

Objects and Classes in Java

An object in Java is the physical as well as logical entity whereas a class in Java is a logical entity only.

What is an object in Java

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

An object has three characteristics:

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

For Example, Pen is an object. Its name is Reynolds; color is white, known as its state. It is used to write, so writing is its behavior.

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

Object Definitions:

- An object is *a real-world entity*.
- An object is *a runtime entity*.
- The object is *an entity which has state and behavior*.
- The object is *an instance of a class*.

What is a class in Java

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

A class in Java can contain:

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

Syntax to declare a class:

1. `class <class_name>{`
2. `field;`
3. `method;`
4. `}`

Instance variable in Java

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

Method in Java

In Java, a method is like a function which is used to expose the behavior of an object.

Advantage of Method

- Code Reusability
 - Code Optimization
-

new keyword in Java

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

Object and Class Example: main within the class

In this example, we have created a Student class which has two data members id and name. We are creating the object of the Student class by new keyword and printing the object's value.

Here, we are creating a main() method inside the class.

File: Student.java

```
1. //Java Program to illustrate how to define a class and fields
2. //Defining a Student class.
3. class Student{
4.     //defining fields
5.     int id;//field or data member or instance variable
6.     String name;
7.     //creating main method inside the Student class
8.     public static void main(String args[]){
9.         //Creating an object or instance
10.        Student s1=new Student();//creating an object of Student
11.        //Printing values of the object
12.        System.out.println(s1.id);//accessing member through reference variable
13.        System.out.println(s1.name);
14.    }
15. }
```

[Test it Now](#)

Output:

```
0
null
```

Object and Class Example: main outside the class

In real time development, we create classes and use it from another class. It is a better approach than previous one. Let's see a simple example, where we are having main() method in another class.

We can have multiple classes in different java files or single java file. If you define multiple classes in a single java source file, it is a good idea to save the file name with the class name which has main() method.

File: TestStudent1.java

```
1. //Java Program to demonstrate having the main method in
2. //another class
3. //Creating Student class.
4. class Student{
5.     int id;
6.     String name;
7. }
8. //Creating another class TestStudent1 which contains the main method
9. class TestStudent1 {
10.     public static void main(String args[]){
11.         Student s1=new Student();
12.         System.out.println(s1.id);
13.         System.out.println(s1.name);
14.     }
15. }
```

[Test it Now](#)

Output:

```
0
null
```

3 Ways to initialize object

There are 3 ways to initialize object in java.

1. By reference variable
2. By method
3. By constructor

1) Object and Class Example: Initialization through reference

Initializing an object means storing data into the object. Let's see a simple example where we are going to initialize the object through a reference variable.

File: TestStudent2.java

```
1. class Student{
2.     int id;
3.     String name;
4. }
5. class TestStudent2{
```

```
6. public static void main(String args[]){
7.     Student s1=new Student();
8.     s1.id=101;
9.     s1.name="Sonoo";
10.    System.out.println(s1.id+" "+s1.name);//printing members with a white space
11. }
12. }
```

[Test it Now](#)

Output:

```
101 Sonoo
```

We can also create multiple objects and store information in it through reference variable.

File: TestStudent3.java

```
1. class Student{
2.     int id;
3.     String name;
4. }
5. class TestStudent3{
6.     public static void main(String args[]){
7.         //Creating objects
8.         Student s1=new Student();
9.         Student s2=new Student();
10.        //Initializing objects
11.        s1.id=101;
12.        s1.name="Sonoo";
13.        s2.id=102;
14.        s2.name="Amit";
15.        //Printing data
16.        System.out.println(s1.id+" "+s1.name);
17.        System.out.println(s2.id+" "+s2.name);
18.    }
19. }
```

[Test it Now](#)

Output:

```
101 Sonoo
102 Amit
```

2) Object and Class Example: Initialization through method

In this example, we are creating the two objects of Student class and initializing the value to these objects by invoking the insertRecord method. Here, we are displaying the state (data) of the objects by invoking the displayInformation() method.

