UNIT 2

JAVA PROGRAMMING (BCA 113)

Object in Java

An entity that has state and behavior is known as an object e.g., chair, bike, marker, pen, table, car, etc. It can be physical or logical (tangible and intangible). The example of an intangible object is the banking system.

An object has three characteristics:

1. **State:** represents the data (value) of an object.
2. **Behavior:** represents the behavior (functionality) of an object such as deposit, withdraw, etc.
3. **Identity:** An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, it is used internally by the JVM to identify each object uniquely

An object is an instance of a class. A class is a template or blueprint from which objects are created. So, an object is the instance(result) of a class.

Class in Java:

A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical.

A class is a user defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type. In general, class declarations can include these components, in order:

1. **Modifiers**: A class can be public or has default access
2. **class keyword:** class keyword is used to create a class.
3. **Class name:** The name should begin with an initial letter (capitalized by convention).
4. **Superclass(if any):** The name of the class's parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.
5. **Interfaces(if any):** A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.
6. **Body:** The class body surrounded by braces, { }.

A class in Java can contain:

- **Fields**
- **Methods**
- **Constructors**
- **Blocks**
- **Nested class and interface**

Syntax to declare a class:

```
class <class_name>{
   field;
   method;
}
```

Instance variable in Java

A variable which is created inside the class but outside the method is known as an instance variable. Instance variable doesn't get memory at compile time. It gets memory at runtime when an object or instance is created. That is why it is known as an instance variable.

NEW keyword in Java

The new keyword is used to allocate memory at runtime. All objects get memory in Heap memory area.

Methods in Java: A method is a collection of statements that perform some specific task and return the result to the caller. A method can perform some specific task without returning anything. Methods allow us to **reuse** the code without retyping the code. In Java, every method must be part of some class that is different from languages like C, C++, and Python.

Types of Method

There are two types of methods in Java:

1. Predefined Method
2. User-defined Method

Predefined Method

In Java, predefined methods are the method that is already defined in the Java class libraries is known as predefined methods. It is also known as the **standard library method** or **built-in method**. We can directly use these methods just by calling them in the program at any point. Some pre-defined methods are **length(), equals(), compareTo(), sqrt(),** etc. When we call any of the predefined methods in our program, a series of codes related to the corresponding method runs in the background that is already stored in the library.

Each and every predefined method is defined inside a class. Such as **print()** method is defined in the **java.io.PrintStream** class. It prints the statement that we write inside the method. For example, **print("Java")**, it prints Java on the console.

Ex.

```
1. public class Demo
2. {
3. public static void main(String[] args)
4. {
5. // using the max() method of Math class
6. System.out.print("The maximum number is: " + Math.max(9,7));
7. }
8. }
```

User Define Function in Java

Syntax:-

Access Apecifire modefire return_type Function name(list of parameter)
{
    Body of function;
}

Access Modifiers in Java

There are two types of modifiers in Java: **access modifiers** and **non-access modifiers**.

The access modifiers in Java specifies the accessibility or scope of a field, method, constructor, or class. We can change the access level of fields, constructors, methods, and class by applying the access modifier on it.

There are four types of Java access modifiers:

1. **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
2. **Default**: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
3. **Protected**: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
4. **Public**: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

Non-Access Modifiers

Java provides a number of non-access modifiers to achieve many other functionality.

- The *static* modifier for creating class methods and variables.
- The *final* modifier for finalizing the implementations of classes, methods, and variables.
- The *abstract* modifier for creating abstract classes and methods.
- The *synchronized* and *volatile* modifiers, which are used for threads.

Method Overloading

With **method overloading**, multiple methods can have the same name with different parameters:

Ex.

int myMethod(int x)

float myMethod(float x)

double myMethod(double x, double y)

Consider the following example, which has two methods that add numbers of different type:

```
public class Sum {
    public int sum(int x, int y)
    {
        return (x + y);
    }
    public int sum(int x, int y, int z)
    {
        return (x + y + z);
    }
    public double sum(double x, double y)
    {
        return (x + y);
    }
    public static void main(String args[])
    {
        Sum s = new Sum();
        System.out.println(s.sum(10, 20));
        System.out.println(s.sum(10, 20, 30));
        System.out.println(s.sum(10.5, 20.5));
```

```
    }
}
```

Constructors in Java

In Java, a constructor is a block of codes similar to the method. It is called when an instance of the class is created. At the time of calling constructor, memory for the object is allocated in the memory.

