Inheritance

Inheritance: Types of Inheritance, Deriving classes using extends keyword, super keyword, method overriding, abstract class, final keyword.

Inheritance in Java Programming

The process by which one class acquires the properties (data members) and functionalities (methods) of another class is called **inheritance**. The aim of inheritance is to provide the reusability of code so that a class has to write only the unique features and rest of the common properties and functionalities can be extended from the another class.

For example, a child inherits the traits of his/her parents. With inheritance, we can reuse the fields and methods of the existing class. Hence, inheritance facilitates Reusability and is an important concept of OOPs.

Or

Inheritance is a process of defining a new class based on an existing class by extending its common data members and methods.

Inheritance allows us to reuse of code, it improves reusability in your java application.

Note: The biggest **advantage of Inheritance** is that the code that is already present in base class need not be rewritten in the child class.
This means that the data members(instance variables) and methods of the parent class can be used in the child class.

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.
  The class whose properties and functionalities are used(inherited) by another class is known as parent class, super class or Base 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.

Syntax: Inheritance in Java

To inherit a class we use **extends** keyword. Here class XYZ is child class and class ABC is parent class. The class XYZ is inheriting the properties and methods of ABC class.

```
class XYZ extends ABC
{
}
```

or

```
class subClass extends superClass
{
  //methods and fields   }
```

Inheritance Example

In this example, we have a base class Teacher and a sub class PhysicsTeacher. Since class PhysicsTeacher extends the designation and college properties and work() method from base class, we need not to declare these properties and method in sub class.

Here we have collegeName, designation and work() method which are common to all the teachers so we have declared them in the base class, this way the child classes like MathTeacher, MusicTeacher and PhysicsTeacher do not need to write this code and can be used directly from base class.

```
class Teacher {
  String designation = "Teacher";
  String collegeName = "Beginnersbook";
void does(){
        System.out.println("Teaching");
  }
}

public class PhysicsTeacher extends Teacher{
String mainSubject = "Physics";
  public static void main(String args[]){
        PhysicsTeacher obj = new PhysicsTeacher();
        System.out.println(obj.collegeName);
      System.out.println(obj.designation);
      System.out.println(obj.mainSubject);
      obj.does();
  }
}
```

Output:

```
Beginnersbook
Teacher
Physics
Teaching
```

Based on the above example we can say that `PhysicsTeacher` **IS-A** `Teacher`. This means that a child class has IS-A relationship with the parent class. This is inheritance is known as **IS-A relationship** between child and parent class

Note:

The derived class inherits all the members and methods that are declared as public or protected. If the members or methods of super class are declared as private then the derived class cannot use them directly. The private members can be accessed only in its own class. Such private members can only be accessed using public or protected getter and setter methods of super class as shown in the example below.

```
class Teacher {
  private String designation = "Teacher";
  private String collegeName = "svecw";

  public String getDesignation() {
        return designation;
  }

  protected String getCollegeName() {
        return collegeName;
  }

  void does(){
        System.out.println("Teaching");
  }
}

public class Main extends Teacher{
  String mainSubject = "Physics";
  public static void main(String args[]){
        Main obj = new Main();

        System.out.println(obj.getCollegeName());
        System.out.println(obj.getDesignation());
        System.out.println(obj.mainSubject);
        obj.does();
  }
}
```

The output is:

```
Beginnersbook
Teacher
Physics
Teaching
```

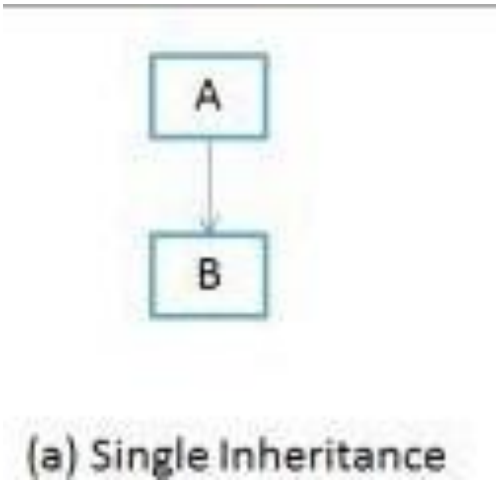
The important point to note in the above example is that the child class is able to access the private members of parent class through **protected methods** of parent class. When we make a instance variable(data member) or method **protected**, this means that they are accessible only in the class itself and in child class.

Types of inheritance in Java: Single,Multiple,Multilevel & Hybrid

Below are Various types of inheritance in Java. We will see each one of them one by one with the help of examples and flow diagrams.

