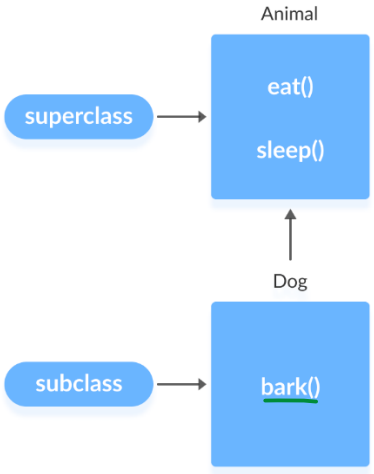
In object-oriented programming, **inheritance** enables new objects to take on the properties of existing objects. A class that is **used as** the basis for **inheritance** is called a superclass or base class. A class that **inherits** from a superclass is called a subclass or



derived class

What are the benefits of inheritance?

One of the key **benefits of inheritance** is to minimize the amount of duplicate code in an application by sharing common code amongst several subclasses, where equivalent code exists in two related classes. This also tends to result in a better organization of code and smaller, simpler compilation units.

Or

Reusability: Inheritance supports the concept of "reusability", i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

Important terminology:

Super Class: The class whose features are inherited is known as super class(or a base class or a parent class).

Sub Class: The class that inherits the other class is known as sub class(or a derived class, extended class, or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.

The keyword used for inheritance is extends.

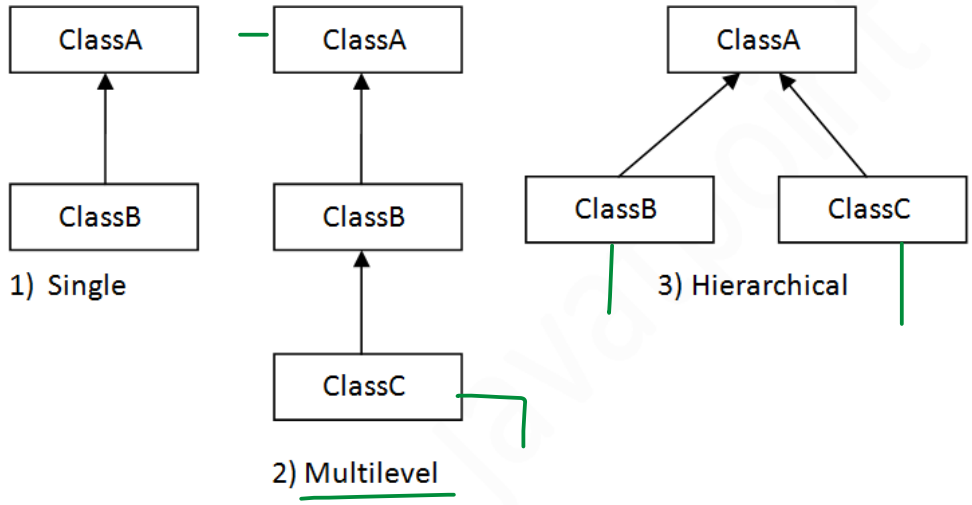
Syntax :

```
class derived-class extends base-class
{
   //methods and fields
}
```

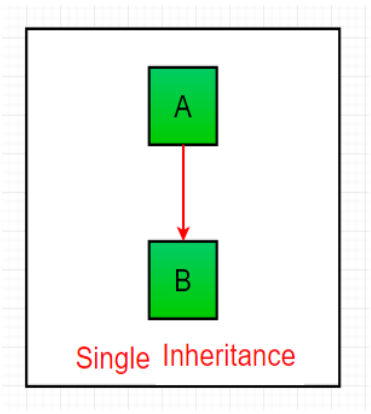
Example

```
class Employee{
 float salary=40000;
}
class Programmer extends Employee{
 int bonus=10000;
 public static void main(String args[]){
   Programmer p=new Programmer();
   System.out.println("Programmer salary is:"+p.salay);
   System.out.println("Bonus of Programmer is:"+p.bonus);
}
}
```

