

Yadab Raj Ojha

Sr. Java Developer / Lecture

Email: yadabrajojha@gmail.com

Blog: <http://yro-tech.blogspot.com/>

Java Tutorial: <https://github.com/yrojha4ever/JavaStud>

LinkedIn: <https://www.linkedin.com/in/yrojha>

Twitter: <https://twitter.com/yrojha4ever>

Object Oriented Programming

Class, Object and Encapsulation

- Class and Objects
- Constructor and Encapsulation
- Properties and Methods
- Types of Methods
- Inner Classes
- package and import
- access modifiers and their uses

Inheritance & Polymorphism

- Inheritance
- Inheritance in Java
- Abstract Classes
- Interfaces
- Polymorphism: Dynamic Binding
- Casting Objects

Object Class

Chapter 4

Java OOPs Concepts

The class is at the core of Java. It is the logical construct upon which the entire Java language is built because it defines the shape and nature of an object. As such, the class forms the basis for object-oriented programming in Java. Any concept you wish to implement in a Java program must be encapsulated within a class.

structured programming:

Algorithms + Data Structures = Programs

OOP reverses the order: puts the data first, then looks at the algorithms to operate on the data.

Class

A *class* is the template or blueprint from which objects are made. Perhaps the most important thing to understand about a class is that it defines a new data type. Once defined, this new type can be used to create objects of that type. Thus, a class is a *template* for an object, and an object is an *instance* of a class. Because an object is an instance of a class, you will often see the two words *object* and *instance* used interchangeably. A class is declared by use of the **class** keyword.

```

class classname {
    type instance-variable1;
    type instance-variable2;
    // ...
    type instance-variableN;

    type methodname1(parameter-list) {
        // body of method
    }
    type methodname2(parameter-list) {
        // body of method
    }
    // ...
    type methodnameN(parameter-list) {
        // body of method
    }
}

```

```

public class Box {
    private double width;
    private double height;
    private double depth;
}

```

As stated, a class defines a new type of data. In this case, the new data type is called **Box**. You will use this name to declare objects of type **Box**. It is important to remember that a class declaration only creates a template; it does not create an actual object. Thus, the preceding code does not cause any objects of type **Box** to come into existence.

To actually create a **Box** object, you will use a statement like the following:

```
Box mybox = new Box(); // create a Box object called mybox
```

After this statement executes, **mybox** will be an instance of **Box**. Thus, it will have “physical” reality.

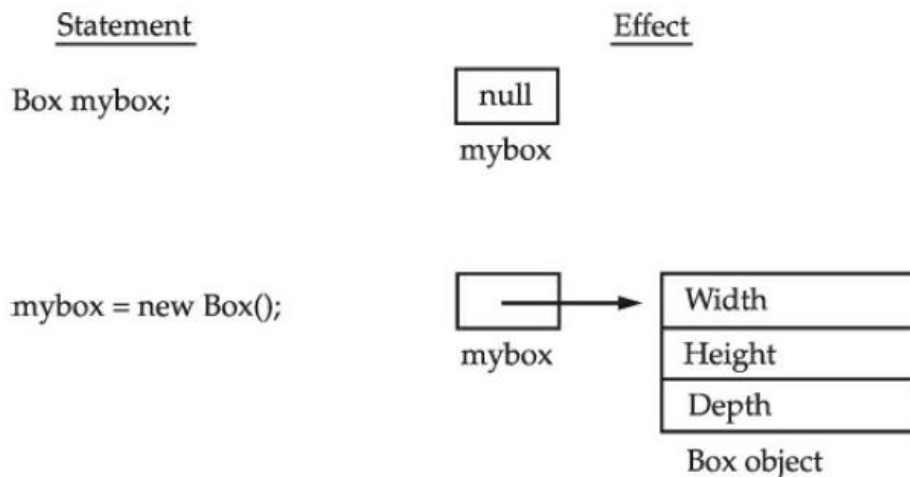
To assign the **width** variable of **mybox** the value 100

```
mybox.width = 100;
```

A Closer Look at new

As just explained, the **new** operator dynamically allocates memory for an object. It has this general form:

```
class-var = new classname ( );
```



Here, *class-var* is a variable of the class type being created. The *classname* is the name of the class that is being instantiated. The class name followed by parentheses specifies the *constructor* for the class. A constructor defines what occurs when an object of a class is created. Constructors are an important part of all classes and have many significant attributes. Most real-world classes explicitly define their own constructors within their class definition. However, if no explicit constructor is specified, then Java will automatically supply a default constructor. This is the case with **Box**.

Objects

To work with OOP, you should be able to identify three key characteristics of objects:

- The object's *behavior*—What can you do with this object, or what methods can you apply to it?
- The object's *state*—How does the object react when you invoke those methods?
- The object's *identity*—How is the object distinguished from others that may have the same behavior and state?

All objects that are instances of the same class share a family resemblance by supporting the same *behavior*. The behavior of an object is defined by the methods that you can call. However, the state of an object does not completely describe it, because each object has a distinct *identity*.

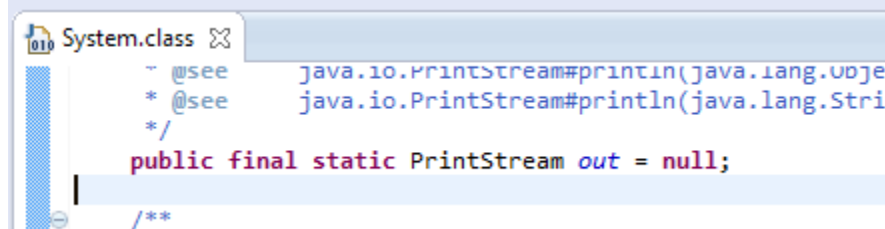
Relationships between Classes

The most common relationships between classes are

- *Dependence* (“uses-a”)
- *Aggregation* (“has-a”)
- *Inheritance* (“is-a”)

The *dependence*, or “uses-a” relationship, is the most obvious and also the most general. For example, the *Order* class uses the *Account* class because *Order* objects need to access *Account* objects to check for credit status. But the *Item* class does not depend on the *Account* class, because *Item* objects never need to worry about customer accounts. Thus, a class depends on another class if its methods use or manipulate objects of that class.

The *aggregation*, or “has-a” relationship, is easy to understand because it is concrete; for example, an *Order* object contains *Item* objects. Containment means that objects of class A contain objects of class B.



The *inheritance*, or “is-a” relationship, expresses a relationship between a more special and a more general class. For example, a *Programmer* class inherits from an *Employee* class.

Using Objects as Parameters

So far, we have only been using simple types as parameters to methods. However, it is both correct and common to pass objects to methods. For example, consider the following short program:

```
PassObjAsParam.java
package oop;

public class PassObjAsParam {

    public static void main(String[] args) {
        Student s1 = new Student(1, 50);
        Student s2 = new Student(1, 50);
        Student s3 = new Student(2, 20);
        System.out.println("Object s1 == s2: " + s1.equals(s2) );
        System.out.println("Object s1 == s3: " + s1.equals(s3) );
    }
}

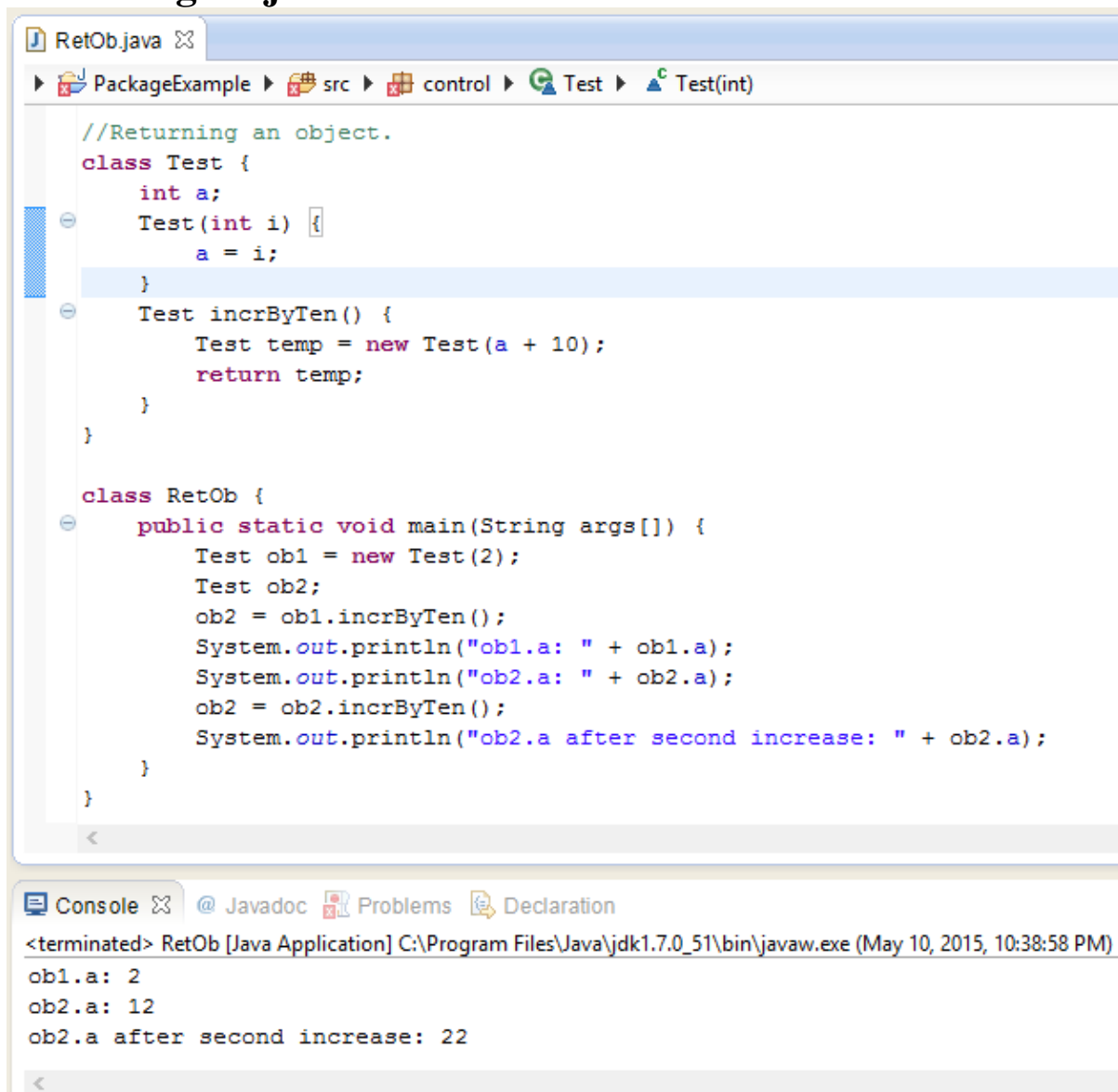
class Student {
    private int id = 10;
    private int rollNo = 30;

    public Student(int id, int rollNo) {
        this.id = id;
        this.rollNo = rollNo;
    }

    boolean equals(Student o) { //Object as a parameter.
        if (this.id == o.id && this.rollNo == o.rollNo) {
            return true;
        }
        return false;
    }
}

<terminated> PassObjAsParam [Java Application] C:\Program Files\Java\jdk1.7.0_79\bin\javaw.exe (Sep 1, 2015, 8:33:
Object s1 == s2: true
Object s1 == s3: false
```

Returning Objects



The screenshot shows an IDE with a Java file named `RetOb.java`. The code defines a `Test` class with an `int` attribute `a`, a constructor `Test(int i)`, and a method `incrByTen()` that returns a new `Test` object with `a` incremented by 10. A `RetOb` class contains a `main` method that creates `Test` objects, calls `incrByTen()`, and prints the values of `a`.

```
//Returning an object.
class Test {
    int a;
    Test(int i) {
        a = i;
    }
    Test incrByTen() {
        Test temp = new Test(a + 10);
        return temp;
    }
}

class RetOb {
    public static void main(String args[]) {
        Test ob1 = new Test(2);
        Test ob2;
        ob2 = ob1.incrByTen();
        System.out.println("ob1.a: " + ob1.a);
        System.out.println("ob2.a: " + ob2.a);
        ob2 = ob2.incrByTen();
        System.out.println("ob2.a after second increase: " + ob2.a);
    }
}
```

The console output shows the execution results:

```
<terminated> RetOb [Java Application] C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe (May 10, 2015, 10:38:58 PM)
ob1.a: 2
ob2.a: 12
ob2.a after second increase: 22
```

A **factory method** is the method that returns the instance of the class. We will learn it later.

Java static keyword

The **static keyword** in java is used for memory management mainly. We can apply java static keyword with variables, methods, blocks and nested class. The static keyword belongs to the class than instance of the class.

