**UNIT: 3 INHERITANCE AND PACKAGES**

- Inheritance Basic, Types of Inheritance.
- Uses of 'super' keyword.
- Method Overriding.
- Run Time Polymorphism: Dynamic Method Dispatch.
- Abstract Method and Class.
-'final' Keyword with Inheritance.
- Defining Package, Understanding of CLASSPATH.
- Importing Packages.
- Access Protection

## What is Inheritance?

- It's a fundamental concept in object-oriented programming (OOP) that allows new classes (subclasses) to acquire properties and behaviors from existing classes (superclasses).
- It promotes code reusability, reduces redundancy, and creates a hierarchical structure among classes.
- Inheritance represents the IS-A relationship which is also known as a parent-child relationship.

**Key Terminology:**

- Superclass (Parent Class): The class being inherited from.
- Subclass (Child Class): The class that inherits from the superclass.
- "extends" keyword: Used to create a subclass that inherits from a superclass.

**Why use inheritance in java**

- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.

**Syntax:**

```
class Subclass-name extends Superclass-name
{
  //methods and fields
}
```

**Example**

```
class Animal
{
   public void eat()
   {
      System.out.println("Animal is eating");
   }
}

class Dog extends Animal
{
// Dog inherits from Animal
   public void bark()
   {
      System.out.println("Woof!");
   }
}
```

## Features:

- **Direct Inheritance**: A subclass inherits directly from only one superclass.
- **Inheritance of Members**: The subclass inherits all non-private members (fields and methods) from the superclass.
- **Method Overriding**: The subclass can override inherited methods to provide its own implementation.
- **Accessing Superclass Members**: The super keyword is used to access superclass members from within the subclass.

## Benefits:

- **Code Reusability**: Subclasses can reuse code from the superclass, reducing redundancy.
- **Extensibility**: New features can be added to subclasses while maintaining a common base.
- **Organization**: Helps structure code in a logical, hierarchical manner.
- **Clarity**: Simplifies class relationships and promotes understanding.

## Limitations:

- Java only supports single inheritance for classes, unlike some other OOP languages.
- For more complex inheritance scenarios, consider multilevel inheritance or interfaces.

## Types of Inheritance

1. Single Inheritance
2. Multilevel Inheritance
3. Hierarchical Inheritance
4. Multiple Inheritance
5. Hybrid Inheritance

### 1. Single Inheritance

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

Single Inheritance

## Example

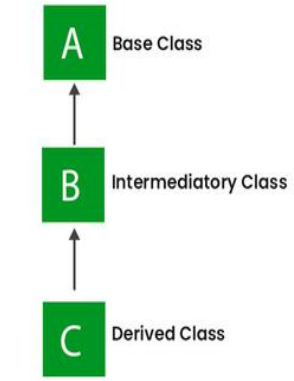
```
// Java program to illustrate the concept of single inheritance
import java.io.*;
import java.lang.*;
import java.util.*;

// Parent class
class One
{
   public void print_geek()
   {
      System.out.println("Geeks");
   }
}

class Two extends One
{
   public void print_for()
    {
        System.out.println("for");
    }
}

// Driver class
public class Main
{
    // Main function
   public static void main(String[] args)
   {
      Two g = new Two();
      g.print_geek();
      g.print_for();
      g.print_geek();
   }
}
```

**Output**

Geeks
for
Geeks

## 2. Multilevel Inheritance

- In Multilevel Inheritance, a derived class will be inheriting a base class, and as well as the derived class also acts as the base class for other classes. In the below image, class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C. In Java, a class cannot directly access the grandparent's members.




Multilevel Inheritance

**Example**

```
// Java program to illustrate the
// concept of Multilevel inheritance
import java.io.*;
import java.lang.*;
import java.util.*;

class One {
   public void print_geek()
   {
      System.out.println("Geeks");
   }
}

class Two extends One {
   public void print_for() { System.out.println("for"); }
}

class Three extends Two {
   public void print_geek()
   {
```

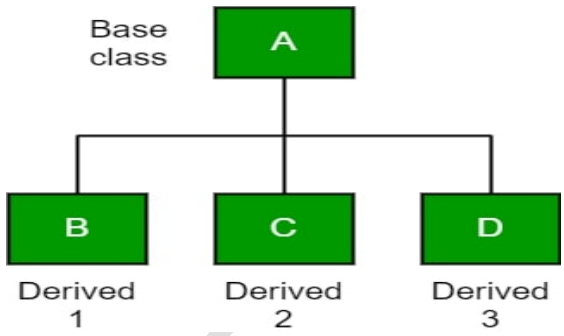
**Output**

```
Geeks
for
Geeks
```

```
        System.out.println("Geeks");
    }
}

// Drived class
public class Main {
    public static void main(String[] args)
    {
        Three g = new Three();
        g.print_geek();
        g.print_for();
        g.print_geek();
    }
}
```

### 3. Hierarchical Inheritance

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




**Example**

```
// Java program to illustrate the concept of Hierarchical  inheritance

class A
{
    public void print_A()
```

```
   {
        System.out.println("Class A");
    }
}

class B extends A
 {
   public void print_B()
   {
       System.out.println("Class B");
   }
}

class C extends A
{
   public void print_C()
   {
         System.out.println("Class C");
   }
}

class D extends A
 {
   public void print_D()
  {
        System.out.println("Class D");
  }
}

// Driver Class
public class Test
 {
   public static void main(String[] args)
   {
      B obj_B = new B();
```

