UNIT – IV

INHERITANCE, PACKAGES AND INTERFACES

4.1 Basics of Inheritance

- Inheritance in Java is used to build new classes from existing classes. The mechanism of deriving a new class from an existing class is called inheritance.

- A class that is inherited is called a *superclass*. *The class that* does the inheriting is called a *subclass*.

- Therefore, a subclass is a specialized version of a superclass. It inherits all of the instance variables and methods defined by the superclass and adds its own, unique elements.

- **Purpose of Inheritance**
- To promote code reuse i.e. supporting reusability.
- To implement Polymorphism.

4.1 Basics of Inheritance

- "extends" key-word is used to inherit a class.

- **Example:**

```
class A
{
        int i;
}

class B extends A
{
        int j;

        void sum()
        {
                System.out.println("Sum : " + ( i + j ));
        }
}
```

4.1 Basics of Inheritance

```
public class Example
{
    public static void main(String args[])
    {
        B obj = new B();

        obj.i = 5;
        obj.j=7;

        obj.sum();
    }
}
```

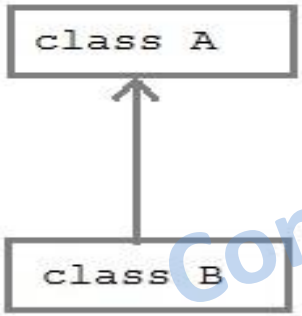

Output:

Sum : 12

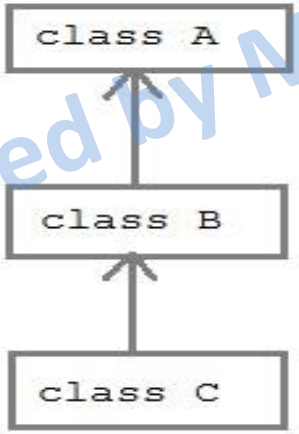
4.1.1 Types of Inheritance

- Java supports following type of inheritance of class.
  - Single Inheritance
  - Multilevel Inheritance
  - Hierarchical Inheritance

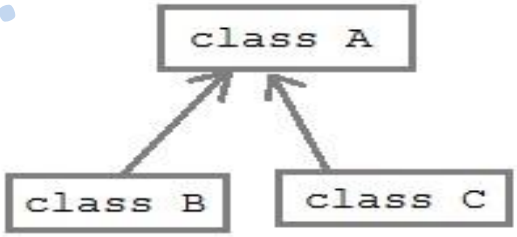
- Multiple inheritance is not supported in Java.



Simple Inheritance



Multilevel inheritance



Heirarchical inheritance

4.1.2 Method Overriding

- When a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to *override the method in the superclass.*

- When an overridden method is called from a subclass object, it will always refer to the version of that method defined by the subclass.

- The version of the method defined by the superclass will be hidden.

4.1.2 Method Overriding

```
class A
 {
       int i;

      A(int a)
      {
          i=a;
      }

      void show()
       {
          System.out.println("I : " + i);
      }
}
```

4.1.2 Method Overriding

```
Class B extends A
{
        int j;

        B(int a, int b)
        {
            i=a;
            j=b;
        }

        void show()
        {
            System.out.println(“j : " + j );
        }
}
```

4.1.2 Method Overriding

```
public Class Example
{
        public static void main(String a[])
        {
                B obj =new B(5,7);
                obj.show();             // this calls show() in B
        }
}
```


Output:

j : 7

4.1.3 Extending Class

- In Java, “extends” key-word is used to inherit a class.
- Code reuse can be achieved by simply extending classes by inheritance.
- Java allows a class to extends only one other class, which is called single inheritance.
- Java doesn’t support multiple inheritance.
- Java supports multi-level inheritance.

