Unit-II

Topics

- INHERITANCE AND POLYMORPHISM: Basic concepts, Types of inheritance, Member access rules, Usage of this and Super key word, Method Overloading, Method overriding, Abstract classes, Dynamic method dispatch, Usage of final keyword.
- PACKAGES AND INTERFACES: Defining package, Access protection, importing packages, Defining and Implementing interfaces, and Extending interfaces

Types of inheritance

What is Inheritance ?:

- The process of acquiring properties and behaviors from one class to another class is called Inheritance.

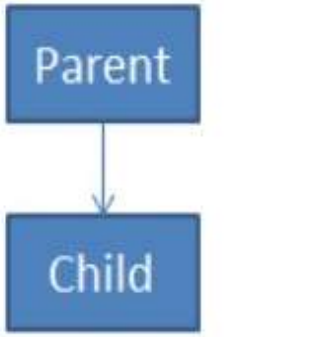
Properties	:	variables
Behaviours	:	methods

- The main purpose of the inheritance is code extensibility whenever we are extending automatically the code is **reused**.

- By using **extends** keyword we are achieving inheritance concept.

- Inheritance is also known as **is-a** relationship means two classes are belongs to the same hierarchy.

- In inheritance one class giving the properties and behavior and another class is taking the properties and behavior.
- In the inheritance the person who is giving the properties is called **parent ,**The person who is taking the properties is called **child**.



Types of Inheritance:

- ❖Single Inheritance
- ❖Multilevel Inheritance
- ❖Hierarchical Inheritance
- ❖Hybrid Inheritance
- ❖Multiple Inheritance

Single inheritance

- In single inheritance, subclasses inherit the features of one superclass.

- class A serves as a base class for the derived class B.

Multilevel Inheritance

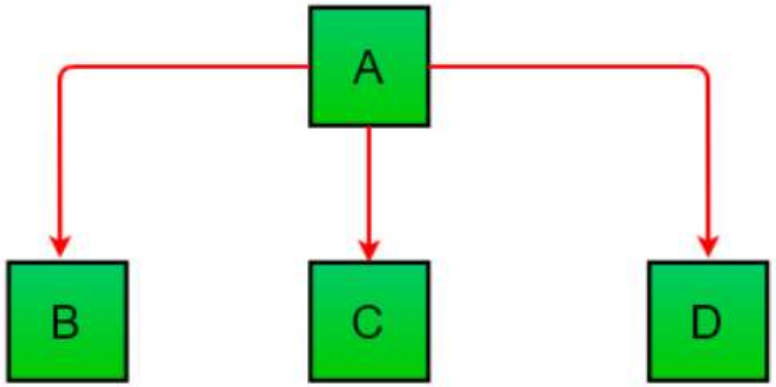
- In Multilevel Inheritance, a derived class will be inheriting a base class and as well as the derived class also act as the base class to other class.

Ex:

Hierarchical Inheritance

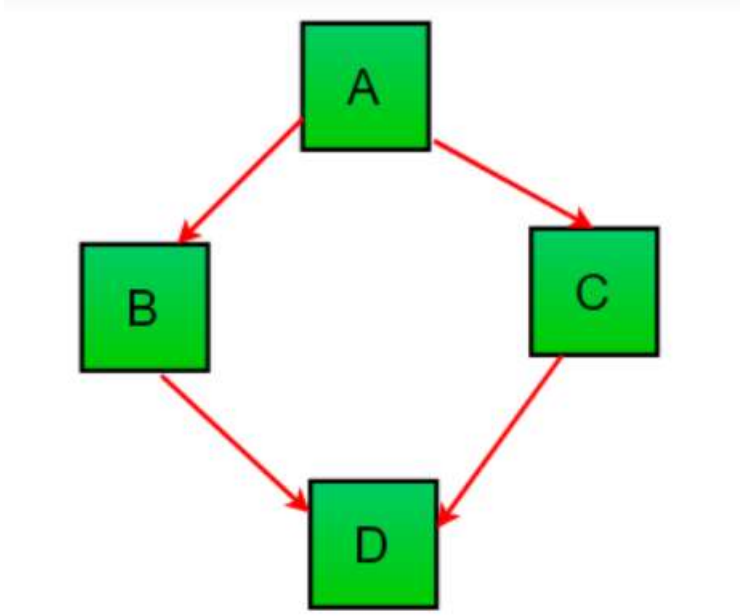
- In Hierarchical Inheritance, one class serves as a superclass for more than one sub class.

