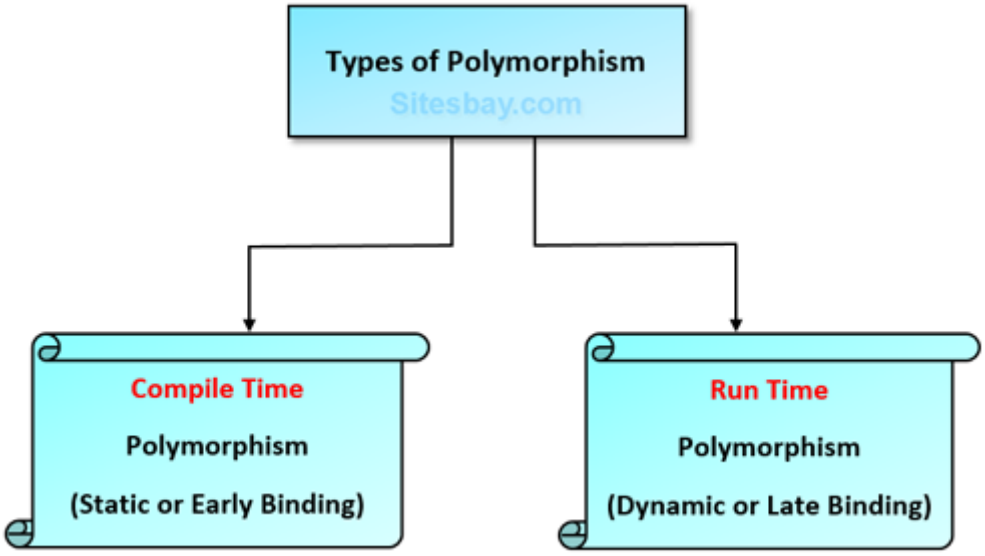
# CHAPTER 8
# POLYMORPHISM

The process of representing one form in multiple forms is known as **Polymorphism**.
Polymorphism is derived from 2 greek words: **poly** and **morphs**. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.

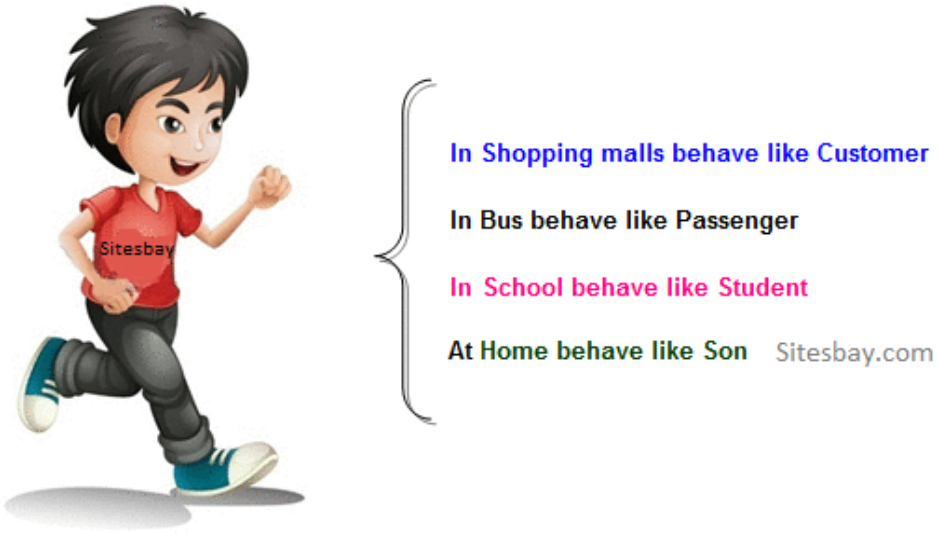
Polymorphism is not a programming concept but it is one of the principal of OOPs. For many objects oriented programming language polymorphism principle is common but whose implementations are varying from one objects oriented programming language to another object oriented programming language.




## Real life example of polymorphism in Java

Suppose if you are in class room that time you behave like a student, when you are in market at that time you behave like a customer, when

you at your home at that time you behave like a son or daughter, Here one person present in different-different behaviors.




**Real life example of polymorphism in Java**

You can see in the below Images, you can see, Man is only one, but he takes multiple roles like - he is a dad to his child, he is an employee, a salesperson and many more. This is known as Polymorphism.


