UNIT II – INHERITANCE AND POLYMORPHISM 9

Inheritance: Types - Access rules, super classes and sub classes – Overriding methods - Overriding vs overloading. Polymorphism: Static binding – Dynamic binding – Method overloading - Runtime polymorphism. Package: Create - Import – Exception handling: Exception - Types – Try and catch - Multiple catch - Nested try – throw - throws – finally - User defined exception.

INHERITANCE

Reusability is another concept of OOPS. It is always better to reuse something that already exists than creating it all over again. This is done by creating new classes and using the properties of existing ones. This mechanism of deriving a new class from an old one is called inheritance.

Inheritance is implemented in two ways:

1. Inheriting from classes (Extending classes)
2. Inheriting from interfaces (Implementing interfaces)

Inheriting from classes:

General form of Inheritance:

```
class Derivedclass_name extends Baseclass_name
{
        //Implementation code
}
```

Super classes: The old class from which properties are derived in a new class is called as base class or super class or parent class

Sub classes: The new class which derives properties from an already existing class is called a subclass or child class.

Advantages:

- It provides ideas of reusability.
- Deriving a new class from the existing one. The new class will have the combined features of both the classes.
- The inheritance mechanism allows the programmer to reuse a class.

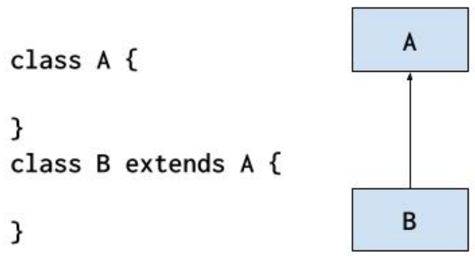
TYPES OF INHERITANCE:

There are four types of Inheritance

1. Single inheritance (only one base class)
2. Multilevel inheritance(derived from an already derived child class)
3. Hierarchical inheritance(one base class, many child classes)
4. Hybrid inheritance(combination of above one inheritance)

1. Single Inheritance:

Single inheritance in Java refers to the inheritance relationship where a subclass extends only one superclass. Here’s an example demonstrating single inheritance.



```
class A {

}
class B extends A {

}
```

It is a type of inheritance in which a child class (or) derived class(or) subclass derives the property of base class (or) super class (or) parent class.

SYNTAX:

```
class childclassname extends baseclass-name
{
Set of statements;
}
```

Example:

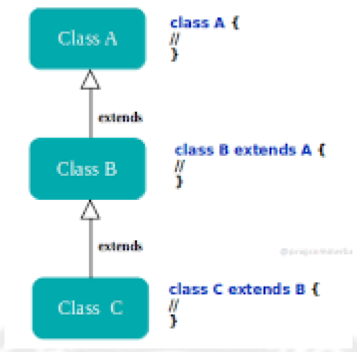
```
class Bicycle
{
        public int gear;
        public int speed;
        public Bicycle(int gear, int speed)
        {
                this.gear = gear;
                this.speed = speed;
        }
        public void applyBrake(int decrement)
        {
                speed -= decrement;
        }
        public void speedUp(int increment)
        {
                speed += increment;
        }
        public String toString()
        {
                return ("No of gears are " + gear + "\n" + "speed of bicycle is " + speed);
        }
}
class MountainBike extends Bicycle
{
        public int seatHeight;
        public MountainBike(int gear, int speed, int startHeight)
        {
                super(gear, speed);
                seatHeight = startHeight;
        }
        public void setHeight(int newValue)
        {
                seatHeight = newValue;
        }
        @Override public String toString()
        {
                return (super.toString() + "\nseat height is "+ seatHeight);
        }
}
public class Test
{
        public static void main(String args[])
        {
                MountainBike mb = new MountainBike(3, 100, 25);
                System.out.println(mb.toString());
        }
}
```

Output:

```
No of gears are 3
speed of bicycle is 100
seat height is 25
```

2. Multilevel Inheritance:

Multilevel inheritance in Java refers to a scenario where a class inherits properties and behaviors from another class, which in turn inherits from another class. This creates a hierarchical structure of classes where each class inherits from the one above it.