Types of inheritance



Single Inheritance : In single inheritance, subclasses inherit the features of **one** superclass. In image below, the class A serves as a base class for the derived class B.




```
Class A
{
   public void methodA()
   {
     System.out.println("Base class method");
   }
}

Class B extends A
{
   public void methodB()
   {
     System.out.println("Child class method");
   }
   public static void main(String args[])
   {
     B obj = new B();
     obj.methodA(); //calling super class method
     obj.methodB(); //calling local method
  }
}
```

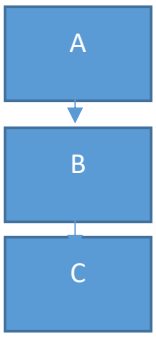
```
class Teacher {
   String designation = "Teacher";
   String collegeName = "Beginnersbook";
   void does(){
       System.out.println("Teaching");
   }
}

public class PhysicsTeacher extends Teacher{
   String mainSubject = "Physics";
   public static void main(String args[]){
       PhysicsTeacher obj = new PhysicsTeacher();
       System.out.println(obj.collegeName);
       System.out.println(obj.designation);
       System.out.println(obj.mainSubject);
       obj.does();
   }
}
```

Output:

```
Beginnersbook
Teacher
Physics
Teaching
```

Multilevel Inheritance

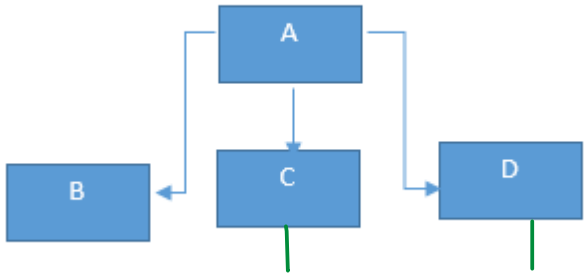
When there is a **chain** of inheritance, it is known as multilevel inheritance. As you can see in the example given below, BabyDog class inherits the Dog class which again inherits the Animal class, so there is a multilevel inheritance.




```
class Animal{
void eat(){System.out.println("eating...");
}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");
}
}
class BabyDog extends Dog{
void weep(){System.out.println("weeping...");
}
}
class TestInheritance2{
public static void main(String args[]){
BabyDog d=new BabyDog();
d.weep();
d.bark();
d.eat();
}
}

Output :
weeping...
barking...
eating...
```

Hierarchical Inheritance :

In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one sub class.In below image, the class A serves as a base class for the derived class B,C and D.

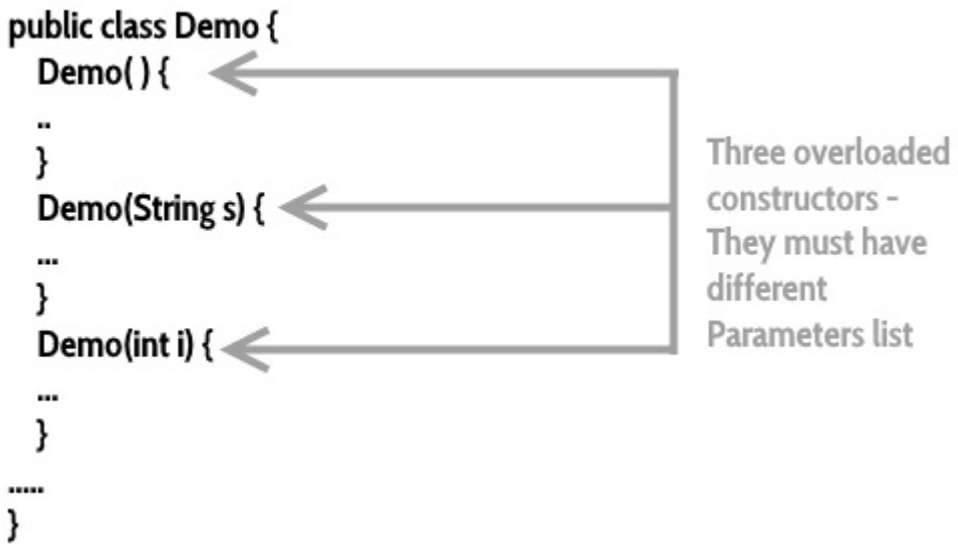



```
class A
{
   public void methodA()
   {
      System.out.println("method of Class A");
   }
}
class B extends A
{
   public void methodB()
   {
      System.out.println("method of Class B");
   }
}
class C extends A
{
  public void methodC()
  {
     System.out.println("method of Class C");
  }
}
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();
  }
}

Outout

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

Constructor Overloading in Java

Constructor overloading is a concept of having more than one constructor with different parameters list, in such a way so that each constructor performs a different task.