**How to achieve Polymorphism in Java ?**

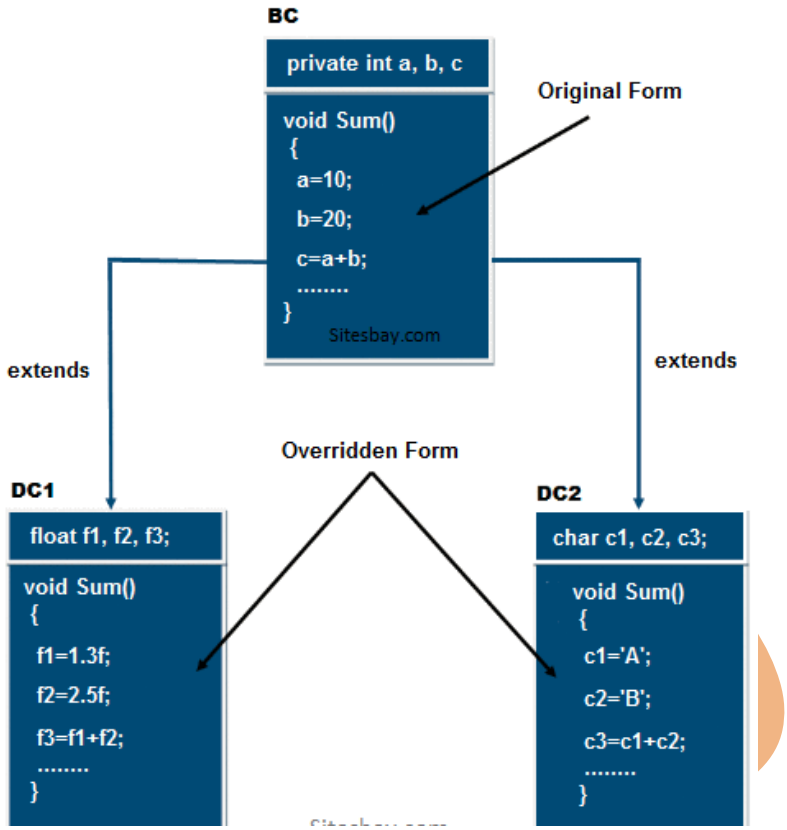
In java programming the Polymorphism principal is implemented with method overriding concept of java.

Polymorphism principal is divided into two sub principal they are:

- Static or Compile time polymorphism
- Dynamic or Runtime polymorphism

Static polymorphism in Java is achieved by method overloading and Dynamic polymorphism in Java is achieved by method overriding.

**Note:** Java programming does not support static polymorphism because of its limitations and java always supports dynamic polymorphism. Here original form or original method always resides in base class and multiple forms represents overridden method which resides in derived classes.




In the above diagram the sum method which is present in Base Class (BC) is called original form and the sum() method which are present in Derived Class1(DC1) and Derived Class (DC2) are called overridden form hence Sum() method is originally available in only one form and it is further implemented in multiple forms. Hence Sum() method is one of the polymorphism method.

**Example1**

```
class Person
{
void walk()
{
System.out.println("Can Run....");
}
}
class Employee extends Person
```

```
{
void walk()
{
System.out.println("Running Fast...");
}
public static void main(String arg[])
{
Person p=new Employee(); //upcasting
p.walk();
}
}
```

**Output**

```
Running fast...
```

In example1 we create two class Person an Employee, Employee class extends Person class feature and override walk() method. We are calling the walk() method by the reference variable of Parent class. Since it refers to the subclass object and subclass method overrides the Parent class method, subclass method is invoked at runtime. Here method invocation is determined by the JVM not compiler, So it is known as runtime polymorphism.

## Dynamic Binding

**Dynamic binding** always says create an object of base class but do not create the object of derived classes. Dynamic binding principal is always used for executing polymorphic applications. The process of binding appropriate versions (overridden method) of derived classes which are inherited from base class with base class object is known as dynamic binding.

### Static polymorphism

The process of binding the overloaded method within object at compile time is known as **Static polymorphism** due to static polymorphism utilization of resources (main memory space) is poor because for each and every overloaded method a memory space is created at compile time when it binds with an object.

## Abstract class in Java

A class that is declared with abstract keyword, is known as abstract class. Before learning abstract class, let's understand the abstraction first.

## Abstraction

**Abstraction** is a process of hiding the implementation details and showing only functionality to the user. Abstraction lets you focus on what the object does instead of how it does it.

Abstraction shows only important things to the user and hides the internal details, for example, when we ride a bike, we only know about how to ride bikes but can not know about how it work? And also we do not know the internal functionality of a bike.


Another real life example of Abstraction is ATM Machine; All are performing operations on the ATM machine like cash withdrawal,

money transfer, retrieve mini-statement…etc. but we can't know internal details about ATM.


Real Life Example of Abstraction

**Note:** Data abstraction can be used to provide security for the data from the unauthorized methods.

## Ways to achieve Abstaction

There are two ways to achieve abstraction in java

1. Abstract class (0 to 100%)
2. Interface (100%)

### Abstract class

A class that is declared as abstract is known as **abstract class**. It needs to be extended and its method implemented. It cannot be instantiated.