4.1.3 Extending Class

- **Syntax:**

```
class superclass
{
        body of superclass
}

class subclass extends superclass
{
        body of subclass
}
```

4.1.3 Extending Class

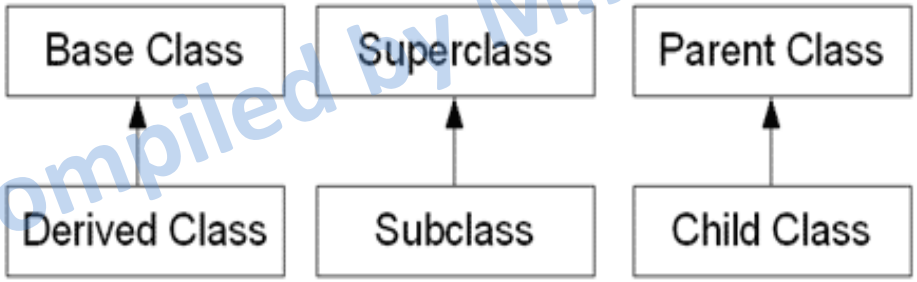
Example:

```
class Vehicle.
{
    ......
}
class Car extends Vehicle
{
    .......
}
```

4.1.4 Super Class & Sub Class

- Inheritance in Java can be best understood in terms of Parent and Child relationship, also known as **Super class**(Parent) and **Sub class**(child).

- Several pairs of terms are used to discuss class relationships.



4.1.4 Super Class & Sub Class

Example:

```
class Vehicle.
{
    ......
}
class Car extends Vehicle
{
    .......
}
```


- In above example we can say that:
  - **Vehicle** is super class of **Car**.
  - **Car** is sub class of **Vehicle**.

4.1.5 Dynamic Method Dispatch

- **A Superclass Variable Can Reference a Subclass Object**

- A reference variable of a superclass can be assigned a reference to any subclass derived from that superclass.

- when a reference to a subclass object is assigned to a superclass reference variable, you will have access only to those parts of the object defined by the superclass.

- It is important to understand that it is the type of the reference variable—not the type of the object that it refers to—that determines what members can be accessed.

4.1.5 Dynamic Method Dispatch

- **A Superclass Variable Can Reference a Subclass Object (Example)**

```
class A
 {
       int i;
       void showi()
       {
           System.out.println("I : " + i );
       }
}
class B extends A
{
       int j;
       void showj()
       {
           System.out.println(“j : " + j);
       }
}
```

4.1.5 Dynamic Method Dispatch

- **A Superclass Variable Can Reference a Subclass Object (Example)**

```
public class Example
{
    public static void main(String args[])
    {
        A obj = new B();
        obj.i = 5;
        // Obj.j= 7;   //can not be accessed because the j is member of class B.
        Obj.showi();
        //Obj.showj(); //can not be accessed because the showj() is member of class B.
    }
}
```

4.1.5 Dynamic Method Dispatch

- Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.
.
- Dynamic method dispatch is important because this is how Java implements run-time polymorphism.

- When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs.

- When different types of objects are referred to, different versions of an overridden method will be called. In other words, it is the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed.

4.1.5 Dynamic Method Dispatch

- **Example (Dynamic Method Dispatch)**

```
class A
{
        void display()
        {
            System.out.println("Inside class - A");
        }
}

class B extends A
{
        void display()
        {
            System.out.println("Inside class - B ");
        }
}

class C extends A
{
        void display()
        {
            System.out.println("Inside class - C ");
        }
}
```

4.1.5 Dynamic Method Dispatch

- **Example (Dynamic Method Dispatch)**

```
public class Example
{
        public static void main(String args[])
        {
                A a = new A();  // object of type A
                B b = new B();  // object of type B
                C c = new C();  // object of type C
                A ref;          // obtain a reference of type A

                ref= a;         // ref refers to an  object of class A
                ref.display();  // call calss A's  display method

                ref= b;         // ref refers to an  object of class B
                ref.display();  // call calss B's  display method

                ref= c;         // ref refers to an  object of class C
                ref.display();  // call calss C's  display method
        }
}
```

4.1.6 Object Class

- There is one special class, Object, defined by Java. All other classes are subclasses of Object.

- That is, Object is a superclass of all other classes. This means that a reference variable of type Object can refer to an object of any other class.

4.1.6 Object Class

Methods	Purpose
Object clone( )	Creates a new object that is the same as the object being cloned.
boolean equals(Object *object)*	*Determines whether one object is equal to* another.
void finalize( )	Called before an unused object is recycled.
class getClass( )	Obtains the class of an object at run time.
int hashCode( )	Returns the hash code associated with the invoking object.
void notify( )	Resumes execution of a thread waiting on the invoking object.
void notifyAll( )	Resumes execution of all threads waiting on the invoking object.
String toString( )	Returns a string that describes the object.
• void wait( ) • void wait (long *milliseconds)*	Waits on another thread of execution.