```
class Student{
  private String name;
  public Student(String n){
    name = n;
  }
  public Student(){
    name = "unknown";
  }
  public void printName(){
    System.out.println(name);
  }
}
class Cu1{
  public static void main(String[] args){
    Student a = new Student("xyz");
    Student b = new Student();
    a.printName();
    b.printName();
  }
}

Output
```

```
xyz
unknown
```

Example 2 Constructor overloading

```
class StudentData
{
   private int stuID;
   private String stuName;
   private int stuAge;
   StudentData()
   {
       //Default constructor
       stuID = 100;
       stuName = "New Student";
       stuAge = 18;
   }
   StudentData(int num1, String str, int num2)
   {
       //Parameterized constructor
       stuID = num1;
       stuName = str;
       stuAge = num2;
   }
   //Getter and setter methods
   public int getStuID() {
       return stuID;
   }
   public void setStuID(int stuID) {
       this.stuID = stuID;
   }
   public String getStuName() {
       return stuName;
   }
   public void setStuName(String stuName) {
       this.stuName = stuName;
   }
   public int getStuAge() {
       return stuAge;
   }
```

```
    public void setStuAge(int stuAge) {
        this.stuAge = stuAge;
    }

    public static void main(String args[])
    {
        //This object creation would call the default constructor
        StudentData myobj = new StudentData();
        System.out.println("Student Name is: "+myobj.getStuName());
        System.out.println("Student Age is: "+myobj.getStuAge());
        System.out.println("Student ID is: "+myobj.getStuID());

        /*This object creation would call the parameterized
         * constructor StudentData(int, String, int)*/
        StudentData myobj2 = new StudentData(555, "Chaitanya", 25);
        System.out.println("Student Name is: "+myobj2.getStuName());
        System.out.println("Student Age is: "+myobj2.getStuAge());
        System.out.println("Student ID is: "+myobj2.getStuID());
    }
}
```

Output:

```
Student Name is: New Student
Student Age is: 18
Student ID is: 100
Student Name is: Chaitanya
Student Age is: 25
Student ID is: 555
```

Method Overloading in Java

Method Overloading is a feature that allows a class to have more than one method having the same name, if their argument lists are different.

Three ways to overload a method or define valid method signature

In order to overload a method, the argument lists of the methods must differ in either of these:

1. Number of parameters.

For example: This is a valid case of overloading

add(int, int)

add(int, int, int)

2. Data type of parameters.

For example:

add(int, int)

add(int, float)

3. Sequence of Data type of parameters.

For example:

add(int, float)

add(float, int)

Example of method overloading

```
class DisplayOverloading
{
    public void disp(char c)
    {
         System.out.println(c);
    }
    public void disp(char c, int num)
    {
         System.out.println(c + " "+num);
    }
}
class Sample
{
   public static void main(String args[])
   {
       DisplayOverloading obj = new DisplayOverloading();
       obj.disp('a');
       obj.disp('a',10);
   }
}
```

Output:

```
a
a 10
```

Method overriding in java

Declaring a method in **sub class** which is already **present** in **parent class** is known as **method overriding.** Overriding is done so that a child class can give its own implementation to a method which is already provided by the parent class. In this case the method in parent class is called overridden method and the method in child class is called overriding method. In this guide, we will see what is method overriding in Java and why we use it.

Example

```
class Human{

   //Overridden method
   public void eat()
   {
      System.out.println("Human is eating");
   }
}
class Boy extends Human{
   //Overriding method
   public void eat(){
      System.out.println("Boy is eating");
   }
   public static void main( String args[]) {
      Boy obj = new Boy();
      //This will call the child class version of eat()
      obj.eat();
   }
}

Output
Boy is eating
```

Advantage of method overriding

The main advantage of method overriding is that the class can give its own specific implementation to a inherited method **without even modifying the parent class code.**

This is helpful when a class has several child classes, so if a child class needs to use the parent class method, it can use it and the other classes that want to have different implementation can use overriding feature to make changes without touching the parent class code.

Method Overriding and Dynamic Method Dispatch

Method Overriding is an example of runtime polymorphism. When a parent class reference points to the child class object then the call to the overridden method is determined at runtime, because during method call which method(parent class or child class) is to be executed is determined by the type of object. This process in which call to the overridden method is resolved at runtime is known

```
lass ABC{
   //Overridden method
   public void disp()
   {
         System.out.println("disp() method
of parent class");
   }
}
class Demo extends ABC{
   //Overriding method
   public void disp(){
         System.out.println("disp() method
of Child class");
   }
   public void newMethod(){
         System.out.println("new method of
child class");
   }
```

```
public static void main( String args[]) {
         /* When Parent class reference refers
to the parent class object
          * then in this case overridden method
(the method of parent class)
          *  is called.
          */
         ABC obj = new ABC();
         obj.disp();

         /* When parent class reference refers
to the child class object
          * then the overriding method (method
of child class) is called.
          * This is called dynamic method
dispatch and runtime polymorphism
          */
         ABC obj2 = new Demo();
         obj2.disp();
   }
```

Output:

```
disp() method of parent class
disp() method of Child class
```

In the above example the call to the disp() method using second object (obj2) is runtime polymorphism (or dynamic method dispatch).

Rules of method overriding in Java

1. Argument list: The argument list of overriding method (method of child class) must match the Overridden method(the method of parent class). The data types of the arguments and their sequence should exactly match.
2. Access Modifier of the overriding method (method of subclass) cannot be more restrictive than the overridden method of parent class. For e.g. if the Access Modifier of parent class method is public then the overriding method (child class method) cannot have private, protected and default Access modifier,because all of these three access modifiers are more restrictive than public.
   For e.g. This is **not allowed** as child class disp method is more restrictive(protected) than base class(public)

```
class MyBaseClass{
   public void disp()
   {
       System.out.println("Parent class method");
   }
}
class MyChildClass extends MyBaseClass{
   protected void disp(){
      System.out.println("Child class method");
   }
   public static void main( String args[]) {
      MyChildClass obj = new MyChildClass();
      obj.disp();
   }
}
```

Output:

```
Exception in thread "main" java.lang.Error: Unresolved
compilation
problem: Cannot reduce the visibility of the inherited method
from MyBaseClass
```

However this is perfectly valid scenario as public is less restrictive than protected. Same access modifier is also a valid one.

```
class MyBaseClass{
   protected void disp()
   {
       System.out.println("Parent class method");
   }
}
class MyChildClass extends MyBaseClass{
```

```
   public void disp(){
      System.out.println("Child class method");
   }
   public static void main( String args[]) {
      MyChildClass obj = new MyChildClass();
      obj.disp();
   }
}
```

Output:

```
Child class method
```

3. private, static and final methods cannot be overridden as they are local to the class. However static methods can be re-declared in the sub class, in this case the sub-class method would act differently and will have nothing to do with the same static method of parent class.
4. Overriding method (method of child class) can throw unchecked exceptions, regardless of whether the overridden method(method of parent class) throws any exception or not. However the overriding method should not throw checked exceptions that are new or broader than the ones declared by the overridden method. We will discuss this in detail with example in the upcoming tutorial.
5. Binding of overridden methods happen at runtime which is known as dynamic binding.
6. If a class is extending an abstract class or implementing an interface then it has to override all the abstract methods unless the class itself is a abstract class.

Final Keyword In Java

The **final keyword** in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

1. variable
2. method
3. class

The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only. We will have detailed learning of these. Let's first learn the basics of final keyword.

Example of final variable

There is a final variable speedlimit, we are going to change the value of this variable, but It can't be changed because final variable once assigned a value can never be changed.

```
class Bike9{
 final int speedlimit=90;//final variable
 void run(){
  speedlimit=400;
 }
 public static void main(String args[]){
 Bike9 obj=new  Bike9();
 obj.run();
 }
}//end of class
```

Output:Compile Time Error

Once final variable declared with any value it can’t be change during execution .

Java final method

If you make any method as final, you cannot override it.

```
class Bike{
  final void run(){System.out.println("running");}
}

class Honda extends Bike{
   void run(){System.out.println("running safely with 100kmph");}

   public static void main(String args[]){
   Honda honda= new Honda();
   honda.run();
    }
}
```

Output:Compile Time Error - (method as final, you cannot override it)

Java final class

If you make any class as final, you cannot extend it.

```
final class Bike{}

class Honda1 extends Bike{
  void run(){System.out.println("running safely with 100kmph");}

  public static void main(String args[]){
  Honda1 honda= new Honda1();
  honda.run();
  }
}
```

Output:Compile Time Error

Super Keyword in Java

The **super** keyword in java is a reference variable that is used to refer parent class objects. The keyword "super" came into the picture with the concept of Inheritance. It is majorly used in the following contexts:

1. **Use of super with variables:** This scenario occurs when a derived class and base class has same data members. In that case there is a possibility of ambiguity for the JVM. We can understand it more clearly using this code snippet:

```
/* Base class vehicle */
class Vehicle
{
    int maxSpeed = 120;
}

/* sub class Car extending vehicle */
class Car extends Vehicle
{
    int maxSpeed = 180;

    void display()
    {
        /* print maxSpeed of base class (vehicle) */
        System.out.println("Maximum Speed: " + super.maxSpeed);
    }
}

/* Driver program to test */
class Test
{
    public static void main(String[] args)
    {
        Car small = new Car();
        small.display();
    }
}
```

```
Ouput

Maximum Speed: 120
```

2. Use of super with methods: This is used when we want to call parent class method. So whenever a parent and child class have same named methods then to resolve ambiguity we use super keyword. This code snippet helps to understand the said usage of super keyword.

```
/* Base class Person */
class Person
{
    void message()
    {
        System.out.println("This is person class");
    }
}

/* Subclass Student */
class Student extends Person
{
    void message()
    {
        System.out.println("This is student class");
    }

    // Note that display() is only in Student class
    void display()
    {
        // will invoke or call current class message() method
        message();

        // will invoke or call parent class message() method
        super.message();
    }
}

/* Driver program to test */
class Test
{
    public static void main(String args[])
    {
        Student s = new Student();

        // calling display() of Student
        s.display();
    }
}
Output

This is student class

This is person class
```

4. **Use of super with constructors:** super keyword can also be used to access the parent class constructor. One more important thing is that, ''super' can call both parametric as well as non parametric constructors depending upon the situation. Following is the code snippet to explain the above concept:

```
/* superclass Person */
class Person
{
    Person()
    {
        System.out.println("Person class Constructor");
    }
}

/* subclass Student extending the Person class */
class Student extends Person
{
    Student()
    {
        // invoke or call parent class constructor
        super();

        System.out.println("Student class Constructor");
    }
}

/* Driver program to test*/
class Test
{
    public static void main(String[] args)
    {
        Student s = new Student();
    }
}

Output

Person class Constructor
Student class Constructor
```

Abstract class in Java

A Java class that is declared using the keyword abstract is called an abstract class. New instances cannot be created for an abstract class but it can be extended. An abstract class can have abstract methods and concrete methods or both. Methods with implementation body are concrete methods. An abstract class can have static fields and methods and they can be used the same way as used in a concrete class. Following is an example for Java abstract class.