**Output**

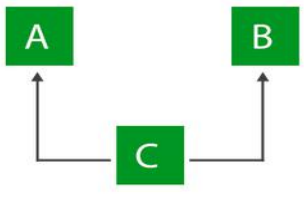
```
Class A
Class B
Class A
Class C
Class A
Class D
```

```
        obj_B.print_A();
        obj_B.print_B();

        C obj_C = new C();
        obj_C.print_A();
        obj_C.print_C();

        D obj_D = new D();
        obj_D.print_A();
        obj_D.print_D();
    }
}
```

Note: Multiple inheritance is not supported in Java through class.

### 4. Multiple Inheritance (Through Interfaces)

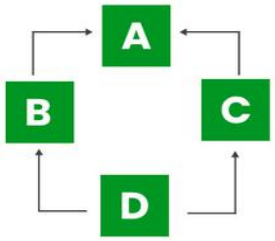
- In Multiple inheritances, one class can have more than one superclass and inherit features from all parent classes.
- Please note that Java does **not** support multiple inheritances with classes. In Java, we can achieve multiple inheritances only through Interfaces. In the image below, Class C is derived from interfaces A and B.




Multiple Inheritance

## 5. Hybrid Inheritance

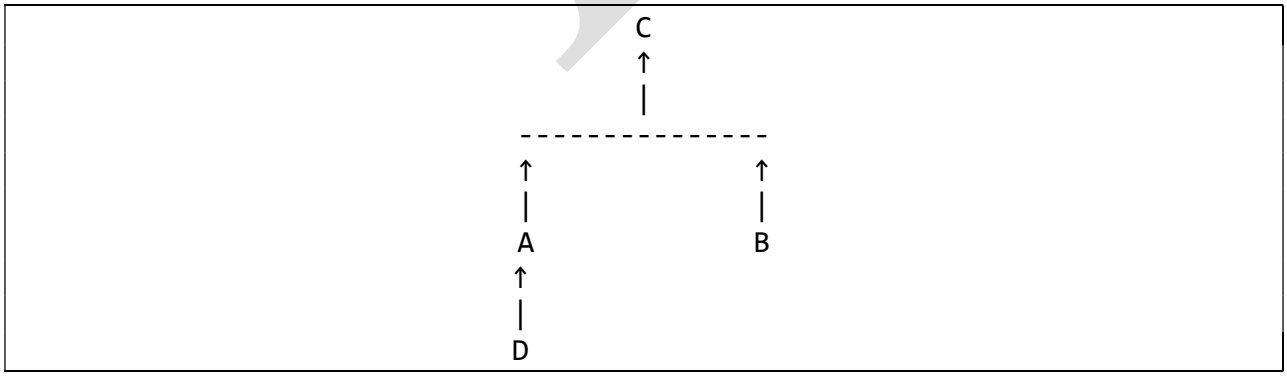
- It is a mix of two or more of the above types of inheritance. Since Java doesn't support multiple inheritances with classes, hybrid inheritance involving multiple inheritance is also not possible with classes. In Java, we can achieve hybrid inheritance only through Interfaces if we want to involve multiple inheritance to implement Hybrid inheritance.
- However, it is important to note that Hybrid inheritance does not necessarily require the use of Multiple Inheritance exclusively. It can be achieved through a combination of Multilevel Inheritance and Hierarchical Inheritance with classes, Hierarchical and Single Inheritance with classes. Therefore, it is indeed possible to implement Hybrid inheritance using classes alone, without relying on multiple inheritance type.



Hybrid Inheritance

**Example**

Lets write this in a program to understand how this works:



This example is just to demonstrate the hybrid inheritance in Java. Although this example is meaningless, you would be able to see that how we have implemented two types of inheritance(single and hierarchical) together to form hybrid inheritance.

**Class A and B extends class C → Hierarchical inheritance**
**Class D extends class A → Single inheritance**

```
class C
{
   public void disp()
   {
     System.out.println("C");
   }
}

class A extends C
{
   public void disp()
   {
     System.out.println("A");
   }
}

class B extends C
{
   public void disp()
   {
     System.out.println("B");
   }

}

class D extends A
{
   public void disp()
   {
     System.out.println("D");
   }
   public static void main(String args[])
```

