

Class, Object, Encapsulation, Inheritance & Polymorphism

OOPs Concepts

Object Oriented Programming (OOP) is a programming concept used in several modern programming languages, like C++, Java and Python. Before Object Oriented Programming programs were viewed as procedures that accepted data and produced an output. There was little emphasis given on the data that went into those programs. Object Oriented Programming works on the principle that objects are the most important part of your program. Manipulating these objects to get results is the goal of Object Oriented Programming.

OOP revolves around objects. You create the objects that you need and then create methods to handle those objects. These methods don't influence the state of the objects, but it's the other way around. Object Oriented Programming also addresses several other weaknesses of the other programming techniques.

A class (a blueprint that forms an object) can, for example, "inherit" the traits of another class, unlike modules. This means that your new class can do what your old class could, and more. There can be several given instances of a class at one time in a program, too.

A Programming Object

When you're creating an object oriented program, your perspective changes so that you view the world as a collection of objects. A flower is a good example of an object. The length of its stem, the color, its fragrance and its design are the properties or attributes of the flower object. These attributes form the data in our program. The values that these attributes take (the blue color of the petals, for example) form the state of the object. What is the task of a flower? You could say it pollinates to produce new flowers. This is a method of the flower object.

An object is both the data and the function(s), or method, which operates on the data. An object can also be defined as an instance of a class, and there can be more than one instance of an object in a program.

Class, Object, Encapsulation, Inheritance & Polymorphism

How do you create an object in Java? Like this, using the new keyword:

```
Flower rose = new Flower;
```

This will create an object called rose in your program.

List of OOP Concepts in Java

There are four main OOP concepts in Java. These are:

Class

When you want to create an object, you create its class first. A class will define the attributes as well as the methods of an object. A class can be used to create similar objects. You can think of a class as a blueprint of an object. If you had a class called “cars”, for example, it could have objects like Mercedes, BMW and Porsche. The properties defined by the class could be speed or the price of these cars, while the methods could be the tasks you could perform with these cars: race, for example.

How would you define a class in Java? Take a look at the example below:

```
public class Flower {  
    String name;  
    String color;  
    void pollination() {  
    }  
}
```

Here, we created a class called Flower, with attributes name and color and a method called pollination.

Class, Object, Encapsulation, Inheritance & Polymorphism

Abstraction:

Abstraction means using simple things to represent complexity. We all know how to turn the TV on, but we don't need to know how it works in order to enjoy it. In Java, abstraction means simple things like objects, classes, and variables represent more complex underlying code and data. This is important because it lets avoid repeating the same work multiple times.

Encapsulation:

This is the practice of keeping fields within a class private and providing access to them via public methods. It's a protective barrier that keeps the data and code safe within the class itself. This way, we can reuse objects like code components or variables without allowing open access to the data system-wide.

Inheritance

This is a special feature of Object Oriented Programming in Java. It lets programmers create new classes that share some of the attributes of existing classes. This lets us build on previous work without reinventing the wheel.

Polymorphism:

This Java OOP concept lets programmers use the same word to mean different things in different contexts. One form of polymorphism in Java is method overloading. That's when different meanings are implied by the code itself. The other form is method overriding. That's when the different meanings are implied by the values of the supplied variables. See more on this below.

How OOP Concepts in Java Work

OOP, concepts in Java work by letting programmers create components that can be reused in different ways, but still maintain security.

How Abstraction Works

Class, Object, Encapsulation, Inheritance & Polymorphism

Abstraction as an OOP concept in Java works by letting programmers create useful, reusable tools. For example, a programmer can create several different types of objects. These can be variables, functions, or data structures. Programmers can also create different classes of objects. These are ways to define the objects.

For instance, a class of variable might be an address. The class might specify that each address object shall have a name, street, city, and zip code. The objects, in this case, might be employee addresses, customer addresses, or supplier addresses.

How Encapsulation Works

Encapsulation lets us re-use functionality without jeopardizing security. It's a powerful OOP concept in Java because it helps us save a lot of time. For example, we may create a piece of code that calls specific data from a database. It may be useful to reuse that code with other databases or processes. Encapsulation lets us do that while keeping our original data private. It also lets us alter our original code without breaking it for others who have adopted it in the meantime.

How Inheritance Works

Inheritance is another labor-saving Java OOP concept. It works by letting a new class adopt the properties of another. We call the inheriting class a subclass or a child class. The original class is often called the parent. We use the keyword `extends` to define a new class that inherits properties from an old class.

How Polymorphism Works

Polymorphism in Java works by using a reference to a parent class to affect an object in the child class. We might create a class called "horse" by extending the "animal" class. That class might also implement the "professional racing" class. The "horse" class is "polymorphic," since it inherits attributes of both the "animal" and "professional racing" class.

Class, Object, Encapsulation, Inheritance & Polymorphism

Two more examples of polymorphism in Java are method overriding and method overloading.

In method overriding, the child class can use the OOP polymorphism concept to override a method of its parent class. That allows a programmer to use one method in different ways depending on whether it's invoked by an object of the parent class or an object of the child class.

In method overloading, a single method may perform different functions depending on the context in which it's called. That is, a single method name might work in different ways depending on what arguments are passed to it.

Class and Objects

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 implemented 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 a 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

```
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  
    }  
}
```

instance of a class, you will often see the two words object and instance used interchangeably. A class is declared by using the class keyword.

Class, Object, Encapsulation, Inheritance & Polymorphism

```
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.

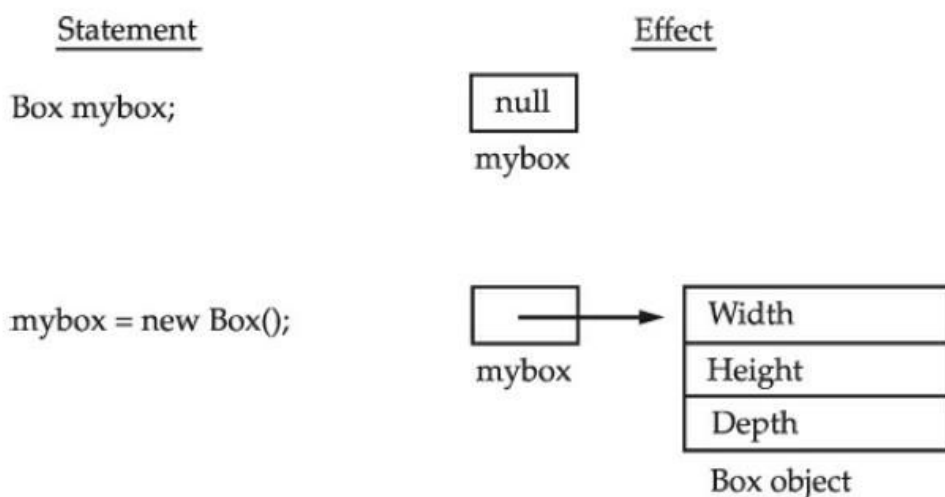
To assign the width variable of mybox the value 100

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* ();



Class, Object, Encapsulation, Inheritance & Polymorphism

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.

Class, Object, Encapsulation, Inheritance & Polymorphism

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.

Block in Java

A block statement is a sequence of zero or more statements enclosed in braces. A block statement is generally used to group together several statements, so they can be used in a situation that requires you to use a single statement

An example of a block statement is given below.

```
{ //block start
    int var = 20;
    var++;
} //block end
```

Please note that all the variables declared in a block statement can only be used within that block.

Class, Object, Encapsulation, Inheritance & Polymorphism

Methods

A method is a block of code which only runs when it is called.

You can pass data, known as parameters, into a method.

Methods are used to perform certain actions, and they are also known as functions.

Why use methods? To reuse code: define the code once, and use it many times.

Create a Method

A method must be declared within a class. It is defined with the name of the method, followed by parentheses (). Java provides some predefined methods, such as `System.out.println()`, but you can also create your own methods to perform certain actions:

Example

```
public class MyClass {  
    static void myMethod() {  
        // code to be executed  
    }  
}
```

Example Explained

myMethod() is the name of the method

static means that the method belongs to the `MyClass` class and not an object of the `MyClass` class.

void means that this method does not have a return value.

Example

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

Class, Object, Encapsulation, Inheritance & Polymorphism

}

Example Explained

a and b are two parameters of methods which pass value to the method

return means it return value of c

Call a Method

To call a method in Java, write the method's name followed by two parentheses () and a semicolon;

In the following example, myMethod() is used to print a text (the action), when it is called:

Example

```
public class MyClass {  
    static void myMethod() {  
        System.out.println("I just got executed!");  
    }  
    public static void main(String[] args) {  
        myMethod();  
    }  
}
```

Outputs:

I just got executed!

A method can also be called multiple times:

```
public class MyClass {  
    static void myMethod() {  
        System.out.println("I just got executed!");  
    }  
    public static void main(String[] args) {  
        myMethod();  
    }  
}
```

Class, Object, Encapsulation, Inheritance & Polymorphism

```
    myMethod();  
    myMethod();  
}  
}
```

Outputs:

I just got executed!

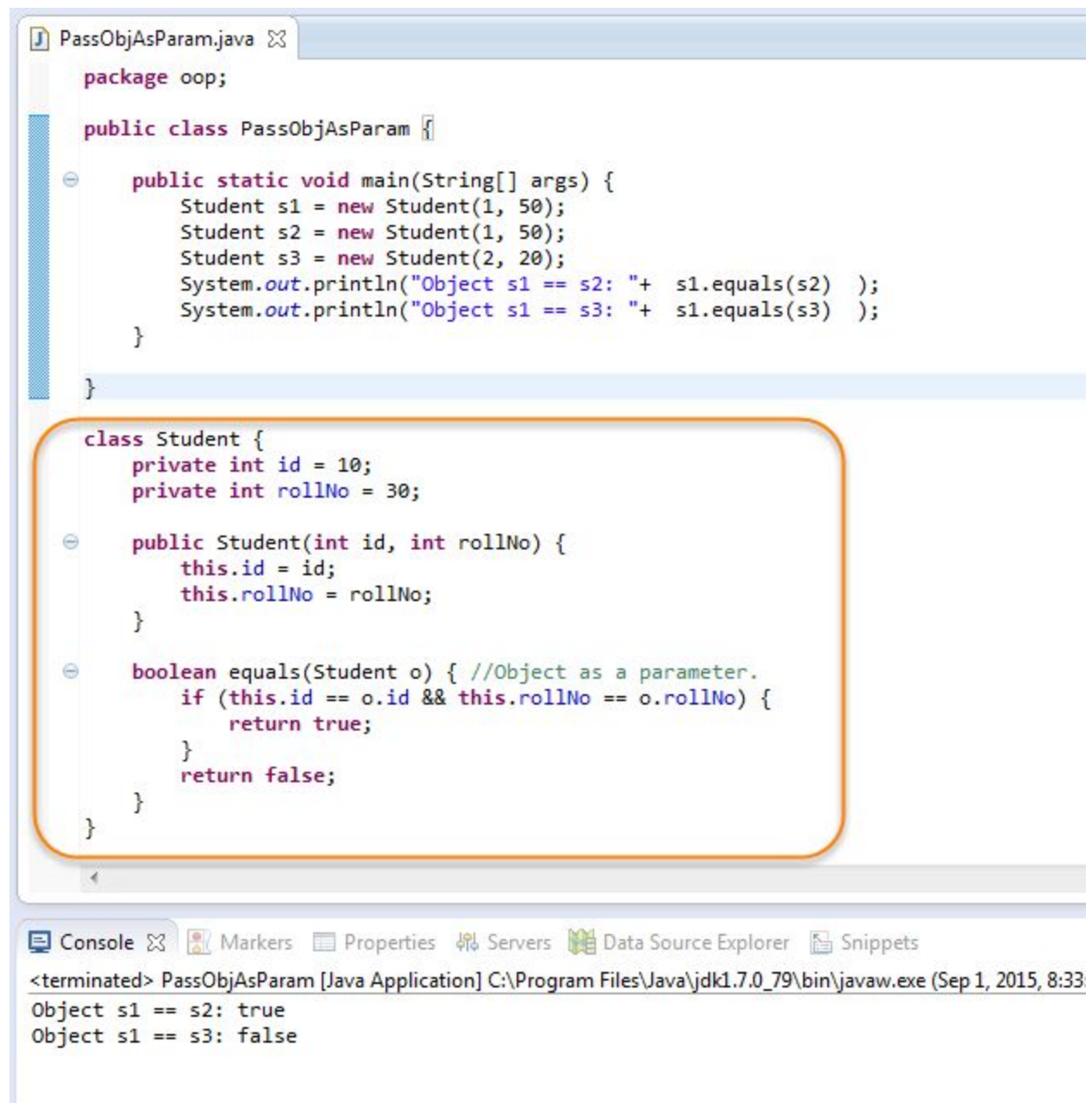
I just got executed!

I just got executed!

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:

Class, Object, Encapsulation, Inheritance & Polymorphism



The screenshot shows an IDE with a Java file named `PassObjAsParam.java`. The code defines a `Student` class with private attributes `id` and `rollNo`, a constructor, and an `equals` method. The `main` method in `PassObjAsParam` creates three `Student` objects: `s1` (id=1, rollNo=50), `s2` (id=1, rollNo=50), and `s3` (id=2, rollNo=20). It then prints the results of `s1.equals(s2)` and `s1.equals(s3)`.