```
//abstract parent class
abstract class Animal{
```

```
   //abstract method
   public abstract void sound();
}
//Dog class extends Animal class
public class Dog extends Animal{

   public void sound(){
       System.out.println("Woof");
   }
   public static void main(String args[]){
       Animal obj = new Dog();
       obj.sound();
   }
}

Output
Woof
```

Java Abstract Method

A method that is declared using the keyword abstract is called an abstract method. Abstract methods are declaration only and it will not have implementation. It will not have a method body. A Java class containing an abstract class must be declared as abstract class. An abstract method can only set a visibility modifier, one of public or protected. That is, an abstract method cannot add static or final modifier to the declaration. Following is an example for Java abstract method.

```
public abstract class Animal {

       String name;


       public abstract String getSound();


       public String getName() {

              return name;

       }

}
```

Java Interfaces

An **interface in Java** is a blueprint of a class. It has static constants and abstract methods.

The interface in Java is *a mechanism to achieve abstraction*. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritance in Java.

In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.

Syntax

```
interface <interface_name>{

    // declare constant fields
    // declare methods that abstract
    // by default.
}
```

Example

```
/* File name : Animal.java */
```

```
// Interface
interface Animal {
  public void animalSound(); // interface method (does not have a body)
  public void sleep(); // interface method (does not have a body)
}

// Pig "implements" the Animal interface
class Pig implements Animal {
  public void animalSound() {
    // The body of animalSound() is provided here
    System.out.println("The pig says: wee wee");
  }
  public void sleep() {
    // The body of sleep() is provided here
    System.out.println("Zzz");
```

```
  }
}

class MyMainClass {
  public static void main(String[] args) {
    Pig myPig = new Pig();  // Create a Pig object
    myPig.animalSound();
    myPig.sleep();
  }
}
```

Output

```
The pig says: wee wee
Zzz
```

Notes on Interfaces:

- Like **abstract classes**, interfaces **cannot** be used to create objects (in the example above, it is not possible to create an "Animal" object in the MyMainClass)
- Interface methods do not have a body - the body is provided by the "implement" class
- On implementation of an interface, you must override all of its methods
- Interface methods are by default `abstract` and `public`
- Interface attributes are by default `public`, `static` and `final`
- An interface cannot contain a constructor (as it cannot be used to create objects)

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.

Example

```
interface printable{
void print();
}
class A6 implements printable{
public void print(){System.out.println("Hello");}

public static void main(String args[]){
A6 obj = new A6();
obj.print();
 }
}
Output
Hello
```

Example