### Syntax to declare the abstract class

1. abstract class <class_name>{}

**abstract method**

A method that is declared as abstract and does not have implementation is known as abstract method.

**Syntax to define the abstract method**

1. abstract return_type <method_name>();//no braces{}

**Example of abstract class that have abstract method**

In this example, Bike the abstract class that contains only one abstract method run. It implementation is provided by the Honda class.

```
abstract class Bike
{
  abstract void run();
}

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

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

## Understanding the real scenario of abstract class

In this example, Shape is the abstract class, its implementation is provided by the Rectangle and Circle classes. Mostly, we don't know about the implementation class (i.e. hidden to the end user) and object of the implementation class is provided by the **factory method**.
A **factory method** is the method that returns the instance of the class. We will learn about the factory method later.
In this example, if you create the instance of Rectangle class, draw method of Rectangle class will be invoked.

```
1.  abstract class Shape{
2. abstract void draw();
3. }
4.
5. class Rectangle extends Shape{
6. void draw(){System.out.println("drawing rectangle");}
7. }
8.
9. class Circle extends Shape{
10.      void draw(){System.out.println("drawing circle");}
11.      }
12.
13.      class Test{
14.      public static void main(String args[]){
15.      Shape s=new Circle();
16.      //In real scenario, Object is provided through factory method

17.      s.draw();
18.      }
19.      }
```

Output:drawing circle

## Abstract class having constructor, data member, methods etc.

***Note: An abstract class can have data member, abstract method, method body, constructor and even main() method.***

```
//example of abstract class that have method body
 abstract class Bike{
   abstract void run();
   void changeGear()
{
System.out.println("gear changed");}
 }

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

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

Output:running safely..
gear changed

```
//example of abstract class having constructor, field and method
abstract class Bike
{
 int limit=30;
 Bike(){System.out.println("constructor is invoked");}
 void getDetails(){System.out.println("it has two wheels");}
```

```
 abstract void run();
}

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

 public static void main(String args[])
{
  Bike obj = new Honda();
  obj.run();
  obj.getDetails();
  System.out.println(obj.limit);
 }
}
```

Output:constructor is invoked
running safely..
it has two wheels
30

***Rule: If there is any abstract method in a class, that class must be abstract.***

```
1. class Bike{
2. abstract void run();
3. }
```

Output:compile time error

***Rule: If you are extending any abstact class that have abstract method, you must either provide the implementation of the method or make this class abstract.***

# Interfaces in Java

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, argument lists and return types, but no method bodies. An **interface** can also contain data members of primitive types, but these are implicitly **static** and **final**. An **interface** provides only a form, but no implementation.

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.

In Java programming language, an interface is a reference type, similar to class. Like classes, interfaces contain methods and variables but with a major difference. The major difference is that interface defines only abstract methods and final fields. This means that interfaces do not specify any code to implement these methods and data fields contain only constants.

Therefore, it is the responsibility of the class that implements an interface to define the code for implementation of these methods.

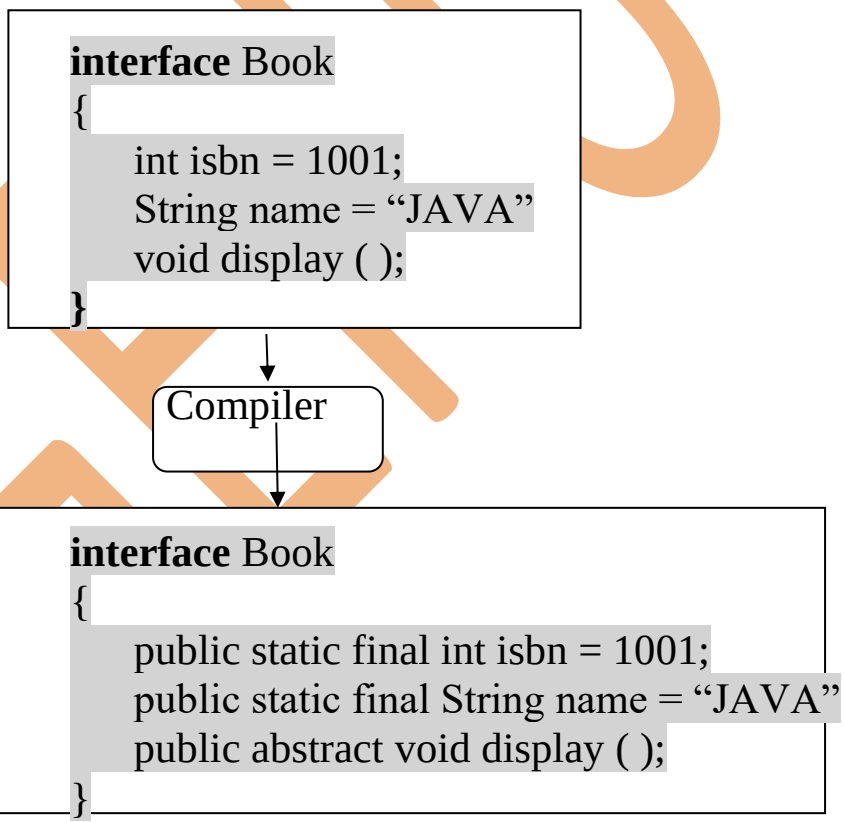
The syntax for defining an interface is similar to creating a new class:

```
interface <InterfaceName>
{
  // constant declarations, if any
     static final <data type> <variable name>  =  <value>;

  // method signatures
     <return type> <methodName>(<parameter list>);

}
```