Ex:



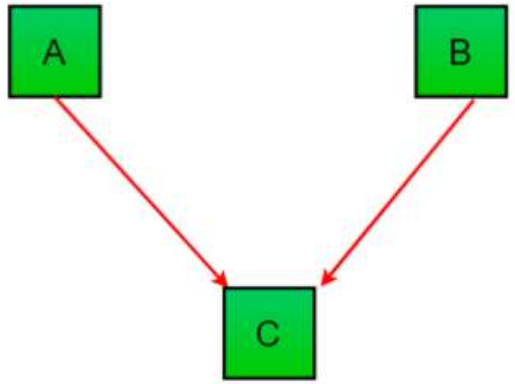
Hybrid Inheritance

- It is a mix of two or more of the above types of inheritance.
- Since java doesn’t support multiple inheritance with classes, the hybrid inheritance is also not possible with classes.



Multiple Inheritance

- In Multiple inheritance ,one class can have more than one parent class and inherit features from all parent classes.



- Java does **not** support **multiple inheritance** with classes.

Object class

- Every class in the java programming is a child class of **Object**.
- The base class for all java classes is **Object** class.
- The default package in the java programming is **java.lang** package.

Ex:

```
class Test
{
}
```

=====>The above class declaration is equal to below one

```
class Test extends Object
{
}
```

Member access rules

Member access rules

public:

- ❖Members with the public access modifier are accessible from anywhere, both within the class hierarchy and from outside the class.
- ❖ Public members of a superclass are inherited and can be accessed in the subclass.

protected:

- ❖Members with the protected access modifier are accessible within the same package and by subclasses, even if they are in a different package.
- ❖Protected members of a superclass are inherited by the subclass.

default:

- If no access modifier is specified (default access), members are accessible within the same package but not outside of it.
- Default members of a superclass are inherited by the subclass if they are in the same package.

private:

- Members with the private access modifier are only accessible within the class where they are declared.
- Private members of a superclass are not inherited by the subclass

this and super key word

this Keyword

this

- In Java, the "this" keyword is a reference to the current object(instance) of the class in which it is used.
- It can be used to refer to instance variables and instance methods of the current object within that object's scope.
- Here are a few common use cases for the "this" keyword in Java:
  - Accessing instance variables
  - Calling another constructor in the same class
  - Passing this for current object as an argument

Accessing instance variables using this

- We can use "this" to distinguish between instance variables and method parameters or local variables when they have the same name.

Ex:

```
class Student
{
    String name;
    String mobile;
    Student(String name,String mobile)
    {
        this.name=name;
        this.mobile=mobile;
    }
}
```

Calling another constructor in the same class using this

- When a class has multiple constructors, you can use "this" to call another constructor from the same class. This is useful for constructor overloading and code reuse.

```
class Employee
{
    private int value;

    public Employee() {
        this(0); // Calls the parameterized constructor with an
initial value of 0
    }

    public Employee(int value) {
        this.value = value;
    }
}
```

Passing this for current object as an argument

➢"this" can be used to pass the current object as an argument to a method or another constructor.

Ex:

```
public class Test
{
    int i=10;
    public static void main(String[] args) {
        Test t=new Test();
        t.m1();
    }
    public void m1()
    {
        System.out.println("m1- method is calling..");
       m2(this);
    }
    public void m2(Test t)
    {
        int i=100;
        System.out.println("Local var     i="+i);
        System.out.println("Instance var i="+t.i);

    }
}
```

super Keyword

super is a keyword in Java which is used to

- Call the Super class **variable**
- Call the super class **methods**
- Call the super class **constructor**
- The most common use of the super keyword is to eliminate the confusion and ambiguity between super classes and subclasses that have methods , Variables with the same name.

Calling Super class variable:

- If the child class and parent class has same data members (Variables). In that case there is a possibility of ambiguity for the JVM.

- To avoid above ambiguity we should use super keyword inside child class

Example:

```
class Parent
{
    int X=100;
    int Y=200;
}
class Child extends Parent
{
    int X=300;
    int Y=400;
    public void add()
    {
        System.out.println("Parent class value ="+(super.X+super.Y));
        System.out.println("Child class value ="+(X+Y));
    }
}
class SuperDemo
{
    public static void main(String[] args) {
        Child c=new Child();
        c.add();
    }
}
```

Calling super class methods

- If the child class and parent class has same data method names. In that case there is a possibility of ambiguity for the JVM.