Example:

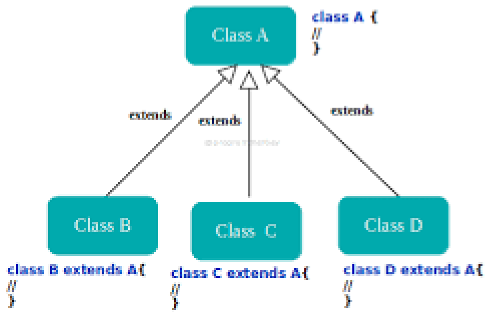
```
class Car
{
        public Car()
        {
                System.out.println("Class Car");
        }
        public void vehicleType()
        {
                System.out.println("Vehicle Type: Car");
        }
}
class Maruti extends Car
{
        public Maruti()
        {
                System.out.println("Class Maruti");
        }
        public void brand()
        {
                System.out.println("Brand: Maruti");
        }
        public void speed()
        {
                System.out.println("Max: 90Kmph");
        }
}
public class Maruti800 extends Maruti
{
        public Maruti800()
        {
                System.out.println("Maruti Model: 800");
        }
        public void speed()
        {
                System.out.println("Max: 80Kmph");
        }
        public static void main(String args[])
        {
                Maruti800 obj=new Maruti800();
                obj.vehicleType();
                obj.brand();
                obj.speed();
        }
}
```

Output:

```
Class Car
Class Maruti
Maruti Model: 800
Vehicle Type: Car
Brand: Maruti
Max: 80Kmph
```

3. Hierarchical Inheritance:

Hierarchical inheritance in Java refers to a scenario where multiple classes inherit properties and behaviors from a single parent class. In this inheritance structure, there is one parent class and multiple child classes that inherit from it.



Example:

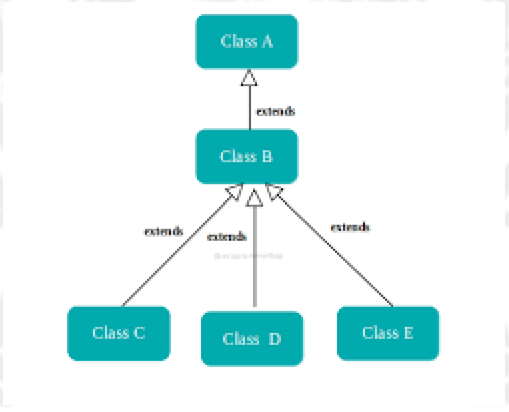
```
class Animal
{
        void eat()
        {
                System.out.println("Animal is eating");
        }
}
// Child class 1 inheriting from Animal
class Dog extends Animal
{
        void bark()
        {
                System.out.println("Dog is barking");
        }
}
// Child class 2 inheriting from Animal
class Cat extends Animal
{
        void meow()
        {
                System.out.println("Cat is meowing");
        }
}
public class Main
{
        public static void main(String[] args)
        {
                Dog dog = new Dog();
                dog.eat();   // Inherited from Animal
                dog.bark();  // Defined in Dog class
                Cat cat = new Cat();
                cat.eat();   // Inherited from Animal
                cat.meow();  // Defined in Cat class
        }
}
```

Output:

```
Animal is eating
Dog is barking
Animal is eating
Cat is meowing
```

3. Hybrid Inheritance:

Hybrid inheritance in Java refers to a combination of multiple inheritance and hierarchical inheritance. In hybrid inheritance, a class is derived from two or more classes, and these derived classes can further have their own subclasses. Java doesn't support multiple inheritance directly due to the diamond problem, but hybrid inheritance can be achieved by combining hierarchical inheritance and interface implementation.



Example:

```
class Animal
{
        void eat()
        {
                System.out.println("Animal is eating");
        }
}
class Dog extends Animal
{
        void bark()
        {
                System.out.println("Dog is barking");
        }
}
class Cat extends Animal
{
        void meow()
        {
                System.out.println("Cat is meowing");
        }
}
interface Domestic
{
        void play();
}
interface DogBehavior extends Domestic
{
        void guard();
}
class DomesticDog implements DogBehavior
{
        public void play()
        {
                System.out.println("Domestic dog is playing");
        }
        public void guard()
        {
                System.out.println("Domestic dog is guarding");
        }
}
public class Main
{
        public static void main(String[] args)
        {
                DomesticDog dog = new DomesticDog();
                dog.play();    // Defined in Domestic interface
                dog.guard();   // Defined in DogBehavior interface
        }
}
```

Output

```
Domestic dog is playing
Domestic dog is guarding
```

Multiple Inheritance

Multiple Inheritance is a feature of an object-oriented concept, where a class can inherit properties of more than one parent class. The problem occurs when there exist methods with the same signature in both the superclasses and subclass. On calling the method, the compiler cannot determine which class method to be called and even on calling which class method gets the priority. Multiple inheritance is implemented in java using Interface

Interface in Java

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 and multiple inheritance.