Here, **interface** is a keyword and *InterfaceName* is any valid java variable.

The interface is **a mechanism to achieve fully abstraction** in java. There can be only abstract methods in the interface. It is used to achieve fully abstraction and multiple inheritance in Java.
Interface also **represents IS-A relationship**.
It cannot be instantiated just like abstract class.

The java compiler adds **public** and **abstract** keywords before the **interface** method and **public, static** and **final** keywords before data members.
In other words, Interface fields are public, static and final by default, and methods are public and abstract.




Here is an example of an interface definition that contains two variables and two methods

```
interface Book
{
    public static final int isbn = 1001;
```

```
    public static final String name = "JAVA"
    public abstract void display ( );
}
```

## Implementing Interface

Once you've implemented an **interface**, that implementation becomes an ordinary class that can be extended in the regular way. You can choose to explicitly declare the method declarations in an **interface** as **public**. But they are **public** even if you don't say it. So when you **implement** an **interface**, the methods from the **interface** must be defined as **public.** Otherwise they would default to "friendly" and you'd be restricting the accessibility.

To use an interface, you write a class that *implements* the interface. When an instantiable class implements an interface, it provides a method body for each of the methods declared in the interface. For example,

```
interface Area
{
    final static float PI = 3.14F;
    float compute(float a , float b);
}


Class Rectangle implements Area
{
   public float compute(float a , float b)
   {
        return(a * b);
   }
}
```

```
class Circle implements Area
{
    public float compute(float x , float y)
    {
      return (PI * x * x );
    }
}


Class DemoInterface
{
   public static void main(String [] args)
   {
      Rectangle rect = new Rectangle ( );
      Circle cir = new Circle( );
      Area  area;
      area = rect;
     System.out.println("Area of Rectangle = " + area.compute(10,20));
      area = cir;
      System.out.println(("Area of Circle = " + area.compute(10,0));
  }
}
```

**Table 5.1 Difference between Class and Interface**

| Class | Interface |
|---|---|
| The members of a class can be constant or variables. | The members of an interface are always declared as constant, i.e. their values are final. |
| The class definition can contain code for each of its methods. That is, the methods can be abstract or non-abstract. | The methods in an interface are abstract in nature, i.e., there is no code associated with them. It is |

| | |
|---|---|
| | later defined by the class that implements the interface. |
| It can be instantiated by declaring objects. | It cannot be used to declare objects. It can only be inherited by a class. |
| It can use various access specifier like public,private or protected. | It can only use the public access specifier. |

## Difference between *abstract* class and *interface*

- An *abstract* class never supports multiple inheritance whereas *interface* support multiple inheritance.
- In *abstract* class method may be abstract or may not be abstract whereas in *interface* each method is implicitly *abstract*.
- In *abstract* class, programmer defines a non-abstract method and constructor where as in *interface* the programmer defines neither the constructor nor nay method.
- In *abstract* class programmer declare any type of variable but in *interface* whatever the variable's are declared are bound to be initialized as they are implicitly *public static* and *final*.
  But both *abstract* class and interface cannot be *instantiated*.

## Extending Interfaces

Like classes, interface can also be extended. That is, an interface can be subinterfaced from other interface. The new subinterface will inherit all the members of the superinterface in the manner similar to subclasses. This is achieved using the keyword extends as shown below:

**interface** <subiterface name> **extends** <superinterface name>
{
    Body of <subinterface name>
}

For example, we can put all the constant in one interface and the methods in the other. This will enable us to use the constants in classes where the method are not required. Example:

```
Interface  ColorConstants
{
    Int colorcode = 1001;
    String colorname =  "PINK"
}

Interface Color extends ColorConstants
{
    Void display ( );
}
```

The interface Color would inherit both the constants *colorcode* and *colorname* into it.