The static can be:

1. variable (also known as class variable)
2. method (also known as class method)
3. block
4. nested class

1) Java static variable

If you declare any variable as static, it is known static variable.

- The static variable can be used to **refer the common property of all objects** (that is not unique for each object) e.g. company name of employees, college name of students etc.
- The **static variable gets memory only once in class area** at the time of class loading.

Advantage of static variable

It makes your program **memory efficient** (i.e it saves memory).

Java static property is shared to all objects.

Example of static variable

StaticVariable.java

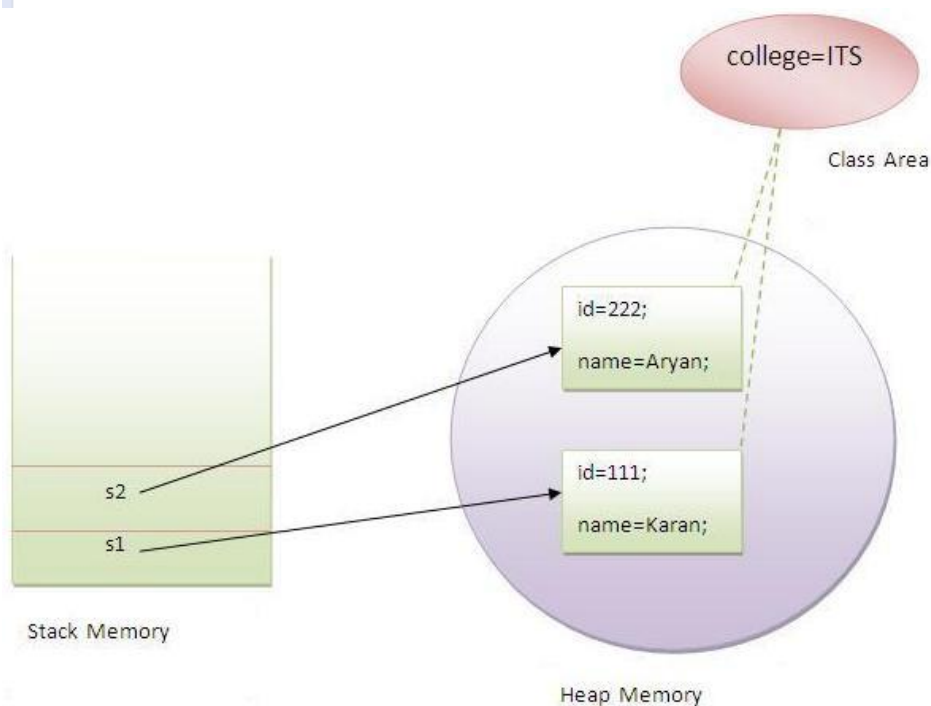
```
public class StaticVariable {  
    int rollno;  
    String name;  
    static String college = "ITS";  
  
    StaticVariable(int r, String n) {  
        rollno = r;  
        name = n;  
    }  
  
    void display() {  
        System.out.println(rollno + " " + name + " " + college + "\n");  
    }  
  
    public static void main(String args[]) {  
        StaticVariable s1 = new StaticVariable(111, "Karan");  
        StaticVariable s2 = new StaticVariable(222, "Aryan");  
  
        s1.display();  
        s2.display();  
    }  
}
```

Console

Markers Properties Servers Data Source Explorer Snippets

<terminated> StaticVariable [Java Application] C:\Program Files\Java\jdk1.7.0_79\bin\javaw.exe (Sep 1, 2015, 9:42:34 AM)
111 Karan ITS

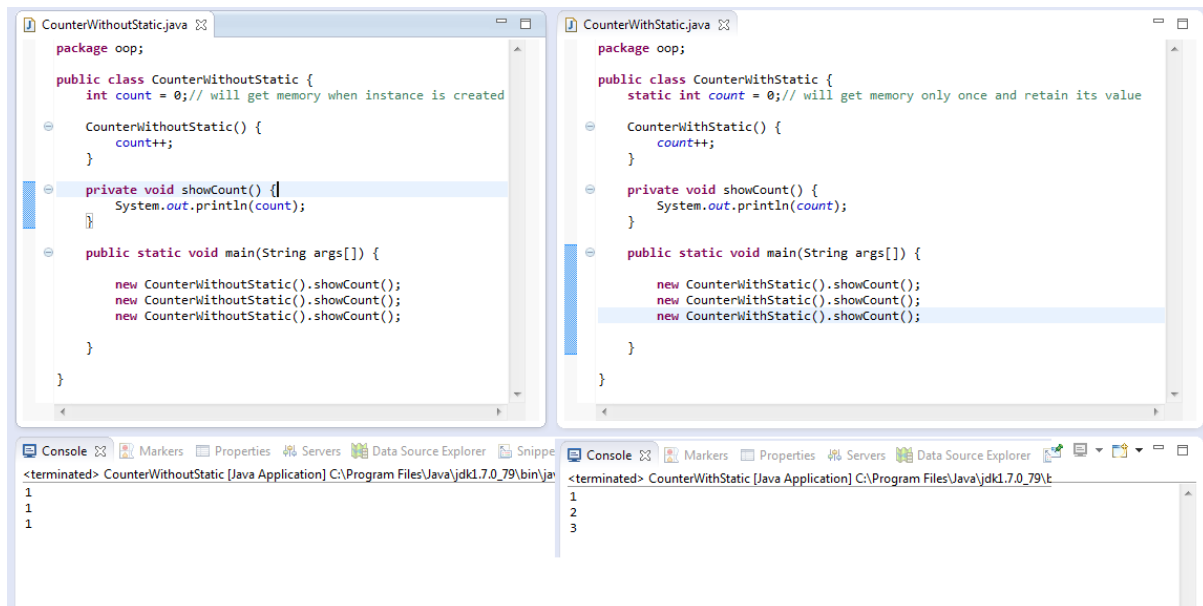
222 Aryan ITS



Program of counter with/without static variable

Without Static Variable: Variable will get memory when object is created.

With Static Variable: Variable will get memory only once and retain its value.



```
package oop;

public class CounterWithoutStatic {
    int count = 0; // will get memory when instance is created

    CounterWithoutStatic() {
        count++;
    }

    private void showCount() {
        System.out.println(count);
    }

    public static void main(String args[]) {
        new CounterWithoutStatic().showCount();
        new CounterWithoutStatic().showCount();
        new CounterWithoutStatic().showCount();
    }
}
```

```
package oop;

public class CounterWithStatic {
    static int count = 0; // will get memory only once and retain its value

    CounterWithStatic() {
        count++;
    }

    private void showCount() {
        System.out.println(count);
    }

    public static void main(String args[]) {
        new CounterWithStatic().showCount();
        new CounterWithStatic().showCount();
        new CounterWithStatic().showCount();
    }
}
```

Console output for CounterWithoutStatic:

```
<terminated> CounterWithoutStatic [Java Application] C:\Program Files\Java\jdk1.7.0_79\bin\jav
1
1
1
```

Console output for CounterWithStatic:

```
<terminated> CounterWithStatic [Java Application] C:\Program Files\Java\jdk1.7.0_79\bin\jav
1
2
3
```

2) Java static method

If you apply static keyword with any method, it is known as static method.

- A **static method belongs to the class rather than object** of a class.
- A static method can be invoked without the need for creating an instance of a class.
- static method can access static data member and can change the value of it.

Example of static method

//Program of changing the common property of all objects (static field).

```
ChangeStaticValue.java
public class ChangeStaticValue {
    int rollno;
    String name;
    static String college = "TRICHANDRA";

    static void change() {
        college = "ASCOL";
    }

    ChangeStaticValue(int r, String n) {
        rollno = r;
        name = n;
    }

    void display() {
        System.out.println(rollno + " " + name + " " + college + "\n");
    }

    public static void main(String args[]) {

        ChangeStaticValue s1 = new ChangeStaticValue(111, "Amrit");
        s1.display();

        ChangeStaticValue.change(); //college:ASCOL
        ChangeStaticValue s2 = new ChangeStaticValue(222, "Bindu");
        ChangeStaticValue s3 = new ChangeStaticValue(333, "Champ");

        s1.display();
        s2.display();
        s3.display();
    }
}
```

Console

<terminated> ChangeStaticValue [Java Application] C:\Program Files\Java\jdk1.7.0_79\bin\javaw.exe (Sep 1, 2015, 9:33:39 AM)

111 Amrit TRICHANDRA S1

111 Amrit ASCOL S1

222 Bindu ASCOL S2

333 Champ ASCOL S3

Restrictions for static method

There are two main restrictions for the static method. They are:

1. The static method can not use non static data member or call non-static method directly.
2. this and super cannot be used in static context.

```
1. class A{
2.     int a=40;//non static
3.
4.     public static void main(String args[]){
5.         System.out.println(a);
6.     }
7. }
```

Test it Now

Output:Compile Time Error

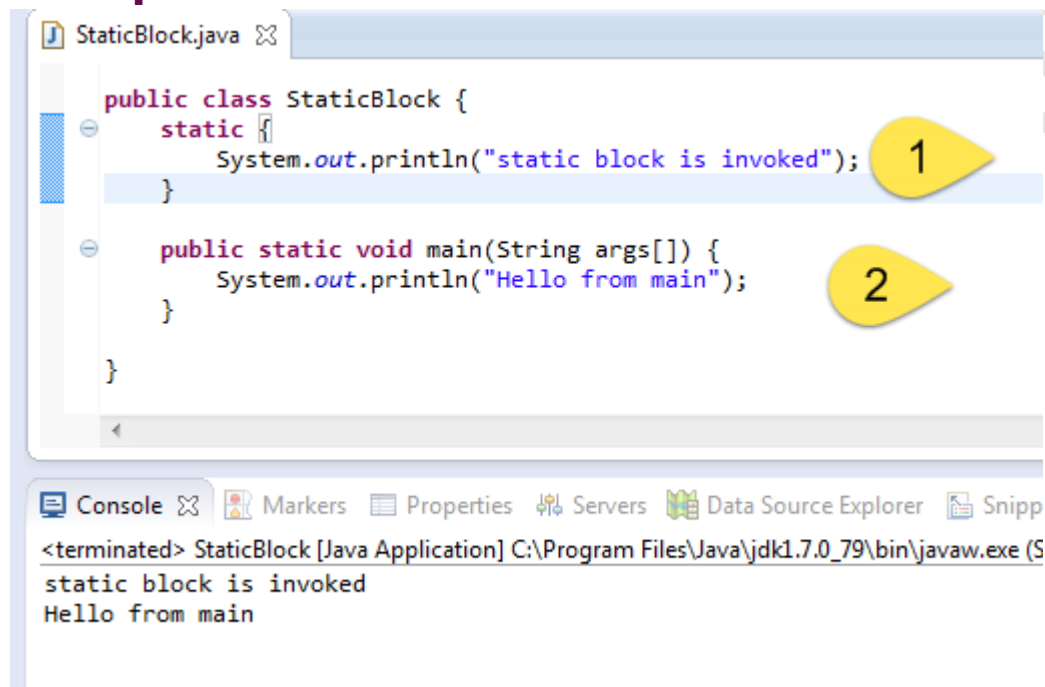
Q) Why java main method is static?

Ans) because object is not required to call static method if it were non-static method, jvm create object first then call main() method that will lead the problem of extra memory allocation.

3) Java static block

- Is used to initialize the static data member.
- It is executed before main method at the time of classloading.

Example of static block



```
StaticBlock.java

public class StaticBlock {
    static {
        System.out.println("static block is invoked");
    }

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

Console

```
<terminated> StaticBlock [Java Application] C:\Program Files\Java\jdk1.7.0_79\bin\javaw.exe (S
static block is invoked
Hello from main
```

Q) Can we execute a program without main() method?

Ans) Yes, one of the way is static block but in previous version of JDK not in JDK 1.7.