1) Single Inheritance

Single inheritance is damn easy to understand. 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

Single Inheritance example program in Java

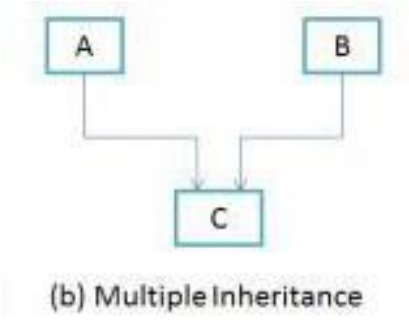
```
Class A {
  public void methodA()
  {
    System.out.println("Base class method");
  }
}

Class B extends A
```

```
{
  public void methodB()
  {
    System.out.println("Child class method");
  }
  public static void main(String args[])
  {
    B obj = new B();
    obj.methodA(); //calling super class method
    obj.methodB(); //calling local method
  }
}
```

2) Multiple Inheritance //java willnot support this inheritance

"**Multiple Inheritance**" refers to the concept of one class extending (Or inherits) more than one base class. The inheritance we learnt earlier had the concept of one base class or parent. The problem with "multiple inheritance" is that the derived class will have to manage the dependency on two base classes.



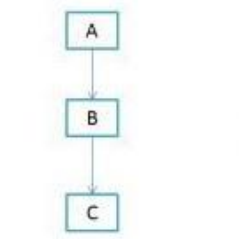
(b) Multiple Inheritance

Note 1: Multiple Inheritance is very rarely used in software projects. Using Multiple inheritance often leads to problems in the hierarchy. This results in unwanted complexity when further extending the class.

Note 2: Most of the new OO languages like **Small Talk, Java, C# do not support Multiple inheritance**. Multiple Inheritance is supported in C++. It does not supported by java.

3) Multilevel Inheritance

Multilevel inheritance refers to a mechanism in OO technology where one can inherit from 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.



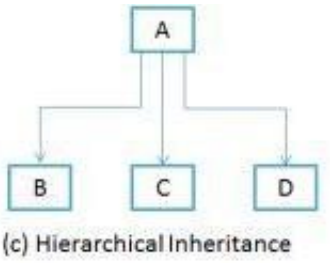
(d) Multilevel Inheritance

Multilevel Inheritance example program in Java

```
Class X
{
  public void methodX()
  {
    System.out.println("Class X method");
  }
}
Class Y extends X
{
public void methodY()
{
System.out.println("class Y method");
}
}
Class Z extends Y
{
  public void methodZ()
  {
    System.out.println("class Z method");
  }
  public static void main(String args[])
  {
    Z obj = new Z();
    obj.methodX(); //calling grand parent class method
    obj.methodY(); //calling parent class method
    obj.methodZ(); //calling local method
 }
}
```

4) 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. Read More at –



(c) Hierarchical Inheritance

Example of Hierarchical Inheritance

We are writing the program where class B, C and D extends class A.

```
class A {
   public void methodA()
   {
      System.out.println("method of Class A");
} }
class B extends A
{
   public void methodB()
   {
      System.out.println("method of Class B");
} }
class C extends A
{
  public void methodC()
  {
     System.out.println("method of Class
C");   } }
class D extends A
{
  public void methodD()
  {
     System.out.println("method of Class D");
} }
class JavaExample
{
  public static void main(String args[])
{
```

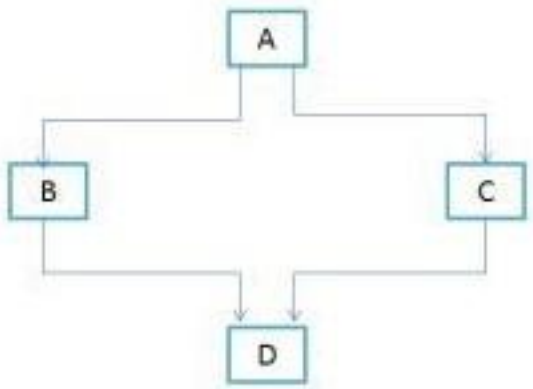
```
B obj1 = new B();
C obj2 = new C();
D obj3 = new D();
    //All classes can access the method of class A
obj1.methodA();
    obj2.methodA();
    obj3.methodA();
  }
}
```

Output:

```
method of Class A
method of Class A
method of Class A
```

5) Hybrid Inheritance

In simple terms you can say that Hybrid inheritance is a combination of **Single** and **Multiple inheritance.** A typical flow diagram would look like below. A hybrid inheritance can be achieved in the java in a same way as multiple inheritance can be!! Using interfaces. yes you heard it right. By using **interfaces** you can have multiple as well as **hybrid inheritance** in Java.