**Output:**

```
D
```

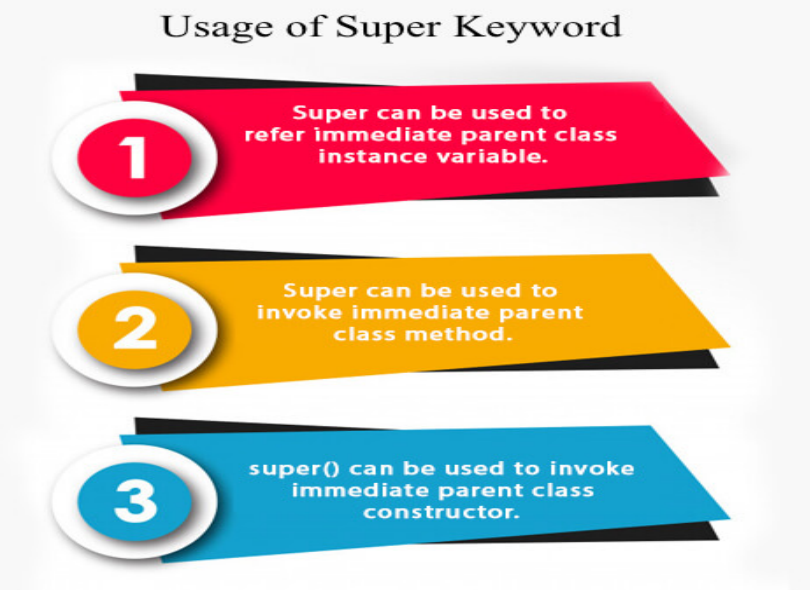
```
    {

        D obj = new D();
        obj.disp();
    }
}
```

## Super Keyword

- The **super** keyword in Java is a reference variable which is used to refer immediate parent class object.
- The super keyword in Java is used in subclasses to access superclass members (attributes, constructors and methods).
- Before we learn about the super keyword, make sure to know about Java inheritance.

### Uses of super keyword

1. super can be used to refer immediate parent class instance variable.
2. super can be used to invoke immediate parent class method.
3. super() can be used to invoke immediate parent class constructor.



Let's understand each of these uses.

## 1) super is used to refer immediate parent class instance variable.

- We can use super keyword to access the data member or field of parent class. It is used if parent class and child class have same fields.

**Example 1: Access superclass attribute**

```
class Animal
{
  protected String type="animal";
}

class Dog extends Animal
{
  public String type="mammal";

  public void printType()
  {
    System.out.println("I am a " + type);
    System.out.println("I am an " + super.type);
  }
}

class Main
{
  public static void main(String[] args)
  {
    Dog dog1 = new Dog();
    dog1.printType();
  }
}
```

**Output**:

```
I am a mammal

I am an animal
```

## 2) super can be used to invoke parent class method

- The super keyword can also be used to invoke parent class method. It should be used if subclass contains the same method as parent class. In other words, it is used if method is overridden.

**Example 1: Method overriding**

```
class Animal
{
  // overridden method
          public void display()
         {
                  System.out.println("I am an animal");
          }
}
class Dog extends Animal
 {
  // overriding method
           public void display()
         {
                   System.out.println("I am a dog");
           }

         public void printMessage()
         {
                  display();
         }
}
class Main
{
  public static void main(String[] args) {
    Dog dog1 = new Dog();
    dog1.printMessage();
  }
}
```

**Output**

I am a dog

**Example 2: super to Call Superclass Method**

```
class Animal {

  // overridden method
  public void display(){
    System.out.println("I am an animal");
  }
}

class Dog extends Animal {

  // overriding method
  @Override
  public void display(){
    System.out.println("I am a dog");
  }

  public void printMessage(){

    // this calls overriding method
    display();

    // this calls overridden method
    super.display();
  }
}

class Main {
  public static void main(String[] args) {
    Dog dog1 = new Dog();
    dog1.printMessage();
  }
}
```

**Output**

```
I am a dog
I am an animal
```

## 3) super is used to invoke parent class constructor.

- The super keyword can also be used to invoke the parent class constructor.

**Example :**

```
class Animal {

  // default or no-arg constructor of class Animal
  Animal() {
    System.out.println("I am an animal");
  }
}

class Dog extends Animal {

  // default or no-arg constructor of class Dog
  Dog() {

    // calling default constructor of the superclass
    super();

    System.out.println("I am a dog");
  }
}

class Main {
  public static void main(String[] args) {
    Dog dog1 = new Dog();
  }
}
```

**Output**

```
I am an animal
I am a dog
```

Here, when an object `dog1` of `Dog` class is created, it automatically calls the default or no-arg constructor of that class.

Inside the subclass constructor, the `super()` statement calls the constructor of the superclass and executes the statements inside it. Hence, we get the output `I am an animal`.

**Example : Call Parameterized Constructor Using super()**

```
class Animal {

  // default or no-arg constructor
  Animal() {
    System.out.println("I am an animal");
  }

  // parameterized constructor
  Animal(String type) {
    System.out.println("Type: "+type);
  }
}

class Dog extends Animal {

  // default constructor
  Dog() {

    // calling parameterized constructor of the superclass
    super("Animal");

    System.out.println("I am a dog");
  }
}

class Main {
  public static void main(String[] args) {
    Dog dog1 = new Dog();
  }
}
```

**Output**

```
Type: Animal
I am a dog
```

- The compiler can automatically call the no-arg constructor. However, it cannot call parameterized constructors.
- If a parameterized constructor has to be called, we need to explicitly define it in the subclass constructor.

## Method Overriding in Java

- If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java**.
- In other words, If a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding.

### Usage of Java Method Overriding

- Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
- Method overriding is used for runtime polymorphism

### Rules for Java Method Overriding

1. The method must have the same name as in the parent class
2. The method must have the same parameter as in the parent class.
3. There must be an IS-A relationship (inheritance).