```
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;
    }

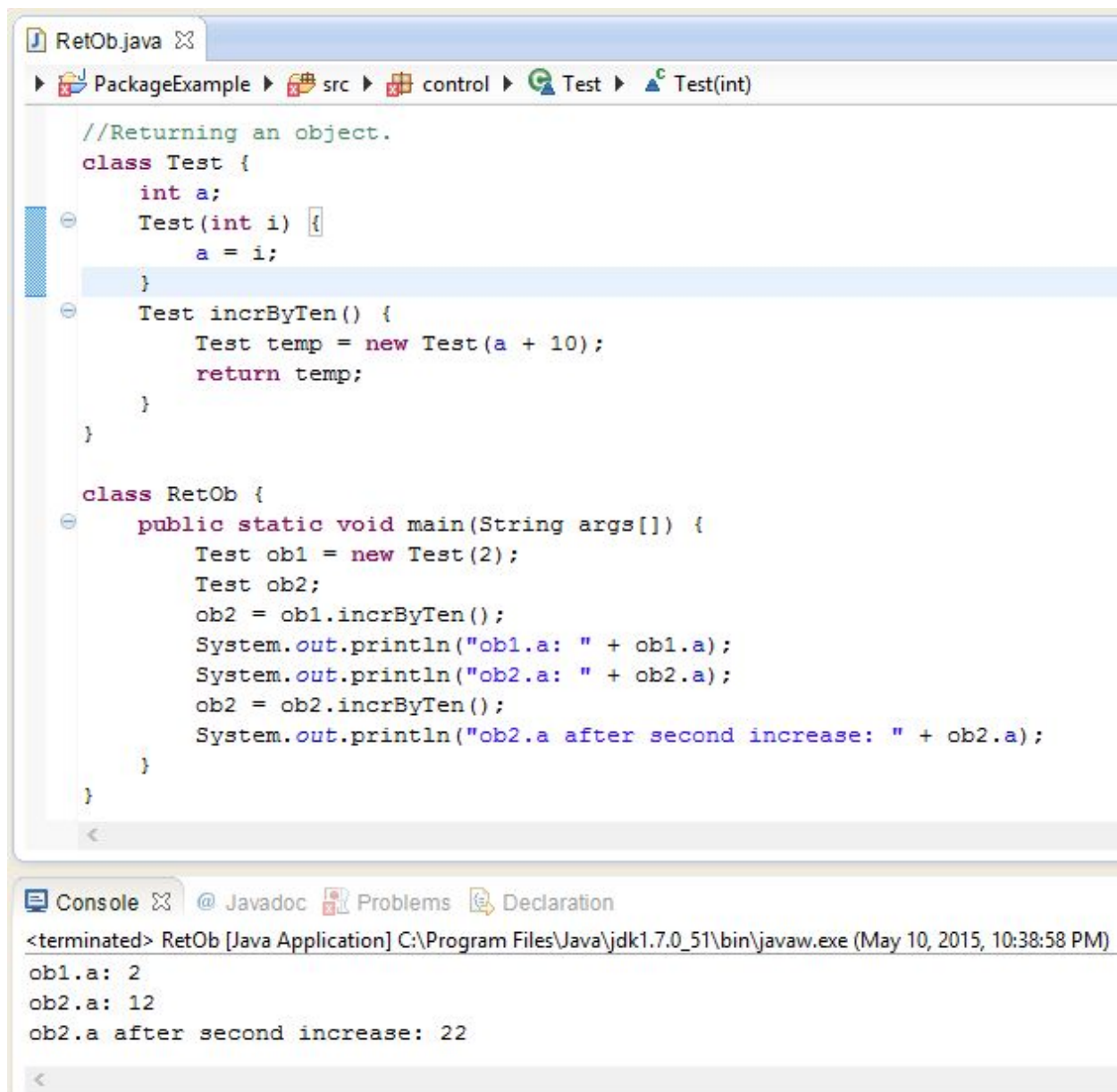
    boolean equals(Student o) { //Object as a parameter.
        if (this.id == o.id && this.rollNo == o.rollNo) {
            return true;
        }
        return false;
    }
}
```

The console output shows the execution results:

```
<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
```

Class, Object, Encapsulation, Inheritance & Polymorphism

Returning Objects



```
RetOb.java
PackageExample ▸ src ▸ control ▸ Test ▸ Test(int)

//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);
    }
}
```

Console | Javadoc | Problems | Declaration

<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.

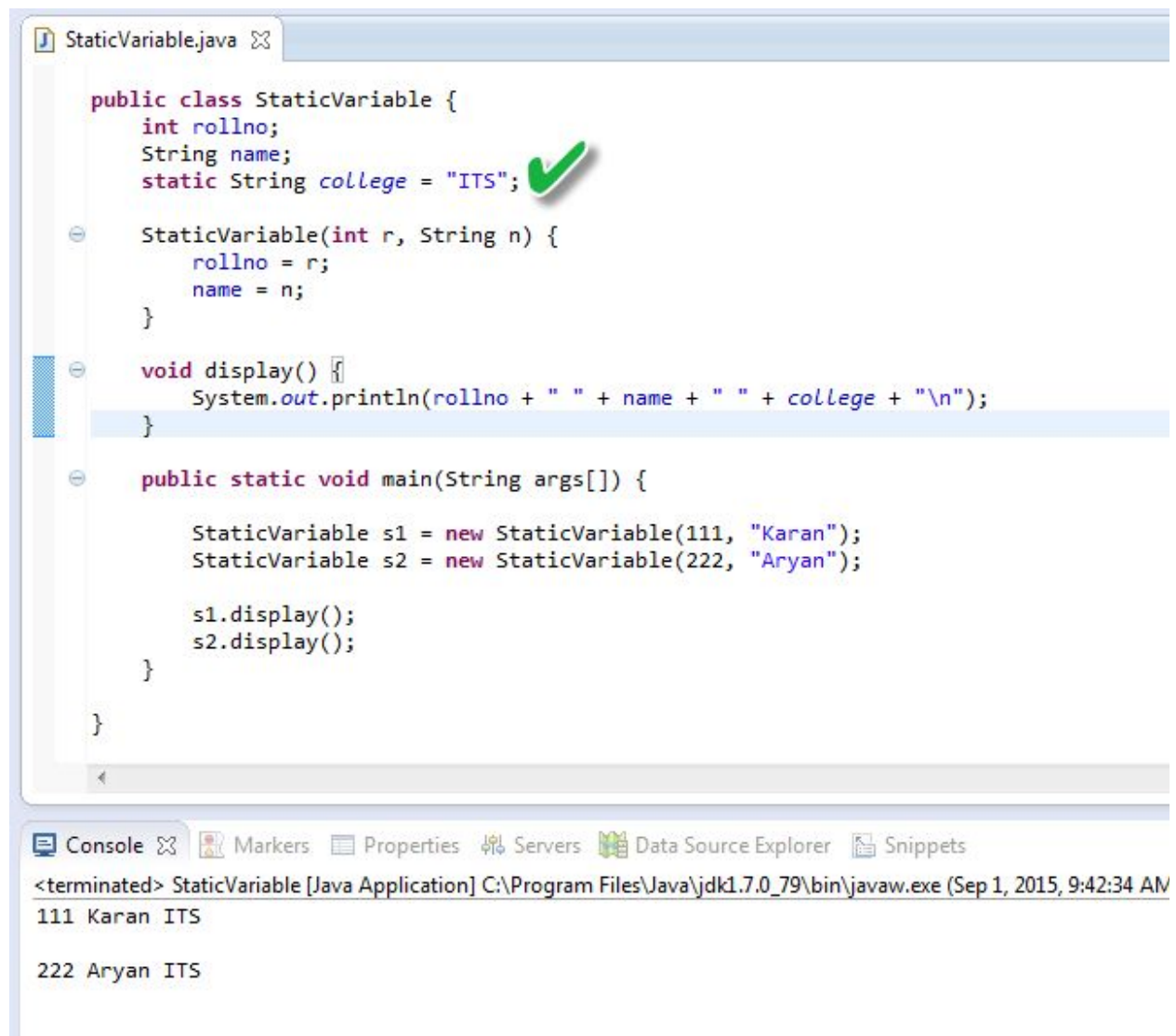
Java static keyword

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 an 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 that static variable. The static variable can be used to refer to 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.

Class, Object, Encapsulation, Inheritance & Polymorphism



```
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

<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

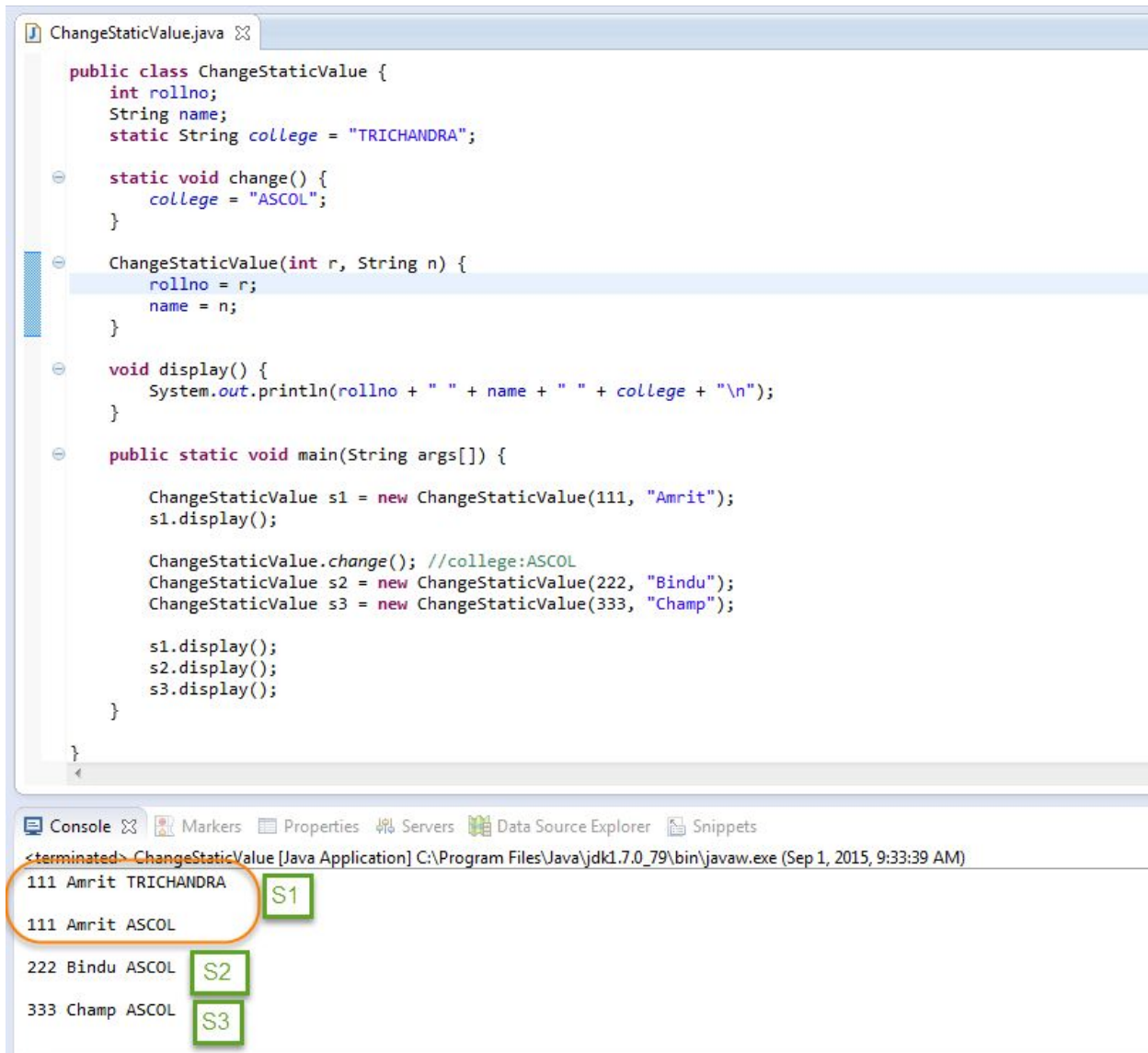
2) Java static method

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

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

Example of static method

Class, Object, Encapsulation, Inheritance & Polymorphism



The screenshot shows an IDE with a Java file named `ChangeStaticValue.java`. The code defines a class `ChangeStaticValue` with a static variable `college` and a static method `change()` to modify it. The `main` method creates three objects, calls `change()` to update the static variable, and then displays the state of each object. The console output shows the state before and after the static change, with labels S1, S2, and S3 identifying the objects.

```
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 Output:

```
<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
111 Amrit ASCOL
222 Bindu ASCOL
333 Champ ASCOL
```

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.

Class, Object, Encapsulation, Inheritance & Polymorphism

```
class A{  
    int a=40;//non static  
    public static void main(String args[]){  
        System.out.println(a);  
    }  
}
```

Output:

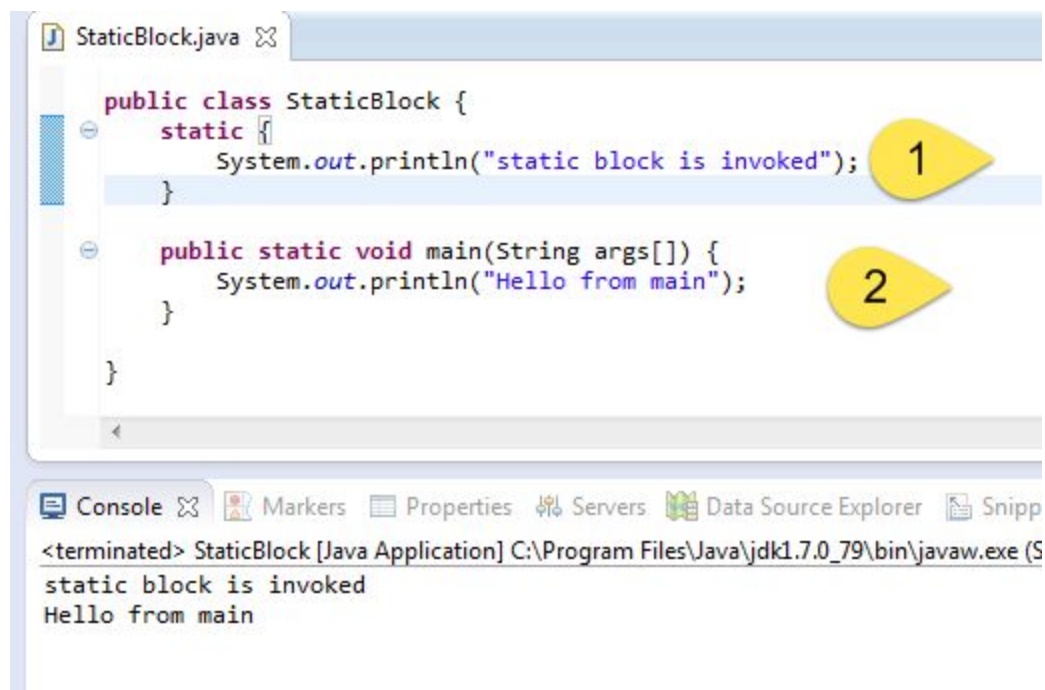
Compile Time Error

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.



Class, Object, Encapsulation, Inheritance & Polymorphism

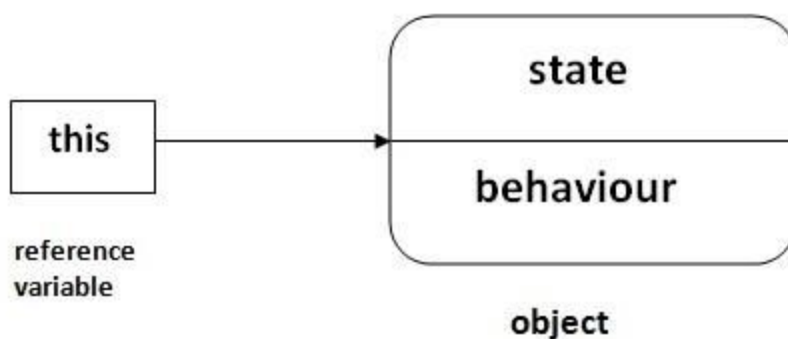
this keyword in java

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 3 usage of java this keyword.

1. this keyword can be used to refer to the 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)



The this keyword can be used to refer to the 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:

```
class Student10{
    int id;
    String name;
    Student10(int id, String name){
        id = id;
        name = name;
    }
    void display(){
```

Class, Object, Encapsulation, Inheritance & Polymorphism

```
        System.out.println(id+" "+name);
    }
    public static void main(String args[]){
        Student10 s1 = new Student10(111,"Karan");
        Student10 s2 = new Student10(321,"Aryan");
        s1.display();
        s2.display();
    }
}
```

Output:

0 null

0 null

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

```
class Student10{
    int id;
    String name;
    Student10(int id, String name){
        this.id = id;
        this.name = name;
    }
    void display(){
        System.out.println(id+" "+name);
    }
    public static void main(String args[]){
        Student10 s1 = new Student10(111,"Karan");
    }
}
```

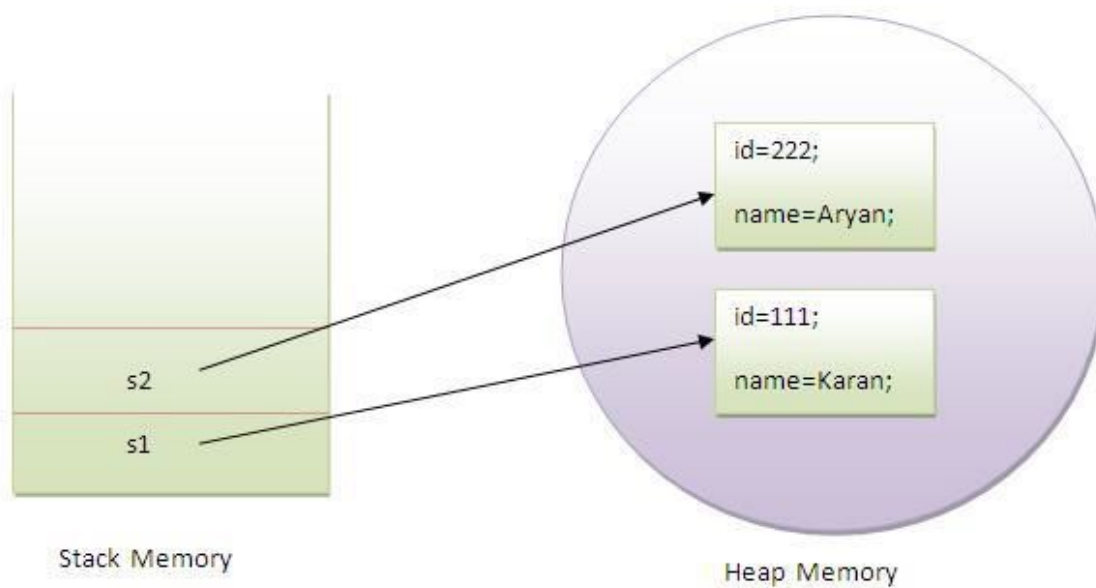
Class, Object, Encapsulation, Inheritance & Polymorphism

```
Student10 s2 = new Student10(321,"Aryan");  
s1.display();  
s2.display();  
}  
}
```

Output:

111 Karan

321 Aryan



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

```
class Student10{  
    int id;  
    String name;
```

Class, Object, Encapsulation, Inheritance & Polymorphism

```
Student10(int i, String n){
    id = i;
    name = n;
}
void display(){
    System.out.println(id+" "+name);
}
public static void main(String args[]){
    Student10 s1 = new Student10(111,"Karan");
    Student10 s2 = new Student10(321,"Aryan");
    s1.display();
    s2.display();
}
}
```

Output:

111 Karan

321 Aryan

this() can be used to invoke 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.

```
class Student13{
    int id;
    String name;
    Student13(){
        System.out.println("default constructor is invoked");
    }
}
```

Class, Object, Encapsulation, Inheritance & Polymorphism

```
Student13(int id, String name){
    this ();//it is used to invoked current class constructor.
    this.id = id;
    this.name = name;
}
void display(){
    System.out.println(id+" "+name);
}
public static void main(String args[]){
    Student13 e1 = new Student13(111,"karan");
    Student13 e2 = new Student13(222,"Aryan");
    e1.display();
    e2.display();
}
}
```

Output:

default constructor is invoked

default constructor is invoked

111 Karan

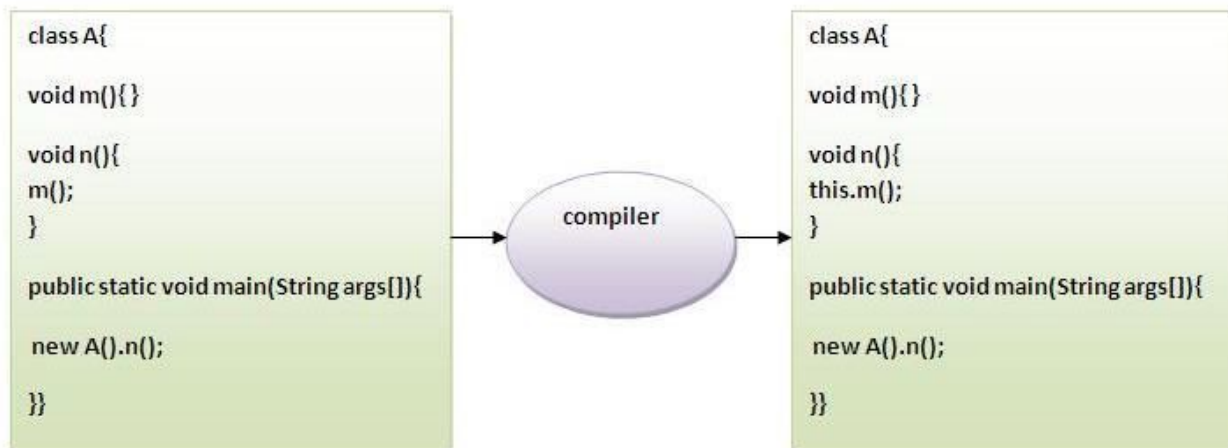
222 Aryan

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

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, the compiler automatically adds this keyword while invoking the method. Let's look at the example:

Class, Object, Encapsulation, Inheritance & Polymorphism



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 the 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
    }
}
```

1

2

Console

<terminated> PrintThisKeyword [Java Application] C:\Program Files\Java\jdk1.7.0_79\bin\javaw.exe (\$

oop.PrintThisKeyword@6276e1db

oop.PrintThisKeyword@6276e1db

Nested Classes

In Java, just like methods, variables of a class too can have another class as its member. Writing a class within another is allowed in Java. The class written within is called the nested class, and the class that holds the inner class is called the outer class.

Syntax

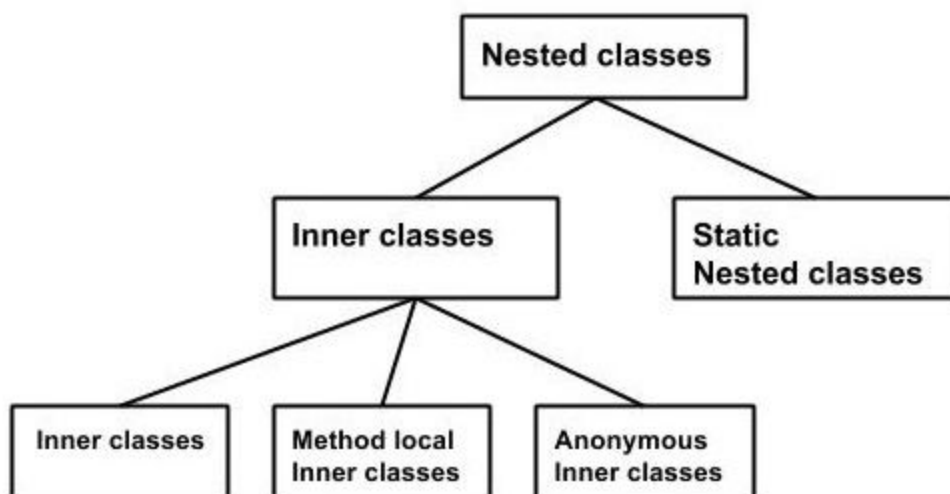
Following is the syntax to write a nested class. Here, the class Outer_Demo is the outer class and the class Inner_Demo is the nested class.

```
class Outer_Demo {  
    class Inner_Demo {  
    }  
}
```

Nested classes are divided into two types:

Non-static nested classes – These are the non-static members of a class.

Static nested classes – These are the static members of a class.



Class, Object, Encapsulation, Inheritance & Polymorphism

Inner Classes

Inner classes are a security mechanism in Java. We know a class cannot be associated with the access modifier private, but if we have the class as a member of another class, then the inner class can be made private. And this is also used to access the private members of a class.

Inner classes are of three types depending on how and where you define them. They are :

- Inner Class
- Method-local Inner Class
- Anonymous Inner Class

Inner Class

Creating an inner class is quite simple. You just need to write a class within a class. Unlike a class, an inner class can be private and once you declare an inner class private, it cannot be accessed from an object outside the class.

Method-local Inner Class

In Java, we can write a class within a method and this will be a local type. Like local variables, the scope of the inner class is restricted within the method.

Anonymous Inner Class

An inner class declared without a class name is known as an anonymous inner class. In case of anonymous inner classes, we declare and instantiate them at the same time. Generally, they are used whenever you need to override the method of a class or an interface.

Static Nested Class

A static inner class is a nested class which is a static member of the outer class. It can be accessed without instantiating the outer class, using other static members. Just like

Class, Object, Encapsulation, Inheritance & Polymorphism

static members, a static nested class does not have access to the instance variables and methods of the outer class.

Access Modifiers in Java

As the name suggests access modifiers in Java helps to restrict the scope of a class, constructor , variable , method or data member. There are four types of access modifiers available in java:

- Default – No keyword required
- Private
- Protected
- Public

	default	private	protected	public
Same Class	Yes	Yes	Yes	Yes
Same package subclass	Yes	No	Yes	Yes
Same package non-subclass	Yes	No	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Default: When no access modifier is specified for a class , method or data member – It is said to be having the default access modifier by default.

The data members, class or methods which are not declared using any access modifiers i.e. having default access modifier are accessible only within the same package.

private: The private access modifier is specified using the keyword private.

- The methods or data members declared as private are accessible only within the class in which they are declared.
- Any other class of the same package will not be able to access these members.
- Top level Classes or interface can not be declared as private because

Class, Object, Encapsulation, Inheritance & Polymorphism

1. private means “only visible within the enclosing class”.
2. protected means “only visible within the enclosing class and any subclasses”

Hence these modifiers in terms of application to classes, they apply only to nested classes and not on top level classes

protected: The protected access modifier is specified using the keyword protected.

- The methods or data members declared as protected are accessible within the same package or sub classes in different packages.

public: The public access modifier is specified using the keyword public.

- The public access modifier has the widest scope among all other access modifiers.
- Classes, methods or data members which are declared as public are accessible from everywhere in the program. There is no restriction on the scope of public data members.

Class, Object, Encapsulation, Inheritance & Polymorphism

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

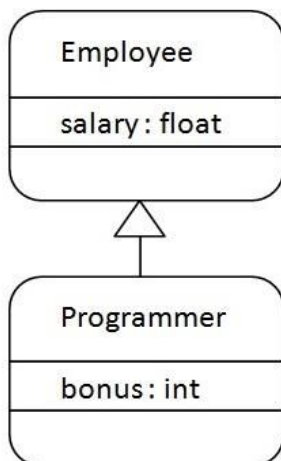
- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.

Syntax of Java Inheritance

```
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. 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, Object, Encapsulation, Inheritance & Polymorphism

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.

