Module - II

Inheritance in Java

"Inheritance" is a mechanism wherein a new class is derived from an existing class. In Java, classes may inherit or acquire the properties and methods of other classes. A class derived from another class is called a subclass, whereas the class from which a subclass is derived is called a superclass.

Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.

is-a relationship

In Java, inheritance is an **is-a** relationship. That is, we use inheritance only if there exists an is-a relationship between two classes. For example,

- **Car** is a **Vehicle**
- **Orange** is a **Fruit**
- **Surgeon** is a **Doctor**
- **Dog** is an **Animal**

Here, **Car** can inherit from **Vehicle**, **Orange** can inherit from **Fruit**, and so on.

Why to use inheritance in java?

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

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.

The syntax of Java Inheritance

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

The **extends keyword** indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

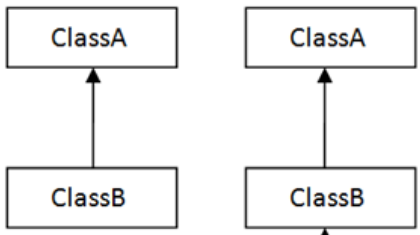
Example program for inheritance:

```
class Vehicle {
  protected String brand = "Ford";        // Vehicle attribute
  public void honk() {                    // Vehicle method
    System.out.println("Tuut, tuut!");
  }
}

class Car extends Vehicle {
  private String modelName = "Mustang";    // Car attribute
  public static void main(String[] args) {

    // Create a myCar object
    Car myCar = new Car();

    // Call the honk() method (from the Vehicle class) on the myCar object
    myCar.honk();
```

```
    // Display the value of the brand attribute (from the Vehicle class) and the value of the modelName from the Car class

    System.out.println(myCar.brand + " " + myCar.modelName);

  }

}
```

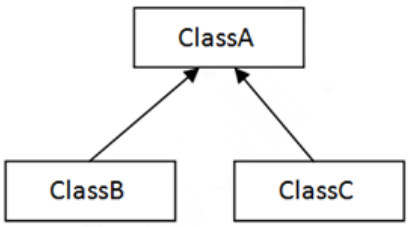
Types of Inheritance:

- Single Inheritance
- Multi-Level Inheritance
- Hierarchical Inheritance
- Multiple Inheritance
- Hybrid Inheritance

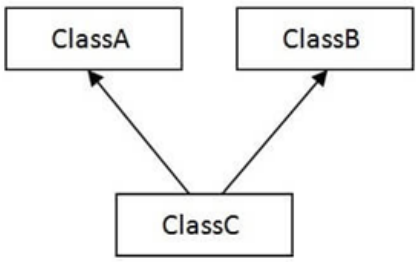


1) Single

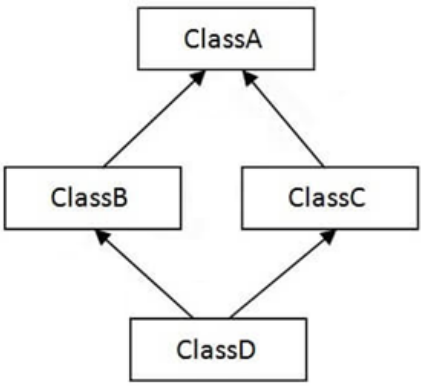
2) Multilevel



3) Hierarchical



4) Multiple



5) Hybrid

Single Inheritance

When a class inherits another class, it is known as a *single inheritance*. In the example given below, Dog class inherits the Animal class, so there is the single inheritance.

Example:

```
class Animal {
void eat()
{
System.out.println("eating...");
}
}
class Dog extends Animal {
void bark()
{
System.out.println("barking...");
}
}
class TestInheritance {
public static void main(String args[])
{
Dog d=new Dog();
d.bark();
d.eat();
}
}
```

Output:

Barking…

Eating…

Multiple Inheritance

In multiple inheritance, a single subclass extends from multiple super classes. To reduce the complexity and simplify the language, ***multiple inheritance is not supported in java***. Consider a scenario where A, B, and C are three classes. The C

class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.

Since compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. Therefore, whether you have same method or different, there will be compile time error.