### Understanding the problem without method overriding

- Let's understand the problem that we may face in the program if we don't use method overriding.

```
//Java Program to demonstrate why we need method overriding
class Vehicle{
  void run(){System.out.println("Vehicle is running");}
}
class Bike extends Vehicle{
  public static void main(String args[]){
  //creating an instance of child class
  Bike obj = new Bike();
  //calling the method with child class instance
  obj.run();
  }
}
```

**Output:**

```
Vehicle is running
```

- **Problem is that I have to provide a specific implementation of run() method in subclass that is why we use method overriding.**

## Example of method overriding

- In this example, we have defined the run method in the subclass as defined in the parent class but it has some specific implementation. The name and parameter of the method are the same, and there is IS-A relationship between the classes, so there is method overriding.

```
//Java Program to illustrate the use of Java Method Overriding
//Creating a parent class.
class Vehicle
{
  //defining a method
  void run()
  {
       System.out.println("Vehicle is running");
  }
}
//Creating a child class
class Bike2 extends Vehicle
{
  //defining the same method as in the parent class
  void run()
  {
       System.out.println("Bike is running safely");
  }
  public static void main(String args[])
 {
  Bike2 obj = new Bike2();//creating object
  obj.run();//calling method
  }
}
```

**Output:**

```
Bike is running safely
```

## Polymorphism

- Polymorphism in Java can be done in two ways, method overloading and method overriding. There are two types of polymorphism in Java.

    1. Compile-time polymorphism
    2. Runtime polymorphism.

- Compile-time polymorphism is a process in which a call to an overridden method is resolved at compile time.

### Purpose of Polymorphism in OOPs

- The primary purpose of polymorphism is to perform a single action in multiple ways. In other words, polymorphism provides one interface with many implementations.

- Poly means many and morphs means forms. True to its name, polymorphism offers different forms to a single function.

### Run Time Polymorphism: Dynamic Method Dispatch.

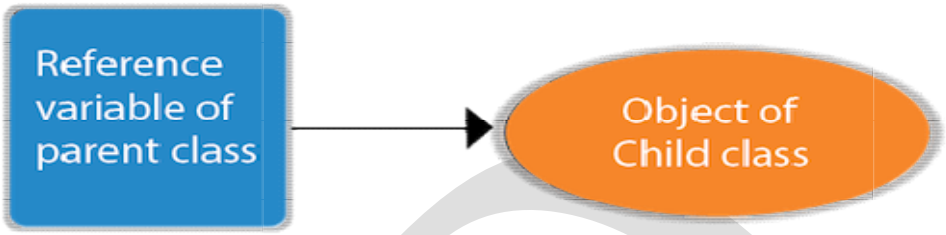
- Runtime polymorphism, also known as the Dynamic Method Dispatch, is a process that resolves a call to an overridden method at runtime. The process involves the use of the reference variable of a superclass to call for an overridden method.

- The determination of the method to be called is based on the object being referred to by the reference variable.

- Let's first understand the upcasting before Runtime Polymorphism.

## Upcasting

- If the reference variable of Parent class refers to the object of Child class, it is known as upcasting. For example:




```
class A{}
class B extends A{}
A a=new B();//upcasting
```

- For upcasting, we can use the reference variable of class type or an interface type. For Example:

```
interface I{}
class A{}
class B extends A implements I{}
```

## Example of Java Runtime Polymorphism

- In this example, we are creating two classes Bike and Splendor. Splendor class extends Bike class and overrides its run() method. We are calling the run method by the reference variable of Parent class. Since it refers to the subclass object and subclass method overrides the Parent class method, the subclass method is invoked at runtime.

- Since method invocation is determined by the JVM not compiler, it is known as runtime polymorphism.

```
class Bike
{
  void run()
  {
     System.out.println("running");
  }
}
class Splendor extends Bike
{
  void run()
   {
     System.out.println("running safely with 60km");
   }
  public static void main(String args[])
  {
    Bike b = new Splendor();//upcasting
    b.run();
  }
}
```

**Output:**

running safely with 60km.

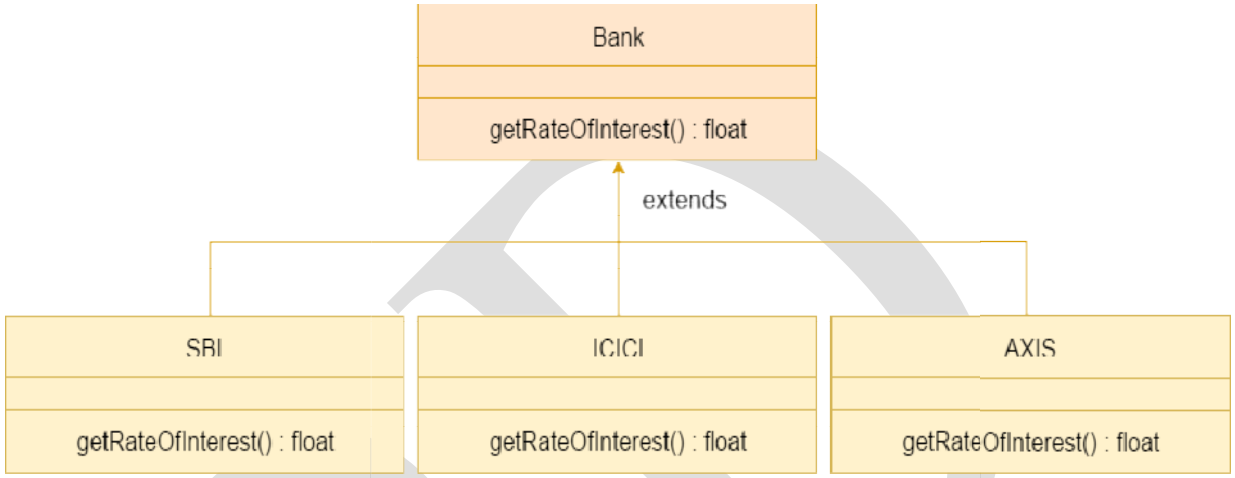
```
class Animal {
  public void move() {
    System.out.println("Animals can move");
  }
}
class Dog extends Animal {
  public void move() {
    System.out.println("Dogs can walk and run");
  }
}
public class TestDog {

  public static void main(String args[]) {

    Animal a = new Animal(); // Animal reference and object
    Animal b = new Dog(); // Animal reference but Dog object

    a.move(); // runs the method in Animal class
    b.move(); // runs the method in Dog class
  }
}
```