4.2 Basics of Package

- A package is a collection of logically related classes, interfaces and sub-packages.
- The package statement defines a name space in which classes are stored.
- Java uses file system directories to store packages.

4.2.1 Creating Package

- To create a package is quite easy: simply include a package command as the first statement in a Java source file.

- **Syntax:**

- **package** package_name;

- **Example:**

- **package** MyPackage;

- Any classes declared within that file will belong to the specified package.

- If you omit the package statement, the class names are put into the default package, which has no name.

- While the default package is fine for sample programs, it is inadequate for real applications. Most of the time, you will define a package for your code.

4.2.1 Creating Package

- **Example**

```
package MyPackage;

public class Example
 {
       public static void main(String[] args)
       {
           System.out.println("Package Demonstration.");
        }
}
```

4.2.1 Creating Package

- Java uses file system directories to store packages. For example, the **.class files for** any classes (Example.class) you declare to be part of MyPackage must be stored in a directory called MyPackage.

- More than one file can include the same package statement. The package statement simply specifies to which package the classes defined in a file belong.

4.2.1 Creating Package

- You can create a hierarchy of packages. To do so, simply separate each package name from the one above it by use of a period. The general form of a multileveled package statement is shown here:

- package *pkg1[.pkg2[.pkg3]];*

- A package hierarchy must be reflected in the file system of your Java development system.

For example, a package declared as:

package *pkg1.pkg2.pkg3;*

- needs to be stored in **pkg1\pkg2\pkg3** on your Windows file system.

4.2.1 Creating Package

- **Example**

```
package p1.p2.p3;

public class Example
 {
       public static void main(String[] args)
       {
           System.out.println("Package Demonstration.");
        }
}
```

4.2.2 Importing Package

- Following is the syntax for including/using a Package in the program.

- Including Package :

```
import package_name.*;
```

- Including Package Member :

```
import package_name.member_name;
```

4.2.2 Importing Package

Example:

Test.java

```
package p1;
public class Test
{
	public void display()
	{
	    System.out.println("Inside class Test.");
	}
}
```

4.2.2 Importing Package

Example.java

```
public class Example
{
	public static void main(String[] args)
	{
		Test t1 = new Test();	//error: cannot access Test
		t1.display();

		Test t2 = new Test();	//error: cannot access Test
		t2.display();
	}
}
```

4.2.2 Importing Package

Example.java

```
public class Example
{
	public static void main(String[] args)
	{
		p1.Test t1 = new p1.Test();
		t1.display();

		p1.Test t2 = new p1.Test();
		t2.display();
	}
}
```

