Defining a Class

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:

Modifiers : A class can be public or has default access.

Class name: The name should begin with an initial letter.

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.

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.

Body: The class body surrounded by braces, { }.

Syntax for declaring a class is

```
<modifier> class <Class name> [ extends superclassname ]
{
        //field declaration
        // constructors
        //method declarations
}
```

Fields declaration

Data is encapsulated in a class by placing data fields inside the body of the class definition. These variables are called instance variables because they are created whenever an object of the class is instantiated. Instance variables are declared in the same way as local variables are declared.

Syntax is **type variable_name;**

Example : **int num;**

String fname;

Method declaration

A Java method is a collection of statements that are grouped together to perform an operation 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 which is different from languages like C and C++.

In general, method declaration has following components:

Modifier-: Defines access type of the method i.e. from where it can be accessed in your application. In Java, there four type of the access specifiers - public, protected, private and default.

The **return type** : The data type of the value returned by the method or void if does not return a value.

Method Name: Should be a valid identifier

Parameter list: Comma separated list of the input parameters are defined, preceded with their data type, within the enclosed parenthesis. If there are no parameters, you must use empty parentheses ().

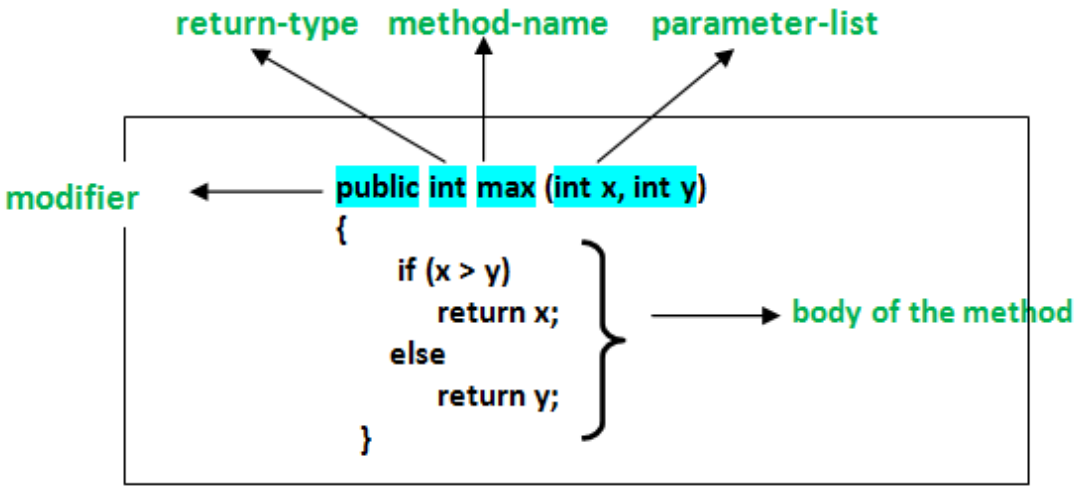
Method body: it is enclosed between braces. The code you need to be executed to perform your intended operations.

Method signature: It consists of the method name and a parameter list (number of parameters, type of the parameters and order of the parameters). The return type and exceptions are not considered as part of it.

The general form of method declaration is

```
type  method_name (parameter list)
{
        Method-body;
}
```

Example :



Method Signature of above function is **int max(int x, int y)**

Example 1: A Simple Class

```
class Rectangle
{
        int length,breadth;
        void setData(int x,int y)
        {
                length=x;
                breadth=y;
        }
        void display()
        {
                System.out.println("Length :"+length);
                System.out.println("Breadth:"+breadth);
        }
}
```

Creating objects

Instance variables and methods in a class are not accessible directly from outside the class. Objects are created for this purpose. Creating an object is referred to as ***instantiating*** an object.

In Java, the **new** keyword is used to create new objects. There are three steps when creating an object from a class. They are

- **Declaration** - A variable declaration with a valid name and an object type.
- **Instantiation** − The **new** keyword is used to create the object.
- **Initialization** − The **new** keyword is followed by a call to a constructor. This call initializes the new object.