```
class Employee{  
    float salary=40000;  
}  
class Programmer extends Employee{  
    int bonus=10000;  
    public static void main(String args[]){  
        Programmer p=new Programmer();  
        System.out.println("Programmer salary is:"+p.salary);  
        System.out.println("Bonus of Programmer is:"+p.bonus);  
    }  
}
```

OUTPUT:

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.

Class, Object, Encapsulation, Inheritance & Polymorphism

```
Car.java
package oop.inheritance;

public class Car extends Vehicle {
    private int cc;
    private int gears;

    public void attributesCar() {
        // The subclass refers to the members of the superclass
        // System.out.println("Color of Car : " + color); //ERROR:private
        // field:color
        System.out.println("Speed of Car : " + super.speed); // super.speed or
        // speed
        System.out.println("Size of Car : " + size);
        System.out.println("CC of Car : " + cc);
        System.out.println("No of gears of Car : " + gears);
        super.attributes(); // WE CAN USE Super in any non static method.
    }

    public static void main(String[] args) {
        Car c1 = new Car();
        // c1.color = "Blue"; //ERROR: private field:color
        c1.speed = 200;
        c1.size = 22;
        c1.cc = 1000;
        c1.gears = 5;
        c1.attributes();
        // super.attributes(); //ERROR: Cannot use super in a static context
        c1.attributesCar();
    }
}
```

```
Vehicle.java
package oop.inheritance;

public class Vehicle {
    private String color;
    public int speed;
    protected int size;

    protected void attributes() { //public or protected.
        System.out.println("Color : " + color);
        System.out.println("Speed : " + speed);
        System.out.println("Size : " + size);
    }
}
```

Console Output:

```
<terminated> Car [Java Application] C:\Program Files\Java\
Color : null
Speed : 200
Size : 22
Speed of Car : 200
Size of Car : 22
CC of Car : 1000
No of gears of Car : 5
Color : null
Speed : 200
Size : 22
```

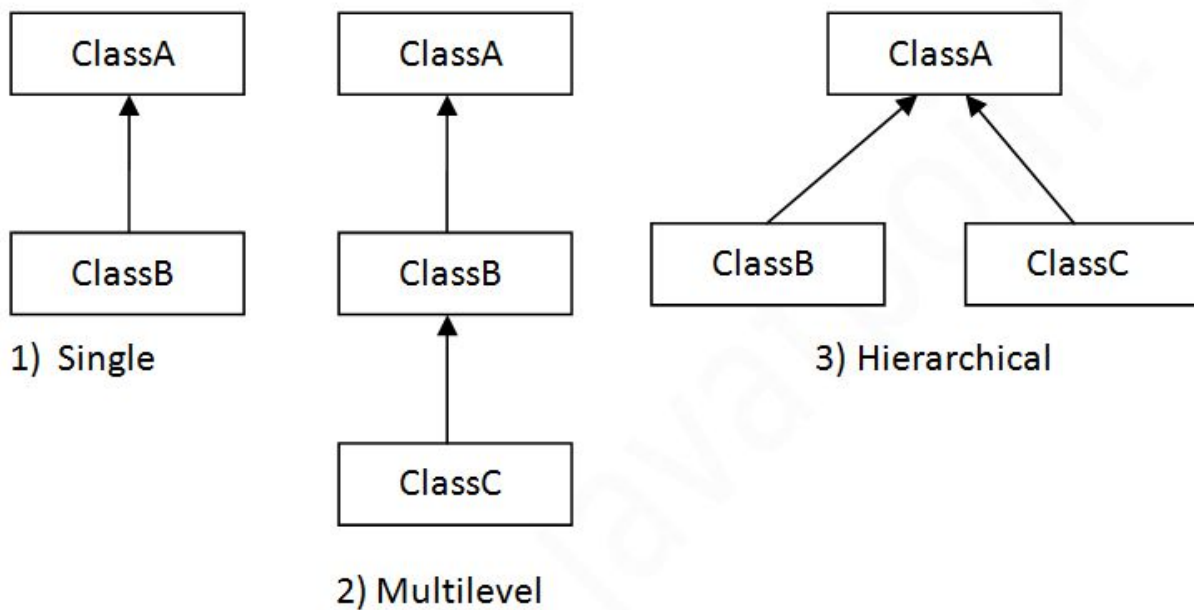
The console output is divided into two sections: "Vehicle" and "Car". The "Vehicle" section shows the output of the `attributes()` method, and the "Car" section shows the output of the `attributesCar()` method. A dashed arrow points from the `Car.java` file to the console output.

Class, Object, Encapsulation, Inheritance & Polymorphism

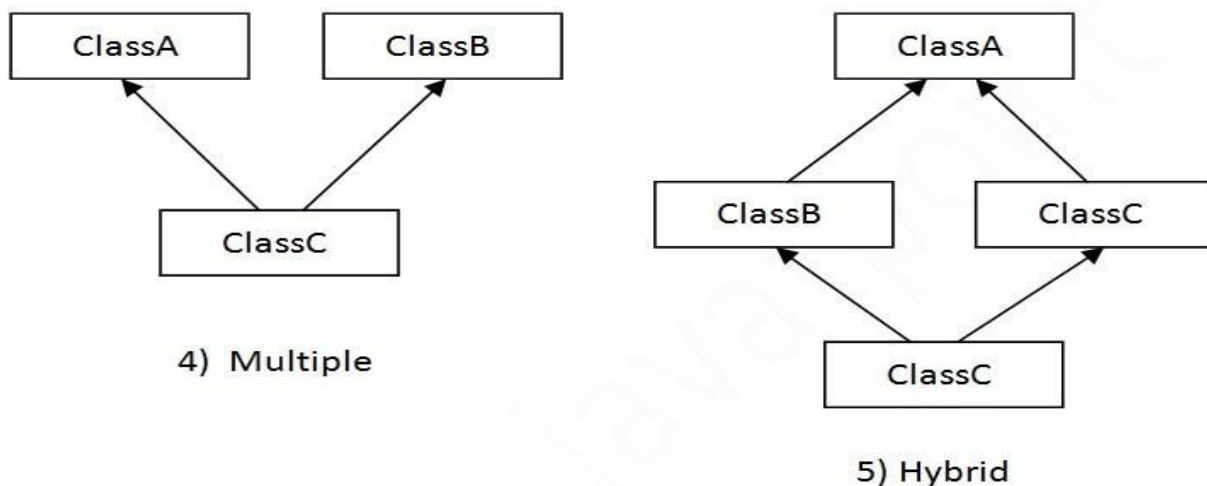
Types of inheritance in java

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 and Hybrid inheritance is not supported in java through class.



Class, Object, Encapsulation, Inheritance & Polymorphism

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 the same method or different, there will be **compile time error** now.

```
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 invoked?
    }
}
```

Class, Object, Encapsulation, Inheritance & Polymorphism

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 the 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

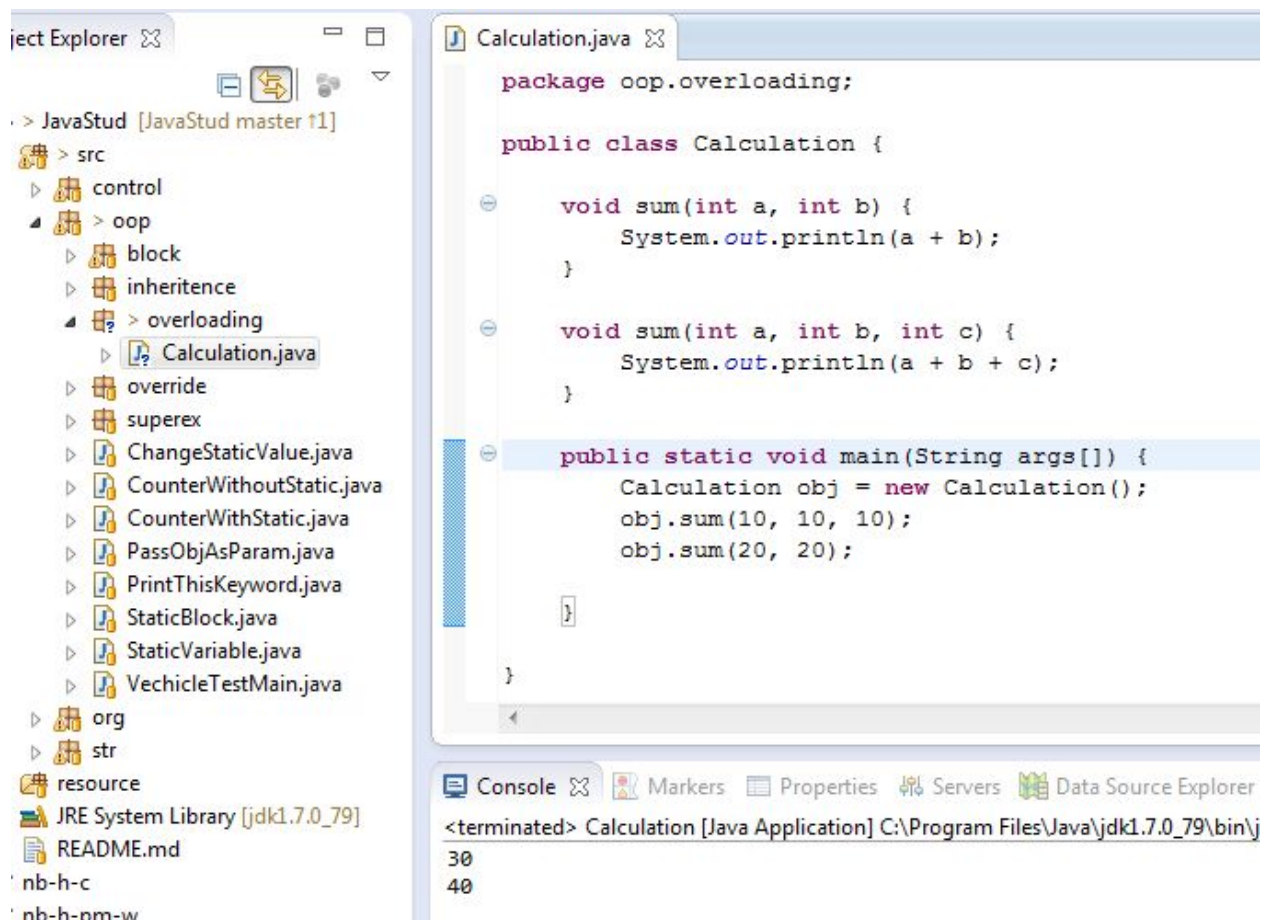
1. By changing number of arguments
2. By changing the data type

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

Class, Object, Encapsulation, Inheritance & Polymorphism

Example of Method Overloading by changing the no. of arguments

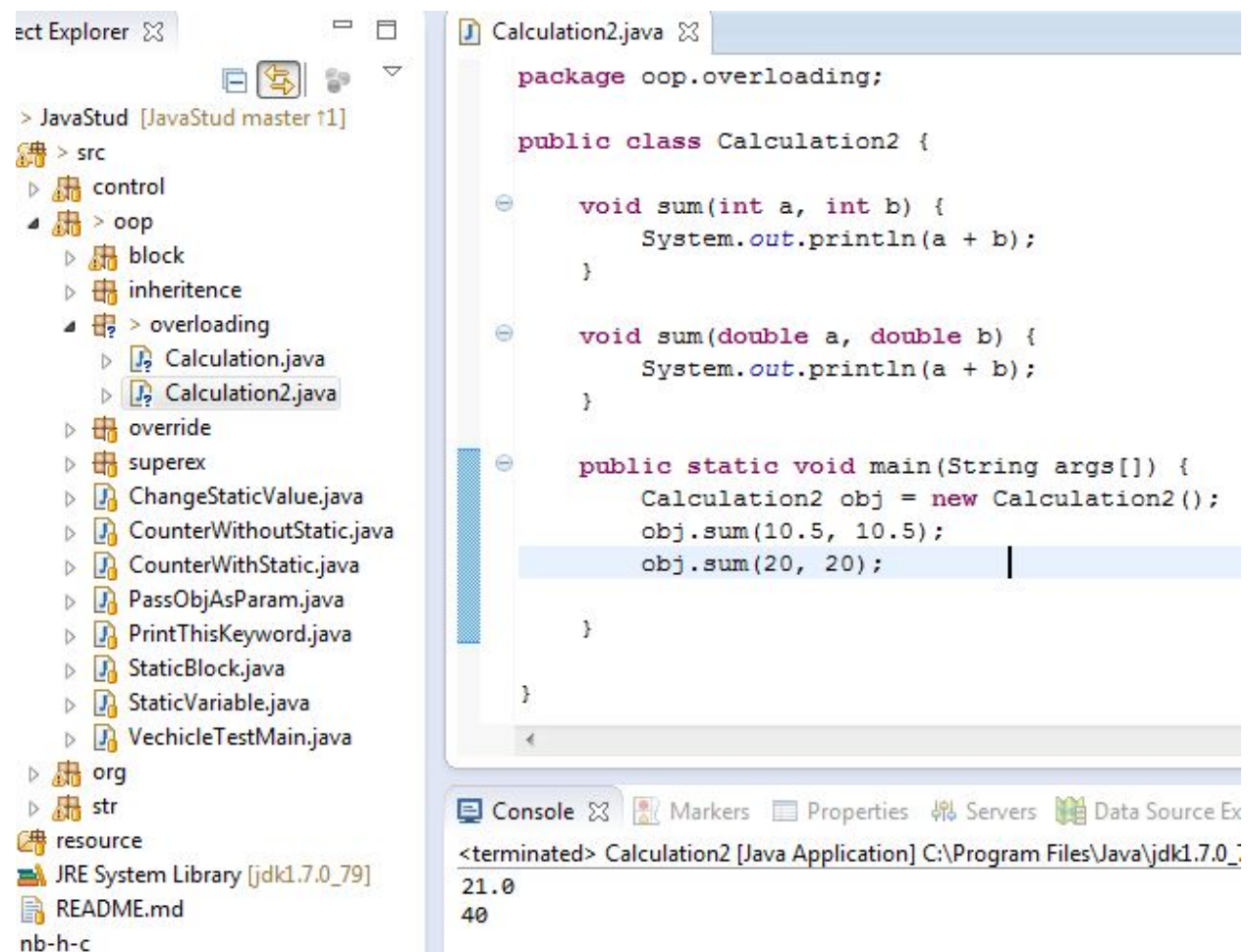
In this example, we have created two overloaded methods, first sum method performs addition of two numbers and second sum method performs addition of three numbers.



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.

Class, Object, Encapsulation, Inheritance & Polymorphism



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 a problem:

```
class Calculation3{
    int sum(int a,int b){
        System.out.println(a+b);
    }
    double sum(int a,int b){
        System.out.println(a+b);
    }
}
```

Class, Object, Encapsulation, Inheritance & Polymorphism