```
1. class A3{
2.     static{
3.         System.out.println("static block is invoked");
4.         System.exit(0);
5.     }
6. }
```

Test it Now

Output:static block is invoked (if not JDK7)

In JDK7 and above, output will be:

Output:Error: Main method not found in class A3, please define the main method as: public static void main(String[] args)

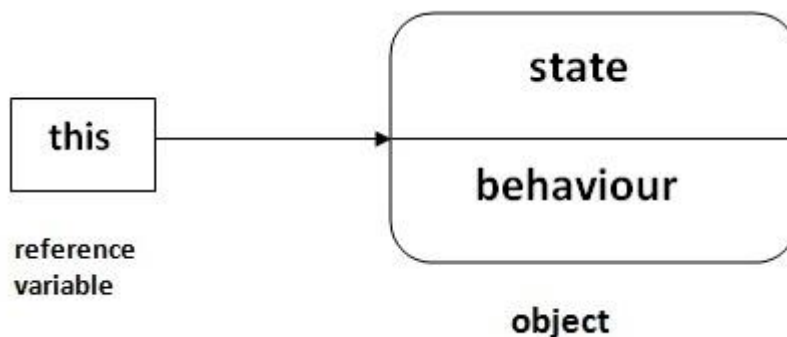
this keyword in java

There can be a lot of usage of **java this keyword**. In java, this is a **reference variable** that refers to the current object.

Usage of java this keyword

Here is given the 6 usage of java this keyword.

1. this keyword can be used to refer current class instance variable.
2. this() can be used to invoke current class constructor.
3. this keyword can be used to invoke current class method (implicitly)



1) The this keyword can be used to refer current class instance variable.

If there is ambiguity between the instance variable and parameter, this keyword resolves the problem of ambiguity.

Understanding the problem without this keyword

Let's understand the problem **if we don't use this keyword** by the example given below:

```
1. class Student10{
2.     int id;
3.     String name;
4.
5.     Student10(int id, String name){
6.         id = id;
7.         name = name;
8.     }
9.     void display(){System.out.println(id+" "+name);}
10.
11.     public static void main(String args[]){
12.         Student10 s1 = new Student10(111,"Karan");
13.         Student10 s2 = new Student10(321,"Aryan");
```

```

14.     s1.display();
15.     s2.display();
16.     }
17.     }

```

Test it Now

```

Output:0 null
       0 null

```

In the above example, parameter (formal arguments) and instance variables are same that is why we are using this keyword to distinguish between local variable and instance variable.

Solution of the above problem by this keyword

```

1.     //example of this keyword
2.     class Student11{
3.         int id;
4.         String name;
5.
6.         Student11(int id,String name){
7.             this.id = id;
8.             this.name = name;
9.         }
10.        void display(){System.out.println(id+" "+name);}
11.        public static void main(String args[]){
12.            Student11 s1 = new Student11(111,"Karan");
13.            Student11 s2 = new Student11(222,"Aryan");
14.            s1.display();
15.            s2.display();
16.        }
17.    }

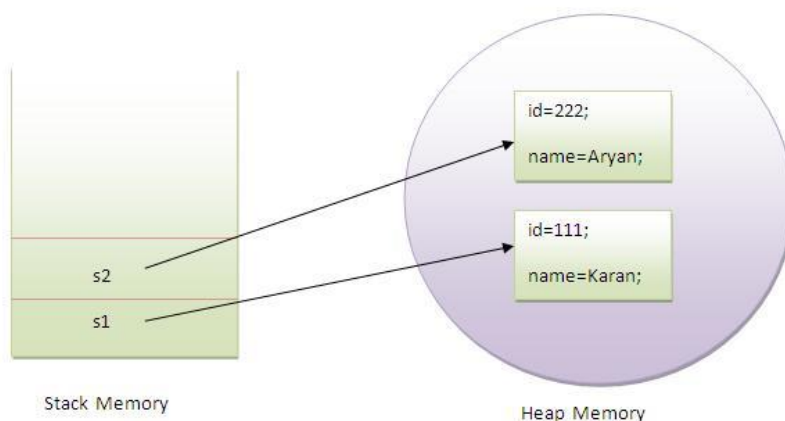
```

Test it Now

```

Output111 Karan
       222 Aryan

```



If local variables(formal arguments) and instance variables are different, there is no need to use this keyword like in the following program:

Program where this keyword is not required

```
1.  class Student12{
2.      int id;
3.      String name;
4.
5.      Student12(int i, String n){
6.          id = i;
7.          name = n;
8.      }
9.      void display(){System.out.println(id+" "+name);}
10.     public static void main(String args[]){
11.         Student12 e1 = new Student12(111,"karan");
12.         Student12 e2 = new Student12(222,"Aryan");
13.         e1.display();
14.         e2.display();
15.     }
16. }
```

Test it Now

```
Output:111 Karan
        222 Aryan
```

2) this() can be used to invoked current class constructor.

The this() constructor call can be used to invoke the current class constructor (constructor chaining). This approach is better if you have many constructors in the class and want to reuse that constructor.

```
1.  //Program of this() constructor call (constructor chaining)
2.
3.  class Student13{
4.      int id;
5.      String name;
6.      Student13(){System.out.println("default constructor is invoked");}
7.
8.      Student13(int id, String name){
9.          this ();//it is used to invoked current class constructor.
10.         this.id = id;
11.         this.name = name;
12.     }
13.     void display(){System.out.println(id+" "+name);}
14.
15.     public static void main(String args[]){
16.         Student13 e1 = new Student13(111,"karan");
17.         Student13 e2 = new Student13(222,"Aryan");
18.         e1.display();
19.         e2.display();
20.     }
21. }
```


Test it Now

Output:

```
default constructor is invoked
default constructor is invoked
111 Karan
222 Aryan
```

Rule: Call to this() must be the first statement in constructor.

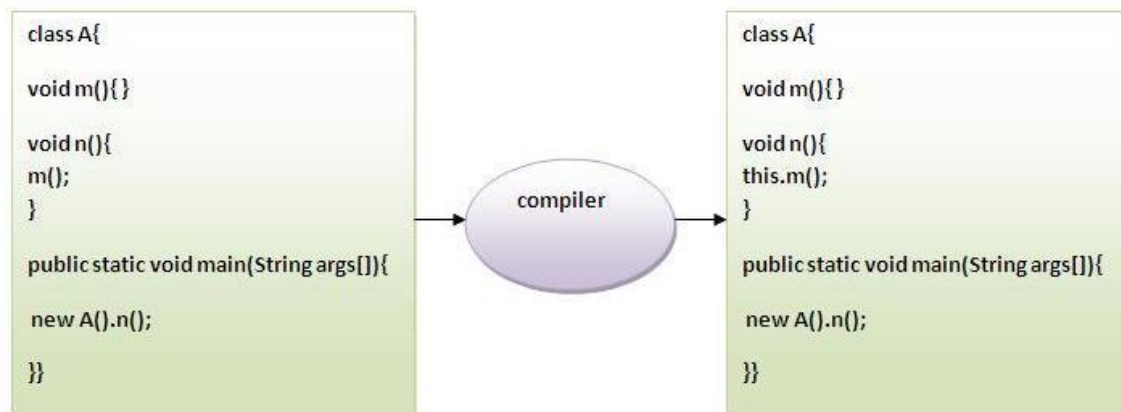
```
1.  class Student15{
2.      int id;
3.      String name;
4.      Student15(){System.out.println("default constructor is invoked");}
5.
6.      Student15(int id, String name){
7.          id = id;
8.          name = name;
9.          this ();//must be the first statement
10.     }
11.     void display(){System.out.println(id+ " "+name);}
12.
13.     public static void main(String args[]){
14.         Student15 e1 = new Student15(111,"karan");
15.         Student15 e2 = new Student15(222,"Aryan");
16.         e1.display();
17.         e2.display();
18.     }
19. }
```

Test it Now

Output:Compile Time Error

3)The this keyword can be used to invoke current class method (implicitly).

You may invoke the method of the current class by using the this keyword. If you don't use the this keyword, compiler automatically adds this keyword while invoking the method. Let's see the example



Proving this keyword

Let's prove that this keyword refers to the current class instance variable. In this program, we are printing the reference variable and this, output of both variables are same.

```
PrintThisKeyword.java
package oop;

public class PrintThisKeyword {

    public void printThis() {
        System.out.println(this); // prints same reference ID
    }

    public static void main(String[] args) {
        PrintThisKeyword obj = new PrintThisKeyword();
        obj.printThis();

        System.out.println(obj); // prints the reference ID
    }
}
```

Console

```
<terminated> PrintThisKeyword [Java Application] C:\Program Files\Java\jdk1.7.0_79\bin\javaw.exe (?
oop.PrintThisKeyword@6276e1db
oop.PrintThisKeyword@6276e1db
```

Inheritance in Java

Inheritance in java is a mechanism in which one object acquires all the properties and behaviours of parent object.

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 parent class, and you can add new methods and fields also.

Inheritance represents the **IS-A relationship**, also known as *parent-child* relationship.

Why use inheritance in java

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

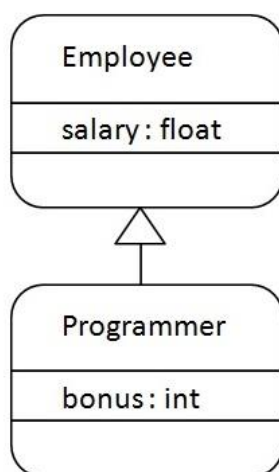
Syntax of Java Inheritance

```
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.

In the terminology of Java, a class that is inherited is called a super class. The new class is called a subclass.

Understanding the simple example of inheritance



	Class	Package	Subclass	World
public	y	y	y	y
protected	y	y	y	n
no modifier	y	y	n	n
private	y	n	n	n

y: accessible
n: not accessible

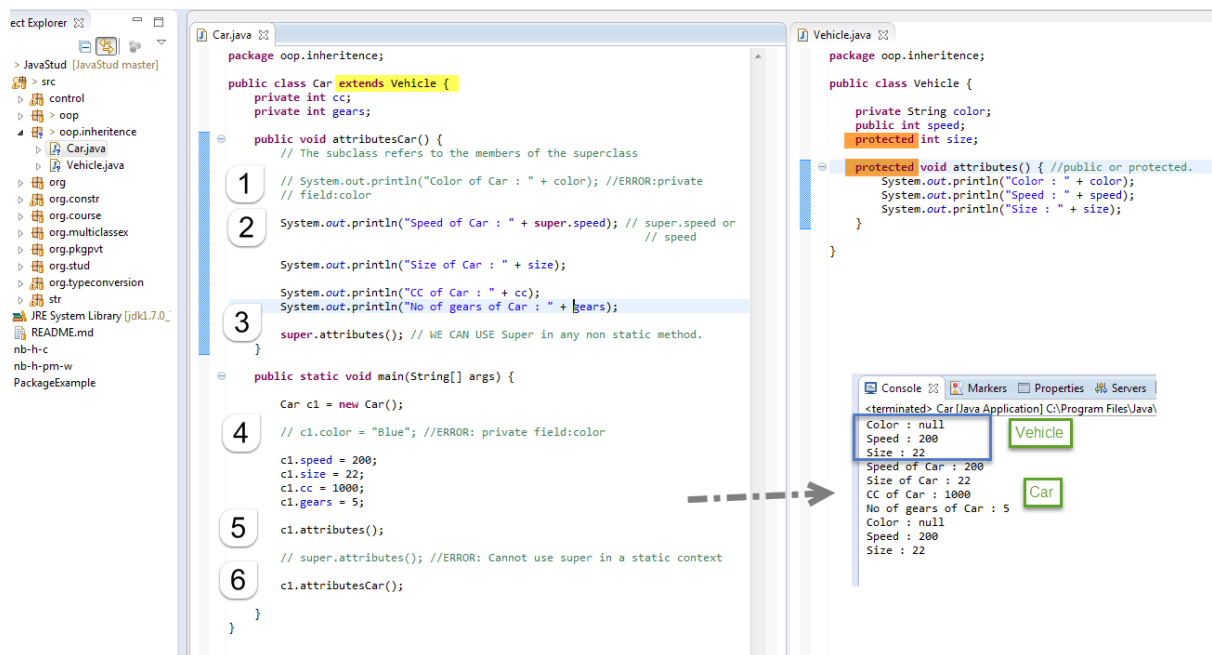
As displayed in the above figure, Programmer is the subclass and Employee is the superclass. Relationship between two classes is **Programmer IS-A Employee**. It means that Programmer is a type of Employee.

```
1. class Employee{
2.     float salary=40000;
3. }
4. class Programmer extends Employee{
5.     int bonus=10000;
6.     public static void main(String args[]){
7.         Programmer p=new Programmer();
8.         System.out.println("Programmer salary is:"+p.salary);
9.         System.out.println("Bonus of Programmer is:"+p.bonus);
10.    }
11. }
```

Test it Now

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

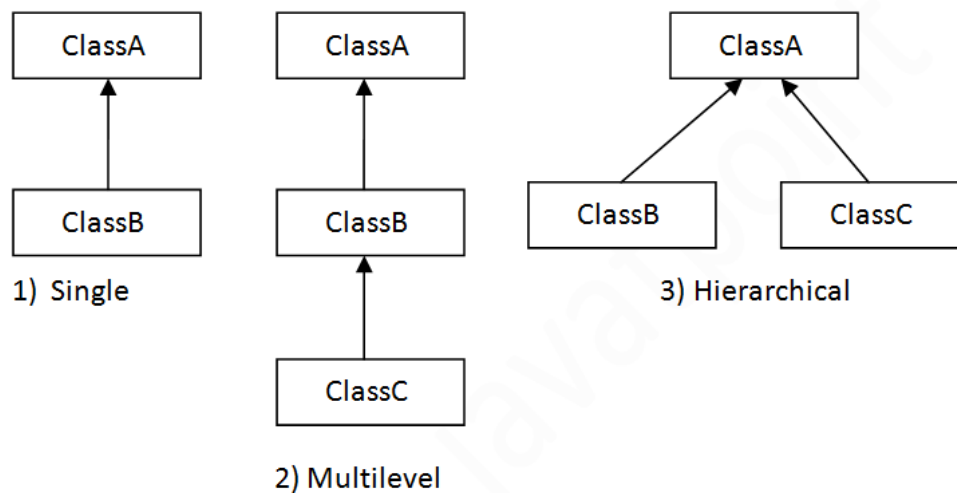
In the above example, Programmer object can access the field of own class as well as of Employee class i.e. code reusability.