Syntax :

Declaration : **classname objectname;**
Initialization : **objectname=new classname();**

Both these statements can be combined as follows:

classname objectname=new classname();

***Example*:** Rectangle rect=new Rectangle();

Accessing Class Members

Objects of a class are created for accessing class members from outside the class. To do this ***dot*** operator is used along with objects. Syntax for accessing class members are

objectname.variablename=value;
objectname.methodname(parameter_list);

Example 2: Class, object and accessing class members

```
class Rectangle
{
        int length,breadth;
        void setData(int x,int y)
        {
                length=x;
```

```
                breadth=y;
        }
        int area()
        {
                int area=length* breadth;
                return area;
        }
        void display()
        {
                System.out.println("Length :"+length);
                System.out.println("Breadth:"+breadth);
        }
}
class RectDemo
{
        public static void main(String args[])
        {
                int A1,A2;
                Rectangle r1=new Rectangle();
                Rectangle r2=new Rectangle();
                r1.length=30; r1.breadth=50;
                r1.display();
                A1=r1.area();
                System.out.println("Area of first rectangle:"+A1);
                r2. setData(10,20);
                r2.display();
                A2=r2.area();
                System.out.println("Area of second rectangle:"+A2);
        }
}
```

Output

```
Length :30
Breadth:50
Area of first rectangle:1500
Length :10
Breadth:20
Area of second rectangle:200
```

Method overloading

Method Overloading is a feature that allows a class to have more than one method having the same name, if their argument lists are different. Method overloading increases the readability of the program. There are three ways to overload a method. In order to overload a method, the argument lists of the methods must differ in either of these:

1. Number of parameters.
2. Data type of parameters.
3. Sequence of Data type of parameters.

If two methods have same name, same parameters and have different return type, then this ***is not a valid method overloading*.**

1. Method Overloading: changing number of parameters

This example shows how method overloading is done by having different number of parameters.

Example 3: Method overloading – changing number of parameters

```
class DisplayOverloading
{
   public void disp(char c)
   {
       System.out.println(c);
   }
   public void disp(char c, int num)
   {
       System.out.println(c + " "+num);
   }
}
class Sample
{
  public static void main(String args[])
  {
     DisplayOverloading obj = new DisplayOverloading();
     obj.disp('a');
     obj.disp('a',10);
  }
```

```
}
```
Output
```
a
a 10
```

2. Method Overloading: Difference in data type of parameters

In this example, method disp() is overloaded based on the data type of parameters – We have two methods with the name disp(), one with parameter of char type and another method with the parameter of int type.

Example 4: Method overloading – Difference in data type of parameters

```
class DisplayOverloading2
{
   public void disp(char c)
   {
       System.out.println(c);
   }
   public void disp(int c)
   {
      System.out.println(c );
   }
}
class Sample2
{
   public static void main(String args[])
   {
       DisplayOverloading2 obj = new DisplayOverloading2();
       obj.disp('a');
       obj.disp(5);
   }
}
```
Output
```
a
5
```

3. Method Overloading: Sequence of data type of arguments

Here method disp() is overloaded based on sequence of data type of parameters. Both the methods have different sequence of data type in argument list. First method is having argument list as (char, int) and second is having (int, char). Since the sequence is different, the method can be overloaded without any issues.

Example 5: Method overloading – Sequence of data type of arguments

```
class DisplayOverloading3
{
  public void disp(char c, int num)
  {
     System.out.println("I'm the first definition of method disp");
  }
  public void disp(int num, char c)
  {
     System.out.println("I'm the second definition of method disp"
);
  }
}
class Sample3
{
  public static void main(String args[])
  {
     DisplayOverloading3 obj = new DisplayOverloading3();
     obj.disp('x', 51 );
     obj.disp(52, 'y');
  }
}
```

Output

```
I'm the first definition of method disp
I'm the second definition of method disp
```

Constructors

A constructor is a special method that is **used to initialize an object**. Every class has a constructor either implicitly or explicitly. If we don't declare a constructor in the class then JVM builds a constructor for that class. This is

known as default constructor. **A constructor has same name as the class name in which it is declared. Constructor must have no explicit return type.** Constructor in Java cannot be abstract, static, final or synchronized. These modifiers are not allowed for constructor. Syntax to declare constructor

```
className (parameter-list)
{
  code-statements

}
```

className is the name of class, as constructor name is same as class name. parameter-list is optional, because constructors can be parameterize and non-parameterize as well.