- To avoid above ambiguity we should use super keyword inside child class

Calling super class methods:

```
class Parent
{
    int X=100;
    int Y=200;
    public int  add()
    {
        return X+Y;
    }
}
class Child extends Parent
{
    int X=300;
    int Y=400;
    public int  add()
    {
        return X+Y;
    }
    public void printResult()
    {
        System.out.println("Parent class vlaue:"+super.add());//Super method to call a method
        System.out.println("Child class vlaue:"+add());
    }
}
class Test
{
    public static void main(String[] args) {
        Child c=new Child();
        c.printResult();
    }
}
```

Calling the super class constructor

- **super()** method is used to call Super class(Parent class) constructor.
- **super()** method call must be inside constructor only. No other method is allowed to call.
- **super()** must be **first statement** of the constructor.
- Both this(), super() methods must be call inside the Constructor. But at a time only one is allowed to call either this() or super().

super() method is used to call Super class(Parent class) constructor.

```
public class Parent
{
    public Parent()
    {
        System.out.println("Parent class constructor:");
    }
}
class Child extends Parent
{
    public Child()
    {
        super();//calling parent class constructor
    }
    public static void main(String[] args) {
        Child c=new Child();
    }
}
```

super() must be first statement of the constructor.

```
class Child extends Parent
{
    public Child()
    {
        System.out.println("Hello..");
        super();
    }
    public static void main(String[] args) {
        Child c=new Child();
    }
}
```

super() method call must be inside constructor only. No other method is allowed to call.

```
class Child extends Parent
{
    public Child()
    {

    }
      public void m1()
      {
              super();

      }

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

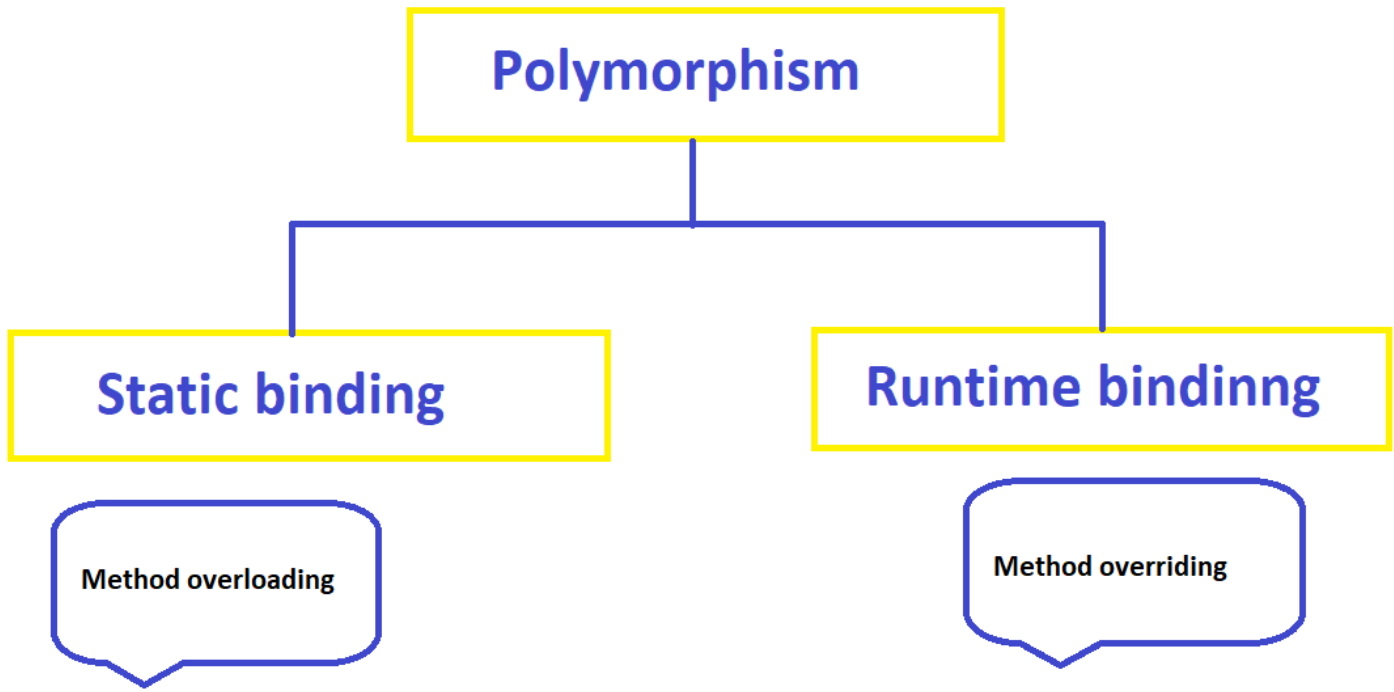
Both this(), super() methods must be call inside the Constructor. But at a time only one is allowed to call either this() or super().