```
interface Bank{
float rateOfInterest();
}
class SBI implements Bank{
public float rateOfInterest(){return 9.15f;}
}
class PNB implements Bank{
public float rateOfInterest(){return 9.7f;}
}
class TestInterface2{
public static void main(String[] args){
Bank b=new SBI();
System.out.println("ROI: "+b.rateOfInterest());
}}
Output
ROI: 9.15
```

Interface extends interface in java

An interface can inherit or say, can extends another interface or other multiple interfaces in java programs.

when a class inherit an interface, we use the keyword "implements" like "class X implements A". When interface inherits another interface, we use "extends" keyword as given in example below.

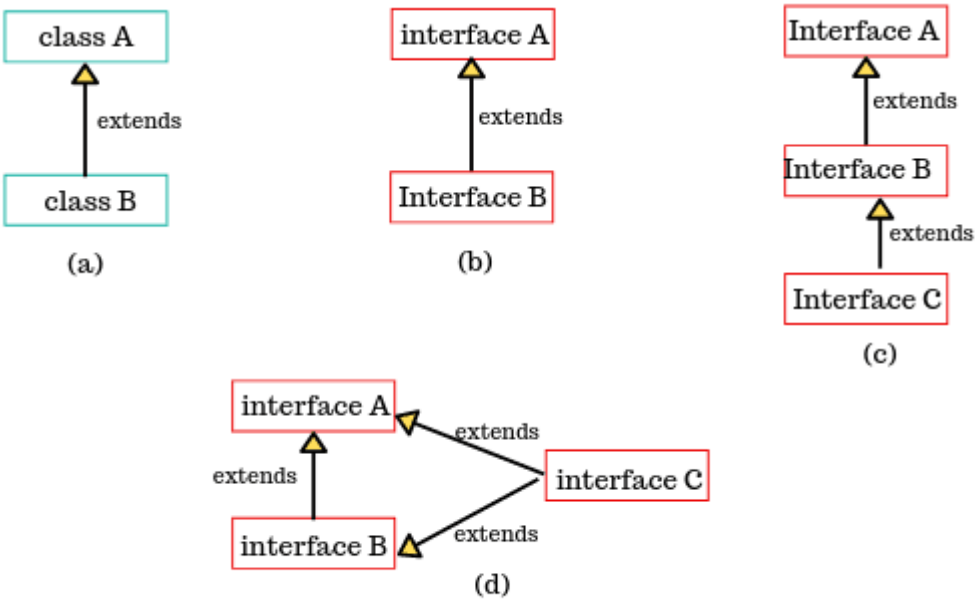


Fig: Various forms of extending Interface in Java

```
interface Inf1{
   public void method1();
}
interface Inf2 extends Inf1 {
   public void method2();
}
public class Demo implements Inf2{
   /* Even though this class is only implementing the
    * interface Inf2, it has to implement all the methods
    * of Inf1 as well because the interface Inf2 extends Inf1
    */
    public void method1(){
       System.out.println("method1");
    }
    public void method2(){
       System.out.println("method2");
    }
    public static void main(String args[]){
       Inf2 obj = new Demo();
       obj.method2();
    }
}
```

Nested Interface in Java

We can declare interfaces as member of a class or another interface. Such an interface is called as member interface or nested interface.

```
import java.util.*;
```

```
class Test
{
    interface Yes
    {
        void show();
    }
}

class Testing implements Test.Yes
{
    public void show()
    {
        System.out.println("show method of interface");
    }
}

class A
{
    public static void main(String[] args)
    {
        Test.Yes obj;
        Testing t = new Testing();
        obj=t;
        obj.show();
    }
}
o/p
show method of interface
```

Interface in another Interface

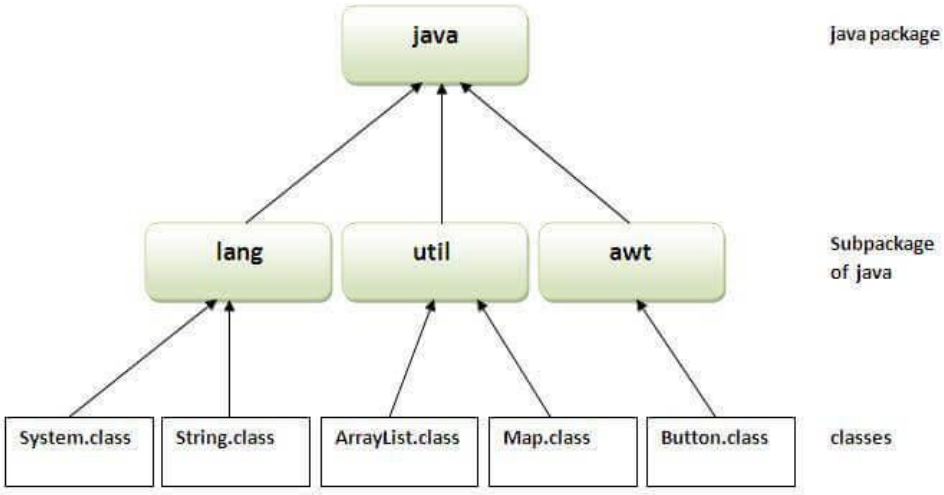
```
// Java program to demonstrate working of
// interface inside another interface.
import java.util.*;
interface Test
{
   interface Yes
   {
      void show();
   }
}