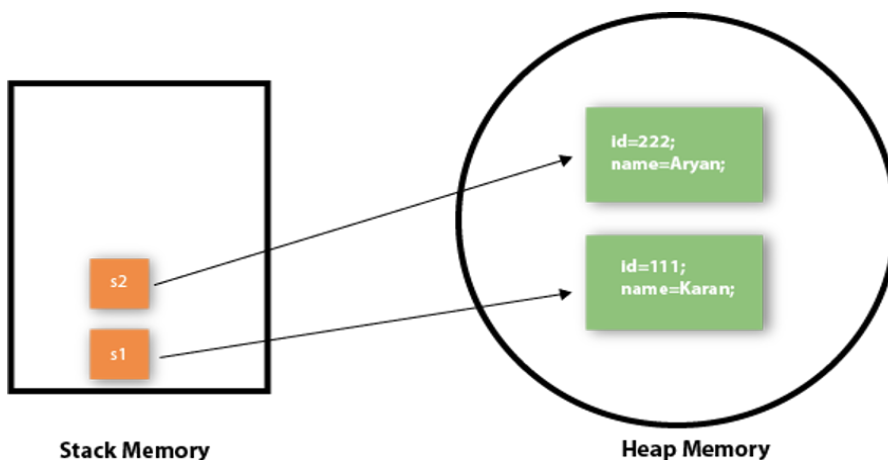
File: TestStudent4.java

```
1. class Student{
2.   int rollno;
3.   String name;
4.   void insertRecord(int r, String n){
5.     rollno=r;
6.     name=n;
7.   }
8.   void displayInformation(){System.out.println(rollno+" "+name);}
9. }
10. class TestStudent4{
11.   public static void main(String args[]){
12.     Student s1=new Student();
13.     Student s2=new Student();
14.     s1.insertRecord(111,"Karan");
15.     s2.insertRecord(222,"Aryan");
16.     s1.displayInformation();
17.     s2.displayInformation();
18.   }
19. }
```

[Test it Now](#)

Output:

```
111 Karan
222 Aryan
```



As you can see in the above figure, object gets the memory in heap memory area. The reference variable refers to the object allocated in the heap memory area. Here, s1 and s2 both are reference variables that refer to the objects allocated in memory.

3) Object and Class Example: Initialization through a constructor

We will learn about constructors in java later.

Object and Class Example: Employee

Let's see an example where we are maintaining records of employees.

File: TestEmployee.java

```
1. class Employee{
2.     int id;
3.     String name;
4.     float salary;
5.     void insert(int i, String n, float s) {
6.         id=i;
7.         name=n;
8.         salary=s;
9.     }
10.    void display(){System.out.println(id+" "+name+" "+salary);}
11. }
12. public class TestEmployee {
13. public static void main(String[] args) {
14.     Employee e1=new Employee();
15.     Employee e2=new Employee();
16.     Employee e3=new Employee();
17.     e1.insert(101,"ajeet",45000);
18.     e2.insert(102,"irfan",25000);
19.     e3.insert(103,"nakul",55000);
20.     e1.display();
21.     e2.display();
22.     e3.display();
23. }
24. }
```

[Test it Now](#)

Output:

```
101 ajeet 45000.0
102 irfan 25000.0
103 nakul 55000.0
```

Object and Class Example: Rectangle

There is given another example that maintains the records of Rectangle class.

File: TestRectangle1.java

```
1. class Rectangle{
2.   int length;
3.   int width;
4.   void insert(int l, int w){
5.     length=l;
6.     width=w;
7.   }
8.   void calculateArea(){System.out.println(length*width);}
9. }
10. class TestRectangle1 {
11.   public static void main(String args[]){
12.     Rectangle r1=new Rectangle();
13.     Rectangle r2=new Rectangle();
14.     r1.insert(11,5);
15.     r2.insert(3,15);
16.     r1.calculateArea();
17.     r2.calculateArea();
18.   }
19. }
```

[Test it Now](#)

Output:

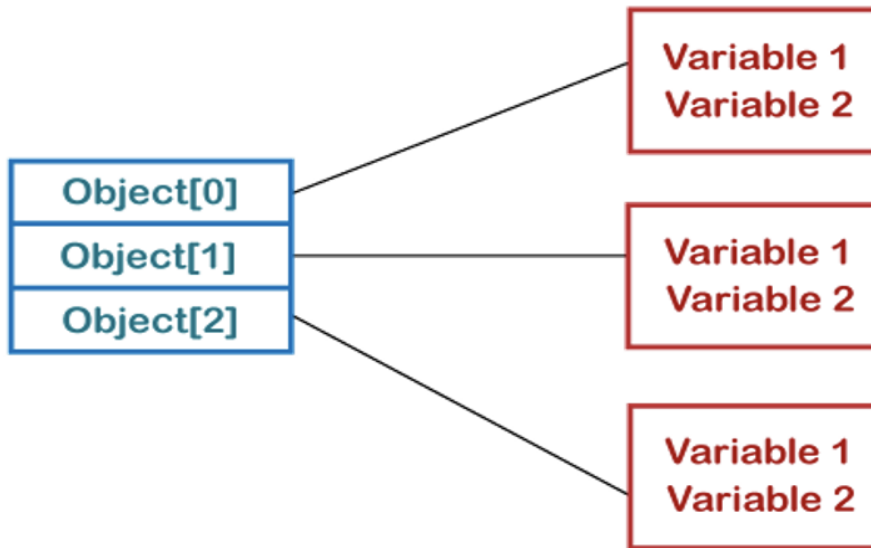
55

45

Array of Objects in Java

Java is an object-oriented programming language. Most of the work is done with the help of **objects**. We know that an array is a collection of the same data type that dynamically creates objects and can have elements of primitive types. Java allows us to store objects in an array. In [Java](#), the class is also a user-defined data type. An array that contains **class type elements** are known as an **array of objects**. It stores the reference variable of the object.

Arrays of Objects



Creating an Array of Objects

Before creating an array of objects, we must create an instance of the class by using the new keyword. We can use any of the following statements to create an array of objects.

Syntax:

1. `ClassName obj[]=new ClassName[array_length];` //declare and instantiate an array of objects
- Or
2. `ClassName[] objArray;`
- Or
3. `ClassName objeArray[];`

Suppose, we have created a class named Employee. We want to keep records of 20 employees of a company having three departments. In this case, we will not create 20 separate variables. Instead of this, we will create an array of objects, as follows.

4. `Employee department1[20];`
5. `Employee department2[20];`
6. `Employee department3[20];`

The above statements create an array of objects with 20 elements.

Let's create an array of objects in a [Java program](#).

In the following program, we have created a class named Product and initialized an array of objects using the constructor. We have created a constructor of the class Product that contains product id and product name. In the main function, we have created individual objects of the class Product. After that, we have passed initial values to each of the objects using the constructor.

ArrayOfObjects.java

```
7.  public class ArrayOfObjects
8.  {
9.      public static void main(String args[])
10. {
11.     //create an array of product object
12.     Product[] obj = new Product[5] ;
13.     //create & initialize actual product objects using constructor
14.     obj[0] = new Product(23907,"Dell Laptop");
15.     obj[1] = new Product(91240,"HP 630");
16.     obj[2] = new Product(29823,"LG OLED TV");
17.     obj[3] = new Product(11908,"MI Note Pro Max 9");
18.     obj[4] = new Product(43590,"Kingston USB");
19.     //display the product object data
20.     System.out.println("Product Object 1:");
21.     obj[0].display();
22.     System.out.println("Product Object 2:");
23.     obj[1].display();
24.     System.out.println("Product Object 3:");
25.     obj[2].display();
26.     System.out.println("Product Object 4:");
27.     obj[3].display();
28.     System.out.println("Product Object 5:");
29.     obj[4].display();
30. }
31. }
32. //Product class with product Id and product name as attributes
33. class Product
34. {
```

```
35. int pro_Id;
36. String pro_name;
37. //Product class constructor
38. Product(int pid, String n)
39. {
40.     pro_Id = pid;
41.     pro_name = n;
42. }
43. public void display()
44. {
45.     System.out.print("Product Id = "+pro_Id + " " + " Product Name = "+pro_name);
46.     System.out.println();
47. }
48. }
```

Output:

```
Product Object 1:
Product Id = 23907    Product Name = Dell Laptop
Product Object 2:
Product Id = 91240    Product Name = HP 630
Product Object 3:
Product Id = 29823    Product Name = LG OLED TV
Product Object 4:
Product Id = 11908    Product Name = MI Note Pro Max 9
Product Object 5:
Product Id = 43590    Product Name = Kingston USB
```

Constructors in Java

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

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

When is a constructor called

Every time an object is created using new() keyword, at least one constructor is called. It calls a default constructor.