**Output**

Animals can move
Dogs can walk and run

## Java Runtime Polymorphism Example: Bank

- Consider a scenario where Bank is a class that provides a method to get the rate of interest. However, the rate of interest may differ according to banks. For example, SBI, ICICI, and AXIS banks are providing 8.4%, 7.3%, and 9.7% rate of interest.




Note: This example is also given in method overriding but there was no upcasting.

```
class Bank
{
        float getRateOfInterest()
        {
                return 0;
        }
}
class SBI extends Bank
{
        float getRateOfInterest()
        {
                return 8.4f;
        }
}
class ICICI extends Bank
{
```

```
        float getRateOfInterest()
        {
                return 7.3f;
        }
}
class AXIS extends Bank
{
        float getRateOfInterest()
        {
                return 9.7f;
        }
}
class TestPolymorphism
{
        public static void main(String args[])
        {
                Bank b;
                b=new SBI();
                System.out.println("SBI Rate of Interest: "+b.getRateOfInterest());

                b=new ICICI();
                System.out.println("ICICI Rate of Interest: "+b.getRateOfInterest());

                b=new AXIS();
                System.out.println("AXIS Rate of Interest: "+b.getRateOfInterest());
        }
}
```

**Output:**

SBI Rate of Interest: 8.4
ICICI Rate of Interest: 7.3
AXIS Rate of Interest: 9.7

## Java Runtime Polymorphism Example: Shape

```
class Shape
{
        void draw()
        {
                System.out.println("drawing...");
        }
}
class Rectangle extends Shape
{
        void draw()
        {
                System.out.println("drawing rectangle...");
        }
}
class Circle extends Shape
{
        void draw()
        {
                System.out.println("drawing circle...");
        }
}
class Triangle extends Shape
{
        void draw()
        {
                System.out.println("drawing triangle...");
        }
}
class TestPolymorphism2
{
    public static void main(String args[])
    {
        Shape s;
        s=new Rectangle();
```

```
            s.draw();
            s=new Circle();
            s.draw();
            s=new Triangle();
            s.draw();
    }
}
```

**Output:**

```
drawing rectangle...
drawing circle...
drawing triangle...
```

## Abstraction

- **Abstraction** is a process of hiding the implementation details and showing only functionality to the user.
- Another way, it shows only essential things to the user and hides the internal details, for example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.
- Abstraction lets you focus on what the object does instead of how it does it.

### Ways to achieve Abstraction

There are two ways to achieve abstraction in java

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

### Abstract class in Java

- A class which is declared as abstract is known as an abstract class. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.

### Points to Remember

- An abstract class must be declared with an abstract keyword.
- It can have abstract and non-abstract methods.
- It cannot be instantiated.

- It can have constructors and static methods also.
- It can have final methods which will force the subclass not to change the body of the method.

## Example of abstract class

```
abstract class A{}
```

## Example of abstract method

```
abstract void printStatus();//no method body and abstract
```

## Example of Abstract class that has an abstract method

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

```
abstract class Bike
{
  abstract void run();
}
class Honda4 extends Bike
{
     void run()
    {
            System.out.println("running safely");
    }
    public static void main(String args[])
   {
            Bike obj = new Honda4();
            obj.run();
    }
}
```

**Output**

```
running safely
```

**Example**

```
abstract class Bank
{
    abstract int getRateOfInterest();
}
class SBI extends Bank
{
    int getRateOfInterest(){return 7;}
}
class PNB extends Bank
{
    int getRateOfInterest(){return 8;}
}

class TestBank
{
    public static void main(String args[])
    {
            Bank b;
            b=new SBI();
            System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
            b=new PNB();
            System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
    }
}
```

**Output**

```
Rate of Interest is: 7 %
Rate of Interest is: 8 %
```

## Abstract class having constructor, data member and methods

- An abstract class can have a data member, abstract method, method body (non-abstract method), constructor, and even main() method.

