4. INHERITANCE

1. Definition :

Inheritance in java is a mechanism in which a class aquires the properties of another class. The idea behind inheritance in java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of parent class, and you can add new methods and fields also.
Inheritance represents the **IS-A relationship**, also known as parent-child relationship. Here a class which is extended is known as Base class or Parent class or Super class The class which is inheriting the properies is known as Derived class or Child class or Sub class.

The subclass can directly access the super class properties. Even it can access protected members of super class from anywhere.

2. Use of inheritance:

Why use inheritance in java

- **For Code Reusability:** The extended class does not have to repeat the existing properties of its parent.
- **For changing the existing behaviour:** If in case child needs to change the existing behaviour of its parent then it can do by overriding the methods of its parent. Thus it achieves run time polymorphism

Syntax :

```
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.

Example:
Programmer is the subclass and Employee is the superclass. Relationship between two classes is **Programmer IS-A Employee**.It means that Programmer is a type of Employee. But Employee is not a type of Programmer.

```
class Employee{
        float salary=40000;

}

public class Programmer extends Employee{

        int bonus=10000;
```

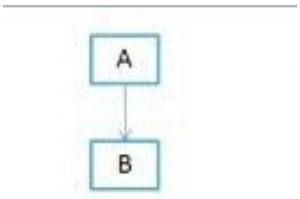
```
	public static void main(String args[]){

		Programmer p=new Programmer();
	//	p.getSalary();	// subclass can access properties of superclass
	//	p.getBonus();
		System.out.println("Salary : "+p.salary);
		System.out.println("Bonus : "+p.bonus);
	}
}
```

Output:
Salary : 40000.0
Bonus : 10000

3. Types of inheritance:

On the basis of class, there can be three types of inheritance in java: single, multilevel, hierarchical, multiple (java does not support) and hybrid (java does not support).

1.Single Inheritance :
When a class extends another one class only then we call it a single inheritance. The below flow diagram shows that class B extends only one class which is A. Here A is a **parent class** of B and B would be a **child class** of A.



(a) Single Inheritance

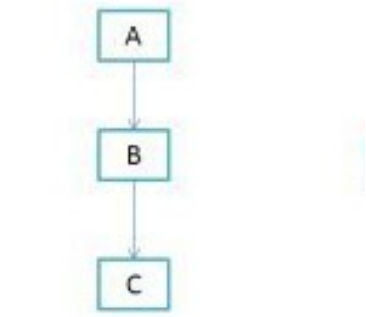
Example:

```
class Employee{
	float salary=40000;
	public void getSalary() {
		System.out.println("Salary : "+40000);
	}
}

public class Programmer extends Employee{
	int bonus=10000;
	public void getBonus() {
```

```
            System.out.println("Bonus : "+bonus);
        }
        public static void main(String args[]) {
            Programmer p=new Programmer();
            p.getSalary();      // subclass can access properties of superclass
            p.getBonus();
        }
}
```

Output:
Salary : 40000
Bonus : 10000

2. Multilevel Inheritance :
Multilevel inheritance refers to a mechanism where a class can inherit a derived class, thereby making this derived class the base class for the new class. As you can see in below flow diagram C is subclass or child class of B and B is a child class of A.



(d) Multilevel Inheritance

Example:
In this example we have three classes – Car, Maruti and Maruti800. We have done a setup – class Maruti extends Car and class Maurit800 extends Maurti. With the help of this Multilevel hierarchy setup our Maurti800 class is able to use the methods of both the classes (Car and Maruti).

```
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");
    }
}

class Maruti800 extends Maruti{

     public Maruti800() {
         System.out.println("Maruti Model: 800");
     }
     public void speed() {

          System.out.println("Max: 80Kmph");
     }
}
public class MultilevelInheritance {

          public static void main(String[] args) {

          Maruti800 obj=new Maruti800();
          obj.vehicleType();
          obj.brand();
          obj.speed();
      }
}
```

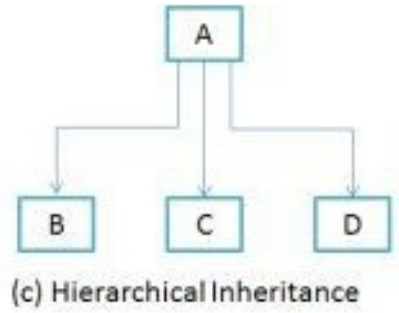
Output :

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

3.Hierarchical Inheritance :

In such kind of inheritance one class is inherited by many **sub classes**. In below example class B,C and D **inherits** the same class A. A is **parent class (or base class)** of B,C & D.

4. INHERITANCE



(c) Hierarchical Inheritance