Example

```
class Car
{
        String name ;
        String model;
        Car( )    //Constructor
        {
                name ="";
                model="";
        }
}
```

Types of Constructor

Java Supports two types of constructors:

- Default Constructor
- Parameterized constructor

Each time a new object is created at least one constructor will be invoked. **Default Constructor:** In Java, a constructor is said to be default constructor if it does not have any parameter. Default constructor can be either user defined or

provided by JVM. If a class does not contain any constructor then during runtime JVM generates a default constructor which is known as system define default constructor. If a class contain a constructor with no parameter then it is known as default constructor defined by user. In this case JVM does not create default constructor. The purpose of creating constructor is to initialize states of an object.

Example 6: Default Constructors

```
class MyClass
{
        int num;
        MyClass()
        {
                num = 100;
        }
}
class ConsDemo
{
        public static void main(String args[])
        {
                MyClass t1 = new MyClass();
                MyClass t2 = new MyClass();
                System.out.println(t1.num + " " + t2.num);
        }
}
```

Output

100 100

Parameterized Constructors: Most often, a constructor that accepts one or more parameters is needed in a program. Parameters are added to a constructor in the same way that they are added to a method; just declare them inside the parentheses after the constructor's name.

Example 7: Parameterized Constructors

```
class MyClass
```

```
{
        int x;
        // Following is the constructor
        MyClass(int i )
        {
                x = i;
        }
}
class ConsDemo
{
        public static void main(String args[])
        {
                MyClass t1 = new MyClass( 10 );
                MyClass t2 = new MyClass( 20 );
                System.out.println(t1.x + " " + t2.x);
        }
}
```

Output

10 20

Constructor Overloading

Like methods, a constructor can also be overloaded. **Overloaded constructors are differentiated on the basis of their type of parameters or number of parameters**. Constructor overloading is not much different than method overloading. In case of method overloading, multiple methods with same name but different signature are defined, whereas in Constructor overloading multiple constructor with different signature are defined but only difference is that constructor doesn't have return type.

Example 8: Constructor Overloading

```
class Cricketer
{
        String name,team;
        int age;
        Cricketer ()   //default constructor.
```

```
        {
                name ="";
                team ="";
                age = 0;
        }
        Cricketer(String n, String t, int a)   //constructor overloaded
        {
                name = n;
                team = t;
                age = a;
        }
        void show()
        {
                 System.out.println ("this is " + name + " of "+team);
        }
}
class test:
{
        public static void main (String[] args)
        {
                Cricketer c1 = new Cricketer();
                Cricketer c2 = new Cricketer("sachin", "India", 32);
                c1.name = "Virat";
                c1.team= "India";
                c1.age = 32;
                c1.show();
                c2.show();
        }
}
```

Output

```
this is virat of india
this is sachin of india
```

Static members

Static is a keyword in Java which is used to declare static stuffs. It can be used to create **variable, method block or a class**. Static variable or method can be

accessed without instance of a class because it belongs to class. Static members are common for all the instances of the class but non-static members are different for each instance of the class.

Static Variables: Static variables defined as a class member can be accessed without object of that class. Static variable is initialized once and shared among different objects of the class. All the object of the class having static variable will have the same instance of static variable. Static variable is used to represent common property of a class. It saves memory.

Example: Suppose there are 100 employee in a company. All employee have its unique name and employee id but company name will be same all 100 employee. Here company name is the common property. So if you create a class to store employee detail, then declare company_name field as static