Types of inheritance in java

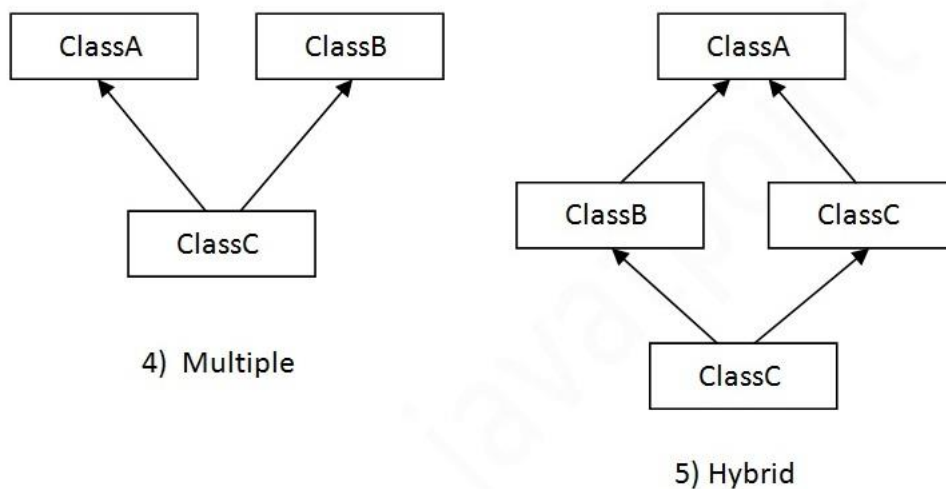
On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.

In java programming, multiple and hybrid inheritance is supported through interface only. We will learn about interfaces later.



Note: Multiple inheritance is not supported in java through class.

When a class extends multiple classes i.e. known as multiple inheritance. For Example:



Q) Why multiple inheritance is not supported in java?

To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

Consider a scenario where A, B and C are three classes. The C class inherits A and B classes. If A and B classes have same method and you call it from child class object, there will be ambiguity to call 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 now.

```
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. }
```

Test it Now

Compile Time Error

Method Overloading in Java

If a class have multiple methods by same name but different parameters, it is known as **Method Overloading**.

If we have to perform only one operation, having same name of the methods increases the readability of the program.

Suppose you have to perform addition of the given numbers but there can be any number of arguments, if you write the method such as a(int,int) for two parameters, and b(int,int,int) for three parameters then it may be difficult for you as well as other programmers to understand the behaviour of the method because its name differs. So, we perform method overloading to figure out the program quickly.

Advantage of method overloading?

Method overloading **increases the readability of the program**.

Different ways to overload the method

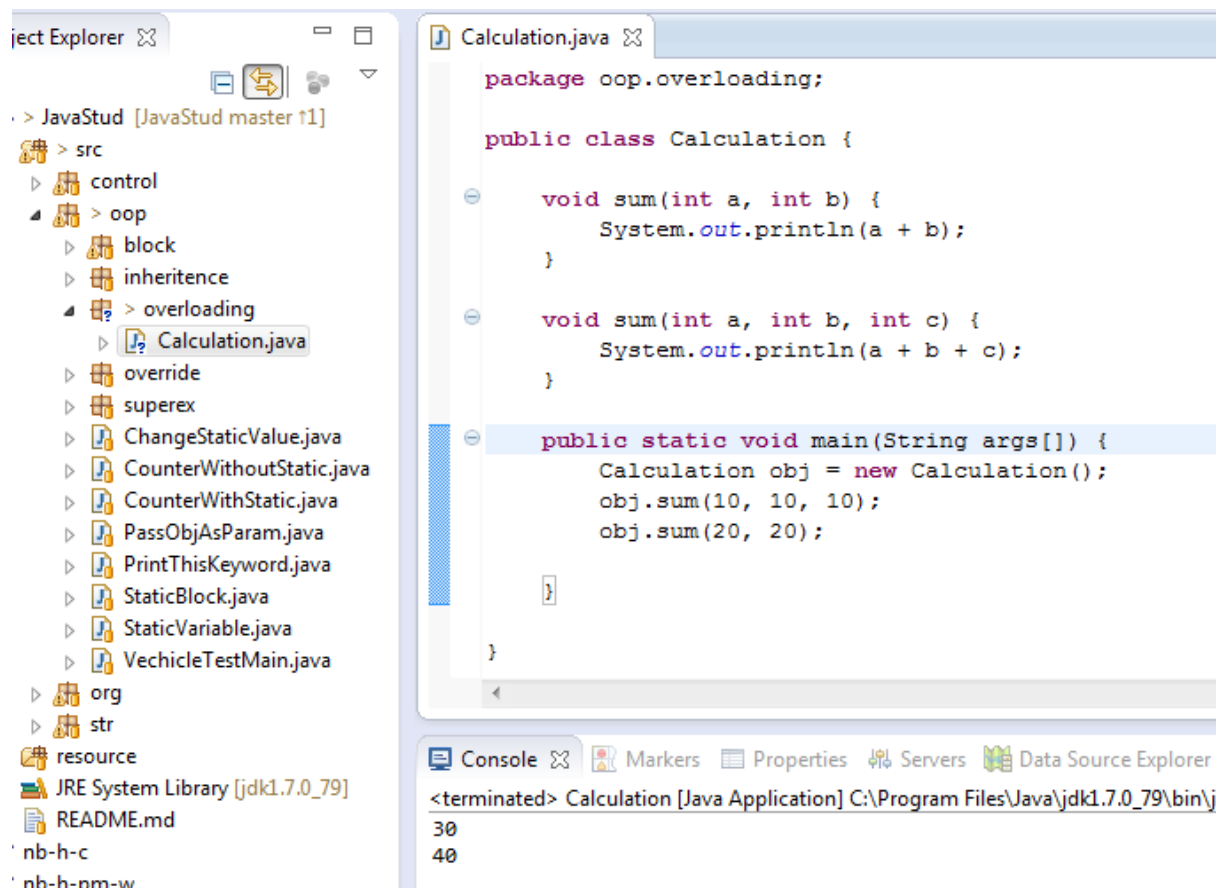
There are two ways to overload the method in java

1. By changing number of arguments
2. By changing the data type

In java, Method Overloading is not possible by changing the return type of the method.

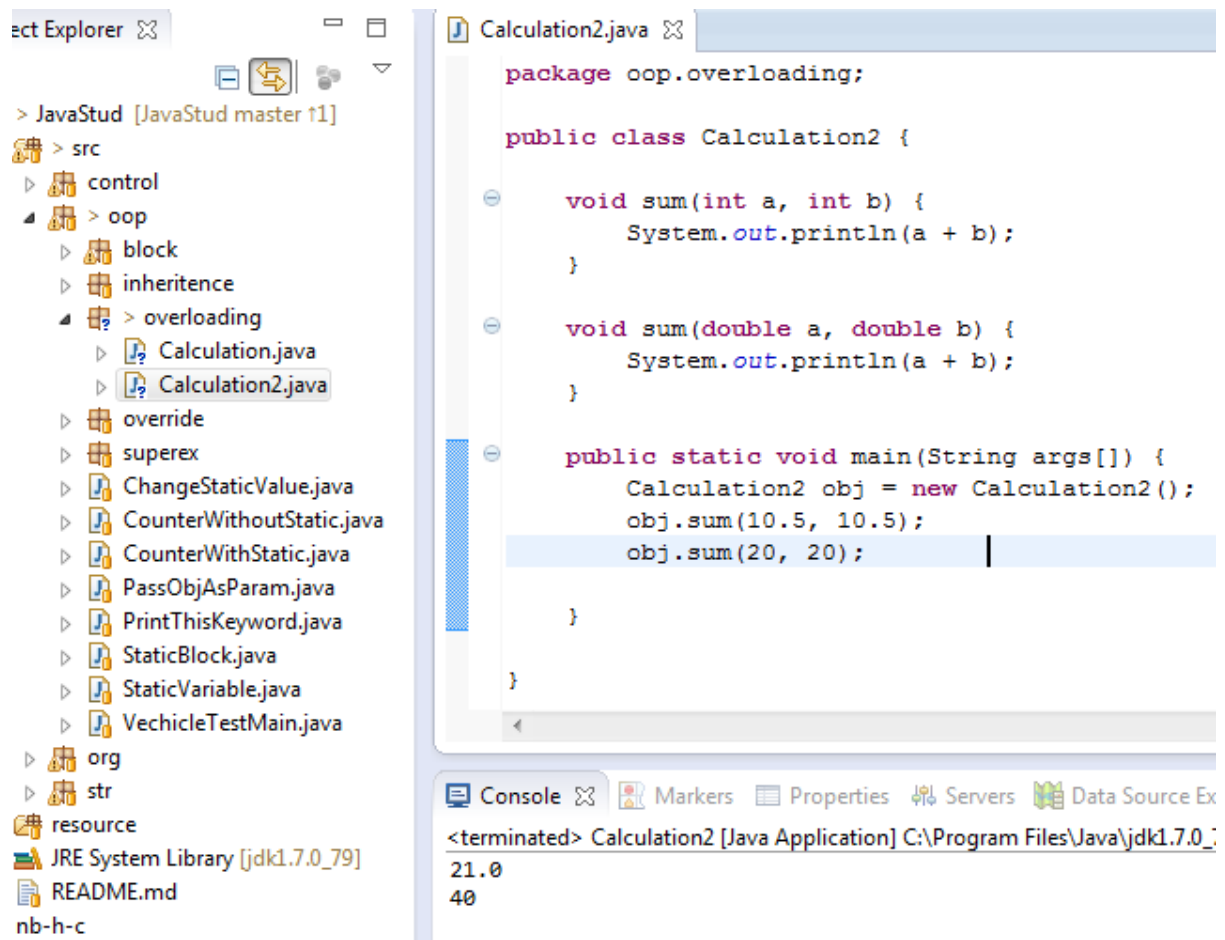
1) Example of Method Overloading by changing the no. of arguments

In this example, we have created two overloaded methods, first sum method performs addition of two numbers and second sum method performs addition of three numbers.



2) Example of Method Overloading by changing data type of argument

In this example, we have created two overloaded methods that differs in data type. The first sum method receives two integer arguments and second sum method receives two double arguments.



Q) Why Method Overloading is not possible by changing the return type of method?

