert Schildt "Java The Complete Reference", McGraw-Hill,2017.
- Savitch, Walter, " Java: An Introduction to roblem solving & programming", Pearson Education Limited, 2019.
- Neil, O. , "Teach yourself JAVA", Tata McGraw-Hill Education, 1999.
- Sarcar, Vaskaran. " The Concept of Inheritance In Interactive Object-Oriented Programming in Java", Apress, Berkeley, CA, 2020.