**NOTE : In Java, when a class implements an interface, the methods defined in the interface must be declared as public in the implementing class because:**

1. Interface Methods Are Public by Default:
   All methods in an interface are implicitly public and abstract (unless they are static or default). This ensures they can be accessed by any class that implements the interface.
2. Preserving the Access Contract:
   When a class implements an interface, it's essentially promising to provide concrete definitions for those methods. If you declare them with a more restrictive access modifier like protected or private, you would be breaking the access contract of the interface — other classes expecting to call those methods wouldn't be able to access them.
3. Method Overriding Rules:
   Java follows strict rules for method overriding:
   - You cannot reduce the visibility of an overridden method.
   - Since interface methods are public, the implementing class must maintain or widen (but not narrow) that visibility — meaning the only option is to declare them public.
4. Compile-Time Error Prevention:
   If you try to implement an interface method with anything other than public, the compiler throws an error like:

**error: attempting to assign weaker access privileges; was public**

## Multiple Inheritance Using Interface

Example1:

```
class Student
{
   int rollnumber;
   void getNumber(int n)
```

```
   {
       rollnumber = n;
   }
   void displayNumber( )
   {
      System.out.ptintln(" Roll No. : " + rollnumber);
   }
}

class Test extends Student
{
   int part1,part2;
   void getMarks(int x, int y)
   {
      part1 =x;
      part2 = y;
   }
   void displayMarks( )
   {
     System.out.println("Marks Obtained ");
     System.out.println(" Part1 = " + part1);
     System.out.println(" Part2 = " + part2);
   }
}

Interface Sports

   float sportwt = 0.2F;
   void displayWt( );
}
class Result extends Test implements Sports
{
   float total;
   public void displayWt( )
```

```
   {
         System.out.println(" Sport Weightage  = " + sportWt);
   }
   void display( )
   {
      total = part1 + part2 + sportwt;
      displayNumber( );
      displayMarks( );
      displayWt( );
      System.out.println("Total Score = " + total);
   }
}

class DemoMultiInheritance
{
   public static void main(String args[])
   {
      Result student1 = new Result( );
      student1.getNumber(1234);
      student1.getMarks(27, 33);
      student1.display( );
   }
}
```

**Example 2:**

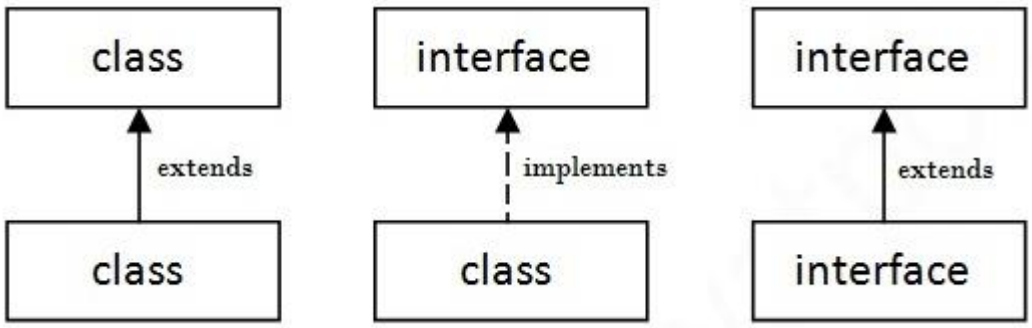
```
interface Transaction {
   void executeTransaction(double amount);
}

class Account {
   String accountNumber;
   double balance;
```

```
    Account(String accountNumber, double balance) {
        this.accountNumber = accountNumber;
        this.balance = balance;
    }

    void displayAccountDetails() {
        System.out.println("Account Number: " + accountNumber);
        System.out.println("Current Balance: Rs." + balance);
    }
}

class SavingsAccount extends Account implements Transaction {
    double interestRate;

    SavingsAccount(String accountNumber, double balance, double
interestRate) {
        super(accountNumber, balance);
        this.interestRate = interestRate;
    }

    void displaySavingsDetails() {
        System.out.println("Interest Rate: " + interestRate + "%");
    }

    @Override
    public void executeTransaction(double amount) {
        if (amount > 0) {
            balance += amount;
            System.out.println("Deposited Rs." + amount);
        } else if (amount < 0 && balance >= amount) {
            balance += amount;
            System.out.println("Withdrawn Rs." + amount);
        } else {
            System.out.println("Insufficient balance for withdrawal!");
```

```
        }
        System.out.println("Updated Balance: Rs." + balance);
    }
}

public class BankingSystem {
    public static void main(String[] args) {
        SavingsAccount savings = new SavingsAccount("1234567890",
50000, 4.5);

        savings.displayAccountDetails();
        savings.displaySavingsDetails();

        savings.executeTransaction(10000); // Deposit Rs. 10000
        savings.executeTransaction(-20000); // Withdraw Rs. 20000
        savings.executeTransaction(-40000); // Attempt to withdraw Rs.
40000 (Insufficient balance)
    }
}
```