In java, method overloading is not possible by changing the return type of the method because there may occur ambiguity. Let's see how ambiguity may occur:

because there was problem:

```

1.  class Calculation3{
2.      int sum(int a,int b){System.out.println(a+b);}
3.      double sum(int a,int b){System.out.println(a+b);}
4.
5.      public static void main(String args[]){
6.          Calculation3 obj=new Calculation3();
7.          int result=obj.sum(20,20); //Compile Time Error
8.
9.      }
10. }

```

Test it Now

int result=obj.sum(20,20); //Here how can java determine which sum() method should be called

Example of Method Overloading with TypePromotion

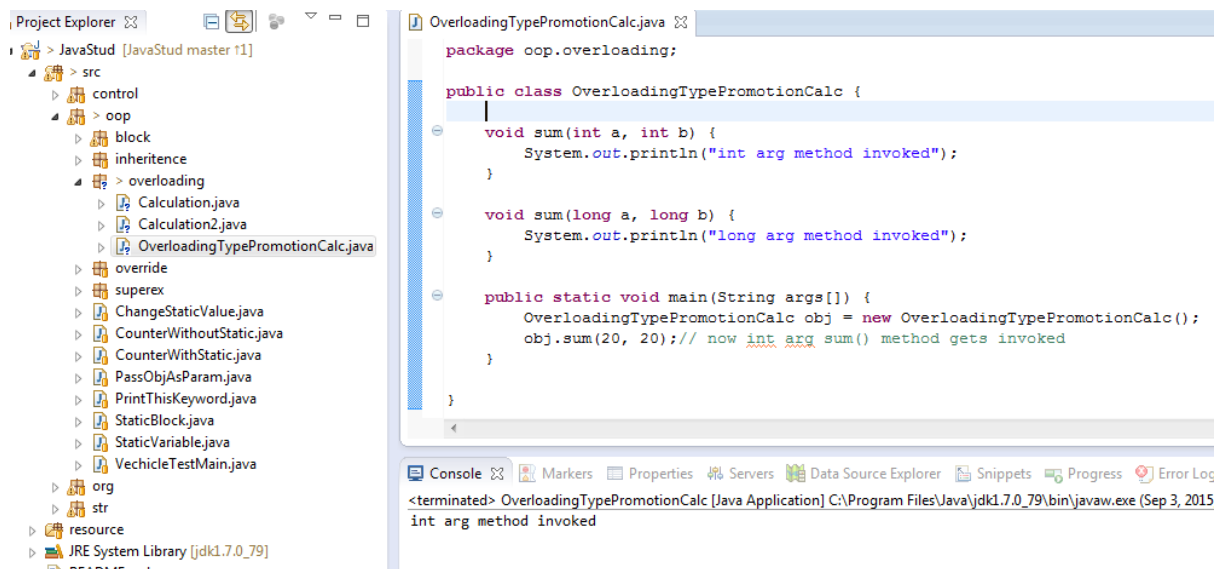
```
1. class OverloadingCalculation1{
2.     void sum(int a,long b){System.out.println(a+b);}
3.     void sum(int a,int b,int c){System.out.println(a+b+c);}
4.
5.     public static void main(String args[]){
6.         OverloadingCalculation1 obj=new OverloadingCalculation1();
7.         obj.sum(20,20);//now second int literal will be promoted to long
8.         obj.sum(20,20,20);
9.
10.    }
11. }
```

Test it Now

```
Output: 40
        60
```

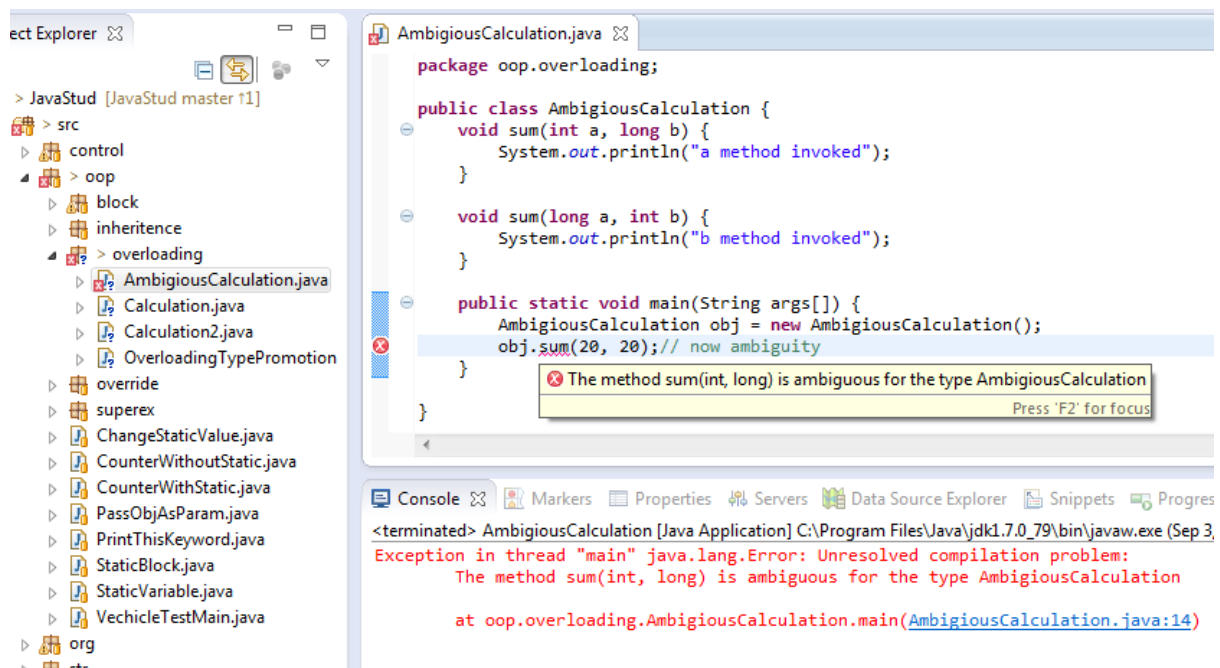
Example of Method Overloading with TypePromotion if matching found

If there are matching type arguments in the method, type promotion is not performed.



Example of Method Overloading with TypePromotion in case ambiguity

If there are no matching type arguments in the method, and each method promotes similar number of arguments, there will be ambiguity.



One type is not de-promoted implicitly for example double cannot be depromoted to any type implicitly.

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 subclass provides the specific implementation of the method that has been provided by one of its parent class, it is known as method overriding.

Usage of Java Method Overriding

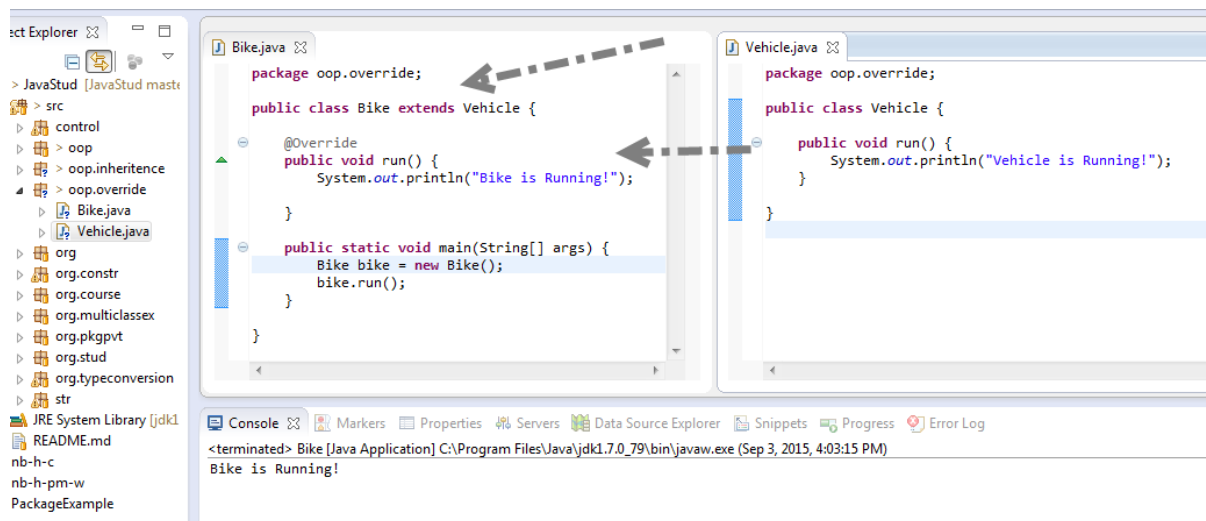
- Method overriding is used to provide specific implementation of a method that is already provided by its super class.
- Method overriding is used for runtime polymorphism

Rules for Java Method Overriding

1. method must have same name as in the parent class
2. method must have same parameter as in the parent class.
3. must be 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 is same and there is IS-A relationship between the classes, so there is method overriding.



Real example of Java Method Overriding



Can we override static method?

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

Why we cannot override static method?

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

Can we override java main method?

No, because main is a static method.

Polymorphism:

Polymorphism means more than one form, same object performing different operations according to the requirement.

Polymorphism can be achieved by using two ways, those are

1. Method overriding
2. Method overloading

Aggregation in Java

If a class have an entity reference, it is known as Aggregation. Aggregation represents HAS-A relationship.

Consider a situation, Employee object contains many informations such as id, name, emailId etc. It contains one more object named address, which contains its own informations such as city, state, country, zipcode etc. as given below.

```
1.  class Employee{
2.      int id;
3.      String name;
4.      Address address; //Address is a class
5.      ...
6.  }
```

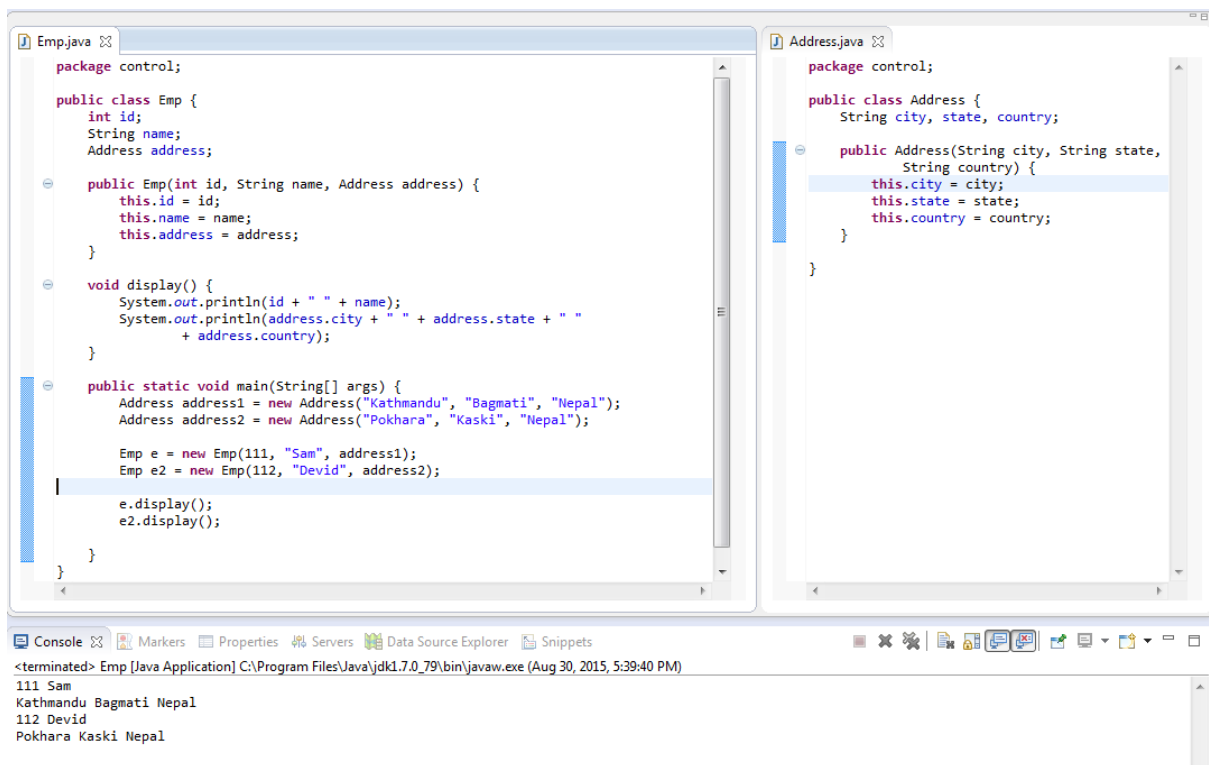
In such case, Employee has an entity reference address, so relationship is Employee HAS-A address.

Why use Aggregation?

- For Code Reusability.

Understanding meaningful example of Aggregation

In this example, Employee has an object of Address, address object contains its own informations such as city, state, country etc. In such case relationship is Employee HAS-A address.



super keyword in java

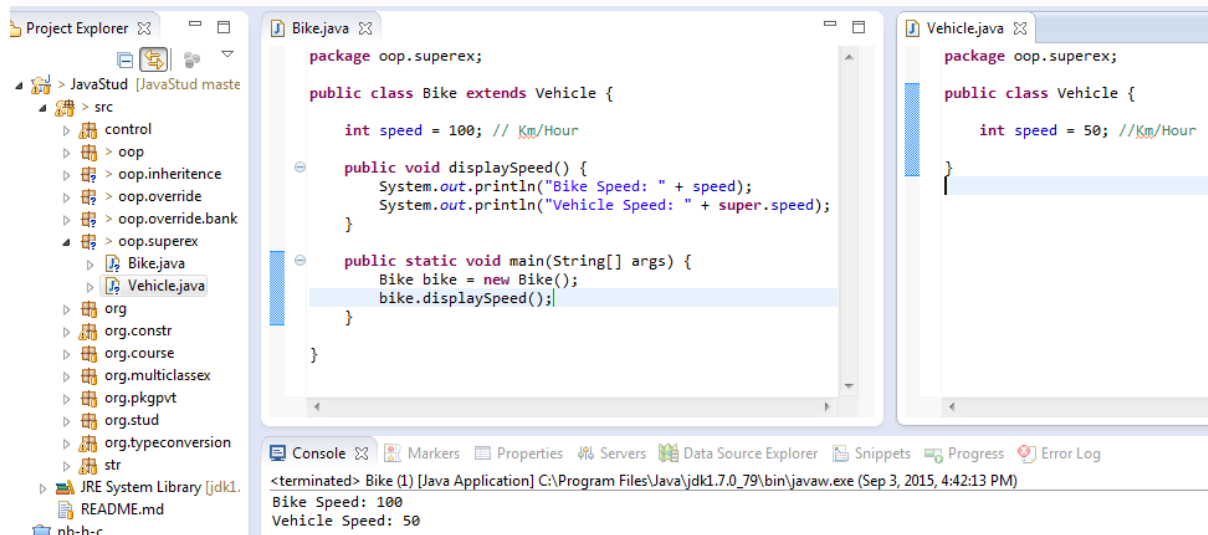
The **super** keyword in java is a reference variable that is used to refer immediate parent class object.

