```
class Student{
 int id;//field or data member or instance variable
 String name;
```

```
public static void main(String args[]){
  Student s1=new Student();//creating an object of Student
  System.out.println(s1.id);//accessing member through
reference variable
  System.out.println(s1.name);   } }
```

Initialization through reference

```
Student s1=new Student();
  s1.id=101;
  s1.name="Sonoo";
  System.out.println(s1.id+" "+s1.name);
```

Initialization through method

```
Student s1=new Student();
 s1.insertRecord(111,"Karan");
```

different ways to create an object in Java

```
new Calculation();//anonymous object
```

Calling method through reference:

```
Calculation c=new Calculation();
c.fact(5);
```

Calling method through anonymous object

```
new Calculation().fact(5);
```

Creating multiple objects by one type only:

```
Rectangle r1=new Rectangle(),r2=new Rectangle();//creating
two objects
```

default constructor:

```
class Bike1{
Bike1(){System.out.println("Bike is created");}
public static void main(String args[]){
Bike1 b=new Bike1();
}
}
```

parameterized constructor:

```
class Student4{     int id;     String name;
       Student4(int i,String n){    id = i;     name = n;     }
   void display(){System.out.println(id+" "+name);}
     public static void main(String args[]){
   Student4 s1 = new Student4(111,"Karan");
   Student4 s2 = new Student4(222,"Aryan");
   s1.display();
   s2.display();    } }
```

Java Copy Constructor

we are going to copy the values of one object into another using java constructor.

```
class Student6{
   int id;
   String name;
   Student6(int i,String n){     id = i;     name = n;     }
       Student6(Student6 s){
   id = s.id;
   name =s.name;     }
   void display(){System.out.println(id+" "+name);}
     public static void main(String args[]){
   Student6 s1 = new Student6(111,"Karan");
   Student6 s2 = new Student6(s1);
   s1.display();
   s2.display();    } }
```

static keyword:

The static can be:

variable (also known as class variable)

method (also known as class method)

block

nested class

counter without static variable:

```
class Counter{
int count=0;//will get memory when instance is created
  Counter(){
count++;
System.out.println(count);
}
  public static void main(String args[]){
 Counter c1=new Counter();
Counter c2=new Counter();
Counter c3=new Counter();   } }
```

Output:1

1

1

counter by static variable

```
class Counter2{
```

static method that performs normal calculation:

```
class Calculate{
```

```
static int count=0;//will get memory only once and retain it
s value
  Counter2(){
count++;
System.out.println(count);  }
  public static void main(String args[]){
  Counter2 c1=new Counter2();
Counter2 c2=new Counter2();
Counter2 c3=new Counter2();    } }
Output:1
      2
      3
```

```
  static int cube(int x){
  return x*x*x;
  }

  public static void main(String args[]){
  int result=Calculate.cube(5);
  System.out.println(result);
  }
}
Output:125
```

```
class A{
 int a=40;//non static

 public static void main(String args[]){
  System.out.println(a);   } }
Output:Compile Time Error
```

Example of static block

```
class A2{
  static{System.out.println("static block is invoked");}
  public static void main(String args[]){
   System.out.println("Hello main");
  }
}
```

```
class Student{
int rollno;
String name;
float fee;
Student(int rollno,String name,float fee){
rollno=rollno;
name=name;
fee=fee;
}
```

Output:

```
0 null 0.0
0 null 0.0
```

```
class Student{
int rollno;
String name;
float fee;
Student(int rollno,String name,float fee){
this.rollno=rollno;
this.name=name;
this.fee=fee;
}
```

Output:

```
111 ankit 5000
112 sumit 6000
```

this: to invoke current class method

```
class A{
void m(){System.out.println("hello m");}
void n(){
System.out.println("hello n");
//m();//same as this.m()
this.m();  } }
class TestThis4{
public static void main(String args[]){
A a=new A();
a.n();  }}
```