```
class Child extends Parent
{
    public Child()
    {
        this();
        super();
    }
    public static void main(String[] args) {
        Child c=new Child();
    }
}
```

Polymorphism

Polymorphism?

- Poly=Many
- Morph=forms
- **Polymorphism** means "many forms", Performing single task in many ways.


Polymorphism
Static binding
Runtime bindinng
Method overloading
Method overriding

Method Overloading

Method overloading:

- It is a process of rewriting a method with different signatures within the same class is called Method overloading.

- Two methods are said to be overloaded methods if and only if two methods are having same name but different argument list.

- Method overloading can be done in one class.

- We can overload any number of methods

- The compiler does not consider return type when differentiating methods, so you cannot declare two methods with the same signature even if they have a different return type.

- Method resolution takes care by the Compiler.

- This process also called Compile time polymorphism (or) Static binding (or) Early binding.

Ex:

```
public class MethodOverloading
{
    public int getSum(int a,int b)
       {
        return a+b;
       }

      public float getSum(float a,float b)
       {
        return a+b;
       }
}
```

Method overriding

Method overriding:

- Rewriting a parent class method in child class is called Method overriding.

- Overriding concept is possible if and only if two classes should be in Parent – Child relationship.

- In Method overriding **JVM** is the responsible for Method resolution based on object type.

- This process also called as Dynamic binding (or) Late binding (or) Dynamic polymorphism.

Rules for overriding

- In method overriding method name and method signature should be same.

- **Return type:** Must be same (optionally covariant type)

- **Modifier :** Method scope should not decrease

- Scope order as follows : **public>protected>default>private**

- Possible cases:
  - public → public
  - protected → protected, public
  - default → default, protected, public
  - private → ✗ **(Private methods are can not be override)**

- **Static methods**: For static Methods overriding concept is not applicable. But Method hiding is possible.
- In Method hiding compiler is the responsible for method resolution.

Ex: Method hiding

```
class Parent
{
     public static void m1()
    {
        System.out.println("Parent class method");
    }

}
class Child extends Parent
{
    public   static void m1()
    {
        System.out.println("Child class method");
    }
}
```

- **final** Methods: final Methods cannot be overridden..

- final → final ,Non final (Overriding is not possible)
- Non final → final (Overriding is possible)

Dynamic method dispatch

Dynamic method dispatch

Usage of final keyword

- **final is a** keyword or **modifier** applicable to
  - variables (for all instance, Static and local variables).
  - methods
  - classes

final Variables:

- If a variable is declared with ***final*** keyword, its value can't be modified, and we can make a variable as a constant.
- We must initialize a final variable, otherwise compiler will generates compile-time error.
- As per Java documentation final variables naming convention should be in all uppercase, use underscore to separate words.

Ex: public static final double PI=3.141592653589793

Initializing a final variable :

There are three ways to initialize a final variable :

- You can initialize a final variable when it is declared. This approach is the most common
- A blank final variable can be initialized inside instance block or inside constructor. If you have more than one constructor in your class then it must be initialized in all of them, otherwise compile time error will be thrown.
- A blank final static variable can be initialized inside static block.

Ex:

```
public class FinalVariable
{
    public final double PI=3.141592653589793;//Final Variable
                                              initialization at declaration
    public final int I;
    public final int J;
    public static final int DATABASE_VERSION;
    FinalVariable()
    {
        I=10;// Final Variable initialization inside constructor block
    }
    {
        J=20;// Final Variable initialization inside instance block
    }
   static  {
        DATABASE_VERSION=2;// Final Variable initialization
                           inside static block

    }
}
```

final Class

- When a class is declared with ***final*** keyword, it is called a final class.

- If a class is declared as final, then we cannot inherit that class i.e., we cannot create any child class for that final class.

  Ex: You can not create child class to the **String** class . Because String is the **final class.**

Ex:

```
final class Parent
{

}
class Child  extends Parent
{
    //Child class is not possible
    //Compile time error
}
```

- **Every method present inside a final class is always final by default but every variable present inside the final class need not be final.**

Example:

```
final class Demo
{
    int a=10;
    void m1()
{
    System.out.println("m1 method is final");
    System.out.println(a=a+1);
}
    public static void main(String[] args)
    {
        Demo d=new Demo();
        d.m1();
    }
}
```

final methods

- If a method is declared with *final* keyword, it is called a final method. A final method **cannot be overridden**.

- If you want to restrict implementation of a method then you can declare it as final.

- For example in Object class we can override some methods like **equals(), toString()** but you cant override method like **wait(), notify(), notifyAll()** because these methods are declared as final.

Eample:

```
class Parent
{
    public  void m1()
    {
        System.out.println("Parent class method ... m1()");
    }

    public final void m2()
    {
        System.out.println("It is method -2");
    }
  }

class Child  extends Parent
{
    public void m1()
    {
        System.out.println("Hello this is child class method..m1()");
    }

    public void m2()
    {
       //Compile thime error
    }

}
```

Note:

- The main **advantage** of **final** modifier is ,We can achieve **security** as no one can be allowed to change our implementation.

- But the main **disadvantage** of final keyword is we are missing key benefits of Oops like inheritance and polymorphism.

- Hence if you have specific requirement you can use but it never recommended to use final modifier.

Abstract classes

Abstraction

What is Abstraction?

- In Object-oriented programming, abstraction is a process of hiding the internal implementation details from the user, only the essential functionality will be provided to the user.

- Abstraction can be achieved with either **abstract classes** or **interfaces**

Normal methods vs Abstract Methods

Normal methods:

- Normal method contains declaration as well as method definition

Ex:

```
public void method()
{
   ---------
   --------body;
   ---------
}
```

Abstract methods:

- The method which is having declaration but not definition such type of methods are called abstract methods.
- Every abstract method should end with “**;**”.
- The child classes are responsible to provide implementation for parent class abstract methods.

Ex: abstract void method(); //abstract method

Note: The methods marked abstract end in a semicolon rather than curly braces.

Normal classes vs Abstract classes

Normal classes:

- Normal class is a java class contains only normal methods.

Ex:
```
class Test
    {
        void m1()
        {
          --
        }
        void m2()
        {
          --
        }
    }
```

Abstract class:

- If a class contains at least one abstract method then it is a abstract class.
- To specify the particular class is abstract and particular method is abstract method to the compiler use **abstract** modifier.
- It is not possible to create an object to the Abstract class. Because it contains the unimplemented methods.
- For any class if we don’t want instantiation then we have to declare that class as abstract i.e., for abstract classes instantiation (creation of object) is not possible.

Ex:

```
abstract class Test
{
    abstract public void m1();
    public void m2()
    {
    }

}
```

➢Abstract method (all) implementation should be done in Child class.

Ex:

```
abstract class Parent
{
    abstract public void m1();

}

class Demo extends Parent
{

    public void m1()
    {
        System.out.println("Method m1() implementation");

    }
}
```

- Even though class does not contain any abstract method still we can declare the class as abstract.
- Abstract class contain zero or more number of abstract methods.
- For abstract classes it is not possible to create an object

Ex:

```
abstract class Test
{
     void m1()
     {
          System.out.println("m1-method");
     }
     void m2()
     {
          System.out.println("m2-method");
     }
     public static void main(String[] args)
     {
          Test t=new Test()                    //Compile time Error
          t.m1();
     }
};
```

Interfaces

- Interface is also one type of class and It contains only abstract methods.

- All interface methods are implicitly public and abstract. In other words, you do not need to actually type the public or abstract modifiers in the method declaration, but the method is still always public and abstract.

- For the every interface compiler will generates .class files

- Each and every interface by default abstract hence it is **not possible to create an object.**

- Interfaces not alternative for abstract class it is extension for abstract classes.

- Interface contains only abstract methods means unimplemented methods.
- Interface also called 100% Abstract class.
- Interfaces giving the information about the functionalities or Services And it will hide the information about internal implementation.
- To provide implementation for abstract methods we have to use **implements** Keyword
- For the interfaces also inheritance concept is applicable.

- By using **interface** keyword we can declare interfaces in Java

Syntax:

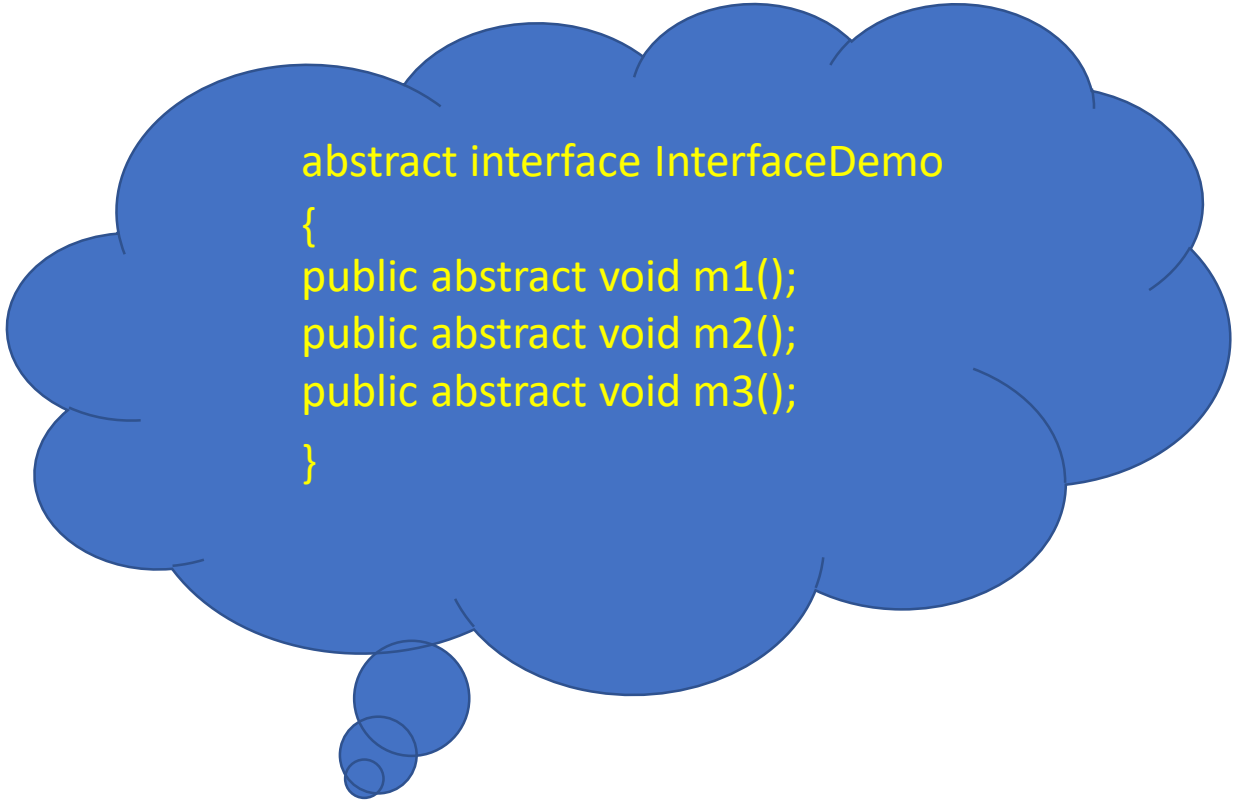
```
interface Interface_Name
  {
        --
  }
```

Ex:

```
interface Demo
{
        void m1();
}
```

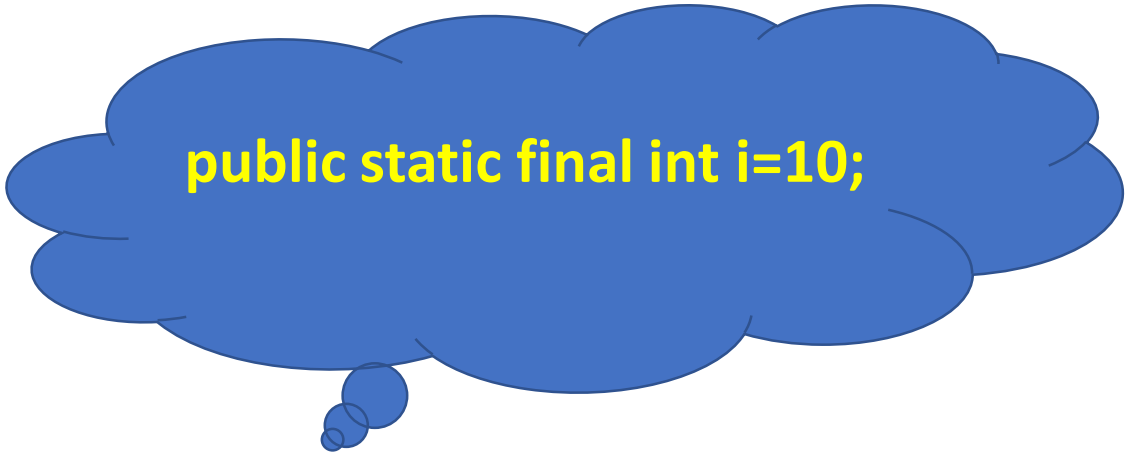
Ex:

```
interface InterfaceDemo
{
void m1();
void m2();
void m3();
}
```



➢Every variable in Interface by default public static final