Whenever you create the instance of subclass, an instance of parent class is created implicitly i.e. referred by super reference variable.

Usage of java super Keyword

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

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



2) super is used to invoke parent class constructor.

The super keyword can also be used to invoke the parent class constructor as given below:

1. `class Vehicle{`
2. `Vehicle(){System.out.println("Vehicle is created");}`
3. `}`
- 4.
5. `class Bike5 extends Vehicle{`
6. `Bike5(){`
7. `super();//will invoke parent class constructor`
8. `System.out.println("Bike is created");`
9. `}`
10. `public static void main(String args[]){`
11. `Bike5 b=new Bike5();`


```
12.
13. }
14. }
```

Test it Now

```
Output:Vehicle is created
       Bike is created
```

3) super can be used to invoke parent class method

The super keyword can also be used to invoke parent class method. It should be used in case subclass contains the same method as parent class as in the example given below:

```
1.  class Person{
2.  void message(){System.out.println("welcome");}
3.  }
4.
5.  class Student16 extends Person{
6.  void message(){System.out.println("welcome to java");}
7.
8.  void display(){
9.  message();//will invoke current class message() method
10. super.message();//will invoke parent class message() method
11. }
12.
13. public static void main(String args[]){
14. Student16 s=new Student16();
15. s.display();
16. }
17. }
```

Test it Now

```
Output:welcome to java
       welcome
```

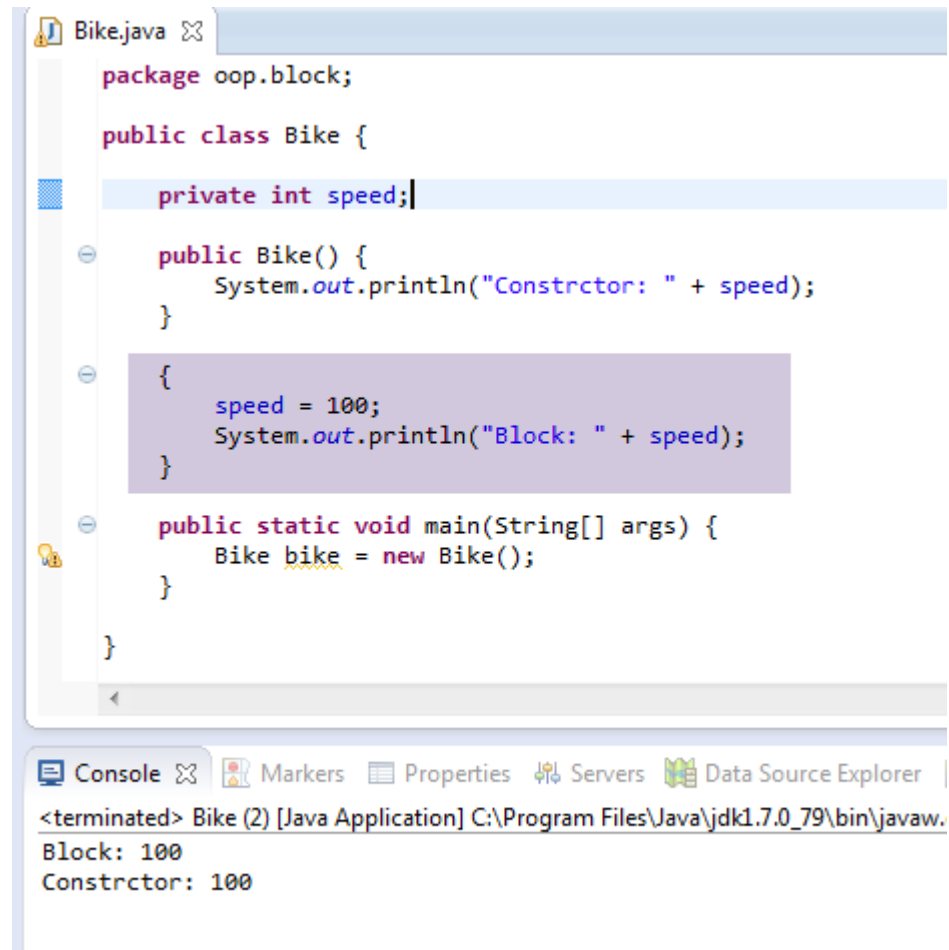
Program in case super is not required

```
1.  class Person{
2.  void message(){System.out.println("welcome");}
3.  }
4.
5.  class Student extends Person{
6.
7.  void display(){
8.  message();//will invoke parent class message() method
9.  }
10.
11. public static void main(String args[]){
12. Student s=new Student ();
13. s.display();
14. }
15. }
```

Test it Now

Output:welcome

Instance initializer block:



```
package oop.block;

public class Bike {

    private int speed;

    public Bike() {
        System.out.println("Constructor: " + speed);
    }

    {
        speed = 100;
        System.out.println("Block: " + speed);
    }

    public static void main(String[] args) {
        Bike bike = new Bike();
    }

}
```

Console Output:

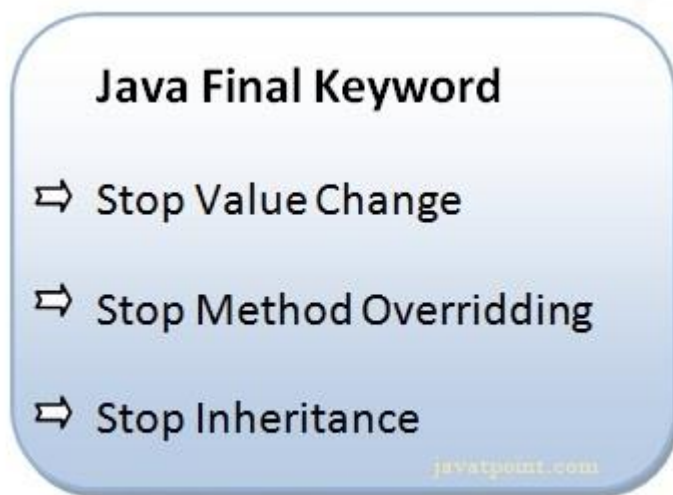
```
<terminated> Bike (2) [Java Application] C:\Program Files\Java\jdk1.7.0_79\bin\javaw.
Block: 100
Constructor: 100
```

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) Java final variable

If you make any variable as final, you cannot change the value of final variable(It will be constant).

Example of final variable

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

```
1. class Bike9{
2.     final int speedlimit=90;//final variable
3.     void run(){
4.         speedlimit=400;
5.     }
6.     public static void main(String args[]){
```

```

7.     Bike9 obj=new Bike9();
8.     obj.run();
9.     }
10.    }//end of class

```

[Test it Now](#)

Output:Compile Time Error

2) Java final method

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

Example of final method

```

1.  class Bike{
2.      final void run(){System.out.println("running");}
3.  }
4.
5.  class Honda extends Bike{
6.      void run(){System.out.println("running safely with 100kmph");}
7.
8.      public static void main(String args[]){
9.          Honda honda= new Honda();
10.         honda.run();
11.     }
12. }

```

[Test it Now](#)

Output:Compile Time Error

3) Java final class

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

Example of final class

```

1.      final class Bike{}
2.
3.      class Honda1 extends Bike{
4.          void run(){System.out.println("running safely with 100kmph");}
5.
6.          public static void main(String args[]){
7.              Honda1 honda= new Honda();
8.              honda.run();
9.          }
10.     }

```

[Test it Now](#)

Output:Compile Time Error

Q) Is final method inherited?

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

```
1.  class Bike{
2.      final void run(){System.out.println("running...");}
3.  }
4.  class Honda2 extends Bike{
5.      public static void main(String args[]){
6.          new Honda2().run();
7.      }
8.  }
```

Test it Now

Output:running...

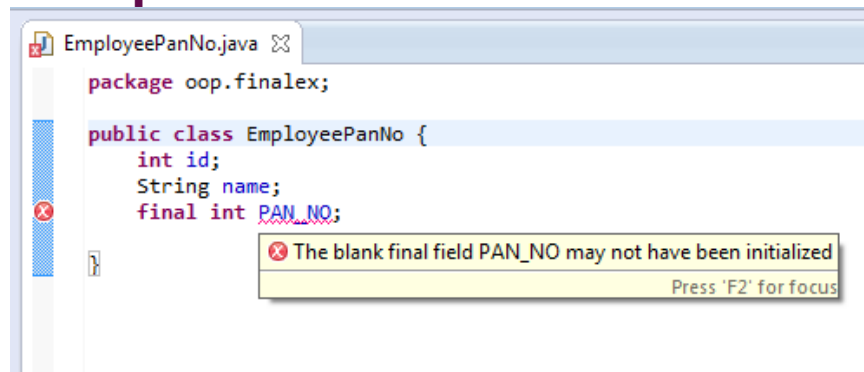
Q) What is blank or uninitialized final variable?

A final variable that is not initialized at the time of declaration is known as blank final variable.

If you want to create a variable that is initialized at the time of creating object and once initialized may not be changed, it is useful. For example PAN CARD number of an employee.

It can be initialized only in constructor.

Example of blank final variable



```
EmployeePanNo.java
package oop.finalex;

public class EmployeePanNo {
    int id;
    String name;
    final int PAN_NO;

    public EmployeePanNo() {
        PAN_NO = 87290;
    }
}
```

static blank final variable

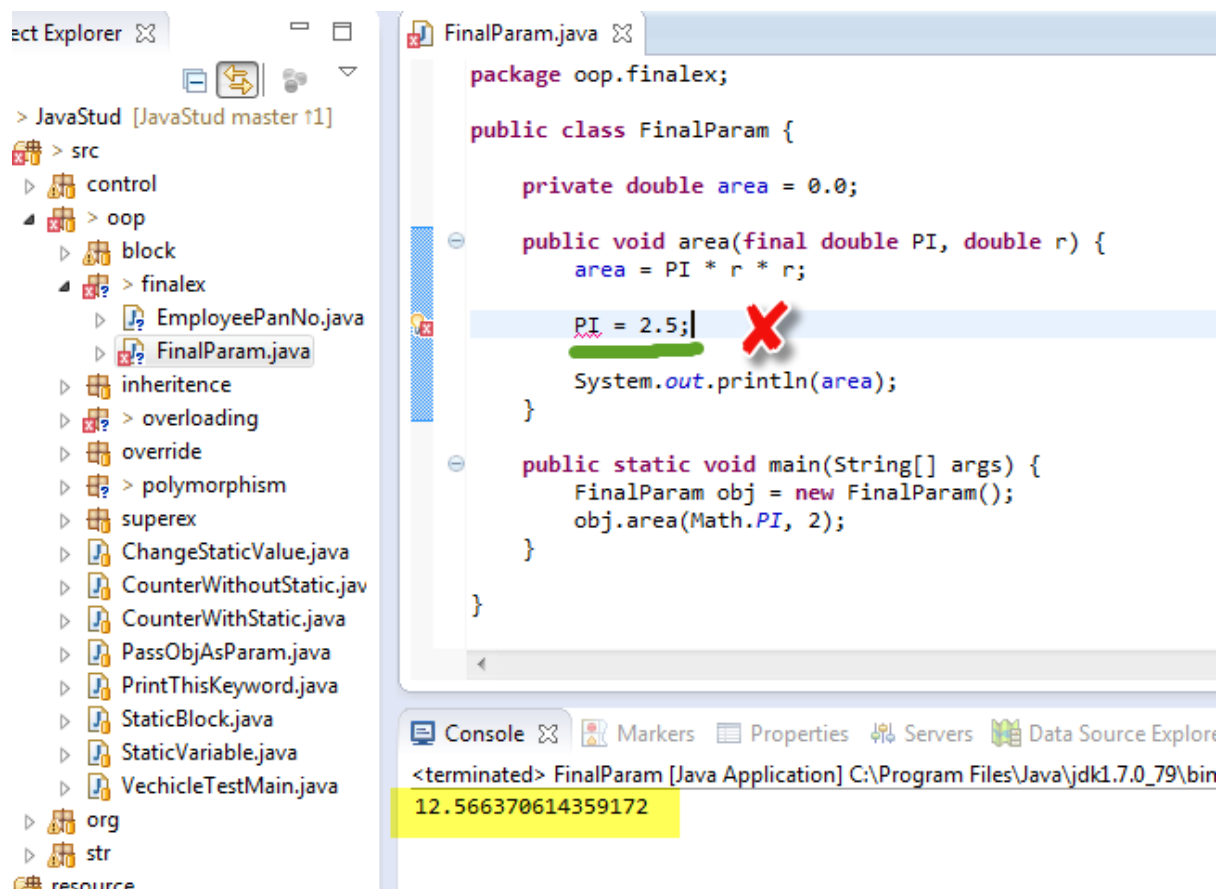
A static final variable that is not initialized at the time of declaration is known as static blank final variable. It can be initialized only in static block.