It is a special type of method which is used to initialize the object.

Every time an object is created using the new() keyword, at least one constructor is called.

It calls a default constructor if there is no constructor available in the class. In such case, Java compiler provides a default constructor by default.

There are two types of constructors in Java: no-arg constructor, and parameterized constructor.

Note: It is called constructor because it constructs the values at the time of object creation. It is not necessary to write a constructor for a class. It is because java compiler creates a default constructor if your class doesn't have any.

Rules for creating Java constructor

There are two rules defined for the constructor.

1. Constructor name must be the same as its class name
2. A Constructor must have no explicit return type
3. A Java constructor cannot be abstract, static, final, and synchronized

Types of Java constructors

There are two types of constructors in Java:

1. Default constructor (no-arg constructor)
2. Parameterized constructor

Java Default Constructor

A constructor is called "Default Constructor" when it doesn't have any parameter.

Syntax of default constructor:

```
1. //Java Program to create and call a default constructor
2. class Bike1{
3. //creating a default constructor
4. Bike1(){System.out.println("Bike is created");}
5. //main method
6. public static void main(String args[]){
7. //calling a default constructor
8. Bike1 b=new Bike1();
9. }
10.}
```

JAVA Parameterized Constructor

A constructor which has a specific number of parameters is called a parameterized constructor.

Why use the parameterized constructor?

The parameterized constructor is used to provide different values to distinct objects. However, you can provide the same values also.

Example of parameterized constructor

In this example, we have created the constructor of Student class that have two parameters. We can have any number of parameters in the constructor.

```
1. //Java Program to demonstrate the use of the parameterized constructor.
2. class Student4{
3.     int id;
4.     String name;
5.     //creating a parameterized constructor
6.     Student4(int i,String n){
7.     id = i;
8.     name = n;
```

```
9.      }
10.     //method to display the values
11.     void display(){System.out.println(id+" "+name);}
12.
13.     public static void main(String args[]){
14.     //creating objects and passing values
15.     Student4 s1 = new Student4(111,"Karan");
16.     Student4 s2 = new Student4(222,"Aryan");
17.     //calling method to display the values of object
18.     s1.display();
19.     s2.display();
20.    }
21.}
```

Constructor Overloading in Java

In Java, a constructor is just like a method but without return type. It can also be overloaded like Java methods.

Constructor overloading in Java is a technique of having more than one constructor with different parameter lists. They are arranged in a way that each constructor performs a different task. They are differentiated by the compiler by the number of parameters in the list and their types.

Example of Constructor Overloading

```
1.  //Java program to overload constructors
2.  class Student5{
3.      int id;
4.      String name;
5.      int age;
6.      //creating two arg constructor
7.      Student5(int i,String n){
8.      id = i;
9.      name = n;
10.     }
11.     //creating three arg constructor
12.     Student5(int i,String n,int a){
13.     id = i;
```

```
14.    name = n;
15.    age=a;
16.    }
17.    void display(){System.out.println(id+" "+name+" "+age);}
18.
19.    public static void main(String args[]){
20.    Student5 s1 = new Student5(111,"Karan");
21.    Student5 s2 = new Student5(222,"Aryan",25);
22.    s1.display();
23.    s2.display();
24.   }
25.}
```

Difference between constructor and method in Java

There are many differences between constructors and methods. They are given below.

Java Constructor	Java Method
A constructor is used to initialize the state of an object.	A method is used to expose the behavior of an object.
A constructor must not have a return type.	A method must have a return type.
The constructor is invoked implicitly.	The method is invoked explicitly.
The Java compiler provides a default constructor if you don't have any constructor in a class.	The method is not provided by the compiler in any case.
The constructor name must be same as the class name.	The method name may or may not be same as the class name.

Java Copy Constructor

There is no copy constructor in Java. However, we can copy the values from one object to another like copy constructor in C++.