Example:

```
class A{
void msg(){System.out.println("Hello");}
}
class B{
void msg(){System.out.println("Welcome");}
}
class C extends A,B{//suppose if it were


 public static void main(String args[]){
   C obj=new C();
   obj.msg();//Now which msg() method would be invoked?
}
}
```

Output:

Compile Time Error

Multilevel Inheritance

When there is a chain of inheritance, it is known as *multilevel inheritance*. As you can see in the example given below, BabyDog class inherits the Dog class which again inherits the Animal class, so there is a multilevel inheritance.

Example:

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

```
class Dog extends Animal {
void bark()
{
System.out.println("barking...");
}
}
class BabyDog extends Dog {
void weep()
{
System.out.println("weeping...");
}
}
class TestInheritance2 {
public static void main(String args[])
{
BabyDog d=new BabyDog();
d.weep();
d.bark();
d.eat();
}
}
```

Output:

```
weeping...
barking...
eating...
```

Hierarchical Inheritance

When two or more classes inherits a single class, it is known as *hierarchical inheritance*. In the example given below, Dog and Cat classes inherits the Animal class, so there is hierarchical inheritance.

Example:

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

```
}
}
class Dog extends Animal {
void bark()
{
System.out.println("barking...");
}
}
class Cat extends Animal {
void meow()
{
System.out.println("meowing...");
}
}
class TestInheritance3 {
public static void main(String args[])
{
Cat c=new Cat();
c.meow();
c.eat();
//c.bark();//C.T.Error
}
}
```

Output:

```
meowing...
eating...
```

Hybrid Inheritance

Hybrid inheritance is a combination of two or more types of inheritance.

Example:

```
class HumanBody
{
public void displayHuman()
```

```
{
System.out.println("Method defined inside HumanBody class");
}
}
interface Male
{
public void show();
}
interface Female
{
public void show();
}
public class Child extends HumanBody implements Male, Female
{
public void show()
{
System.out.println("Implementation of show() method defined in interfaces Male a
nd Female");
}
public void displayChild()
{
System.out.println("Method defined inside Child class");
}
public static void main(String args[])
{
Child obj = new Child();
System.out.println("Implementation of Hybrid Inheritance in Java");
obj.show();
obj.displayChild();
}
}
```

Output:

```
Implementation of Hybrid Inheritance in Java
Implementation of show() method defined in interfaces Male and Female
Method defined inside Child class
```

Polymorphism

Polymorphism is one of the OOPs feature that allows us to perform a single action in different ways. Polymorphism is the capability of a method to do different things based on the object that it is acting upon. In other words, polymorphism allows you define one interface and have multiple implementations.

There are two types of polymorphism in java:

1) **Static Polymorphism** also known as compile time polymorphism. Polymorphism that is resolved during compiler time is known as static polymorphism. ***Method overloading*** is an example of compile time polymorphism.

2) **Dynamic Polymorphism** also known as runtime polymorphism. It is also known as Dynamic Method Dispatch. Dynamic polymorphism is a process in which a call to an overridden method is resolved at runtime, that's why it is called runtime polymorphism. ***Method overriding*** is an example of compile time polymorphism.

Method Overloading

Method Overloading is a feature that allows a class to have multiple methods with the same name but with different number, sequence or type of parameters. In short **multiple methods with same name but with different signatures**. For example the signature of method add(int a, int b) having two int parameters is different from signature of method add(int a, int b, int c) having three int parameters. Method overloading is an example of Static Polymorphism. Static Polymorphism is also known as compile time binding or early binding.