Calling parameterized constructor from default constructor:

```
class A{
A(){
this(5);
System.out.println("hello a");
}
A(int x){
System.out.println(x);
} }
class TestThis6{
public static void main(String args[]){
A a=new A();  }}
```

this: to invoke current class method

```
class A{
void m(){System.out.println("hello m");}
void n(){
System.out.println("hello n");
//m();//same as this.m()
this.m();
}
}
class TestThis4{
public static void main(String args[]){
A a=new A();
a.n();
}}
```

this() : to invoke current class constructor:

```
class A{
A(){System.out.println("hello a");}
A(int x){
this();
System.out.println(x);
}
}
class TestThis5{
public static void main(String args[]){
    A a=new A(10);
    }}
```

this: to pass as an argument in the method

```
class S2{
  void m(S2 obj){
  System.out.println("method is invoked");    }
  void p(){
  m(this);   }
  public static void main(String args[]){
  S2 s1 = new S2();
    s1.p();    } }
```

this keyword can be used to return current class instance:-

```
return_type method_name(){  return this;  }
```

this: to pass as argument in the constructor call

```
class B{
  A4 obj;
  B(A4 obj){
    this.obj=obj;    }
  void display(){
    System.out.println(obj.data);//using data member o
f A4 class    } }
     class A4{
      int data=10;
      A4(){
       B b=new B(this);
       b.display();    }
      public static void main(String args[]){
       A4 a=new A4();    } }
```

```
class A{
A getA(){
return this;
}
void msg(){System.out.println("Hello java");}  }
class Test1{
public static void main(String args[]){
new A().getA().msg();  }  }
```

Why multiple inheritance is not supported in java?

```
class A{ void msg(){System.out.println("Hello");}  }
class B{
void msg(){System.out.println("Welcome");}  }
class C extends A,B{//suppose if it were
 Public Static void main(String args[]){
   C obj=new C();
   obj.msg();//Now which msg() method would be invo
ked?  } }
```

Aggregation in Java

```
class Operation{   int square(int n){
  return n*n;   } }
  class Circle{
 Operation op;//aggregation
 double pi=3.14;
```

Overloading is not possible by changing the return type of method

```
class Adder{
static int add(int a,int b){return a+b;}
static double add(int a,int b){return a+b;}
}
class TestOverloading3{
public static void main(String[] args){
System.out.println(Adder.add(11,11));//ambiguity
}}
```

Output:

```
Compile Time Error: method add(int,int) is already defined in class Adder
```

Method Overloading with TypePromotion

```
class OverloadingCalculation1{
  void sum(int a,long b){System.out.println(a+b);
}
  void sum(int a,int b,int c){System.out.println(a+
b+c);}
  public static void main(String args[]){
  OverloadingCalculation1 obj=new OverloadingCalc
ulation1();
  obj.sum(20,20);//now second int literal will be pro
moted to long
  obj.sum(20,20,20);

  }
}
```

Understanding meaningful example of Aggregation

```
public class Address {
String city,state,country;
public Address(String city, String state, String country
) {
   this.city = city;
   this.state = state;
   this.country = country;
}   }
```

```
public class Emp {
int id;
String name;
Address address;
  public Emp(int id, String name,Address address) {
   this.id = id;
   this.name = name;
   this.address=address;  }
void display(){
System.out.println(id+" "+name);
System.out.println(address.city+" "+address.state+" "
+address.country);  }
public static void main(String[] args) {
Address address1=new Address("gzb","UP","india");
Address address2=new Address("gno","UP","india");
Emp e=new Emp(111,"varun",address1);
Emp e2=new Emp(112,"arun",address2);
e.display();
e2.display(); } }
```

Usage of java super Keyword

1. super can be used to refer immediate parent class instance variable.
2. super can be used to invoke immediate parent class method.
3. super() can be used to invoke immediate parent
4. class constructor.

