5.1 Introduction

Inheritance is a mechanism in which one object acquires all the properties and behaviours of a parent object. It is an important part of OOPs (Object Oriented programming system).

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 the parent class. Moreover, you can add new methods and fields in your current class also.

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

We use inheritance for:

- Method Overriding so that runtime polymorphism can be achieved
- 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

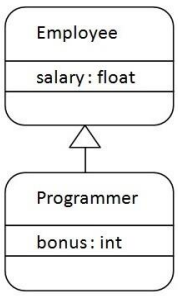
```
class Subclass-name extends Superclass-name
{
  //methods and fields
}
```

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.

In the terminology of Java, a class which is inherited is called a parent or superclass, and the new class is called child or subclass.

Example

As displayed in the figure, Programmer is the subclass and Employee is the superclass. The relationship between the two classes is **Programmer IS-A Employee**. It means that Programmer is a type of Employee.




```
class Employee{
      float salary=40000;
}
class Programmer extends Employee{
      int bonus=10000;
      public static void main(String args[]){
            Programmer p=new Programmer();
            System.out.println("Programmer salary is:"+p.salary);
            System.out.println("Bonus of Programmer is:"+p.bonus);
      }
}
```

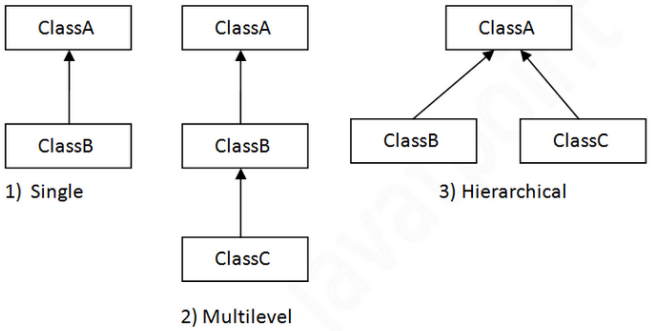
```
Programmer salary is:40000.0
Bonus of programmer is:10000
```

In the above example, Programmer object can access the field of own class as well as of Employee class i.e. code reusability.

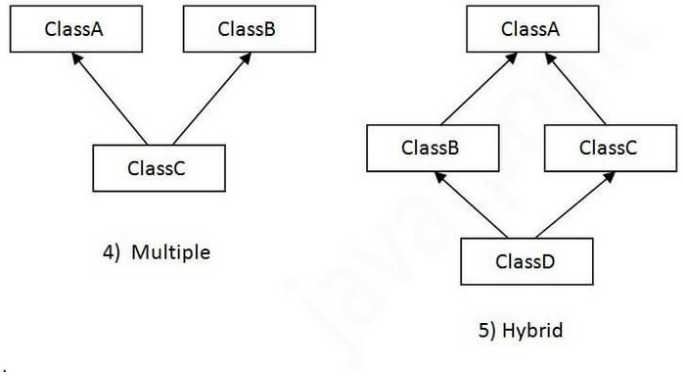
5.2 Types of Inheritance

On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.

In java programming, multiple and hybrid inheritance is supported through interface only.



When one class inherits multiple classes, it is known as multiple inheritance. For Example:



Single Inheritance Example

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.

File: TestInheritance.java

```
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...

Multilevel Inheritance Example

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.

File: TestInheritance2.java

```
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 Example

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.

File: TestInheritance3.java

```
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(); // Compilation Error
   }
}
```

Output:
meowing...
eating...

Q) Why multiple inheritance is not supported in java?

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. So whether you have same method or different, there will be compile time error.

```
class A {
   void msg() {
      System.out.println("Hello");
   }
}

class B {
   void msg() {
      System.out.println("Welcome");
   }
}

class C extends B implements A {
   public static void main(String args[]) {
      C obj = new C();
      obj.msg(); // This will invoke the msg() method from class B
   }
}
```

Compile Time Error

5.3 Method Overriding

if subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java**.

In other words, If a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding.

Usage of Java Method Overriding

- Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
- Method overriding is used for runtime polymorphism

Rules for Java Method Overriding

1. The method must have the same name as in the parent class
2. The method must have the same parameter as in the parent class.
3. There must be an IS-A relationship (inheritance).