Example 9: Static variable

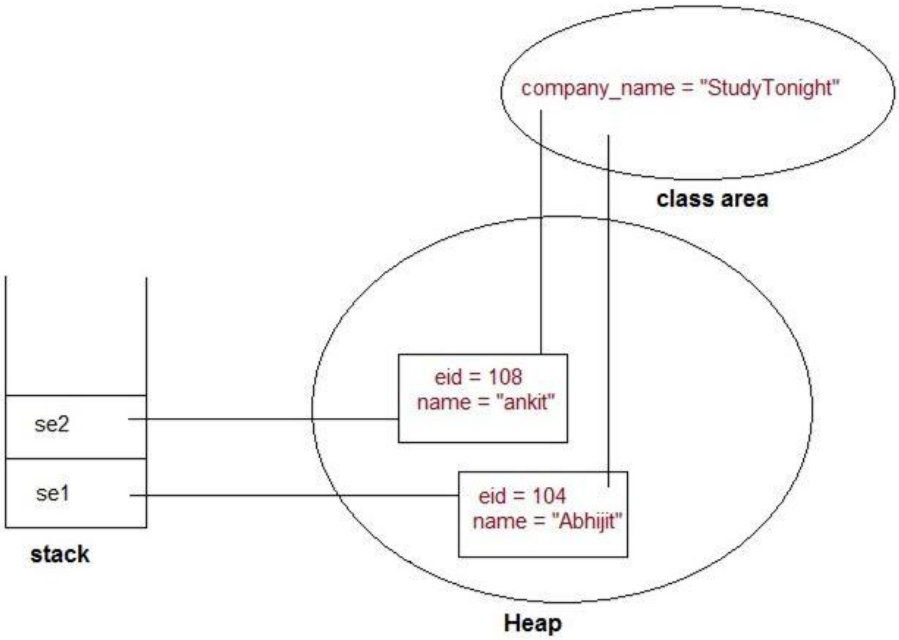
```
class Employee
{
	int eid;
	String name;
	static String company = "Studytonight";
	public void show()
	{
		System.out.println(eid + "-" + name + "-" + company);
	}
	public static void main( String[] args )
	{
		Employee se1 = new Employee();
		se1.eid = 104;
		se1.name = "Abhijit";
		se1.show();

		Employee se2 = new Employee();
		se2.eid = 108;
		se2.name = "ankit";
		se2.show();
```

```
    }
}
```

Output

```
104-Abhijit-Studytonight
108-ankit-Studytonight
```

The following image shows different memory areas used by the program and how static variable is shared among the objects of the class.



When a class variable is defined using the static keyword, then the variable is defined only once and is used by all the instances of the class. Hence if any of the class instance modifies it then it is changed for all the other instances of the class.

Example 10: Static variable vs instance variables

```
class Test
{
    static int x = 100;
    int y = 100;
    void increment()
    {
        x++; y++;
    }
    public static void main( String[] args )
    {
```

```
                Test t1 = new Test();
                Test t2 = new Test();
                t1.increment();
                t2.increment();
                System.out.println(t2.y);
                System.out.println(Test.x);  //accessed without any instance of class.
        }
}
```

Output

```
101
102
```

Static Method: A method can also be declared as static. Static methods do not need instance of its class for being accessed. main() method is the most common example of static method. main() method is declared as static because it is called before any object of the class is created.

Example 11: Static method

```
class Test
{
        static void square(int x)
        {
                System.out.println(x*x);
        }
        public static void main (String[] arg)
        {
                square(8)   //static method square () is called without any instance of class.
        }
}
```

Output

```
64
```

Static block: Static block is used to initialize static data members. Static block executes even before main() method. It executes when the class is loaded in the memory. A class can have multiple Static blocks, which will execute in the same sequence in which they are programmed.

Example 12: Static block

```
class ST_Employee
{
```

```
        int eid;
        String name;
        static String company_name;

        static
        {
                company_name ="StudyTonight"; // invoked before main() method
        }
        void show()
        {
                System.out.println(eid+" "+name+" "+company_name);
        }
        public static void main( String[] args )
        {
                ST_Employee se1 = new ST_Employee();
                se1.eid = 104;
                se1.name = "Abhijit";
                se1.show();
        }

}
```

Output

104 Abhijit StudyTonight

Why main() method is static in java?

Because static methods can be called without any instance of a class and main() is called before any instance of a class is created.

Inheritance

Inheritance is one of the key features of Object Oriented Programming. Inheritance provided mechanism that allowed a class to inherit property of another class. When a Class extends another class it inherits all non-private members including fields and methods. Inheritance in Java can be best understood in terms of Parent and Child relationship, also known as Super class

(Parent) and Sub class(child) in Java language. Inheritance defines is-a relationship between a Super class and its Sub class. ***extends*** keyword is used to describe inheritance in Java.

Inheritance promotes the code reusability. i.e the same methods and variables which are defined in a parent/super/base class can be used in the child/sub/derived class. It promotes polymorphism by allowing method overriding. Main disadvantage of using inheritance is that the two classes (parent and child class) gets tightly coupled. This means that if we change code of parent class, it will affect to all the child classes which is inheriting/deriving the parent class, and hence, it cannot be independent of each other.