(e) Hybrid Inheritance

Hybrid inheritance: Combination of more than one types of inheritance in a single program. For example class A & B extends class C and another class D extends class

A then this is a hybrid inheritance example because it is a combination of single and hierarchical inheritance.

hybrid inheritance in java with example program

A hybrid inheritance is a combination of more than one types of inheritance. For example when class A and B extends class C & another class D extends class A then this is a hybrid inheritance, because it is a combination of single and hierarchical inheritance. Let me show you this diagrammatically:

↑

Hybrid Inheritance Example

Lets write this in a program to understand how this works:

Program: This example is just to demonstrate the hybrid inheritance in Java.

```

```

Although this example is meaningless, you would be able to see that how we have implemented two types of inheritance(single and hierarchical) together to form hybrid inheritance.
Class A and B extends class C → Hierarchical inheritance
Class D extends class A → Single inheritance

```
class C {
  public void disp()
  {
        System.out.println("C");
  }
}

class A extends C
{
  public void disp()
  {
        System.out.println("A");
  }
}

class B extends C
```

```
{
  public void disp()
  {
        System.out.println("B");
  }

}

class D extends A
{
  public void disp()
  {
        System.out.println("D");
  }
  public static void main(String args[]){

        D obj = new D();
        obj.disp();
  }
}
```

Output:

```
D
```

Constructors and Inheritance constructor of sub class is invoked when we create the object of subclass, it by default invokes the default constructor of super class. Hence, in inheritance the objects are constructed top-down.

The superclass constructor can be called explicitly using the super keyword, but it should be first statement in a constructor.
The super keyword refers to the superclass, immediately above of the calling class in the hierarchy. The use of multiple super keywords to access an ancestor class other than the direct parent is not permitted.

```
class ParentClass{
  //Parent class constructor
  ParentClass(){
        System.out.println("Constructor of Parent");
  }
}
class JavaExample extends ParentClass{
```

```
  JavaExample(){
        /* It by default invokes the constructor of parent class
         * You can use super() to call the constructor of parent.
         * It should be the first statement in the child class
         * constructor, you can also call the parameterized constructor
         * of parent class by using super like this: super(10), now
         * this will invoke the parameterized constructor of int arg
         */
        System.out.println("Constructor of Child");
  }
  public static void main(String args[]){
        //Creating the object of child class
        JavaExample obj=new JavaExample();
  }
}
```

Output:

```
Constructor of Parent
Constructor of Child
```

Super keyword in java

The super keyword refers to the objects of immediate parent class.

The use of super keyword

1) To access the data members of parent class when both parent and child class have **member with same name.**

2) To explicitly call the **no-arg** and **parameterized constructor** of parent class

3) To access the method of parent class when child class has **overridden** that method.

1) How to use super keyword to access the variables of parent class

When you have a variable in child class which is already present in the parent class then in order to access the variable of parent class, you need to use the super keyword.

lets take an example to understand this: In the following program, we have a data member num declared in the child class, the member with the same name is already present in the parent class. There is no way you can access the num variable of parent class without using super keyword. .

```
/Parent class or Superclass or base class class Superclass
{
  int num = 100;
}
//Child class or subclass or derived class class
Subclass extends Superclass
{
  /* The same variable num is declared in the Subclass
   */
   int num = 110;
   void printNumber(){
         System.out.println(num);
   }
   public static void main(String args[]){
         Subclass obj= new Subclass();
         obj.printNumber();
   }
}
```

Output:

```
* which is already present in the Superclass
110
```

Accessing the num variable of parent class:

By calling a variable like this, we can access the variable of parent class if both the classes (parent and child) have same variable.

```
super.variable_name
```

Let's take the same example that we have seen above, this time in print statement we are passing `super.num` instead of num.

```
class Superclass
{
  int num = 100;
}
class Subclass extends Superclass
{
  int num = 110;
void printNumber(){
        /* Note that instead of writing num we are
* writing super.num in the print statement
* this refers to the num variable of Superclass
         */
        System.out.println(super.num);
  }
  public static void main(String args[]){
        Subclass obj= new Subclass();
        obj.printNumber();
  }
}
```

Output:

100

As you can see by using super.num we accessed the num variable of parent class.

```
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.

3) How to use super keyword in case of method overriding

When a child class declares a same method which is already present in the parent class then this is called method overriding. For now you just need to remember this: When a child class overrides a method of parent class, then the call to the method from child class object always call the child class version of the method. However by using super keyword like this: super.method_name you can call the method of parent class (the method which is overridden). In case of method overriding, these terminologies are used: Overridden method: The method of parent class Overriding method: The method of child class Lets take an example to understand this concept:

```
class Parentclass
{
  //Overridden method
  void display(){
        System.out.println("Parent class method");
  }
} class Subclass extends
Parentclass
{
  //Overriding method
  void display(){
        System.out.println("Child class method");
  }
  void printMsg(){
        //This would call Overriding method
        display();
        //This would call Overridden method
        super.display();
  }
  public static void main(String args[]){
        Subclass obj= new Subclass();
        obj.printMsg();
  }
}
```

Output:

```
Child class method
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.