Method Overloading: changing no. of arguments

```
class Adder{
static int add(int a,int b){return a+b;}
static int add(int a,int b,int c){return a+b+c;}
}
class TestOverloading1{
public static void main(String[] args){
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(11,11,11));
}}
```

Method Overloading and Type Promotion

```
class OverloadingCalculation1{
  void sum(int a,long b){System.out.println(a+b);
}
  void sum(int a,int b,int c){System.out.println(a+
b+c);}

  public static void main(String args[]){
  OverloadingCalculation1 obj=new OverloadingCalc
ulation1();
  obj.sum(20,20);//now second int literal will be pro
moted to long
  obj.sum(20,20,20);

  }
}
```

Role of Private Constructor

```
class A{

private A(){}//private constructor

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

Method Overloading: changing data type of arguments

```
class Adder{
static int add(int a, int b){return a+b;}
static double add(double a, double b){return a+b;
}
}
class TestOverloading2{
public static void main(String[] args){
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(12.3,12.6));
}}
```

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**.

```
class Vehicle{
void run(){System.out.println("Vehicle is running");}
}
class Bike2 extends Vehicle{
void run(){System.out.println("Bike is running safely")
;}

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

protected access modifier

The **protected access modifier** is accessible within package and outside the package but through inheritance only.

The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.

```
//save by A.java
package pack;
public class A{
protected void msg(){System.out.println("Hello");}
}
```

Encapsulation in Java

Encapsulation in java is a *process of wrapping code and data together into a single unit*, for example capsule i.e. mixed of several medicines.

The **Java Bean** class is the example of fully encapsulated class.

```
//save as Student.java
package com.javatpoint;
public class Student{
private String name;
  public String getName(){
return name;
}
public void setName(String name){
this.name=name
).      } }
```

```
//save as Test.java
package com.javatpoint;
class Test{
public static void main(String[] args){
Student s=new Student();
s.setName("vijay");
System.out.println(s.getName());
}
}
..
```

super can be used to invoke parent class method

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

```
//save by B.java
package mypack;
import pack.*;

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

```
Output:Hello
```

Object class in Java

The **Object class** is the parent class of all the classes in java by default. In other words, it is the topmost class of java.

```
Object obj=getObject();//we don't know what object w
ill be returned from this method
```

super keyword in java

The **super** keyword in java is a reference variable which is used to refer immediate parent class object.

Usage of java super Keyword

1. super can be used to refer immediate parent class instance variable.
2. super can be used to invoke immediate parent class method.
3. super() can be used to invoke immediate parent class constructor.

super is used to refer immediate parent class instance variable

```
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 Anim
```

```
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();
. }}
```

super is used to invoke parent class constructor.

```
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();
. }}
```

Instance initializer block

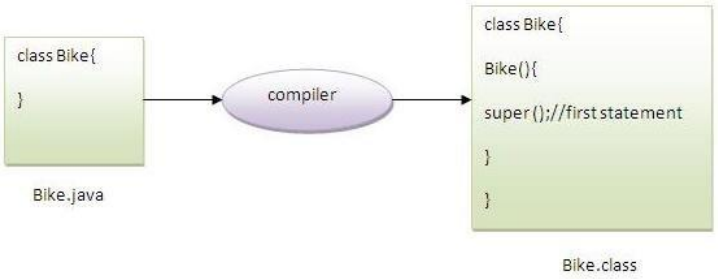
Instance Initializer block **is used to initialize the instance data member. It run each time when object of the class is created.**

Example of instance initializer block

```
class Bike7{
   int speed;
       Bike7(){System.out.println("speed is "+spee
d);}
```

```
al class
}
. }
. class TestSuper1{
. public static void main(String args[]){
. Dog d=new Dog();
. d.printColor();
. }}
```

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



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

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

What is invoked first, instance initializer block or constructor?

```
class Bike8{
   int speed;

   Bike8(){System.out.println("constructor is invoked")
```

```
    {speed=100;}

    public static void main(String args[]){
    Bike7 b1=new Bike7();
    Bike7 b2=new Bike7();
    }
}
```

There are three places in java where you can perform operations:

1. method
2. constructor
3. block

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

```
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
```

```
;}

    {System.out.println("instance initializer block invoked");}

    public static void main(String args[]){
    Bike8 b1=new Bike8();
    Bike8 b2=new Bike8();
    }
}
```

```
Output:instance initializer block invoked
       constructor is invoked
       instance initializer block invoked
       constructor is invoked
```

If you make any method as final, you cannot override it.

```
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();
```

Java final class

If you make any class as final, you cannot extend it.