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

Rules for creating Java constructor

There are two rules defined for the constructor.

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

Note: We can use access modifiers while declaring a constructor. It controls the object creation. In other words, we can have private, protected, public or default constructor in Java.

Types of Java constructors

There are two types of constructors in Java:

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

Java Default Constructor

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

Syntax of default constructor:

1. `<class_name>(){}`

Example of default constructor

In this example, we are creating the no-arg constructor in the Bike class. It will be invoked at the time of object creation.

1. `//Java Program to create and call a default constructor`
2. `class Bike1 {`
3. `//creating a default constructor`
4. `Bike1(){System.out.println("Bike is created");}`

```

5. //main method
6. public static void main(String args[]){
7. //calling a default constructor
8. Bike1 b=new Bike1();
9. }
10. }

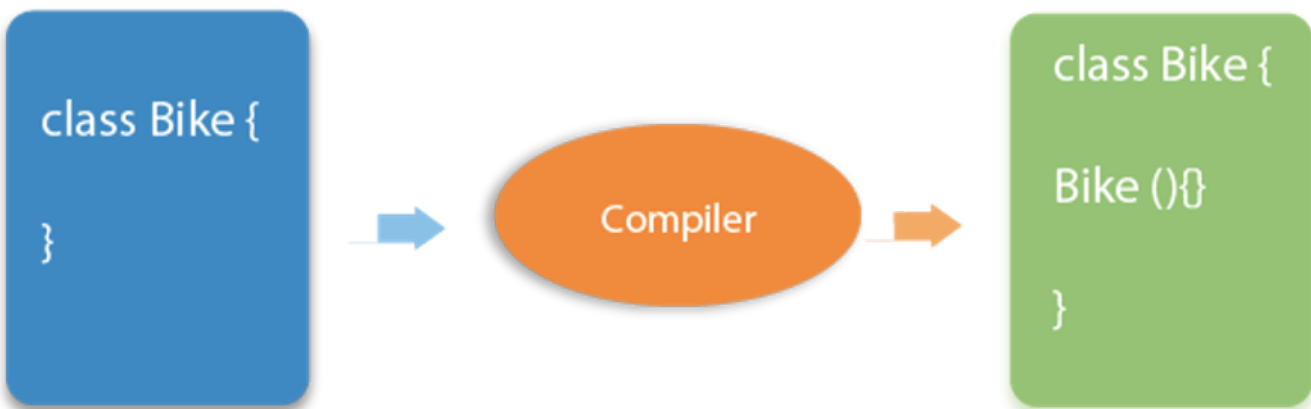
```

[Test it Now](#)

Output:

Bike is created

Rule: If there is no constructor in a class, compiler automatically creates a default constructor.



What is the purpose of a default constructor?

The default constructor is used to provide the default values to the object like 0, null, etc., depending on the type.

Example of default constructor that displays the default values

```

1. //Let us see another example of default constructor
2. //which displays the default values
3. class Student3{
4. int id;
5. String name;
6. //method to display the value of id and name
7. void display(){System.out.println(id+" "+name);}
8.
9. public static void main(String args[]){
10. //creating objects
11. Student3 s1=new Student3();
12. Student3 s2=new Student3();
13. //displaying values of the object
14. s1.display();
15. s2.display();
16. }
17. }
18.

```

Output:

```
0 null
0 null
```

Explanation: In the above class, you are not creating any constructor so compiler provides you a default constructor. Here 0 and null values are provided by default constructor.

Java Parameterized Constructor

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

Why use the parameterized constructor?