Syntax :

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

The extends keyword indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

Example 13: Inheritance example

```
class Vehicle
{
        // variable defined
        String vehicleType;
}
public class Car extends Vehicle
{
        String modelType;
        public void showDetail()
        {
        vehicleType = "Car";    //accessing Vehicle class member variable
        modelType = "Sports";
        System.out.println(modelType + " " + vehicleType);
        }
```

```
        public static void main(String[] args)
        {
        Car car = new Car();
        car.showDetail();
        }
}
```

Output

Sports Car

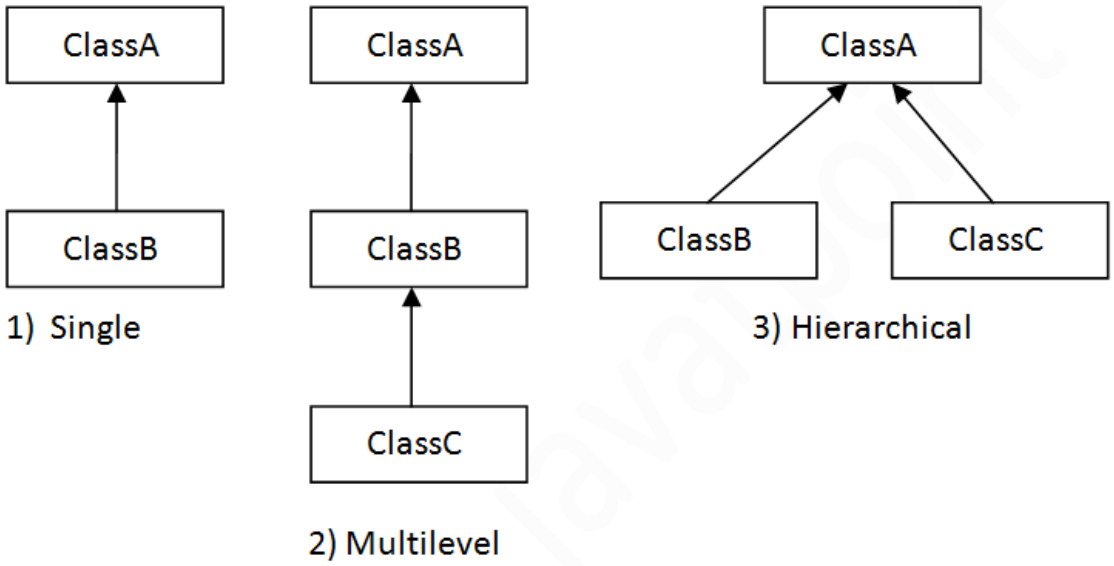
Types of Inheritance

Java mainly supports only three types of inheritance that are listed below.

- Single Inheritance
- Multilevel Inheritance
- Heirarchical Inheritance

NOTE: Multiple inheritance is not directly supported in java

Following figure gives a view of type of inheritance.



Single Inheritance : When a class extends to another class then it forms single inheritance. In the below example, there are two classes in which class A extends to class B that forms single inheritance.

Example 14: Single Inheritance

```
class A
{
        int a = 10;
        void show()
```

```
        {
                System.out.println("a = "+a);
        }
}

public class B extends A
{
        public static void main(String[] args)
        {
                B b = new B();
                b.show();
   }
}
```

Output

```
a=10
```

Here, show() method is declared in class A, but using child class Demo object, it is invoked. That shows the inheritance between these two classes.

Multilevel Inheritance : When a class extends to another class that also extends some other class forms a multilevel inheritance. For example a class C extends to class B that also extends to class A and all the data members an methods of class A and B are now accessible in class C.

Example 15: Multilevel Inheritance

```
class A
{
        int a = 10;
        void show()
        {
                System.out.println("a = "+a);
        }
}

class B extends A
{
        int b = 10;
        void showB()
        {
```

```
                System.out.println("b = "+b);
        }
}

public class C extends B
{
        public static void main(String[] args)
        {
                C c = new C();
                c.show();
                c.showB();
        }
}
```

Output

```
a=10
b=10
```

Hierarchical Inheritance: When a class is extended by two or more classes, it forms hierarchical inheritance. For eg. class B extends to class A and class C also extends to class A in that case both B and C share properties of class A.

Example 16: Hierarchical Inheritance

```
class A
{
        int a = 10;
        void show()
        {
                System.out.println("a = "+a);
        }
}

class B extends A
{
        int b = 10;
        void showB()
        {
                System.out.println("b = "+b);
        }
```

```
}

public class C extends A
{
        public static void main(String[] args)
        {
                C c = new C();
                c.show();
                B b = new B();
                b.show();
        }
}
```

Output

```
a = 10
a = 10
```

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.