Ways to overload a method

In order to overload a method, the argument lists of the methods must differ in either of these:

1. Number of parameters.

Example:

add(int, int)

add(int, int, int)

Example Program:

class DisplayOverloading

```
{
    public void disp(char c)
    {
        System.out.println(c);
    }
    public void disp(char c, int num)
    {
        System.out.println(c + " "+num);
    }
}
class Sample
{
   public static void main(String args[])
   {
      DisplayOverloading obj = new DisplayOverloading();
      obj.disp('a');
      obj.disp('a',10);
   }
}
```

Output:

```
a
a       10
```

2. Data type of parameters.

Example:

```
add(int, int)
add(int, float)
```

Example Program:

```
class DisplayOverloading2
{
```

```
    public void disp(char c)
    {
        System.out.println(c);
    }
    public void disp(int c)
    {
       System.out.println(c );
    }
}

class Sample2
{
    public static void main(String args[])
    {
        DisplayOverloading2 obj = new DisplayOverloading2();
        obj.disp('a');
        obj.disp(5);
    }
}
```

Output:

```
a
5
```

3. Sequence of Data type of parameters.

Example:

```
add(int, float)
add(float, int)
```

Example Program:

```
class DisplayOverloading3
{
```

```
    public void disp(char c, int num)
    {
        System.out.println("I'm the first definition of method disp");
    }
    public void disp(int num, char c)
    {
        System.out.println("I'm the second definition of method disp" );
    }
}
class Sample3
{
    public static void main(String args[])
    {
        DisplayOverloading3 obj = new DisplayOverloading3();
        obj.disp('x', 51 );
        obj.disp(52, 'y');
    }
}
```

Output:

I'm the first definition of method disp

I'm the second definition of method disp

Method Overriding

Declaring a method in **sub class** which is already present in **parent class** is known as method overriding. Overriding is done so that a child class can give its own implementation to a method which is already provided by the parent class. In this case the method in parent class is called overridden method and the method in child class is called overriding method.

Lets take a simple example to understand this. We have two classes: A child class Boy and a parent class Human. The Boy class extends Human class. Both the

classes have a common method void eat(). Boy class is giving its own implementation to the eat() method or in other words it is overriding the eat() method.

The purpose of Method Overriding is clear here. Child class wants to give its own implementation so that when it calls this method, it prints Boy is eating instead of Human is eating. The main advantage of method overriding is that the class can give its own specific implementation to a inherited method without even modifying the parent class code.

Example program:

```
class Human{
  //Overridden method
  public void eat()
  {
    System.out.println("Human is eating");
  }
}
class Boy extends Human{
  //Overriding method
  public void eat(){
    System.out.println("Boy is eating");
  }
  public static void main( String args[]) {
    Boy obj = new Boy();
    //This will call the child class version of eat()
    obj.eat();
  }
}
```

Output:

```
Boy is eating
```

Abstract Class

A class that is declared using "**abstract**" keyword is known as abstract class. It can have abstract methods(methods without body) as well as concrete methods (regular methods with body). A normal class(non-abstract class) cannot have abstract methods. An abstract class can not be **instantiated**, which means you are not allowed to create an **object** of it. Because these classes are incomplete, they have abstract methods that have no body so if java allows you to create object of this class then if someone calls the abstract method using that object then, there would be no actual implementation of the method to invoke.
Also because an object is concrete. An abstract class is like a template, so you have to extend it and build on it before you can use it.

Rules for abstract class

- 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: Abstract class declaration

```
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 of Abstract class and method

```
abstract class MyClass{
  public void disp(){
    System.out.println("Concrete method of parent class");
  }
  abstract public void disp2();
}

class Demo extends MyClass{
  /* Must Override this method while extending
   * MyClas
   */
  public void disp2()
  {
     System.out.println("overriding abstract method");
  }
  public static void main(String args[]){
     Demo obj = new Demo();
     obj.disp2();
  }
}
```

Output:

overriding abstract method

Example to demonstrate that object creation of abstract class is not allowed

```
abstract class AbstractDemo{
  public void myMethod(){
    System.out.println("Hello");
  }
  abstract public void anotherMethod();
}
public class Demo extends AbstractDemo{

  public void anotherMethod() {
      System.out.print("Abstract method");
  }
  public static void main(String args[])
  {
    //error: You can't create object of it
    AbstractDemo obj = new AbstractDemo();
    obj.anotherMethod();
  }
}
```

Output:

```
Unresolved compilation problem: Cannot instantiate the type AbstractDemo.
```

Interface in java

Abstract class which is used for achieving partial abstraction. Unlike abstract class an interface is used for full abstraction. Abstraction is a process where you show only “relevant” data and “hide” unnecessary details of an object from the user. Interface looks like a class but it is not a class. An interface can have methods and variables just like the class but the methods declared in interface are by default abstract. Also, the variables declared in an interface are public, static & final by default.

Why to 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.

Syntax:

```
interface MyInterface
{
  /* All the methods are public abstract by default
   * As you see they have no body
   */
  public void method1();
  public void method2();
}
```

Example Program:

```
interface MyInterface
{
  /* compiler will treat them as:
   * public abstract void method1();
   * public abstract void method2();
   */
  public void method1();
  public void method2();
}
class Demo implements MyInterface
{
  /* This class must have to implement both the abstract methods
   * else you will get compilation error
```

```
 */
    public void method1()
    {
        System.out.println("implementation of method1");
    }
    public void method2()
    {
        System.out.println("implementation of method2");
    }
    public static void main(String arg[])
    {
        MyInterface obj = new Demo();
        obj.method1();
    }
}
```

Output:

implementation of method1

Packages in java

A package as the name suggests is a pack(group) of classes, interfaces and other packages. In java we use packages to organize our classes and interfaces. We have two **types of packages in Java**: built-in packages and the packages we can create (also known as user defined package). A class can have only one package declaration. 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:

```
import java.util.Scanner
```

Here:

→ java is a top level package

→ util is a sub package

→ and Scanner is a class which is present in the sub package util.

Example for user defined package and implementation:

Calculator.java file created inside a package letmecalculate

```
package letmecalculate;
public class Calculator {
  public int add(int a, int b){
        return a+b;
  }
  public static void main(String args[]){
        Calculator obj = new Calculator();
        System.out.println(obj.add(10, 20));
  }
}
```

Now lets see how to use this package in another program.

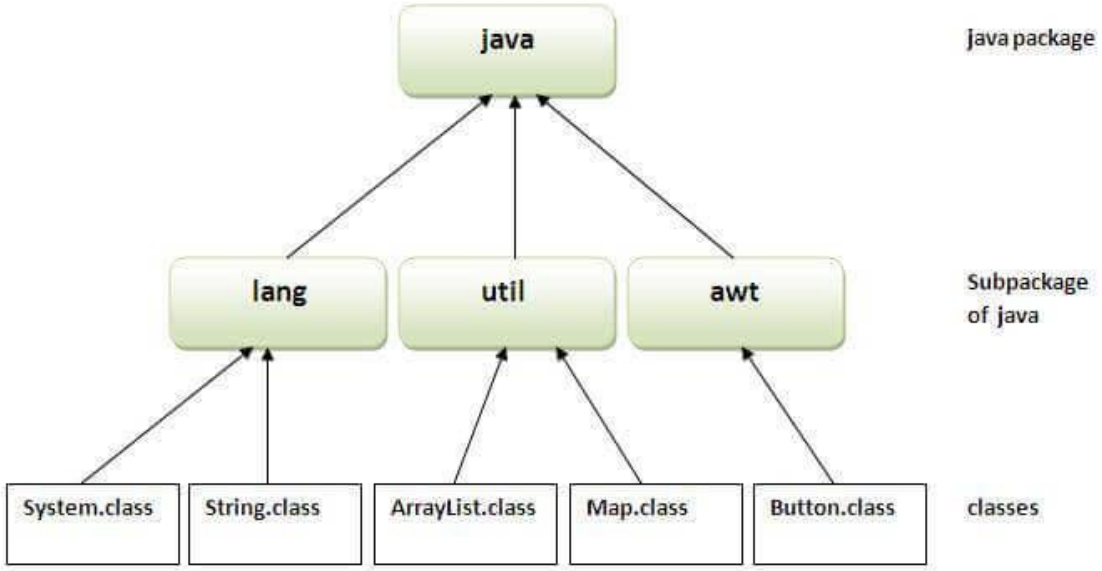
```
import letmecalculate.Calculator;
public class Demo{
  public static void main(String args[]){
        Calculator obj = new Calculator();
        System.out.println(obj.add(100, 200));
  }
}
```

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.



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.