Understanding the problem without method overriding
Let's understand the problem that we may face in the program if we don't use method overriding.

```
// Java Program to demonstrate why we need method overriding
// Here, we are calling the method of the parent class with a child
// class object.

// Creating a parent class
class Vehicle {
    void run() {
        System.out.println("Vehicle is running");
    }
}

// Creating a child class
class Bike extends Vehicle {
    public static void main(String args[]) {
        // creating an instance of the child class
        Bike obj = new Bike();
        // calling the method with a child class instance
        obj.run();
    }
}
```

Output:
Vehicle is running

Problem is that I have to provide a specific implementation of run() method in subclass that is why we use method overriding.

Example of method overriding

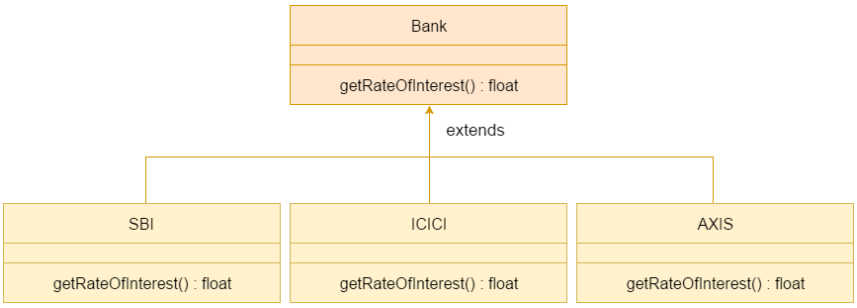
In this example, we have defined the run method in the subclass as defined in the parent class but it has some specific implementation. The name and parameter of the method are the same, and there is IS-A relationship between the classes, so there is method overriding.

```
// Java Program to illustrate the use of Java Method Overriding
// Creating a parent class.
class Vehicle {
    // defining a method
    void run() {
        System.out.println("Vehicle is running");
    }
}

// Creating a child class
class Bike2 extends Vehicle {
    // defining the same method as in the parent class
    void run() {
        System.out.println("Bike is running safely");
    }

    public static void main(String args[]) {
        Bike2 obj = new Bike2(); // creating object
        obj.run(); // calling method
    }
}
```

Output:
Bike is running safely

Example:

Consider a scenario where Bank is a class that provides functionality to get the rate of interest. However, the rate of interest varies according to banks. For example, SBI, ICICI and AXIS banks could provide 8%, 7%, and 9% rate of interest.



Java method overriding is mostly used in Runtime Polymorphism which we will learn in next pages.

```
// Java Program to demonstrate the real scenario of Java Method Overriding
// where three classes are overriding the method of a parent class.

// Creating a parent class.
class Bank {
   int getRateOfInterest() {
      return 0;
   }
}

// Creating child classes.
class SBI extends Bank {
   int getRateOfInterest() {
      return 8;
   }
}

class ICICI extends Bank {
   int getRateOfInterest() {
      return 7;
   }
}

class AXIS extends Bank {
   int getRateOfInterest() {
      return 9;
   }
}

// Test class to create objects and call the methods
class Test2 {
   public static void main(String args[]) {
      SBI s = new SBI();
      ICICI i = new ICICI();
      AXIS a = new AXIS();

      System.out.println("SBI Rate of Interest: " + s.getRateOfInterest());
      System.out.println("ICICI Rate of Interest: " + i.getRateOfInterest());
      System.out.println("AXIS Rate of Interest: " + a.getRateOfInterest());
   }
}
```

Output:

```
SBI Rate of Interest: 8
ICICI Rate of Interest: 7
AXIS Rate of Interest: 9
```

Can we override static method?

No, a static method cannot be overridden. It can be proved by runtime polymorphism, so we will learn it later.

Why can we not override static method?

It is because the static method is bound with class whereas instance method is bound with an object. Static belongs to the class area, and an instance belongs to the heap area.

Can we override java main method?

No, because the main is a static method.

Difference between method overloading and method overriding in java

There are many differences between method overloading and method overriding in java. A list of differences between method overloading and method overriding are given below:

Method Overloading	Method Overriding
Method overloading is used *to increase the readability* of the program.	Method overriding is used *to provide the specific implementation* of the method that is already provided by its super class.
Method overloading is performed *within class*.	Method overriding occurs *in two classes* that have IS-A (inheritance) relationship.
In case of method overloading, *parameter must be different*.	In case of method overriding, *parameter must be same*.
Method overloading is the example of *compile time polymorphism*.	Method overriding is the example of *run time polymorphism*.
In java, method overloading can't be performed by changing return type of the method only. *Return type can be same or different* in method overloading. But you must have to change the parameter.	*Return type must be same or covariant* in method overriding.