```
public static void main(String args[]){  
    Calculation3 obj=new Calculation3();  
    int result=obj.sum(20,20); //Compile Time Error  
}  
}
```

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

```
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 OverloadingCalculation1();  
        obj.sum(20,20); //now second int literal will be promoted to long  
        obj.sum(20,20,20);  
    }  
}
```

Test it Now Output:

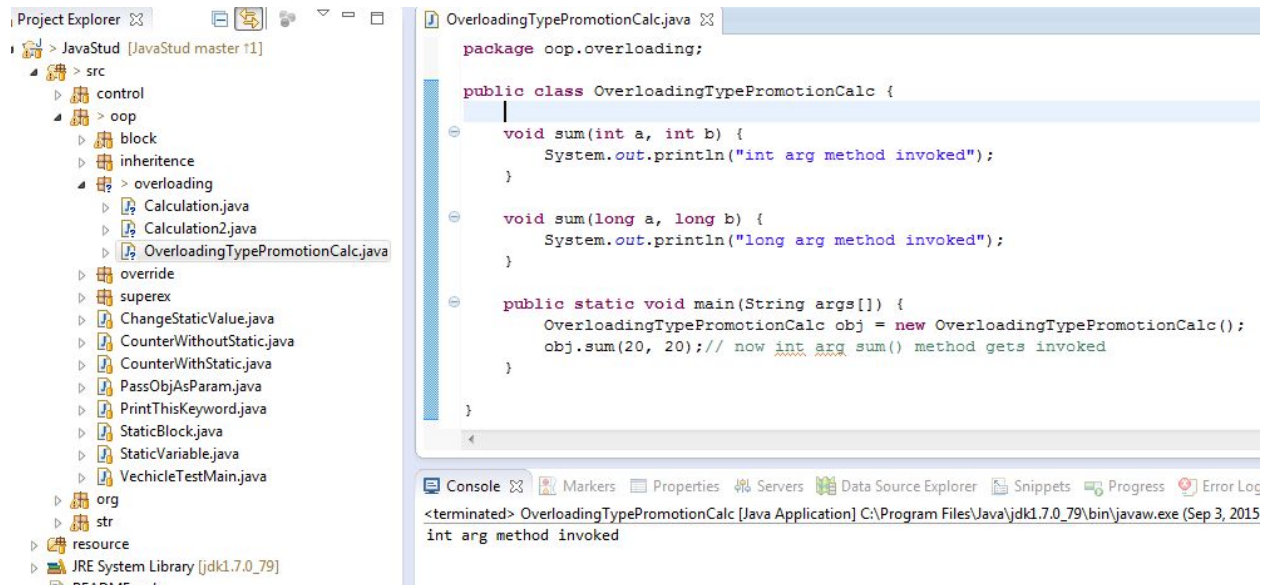
40

60

Class, Object, Encapsulation, Inheritance & Polymorphism

Example of Method Overloading with TypePromotion

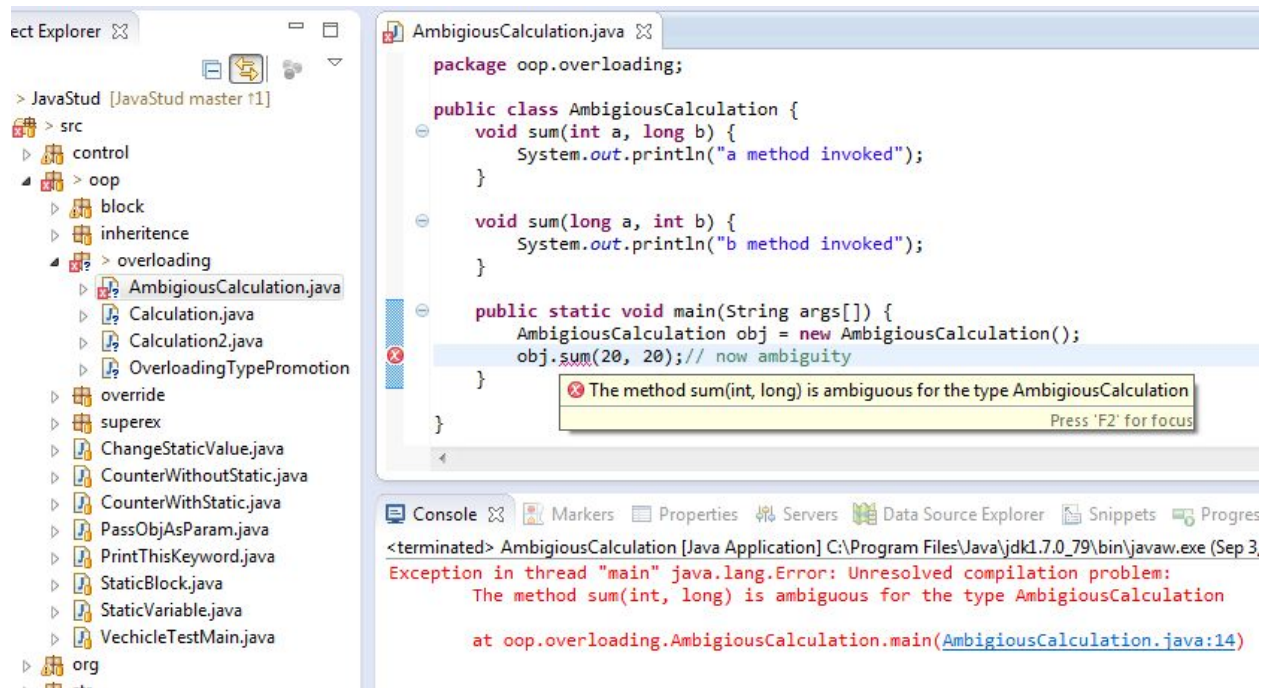
if matching found If there are matching type arguments in the method, type promotion is not performed.



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.

Class, Object, Encapsulation, Inheritance & Polymorphism



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

Class, Object, Encapsulation, Inheritance & Polymorphism

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

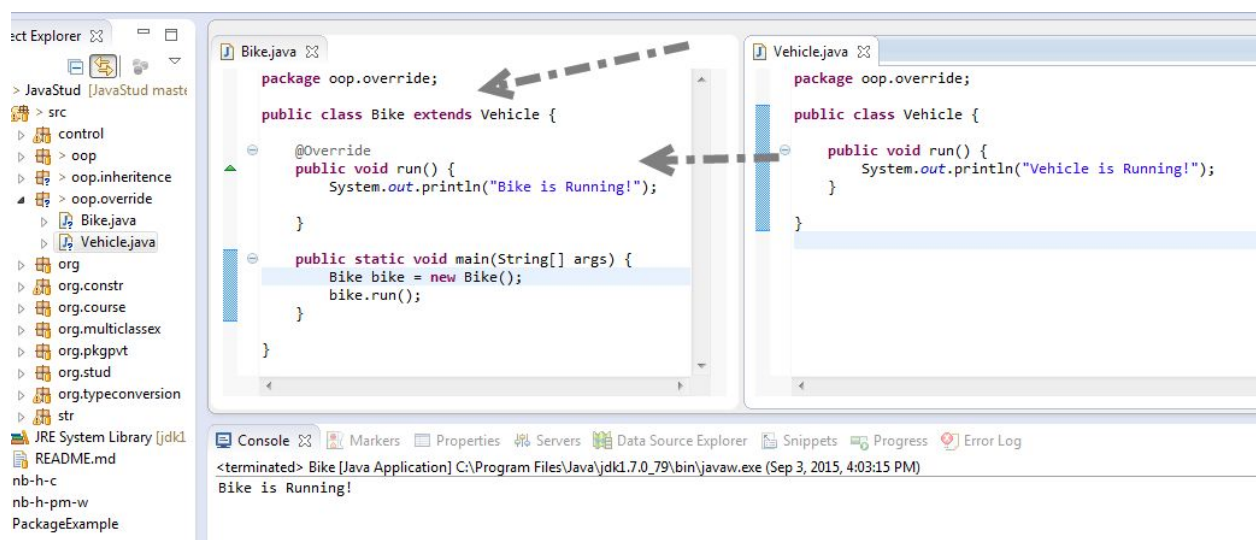
- Method overriding is used to provide specific implementation of a method that is already provided by its super class.
- Method overriding is used for runtime polymorphism

Rules for Java Method Overriding

- method must have same name as in the parent class
- method must have same parameter as in the parent class.
- must be IS-A relationship (inheritance).

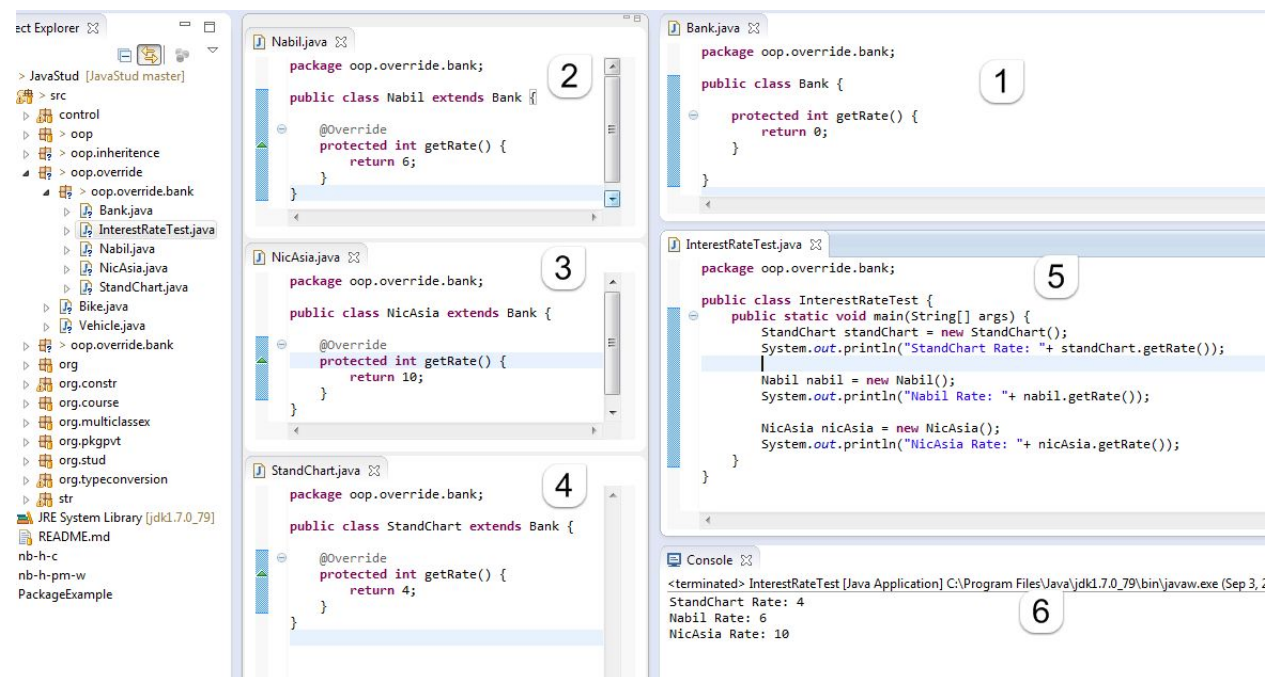
Example of method overriding

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 the same and there is IS-A relationship between the classes, so there is method overriding.



Class, Object, Encapsulation, Inheritance & Polymorphism

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

- Method overriding
- Method overloading

Class, Object, Encapsulation, Inheritance & Polymorphism

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 information such as city, state, country, zip code etc. as given below.

```
class Employee{  
    int id;  
    String name;  
    Address address;//Address is a class  
    .....  
}
```

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

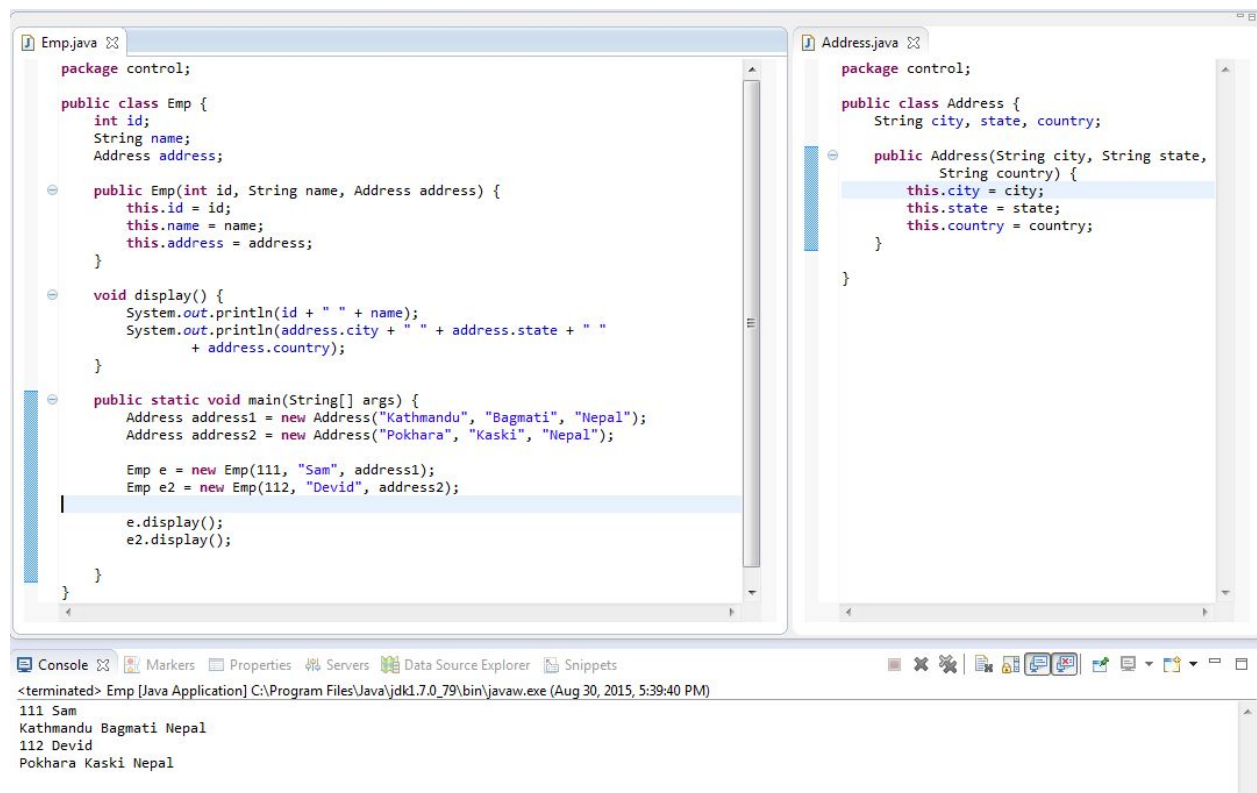
Why use Aggregation?

- For Code Reusability.

Understanding meaningful example of Aggregation

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

Class, Object, Encapsulation, Inheritance & Polymorphism



