**Unit - III**

Inheritance: types of inheritance, use of super, method overriding, final class, abstract class, wrapper classes. Interface, Packages and visibility controls

**Inheritance**

Inheritance is one of the key features of Object Oriented Programming. Inheritance provided mechanism that allowed **a class to inherit property of another class**. When a Class extends another class it inherits all non-private members including fields and methods. Inheritance in Java can be best understood in terms of Parent and Child relationship, also known as **Super class**(Parent) and **Sub class**(child) in Java language.

Inheritance defines **is-a** relationship between a Super class and its Sub class. extends and implements keywords are used to describe inheritance in Java.

**Purpose of Inheritance**

1. It promotes the code reusabilty i.e the same methods and variables which are defined in a parent/super/base class can be used in the child/sub/derived class.
2. It promotes polymorphism by allowing method overriding.

**Terms used in Inheritance**

* **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
* **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
* **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
* **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

**Disadvantages of Inheritance**

Main disadvantage of using inheritance is that the two classes (parent and child class) gets **tightly coupled**.

This means that if we change code of parent class, it will affect to all the child classes which is inheriting/deriving the parent class, and hence, **it cannot be independent of each other**.

**Extends Keyword**

**extends** is the keyword used to inherit the properties of a class.

**Types of Inheritance**

There are various types of inheritance as demonstrated below.



|  |  |
| --- | --- |
| **Single Level Inheritance** | **Multi Level Inheritance** |
| class one  {  void show()  {  System.out.println("hello one class");  }  }  class two extends one  {  public static void main(String args[])  {  two t=new two();  t.show();  }  } | class one  {  void show1()  {  System.out.println("hello one class");  }  }  class two extends one  {  void show2()  {  System.out.println("hello two class");  }  }  class three extends two  {  public static void main(String args[])  {  three t=new three();  t.show1();  t.show2();  }  } |

**Super keyword**

The **super** keyword is similar to **this** keyword. Following are the scenarios where the super keyword is used.

It use to show value of base class into derivred class when overriding.

* It is used to **differentiate the members** of superclass from the members of subclass, if they have same names.
* It is used to **invoke the superclass** constructor from subclass.

**Method Overriding in Java**

When a method in a sub class has same name, same number of arguments and same type signature as a method in its super class, then the method is known as overridden method. Method overriding is also referred to as runtime polymorphism.

|  |  |
| --- | --- |
| **Super Keyword** | **Method Overriding** |
| class one  {  void show()  {  System.out.println("Welcome");  }  }  class two extends one  {  void show()  {  **super.show();**  System.out.println("Bye bye");  }  public static void main(String args[])  {  two t=new two();  t.show();  }  } | class one  {  void show()  {  System.out.println("Welcome");  }  }  class two extends one  {  void show()  {  System.out.println("Bye bye");  }  public static void main(String args[])  {  two t=new two();  t.show();  }  } |

**Rules for Method Overriding**

* The method must have the same name as in the parent class
* The method must have the same parameter as in the parent class.
* A method declared final cannot be overridden.
* A method declared static cannot be overridden but can be re-declared.
* If a method cannot be inherited, then it cannot be overridden.
* Constructors cannot be overridden

**Difference between Overloading and Overriding**

Method overloading and Method overriding seems to be similar concepts but they are not. Let's see some differences between both of them:

| **Method Overloading** | **Method Overriding** |
| --- | --- |
| Parameter must be different and name must be same. | Both name and parameter must be same. |
| Compile time polymorphism. | Runtime polymorphism. |
| Increase readability of code. | Increase reusability of code. |
| Method overloading is performed within class. | Method overriding occurs in two classes that have IS-A (inheritance) relationship. |

**Final Class**

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

1. variable
2. method
3. class

The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only.

**Java final variable**

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

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

|  |  |
| --- | --- |
| **Final Variable** | **Output** |
| class demo1  {  final int num=90; //final variable  void show()  {  num=100;  System.out.println(num);  }  public static void main(String args[]){  demo1 ob=new demo1();  ob.show();  }  }//end of class | We can’t assign a value to final var num because it is already have a value and we declare as a final var. |

**Final method**

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

|  |  |
| --- | --- |
| **Final Method** | **Output** |
| class one  {  final void show()  {  System.out.println("Welcome");  }  }  class two extends one  {  void show()  {    System.out.println("Bye bye");  }  public static void main(String args[])  {  two t=new two();  t.show();  }  } | Cannot override because we declare method as a final |

**Java final class**

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

|  |  |
| --- | --- |
| **Final Class** | **Output** |
| final class one  {  void show()  {  System.out.println("Welcome");  }  }  class two extends one  {  public static void main(String args[])  {  two t=new two();  t.show();  }  } | When we declare class a final it cannot be extends / inherit with another class. |

**Wrapper classes in Java**

The **wrapper class in Java** provides the mechanism to convert primitive into object and object into primitive.