5.4 Using Super Keyword

- Whenever a subclass needs to **refer to its immediate superclass,** it can do so by use of the keyword super.
  - super has two general forms –
  - The first calls the superclass' constructor – super();
  - The second is used to access a member of the superclass that has been hidden by a member a subclass (similar to **this** referring object of current class) – super.member;
- A subclass can call a constructor defined by its superclass by use of the following form of super:

  super(arg-list);

  Here, arg-list specifies any arguments needed by the constructor in the superclass.
- **super**( ) must **always be the first statement executed inside a subclass' constructor.**
- When a subclass calls super( ), it is **calling the constructor of its immediate superclass.**
- Thus, super( ) always refers to the **superclass immediately above the calling class.**
- This is true even in **a multileveled hierarchy**.
- The second form of super acts somewhat like this, except that it always refers to the superclass of the subclass in which it is used.
- This usage has the following general form:

  super.member;
- Here, member can be either a method or an instance variable.
- This second form of super is most applicable to situations in which member names of a subclass hide members by the same name in the superclass.

Usage of Java super Keyword

1) super is used to refer immediate parent class instance variable.

We can use super keyword to access the data member or field of parent class. It is used if parent class and child class have same fields.

```
class Animal{
        String color="white";
```

```
}
class Dog extends Animal{
        String color="black";
        void printColor(){
                System.out.println(color);//prints color of Dog class
                System.out.println(super.color);//prints color of Animal class
        }
}
class TestSuper1{
        public static void main(String args[]){
                Dog d=new Dog();
                d.printColor();
}}
```

Output:

black
white

In the above example, Animal and Dog both classes have a common property color. If we print color property, it will print the color of current class by default. To access the parent property, we need to use super keyword.

2) super can be used to invoke parent class method

The super keyword can also be used to invoke parent class method. It should be used if subclass contains the same method as parent class. In other words, it is used if method is overridden.

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
        void eat(){System.out.println("eating bread...");}
        void bark(){System.out.println("barking...");}
        void work(){
                super.eat();
                bark();
        }
}
class TestSuper2{
public static void main(String args[]){
Dog d=new Dog();
d.work();
}}
```

Output:

eating...
barking...

In the above example Animal and Dog both classes have eat() method if we call eat() method from Dog class, it will call the eat() method of Dog class by default because priority is given to local.

To call the parent class method, we need to use super keyword.

3) super is used to invoke parent class constructor.

The super keyword can also be used to invoke the parent class constructor. Let's see a simple example:

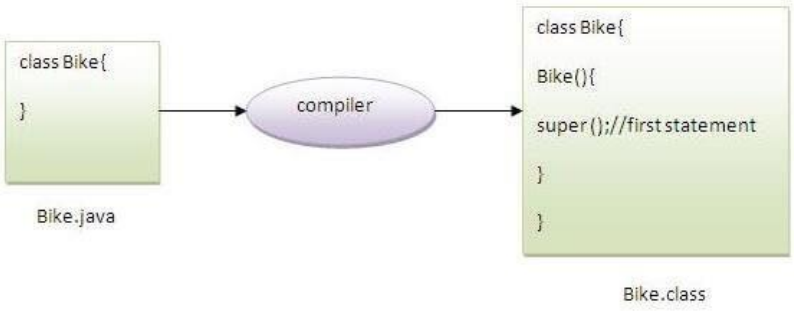
```
class Animal{
        Animal(){System.out.println("animal is created");}
}
class Dog extends Animal{
        Dog(){
                super();
                System.out.println("dog is created");
        }
}
class TestSuper3{
```

Output:

animal is created
dog is created

```
        public static void main(String args[]){
        Dog d=new Dog();
}}
```

Note: super() is added in each class constructor automatically by compiler if there is no super() or this().



As we know well that default constructor is provided by compiler automatically if there is no constructor. But, it also adds super() as the first statement.

Another example of super keyword where super() is provided by the compiler implicitly.