```
Emp.java
package control;

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("Kathmandu", "Bagmati", "Nepal");
        Address address2 = new Address("Pokhara", "Kaski", "Nepal");

        Emp e = new Emp(111, "Sam", address1);
        Emp e2 = new Emp(112, "Devid", address2);

        e.display();
        e2.display();
    }
}

Address.java
package control;

public class Address {
    String city, state, country;

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

Console
<terminated> Emp [Java Application] C:\Program Files\Java\jdk1.7.0_79\bin\javaw.exe (Aug 30, 2015, 5:39:40 PM)
111 Sam
Kathmandu Bagmati Nepal
112 Devid
Pokhara Kaski Nepal
```

Class, Object, Encapsulation, Inheritance & Polymorphism

super keyword in java

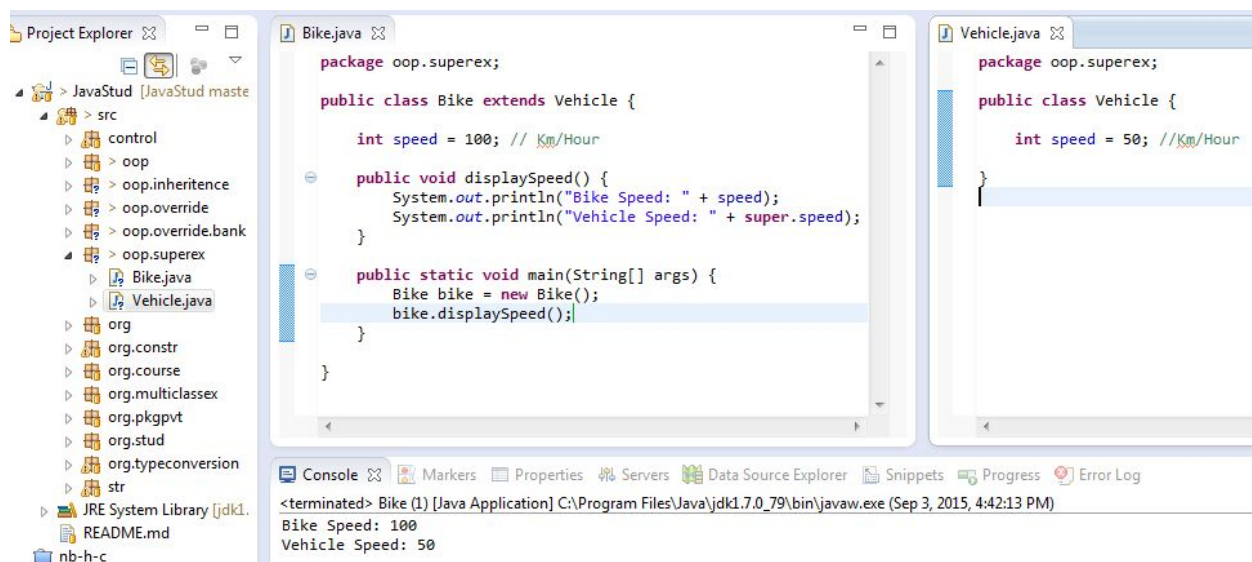
The super keyword in java is a reference variable that is used to refer immediate parent class object.

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

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

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



Class, Object, Encapsulation, Inheritance & Polymorphism

super is used to invoke parent class constructor

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

```
class Vehicle{
    Vehicle(){
        System.out.println("Vehicle is created");
    }
}
class Bike5 extends Vehicle{
    Bike5(){
        super();//will invoke parent class constructor
        System.out.println("Bike is created");
    }
    public static void main(String args[]){
        Bike5 b=new Bike5();
    }
}
```

Output:

Vehicle is created

Bike is created

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:

```
class Person{
    void message(){
```

Class, Object, Encapsulation, Inheritance & Polymorphism

```
        System.out.println("welcome");
    }
}
class Student16 extends Person{
    void message(){
        System.out.println("welcome to java");
    }
    void display(){
        message();//will invoke current class message() method
        super.message();//will invoke parent class message() method
    }
    public static void main(String args[]){
        Student16 s=new Student16();
        s.display();
    }
}
```

Output:

welcome to java

Welcome

Program in case super is not required

```
class Person{
    void message(){
        System.out.println("welcome");
    }
}
class Student extends Person{
    void display(){
```

Class, Object, Encapsulation, Inheritance & Polymorphism

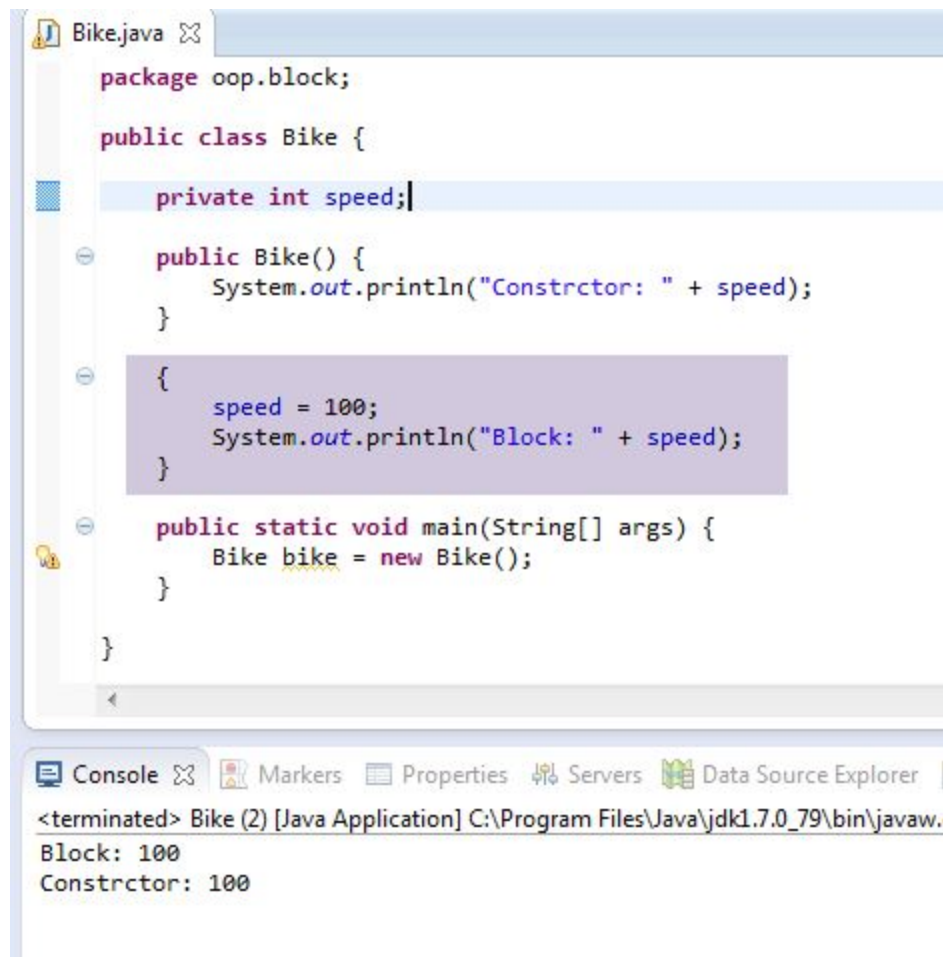
```
        message();//will invoke parent class message() method
    }
    public static void main(String args[]){
        Student s=new Student ();
        s.display();
    }
}
```

Output:

Welcome

Class, Object, Encapsulation, Inheritance & Polymorphism

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

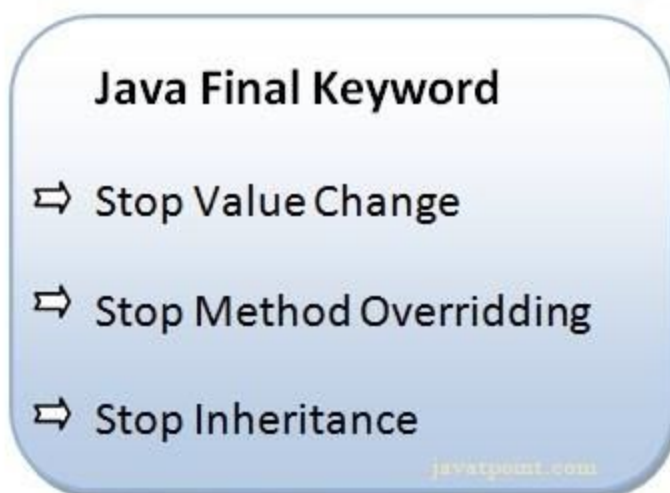
<terminated> Bike (2) [Java Application] C:\Program Files\Java\jdk1.7.0_79\bin\javaw.
Block: 100
Constructor: 100

Final Keyword In Java

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

- variable
- method
- 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 the final keyword.



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

Class, Object, Encapsulation, Inheritance & Polymorphism

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

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

Output:Compile Time Error

Java final method

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

Example of final method

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

class Honda extends Bike{
    void run(){
        System.out.println("running safely with 100kmph");
    }
    public static void main(String args[]){
```

Class, Object, Encapsulation, Inheritance & Polymorphism

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

Output: Compile Time Error

Java final class

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

Example of final class

```
final class Bike{  
class Honda1 extends Bike{  
    void run(){  
        System.out.println("running safely with 100kmph");  
    }  
    public static void main(String args[]){  
        Honda1 honda= new Honda();  
        honda.run();  
    }  
}
```

Output:Compile Time Error

Is final method inherited?

Yes, the final method is inherited but you cannot override it.

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

Class, Object, Encapsulation, Inheritance & Polymorphism

```
}
```

```
class Honda2 extends Bike{  
    public static void main(String args[]){  
        new Honda2().run();  
    }  
}
```

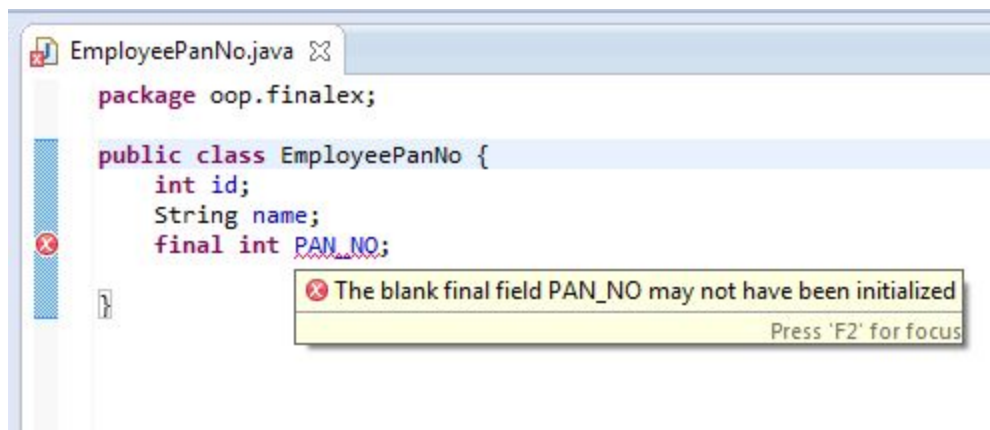
Output: running.....

What is blank or uninitialized final variable?

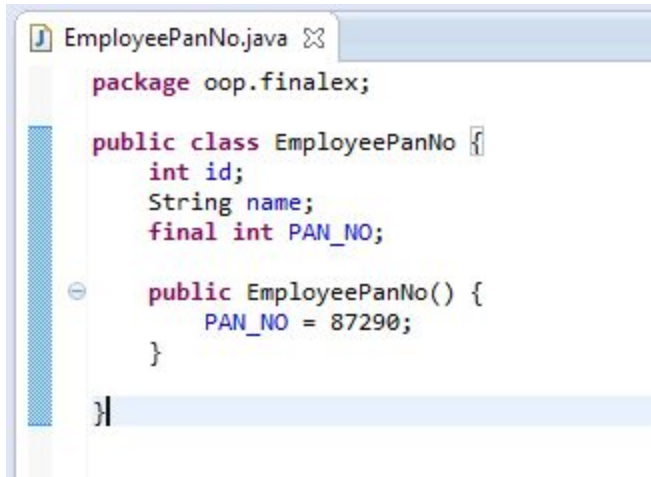
A final variable that is not initialized at the time of declaration is known as blank final variable. If you want to create a variable that is initialized at the time of creating object and once initialized may not be changed, it is useful.

For example PAN CARD number of an employee.

It can be initialized only in constructor.