There are many ways to copy the values of one object into another in Java. They are:

1. By constructor
2. By assigning the values of one object into another
3. By clone() method of Object class

In this example, we are going to copy the values of one object into another using Java constructor.

```
1. //Java program to initialize the values from one object to another object.
2. class Student6{
3.     int id;
4.     String name;
5.     //constructor to initialize integer and string
6.     Student6(int i,String n){
7.     id = i;
8.     name = n;
9.     }
10.    //constructor to initialize another object
11.    Student6(Student6 s){
12.    id = s.id;
13.    name =s.name;
14.    }
15.    void display(){System.out.println(id+" "+name);}
16.
17.    public static void main(String args[]){
18.    Student6 s1 = new Student6(111,"Karan");
19.    Student6 s2 = new Student6(s1);
20.    s1.display();
21.    s2.display();
22.   }
23.}
```

Copying values without constructor

We can copy the values of one object into another by assigning the objects values to another object. In this case, there is no need to create the constructor.

```
1. class Student7{
2.     int id;
3.     String name;
4.     Student7(int i,String n){
5.     id = i;
6.     name = n;
7.     }
8.     Student7(){}
9.     void display(){System.out.println(id+" "+name);}
10.
11.    public static void main(String args[]){
12.    Student7 s1 = new Student7(111,"Karan");
13.    Student7 s2 = new Student7();
14.    s2.id=s1.id;
15.    s2.name=s1.name;
16.    s1.display();
17.    s2.display();
18.   }
19.}
```

Static Methods

In Java, static members are those which belongs to the class and you can access these members without instantiating the class.

The static keyword can be used with methods, fields, classes (inner/nested), blocks.

Static Methods − You can create a static method by using the keyword *static*. Static methods can access only static fields, methods. To access static methods there is no need to instantiate the class, you can do it just using the class name as –

Ex.:-

```
public class MyClass {
   public static void sample(){
      System.out.println("Hello");
   }
   public static void main(String args[]){
      MyClass.sample();
   }
}
```

Static Fields − You can create a static field by using the keyword static. The static fields have the same value in all the instances of the class. These are created and initialized when the class is loaded for the first time. Just like static methods you can access static fields using the class name (without instantiation).

Example

```
public class MyClass {
   public static int data = 20;
   public static void main(String args[]){
      System.out.println(MyClass.data);
   }
   Java Arrays with Answers
   27
}
```

Static Blocks − These are a block of codes with a static keyword. In general, these are used to initialize the static members. JVM executes static blocks before the main method at the time of class loading.

Example

```
public class MyClass {
   static{
```

```
    System.out.println("Hello this is a static block");
  }
  public static void main(String args[]){
    System.out.println("This is main method");
  }
}
```

Recursion in Java

Recursion in java is a process in which a method calls itself continuously. A method in java that calls itself is called recursive method.

Ex.

```
1. public class RecursionExample1 {
2. static void p(){
3. System.out.println("hello");
4. p();
5. }
6.
7. public static void main(String[] args) {
8. p();
9. }
10.}
```

Factorial Number

```
1. public class RecursionExample3 {
2.    static int factorial(int n){
3.         if (n == 1)
4.           return 1;
5.         else
6.           return(n * factorial(n-1));
7.    }
8.
9. public static void main(String[] args) {
10.System.out.println("Factorial of 5 is: "+factorial(5));
```

Factorial of 5 is: 120

Fibonacci Series

```
1.  public class RecursionExample4 {
2.     static int n1=0,n2=1,n3=0;
3.      static void printFibo(int count){
4.         if(count>0){
5.             n3 = n1 + n2;
6.             n1 = n2;
7.             n2 = n3;
8.             System.out.print(" "+n3);
9.             printFibo(count-1);
10.        }
11.    }

12.public static void main(String[] args) {
13.    int count=15;
14.     System.out.print(n1+" "+n2);//printing 0 and 1
15.     printFibo(count-2);//n-2 because 2 numbers are already printed
16.}
17.}
```

Inheritance in Java

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors 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.