```
class Animal{
        Animal(){System.out.println("animal is created");}
}
class Dog extends Animal{
        Dog(){
                System.out.println("dog is created");
        }
}
class TestSuper4{
        public static void main(String args[]){
                Dog d=new Dog();
}}
```

Output:

animal is created
dog is created

5.5 Execution of Constructors in Multilevel Inheritance

In multilevel inheritance, all the upper class constructors are executed when an instance of bottom most child class is created.

OrderofExecution2.java

```
class College {
   /* Constructor */
   College() {
      System.out.println("College constructor executed");
   }
}

class Department extends College {
   /* Constructor */
   Department() {
      System.out.println("Department constructor executed");
   }
}

class Student extends Department {
   /* Constructor */
   Student() {
      System.out.println("Student constructor executed");
```

```
    }
}

public class OrderofExecution2 {
    /* Driver Code */
    public static void main(String ar[]) {
        /* Create an instance of Student class */
        System.out.println("Order of constructor execution in Multilevel inheritance...");
        new Student();
    }
}
```

Output:
Order of constructor execution in Multilevel inheritance...
College constructor executed
Department constructor executed
Student constructor executed

In the above code, an instance of ***Student*** class is created and it invokes the constructors of ***College, Department*** and ***Student*** accordingly.

5.6 Abstract Classes and Methods

A class which is declared with the abstract keyword is known as an abstract class in Java. 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 in Java

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 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 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 Method in Java
A method which is declared as abstract and does not have implementation is known as an abstract method.

Example of abstract method

```
abstract void printStatus();//no method body and abstract
```

Example of Abstract class that has an abstract method

File: TestBank.java

```
abstract class Bank{
    abstract int getRateOfInterest();
}
class SBI extends Bank{
    int getRateOfInterest(){return 7;}
}
class PNB extends Bank{
    int getRateOfInterest(){return 8;}
}

class TestBank{
    public static void main(String args[]){
        Bank b;
        b=new SBI();
        System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
        b=new PNB();
        System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
    }
}
```

Output:
Rate of Interest is: 7 %
Rate of Interest is: 8 %

Understanding the real scenario of Abstract class
In this example, Shape is the abstract class, and its implementation is provided by the Rectangle and Circle classes.
Mostly, we don't know about the implementation class (which is hidden to the end user), and an object of the implementation class is provided by the **factory method**.

A **factory method** is a method that returns the instance of the class. A Factory Pattern or Factory Method Pattern says that just define an interface or abstract class for creating an object but let the subclasses decide which class to instantiate. In other words, subclasses are responsible to create the instance of the class.
In this example, if you create the instance of Rectangle class, draw() method of Rectangle class will be invoked.
File: TestAbstraction1.java

```
abstract class Shape {
  abstract void draw();
}

// In a real scenario, implementation is provided by others i.e. unknown by end user
class Rectangle extends Shape {
  void draw() {
    System.out.println("drawing rectangle");
  }
}

class Circle1 extends Shape {
  void draw() {
    System.out.println("drawing circle");
  }
}
```

```
// In a real scenario, the method is called by the programmer or user
class TestAbstraction1 {
    public static void main(String args[]) {
        Shape s = new Circle1(); // In a real scenario, the object is provided through a method, e.g., getShape() method
        s.draw();
    }
}
```

Output:
drawing circle

Abstract class having constructor, data member and methods
An abstract class can have a data member, abstract method, method body (non-abstract method), constructor, and even main() method.

File: TestAbstraction2.java

```
// Example of an abstract class that has abstract and non-abstract methods
abstract class Bike {
    Bike() {
        System.out.println("bike is created");
    }

    abstract void run();

    void changeGear() {
        System.out.println("gear changed");
    }
}

// Creating a Child class which inherits Abstract class
class Honda extends Bike {
    void run() {
        System.out.println("running safely..");
    }
}

// Creating a Test class which calls abstract and non-abstract methods
class TestAbstraction2 {
    public static void main(String args[]) {
        Bike obj = new Honda();
        obj.run();
        obj.changeGear();
    }
}
```

Output:
bike is created
running safely..
gear changed

Rule: If there is an abstract method in a class, that class must be abstract.

```
class Bike12{
        abstract void run();
}
```

Output:
compile time error