Class, Object, Encapsulation, Inheritance & Polymorphism



```
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.

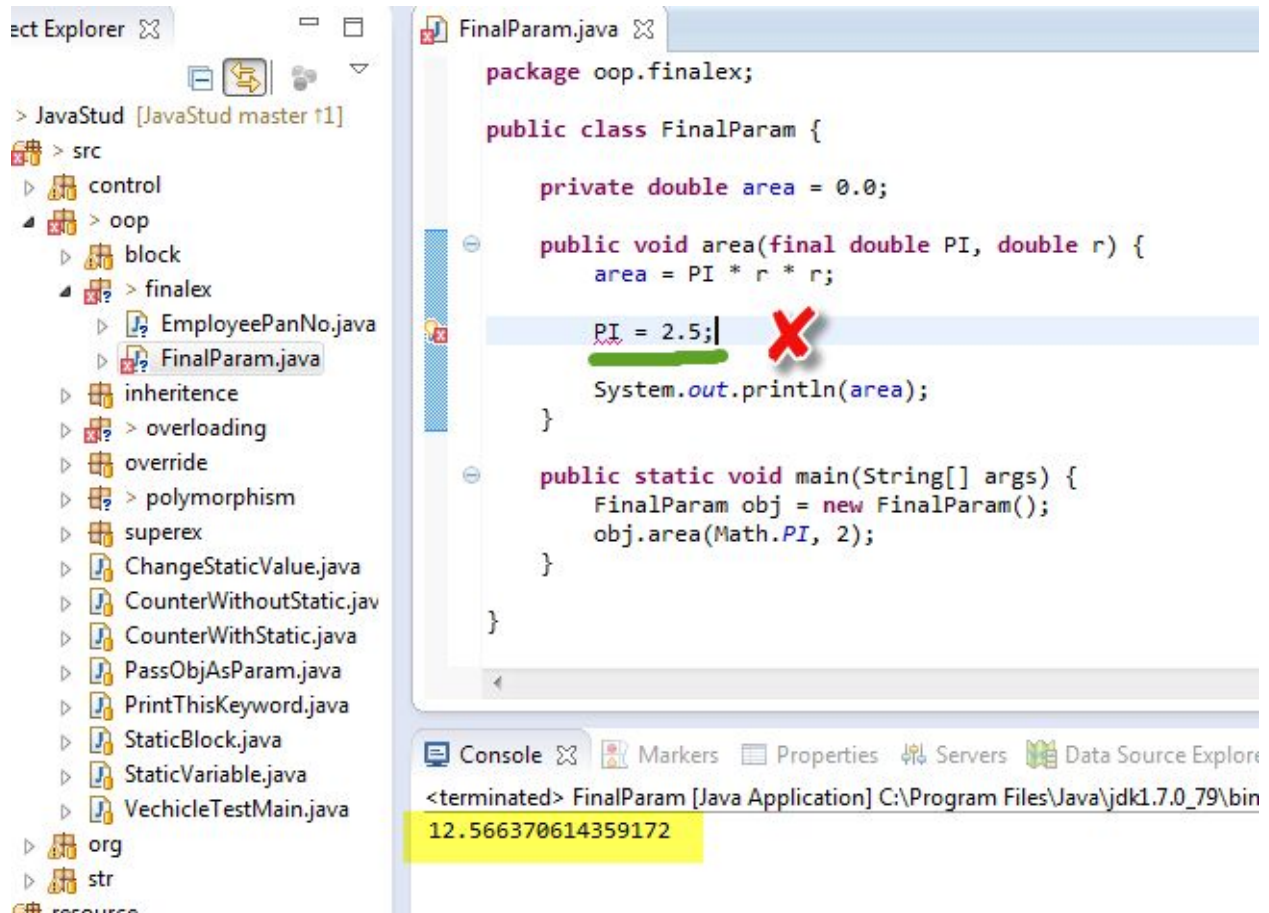
Example of static blank final variable

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

Class, Object, Encapsulation, Inheritance & Polymorphism

What is final parameter?

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



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 in 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 methods in java, it is an 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:

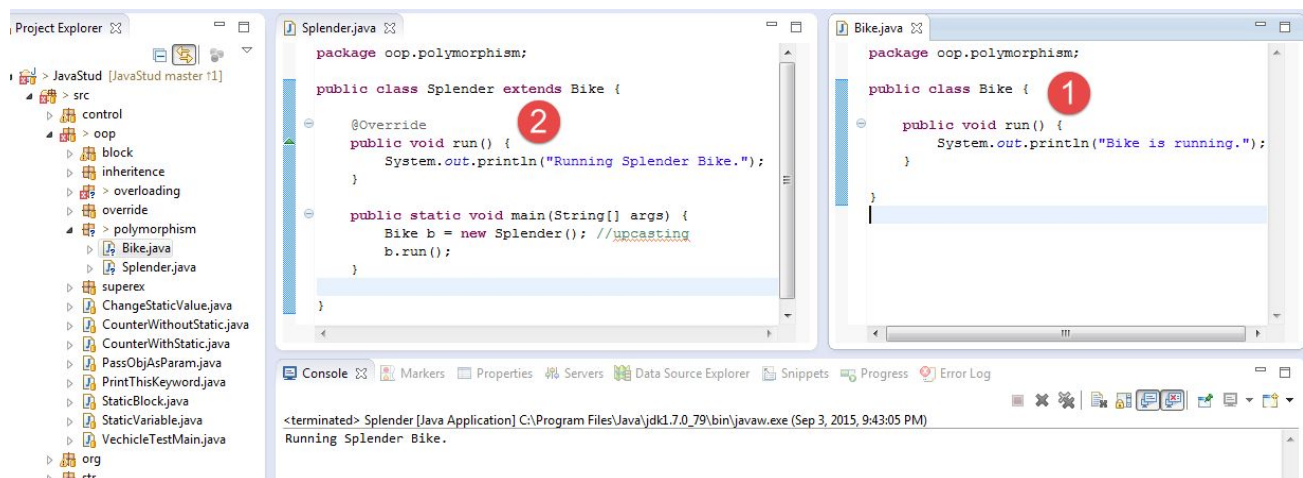


```
class A{}  
class B extends A{}  
A a=new B();//upcasting
```

Class, Object, Encapsulation, Inheritance & Polymorphism

Example of Java Runtime Polymorphism

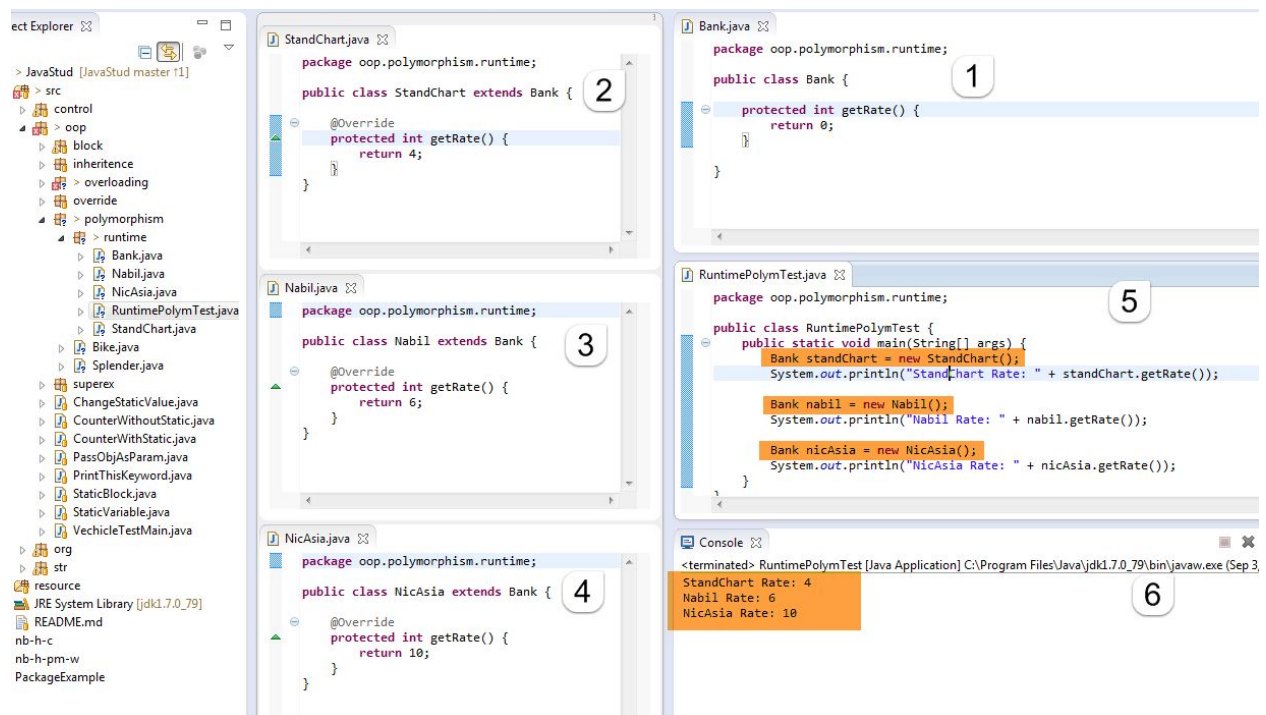
In this example, we are creating two classes: Bike and Splendor. The Splendor class extends the Bike class and overrides its run() method. We are calling the run method by the reference variable of Parent class. Since it refers to the subclass object and subclass method overrides the Parent class method,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

Class, Object, Encapsulation, Inheritance & Polymorphism



Java Runtime Polymorphism with data member

Method is overridden not the data members, 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 always access the datamember of the Parent class.

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

```
class Bike{
    int speedLimit=90;
}

class Honda3 extends Bike{
    int speedLimit=150;
```

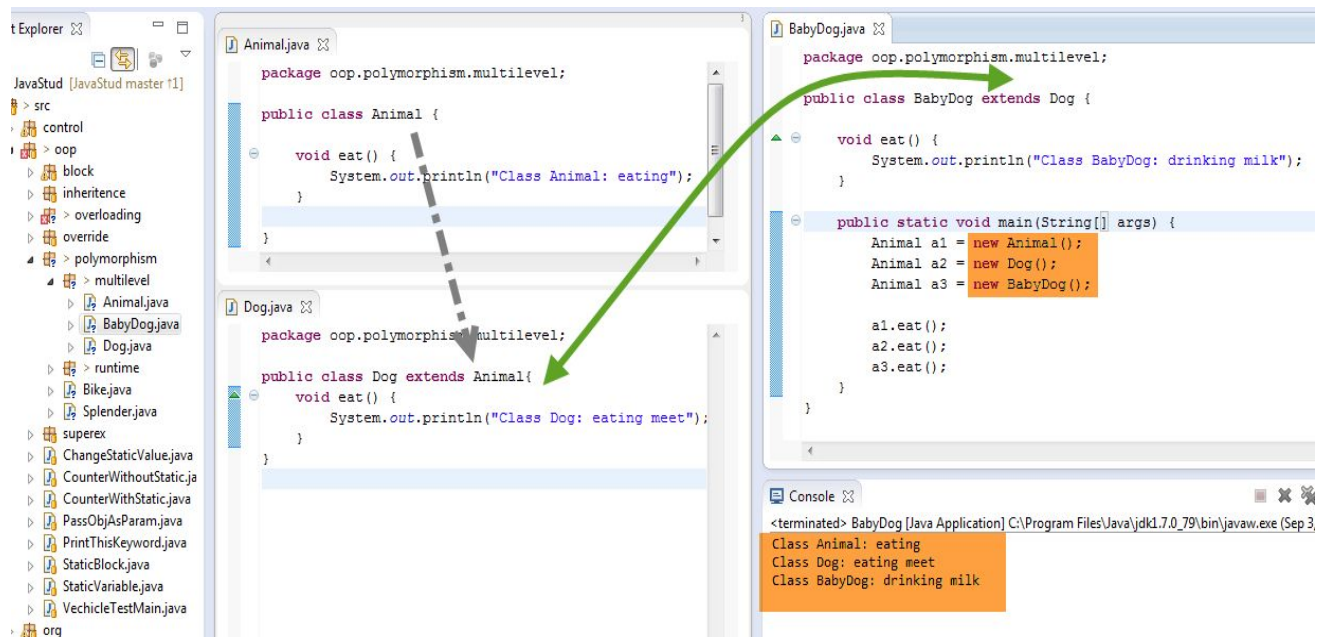
Class, Object, Encapsulation, Inheritance & Polymorphism

```
public static void main(String args[]){  
    Bike obj=new Honda3();  
    System.out.println(obj.speedLimit);//90  
}  
}
```

Output:

90

Java Runtime Polymorphism with Multilevel Inheritance



Class, Object, Encapsulation, Inheritance & Polymorphism

Static Binding and Dynamic Binding

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

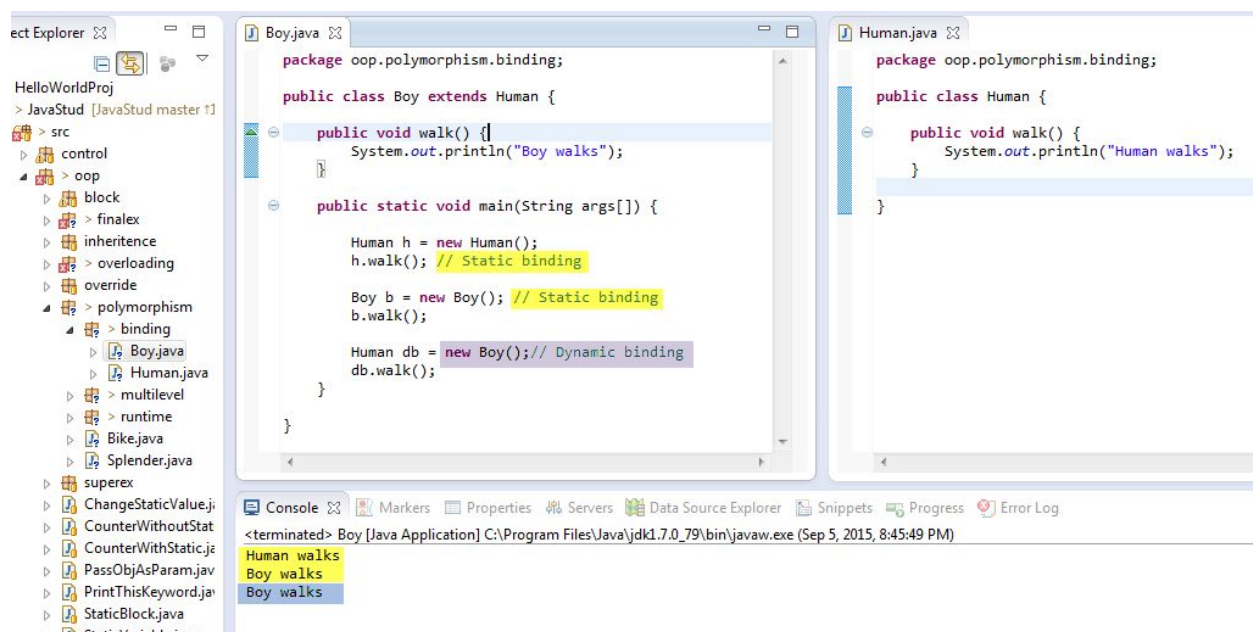
- static binding (also known as early binding).
- dynamic binding (also known as late binding).

Static binding

When type of the object is determined at compile 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

- Abstract class (0 to 100%)
- 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 has at least one abstract method, then the class must be declared abstract.

Example abstract class