4.2.2 Importing Package

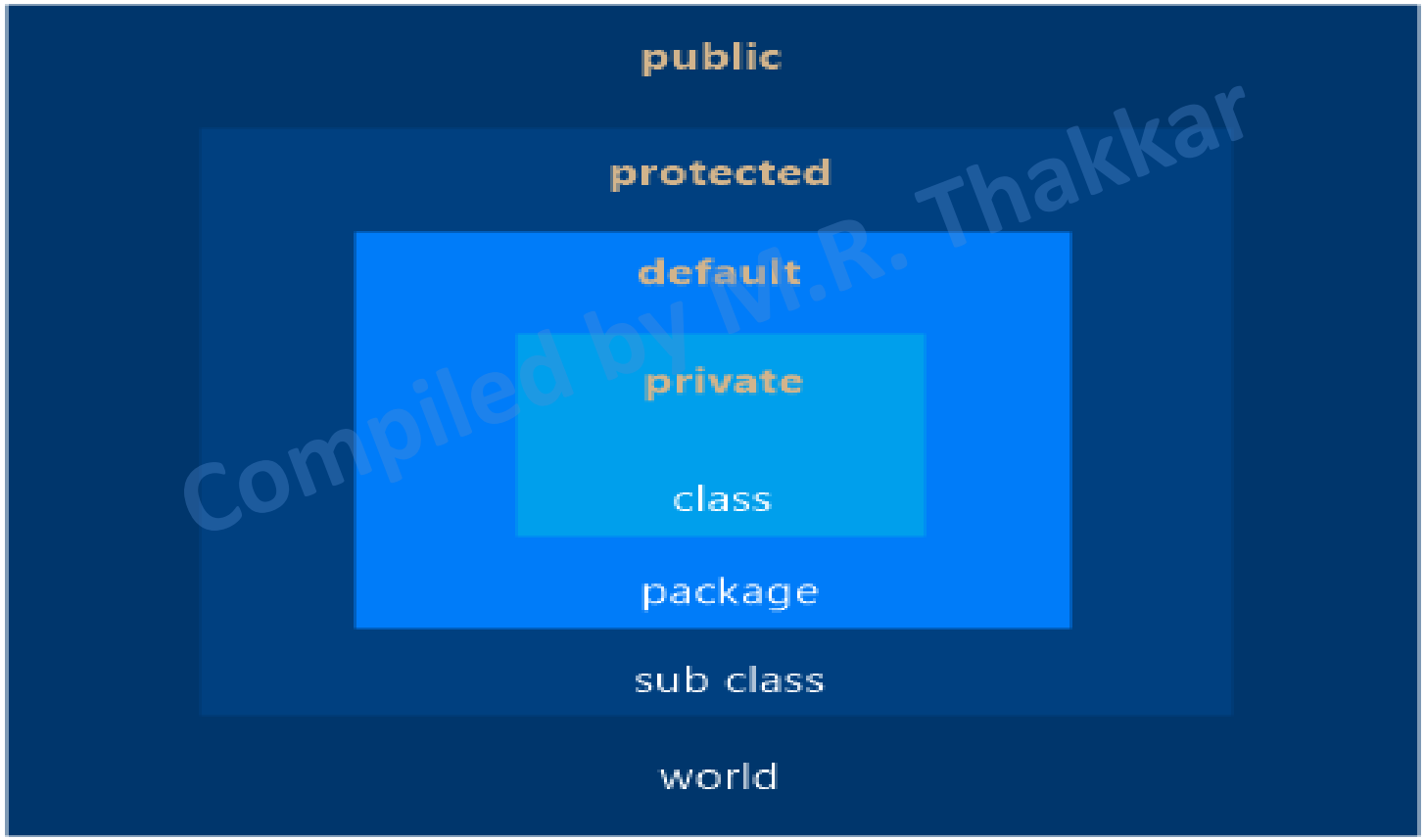
Example.java

```
import p1.Test;
public class Example
{
	public static void main(String[] args)
	{
		Test t1 = new Test();
		t1.display();

		Test t2 = new Test();
		t2.display();
	}
}
```

4.2.3 Access Rule for Package

- **Default (or Package): -** When a member does not have an explicit access specification, it is visible to subclasses as well as to other classes in the same package. This is the default access.



4.2.3 Access Rule for Package

	Private	Package	Protected	Public
Same Class	Yes	Yes	Yes	Yes
Same Package – Sub Class	No	Yes	Yes	Yes
Same Package - Different Class	No	Yes	Yes	Yes
Different Package – Sub Class	No	No	Yes	Yes
Different Package – Different Class	No	No	No	Yes

4.2.3 Access Rule for Package

Example:

```
package  p1;

public class Protection
{
                int   n = 1;
    public  int  n_pub = 2;
}
```

4.2.3 Access Rule for Package

```
import  p1.Protection;
public class Example
{
    public static void main(String[] args)
    {
        Protection p1 = new Protection();
        System.out.println("n = " + p1.n);  //error
        System.out.println("n_pub = " +p1.n_pub);
    }
}
```

4.3 Basics of Interface

- Interfaces are syntactically similar to classes, but they lack instance variables, and their methods are declared without any body.

- Using the keyword interface, you can fully abstract a class' interface from its implementation.

- That is, using interface, you can specify what a class must do, but not how it does it.

4.3.1 Defining Interface

- An interface is defined much like a class. This is the general form of an interface:

```
Interface interfacename
{
        type final-varname1 = value;
        type final-varname2 = value;
        // ...
        type final-varnameN = value;

    return-type methodname1( parameter-list );
    return-type methodname2( parameter-list );
                    // ...
    return-type methodnameN( parameter-list );
}
```

4.3.1 Defining Interface

- **interface Example**

```
interface Test
{
    void display();
}
```

- Once it is defined, any number of classes can implement an interface. Also, one class can implement any number of interfaces.