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

Syntax:

```
interface <interface_name>
{
        //declare constant fields
        //declare methods that abstract by default.
}
```

Example:

```
interface IPrintable
{
        void print();
}
class A6 implements IPrintable
{
        public void print()
        {
                System.out.println("Hello");
```

```
	}
	public static void main(String args[])
	{
		A6 obj = new A6();
		obj.print();
	}
}
```

Output:

```
Hello
```

Example for Multiple inheritance

```
interface ICharacter
{
	void attack();
}
interface IWeapon
{
	void use();
}
class Warrior implements Character, Weapon
{
	public void attack()
	{
		System.out.println("Warrior attacks with a sword.");
	}
	public void use()
	{
		System.out.println("Warrior uses a sword.");
	}
}
class Mage implements Character, Weapon
{
	public void attack()
	{
		System.out.println("Mage attacks with a wand.");
	}
	public void use()
	{
```

```
                System.out.println("Mage uses a wand.");
        }
}
public class MultipleInheritance
{
        public static void main(String[] args)
        {
                Warrior warrior = new Warrior();
                Mage mage = new Mage();
                warrior.attack(); // Output: Warrior attacks with a sword.
                warrior.use(); // Output: Warrior uses a sword.
                mage.attack(); // Output: Mage attacks with a wand.
                mage.use(); // Output: Mage uses a wand.
        }
}
```

Output:

```
Warrior attacks with a sword.
Warrior uses a sword.
Mage attacks with a wand.
Mage uses a wand.
```

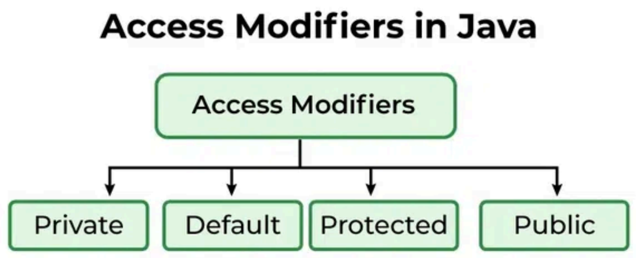
ACCESS RULES IN INHERITANCE

In Java, Access modifiers help to restrict the scope of a class, constructor, variable, method, or data member. It provides security, accessibility, etc. to the user depending upon the access modifier used with the element. In this article, let us learn about Java Access Modifiers, their types, and the uses of access modifiers.

Types of Access Modifiers

There are 4 types of access modifiers available in Java:

1. Default – No keyword required
2. Private
3. Protected
4. Public



1. Default Access Modifier

When no access modifier is specified for a class, method, or data member, it is said to be having the default access modifier by default. The default access modifiers are accessible only within the same package.

2. Private Access Modifier

The private access modifier is specified using the keyword private. The methods or data members declared as private are accessible only within the class in which they are declared. Any other class of the same package will not be able to access these members. Top-level classes or interfaces can not be declared as private because, private means "only visible within the enclosing class". protected means "only visible within the enclosing class and any subclasses".

```
package p1;
class A
{
        private void display()
        {
                System.out.println("Inside Private");
        }
}
class B
{
        public static void main(String args[])
        {
                A obj = new A();
                obj.display();
        }
}
```

3. Protected Access Modifier

The protected access modifier is specified using the keyword protected. The methods or data members declared as protected are accessible within the same package or subclasses in different packages.