Example of static blank final variable

```
1. class A{
2.     static final int data;//static blank final variable
3.     static{ data=50;}
4.     public static void main(String args[]){
5.         System.out.println(A.data);
6.     }
7. }
```

Q) What is final parameter?

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



Q) 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*.

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

Let's first understand the upcasting before Runtime Polymorphism.

Upcasting

When reference variable of Parent class refers to the object of Child class, it is known as upcasting. For example:

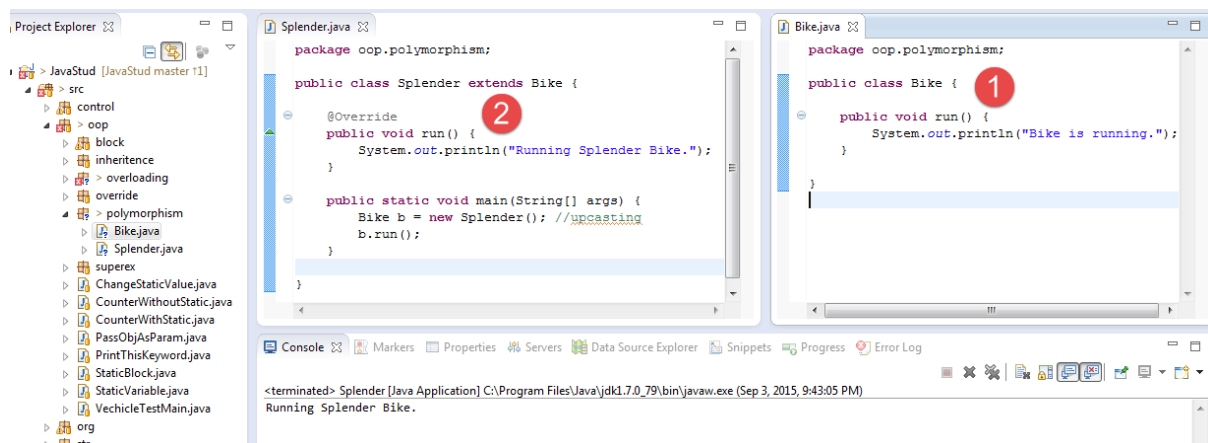


1. `class A{}`
2. `class B extends A{}`
1. `A a=new B();//upcasting`

Example of Java Runtime Polymorphism

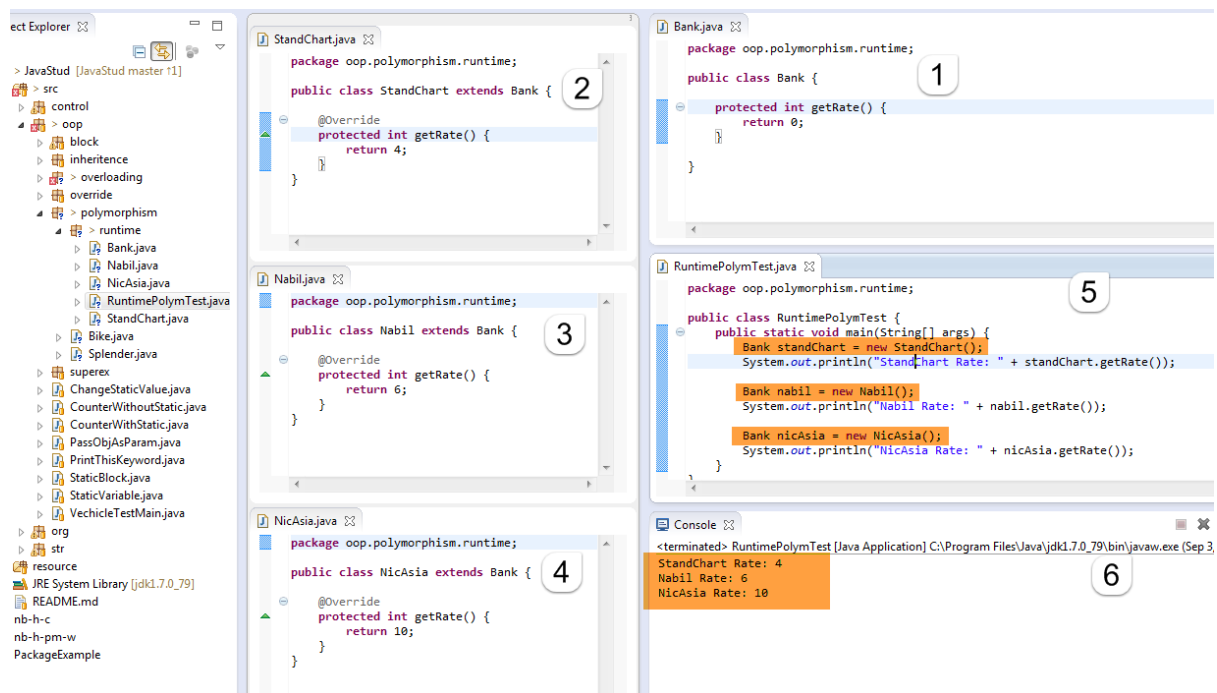
In this example, we are creating two classes Bike and Splendar. Splendar class extends Bike class and overrides its run() method. We are calling the run method by the reference variable of Parent class. Since it refers to the subclass object and subclass method overrides the Parent class method, subclass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

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



Real example of Java Runtime Polymorphism

Note: It is also given in method overriding but there was no upcasting.



Java Runtime Polymorphism with data member

Method is overridden not the datamembers, so runtime polymorphism can't be achieved by data members.

In the example given below, both the classes have a datamember speedlimit, we are accessing the datamember by the reference variable of Parent class which refers to the subclass object. Since we are accessing the datamember which is not overridden, hence it will access the datamember of Parent class always.

Rule: Runtime polymorphism can't be achieved by data members.

```

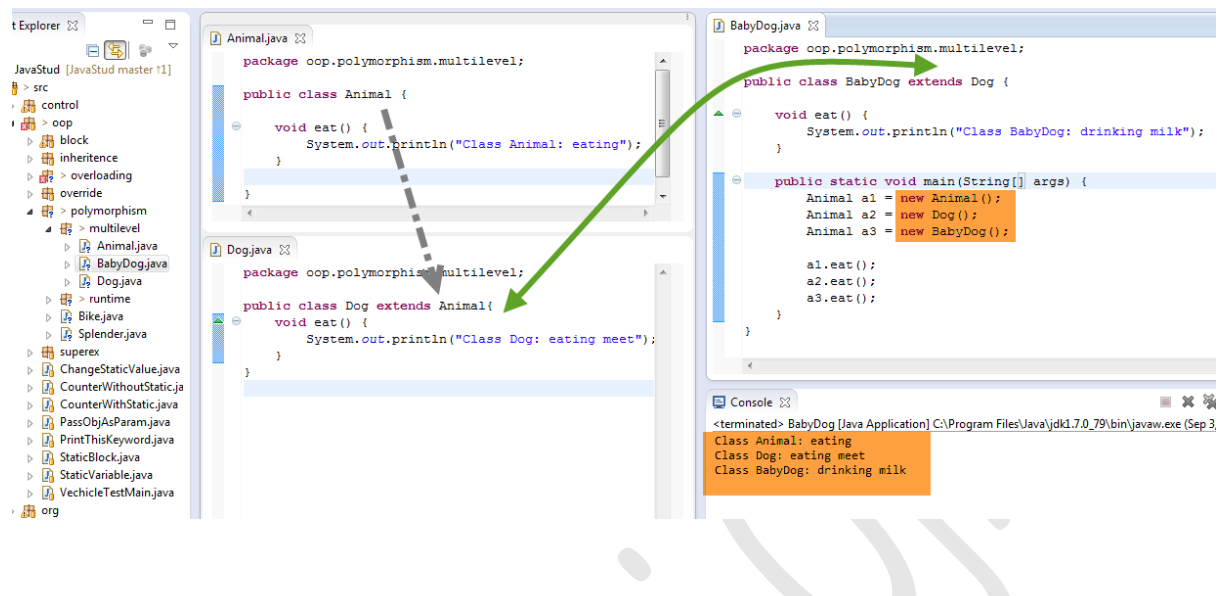
1. class Bike{
2.     int speedlimit=90;
3. }
4. class Honda3 extends Bike{
5.     int speedlimit=150;
6. }
7. public static void main(String args[]){
8.     Bike obj=new Honda3();
9.     System.out.println(obj.speedlimit);//90
10. }

```

Test it Now

Output: 90

Java Runtime Polymorphism with Multilevel Inheritance



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).

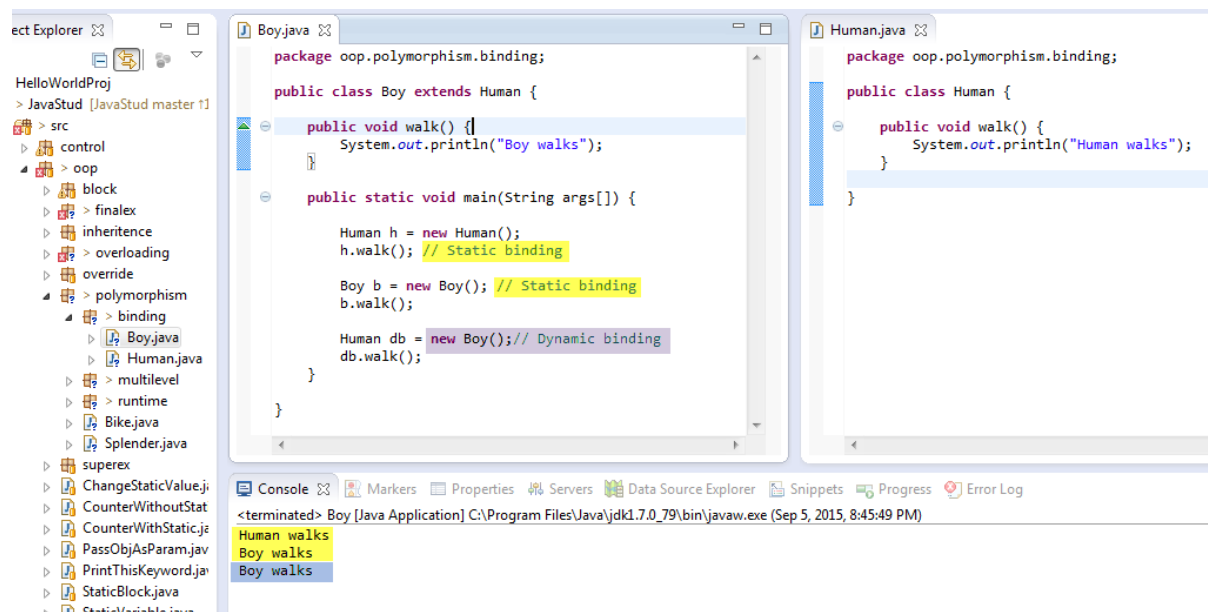
Static binding

When type of the object is determined at compiled time (by the compiler), it is known as static binding.

If there is any private, final or static method in a class, there is static binding.

Dynamic binding

When type of the object is determined at run-time, it is known as dynamic binding.



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 email, you just type the text and send the email. You don't know the internal processing about the email delivery.

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

Ways to achieve Abstraction

There are two ways to achieve abstraction in java

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

Abstract class in Java

A class that is declared as abstract is known as **abstract class**.

- If a class is declared abstract it cannot be instantiated.
- **Abstract classes cannot be instantiated, but they can be subclassed.** It may or may not include abstract methods.
- **When an abstract class is subclassed, the subclass usually provides implementations for all of the abstract methods in its parent class. However, if it does not, then the subclass must also be declared abstract.**
- But, if a class have at least one abstract method, then the class **must** be declared abstract.

Example abstract class

```
1. abstract class Bank {}
```

abstract method

A method that is declared as abstract and does not have implementation is known as abstract method.