Since J2SE 5.0, **autoboxing** and **unboxing** feature convert primitives into objects and objects into primitives automatically. The automatic conversion of primitive into an object is known as autoboxing and vice-versa unboxing.

**Use of Wrapper classes in Java**

Java is an object-oriented programming language, so we need to deal with objects many times like in Collections, Serialization, Synchronization, etc. Let us see the different scenarios, where we need to use the wrapper classes.

* **Change the value in Method:** Java supports only call by value. So, if we pass a primitive value, it will not change the original value. But, if we convert the primitive value in an object, it will change the original value.
* **Serialization:** We need to convert the objects into streams to perform the serialization. If we have a primitive value, we can convert it in objects through the wrapper classes.
* **Synchronization:** Java synchronization works with objects in Multithreading.
* **java.util package:** The java.util package provides the utility classes to deal with objects.
* **Collection Framework:** Java collection framework works with objects only. All classes of the collection framework (ArrayList, LinkedList, Vector, HashSet, LinkedHashSet, TreeSet, PriorityQueue, ArrayDeque, etc.) deal with objects only.

The eight classes of the *java.lang* package are known as wrapper classes in Java. The list of eight wrapper classes are given below:

|  |  |
| --- | --- |
| Primitive Type | Wrapper class |
| boolean | [Boolean](https://www.javatpoint.com/java-boolean) |
| char | [Character](https://www.javatpoint.com/post/java-character) |
| byte | [Byte](https://www.javatpoint.com/java-byte) |
| short | [Short](https://www.javatpoint.com/java-short) |
| int | [Integer](https://www.javatpoint.com/java-integer) |
| long | [Long](https://www.javatpoint.com/java-long) |
| float | [Float](https://www.javatpoint.com/java-float) |
| double | [Double](https://www.javatpoint.com/java-double) |

**Autoboxing**

The automatic conversion of primitive data type into its corresponding wrapper class is known as autoboxing, for example, byte to Byte, char to Character, int to Integer, long to Long, float to Float, boolean to Boolean, double to Double, and short to Short.

Since Java 5, we do not need to use the valueOf() method of wrapper classes to convert the primitive into objects.

**Wrapper class Example: Primitive to Wrapper**

//Java program to convert primitive into objects

//Autoboxing example of int to Integer

**class** demo{

**public** **static** **void** main(String args[]){

//Converting int into Integer

**int** a=20;

Integer i=Integer.valueOf(a);//converting int into Integer explicitly

System.out.println(a+" "+i);

**Output : 20 20**

**Unboxing**

The automatic conversion of wrapper type into its corresponding primitive type is known as unboxing. It is the reverse process of autoboxing. Since Java 5, we do not need to use the intValue() method of wrapper classes to convert the wrapper type into primitives.

**Wrapper class Example: Wrapper to Primitive**

//Java program to convert object into primitives

//Unboxing example of Integer to int

**class** demo {

**public** **static** **void** main(String args[]){

//Converting Integer to int

Integer a=**new** Integer(3);

**int** i=a.intValue();//converting Integer to int explicitly

System.out.println(a+" "+i);

}}

**Output : 3 3**

**Interface**

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

The interface in Java is *a mechanism to achieve*[*abstraction*](https://www.javatpoint.com/abstract-class-in-java). There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple [inheritance in Java](https://www.javatpoint.com/inheritance-in-java).

An interface is a reference type in Java. It is similar to class. It is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface. Writing an interface is similar to writing a class. But a class describes the attributes and behaviors of an object. And an interface contains behaviors that a class implements.

Unless the class that implements the interface is abstract, all the methods of the interface need to be defined in the class.In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.

**Why use Java interface**

There are mainly three reasons to use interface. They are given below.

* It is used to achieve abstraction.
* By interface, we can support the functionality of multiple inheritance.
* It can be used to achieve loose coupling.

