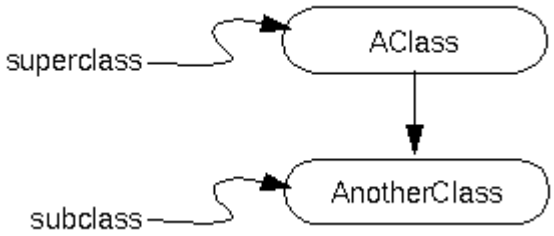
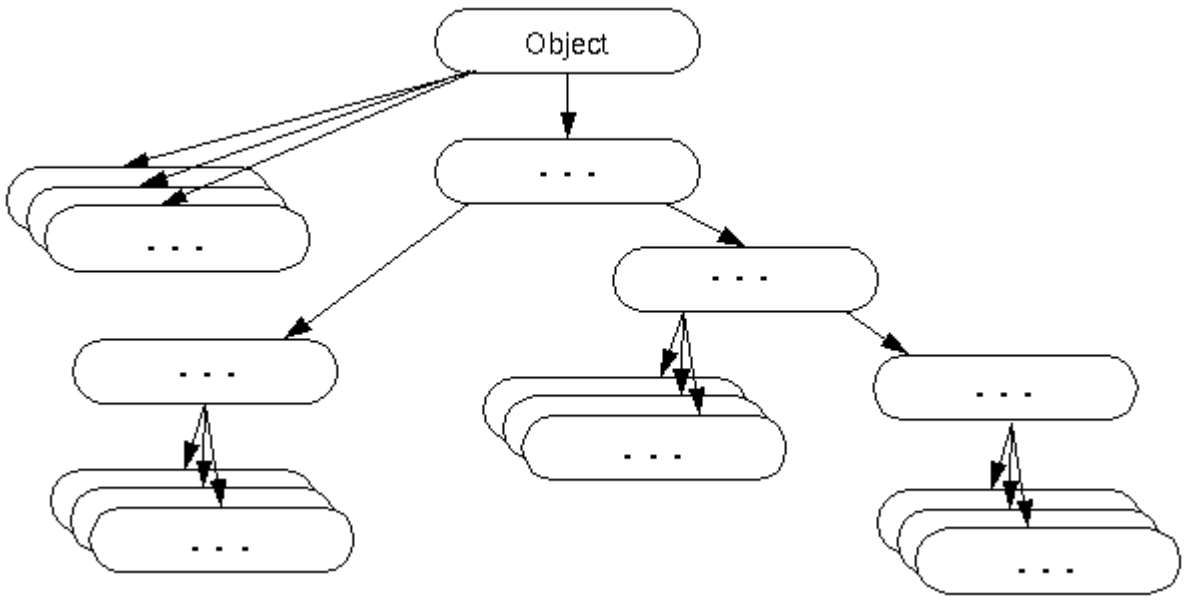
Defining an subclass:

Subclasses, Superclass's, and Inheritance

To recap what you've seen before, classes can be derived from other classes. The derived class (the class that is derived from another class) is called a *subclass*. The class from which it's derived is called the *superclass*. The following figure illustrates these two types of classes:



In fact, in Java, all classes must be derived from some class. Which leads to the question "Where does it all begin?" The top-most class, the class from which all other classes are derived, is the Object class defined in java.lang. Object is the root of a hierarchy of classes, as illustrated in the following figure.



The subclass inherits state and behavior in the form of variables and methods from its superclass. The subclass can use just the items inherited from its superclass as is, or the subclass can modify or override it. So, as you drop down in the hierarchy, the classes become more and more specialized:

Definition: A subclass is a class that derives from another class. A subclass inherits state and behaviour from all of its ancestors. The term superclass refers to a class's direct ancestor as well as all of its ascendant classes.

Creating Subclasses

To create a subclass of another class use the extends clause in your class declaration. (The Class Declaration explains all of the components of a class declaration in detail.) As a subclass, your class inherits member variables and methods from its superclass. Your class can choose to hide variables or override methods inherited from its superclass.

Subclass Constructor:

Subclass Constructors Look again at the PlaneCircle() constructor method of Example: public PlaneCircle(double r, double x, double y)

{ super(r); // Invoke the constructor of the superclass, Circle()

this.cx = x; // Initialize the instance field cx

this.cy = y; // Initialize the instance field cy

}

This constructor explicitly initializes the cx and cy fields newly defined by PlaneCircle, but it relies on the superclass Circle() constructor to initialize the inherited fields of the class. To invoke the superclass constructor, our constructor calls super(). super is a reserved word in Java. One of its uses is to invoke the constructor method of a superclass from within the constructor method of a subclass. This use is analogous to the use of this() to invoke one constructor method of a class from within another constructor method of the same class. Using super() to invoke a constructor is subject to the same restrictions as using this() to invoke a constructor:

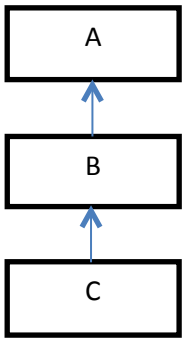
- super() can be used in this way only within a constructor method.
- The call to the superclass constructor must appear as the first statement within the constructor method, even before local variable declarations.