```
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 Honda();
  honda.run();
  }
}
```

```
Output:Compile Time Error
```

What is blank or uninitialized final variable?

```
class Student{
int id;
String name;
final String PAN_CARD_NUMBER;
...
}
```

static blank final variable

A static final variable that is not initialized at the time of declaration is known as static blank final variable. It can be initialized only in static block.

Example of static blank final variable

```
class A{
  static final int data;//static blank final variable
  honda.run();    } }
```

```
Output:Compile Time Error
```

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...
```

Can we initialize blank final variable?

Yes, but only in constructor. For example:

```
class Bike10{
  final int speedlimit;//blank final variable
     Bike10(){    speedlimit=70;
  System.out.println(speedlimit);
  }      public static void main(String args[]){
    new Bike10();   } }
```

```
Output:70
```

Can we declare a constructor final?

No, because constructor is never inherited.

Polymorphism in Java

Polymorphism in java is a concept by which we can perform a *single action by different ways*. Polymorphism is derived from 2 greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.

There are two types of polymorphism in java: compile time polymorphism and runtime polymorphism. We

```
  static{ data=50;}
  public static void main(String args[]){
    System.out.println(A.data);
  }
}
```

What is final parameter?

If you declare any parameter as final, you cannot change the value of it.

```
class Bike11{
  int cube(final int n){
   n=n+2;//can't be changed as n is final
   n*n*n;
  }
  public static void main(String args[]){
    Bike11 b=new Bike11();
    b.cube(5);
  }
}
```

```
Output:Compile Time Error
```

Upcasting


```
class A{}
class B extends A{}

A a=new B();//upcasting
```

can perform polymorphism in java by method overloading and method overriding.

If you overload static method in java, it is the example of compile time polymorphism. Here, we will focus on runtime polymorphism in java.

Runtime Polymorphism in Java

Runtime polymorphism or **Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.

Example of Java Runtime Polymorphism

In this example, we are creating two classes Bike and Splendar. Splendar class extends Bike class and overrides its run() method. We are calling the run method by the reference variable of Parent class. Since it refers to the subclass object and subclass method overrides the Parent class method, subclass method is invoked at runtime.

Since method invocation is determined by the JVM not compiler, it is known as runtime polymorphism.

```
class Bike{
  void run(){System.out.println("running");}
}
class Splender extends Bike{
  void run(){System.out.println("running safely with 60km");}

  public static void main(String args[]){
    Bike b = new Splender();//upcasting
    b.run();
  } }
```

```
Output:running safely with 60km.
```

Java Runtime Polymorphism Example: Bank

Consider a scenario, Bank is a class that provides method to get the rate of interest. But, rate of interest may differ according to banks. For example, SBI, ICICI and AXIS banks are providing 8.4%, 7.3% and 9.7% rate of interest.

```
class Bank{

float getRateOfInterest(){return 0;}  }

class SBI extends Bank{
float getRateOfInterest(){return 8.4f;}  }
class ICICI extends Bank{
float getRateOfInterest(){return 7.3f;}  }
class AXIS extends Bank{
float getRateOfInterest(){return 9.7f;}  }
class TestPolymorphism{
public static void main(String args[]){
Bank b;
b=new SBI();
System.out.println("SBI Rate of Interest: "+b.getRateOfInterest());
b=new ICICI();
System.out.println("ICICI Rate of Interest: "+b.getRateOfInterest());
b=new AXIS();
System.out.println("AXIS Rate of Interest: "+b.getRateOfInterest());  }  }
```

```
SBI Rate of Interest: 8.4
ICICI Rate of Interest: 7.3
AXIS Rate of Interest: 9.7
```

Java Runtime Polymorphism Example: Animal

```
class Animal{
void eat(){System.out.println("eating...");}  }
class Dog extends Animal{
void eat(){System.out.println("eating bread...");}  }
class Cat extends Animal{
void eat(){System.out.println("eating rat...");}  }
class Lion extends Animal{
void eat(){System.out.println("eating meat...");}  }
class TestPolymorphism3{
public static void main(String[] args){
Animal a;
a=new Dog();
a.eat();
a=new Cat();
a.eat();
a=new Lion();
a.eat();  }}
```

```
eating bread...
eating rat...
eating meat...
```