2) Use of super keyword to invoke constructor of parent class

When we create the object of sub class, the new keyword invokes the constructor of child class, which implicitly invokes the constructor of parent class. So the order to execution when we create the object of child class is: parent class constructor is executed first and then the child class constructor is executed. It happens because compiler itself adds super()(this invokes the no-arg constructor of parent class) as the first statement in the constructor of child class.

Let's see an example to understand what I have explained above:

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:

Example:

```
class Animal{
Animal(){System.out.println("animal is created");}
}
class Dog extends Animal{
Dog(){
super();
System.out.println("dog is created");
}
}
class TestSuper3{
public static void main(String args[]){
Dog d=new Dog();
}}
```

Output:

```
animal is created
dog is created
```

Example:

```
class Parentclass
```

```
{
   Parentclass(){
         System.out.println("Constructor of parent class");
```

```
  }
}
class Subclass extends Parentclass
{
  Subclass(){
        /* Compile implicitly adds super() here as the
         *  first statement of this constructor.
         */
        System.out.println("Constructor of child class");
  }
  Subclass(int num){
        /* Even though it is a parameterized constructor.
         * The compiler still adds the no-arg super() here
         */
        System.out.println("arg constructor of child class");
  }
  void display(){
        System.out.println("Hello!");
  }
  public static void main(String args[]){
        /* Creating object using default constructor. This
         * will invoke child class constructor, which will
         * invoke parent class constructor
         */
        Subclass obj= new Subclass();
        //Calling sub class method
        obj.display();
        /* Creating second object using arg constructor
         * it will invoke arg constructor of child class which will
         * invoke no-arg constructor of parent class automatically
         */
        Subclass obj2= new Subclass(10);
        obj2.display();
  }
}
```

Output:

```
Constructor of parent class
Constructor of child class
Hello!

Constructor of parent class
arg constructor of child class
Hello!
```

Parameterized super() call to invoke parameterized constructor of parent class

We can call super() explicitly in the constructor of child class, but it would not make any sense because it would be redundant. It's like explicitly doing something which would be implicitly done otherwise.
However when we have a constructor in parent class that takes arguments then we can use parameterized super, like `super(100);` to invoke parameterized constructor of parent class from the constructor of child class.
Let's see an example to understand this:

Example:

```
class Parentclass
{
  //no-arg constructor
  Parentclass(){
        System.out.println("no-arg constructor of parent class");
  }
  //arg or parameterized constructor
  Parentclass(String str){
        System.out.println("parameterized constructor of parent class");
  }
}
class Subclass extends Parentclass
{
  Subclass(){
     /* super() must be added to the first statement of constructor
        * otherwise you will get a compilation error. Another important
        * point to note is that when we explicitly use super in constructor
        * the compiler doesn't invoke the parent constructor automatically.
        */
        super("Hahaha");
        System.out.println("Constructor of child class");

  }
  void display(){
        System.out.println("Hello");
  }
  public static void main(String args[]){
        Subclass obj= new Subclass();
        obj.display();
  }
}
```

Output:

```
parameterized constructor of parent class
Constructor of child class
Hello
```

Example : Call Parameterized Constructor Using super()

```
class Animal {
  // default or no-arg constructor
  Animal() {
    System.out.println("I am an animal");
  }

  // parameterized constructor
  Animal(String type) {
    System.out.println("Type: "+type);
  }
}
class Dog extends Animal {

  // default constructor
  Dog() {

    // calling parameterized constructor of the superclass
    super("Animal");

    System.out.println("I am a dog");
  }
}
class Main {
  public static void main(String[] args) {
    Dog dog1 = new Dog();
  }
}
```

Output

```
Type: Animal
I am a dog
```

The compiler can automatically call the no-arg constructor. However, it cannot call parameterized constructors.

If a parameterized constructor has to be called, we need to explicitly define it in the subclass constructor.

There are few important points to note in this example:
1) super()(or parameterized super must be the first statement in constructor otherwise you will get the compilation error: "Constructor call must be the first statement in a constructor"

2) When we explicitly placed super in the constructor, the java compiler didn't call the default no-arg constructor of parent class.

Super keyword in multilevel inheritance:

```
* multi-level inheritance  using super keyword
super keyword is used when method names of every class are same
in here 3 classes have display method  we call every classes
method using super keyword in its child class
 */
  class First
{
   void display()
    {
   System.out.println("    Class one ");
   } }
 class Second  extends First
{
   void display()
   {
 super.display();//calls parent class member
    System.out.println("    Class Two ");
   } }
class Third extends Second
{
   void display()
   {
      super.display();
  System.out.println("    Class Three\n");
sign();
   } }
public class test
{
  public static void main(String[] args)
  {
  System.out.println("multi-level inheritance using\nSUPER keyword \n");
   Third  obj = new Third();
obj.display();
   }
}
```

```
multi-level inheritance using
SUPER keyword
```

Class one
Class Two
Class Three

If all classes having the same name methods(display) we have to use super keyword to call all methods.