Overriding Methods

Method overriding is a process of overriding base class method by derived class method with more specific definition. Method overriding performs only if two classes have is-a relationship. It means class must have inheritance. In other words, it is performed between two classes using inheritance relation. In overriding, method of both classes must have same name and equal number of parameters. Method overriding is also referred to as runtime polymorphism because calling method is decided by JVM during runtime. The key benefit of

overriding is the ability to define method that's specific to a particular subclass type.

Rules for Method Overriding

- Method name must be same for both parent and child classes.
- Access modifier of child method must not restrictive than parent class method.
- Private, final and static methods cannot be overridden.
- There must be an IS-A relationship between classes (inheritance).

Example 17: Method overriding

```
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 18: Method overriding

```
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
```

Difference between Overloading and Overriding

Method overloading and Method overriding seems to be similar concepts but they are not.

Method Overloading	Method Overriding
Parameter must be different and name must be same.	Both name and parameter must be same.
Compile time polymorphism.	Runtime polymorphism.
Increase readability of code.	Increase reusability of code.
Access specifier can be changed.	Access specifier cannot be more restrictive than original method(can be less restrictive).
It is Compiled Time Polymorphism.	It is Run Time Polymorphism.
It is performed within a class	It is performed between two classes using inheritance relation.
It is performed between two classes using inheritance relation.	It requires always inheritance.
It should have methods with the same name but a different signature.	It should have methods with same name and signature.
It can not have the same return type.	It should always have the same return type.
It can be performed using the static method	It cannot be performed using the static method

It uses static binding	It uses the dynamic binding.
Access modifiers and Non-access modifiers can be changed.	Access modifiers and Non-access modifiers can not be changed.
It is code refinement technique.	It is a code replacement technique.
No keywords are used while defining the method.	Virtual keyword is used in the base class and overrides keyword is used in the derived class.
Private, static, final methods can be overloaded	Private, static, final methods can not be overloaded
No restriction is Throws Clause.	Restriction in only checked exception.
It is also known as Compile time polymorphism or static polymorphism or early binding	It is also known as Run time polymorphism or Dynamic polymorphism or Late binding

super keyword

The super keyword in Java is a reference variable which is used to refer immediate parent class object. Whenever an instance of subclass is created, an instance of parent class is created implicitly which is referred by super reference variable. The ***super*** keyword can be used

- ***to refer immediate parent class instance variable.***
- ***to invoke immediate parent class method when method is overrided.***

And super() can be used to invoke immediate parent class constructor.

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.

In the following example 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.

Example 19: 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 which is already
        present in the Superclass   */
        int num = 110;
        void printNumber()
        {
                System.out.println(super.num);
                System.out.println(num);
        }
        public static void main(String args[])
        {
                Subclass obj= new Subclass();
                obj.printNumber();
        }
}
```

Output

```
100
110
```

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. 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(*syntax:* **super.method_name**) you can call the method of parent class (the method which is overridden).

Example 20: super keyword for invoking parent class method

```
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
```

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 when we create the object of child class, while execution, 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-argument constructor of parent class) as the first statement in the constructor of child class.

Example 21: super keyword for invoking parent class constructor

```
class Parentclass
{
        Parentclass()
        {
                System.out.println("Constructor of parent class");
        }
}
class Subclass extends Parentclass
{
        Subclass()
        {
                /* Compiler 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!
```

Dynamic method dispatch

Method overriding is one of the ways in which Java supports Runtime Polymorphism. Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.

When an overridden method is called through a superclass reference, Java determines which version (superclass/subclasses) of that method is to be executed based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time.

At run-time, it depends on the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed. A superclass reference variable can refer to a subclass object. This is also known as **upcasting**. Java uses this fact to resolve calls to overridden methods at run time.