Why use inheritance in java

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

Terms used in Inheritance

- **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
- **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

The syntax of Java Inheritance

```
1. class Subclass-name extends Superclass-name
2. {
3.    //methods and fields
4. }
```

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:

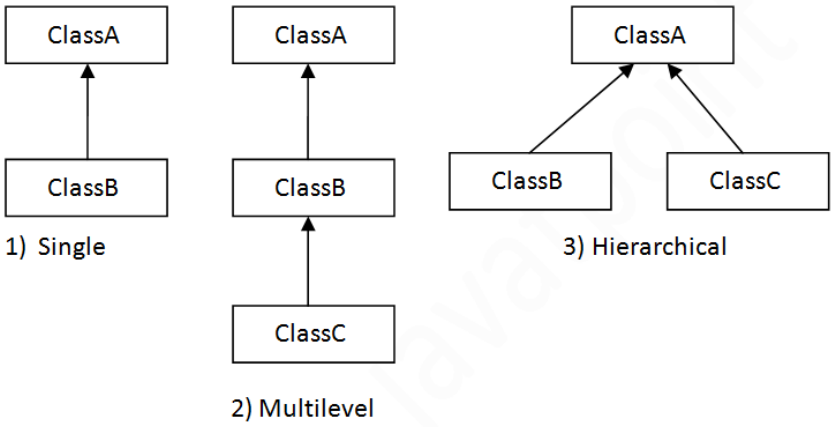
```
class Employee{
 float salary=40000;
}
class Programmer extends Employee{
 int bonus=10000;
 public static void main(String args[]){
```

```
  Programmer p=new Programmer();
  System.out.println("Programmer salary is:"+p.salary);
  System.out.println("Bonus of Programmer is:"+p.bonus);
}
}
```

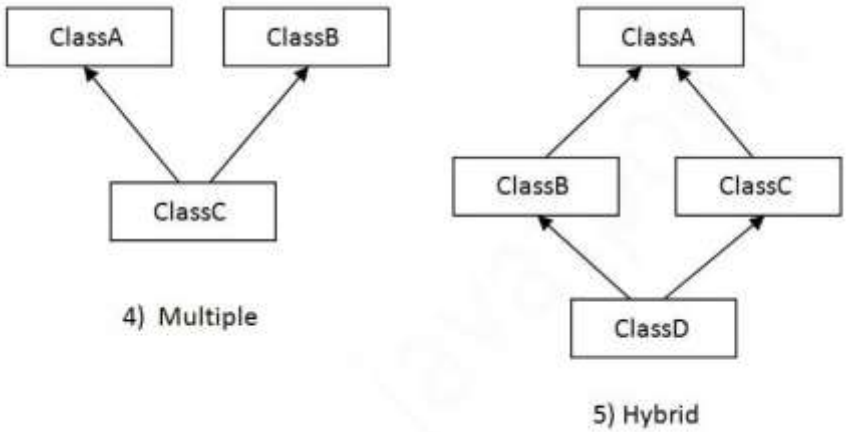
Types of inheritance in java

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. We will learn about interfaces later.



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.

```
1. class Animal{
2. void eat(){System.out.println("eating...");}
3. }
4. class Dog extends Animal{
5. void bark(){System.out.println("barking...");}
6. }
7. class TestInheritance{
8. public static void main(String args[]){
9. Dog d=new Dog();
10.d.bark();
11.d.eat();
12.}}
```

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.

```
1. class Animal{
2. void eat(){System.out.println("eating...");}
3. }
4. class Dog extends Animal{
5. void bark(){System.out.println("barking...");}
6. }
7. class BabyDog extends Dog{
8. void weep(){System.out.println("weeping...");}
9. }
10.class TestInheritance2{
11.public static void main(String args[]){
12.BabyDog d=new BabyDog();
```

```
13.d.weep();
14.d.bark();
15.d.eat();
16.}}
```

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.

```
1. class Animal{
2. void eat(){System.out.println("eating...");}
3. }
4. class Dog extends Animal{
5. void bark(){System.out.println("barking...");}
6. }
7. class Cat extends Animal{
8. void meow(){System.out.println("meowing...");}
9. }
10.class TestInheritance3{
11.public static void main(String args[]){
12.Cat c=new Cat();
13.c.meow();
14.c.eat();
15.//c.bark();//C.T.Error
16.}}
```

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.

```
1. class A{
2. void msg(){System.out.println("Hello");}
3. }
4. class B{
5. void msg(){System.out.println("Welcome");}
6. }
7. class C extends A,B{//suppose if it were
8.
9.  public static void main(String args[]){
10.  C obj=new C();
11.  obj.msg();//Now which msg() method would be invoked?
12.}
13.}
```

Output: compile time error

Method Overriding in Java

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.

```
//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
   }
}
```

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.

Final Keyword In Java

The **final keyword** in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

1. variable
2. method
3. class

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.