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

Example of parameterized constructor

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

1. //Java Program to demonstrate the use of parameterized constructor

```
class Student4{

    int id;

    String name;

    //creating a parameterized constructor

    Student4(int i,String n){

        id = i;

        name = n;

    }

    //method to display the values

    void display(){System.out.println(id+" "+name);}


    public static void main(String args[]){

        //creating objects and passing values

        Student4 s1 = new Student4(111,"Karan");

        Student4 s2 = new Student4(222,"Aryan");
```

```
//calling method to display the values of object

s1.display();

s2.display();

}

}
```

[Test it Now](#)

Output:

```
111 Karan
222 Aryan
```

Constructor Overloading in Java

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

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

Example of Constructor Overloading

1. //Java program to overload constructors in java

```
class Student5{

    int id;

    String name;

    int age;

    //creating two arg constructor

    Student5(int i,String n){

        id = i;

        name = n;

    }

    //creating three arg constructor

    Student5(int i,String n,int a){

        id = i;

        name = n;

        age=a;

    }

}
```

```

    }

    void display(){System.out.println(id+" "+name+" "+age);}

    public static void main(String args[]){
        Student5 s1 = new Student5(111,"Karan");
        Student5 s2 = new Student5(222,"Aryan",25);
        s1.display();
        s2.display();
    }
}

```

[Test it Now](#)

Output:

```

111 Karan 0
222 Aryan 25

```

Difference between constructor and method in Java

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

Java Constructor

- A constructor is used to initialize the state of an object.
- A constructor must not have a return type.
- The constructor is invoked implicitly.
- The Java compiler provides a default constructor if you don't have any constructor in a class.
- The constructor name must be same as the class name.

Java Method

- A method is used to expose the behaviour of an object.
- A method must have a return type.
- The method is invoked explicitly.
- The method is not provided by the compiler in any case.
- The method name may or may not be same as class name.

Java Copy Constructor

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

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

- By constructor
- By assigning the values of one object into another
- By clone() method of Object class

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

1. //Java program to initialize the values from one object to another

```
class Student6{
    int id;
    String name;
    //constructor to initialize integer and string
    Student6(int i,String n){
        id = i;
        name = n;
    }
    //constructor to initialize another object
    Student6(Student6 s){
        id = s.id;
        name =s.name;
    }
    void display(){System.out.println(id+" "+name);}

    public static void main(String args[]){
        Student6 s1 = new Student6(111,"Karan");
        Student6 s2 = new Student6(s1);
        s1.display();
        s2.display();
    }
}
```

[Test it Now](#)

Output:

```
111 Karan
111 Karan
```

Copying values without constructor

Constructors in Java

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

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

When is a constructor called

Every time an object is created using new() keyword, at least one constructor is called. It calls a default constructor.

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

Rules for creating Java constructor

There are two rules defined for the constructor.

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

Note: We can use access modifiers while declaring a constructor. It controls the object creation. In other words, we can have private, protected, public or default constructor in Java.

Types of Java constructors

There are two types of constructors in Java:

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

Java Default Constructor

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

Syntax of default constructor:

1. `<class_name>(){}`

Example of default constructor

In this example, we are creating the no-arg constructor in the Bike class. It will be invoked at the time of object creation.

1. `//Java Program to create and call a default constructor`

```

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

```

[Test it Now](#)

Output:

Bike is created

Rule: If there is no constructor in a class, compiler automatically creates a default constructor.

What is the purpose of a default constructor?

The default constructor is used to provide the default values to the object like 0, null, etc., depending on the type.

Example of default constructor that displays the default values

```

1. //Let us see another example of default constructor
2. //which displays the default values
3. class Student3 {
4. int id;
5. String name;
6. //method to display the value of id and name
7. void display(){System.out.println(id+" "+name);}
8.
9. public static void main(String args[]){
10. //creating objects
11. Student3 s1=new Student3();
12. Student3 s2=new Student3();
13. //displaying values of the object
14. s1.display();
15. s2.display();
16. }
17. }
18.

```

Output:

```

0 null
0 null

```

Explanation: In the above class, you are not creating any constructor so compiler provides you a default constructor. Here 0 and null values are provided by default constructor.

Java Parameterized Constructor

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

Why use the parameterized constructor?