4.3.3 Implementing Interface

- To implement an interface, include the **implements** clause in a class definition, and then implements the methods defined by the interface.
- The general form of a class that includes the **implements** clause looks like**:**

class ***classname*** implements ***interface_name [,interface_name...]]***

```
{
        // class-body
}
```


- If a class implements more than one interface, the interfaces are separated with a comma.

4.3.3 Implementing Interface

- The methods that implement an interface must be declared **public.** Also, the signature of the implementing method must match exactly the signature specified in the **interface definition.**

```
class InterfaceDemo implements Test
{
    // Implement Test interface
     public void display()
     {
         System.out.println(“Implementing Display method of Test.");
     }
}
```

4.3.3 Implementing Interface

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

4.3.2 Inheritance on Interface

- One interface can inherit another by use of the keyword extends**.** The syntax is the same as for inheriting classes.

- When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain.

4.3.2 Inheritance on Interface

```
interface A
{
    void method1();
}
interface B extends A
{
    void method2();
}
// B now includes method1() and  -- it adds method2().
```

4.3.2 Inheritance on Interface

```
class InterfaceDemo implements B
{
        public void method1()
         {
            System.out.println("Inside Method - 1");
        }
        public void method2()
        {
            System.out.println("Inside Method - 2");
        }
}
```

4.3.2 Inheritance on Interface

```
public class Example
{
    public static void main(String args[])
    {
            InterfaceDemo obj = new InterfaceDemo();
            obj.method1();
             obj.method2();
    }
}
```

4.3.4 Multiple Inheritance using Interface

- When you work with classes, you are limited to inheriting from only one base class.
- Interfaces relax this constraint somewhat by allowing you undertake a kind of multiple inheritance by combining multiple interfaces within a class.
- To do this, you simply place interface names in sequence following the implements keyword separated by commas.

4.3.4 Multiple Inheritance using Interface

```
interface A
{
    void method1();
}
interface B
{
    void method2();
}
```

4.3.4 Multiple Inheritance using Interface

```
class InterfaceDemo implements A , B
{
        public void method1()
         {
            System.out.println(“Inside Method - 1");
        }
        public void method2()
        {
            System.out.println(“Inside Method - 2");
        }
}
```

4.3.4 Multiple Inheritance using Interface

```
public class Example
{
    public static void main(String args[])
    {
        InterfaceDemo obj = new InterfaceDemo();
        obj.method1();
         obj.method2();
    }
}
```

4.4.1 Abstract Class

- Any method without definition (body) within a class is known as abstract method.

- **Syntax:**

  abstract return_type method_name (); // No definition

- If a class contains any abstract method, then the class is declared as abstract class.

- An abstract class is one which contains some defined method and some undefined method. An abstract class is declared with abstract keyword.

4.4.1 Abstract Class

- **Features of abstract class :**

- An abstract class must have an abstract method.
- Abstract classes can have Constructors, Member variables and Normal methods.
- Abstract classes are never instantiated.
- Abstract classes are not Interfaces.
- When you extend Abstract class, you must define the abstract method in the child class, or make the child class abstract.

- **Syntax:**

```
abstract class class_name
{
    // class-body
}
```

4.4.1 Abstract Class

- **Example:**

```
abstract class A
{
    abstract void method1();

    void method2()
    {
        System.out.println("Inside Method-2");
    }
}
```

4.4.1 Abstract Class

- **Example:**

```
class B extends A
 {
       void method1()
        {
           System.out.println(“Inside Method-1");
        }
}
```

4.4.1 Abstract Class

- **Example:**

```
public class Example
{
    public static void main(String args[])
    {
        B b = new B();
        b.method1();
        b.method2();
    }
}
```

4.4.2 Final Class

- Sometimes you need to prevent a class from being inherited.
- To do this, precede the class declaration with **final.**
- Declaring a class as final implicitly declares all of its methods as final, too.

4.4.2 Final Class

```
final class A
{
    // ...
}


// The following class is illegal.
class B extends A
{
        // ERROR! Can't subclass A
        // ...
}
```

****