Example abstract method

```
1. abstract void getRate ();//no body and abstract
```

Understanding the real scenario of abstract class

The screenshot illustrates a real-world scenario of using an abstract class. It shows the following code:

1. Bank.java (Abstract Class)

```
package oop.abst;

public abstract class Bank {

    public abstract String getBankName( );

    protected abstract int getRate( );

    public int serviceChargeRate = 4;

    protected double getDollarExchangeRate( ) {
        // Write your logic to fetch dollar exchange rate.
        // For example we use constant
        return 101.5;
    }
}
```

2. Nabil.java (Subclass)

```
package oop.abst;

public class Nabil extends Bank {

    @Override
    public String getBankName( ) {
        return "Nabil";
    }

    @Override
    protected int getRate( ) {
        return 6;
    }

    // You can write other variable and method here.
}
```

3. NicAsia.java (Subclass)

```
package oop.abst;

public class NicAsia extends Bank {

    @Override
    public String getBankName( ) {
        return "NIC Asia";
    }

    @Override
    protected int getRate( ) {
        return 10;
    }

    // You can write other variable and method here.
}
```

4. StandChart.java (Subclass)

```
package oop.abst;

public class StandChart extends Bank {

    @Override
    public String getBankName( ) {
        return "Standard Chartered";
    }

    @Override
    protected int getRate( ) {
        return 4;
    }

    // You can write other variable and method here.
}
```

AbstractClassTest.java (Test Class)

```
package oop.abst;

public class AbstractClassTest {

    public static void main( String[] args ) {

        Bank n = new Nabil( ); //OR This is also right: Nabil n = new Nabil( );
        printBankInfo( n.getBankName( ), n.getRate( ), n.getDollarExchangeRate( ), n.serviceChargeRate );

        NicAsia na = new NicAsia( );
        printBankInfo( na.getBankName( ), na.getRate( ), na.getDollarExchangeRate( ), na.serviceChargeRate );

        StandChart sc = new StandChart( );
        printBankInfo( sc.getBankName( ), sc.getRate( ), sc.getDollarExchangeRate( ), sc.serviceChargeRate );

    }

    public static void printBankInfo( String bankName, double rate, double dollarExcRate, int serviceChargeRate ) {
        System.out.println( "Bank Name: " + bankName + "\tInterest Rate: " + rate +
            "\tDollar Exchange Rate: " + dollarExcRate + "\tService Charge Rate: " + serviceChargeRate );
    }
}
```

Console Output

```
<terminated> AbstractClassTest [Java Application] C:\Program Files\Java\jdk1.7.0_79\bin\javaw.exe (Sep 5, 2015, 10:27:11 PM)
Bank Name: Nabil      Interest Rate: 6.0      Dollar Exchange Rate: 101.5      Service Charge Rate: 4
Bank Name: NIC Asia  Interest Rate: 10.0     Dollar Exchange Rate: 101.5     Service Charge Rate: 4
Bank Name: Standard Chartered Interest Rate: 4.0     Dollar Exchange Rate: 101.5     Service Charge Rate: 4
```

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

1. **class** Bike{
2. **abstract void** run();
3. }

Test it Now

compile time error

Rule: If you are extending any abstract class that have abstract method, you must either provide the implementation of the method or make this class abstract.

Interface in Java

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

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

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

It cannot be instantiated just like abstract class.

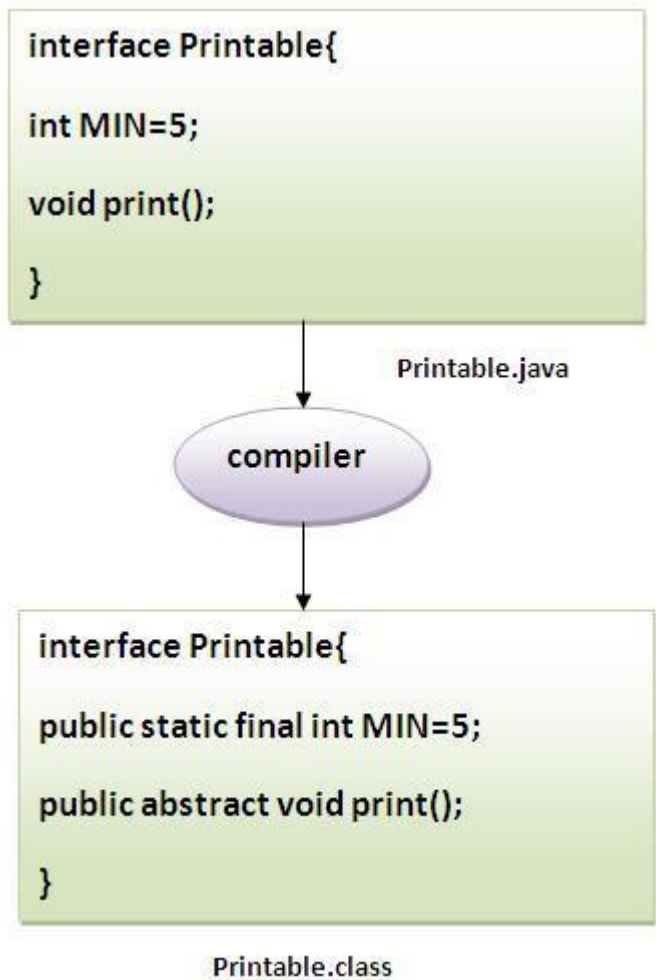
Why use Java interface?

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

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

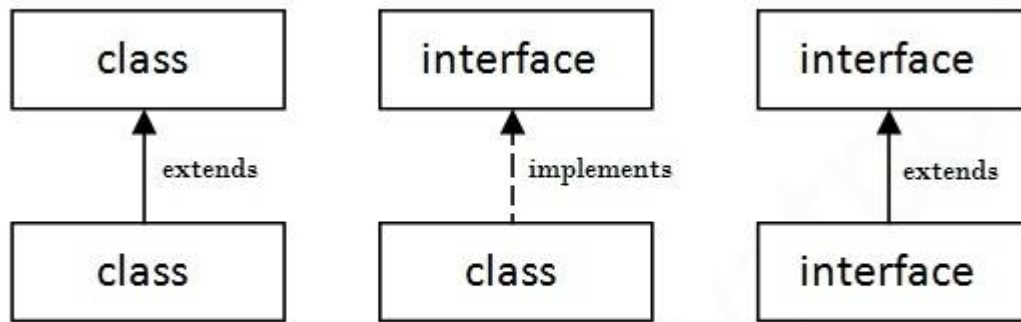
The java compiler adds public and abstract keywords before the interface method and public, static and final keywords before data members.

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



Understanding relationship between classes and interfaces

As shown in the figure given below, a class extends another class, an interface extends another interface but a **class implements an interface**.



Simple example of Java interface

In this example, Printable interface have only one method, its implementation is provided in the A class.

```
1.  interface Printable{
2.  void print();
3.  }
4.
5.  class PrintableImpl implements Printable{
6.  public void print(){System.out.println("Hello");}
7.
8.  public static void main(String args[]){
9.  PrintableImpl obj = new PrintableImpl();
10. obj.print();
11. }
12. }
```

Test it Now

Output:Hello

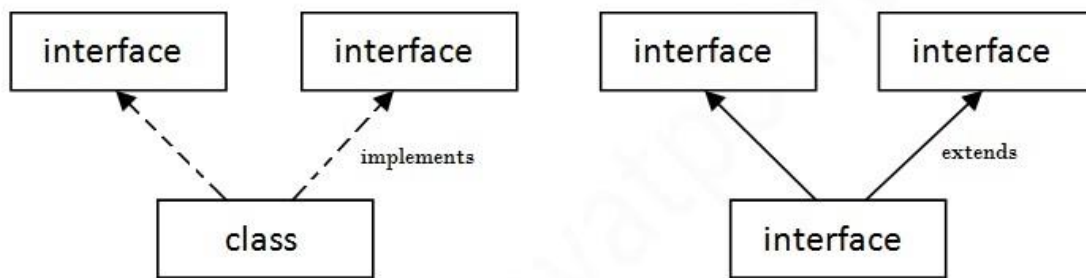
Real World Example:

The screenshot displays an IDE with the following components:

- Project Explorer:** Shows a project named 'JavaStud' with a package 'oop' containing interfaces and classes.
- Bank.java (1):** Defines the `Bank` interface with methods `getBankName()` and `getRate()`, and a constant `serviceChargeRate`.
- Nabil.java (2):** Implements `Bank` with `getBankName()` returning "Nabil" and `getRate()` returning 6.
- NicAsia.java (3):** Implements `Bank` with `getBankName()` returning "NIC Asia" and `getRate()` returning 10.
- StandChart.java (4):** Implements `Bank` with `getBankName()` returning "Standard Chartered" and `getRate()` returning 4.
- InterfaceImplTest.java:** A test class with a `main` method that creates instances of `Nabil`, `NicAsia`, and `StandChart`, and prints their details. It also includes a static method `printBankInfo` that takes bank details and prints them.
- Console:** Shows the output of the program, displaying the bank name, interest rate, and service charge rate for each bank.

Multiple inheritance in Java by interface

If a class implements multiple interfaces, or an interface extends multiple interfaces i.e. known as multiple inheritance.



Multiple Inheritance in Java

```

1. interface Printable{
2. void print();
3. }
4.
5. interface Showable{
6. void show();
7. }
8.
9. class Impl implements Printable,Showable{
10.
11. public void print(){System.out.println("Print Hello");}
12. public void show(){System.out.println("Show Welcome");}
13.
14. public static void main(String args[]){
15. Impl obj = new Impl ();
16. obj.print();
17. obj.show();
18. }
19. }

```

Test it Now

```

Output:Print Hello
        Show Welcome

```

Q) Multiple inheritance is not supported through class in java but it is possible by interface, why?

As we have explained in the inheritance chapter, multiple inheritance is not supported in case of class. But it is supported in case of interface because there is no ambiguity as implementation is provided by the implementation class. For example:

```

1. interface Printable{
2. void print();
3. }
4.
5. interface Showable{
6. void print();
7. }
8.

```

```

9.    class testinterface1 implements Printable,Showable{
10.
11.    public void print(){System.out.println("Hello");}
12.
13.    public static void main(String args[]){
14.    testinterface1 obj = new testinterface1();
15.    obj.print();
16.    }
17.    }

```

Test it Now

Hello

As you can see in the above example, Printable and Showable interface have same methods but its implementation is provided by class A, so there is no ambiguity.

Interface inheritance

A class implements interface but one interface extends another interface .

```

1.    interface Printable{
2.    void print();
3.    }
4.    interface Showable extends Printable{
5.    void show();
6.    }
7.    class Testinterface2 implements Showable{
8.
9.    public void print(){System.out.println("Hello");}
10.   public void show(){System.out.println("Welcome");}
11.
12.   public static void main(String args[]){
13.   Testinterface2 obj = new Testinterface2();
14.   obj.print();
15.   obj.show();
16.   }
17.   }

```

Test it Now

Hello

Welcome

Q) What is marker or tagged interface?

An interface that have no member is known as marker or tagged interface. For example: Serializable, Cloneable, Remote etc. They are used to provide some essential information to the JVM so that JVM may perform some useful operation.

1. `//How Serializable interface is written?`
2. `public interface Serializable{`
3. `}`

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.
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 have static methods, main method and constructor.	Interface can't have static methods, main method or constructor.
5) Abstract class can provide the implementation of interface.	Interface can't provide the implementation of abstract class.
6) The abstract keyword is used to declare abstract class.	The interface keyword is used to declare interface.
7) Example: <pre>public abstract class Shape{ public abstract void draw(); }</pre>	Example: <pre>public interface Drawable{ void draw(); }</pre>

Simply, abstract class achieves partial abstraction (0 to 100%) whereas interface achieves fully abstraction (100%).

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

Methods of Object class

The Object class provides many methods. They are as follows:

Method	Description
public final Class getClass()	returns the Class class object of this object. The Class class can further be used to get the metadata of this class.
public int hashCode()	returns the hashcode number for this object.
public boolean equals(Object obj)	compares the given object to this object.
protected Object clone() throws CloneNotSupportedException	creates and returns the exact copy (clone) of this object.
public String toString()	returns the string representation of this object.
public final void notify()	wakes up single thread, waiting on this object's monitor.
public final void notifyAll()	wakes up all the threads, waiting on this object's monitor.
public final void wait(long timeout) throws InterruptedException	causes the current thread to wait for the specified milliseconds, until another thread notifies (invokes notify() or notifyAll() method).
public final void wait(long timeout, int nanos) throws InterruptedException	causes the current thread to wait for the specified milliseconds and nanoseconds, until another thread notifies (invokes notify() or notifyAll() method).
public final void wait() throws InterruptedException	causes the current thread to wait, until another thread notifies (invokes notify() or notifyAll() method).
protected void finalize() throws Throwable	is invoked by the garbage collector before object is being garbage collected.