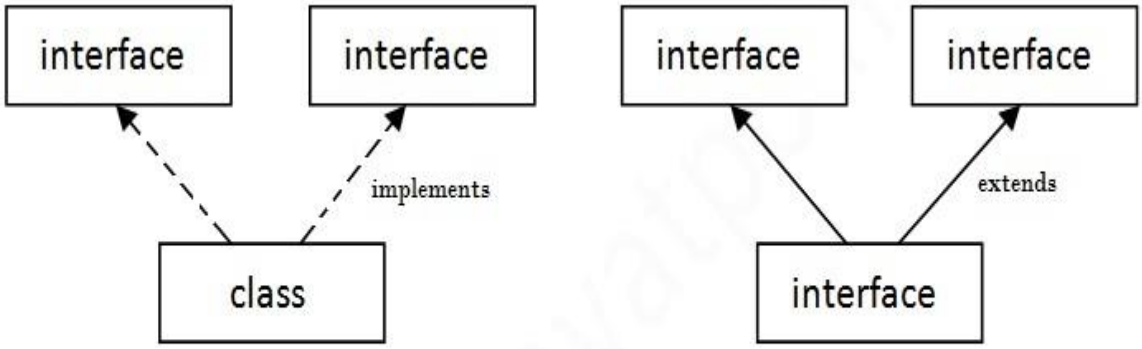
**Understanding relationship between classes and interfaces**

As shown in the figure given below, a class extends another class, an interface extends another interface but a **class implements an interface**.



## Multiple inheritance in Java by interface

If a class implements multiple interfaces, or an interface extends multiple interfaces i.e. known as multiple inheritance.




**Multiple Inheritance in Java**

# Package in Java

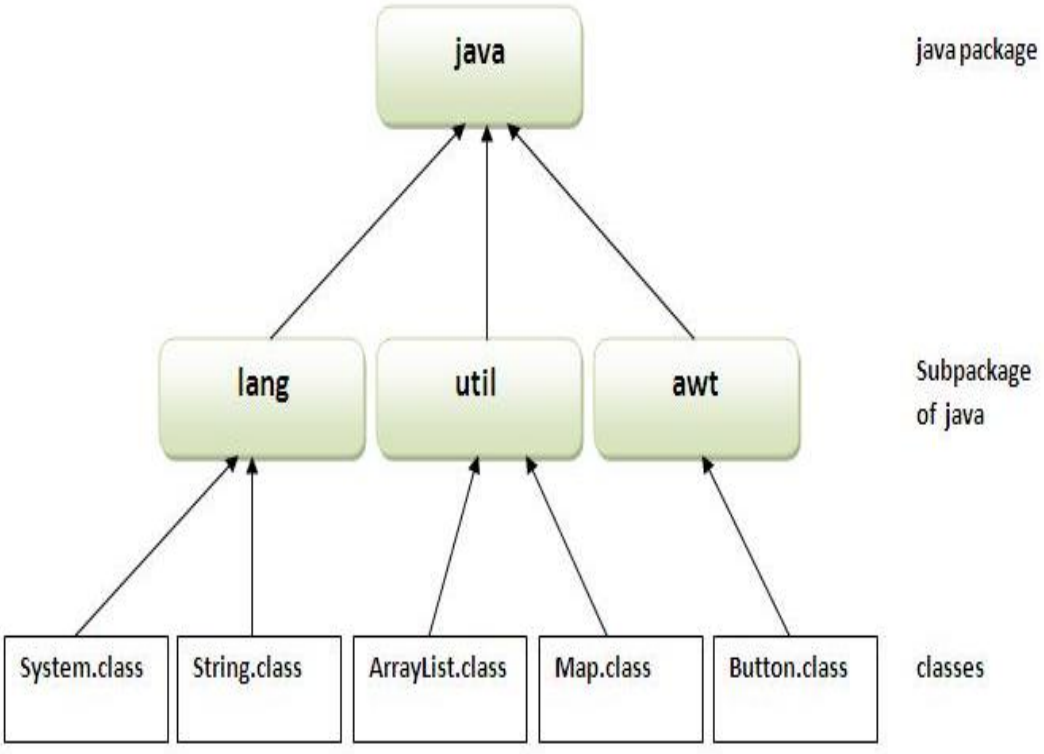
A **package** is a group of similar types of classes, interfaces and sub-packages.

Package can be categorized in two form, built-in package and user-defined package. There are many built-in packages such as ***java***, ***lang, awt, javax, swing, net, io, util, sql*** etc.

Here, we will have the detailed learning of creating and using user-defined packages.