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).

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.

```
//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

//Java Program to illustrate the use of Java Method Overriding **Method Overriding Example**

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.

```
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
```

Advantage of method overriding

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**.

This is helpful when a class has several child classes, so if a child class needs to use the parent class method, it can use it and the other classes that want to have different implementation can use overriding feature to make changes without touching the parent class code.

Method Overriding and Dynamic Method Dispatch

Method Overriding is an example of runtime polymorphism. When a parent class reference points to the child class object then the call to the overridden method is determined at runtime, because during method call which method(parent class or child class) is to be executed is determined by the type of object.

This process in which call to the overridden method is resolved at runtime is known as dynamic method dispatch. Lets see an example to understand this:

```
class ABC{
  //Overridden method
  public void disp()
  {
        System.out.println("disp() method of parent class");
  }
}
class Demo extends ABC{
  //Overriding method
  public void disp(){
        System.out.println("disp() method of Child class");
  }
  public void newMethod(){
        System.out.println("new method of child class");
  }
  public static void main( String args[]) {
        /* When Parent class reference refers to the parent class object
         * then in this case overridden method (the method of parent class)
         *  is called.
         */
        ABC obj = new ABC();
        obj.disp();

        /* When parent class reference refers to the child class object
         * then the overriding method (method of child class) is called.
         * This is called dynamic method dispatch and runtime polymorphism
         */
        ABC obj2 = new Demo();
        obj2.disp();
  }
}
```

Output:

```
disp() method of parent class
disp() method of Child class
```

In the above example the call to the disp() method using second object (obj2) is runtime polymorphism (or dynamic method dispatch).

Note: In dynamic method dispatch the object can call the overriding methods of child class and all the non-overridden methods of base class but it cannot call the methods which are newly declared in the child class. In the above example the object obj2 is calling the disp().

However if you try to call the newMethod() method (which has been newly declared in Demo class) using obj2 then you would give compilation error with the following message:

```
Exception in thread "main" java.lang.Error: Unresolved compilation  problem:
The method xyz() is undefined for the type ABC
```

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:

No.	Method Overloading	Method Overriding
1)	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.

2)	Method overloading is performed *within class*.	Method overriding occurs *in two classes* that have IS-A (inheritance) relationship.
3)	In case of method overloading, *parameter must be different*.	In case of method overriding, *parameter must be*
		same.
4)	Method overloading is the example of *compile time polymorphism*.	Method overriding is the example of *run time polymorphism*.
5)	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.

Java Method Overloading example

```
class OverloadingExample{
static int add(int a,int b){return a+b;}
static int add(int a,int b,int c){return a+b+c;}
}
```

Java Method Overriding example

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

Features of Method overriding

There are the following features of method overriding in Java. They are as follows:

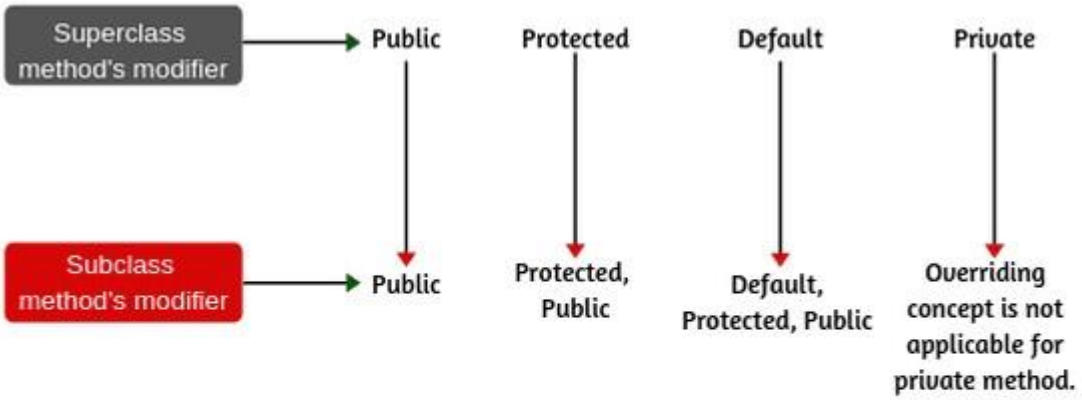
1. Method overriding technique supports the runtime polymorphism.
2. It allows a subclass to provide its own implementation of the method which is already provided by the superclass.
3. Only the instance method can be overridden in Java.
4. An instance variable can never be overridden in Java.
5. The overriding method can not be more restrictive than the overridden method of the superclass.
6. Overriding concept is not applicable for private, final, static, and main method in Java.

7. Method overriding can be done by changing the covariant return type only.
8. Overriding method cannot throw any checked exception.

Method Overriding rules in Java

When you are overriding super class method in a subclass, you need to follow certain rules. They are as follows.

1. Subclass method name must be the same as super class method name.
2. The parameters of subclass method must be the same as superclass method parameters. i.e In overriding, method name and argument types must be matched. In other words, the method signature must be the same or matched. 3. Must be Is-A relationship (Inheritance).
4. Subclass method's return type must be the same as superclass method return type. But this rule is applicable until Java 1.4 version only. From Java 1.5 version onwards, covariant return types are also allowed.
5. Subclass method's access modifier must be same or higher than superclass method access modifier.



The access modifier of the overriding method cannot be more restrictive than overridden method of superclass.

Private > Default > Protected > Public

More restrictive → Less restrictive

Fig: Applicable access modifiers to the overriding method

Abstract Class in Java with example

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.

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 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.
- . 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.

Why we need an abstract class?

Lets say we have a class Animal that has a method sound() and the subclasses of it like Dog, Lion, Horse, Cat etc. Since the animal sound differs from one animal to another, there is no point to implement this method in parent class. This is because every child class must override this method to give its own implementation details, like Lion class will say "Roar" in this method and Dog class will say "Woof".

So when we know that all the animal child classes will and should override this method, then there is no point to implement this method in parent class. Thus, making this method abstract would be the good choice as by making this method abstract we force all the sub classes to implement this method(otherwise you will get compilation error), also we need not to give any implementation to this method in parent class.

Since the Animal class has an abstract method, you must need to declare this class abstract.

Now each animal must have a sound, by making this method abstract we made it compulsory to the child class to give implementation details to this method. This way we ensures that every animal has a sound.

Abstract class Example