```
interface InterfaceDemo
{
  int i=10;
}
```

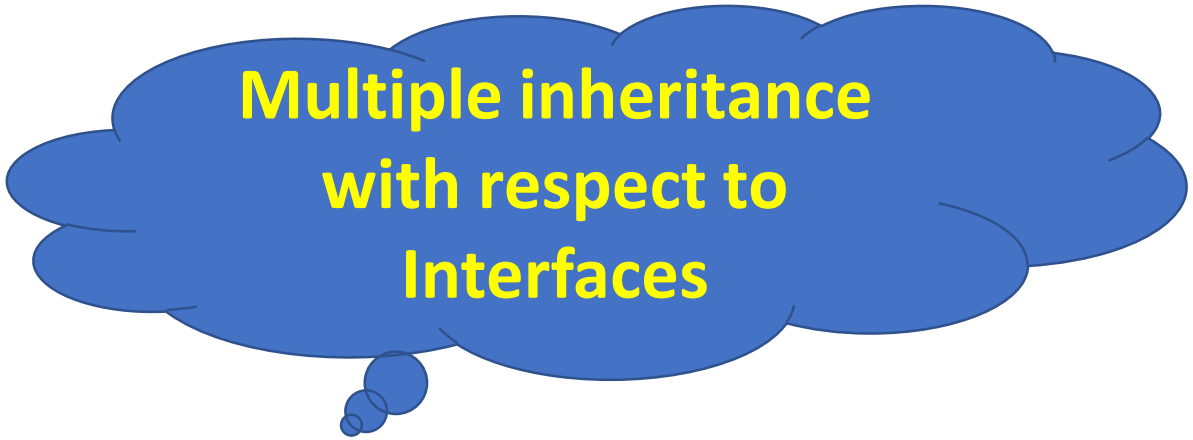



```
interface InterfaceDemo1
{
    public void method1();
    public void method2();
}
interface InterfaceDemo2 extends InterfaceDemo1
{
    public void method3();
    public void method4();
}
class Demo implements InterfaceDemo2
{
    @Override
    public void method1() {
        System.out.println("InterfaceDemo1-  method1() implementation");

    }
    @Override
    public void method2() {
        System.out.println("InterfaceDemo2-  method2() implementation");

    }
    @Override
    public void method3() {
        System.out.println("InterfaceDemo3-  method3() implementation");
    }
    @Override
    public void method4() {
        System.out.println("InterfaceDemo4-  method4() implementation");