**File: TestAbstraction2.java**

```
//Example of an abstract class that has abstract and non-abstract methods

abstract class Bike
{
  Bike()
  {
              System.out.println("bike is created");
  }
  abstract void run();
  void changeGear()
 {
    System.out.println("gear changed");
 }
}

//Creating a Child class which inherits Abstract class
 class Honda extends Bike
{
  void run()
  {
     System.out.println("running safely..");
  }
 }

//Creating a Test class which calls abstract and non-abstract methods
 class TestAbstraction2
{
    public static void main(String args[])
   {
```

**Output**

```
bike is created
running safely..
gear changed
```

```
    Bike obj = new Honda();
    obj.run();
    obj.changeGear();
 }
}
```

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

```
class Bike12
{
abstract void run();
}
```

compile time error

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

## Another real scenario of abstract class

- The abstract class can also be used to provide some implementation of the interface. In such case, the end user may not be forced to override all the methods of the interface.

**Note: If you are beginner to java, learn interface first and skip this example.**

```
interface A
{
    void a();
    void b();
    void c();
    void d();
}

abstract class B implements A
{
    public void c()
    {
```

```
            System.out.println("I am c");
    }
}

class M extends B
{
    public void a()
    {
            System.out.println("I am a");
    }
    public void b()
    {
            System.out.println("I am b");
    }
    public void d()
    {
            System.out.println("I am d");
    }
}

class Test5
{
    public static void main(String args[])
    {
            A a=new M();
            a.a();
            a.b();
            a.c();
            a.d();
    }
}
```

**Output:**

```
I am a
I am b
I am c
I am d
```

## 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

Consider the below table to understand where we can use the final keyword:

| Type | Description |
|---|---|
| **Final Variable** | Variable with final keyword cannot be assigned again |
| **Final Method** | Method with final keyword cannot be overridden by its subclasses |
| **Final Class** | Class with final keywords cannot be extended or inherited from other classes |

## 1) Java final variable

- If you make any variable as final, you cannot change the value of final variable(It will be constant).

**For example,**

```
class Main
{
      public static void main(String[] args)
     {
                  // create a final variable
                final int AGE = 32;

   // try to change the final variable
   AGE = 45;
   System.out.println("Age: " + AGE);
  }
}
```

- In the above program, we have created a final variable named age. And we have tried to change the value of the final variable.
- When we run the program, we will get a compilation error with the following message.

```
cannot assign a value to final variable AGE
AGE = 45;
```

## 2) Java final method

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

**For example,**

```
class FinalDemo
{
   // create a final method
   public final void display()
   {
     System.out.println("This is a final method.");
   }
}
class Main extends FinalDemo
{
  // try to override final method
  public final void display()
   {
    System.out.println("The final method is overridden.");
  }

  public static void main(String[] args)
   {
    Main obj = new Main();
    obj.display();
  }
}
```

In the above example, we have created a final method named `display()` inside the `FinalDemo` class.

Here, the `Main` class inherits the `FinalDemo` class.

We have tried to override the final method in the `Main` class. When we run the program, we will get a compilation error with the following message.

```
display() in Main cannot override display() in FinalDemo
  public final void display() {
                ^
  overridden method is final
```

## 3. Java final Class

- In Java, the final class cannot be inherited by another class. For example,

```
// create a final class
final class FinalClass
{
  public void display()
   {
    System.out.println("This is a final method.");
  }
}
// try to extend the final class
class Main extends FinalClass
{
  public  void display()
   {
    System.out.println("The final method is overridden.");
  }
  public static void main(String[] args) {
    Main obj = new Main();
    obj.display();
  }
}
```

In the above example, we have created a final class named FinalClass. Here, we have tried to inherit the final class by the Main class.

When we run the program, we will get a compilation error with the following message.

```
cannot inherit from final FinalClass
class Main extends FinalClass {
```

## Java Package

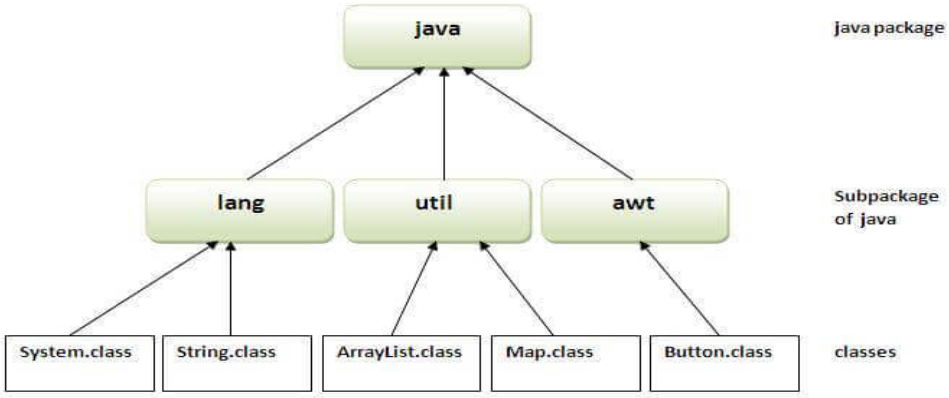
- A java package is a group of similar types of classes, interfaces and sub-packages.
- Package in java can be categorized in two form, built-in package and user-defined package.
- There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.
- Here, we will have the detailed learning of creating and using user-defined packages.

### Advantage of 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.



**Simple example of java package**

- The package keyword is used to create a package in java.