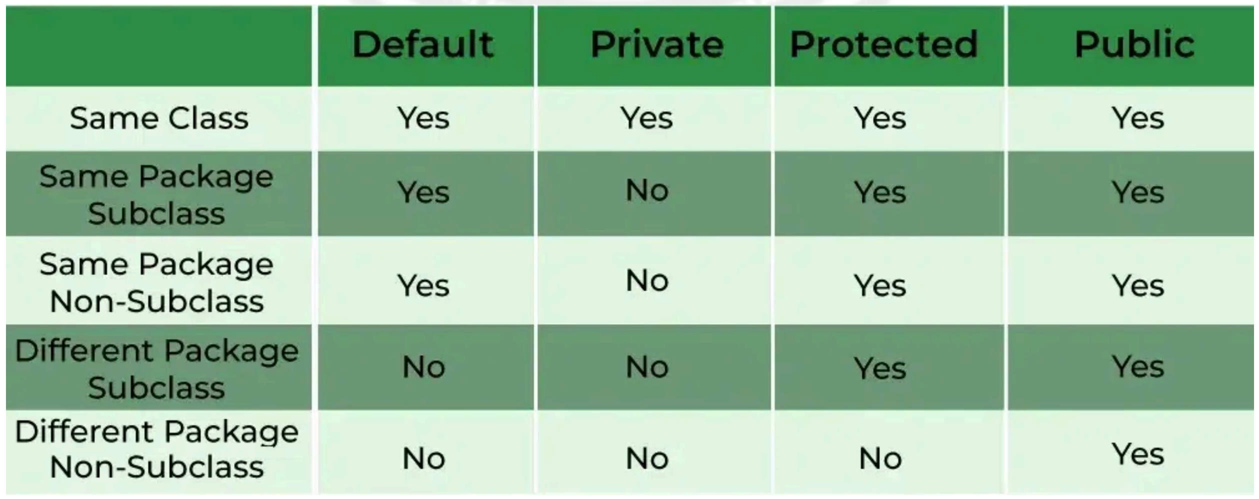
```
package p1;
public class A
{
        protected void display()
        {
                System.out.println("Inside Protected");
        }
}
```

Public Access Modifier

The public access modifier is specified using the keyword public. The public access modifier has the widest scope among all other access modifiers. Classes, methods, or data members that are declared as public are accessible from everywhere in the program. There is no restriction on the scope of public data members.

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

Comparison Table of Access Modifiers in Java



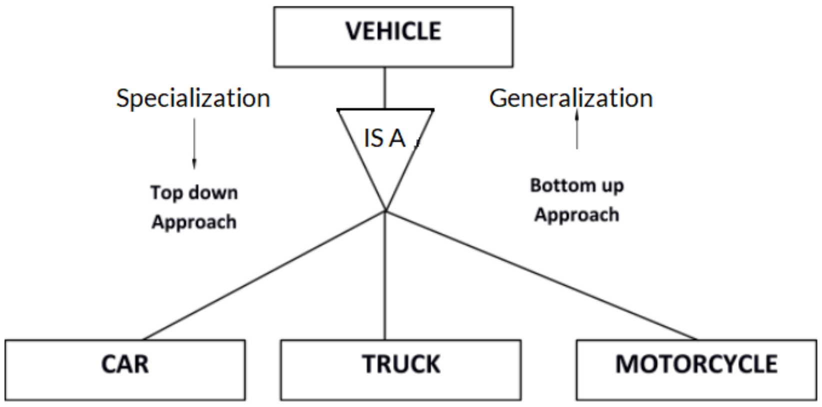
	Default	Private	Protected	Public
Same Class	Yes	Yes	Yes	Yes
Same Package Subclass	Yes	No	Yes	Yes
Same Package Non-Subclass	Yes	No	Yes	Yes
Different Package Subclass	No	No	Yes	Yes
Different Package Non-Subclass	No	No	No	Yes

Superclasses

A superclass is the class from which many subclasses can be created. The subclasses inherit the characteristics of a superclass. The superclass is also known as the parent class or base class.

Subclasses

A subclass is a class derived from the superclass. It inherits the properties of the superclass and also contains attributes of its own. An example is:



"super" KEYWORD

Usage of super keyword

- super() invokes the constructor of the parent class.
- super.variable_name refers to the variable in the parent class.
- super.method_name refers to the method of the parent class.

1. super() invokes the constructor of the parent class

super() will invoke the constructor of the parent class. Even when you don't add super() keyword the compiler will add one and will invoke the Parent Class constructor.