**Rules for using Interface**

* You cannot instantiate an interface.
* An interface does not contain any constructors.
* All of the methods in an interface are abstract.
* Methods inside Interface must not be static, final, native or strictfp.
* All variables declared inside interface are implicitly public static final variables(constants).
* All methods declared inside Java Interfaces are implicitly public and abstract, even if you don't use public or abstract keyword.
* Interface can extend one or more other interface.
* Interface cannot implement a class.
* Interface can be nested inside another interface.

**How to declare an interface**

An interface is declared by using the interface keyword. It provides total abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default. A class that implements an interface must implement all the methods declared in the interface.

**The 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**.



**Implementing Interfaces**

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

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

**Extending Interfaces**

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

|  |  |
| --- | --- |
| **Interface Implements with class** | **Interface extends with interface** |
| interface A1  {  void show();  }  class demo implements A1  {  public void show()  {  System.out.println("hello A1 Interface");  }  public static void main(String args[])  {  demo ob=new demo();  ob.show();  }  } | interface A1  {  void show1();  }  interface A2 extends A1  {  void show2();  }  class demo implements A2  {  public void show1()  {  System.out.println("hello A1 Interface");  }  public void show2()  {  System.out.println("hello A2 Interface");  }  public static void main(String args[])  {  demo ob=new demo();  ob.show1();  ob.show2();  }  } |

**Abstract class:**

A class which is declared with the abstract keyword is known as an abstract class in [Java](https://www.javatpoint.com/java-tutorial). It can have abstract and non-abstract methods (method with the body).

Before learning the Java abstract class, let's understand the abstraction in Java first.**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](https://www.javatpoint.com/object-and-class-in-java) 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](https://www.javatpoint.com/java-constructor) 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{}**

## Abstract class Example

//abstract parent class

abstract class Animal{

//abstract method

public abstract void sound();

}

//Dog class extends Animal class

public class Dog extends Animal{

public void sound(){

System.out.println("Woof");

}

public static void main(String args[]){

Animal obj = new Dog();

obj.sound();

}

}

**Output: Woof**

**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.

**Java Packages: Built-in and User defined**

* **Built-in Package:** Existing Java package for example java.lang, java.utiletc.
* **User-defined-package:** Java package created by user to categorize their project's classes and interface.

**Additional points about package:**

* A package is always defined as a separate folder having the same name as the package name.
* Store all the classes in that package folder.
* All classes of the package which we wish to access outside the package must be declared public.
* All classes within the package must have the package statement as its first line.
* All classes of the package must be compiled before use (So that they are error free)

**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. 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.

**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**

**Visibility controls**

There are two types of modifiers in Java: **access modifiers** and **non-access modifiers**.

The access modifiers in Java specifies the accessibility or scope of a field, method, constructor, or class. We can change the access level of fields, constructors, methods, and class by applying the access modifier on it.

There are four types of Java access modifiers:

1. **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.

Methods, variables, and constructors that are declared private can only be accessed within the declared class itself.Private access modifier is the most restrictive access level. Class and interfaces cannot be private.

Variables that are declared private can be accessed outside the class, if public getter methods are present in the class.Using the private modifier is the main way that an object encapsulates itself and hides data from the outside world.

1. **Default**: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.

Default access modifier means we do not explicitly declare an access modifier for a class, field, method, etc.A variable or method declared without any access control modifier is available to any other class in the same package. The fields in an interface are implicitly public static final and the methods in an interface are by default public.

1. **Protected**: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.

Variables, methods, and constructors, which are declared protected in a superclass can be accessed only by the subclasses in other package or any class within the package of the protected members' class.

The protected access modifier cannot be applied to class and interfaces. Methods, fields can be declared protected, however methods and fields in a interface cannot be declared protected.Protected access gives the subclass a chance to use the helper method or variable, while preventing a nonrelated class from trying to use it.

1. **Public**: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

A class, method, constructor, interface, etc. declared public can be accessed from any other class. Therefore, fields, methods, blocks declared inside a public class can be accessed from any class belonging to the Java Universe.

However, if the public class we are trying to access is in a different package, then the public class still needs to be imported. Because of class inheritance, all public methods and variables of a class are inherited by its subclasses.

There are many non-access modifiers, such as static, abstract, synchronized, native, volatile, transient, etc. Here, we are going to learn the access modifiers only.

**Understanding Java Access Modifiers**

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