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

**How to compile java package**

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

**javac -d directory javafilename**

For **example**

**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 java package program**

- 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 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.

## 1) 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.*

```
//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**

## 2) 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

```
//save by A.java

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

```
//save by B.java
package mypack;
import pack.A;

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

**Output:**
Hello

## 3) 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**

```
//save by A.java
package pack;
public class A
{
  public void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
class B
{
  public static void main(String args[])
  {
   pack.A obj = new pack.A();//using fully qualified name
   obj.msg();
  }
}
```

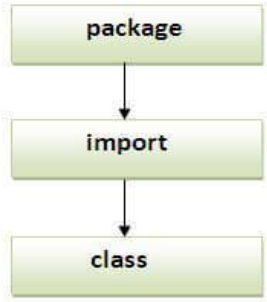
**Output:**

Hello

**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.

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




## Subpackage in java

- 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.

> The standard of defining package is domain.company.package e.g. com.javatpoint.bean or org.sssit.dao.

## Example of Subpackage

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

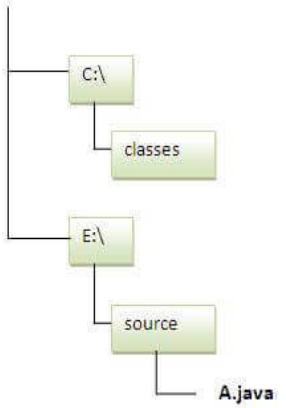
**Compile:** javac -d . Simple.java

**Run:** java com.javatpoint.core.Simple

**Output**: `Hello subpackage`

## How to send the class file to another directory or drive?

- There is a scenario, I want to put the class file of A.java source file in classes folder of c: drive. For example:




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

**To Compile**: e:\sources> javac -d c:\classes Simple.java

**To Run**:

To run this program from e:\source directory, you need to set classpath of the directory where the class file resides.

**e:\sources> set classpath=c:\classes;.;**

**e:\sources> java mypack.Simple**

## Another way to run this program by -classpath switch of java:

- The -classpath switch can be used with javac and java tool.
- To run this program from e:\source directory, you can use -classpath switch of java that tells where to look for class file. For example:

**e:\sources> java -classpath c:\classes mypack.Simple**

**Output :** Welcome to package

## Ways to load the class files or jar files

- There are two ways to load the class files temporary and permanent.
  - Temporary
    - By setting the classpath in the command prompt
    - By -classpath switch
  - Permanent
    - By setting the classpath in the environment variables
    - By creating the jar file, that contains all the class files, and copying the jar file in the jre/lib/ext folder.

*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 Compilte 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 javatpoint;
public class A{}
//save as B.java

package javatpoint;
public class B{}
```

## Accessibility of Packages in Java

- As you may know, in Java, we do everything inside classes. So, limiting access to variables and methods by different files must be handled for security purposes. We mainly focus on package-level accessibility here, but you can check out the table below to know more.

Note: **Java has four modifiers: public, protected, private, and default.**

- Except for private members of a class, all other members can be accessed by all other classes defined inside the same package. Sub-classes defined inside the different packages only access to public and protected members.
- Here's the table to get more clarity.

| Access Modifier | Same class | Same package subclass | Same package non-subclass | Difference Package subclass | Different package non-subclass |
|---|---|---|---|---|---|
| Default | Yes | Yes | Yes | No | No |
| Private | Yes | No | No | No | No |
| Protected | Yes | Yes | Yes | Yes | No |
| Public | Yes | Yes | Yes | Yes | Yes |

- From the above table, you can infer whether data members of a class with certain modifiers can be accessed by:
  1. The same class in which these data members are defined
  2. The classes available inside the same package
  3. The sub-class of this Java file that is stored outside this package
  4. All other classes are available anywhere inside the system.

Let's understand them in more detail.

**Same class**

- If we make a class named A and define data members inside it with any modifiers (public, protected, default, and private), then we can access them inside class A.

**Same Package**

- We have single or multiple classes inside a package. Let's say we've two classes inside a package named A and B. If we try to import class A inside class B, then except for private data members of class A all others will be accessible to class B.

**Same Package Sub-class**

- Let's create two classes Class A and B where Class B inherits from Class A. In such a scenario, Class B can access all members of Class A except private members. It can access the default and protected members as it belongs to the same package.

**Different Package Sub-class**

- Say you have two packages, packageA, and packageB. The packageA contains class A and packageB contains class B. Class B is a subclass of class A but since they both are in different packages, that's why class B can access only public and protected members of class A, default and private members will be hidden.

**Global**

- Let's say a class B inside a packageB wants to access class A of packageA. Now, class B is **NOT** a subclass of class A. In this case, only public members of class A will be accessible to class B.

## How to Set CLASSPATH in Java

- **CLASSPATH:** CLASSPATH is an environment variable which is used by Application ClassLoader to locate and load the .class files. The CLASSPATH defines the path, to find third-party and user-defined classes that are not extensions or part of Java platform. Include all the directories which contain .class files and JAR files when setting the CLASSPATH.

  You need to set the CLASSPATH if:

  - You need to load a class that is not present in the current directory or any sub-directories.
  - You need to load a class that is not in a location specified by the extensions mechanism.