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

Example of parameterized constructor

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

1. //Java Program to demonstrate the use of parameterized constructor

```
class Student4{  
  
    int id;  
  
    String name;  
  
    //creating a parameterized constructor  
  
    Student4(int i,String n){  
  
        id = i;  
  
        name = n;  
  
    }  
  
    //method to display the values  
  
    void display(){System.out.println(id+" "+name);}  
  
  
  
  
    public static void main(String args[]){  
  
        //creating objects and passing values  
  
        Student4 s1 = new Student4(111,"Karan");  
  
        Student4 s2 = new Student4(222,"Aryan");  
  
        //calling method to display the values of object  
  
        s1.display();  
  
        s2.display();  
  
    }  
  
}
```

[Test it Now](#)

Output:

```
111 Karan  
222 Aryan
```

Constructor Overloading in Java

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

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

Example of Constructor Overloading

1. //Java program to overload constructors in java

```
class Student5 {  
    int id;  
    String name;  
    int age;  
    //creating two arg constructor  
    Student5(int i,String n){  
        id = i;  
        name = n;  
    }  
    //creating three arg constructor  
    Student5(int i,String n,int a){  
        id = i;  
        name = n;  
        age=a;  
    }  
    void display(){System.out.println(id+" "+name+" "+age);}
```

```

public static void main(String args[]){

Student5 s1 = new Student5(111,"Karan");

Student5 s2 = new Student5(222,"Aryan",25);

s1.display();

s2.display();

}

}

```

[Test it Now](#)

Output:

```

111 Karan 0
222 Aryan 25

```

Difference between constructor and method in Java

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

Java Constructor

A constructor is used to initialize the state of an object.

A constructor must not have a return type.

The constructor is invoked implicitly.

The Java compiler provides a default constructor if you don't have any constructor in a class.

The constructor name must be same as the class name.

Java Method

A method is used to expose the behavior of an object.

A method must have a return type.

The method is invoked explicitly.

The method is not provided by the compiler in any case.

The method name may or may not be same as class name.

Java Copy Constructor

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

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

- By constructor
- By assigning the values of one object into another
- By clone() method of Object class

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

1. //Java program to initialize the values from one object to another

```
class Student6{

    int id;

    String name;

    //constructor to initialize integer and string

    Student6(int i,String n){

        id = i;

        name = n;

    }

    //constructor to initialize another object

    Student6(Student6 s){

        id = s.id;

        name =s.name;

    }

    void display(){System.out.println(id+" "+name);}


    public static void main(String args[]){

        Student6 s1 = new Student6(111,"Karan");

        Student6 s2 = new Student6(s1);

        s1.display();

        s2.display();

    }

}
```

[Test it Now](#)

Output:

```
111 Karan
111 Karan
```

Copying values without constructor

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

```
class Student7{

    int id;

    String name;

    Student7(int i,String n){

        id = i;

        name = n;

    }

    Student7(){

    }

    void display(){System.out.println(id+" "+name);}

}

public static void main(String args[]){

    Student7 s1 = new Student7(111,"Karan");

    Student7 s2 = new Student7();

    s2.id=s1.id;

    s2.name=s1.name;

    s1.display();

    s2.display();

}

}
```

[Test it Now](#)

Output:

```
111 Karan
111 Karan
```

Q) Does constructor return any value?

Yes, it is the current class instance (You cannot use return type yet it returns a value).

Can constructor perform other tasks instead of initialization?

Yes, like object creation, starting a thread, calling a method, etc. You can perform any operation in the constructor as you perform in the method.

Method Overloading in Java

If a class has multiple methods having same name but different in 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 behavior 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 only.

1) Method Overloading: changing no. of arguments

In this example, we have created two methods, first `add()` method performs addition of two numbers and second `add` method performs addition of three numbers.