Example 22: Dynamic Method Dispatch

```
//A Java program to illustrate Dynamic Method
// Dispatch using hierarchical inheritance
```

```
class A
{
        void m1()
        {
                System.out.println("Inside A's m1 method");
        }
}

class B extends A
{       // overriding m1()
        void m1()
        {
                System.out.println("Inside B's m1 method");
        }
}

class C extends A
{       // overriding m1()
        void m1()
        {
                System.out.println("Inside C's m1 method");
        }
}

class Dispatch
{
        public static void main(String args[])
        {
                A a = new A();      // object of type A
                B b = new B();              // object of type B
                C c = new C();              // object of type C
                A ref;              // obtain a reference of type A
                ref = a;            // ref refers to an A object
                ref.m1();           // calling A's version of m1()
                ref = b;            // now ref refers to a B object
                ref.m1();           // calling B's version of m1()
                ref = c;            // now ref refers to a C object
```

```
            ref.m1();           // calling C's version of m1()
        }
}
```

Output

```
Inside A's m1 method
Inside B's m1 method
Inside C's m1 method
```

final keyword

Final modifier is used to declare a field as final. It can be used with variable, method or a class.

If we declare a variable as final then it prevents its content from being modified. The variable acts like a constant. Final field must be initialized when it is declared.

If we declare a method as final then it prevents it from being overridden.

If we declare a class as final the it prevents from being inherited. We cannot inherit final class in Java.

Example 23: final variable

```
class Test
{
        final int a = 10;
        public static void main(String[] args)
        {
                Test test = new Test();
                test.a = 15; // compile error
                System.out.println("a = "+test.a);
        }
}
```

Output

```
error: The final field Test.a cannot be assigned
```

A method which is declared using final keyword known as final method. It is useful when we want to prevent a method from overridden.

Example 24: final method

```
class StudyTonight
{
	final void learn()
	{
		System.out.println("learning something new");
	}
}
// concept of Inheritance
class Student extends StudyTonight
{
	void learn()
	{
		System.out.println("learning something interesting");
	}
	public static void main(String args[])
	{
		Student object= new Student();
		object.learn();
	}
}
```

Output

```
error: Cannot override the final method from StudyTonight
```

A class can also be declared as final. A class declared as final cannot be inherited. The String class in java.lang package is an example of a final class.

Example 25: final class

```
final class ABC
{
	int a = 10;
	void show()
	{
		System.out.println("a = "+a);
	}
}
public class Demo extends ABC
```

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

Output

error:The type Demo cannot subclass the final class ABC

Abstract methods and classes

A class which is declared using ***abstract*** keyword is known as an **abstract class**. An abstract class may or may not have abstract methods. An abstract class can have concrete methods (normal methods). **An abstract class cannot be instantiated, which means you are not allowed to create an object of it.**

Points to be remember :

- An abstract class must be declared with an abstract keyword.
- It can have abstract and non-abstract methods.
- It cannot be instantiated.
- It is used for abstraction.

Syntax :

```
abstract class class_name { }
```

Method that are declared without any body within an abstract class are called abstract method. The method body will be defined by its subclass. Abstract method can never be final and static. Any class that extends an abstract class must implement all the abstract methods.

Syntax:

```
abstract return_type function_name (); //No definition
```

Example 26: abstract class and method

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

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

class TestAbstraction1
{
	public static void main(String args[])
	{
		Shape s=new Circle1();
		s.draw();
	}
}
```

Output

```
drawing circle
```

Example 27: abstract class and method

```
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 %

Interfaces

Interface looks like a class but it is not a class. An interface can have methods and variables just like the class but the methods declared in interface are by default abstract (only method signature, no body). Also, the variables declared in an interface are public, static & final by default. Interface is a concept which is used to achieve abstraction in Java. This is the only way by which we can achieve full abstraction means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default. Interfaces are syntactically similar to classes, but it is not possible to create an object for an Interface and their methods are declared without any body. An interface is declared by using the **interface** keyword. A class that implements an interface must implement all the methods declared in the interface.

Advantages of Interface

- **It Support multiple inheritance**

- It helps to achieve abstraction
- Interface can extend one or more other interface.
- Interface cannot implement a class.
- Interface can be nested inside another interface.

All methods declared inside interfaces are implicitly public and abstract, even if you don't use public or abstract keyword.

Syntax:

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

When an interface inherits another interface extends keyword is used whereas class use ***implements*** keyword to inherit an interface.

Example 28: interface

```
interface Moveable
{
        int AVG-SPEED = 40;
        void move();
}

class Vehicle implements Moveable
{
        public void move()
        {
                System .out. print in ("Average speed is"+AVG-SPEED");
        }
        public static void main (String[] arg)
        {
                Vehicle vc = new Vehicle();
                vc.move();
        }
```

```
}
```

Output

Average speed is 40.