```
abstract class Bank {}
```

abstract method

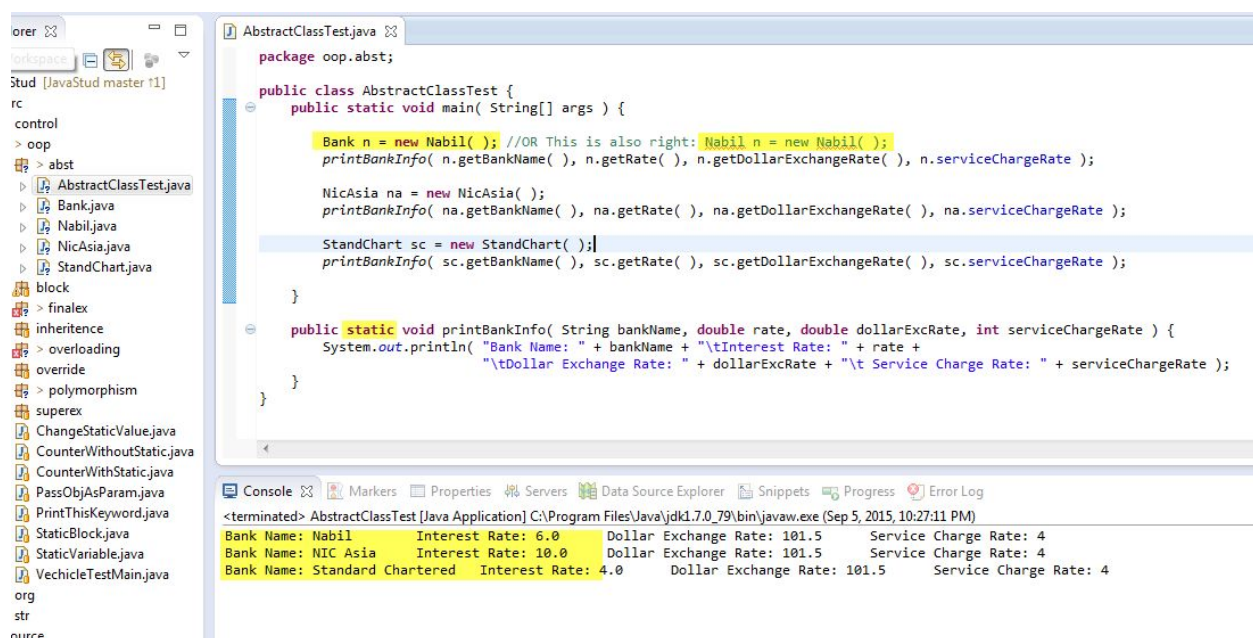
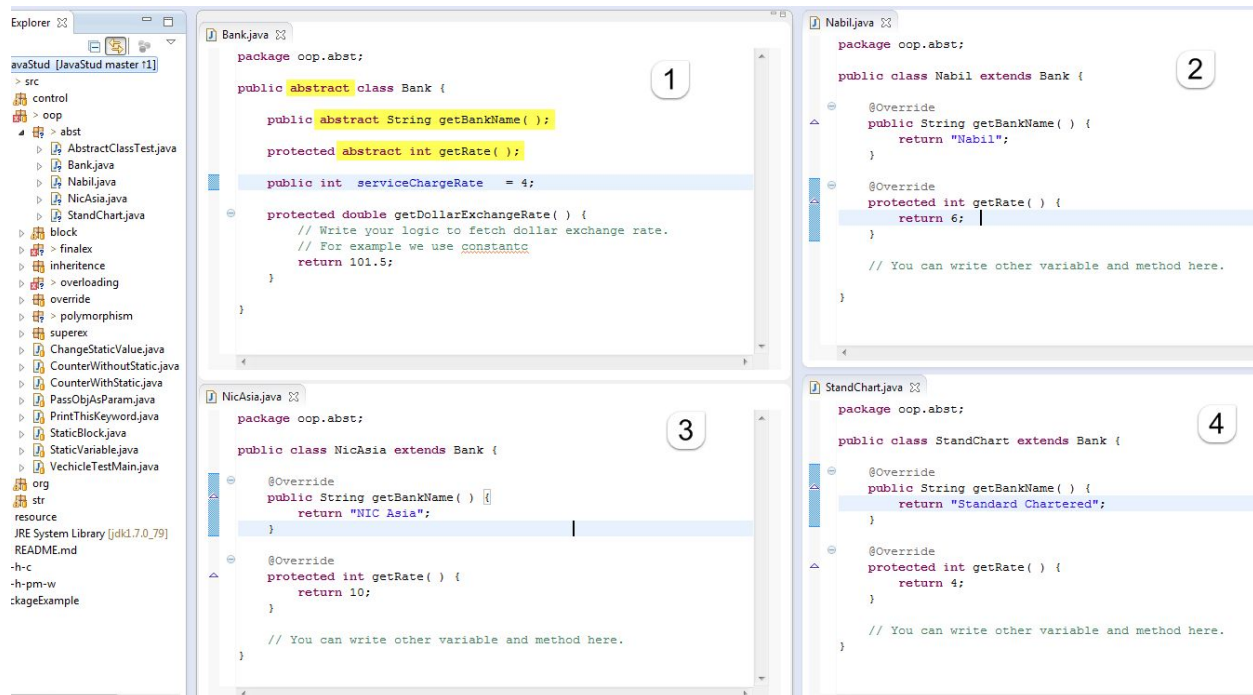
A method that is declared as abstract and does not have implementation is known as abstract method. Example abstract method

Class, Object, Encapsulation, Inheritance & Polymorphism

abstract void getRate ();//no body and abstract

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

Understanding the real scenario of abstract class



Class, Object, Encapsulation, Inheritance & Polymorphism

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

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

It cannot be instantiated just like abstract class.

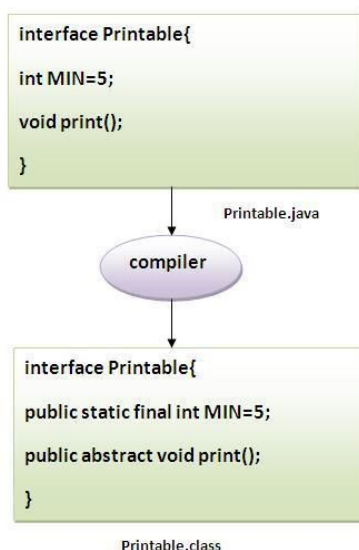
Why use Java interface?

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

- It is used to achieve full 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.

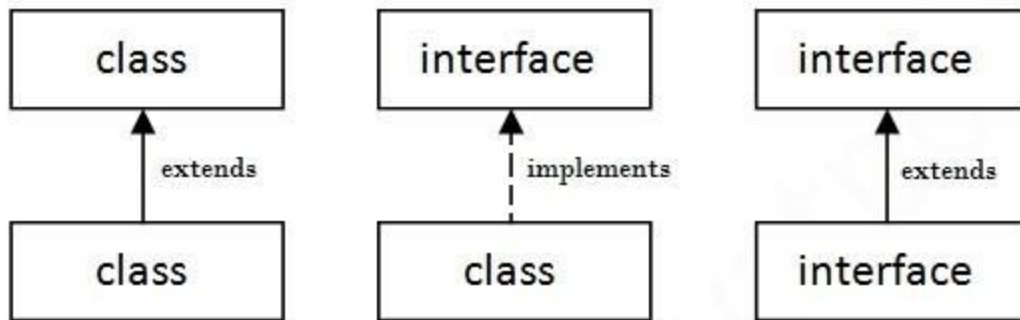
In other words, Interface fields are public, static and final by default, and methods are public and abstract.



Class, Object, Encapsulation, Inheritance & Polymorphism

Understanding the 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 has only one method, its implementation is provided in the A class.

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

Output: Hello

Class, Object, Encapsulation, Inheritance & Polymorphism

Real World Example:

The screenshot displays four Java files in an IDE, illustrating multiple inheritance by interface. The files are: 1. Bank.java (Interface), 2. Nabil.java (Implementation), 3. NicAsia.java (Implementation), and 4. StandChart.java (Implementation). The InterfaceImplTest.java file is also shown, testing the implementation. The console output shows the results of the test.

```
package oop.interf;

public class Bank {

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

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

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

```
package oop.interf;

public class Nabil implements Bank {

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

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

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

```
package oop.interf;

public class NicAsia implements Bank {

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

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

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

```
package oop.interf;

public class StandChart implements Bank {

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

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

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

```
package oop.interf;

public class InterfaceImplTest {

    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.serviceChargeRate );

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

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

    }

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

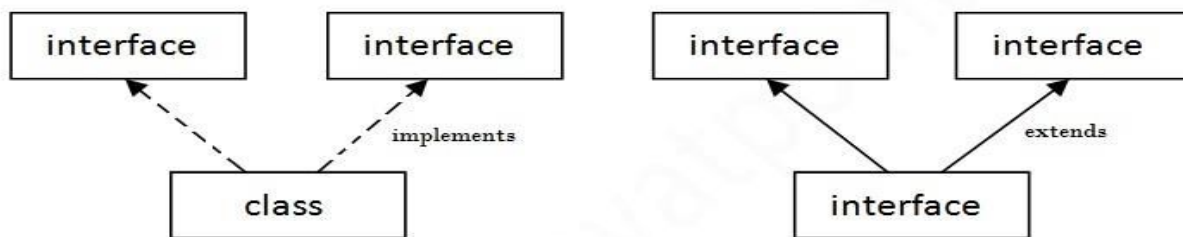
Console Output:

```
<terminated> InterfaceImplTest [Java Application] C:\Program Files\Java\jdk1.7.0_79\bin\javaw.exe (Sep 5, 2015, 11:15:06 PM)
Bank Name: Nabil      Interest Rate: 6.0      Service Charge Rate: 4
Bank Name: NIC Asia   Interest Rate: 10.0     Service Charge Rate: 4
Bank Name: Standard Chartered Interest Rate: 4.0     Service Charge Rate: 4
```

Multiple inheritance in Java by interface

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

Class, Object, Encapsulation, Inheritance & Polymorphism



Multiple Inheritance in Java

```
interface Printable{
    void print();
}
interface Showable{
    void show();
}
class Impl implements Printable,Showable{
    public void print(){
        System.out.println("Print Hello");
    }
    public void show(){
        System.out.println("Show Welcome");
    }
    public static void main(String args[]){
        Impl obj = new Impl ();
        obj.print(); 17. obj.show();
    }
}
```

Output:

Print Hello

Show Welcome

Class, Object, Encapsulation, Inheritance & Polymorphism

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:

```
interface Printable{
    void print();
}
interface Showable{
    void print();
}
class testinterface1 implements Printable,Showable{
    public void print(){
        System.out.println("Hello");
    }
    public static void main(String args[]){
        testinterface1 obj = new testinterface1();
        obj.print();
    }
}
```

Output:

Hello

As you can see in the above example, Printable and Showable interface have the 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 .

```
interface Printable{
```

Class, Object, Encapsulation, Inheritance & Polymorphism

```
    void print();
}
interface Showable extends Printable{
    void show();
}
class Testinterface2 implements Showable{
    public void print(){System.out.println("Hello");}
    public void show(){System.out.println("Welcome");}
    public static void main(String args[]){
        Testinterface2 obj = new Testinterface2();
        obj.print();
        obj.show();
    }
}
```

Output:

Hello

Welcome

What is marker or tagged interface?

An interface that has 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.

//How Serializable interface is written?

```
public interface Serializable{
}
```

Class, Object, Encapsulation, Inheritance & Polymorphism

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
Abstract class can have abstract and non-abstract methods.	Interface can have only abstract methods.
Abstract class doesn't support multiple inheritance.	Interface supports multiple inheritance.
Abstract class can have final, non-final, static and non-static variables.	Interface has only static and final variables.
Abstract class can have static methods, main method and constructor.	Interface can't have static methods, main method or constructor.
Abstract class can provide the implementation of interface.	Interface can't provide the implementation of abstract class.
The abstract keyword is used to declare abstract class.	The interface keyword is used to declare interface.
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%).

Class, Object, Encapsulation, Inheritance & Polymorphism

Difference between method overloading and method overriding in java

There are many differences between method overloading and method overriding in java.

A list of differences between method overloading and method overriding are given below:

No.	Method Overloading	Method Overriding
1)	Method overloading is used to <i>increase the readability</i> of the program.	Method overriding is used to <i>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.

Object Class

Object class is present in java.lang package. Every class in Java is directly or indirectly derived from the Object class. If a Class does not extend any other class then it is direct child class of Object and if it extends another class then it is indirectly derived. Therefore the Object class methods are available to all Java classes. Hence Object class acts as a root of inheritance hierarchy in any Java Program.

Using Object class methods

There are methods in Object class some of them are:

toString() : toString() provides a string representation of an Object and used to convert an object to String.

hashCode() : For every object, JVM generates a unique number which is hashCode. It returns distinct integers for distinct objects.

equals(Object obj) : Compares the given object to “this” object (the object on which the method is called).

Note : It is generally necessary to override the hashCode() method whenever this method is overridden, so as to maintain the general contract for the hashCode method, which states that equal objects must have equal hash codes.

getClass() : Returns the class object of “this” object and used to get the actual runtime class of the object.

Class, Object, Encapsulation, Inheritance & Polymorphism

Methods of Object class

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.