```
//abstract parent class abstract
class Animal{    //abstract
method
  public abstract void sound();
}
//Dog class extends Animal class
public class Dog extends Animal{

  public void sound(){
        System.out.println("Woof");
  }
  public static void main(String args[]){
        Animal obj = new Dog();
        obj.sound();
  }
}
```

Output:

```
Woof
```

Hence for such kind of scenarios we generally declare the class as abstract and later **concrete classes** extend these classes and override the methods accordingly and can have their own methods as well.

Abstract class declaration

An abstract class outlines the methods but not necessarily implements all the methods.

```
//Declaration using abstract keyword
abstract class A{
  //This is abstract method
  abstract void myMethod();

  //This is concrete method with body
  void anotherMethod(){
    //Does something
  }
}
```

Rules

Note 1: As we seen in the above example, there are cases when it is difficult or often unnecessary to implement all the methods in parent class. In these cases, we can declare the parent class as abstract, which makes it a special class which is not complete on its own.

A class derived from the abstract class must implement all those methods that are declared as abstract in the parent class.

Note 2: Abstract class cannot be instantiated which means you cannot create the object of it. To use this class, you need to create another class that extends this this class and provides the implementation of abstract methods, then you can use the object of that child class to call non-abstract methods of parent class as well as implemented methods(those that were abstract in parent but implemented in child class).

Note 3: If a child does not implement all the abstract methods of abstract parent class, then the child class must need to be declared abstract as well.

Do you know? Since abstract class allows concrete methods as well, it does not provide 100% abstraction. You can say that it provides partial abstraction. Abstraction is a process where you show only "relevant" data and "hide" unnecessary details of an object from the user.

Interfaces on the other hand are used for 100% abstraction.

Why can't we create the object of an abstract class?

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 What would happen?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.

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

As discussed above, we cannot instantiate an abstract class. This program throws a compilation error.

```
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
```

Note: The class that extends the abstract class, have to implement all the abstract methods of it, else you have to declare that class abstract as well.

Abstract class vs Concrete class

A class which is not abstract is referred as **Concrete class**. In the above example that we have seen in the beginning of this guide, `Animal` is a abstract class and `Cat`, `Dog` & `Lion` are concrete classes.

Key Points:

1. An abstract class has no use until unless it is extended by some other class.
2. If you declare an **abstract method** in a class then you must declare the class abstract as well. you can't have abstract method in a concrete class. It's vice versa is not always true: If a class is not having any abstract method then also it can be marked as abstract.
3. It can have non-abstract method (concrete) as well. **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
```

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. We will learn about the factory method later.