**Advantage of Package**

- Package is used to categorize the classes and interfaces so that they can be easily maintained.
- The classes contained in the package of other program can be easily reused.
- Package provides access protection.
- Package removes naming collision




**Simple example of package**

The **package keyword** is used to create a package.

```
//save as Simple.java
package mypack;
public class Simple{
 public static void main(String args[])
{
   System.out.println("Welcome to package");
  }
}
```

**How to compile the Package (if not using IDE)**

If you are not using any IDE, you need to follow the **syntax** given below:

1. javac -d directory javafilename

For **example**

1. javac -d . Simple.java

The -d switch specifies the destination where to put the generated class file. You can use any directory name like /home (in case of Linux), d:/abc (in case of windows) etc. If you want to keep the package within the same directory, you can use . (dot).

**How to run the Package (if not using IDE)**

You need to use fully qualified name e.g. mypack. Simple etc to run the class.

**To Compile:** javac -d . Simple.java

**To Run:** java mypack.Simple

Output:Welcome to package

The -d is a switch that tells the compiler where to put the class file i.e. it represents destination. The . represents the current folder.

## How to access package from another package?

There are three ways to access the package from outside the package.

1. import package.*;
2. import package.classname;
3. fully qualified name.

***Using packagename.****

If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages.

The import keyword is used to make the classes and interface of another package accessible to the current package.

**Example of package that import the packagename.***

1. //save by A.java
2. 
3. package pack;
4. public class A

```
5. {
6.    public void msg()
7.    {
8.       System.out.println("Hello");
9.    }
10.       }
```

```
1. //save by B.java
2. package mypack;
3. import pack.*;
4. class B
5. {
6.   public static void main(String args[])
7. {
8.    A obj = new A();
9.    obj.msg();
10.       }
11.       }
```

Output:Hello

**Using packagename.classname**

If you import *package.classname* then only declared class of this package will be accessible.

**Example of package by import package.classname**

```
1. //save by A.java
2.
3. package pack;
4. public class A{
5.   public void msg(){System.out.println("Hello");}
6. }
```

```
1. //save by B.java
```

```
2.
3. package mypack;
4. import pack.A;
5.
6. class B{
7.   public static void main(String args[]){
8.    A obj = new A();
9.    obj.msg();
10.         }
11.       }
```

Output:Hello

**Using fully qualified name**

If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.
It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

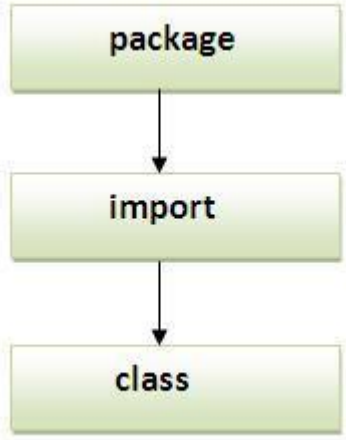
**Example of package by import fully qualified name**

```
1. //save by A.java
2.
3. package pack;
4. public class A{
5.   public void msg(){System.out.println("Hello");}
6. }
1. //save by B.java
2.
3. package mypack;
4. class B{
```

```
5.  public static void main(String args[]){
6.   pack.A obj = new pack.A();//using fully qualified name
7.   obj.msg();
8.  }
9. }
```

Output:Hello

**Note: Sequence of the program must be package then import then class.**




## Subpackage

Package inside the package is called the **subpackage**. It should be created **to categorize the package further**.
Let's take an example, Sun Microsystem has definded a package named java that contains many classes like System, String, Reader, Writer, Socket etc. These classes represent a particular group e.g. Reader and Writer classes are for Input/Output operation, Socket and ServerSocket classes are for networking etc and so on. So, Sun has subcategorized the java package into subpackages such as lang, net, io etc. and put the

Input/Output related classes in io package, Server and ServerSocket classes in net packages and so on.

## Note: If you import a package, subpackages will not be imported.

If you import a package, all the classes and interface of that package will be imported excluding the classes and interfaces of the subpackages. Hence, you need to import the subpackage as well.

### Example of Subpackage

```
package com.gehu.core;
class Simple{
  public static void main(String args[]){
   System.out.println("Hello subpackage");
  }
}
```

**To Compile:** javac -d . Simple.java

**To Run:** java p1.Simple

Output:Hello subpackage

**Rule: There can be only one public class in a java source file and it must be saved by the public class name.**

```
1. //save as C.java otherwise Compile Time Error
2.
3. class A{}
4. class B{}
5. public class C{}
```

# Understanding all java access modifiers

Although a subclass (or derived class) inherits all the members of its super class, yet it can access only those variables/methods for it has access permissions. In other words, the access of inherited members depends upon their access modifiers i.e., whether they are *private* or *public* or *protected* or something else.

There are two types of access modifiers in java: **access modifier** and **non-access modifier**. The access modifiers specifies accessibility (scope) of a datamember, method, constructor or class.