Java Runtime Polymorphism with Multilevel Inheritance

```
class Animal{
void eat(){System.out.println("eating");}  }
class Dog extends Animal{
void eat(){System.out.println("eating fruits");}  }
class BabyDog extends Dog{
void eat(){System.out.println("drinking milk");}
public static void main(String args[]){
Animal a1,a2,a3;
a1=new Animal();
a2=new Dog();
a3=new BabyDog();
a1.eat();
a2.eat();
a3.eat();  }  }
```

output:----

```
eating
eating fruits
drinking Milk
```

Static Binding and Dynamic Binding

Connecting a method call to the method body is known as binding.
There are two types of binding

1. static binding (also known as early binding).
2. dynamic binding (also known as late binding).

Example of static binding

```
class Dog{
 private void eat(){System.out.println("dog is eating...");}

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

Example of dynamic binding

```
class Animal{
 void eat(){System.out.println("animal is eating...");}
}

class Dog extends Animal{
 void eat(){System.out.println("dog is eating...");}
   public static void main(String args[]){
  Animal a=new Dog();
  a.eat();
} }
```

Output:dog is eating...

The **java instanceof operator** is used to test whether the object is an instance of the specified type (class or subclass or interface).

The instanceof in java is also known as type *comparison operator* because it compares the instance with type. It returns either true or false. If we apply the instanceof operator with any variable that has null value, it returns false.

example of java instanceof operator

```
class Animal{}
class Dog1 extends Animal{//Dog inherits Animal
   public static void main(String args[]){
 Dog1 d=new Dog1();
 System.out.println(d instanceof Animal);//true
 } }
```

```
Output:true
```

```
class Dog2{
 public static void main(String args[]){
  Dog2 d=null;
  System.out.println(d instanceof Dog2);//false
 } }
```

```
Output:false
```

Abstraction in Java

Abstraction is a process of hiding the implementation details and showing only functionality to the user.

Another way, it shows only important things to the user and hides the internal details for example sending sms, you just type the text and send the message. You don't know the internal processing about the message delivery.

Abstraction lets you focus on what the object does instead of how it does it.

Ways to achieve Abstaction

There are two ways to achieve abstraction in java

1. Abstract class (0 to 100%)
2. Interface (100%)

Abstract class in Java

A class that is declared as abstract is known as **abstract class**. It needs to be extended and its method implemented. It cannot be instantiated.

```
abstract class Bike{
  abstract void run();  }
class Honda4 extends Bike{
void run(){System.out.println("running safely..")
;}  public static void main(String args[]){
 Bike obj = new Honda4();
 obj.run();  } }
running safely..
```

Abstract class having constructor, data member, methods etc.

```
//example of abstract class that have method body
 abstract class Bike{
   Bike(){System.out.println("bike is created");}
   abstract void run();
   void changeGear(){System.out.println("gear chang
ed");}
 }
   class Honda extends Bike{
 void run(){System.out.println("running safely..");}
 }
 class TestAbstraction2{
 public static void main(String args[]){
  Bike obj = new Honda();
  obj.run();
  obj.changeGear();   } }
```