```
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");
    }
}

class BMW extends Car{

     public BMW() {
         System.out.println("Class BMW");
     }
     public void engine() {
          System.out.println("Engine: 4 stroke");
                                                    }
```

```
}
public class HierarchicalInheritance {

        public static void main(String[] args) {

                System.out.println("Subclass Maruti");
                Maruti m=new Maruti();
                m.vehicleType();
                m.brand();
                m.speed();

                System.out.println("Subclass BMW");
                BMW b=new BMW();
                b.vehicleType();
                 b.engine();

        }
}
```

Output:

```
Subclass Maruti
Class Car
Class Maruti
Vehicle Type: Car
Brand: Maruti
Max: 90Kmph
Subclass BMW
Class Car
Class BMW
Vehicle Type: Car
Engine: 4 stroke
```

4. Multiple Inheritance :

The type of inheritance where child class may have multiple parents but this type of inheritance is not supported by java.

5. Hybrid inheritance:

It is a combination of all types of inheritances but this is also not supported by java.

Note: Java does not support Multiple Inheritance as if parents have method with same name

and same type signatures then it creates ambiguity when subclass tries to access that method as it gets confused which method to call.

4. Use of super keyword:

The subclass can refer its immediate super class by using super keyword.

There are three uses of super keyword:

1. to refer super class constructor from subclass constructor

- syntax: super(paramaters);
- it should be first statement in sub class constructor.
- constructors can be refered only from other constrcutors

2. to access super class instance variables from subclass

- syntax: super.variable-name
- use: generally it is used when subclass and superclass have same instance variable name and subclass want to differentiate between those.

3. to call super class methods from subclass

- syntax: super.method-name(parameters);
- use: generally it is used to call overriden methods of superclass from subclass.

Example:

```
class Person
{
	int id;
	String name;
	String organization;
	protected String gender;

	Person() {

	}

	Person(int id, String name, String organization, String gender) {
		this.id=id;
		this.name=name;
		this.organization=organization;
		this.gender=gender;
	}
```

```
	public void displayData() {
System.out.println("Id:"+id+"\nname:"+name+"\norganization:"+organization+"\ngender:"+gender)
;
	}
}

class Employee extends Person
{
	double basicsalary;
	int casualleaves;
	String gender;


	Employee(int id, String name, String organization, String gender) {
		super(id,name,organization,gender);
	}
	public void setData(double basicsalary, int casualleaves) {
		this.basicsalary=basicsalary;
		this.casualleaves=casualleaves;
	}
	public void setGender(String gender) {
		super.displayData();
		this.gender=gender;
		super.gender=this.gender;
		System.out.println("\nAfter Changing the gender\n");

		super.displayData();
	}
	public void displatEmpData() {
		System.out.println("basic salary:"+basicsalary+"\nCasual leaves:"+casualleaves);
	}
}

public class SuperDemo {

	public static void main(String[] args) {

		Employee e=new Employee(101,"Vilas Sangre","Coder Technologies","Male");
		e.setGender("Female");
		e.setData(45000,15);
		e.displatEmpData();
	}
```

```
}
```

Output:

```
Id:101
name:Vilas Sangre
organization:Coder Technologies
gender:Male

After Changing the gender

Id:101
name:Vilas Sangre
organization:Coder Technologies
gender:Female
basic salary:45000.0
Casual leaves:15
```

Superclass variable can refer subclass object but it can not access subclass own properties directly.

Example:

```
Person p=new Employee(101,"Vilas Sangre","Coder Technologies","Male");

        p.displayData();    // right
        p.setGender("Female");   // wrong
        p.setData(45000,15);      // wrong
        p.displatEmpData();     // wrong
```

for that we need to again downcast it. Please refer up casting and down casting in the end of this chapter.

5. Method Overriding:

If a subclass want to change the original property of superclass. It can use the same method signature and can add its own implementation.

When in superclass as well as subclass, there is same method name and type signatures then by default, a subclass method overrides the superclass method.

It is runtime polymorphism. Call to an overridden method is decided at run time and call goes to subclass same method.

Example:

```
class Organization
```

```
{
        int probessionperiod;
        int casualleaves;
        String qualification;

        public void setPolicy() {
                probessionperiod=1;
                casualleaves=15;
                qualification="M.E.";
        }
        public void getPolicy() {
                System.out.println("Probession period will be: "+probessionperiod+" months");
                System.out.println("Casual leaves are: "+casualleaves+" in a year");
                System.out.println("Qualification must be: "+qualification);
        }
}

class TrainingOrganization extends Organization
{
        String certification;
        public void setPolicy() {
                probessionperiod=2;
                casualleaves=8;
                qualification="Degree";
                certification="J2EE";
        }
}

public class OverridingDemo {

        public static void main(String[] args) {

                Organization to=new TrainingOrganization();   // dynamic binding
                to.setPolicy();                               // call goes to subclass method
                to.getPolicy();