The arguments passed to super() must match the parameters of the superclass constructor. If the superclass defines more than one constructor, super() can be used to invoke any one of them, depending on the arguments passed.

Multilevel Inheritance:

Multilevel inheritance in java with example

When a class extends a class, which extends anther class then this is called **multilevel inheritance**. For example class C extends class B and class B extends class A then this type of inheritance is known as multilevel inheritance.
Lets see this in a diagram:



Multilevel Inheritance

It's pretty clear with the diagram that in Multilevel inheritance there is a concept of grand parent class. If we take the example of this diagram, then class C inherits class B and class B inherits class A which means B is a parent class of C and A is a parent class of B. So in this case class C is implicitly inheriting the properties and methods of class A along with class B that's what is called multilevel inheritance.

Multilevel Inheritance Example

In this example we have three classes – Car, Maruti and Maruti800. We have done a setup – class Maruti extends Car and class Maruti800 extends Maruti. With the help of this Multilevel hierarchy setup our Maruti800 class is able to use the methods of both the classes (Car and Maruti).

```
class Car{
  public Car()
  {
        System.out.println("Class Car");
  }
  public void vehicleType()
  {
        System.out.println("Vehicle Type: Car");
  }
}
class Maruti extends Car{
  public Maruti()
  {
        System.out.println("Class Maruti");
  }
  public void brand()
  {
        System.out.println("Brand: Maruti");
  }
  public void speed()
  {
        System.out.println("Max: 90Kmph");
  }
}
public class Maruti800 extends Maruti{

  public Maruti800()
  {
        System.out.println("Maruti Model: 800");
  }
  public void speed()
  {
        System.out.println("Max: 80Kmph");
  }
```

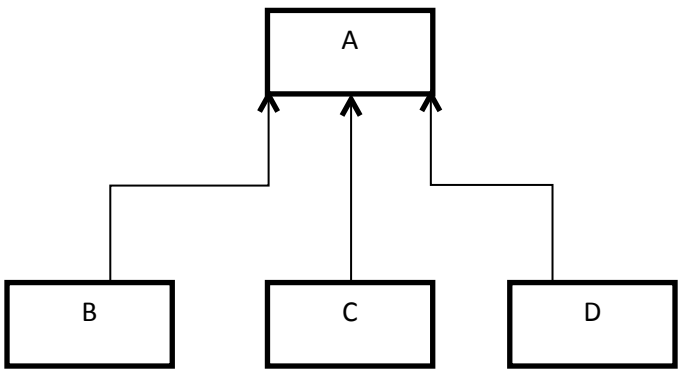
```
  public static void main(String args[])
  {
            Maruti800 obj=new Maruti800();
            obj.vehicleType();
            obj.brand();
            obj.speed();
  }
}
```

Output:

```
Class Car
Class Maruti
Maruti Model: 800
Vehicle Type: Car
Brand: Maruti
Max: 80Kmph
```

Hierarchical Inheritance in java with example program

When more than one classes inherit a same class then this is called hierarchical inheritance. For example class B, C and D extends a same class A. Lets see the diagram representation of this:



Hierarchical Inheritance

As you can see in the above diagram that when a class has more than one child classes (sub classes) or in other words more than one child classes have the same parent class then this type of inheritance is known as **hierarchical inheritance**.

Example of Hierarchical Inheritance

We are writing the program where class B, C and D extends class A.

```
class A
{
  public void methodA()
```

```
  {
    System.out.println("method of Class A");
  }
}
class B extends A
{
  public void methodB()
  {
    System.out.println("method of Class B");
  }
}
class C extends A
{
 public void methodC()
 {
   System.out.println("method of Class C");
 }
}
class D extends A
{
 public void methodD()
 {
   System.out.println("method of Class D");
 }
}
class JavaExample
{
 public static void main(String args[])
 {
   B obj1 = new B();
   C obj2 = new C();
   D obj3 = new D();
   //All classes can access the method of class A
   obj1.methodA();
   obj2.methodA();
   obj3.methodA();
 }
}
```

Output:

```
method of Class A
method of Class A
method of Class A
```

Java - Interfaces

An interface is a reference type in Java. It is similar to class. It is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface.

Along with abstract methods, an interface may also contain constants, default methods, static methods, and nested types. Method bodies exist only for default methods and static methods.

Writing an interface is similar to writing a class. But a class describes the attributes and behaviors of an object. And an interface contains behaviors that a class implements.

Unless the class that implements the interface is abstract, all the methods of the interface need to be defined in the class.

An interface is similar to a class in the following ways −

- An interface can contain any number of methods.
- An interface is written in a file with a **.java** extension, with the name of the interface matching the name of the file.
- The byte code of an interface appears in a **.class** file.
- Interfaces appear in packages, and their corresponding bytecode file must be in a directory structure that matches the package name.