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

```
drawing circle
```

Another example of Abstract class in java

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 %

Abstract class having constructor, data member and methods

An abstract class can have a data member, abstract method, method body (nonabstract 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
```

Example 1: Inheritance of Abstract Class

```
abstract class Animal {
public void displayInfo() {
```

```
  System.out.println("I am an animal.");
 }
}

class Dog extends Animal {

}
class Main {
 public static void main(String[] args) {
  Dog d1 = new Dog();
  d1.displayInfo();
 }
}
```

Output:

```
I am an animal.
```

In the above example, we have created an abstract class `Animal`. We cannot create objects of `Animal`. To access the `displayInfo()` of `Animal`, we have inherited a subclass `Dog` of `Animal`.
We then used the `d1` object of `Dog` to access the method `displayInfo()`.

Overriding of Abstract Methods

In Java, it is mandatory to override abstract methods of the superclass in the subclass. It is because the subclass inherits abstract methods of the superclass.

Since our subclass includes abstract methods, we need to override them.

Note: If the subclass is also declared abstract, it's not mandatory to override abstract methods.

Example 2: Overriding Abstract Method

```
    Dog d1 = new Dog();
    d1.makeSound();
    d1.eat();
  }
```

```
abstract class Animal {
abstract void makeSound();

  public void eat() {
    System.out.println("I can eat.");
  }
}
class Dog extends Animal {
public void makeSound() {
    System.out.println("Bark bark");
  }
}
class Main {
  public static void main(String[] args) {
```

```
}
```

Output:

```
Bark bark.
I can eat.
```

In the above example, we have created an abstract class `Animal`. The class contains an abstract method `makeSound()` and a non-abstract method `eat()`.
We have inherited a subclass `Dog` from the superclass `Animal`. Here, the subclass `Dog` overrides the abstract method `displayInfo()`.
We then created an object `d1` of `Dog`. Using the object, we then called `d1.displayInfo()` and `d1.eat()` methods.

Accesses Constructor of Abstract Classes

Similar to non-abstract classes, we access the constructor of an abstract class from the subclass using the `super` keyword. For example,

```
abstract class Animal {
  Animal() {
    ....
  }
}

class Dog extends Animal {
  Dog() {
    super();
    ...
  }
}
```

Here, we have used the `super()` inside the constructor of `Dog` to access the constructor of the `Animal`.
Note that the `super` should always be the first statement of the subclass constructor. Visit Java super keyword to learn more.

Example 3: Java Abstraction

```
abstract class Animal {
  abstract void makeSound();
}
```

```
class Dog extends Animal {
  public void makeSound() {
    System.out.println("Bark bark.");
  }
}

class Cat extends Animal {
  public void makeSound() {
    System.out.println("Meows ");
  }
}

class Main {
  public static void main(String[] args) {
    Dog d1 = new Dog();
    d1.makeSound();

    Cat c1 = new Cat();
    c1.makeSound();
  }
}
```

Output:

```
Bark bark
Meows
```

In the above example, we have created a superclass `Animal`. The superclass `Animal` has an abstract method `makeSound()`.
The `makeSound()` method cannot be implemented inside `Animal`. It is because every animal makes different sounds. So, all the subclasses of `Animal` would have different implementation of `makeSound()`.
We cannot implement `makeSound()` in `Animal` in such a way that it can be correct for all subclasses of `Animal`. So, the implementation of `makeSound()` in `Animal` is kept hidden.
In the above example, `Dog` makes its own implementation of `makeSound()` and `Cat` makes its own implementation of `makeSound()`.

Key Points to Remember

- We use the `abstract` keyword to create abstract classes and methods.
- An abstract method doesn't have any implementation (method body).
- A class containing abstract methods should also be abstract.

- We cannot create objects of an abstract class.
- To implement features of an abstract class, we inherit subclasses from it and create objects of the subclass.
- A subclass must override all abstract methods of an abstract class. However, if the subclass is declared abstract, it's not mandatory to override abstract methods.
- We can access the static attributes and methods of an abstract class using the reference of the abstract class. For example,

```
Animal.staticMethod();
```

Final Keyword In Java

In this tutorial we will learn the usage of final keyword. final keyword can be used along with variables, methods and classes. We will cover following topics in detail.

1) final variable
2) final method
3) final class

1) final variable

The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only. We will have detailed learning of these. Let's first learn the basics of final keyword.

final variables are nothing but constants. We cannot change the value of a final variable once it is initialized. Lets have a look at the below code:

```
class Demo{
  final int MAX_VALUE=99;
  void myMethod(){
    MAX_VALUE=101;
  }
  public static void main(String args[]){
    Demo obj=new Demo();
    obj.myMethod();
  }
}
```

Output:

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
	The final field Demo.MAX_VALUE cannot be assigned

	at beginnersbook.com.Demo.myMethod(Details.java:6)
	at beginnersbook.com.Demo.main(Details.java:10)
```

We got a compilation error in the above program because we tried to change the value of a final variable "MAX_VALUE".

Note: It is considered as a good practice to have constant names in UPPER CASE(CAPS).