- The CLASSPATH depends on what you are setting the CLASSPATH. The CLASSPATH has a directory name or file name at the end. The following points describe what should be the end of the CLASSPATH.
  - If a JAR or zip, the file contains class files, the CLASSPATH end with the name of the zip or JAR file.
  - If class files placed in an unnamed package, the CLASSPATH ends with the directory that contains the class files.
  - If class files placed in a named package, the CLASSPATH ends with the directory that contains the root package in the full package name, that is the first package in the full package name.

- The default value of CLASSPATH is a dot (.). It means the only current directory searched. The default value of CLASSPATH overrides when you set the CLASSPATH variable or using the -classpath command (for short -cp). Put a dot (.) in the new setting if you want to include the current directory in the search path.

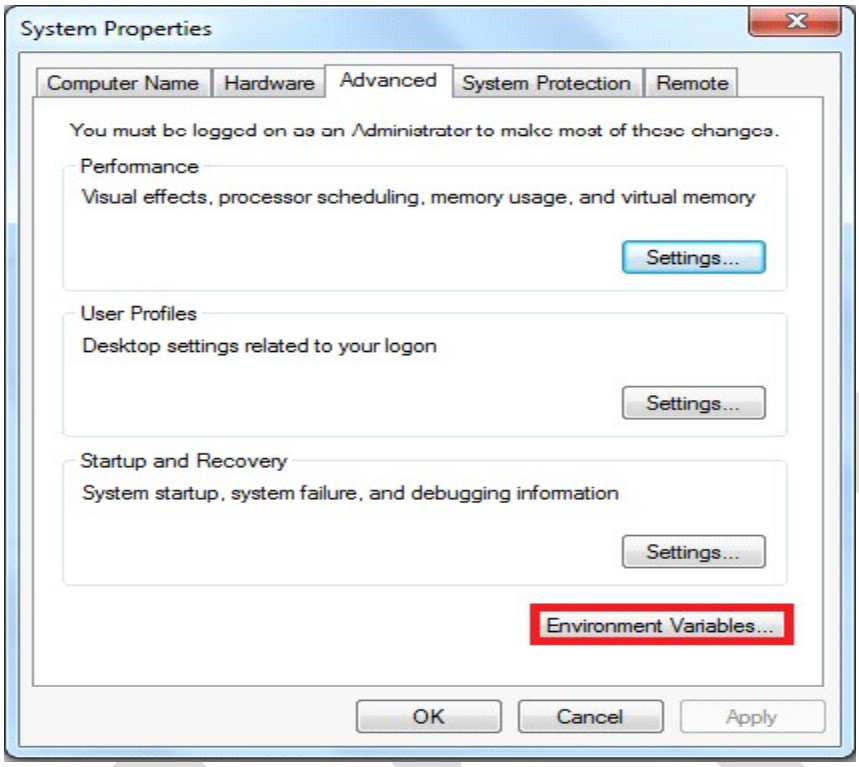
- If CLASSPATH finds a class file which is present in the current directory, then it will load the class and use it, irrespective of the same name class presents in another directory which is also included in the CLASSPATH.
- If you want to set multiple classpaths, then you need to separate each CLASSPATH by a semicolon (;).
- The third-party applications (MySQL and Oracle) that use the JVM can modify the CLASSPATH environment variable to include the libraries they use. The classes can be stored in directories or archives files. The classes of the Java platform are stored in rt.jar.
- There are two ways to ways to set CLASSPATH: through Command Prompt or by setting Environment Variable.

Let's see how to set CLASSPATH of MySQL database:

**Step 1:** Click on the Windows button and choose Control Panel. Select System.


**Step 2:** Click on **Advanced System Settings**.

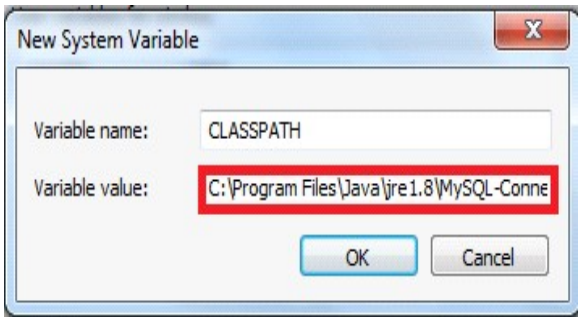
**Step 3:** A dialog box will open. Click on Environment Variables.




**Step 4:** If the CLASSPATH already exists in System Variables, click on the Edit button then put a semicolon (;) at the end. Paste the Path of MySQL-Connector Java.jar file.

- If the CLASSPATH doesn't exist in System Variables, then click on the New button and type Variable name as CLASSPATH and Variable value as *C:\Program Files\Java\jre1.8\MySQL-Connector Java.jar;.;*

Remember: Put ;.; at the end of the CLASSPATH.



## Difference between PATH and CLASSPATH

| PATH | CLASSPATH |
|---|---|
| PATH is an environment variable. | CLASSPATH is also an environment variable. |
| It is used by the operating system to find the executable files (.exe). | It is used by Application ClassLoader to locate the .class file. |
| You are required to include the directory which contains .exe files. | You are required to include all the directories which contain .class and JAR files. |
| PATH environment variable once set, cannot be overridden. | The CLASSPATH environment variable can be overridden by using the command line option -cp or -CLASSPATH to both javac and java command. |