1) final variable

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).

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();
    obj.myMethod();
  }
}
```

Output:

```
100
```

Whats the use of blank final variable?

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
```

More about blank final variable at StackOverflow and Wiki.

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.

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
```

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:

```
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.

Java Object finalize() Method

Finalize() is the method of Object class. This method is called just before an object is garbage collected. finalize() method overrides to dispose system resources, perform clean-up activities and minimize memory leaks.

Syntax

1. **protected void** finalize() **throws** Throwable

Throw

Throwable - the Exception is raised by this method

Example 1

```
public class JavafinalizeExample1 {
    public static void main(String[] args)
    {
       JavafinalizeExample1 obj = new JavafinalizeExample1();
       System.out.println(obj.hashCode());
       obj = null;
       // calling garbage collector
       System.gc();
       System.out.println("end of garbage collection");

    }
    @Override
    protected void finalize()
    {
       System.out.println("finalize method called");
    }
}
```

Output:

```
2018699554
end of garbage collection
finalize method called
```

Abstract Classes and Methods

Data **abstraction** is the process of hiding certain details and showing only essential information to the user.
Abstraction can be achieved with either **abstract classes** or **interfaces**

The abstract keyword is a non-access modifier, used for classes and methods:

- **Abstract class:** is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).

- **Abstract method:** can only be used in an abstract class, and it does not have a body. The body is provided by the subclass (inherited from).

An abstract class can have both abstract and regular methods:

```
abstract class Animal {
  public abstract void animalSound();
  public void sleep() {
    System.out.println("Zzz");
  }
}
```

From the example above, it is not possible to create an object of the Animal class:

```
// Abstract class
abstract class Animal {
  // Abstract method (does not have a body)
  public abstract void animalSound();
  // Regular method
  public void sleep() {
    System.out.println("Zzz");
  }
}

// Subclass (inherit from Animal)
```

```
class Pig extends Animal {
  public void animalSound() {
    // The body of animalSound() is provided here
    System.out.println("The pig says: wee wee");
  }
}

class Main {
  public static void main(String[] args) {
    Pig myPig = new Pig(); // Create a Pig object
    myPig.animalSound();
    myPig.sleep();
  }
}
```

Visibility control (visibility access) in Java (or) Access modifiers:

It is possible to inherit all the members of a class by a subclass using the keyword **extends.** The variables and methods of a class are visible everywhere in the program. However, it may be necessary in some situations we may want them to be not accessible outside. We can achieve this in Java by applying *visibility modifiers* to instance variables and methods. The visibility modifiers are also known as *access modifiers.* Access modifiers determine the accessibility of the members of a class.

Java provides three types of visibility modifiers: **public,**
private and **protected.** They provide different levels of protection as described below.

Public Access: Any variable or method is visible to the entire class in which it is defined. But, to make a member accessible outside with objects, we simply declare the variable or method as public. A variable or method declared as **public** has the widest possible visibility and accessible everywhere.

Friendly Access (Default): When no access modifier is specified, the member defaults to a limited version of public accessibility known as "friendly" level of access. The difference between the "public" access and the "friendly" access is that the **public** modifier makes fields visible in all classes, regardless of their packages while the friendly access makes fields visible only in the same package, but not in other packages.

Protected Access: The visibility level of a "protected" field lies in between the public access and friendly access. That is, the **protected** modifier makes the fields visible not only to all classes and subclasses in the same package but also to subclasses in other packages

Private Access: private fields have the highest degree of protection. They are accessible only with their own class. They cannot be inherited by subclasses and therefore not accessible in subclasses. In the case of overriding public methods cannot be redefined as private type.

Private protected Access: A field can be declared with two keywords **private** and **protected** together. This gives a visibility level in between the "protected" access and "private" access. This modifier makes the fields visible in all subclasses regardless of what package they are in. Remember, these fields are not accessible by other classes in the same package.

The following table summarises the visibility provided by various access modifiers.

Access modifier →	**public**	**protected**	**friendly**	**private protected**	**private**
Own class	✓	✓	✓	✓	✓
Sub class in same package	✓	✓	✓	✓	×
Other classes In same package	✓	✓	✓	×	×
Sub class in other package	✓	✓	×	✓	×
Other classes In other package	✓	×	×	×	×