class Testing implements Test.Yes
{
   public void show()
   {
      System.out.println("show method of interface");
   }
}

class A
{
   public static void main(String[] args)
   {
     Test.Yes obj;
     Testing t = new Testing();
     obj = t;
     obj.show();
```

```
    }
}
Output

show method of interface
```

Java Packages & API

A package in Java is used to group related classes. Think of it as **a folder in a file directory.** We use packages to avoid name conflicts, and to write a better maintainable code. Packages are divided into two categories:

- **Built-in Packages (packages from the Java API)**
- **User-defined Packages (create your own packages)**



Advantage of Java Package

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.

2) Java package provides access protection.

3) Java package removes naming collision.

Built-in Packages

The Java API is a library of prewritten classes, that are free to use, included in the Java Development Environment.

The library contains components for managing input, database programming, and much much more. The library is divided

into **packages** and **classes**. Meaning you can either import a single class (along with its methods and attributes), or a whole package that contain all the classes that belong to the specified package.

To use a class or a package from the library, you need to use the `import` keyword:

Syntax

```
import package.name.Class;   // Import a single class
import package.name.*;   // Import the whole package
```

Example

```
import java.util.Scanner;

class MyClass {
  public static void main(String[] args) {
    Scanner myObj = new Scanner(System.in);
    System.out.println("Enter username");

    String userName = myObj.nextLine();
    System.out.println("Username is: " + userName);
  }
}
```

Output

```
Enter username
Ahinsa
Username is: Ahinsa
```

Built-in Packages

These packages consist of a large number of classes which are a part of Java **API**.Some of the commonly used built-in packages are:

1) **java.lang:** Contains language support classes(e.g classed which defines primitive data types, math operations). This package is automatically imported.

2) **java.io:** Contains classed for supporting input / output operations.

3) **java.util:** Contains utility classes which implement data structures like Linked List, Dictionary and support ; for Date / Time operations.

4) **java.applet:** Contains classes for creating Applets.

5) **java.awt:** Contain classes for implementing the components for graphical user interfaces (like button , ;menus etc).

6) **java.net:** Contain classes for supporting networking operations.

User-defined packages

These are the packages that are defined by the user. First we create a directory **myPackage** (name should be same as the name of the

package). Then create the **MyClass** inside the directory with the first statement being the **package names.**

```
// Name of the package must be same as the directory
// under which this file is saved
package myPackage;

public class MyClass
{
    public void getNames(String s)
    {
        System.out.println(s);
    }
}
```

Now we can use the **MyClass** class in our program.

```
/* import 'MyClass' class from 'names' myPackage */
import myPackage.MyClass;

public class PrintName
{
   public static void main(String args[])
   {
      // Initializing the String variable
      // with a value
      String name = "GeeksforGeeks";

      // Creating an instance of class MyClass in
      // the package.
      MyClass obj = new MyClass();
```

```
        obj.getNames(name);
    }
}
```

Note : MyClass.java must be saved inside the **myPackage** directory since it is a part of the package.

Using Static Import

Static import is a feature introduced in **Java** programming language (versions 5 and above) that allows members (fields and methods) defined in a class as public **static** to be used in Java code without specifying the class in which the field is defined.
Following program demonstrates **static import** :

Note static keyword after import.

```
import static java.lang.System.*;

class StaticImportDemo
{
   public static void main(String args[])
   {
        // We don't need to use 'System.out'
        // as imported using static.
        out.println(" Ahinsa Polytechnic ");
   }
}
```

Output:

Ahinsa Polytechnic