However, an interface is different from a class in several ways, including −

- You cannot instantiate an interface.
- An interface does not contain any constructors.
- All of the methods in an interface are abstract.
- An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.
- An interface is not extended by a class; it is implemented by a class.
- An interface can extend multiple interfaces.

Declaring/Defining Interfaces

The **interface** keyword is used to declare an interface. Here is a simple example to declare an interface −

Example

Following is an example of an interface −

```
/* File name : NameOfInterface.java */
import java.lang.*;
// Any number of import statements

public interface NameOfInterface {
   // Any number of final, static fields
   // Any number of abstract method declarations\
}
```

Interfaces have the following properties −

- An interface is implicitly abstract. You do not need to use the **abstract** keyword while declaring an interface.

- Each method in an interface is also implicitly abstract, so the abstract keyword is not needed.
- Methods in an interface are implicitly public.

Example

```
/* File name : Animal.java */
interface Animal {
   public void eat();
   public void travel();
}
```

Implementing Interfaces

When a class implements an interface, you can think of the class as signing a contract, agreeing to perform the specific behaviors of the interface. If a class does not perform all the behaviors of the interface, the class must declare itself as abstract.

A class uses the **implements** keyword to implement an interface. The implements keyword appears in the class declaration following the extends portion of the declaration.

Example

```
/* File name : MammalInt.java */
public class MammalInt implements Animal {

   public void eat() {
      System.out.println("Mammal eats");
   }

   public void travel() {
      System.out.println("Mammal travels");
   }

   public int noOfLegs() {
      return 0;
   }

   public static void main(String args[]) {
      MammalInt m = new MammalInt();
      m.eat();
      m.travel();
   }
}
```

This will produce the following result −

Output

```
Mammal eats
Mammal travels
```

When overriding methods defined in interfaces, there are several rules to be followed −

- Checked exceptions should not be declared on implementation methods other than the ones declared by the interface method or subclasses of those declared by the interface method.
- The signature of the interface method and the same return type or subtype should be maintained when overriding the methods.
- An implementation class itself can be abstract and if so, interface methods need not be implemented.

When implementation interfaces, there are several rules −

- A class can implement more than one interface at a time.
- A class can extend only one class, but implement many interfaces.
- An interface can extend another interface, in a similar way as a class can extend another class.

Extending Interfaces

An interface can extend another interface in the same way that a class can extend another class. The **extends** keyword is used to extend an interface, and the child interface inherits the methods of the parent interface.

The following Sports interface is extended by Hockey and Football interfaces.

Example

```
// Filename: Sports.java
public interface Sports {
   public void setHomeTeam(String name);
   public void setVisitingTeam(String name);
}

// Filename: Football.java
public interface Football extends Sports {
   public void homeTeamScored(int points);
   public void visitingTeamScored(int points);
   public void endOfQuarter(int quarter);
}

// Filename: Hockey.java
public interface Hockey extends Sports {
   public void homeGoalScored();
   public void visitingGoalScored();
   public void endOfPeriod(int period);
   public void overtimePeriod(int ot);
}
```

The Hockey interface has four methods, but it inherits two from Sports; thus, a class that implements Hockey needs to implement all six methods. Similarly, a class that implements Football needs to define the three methods from Football and the two methods from Sports.

Extending Multiple Interfaces

A Java class can only extend one parent class. Multiple inheritance is not allowed. Interfaces are not classes, however, and an interface can extend more than one parent interface.

The extends keyword is used once, and the parent interfaces are declared in a comma-separated list.

For example, if the Hockey interface extended both Sports and Event, it would be declared as −

Example

```
public interface Hockey extends Sports, Event
```

Accessing Interface Variables

Interface Variables must be Final

An interface does not have instance variables. The members of an interface are always declared as static and final, that the variable cannot be modified by the methods in the class. Such variables will be inherited by the class that implements the interface.

Program

```
interface Data {


   int data1 = 50;

   int data2 = 100;

}

class ShowData implements Data {


   void interfaceValues()

   {

      System.out.println("data1 = "+data1);

      System.out.println("data2 = "+data2);
```

```
    }


    void modifyInterfaceValues()
    {
        data1 += 20;
        data2 += 40;
    }
}
public class Javaapp {

    public static void main(String[] args) {

        System.out.println("data1 = "+Data.data1);
        System.out.println("data2 = "+Data.data1);
        ShowData obj = new ShowData();
        obj.interfaceValues();
        obj.modifyInterfaceValues();
        obj.interfaceValues();
    }
```

```
}
```

Program Output

```
data1 = 50
data2 = 50
data1 = 50
data2 = 100
Uncompilable source code - cannot assign a value to final variable data1
        at ShowData.modifyInterfaceValues(Javaapp.java:16)
        at Javaapp.main(Javaapp.java:28)
C:\Users\User\AppData\Local\NetBeans\Cache\8.2\executor-snippets\run.xml:53: Java returned: 1
BUILD FAILED (total time: 0 seconds)
```