Example of final variable

There is a final variable speedlimit, we are going to change the value of this variable, but It can't be changed because final variable once assigned a value can never be changed.

```
class Bike9{
 final int speedlimit=90;//final variable
 void run(){
  speedlimit=400;
 }
 public static void main(String args[]){
 Bike9 obj=new  Bike9();
 obj.run();
 }
}//end of class
```

Output: Compile Time Error

Blank final variable

A final variable that is not initialized at the time of declaration is known as **blank final variable**. We **must initialize the blank final variable in constructor** of the class otherwise it will throw a compilation error (Error: variable MAX_VALUE might not have been initialized).

This is how a blank final variable is used in a class:

```
class Demo{      //Blank final variable
  final int MAX_VALUE;

  Demo(){
    //It must be initialized in constructor
     MAX_VALUE=100;
  }
  void myMethod(){
    System.out.println(MAX_VALUE);
```

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

  }
}
```

Output:

```
100
```

Whats the use of blank final variable?

obj.myMethod();

Lets say we have a Student class which is having a field called Roll No. Since Roll No should not be changed once the student is registered, we can declare it as a final variable in a class but we cannot initialize roll no in advance for all the students(otherwise all students would be having same roll no). In such case we can declare roll no variable as blank final and we initialize this value during object creation like this:

```
class StudentData{      //Blank final variable
  final int ROLL_NO;

  StudentData(int rnum){
    //It must be initialized in constructor
    ROLL_NO=rnum;
  }
  void myMethod(){
    System.out.println("Roll no is:"+ROLL_NO);
  }
  public static void main(String args[]){

    StudentData obj=new  StudentData(1234);
    obj.myMethod();
  }
}
```

Output:

```
Roll no is:1234
```

.Uninitialized static final variable

A static final variable that is not initialized during declaration can only be initialized in static block. Example:

```
class Example{
  //static blank final variable      static final int ROLL_NO;
```

```
   static{
     ROLL_NO=1230;
   }
   public static void main(String args[]){
     System.out.println(Example.ROLL_NO);
   }
}
```

Output:

```
1230
```

2) final method

A final method cannot be overridden. Which means even though a sub class can call the final method of parent class without any issues but it cannot override it.

The final keyword in Java can also be applied to methods. A Java method with the final keyword is called a final method and it can not be overridden in the subclass. You should make a method final in Java if you think it's complete and its behavior should remain constant in sub-classes. **Example:**

```
class XYZ{      final void demo(){
     System.out.println("XYZ Class Method");
   }
}

class ABC extends XYZ{      void demo(){
     System.out.println("ABC Class Method");
   }

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

The above program would throw a compilation error, however we can use the parent class final method in sub class without any issues. Lets have a look at this code: This program would run fine as we are not overriding the final method. That shows that final methods are inherited but they are not eligible for overriding.

```
class XYZ{      final void demo(){
     System.out.println("XYZ Class Method");
   }
}
```

```
class ABC extends XYZ{      public static
void main(String args[]){
     ABC       obj=       new       ABC();
obj.demo();
   }
}
```

Output:

```
XYZ Class Method
```

Example of final method

```
class Bike{
  final void run(){System.out.println("running");}
}

class Honda extends Bike{
   void run(){System.out.println("running safely with 100kmph");}

   public static void main(String args[]){
   Honda honda= new Honda();
   honda.run();
   }
}
```

Output: Compile Time Error

3) final class

We cannot extend a final class. Consider the below example:

```
final class XYZ{
}
```

```
class ABC extends XYZ{
  void demo(){
    System.out.println("My Method");
  }
  public static void main(String args[]){
    ABC obj= new ABC();
    obj.demo();
  }
}
```

Output:

Example of final class

```
final class Bike{}

class Honda1 extends Bike{     void run(){System.out.println("running safely with 100kmph");}

  public static void main(String args[]){
  Honda1 honda= new Honda1();
  honda.run();
  }
}
```

Output: Compile Time Error

```
The type ABC cannot subclass the final class XYZ
```

Points to Remember:

1) A constructor cannot be declared as final.
2) Local final variable must be initializing during declaration.
3) All variables declared in an interface are by default final.
4) We cannot change the value of a final variable.
5) A final method cannot be overridden.
6) A final class not be inherited.
7) If method parameters are declared final then the value of these parameters cannot be changed.
8) It is a good practice to name final variable in all CAPS.
9) final, finally and finalize are three different terms. finally is used in exception handling and finalize is a method that is called by JVM during garbage collection.

Q) Is final method inherited?

Ans) Yes, final method is inherited but you cannot override it. For Example:

```
class Bike{     final void
run(){System.out.println("running...");}   }
 class Honda2 extends Bike{
   public static void main(String args[]){
    new Honda2().run();
   }
 }
```

Output:running...