                TrainingOrganization t=(TrainingOrganization)to;   // downcasting
                t.getCertification();
        }
}
```

Output:

Probession period will be: 2 months
Casual leaves are: 8 in a year
Qualification must be: Degree
Certification should be:J2EE

4. INHERITANCE

5.1. Rules while overriding the method.

When a subclass is overriding the method of superclass then following factors must be considered for overriding method in subclass.

1. **Access modifier / Scope :** must not be more restrictive than original method i.e. a sub class can not reduce visibility of overriden method of superclass
2. **Return type :** there are following rules if return type is
    1. **primitive :** should be same like original method
    2. **non-primitive :** co-variants are allowed. Means if super class' method is returning any object then subclass' overrding method can return object of its subclass type. It is added from Java5.
3. **method name :** should be same as overriden method
4. **type signatures :** both the number of parameters and data type of parameters should be same as original method
5. **exception handling :**
    1. **If the superclass method does not declare an exception :** If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but it can declare unchecked exception.
    2. **If the superclass method declares an exception :** If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception.

Example :

1. Scope :

```
class Father {
        public void dream(){}
}

class Son extends Father {
        protected void dream(){}            // wrong : scope is restricted from public to protected
}
```

2. return type : primitive

```
class Father {
        public void dream(){}
}

class Son extends Father {
        public int dream(){}            // wrong : return type is incompatible with Father.dream()
}
```

3. return type : covariants

```
class Father {
```

```
        public Number dream(){
                return null;
        }
}

class Son extends Father {
        public Integer dream(){     // right:Integer is subclass of Number.  i.e. co-variants
                return null;
        }
}
```

4. exception throwing

```
import java.io.IOException;
class Father {
        public void dream(){}
}

class Son extends Father {
        public void dream()throws IOException    // wrong : see the rule 5.1
        {}
}

import java.io.FileNotFoundException;
import java.io.IOException;
class Father {
        public void dream()throws IOException
        {}
}

class Son extends Father {
        public void dream()throws FileNotFoundException   // right : see the rule 5.2
        {}
}
```

6. Resolving Method Overriding:

If we need to call both the methods, we need to resolve the overriding. There are two ways to resolve.

Resolving a call to an overridden method

1. at compile time – using super keyword
2. at run time – using dynamic method dispatch

1. Resolving at compile time :

4. INHERITANCE

We can call the superclass method from sub class method using super keyword.

Example:

Organizations.java

```
class Organizations {

	int probessionperiod;
	int casualleaves;
	String qualification;

	public void setPolicy() {
		probessionperiod=1;
		casualleaves=15;
		qualification="M.E.";
	}
	public void getPolicy() {
		System.out.println("Probession period will be: "+probessionperiod+" months");
		System.out.println("Casual leaves are: "+casualleaves+" in a year");
		System.out.println("Qualification must be: "+qualification);
	}
}
```

TrainingOrganizations.java

```
class TrainingOrganizations extends Organizations
{
	String certification;
	public void setPolicy() {
		super.setPolicy();   // call goes to superclass method
		super.getPolicy();

		System.out.println("new policies are:"):
		probessionperiod=2;
		casualleaves=8;
		qualification="Degree";
		certification="J2EE";
	}
	public void getCertification() {
		System.out.println("Certification should be:"+certification);
	}
}
```

CompileTimeResolvingOverriding.java

```
public class CompileTimeResolvingOverriding {

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

```
                Organizations to=new TrainingOrganizations();   // dynamic binding , upcasting
                to.setPolicy();    // call goes to subclass method
                to.getPolicy();

                TrainingOrganizations t=(TrainingOrganizations)to;   // downcasting
                t.getCertification();
        }
}
```

2. Resolving at run time : Dynamic Method Dispatch:

Definition of Dynamic Method Dispatch : Calling the methods dynamically by changing the object which is refered.

We can resolve the call to an overridden method at runtime by calling the methods dynamically, by referring the objects dynamically. Hence there is no need of super keyword . This way is used when

1. We want to achieve Run time polymorphism
2. We can not change the implementation of subclass methods by adding the statement of super.