```
bike is created
running safely..
gear changed
```

An **interface in java** is a blueprint of a class. It has static constants and abstract methods.

The interface in java is **a mechanism to achieve abstraction**. There can be only abstract methods in the java interface not method body. It is used to achieve abstraction and multiple inheritance in Java.

Java Interface also **represents IS-A relationship**.

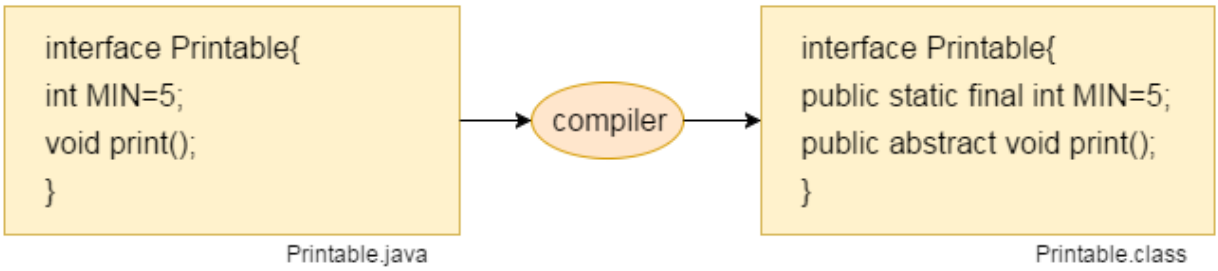
It cannot be instantiated just like abstract class.

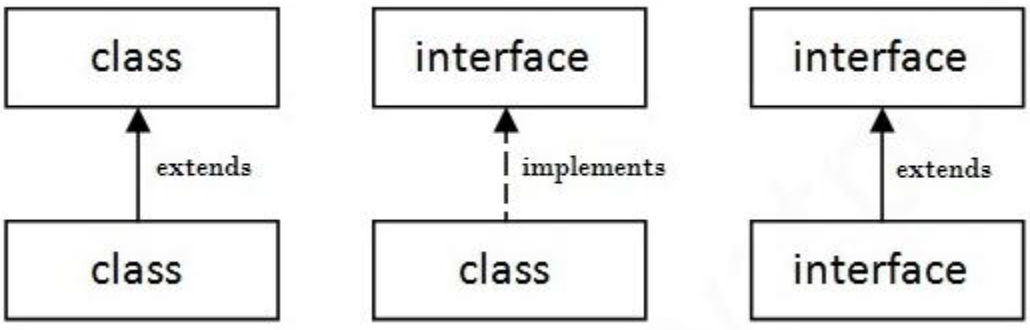
Why use Java interface?

There are mainly three reasons to use interface. They are given below.

- It is used to achieve abstraction.
- By interface, we can support the functionality of multiple inheritance.
- It can be used to achieve loose coupling.

In other words, Interface fields are public, static and final by default, and methods are public and abstract.





Understanding relationship between classes and interfaces

```
interface printable{
void print();  }
class A6 implements printable{
public void print(){System.out.println("Hello");}
public static void main(String args[]){
A6 obj = new A6();
obj.print();
 } }
```

```
Hello
```

```
interface Bank{
float rateOfInterest();  }
class SBI implements Bank{
public float rateOfInterest(){return 9.15f;}  }
class PNB implements Bank{
public float rateOfInterest(){return 9.7f;}  }
class TestInterface2{
public static void main(String[] args){

Bank b=new SBI();
System.out.println("ROI: "+b.rateOfInterest());  }}
```

```
ROI: 9.15
```

Since Java 8, we can have static method in interface. Let's see an example:

```
interface Drawable{
void draw();
static int cube(int x){return x*x*x;}
}
class Rectangle implements Drawable{
public void draw(){System.out.println("drawing rectangle");}
}

class TestInterfaceStatic{
public static void main(String args[]){
Drawable d=new Rectangle();
d.draw();
System.out.println(Drawable.cube(3));
}}
```

```
drawing rectangle
27
```

Nested Interface in Java

```
interface printable{
 void print();
 interface MessagePrintable{
   void msg();
 }
}
```

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. Since Java 8, 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) **Example:**
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

Example of abstract class and interface in Java

```
//Creating interface that has 4 methods
interface A{
void a();//bydefault, public and abstract
void b();  void c();  void d();  }
//Creating abstract class that provides the implementation of one m
ethod of A interface
abstract class B implements A{
public void c(){System.out.println("I am C");}
}    //Creating subclass of abstract class, now we need to provide th
e implementation of rest of the methods
class M extends B{
public void a(){System.out.println("I am a");}
public void b(){System.out.println("I am b");}
public void d(){System.out.println("I am d");}  }
//Creating a test class that calls the methods of A interface
class Test5{
public static void main(String args[]){
A a=new M();  a.a();  a.b();  a.c();  a.d();  }}
```

```
I am a
I am b
I am c
I am d
```

Access Modifiers in java

There are 4 types of java access modifiers:

1.private

2.default

3.protected

4.public

Let's understand the access modifiers by a simple table.

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y