    }
}
```




```
interface InterfaceDemo1
{
    public void method1();
}
interface InterfaceDemo2
{
    public void method2();
    public void method3();
}
class Demo implements InterfaceDemo1,InterfaceDemo2
{
    @Override
    public void method1() {
        System.out.println("InterfaceDemo1-  method1() implementation");
    }
    @Override
    public void method2() {
        System.out.println("InterfaceDemo2-  method2() implementation");
    }
    @Override
    public void method3() {
        System.out.println("InterfaceDemo3-  method3() implementation");
    }
}
```

Packages

Access control

Access modifiers:

- The access modifiers in Java talks about the accessibility or scope of a Variables ,Methods, Constructor, or Class.
- Depends on our requirements we can change the access level.
- In Java we have 4 types of access modifiers:
  - private
  - Default(No access modifier)
  - protected
  - public

private:

- Private modifier is applicable to methods , constructor and data members**(Variables).**
- private members are accessible only **within the class** in which they are declared. Out side of the class we cannot access.
- We cannot declare class with **private** modifier. But it is possible to declare for inner class.
- **private** modifier having less scope to comparative other access modifiers i.e, more restrictive.

default modifier(No modifier):

- If we are not specifying any access modifier for a class , method or data member ,It is said to be having the **default** access modifier by default.
- The data members, class or methods which are not declared using any access modifiers i.e. having default access modifier are accessible **only within the same package**.

- **default** modifier having more scope than **private** and less scope than protected and public modifiers.

protected:

- The methods or data members declared as protected are **accessible within same package or sub classes in different package.**

- **protected** modifier having more scope than **default, private** modifiers and less scope than **public** modifier.

public:

- The public access modifier has the **more scope** among all other access modifiers.
- Classes, methods or data members which are declared as public are **accessible from every where** in the program.
- There is no restriction on the scope of a public data members.

Scope order:

public > protected > default > private

	Private	No modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different Package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Packages

Information regarding packages:

- The package contains group of related classes , interfaces, sub packages.
- The package is an **encapsulation mechanism** it is binding the related classes and interfaces.
- We can declare a package with the help of **package** keyword.
- Package is nothing but physical directory (folder) structure and it is providing clear-cut separation between the project modules.
- Whenever we are dividing the project into the packages(modules) the shareability of the project will be increased.

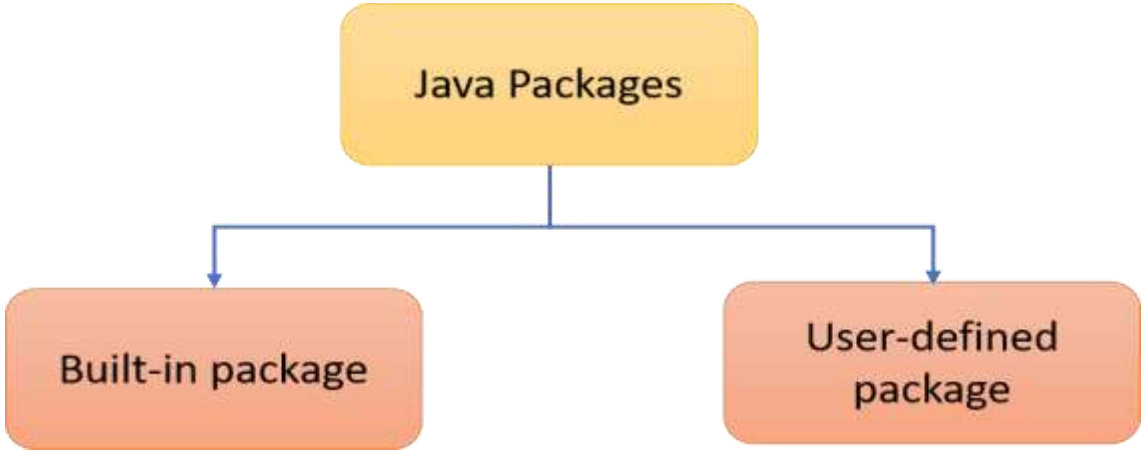
Syntax:

package package_name;

Ex: package com.mlrit;

Packages are divided into two types

- Predefined packages
- User defined packages



Predefined packages:

Java predefined packages contains all predefined classes and interfaces.

Ex:

java.lang ,

Java.io,

Java.awt ,

Java.util

Java.net.. etc.

Java.lang:

- The most commonly required classes and interfaces to write a sample program is encapsulated into a separate package is called **java.lang** package.

- It is a default package , No need to import this package.

Ex: String(class)

StringBuffer(class)

Object(class)

Runnable(interface)

Cloneable(nterface)

Java.io :

- The classes which are used to perform the input output operations that are present in the **java.io** packages.

Ex:

FileInputStream(class)
FileOutputStream(class)
FileWriter(class)
FileReader(class)
Serializable(Inteface)

java.net :

The classes which are required for connection establishment in the network that classes are present in the **java.net** package.

Ex:

HttpURLConnection
Socket
URL
ServerSocket
InetAddress
SocketOptions (Interface)

Java.util

- Contains the collections framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes.

Ex:

Calender
Date
Scanner
Arrays
ArrayList
Collection<E> (Interface)
Iterator<E> (Interface)
.. etc

java.sql

- Provides the API for accessing and processing data stored in a data source (usually a relational database) using the Java™ programming language.

Ex:

Date

DriverManager

Blob (Interface)

java.awt:

The classes which are used to prepare graphical user interface those classes are present in the **java.awt** package.

Ex: Button (class)

Checkbox(class)

Choice (Class)

List (class)

ActiveEvent (Interface)

User defined packages:

- The packages which are defined by the user are called user defined packages.
- In the single source file it is possible to take the only one package. If we are trying to take two packages at that time compiler raise a compilation error.
- In the source file it is possible to take single package statement.
- While taking package name we have to follow some coding standards.

Rules to follow while writing package:

- The package name is must reflect with your organization name and package name is reverse of the organization domain name.

Domain name: **www.example.com**

Package name: **package com.example**;

- The package must be the first statement of the source file and it is possible to declare **at most one** package within the source file .

- The import statement must be in between the package and class statement. And it is possible to declare **any number of import statements** within the source file.

- The class declaration must be after package and import statement and it is possible to declare any number of class within the source file.

- It is possible to declare **at most one public class.**
- It is possible to declare any number of non-public classes.
- The package and import statements are applicable for all the classes present in the source file.
- It is possible to declare comments at beginning and ending of any line of declaration it is possible to declare any number of comments within the source file.

Advantages of Packages:

- Java package is used to categorize the classes and interfaces so that they can be easily maintained.
- Java package removes **naming collision** or **naming conflicts**.
- Packages provide **reusability** of code .
- Java package provides **access protection**.

Path and Class Path:

- Path and Classpath both are operating system level environment variables.

- Path is used define where the system can find the **executables (.exe) or bin** files.

- Classpath is used to specify the location **.class** files.

Static import

- Static import allows you to access the static member of a class directly without using the fully qualified name.

Example:

```
import static java.lang.Math.*;
class Test
{
    public static void main(String[] args)
    {
        System.out.println(sqrt(16));
        System.out.println(min(20,30));
    }
}
```