Example:

```
class ParentClass
{
        ParentClass()
        {
                System.out.println("Parent Class default Constructor");
        }
}
public class SubClass extends ParentClass
{
        SubClass()
```

```
	{
		System.out.println("Child Class default Constructor");
	}
	public static void main(String args[])
	{
		SubClass s = new SubClass();
	}
}
```

Output:

```
Parent Class default Constructor
Child Class default Constructor
```

Even when we add explicitly also it behaves the same way as it did before.

```
class ParentClass
{
	public ParentClass()
	{
		System.out.println("Parent Class default Constructor");
	}
}
public class SubClass extends ParentClass
{
	SubClass()
	{
		super();
		System.out.println("Child Class default Constructor");
	}
	public static void main(String args[])
	{
		SubClass s = new SubClass();
	}
}
```

Output:

```
Parent Class default Constructor
Child Class default Constructor
```

You can also call the parameterized constructor of the Parent Class. For example, super(10) will call a parameterized constructor of the Parent class.

```
class ParentClass
{
        ParentClass()
        {
                System.out.println("Parent Class default Constructor called");
        }
        ParentClass(int val)
        {
                System.out.println("Parent Class parameterized Constructor, value: "+val);
        }
}
public class SubClass extends ParentClass
{
        SubClass()
        {
                super();
                System.out.println("Child Class default Constructor called");
        }
        SubClass(int val)
        {
                super(10);
                System.out.println("Child Class parameterized Constructor, value: "+val);
        }
        public static void main(String args[])
        {
                SubClass s = new SubClass();
                SubClass s1 = new SubClass(10);
        }
}
```

Output

```
Parent Class default Constructor called
Child Class default Constructor called
Parent Class parameterized Constructor, value: 10
Child Class parameterized Constructor, value: 10
```

2. super.variable_name refers to the variable in the parent class

When we have the same variable in both parent and subclass

```
class ParentClass
{
	int val=999;
}
public class SubClass extends ParentClass
{
	int val=123;
	void disp()
	{
		System.out.println("Value is : "+val);
	}
	public static void main(String args[])
	{
		SubClass s = new SubClass();
		s.disp();
	}
}
```

Output

```
Value is : 123
```

This will call only the val of the subclass only. Without super keyword, you cannot call the val which is present in the Parent Class.

```
class ParentClass
{
	int val=999;
}
public class SubClass extends ParentClass
{
	int val=123;
	void disp()
	{
		System.out.println("Value is : "+super.val);
	}
	public static void main(String args[])
	{
		SubClass s = new SubClass();
```

```
                s.disp();
        }
}
```

Output

```
Value is : 999
```

3. super.method_nae refers to the method of the parent class

When you override the Parent Class method in the Child Class without super keywords support you will not be able to call the Parent Class method. Let's look into the below example

```
class ParentClass
{
        void disp()
        {
                System.out.println("Parent Class method");
        }
}
public class SubClass extends ParentClass
{
        void disp()
        {
                System.out.println("Child Class method");
        }
        void show()
        {
                disp();
        }
        public static void main(String args[])
        {
                SubClass s = new SubClass();
                s.show();
        }
}
```

Output:

```
Child Class method
```

Here we have overridden the Parent Class disp() method in the SubClass and hence SubClass disp() method is called. If we want to call the Parent Class disp() method also means then we have to use the super keyword for it.

```
class ParentClass
{
        void disp()
        {
                System.out.println("Parent Class method");
        }
}
public class SubClass extends ParentClass
{
        void disp()
        {
                System.out.println("Child Class method");
        }
        void show()
        {
                //Calling SubClass disp() method
                disp();
                //Calling ParentClass disp()
                method super.disp();
        }
        public static void main(String args[])
        {
                SubClass s = new SubClass();
                s.show();
        }
}
```

Output

```
Child Class method
Parent Class method
```

When there is no method overriding then by default Parent Class disp() method will be called.

```
class ParentClass
{
        public void disp()
        {
                System.out.println("Parent Class method");
        }
}
public class SubClass extends ParentClass
{
        public void show()
        {
                disp();
        }
        public static void main(String args[])
        {
                SubClass s = new SubClass();
                s.show();
        }
}
```

Output:

```
Parent Class method
```