There are 4 types of access modifiers:

1. private
2. public
3. protected
4. default

There are many non-access modifiers such as **static**, **abstract**, **synchronized**, **volatile**, **transient** etc.

Let's understand the access modifiers by a simple table.

| Member Type | Inside own class | Inside subclasses | | Inside non-subclasses | |
|---|---|---|---|---|---|
| | | In the same package | In other package | In the same package | In other package |
| **public** | **Y** | **Y** | **Y** | **Y** | **Y** |
| **protected** | **Y** | **Y** | **Y** | **Y** | **N** |

| | | | | | |
|---|---|---|---|---|---|
| **Default** | **Y** | **Y** | **N** | **Y** | **N** |
| **Private** | **Y** | **N** | **N** | **N** | **N** |

- **Visible to the package the default, No modifiers are needed.**
- **Visible to the class only (private).**
- **Visible to the entire system (public).**
- **Visible to the package and all subclasses (protected).**

## 1. private

The private access modifier is accessible only within class.

**Simple example of private access modifier**

In this example, we have created two classes A and Simple. A class contains private data member and private method. We are accessing these private members from outside the class, so there is compile time error.

```
class A{
private int data=40;
private void msg(){System.out.println("Hello java");}
}

public class Simple{
 public static void main(String args[]){
   A obj=new A();
   System.out.println(obj.data);//Compile Time Error
   obj.msg();//Compile Time Error
   }
}
```

## 2. default

If you don't use any modifier, it is treated as **default** by default. The default modifier is accessible only within package.

### Example of default access modifier

In this example, we have created two packages pack and mypack. We are accessing the A class from outside its package, since A class is not public, so it cannot be accessed from outside the package.

```
//save by A.java

package pack;
class A
{
  void msg()
  {
    System.out.println("Hello");
  }
}

//save by B.java

package mypack;
import pack.*;

class B
{
  public static void main(String args[])
  {
     A obj = new A();//Compile Time Error
     obj.msg();//Compile Time Error
  }
}
```

In the above example, the scope of class A and its method msg() is default so it cannot be accessed from outside the package.

## 3. protected

The **protected access modifier** is accessible within package and outside the package but through inheritance only.
The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.

**Example of protected access modifier**

In this example, we have created the two packages pack and mypack. The A class of pack package is public, so can be accessed from outside the package. But msg method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

```
//save by A.java

package pack;
public class A{
protected void msg(){System.out.println("Hello");}
}
//save by B.java

package mypack;
import pack.*;

class B extends A{
  public static void main(String args[]){
   B obj = new B();
   obj.msg();
  }
}
```

Output:Hello

## 4. public

The **public access modifier** is accessible everywhere. It has the widest scope among all other modifiers.

**Example of public access modifier**

```
//save by A.java

package pack;
public class A{
public void msg(){System.out.println("Hello");}
}
//save by B.java

package mypack;
import pack.*;

class B{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
  }
}
```

Output:Hello

## Static Import:

The static import feature of Java 5 facilitate the java programmer to access any static member of a class directly. There is no need to qualify it by the class name.

**Advantage of static import:**

- Less coding is required if you have access any static member of a class oftenly.

**Disadvantage of static import:**

- If you overuse the static import feature, it makes the program unreadable and un-maintainable.

**Simple Example of static import**

```
import static java.lang.System.*;
class StaticImportExample{
public static void main(String args[]){

out.println("Hello");//Now no need of System.out
out.println("Java");

}
}
```

Output:Hello
Java

**What is the difference between import and static import?**

The import provides accessibility to classes and interface whereas static import provides accessibility to static members of the class.

**Rule:** There can be only one public class in a java source file and it must be saved by the public class name.

```
//save as C.java otherwise Compile Time Error
```

```
class A{}
class B{}
public class C{}
```

**How to put two public classes in a package?**

If you want to put two public classes in a package, have two java source files containing one public class, but keep the package name same. For example:

```
//save as A.java
  package gehu;
public class A{}
//save as B.java

package gehu;
public class B{}
```

**Role of Private Constructor:**

If you make any class constructor private, you cannot create the instance of that class from outside the class. For example:

```
class A{
private A(){}//private constructor

void msg(){System.out.println("Hello java");}
}

public class Simple{
 public static void main(String args[]){
   A obj=new A();//Compile Time Error
 }
}
```

***Note: A class cannot be private or protected except nested class.***

Following table shows what access specifiers may be assigned to different elements. Note that all four levels may be applied to all elements except classes. Classes may be declared with only public and private access specifiers

| | **public** | **private** | **protected** | **< unspecified >** |
|---|---|---|---|---|
| **class** | allowed | not allowed | not allowed | allowed |
| **constructor** | allowed | allowed | allowed | allowed |
| **variable** | allowed | allowed | allowed | allowed |
| **method** | allowed | allowed | allowed | allowed |