Example 29: interface- Multiple inheritance

```
class ClassA
{
	void dispA()
	{
		System.out.println("disp() method of ClassA");
	}
}
interface InterfaceB
{
	void show();
}
interface InterfaceC
{
	void show();
}
class ClassD extends ClassA implements InterfaceB,InterfaceC
{
	void show()
	{
		System.out.println("show() method implementation");
	}
	void dispD()
	{
		System.out.println("disp() method of ClassD");
	}
	public static void main(String args[])
	{
		ClassD d = new ClassD();
		d.dispD();
		d.show();
	}
}
```

Output

disp() method of ClassD
show() method implementation

Differentiate Between Concrete Class, Abstract Class, Final Class, Interface.

Class, interface, abstract class, final class are the important component of the Java language.

Concrete Class: A class that has all its methods implemented, no method is present without body is known as concrete class. In other words, a class that contains only non-abstract method will be called concrete class.

Abstract Class: A class which is declared as abstract using abstract keyword is known as abstract class. Abstract contains abstract methods and used to achieve abstraction, an important feature of OOP programming. Abstract class cannot be instantiated.

Interface: Interface is a blueprint of an class and used to achieve abstraction in Java. Interface contains abstract methods and default, private methods. We cannot create object of the interface. Interface can be used to implement multiple inheritance in Java.

Final class: Final class is a class, which is declared using final keyword. Final class are used to prevent inheritance, since we cannot inherit final class. We can create its object and can create static and non-static methods as well.

Difference between abstract class and interface

Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated. But there are many differences between abstract class and interface that are given below.

	Abstract class	**Interface**
1	Abstract class can **have abstract and non-abstract** methods.	Interface can have **only abstract** methods. It can have **default and static methods** also.

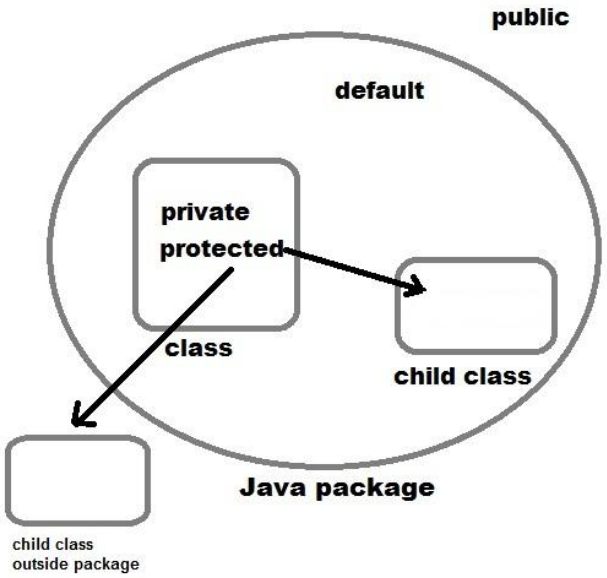
2	Abstract class **doesn't support multiple inheritance**.	Interface **supports multiple inheritance**.
3	Abstract class **can have final, non-final, static and non-static variables**.	Interface has **only static and final variables**.
4	Abstract class **can provide the implementation of interface**.	Interface **can't provide the implementation of abstract class**.
5	The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.
6	An **abstract class** can extend another Java class and implement multiple Java interfaces.	An **interface** can extend another Java interface only.
7	An **abstract class** can be extended using keyword "extends".	An **interface** can be implemented using keyword "implements".
8	A Java **abstract class** can have class members like private, protected, etc.	Members of a Java interface are public by default.

9	**Example:**
	public abstract class Shape{
	public abstract void draw();
	}
		Example:
		public interface Drawable{
		void draw();
		}

Visibility control

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.

Various modifiers and their scope of accessibilities are shown in following table

Access Modifier	**public**	**protected**	**Friendly (default)**	**private**
Same Class	Yes	Yes	Yes	Yes
Subclass in same package	Yes	Yes	Yes	No
Other classes in same package	Yes	Yes	Yes	No
Subclasses in other packages	Yes	Yes	No	No
Non subclasses in other packages	Yes	No	No	No

Example 30: visibility control - private

```
class A
{
        private int data=40;
        private void msg()
        {
              System.out.println("Hello java");
        }
}
public class Simple
{
        public static void main(String args[])
        {
                A obj=new A();
                System.out.println(obj.data);//Compile Time Error
                obj.msg();//Compile Time Error
        }
}
```