Then the call to above methods from main will be:

```
public class DynamicMethodDispatch {
        public static void main(String[] args) {
                Organization or;
                or=new Organization();
                or.setPolicy();
                or.getPolicy();
                or=new TrainingOrganization();   // upcasting, dynamic binding
                or.setPolicy();                                  // dynamic method dispatch
                or.getPolicy();
        }
}
```

7. Up casting and Down casting:

In upcasting, children is casted with its parent and it downcasting parent is casted with its children. Super class variable can refer its subclass object directly in up casting, but subclass variable can not directly refer its super class object. If a parent is casted with its children, it throws **ClassCastException.**

Example:

Super class name: Organization

Sub class name: TrainingOrganization

Up casting:

```
Organization or;
or=new TrainingOrganization();
```

It is simliar to or=(Organization)**new** TrainingOrganization();

Down casting:

```
1) TrainingOrganization to;
to=(TrainingOrganization)or;    // superclass reference variable is casted with subclass
2) to=(TrainingOrganization) new Organization();    // superclass object is casted with
```

subclass but is gives **ClassCastException**.

Reason why above downcasting is not supported:

Every TraininOrganization can be an Organization. So we can cast subclass object with superclass. But every Organization can not be a TrainingOrganization so we can not cast the superclass object with subclass. Thus the downcasting given in point 2 is not supported.

8. IS-A relationship and HAS-A relationship:

1. IS-A relationship:

This refers to inheritance or implementation. Expressed using keyword “extends”. Main advantage is code reusability.

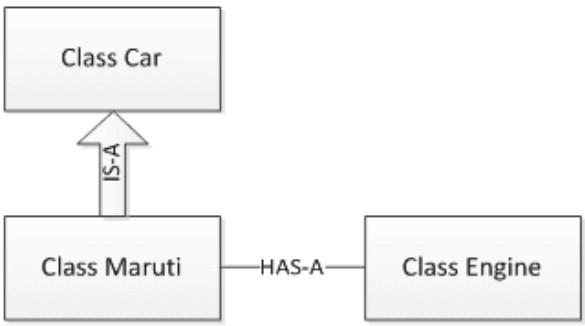
2. HAS-A relationship:

An instance of one class “has a” reference to an instance of another class or another instance of same class. It is also known as “composition” or “aggregation”. There is no specific keyword to implement HAS-A relationship but mostly we are depended upon “new” keyword.

Composition: A “university” has several “departments”.Without existence of “university” there is no chance for the “departments” to exist. Hence “university” and “departments” are strongly associated and this strong association is known as *composition*.

Aggregation: “department” has several “professors”. Without existence of “departments” there is good chance for the “professors” to exist. Hence “professors” and “department” are loosely associated and this loose association is known as *Aggregation*.

Example:



4. INHERITANCE

Maruti **IS A** Car. Maruti **HAS An** Engine.

```
class Car{
  // body of class
}
```

```
IS A:        Class Maruti extends Car        // IS A relation ship
             {
HAS  A:              Engine e;               // HAS A relationship
             }
```

```
class Engine{

}
```

Thus by inheritance we achieve IS-A relationship and if one object refers another object then we achieve HAS-A relationship.

9. Difference between Method Overloading and Method Overriding

Method Overloading	Method Overrding
Same name different type signatures (either number of parameters or datatype is different)	Same name and same type signatures(both number of parameters and datatype should be same). here subclass method overrides superclass method
It can be in same class or in inheried class	It is possible only in inheritance
It is compile time polymorphism	It is run time polymorphism
Access modifier and return type does not matter while overloading the methods as compiler only checks the parameter matching	Access modifier and return type matters while overrding the method. There are certain rules reagarding giving scope and return type in sub class
It increases readability of code	It increases reusability of code

10. Important Points to remember:

1. In inheritance, calling sequence of constructors in always from superclass constructor to subclass constructor.
2. The static methods can not be overriden as these are class methods and overrding works on instance at run time. We can have same static method in subclass but the methods are statically bounded with their class name. Thus calling one methods hides other method. This concept is method over-hiding but not overrding.

3. The private methods can not be overriden as private methods are not accessible in subclass. These can be accessed within the superclass by its own object. We can have same method in subclass but it will be its independant copy.
4. The contructors can not be overriden as the constructor has same name as classname.

ASSIGNMENTS

1. Extend the Vehical class by Car class and Car class by Duster class. Inherit the properties from Vehical and add new properties in Car and Duster.
2. Show is use of super keyword in Car and Duster.
3. Replace some exisitng properties of Vehical in Car and Replace the some existing properties of Car in Duster.
4. Show the use of Dynamic Method Disptach in above inheritance.
5. Restrict the class Duster to extend further.
6. Restrict any of the method of Car to override.
7. Implement the Has-A relationship in between Car class and Engine class.