In this example, we are creating static methods so that we don't need to create instance for calling methods.


```
class Adder

{

static int add(int a,int b){return a+b;}

static int add(int a,int b,int c){return a+b+c;}

}

class TestOverloading1 {

public static void main(String[] args){

System.out.println(Adder.add(11,11));

System.out.println(Adder.add(11,11,11));

}}
```

[Test it Now](#)

Output:

22
33

2) Method Overloading: changing data type of arguments

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

```
class Adder{

static int add(int a, int b){return a+b;}

static double add(double a, double b){return a+b;}

}

class TestOverloading2 {

public static void main(String[] args){

System.out.println(Adder.add(11,11));

System.out.println(Adder.add(12.3,12.6));

}}
```

[Test it Now](#)

Output:

22
24.9

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

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

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

[Test it Now](#)

Output:

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

System.out.println(Adder.add(11,11)); //Here, how can java determine which sum() method should be called?

Note: Compile Time Error is better than Run Time Error. So, java compiler renders compiler time error if you declare the same method having same parameters.

Can we overload java main() method?

Yes, by method overloading. You can have any number of main methods in a class by method overloading. But JVM calls main() method which receives string array as arguments only. Let's see the simple example:

1. `class TestOverloading4{`
2. `public static void main(String[] args){System.out.println("main with String[]");}`
3. `public static void main(String args){System.out.println("main with String");}`
4. `public static void main(){System.out.println("main without args");}`
5. `}`

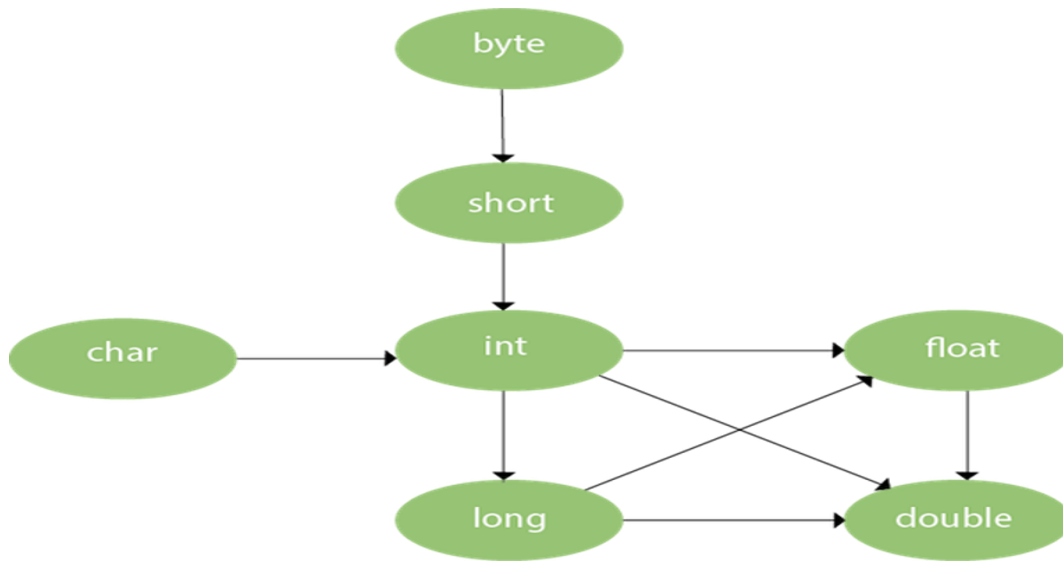
[Test it Now](#)

Output:

```
main with String[]
```

Method Overloading and Type Promotion

One type is promoted to another implicitly if no matching datatype is found. Let's understand the concept by the figure given below:



As displayed in the above diagram, byte can be promoted to short, int, long, float or double. The short datatype can be promoted to int, long, float or double. The char datatype can be promoted to int, long, float or double and so on.

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

Example of Method Overloading with Type Promotion if matching found

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

```
class OverloadingCalculation2 {  
  
    void sum(int a,int b){System.out.println("int arg method invoked");}  
  
    void sum(long a,long b){System.out.println("long arg method invoked");}  
  
  
    public static void main(String args[]){  
  
        OverloadingCalculation2 obj=new OverloadingCalculation2();  
  
        obj.sum(20,20);//now int arg sum() method gets invoked  
  
    }  
  
}
```

[Test it Now](#)

Output:intarg method invoked

Example of Method Overloading with Type Promotion in case of ambiguity

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

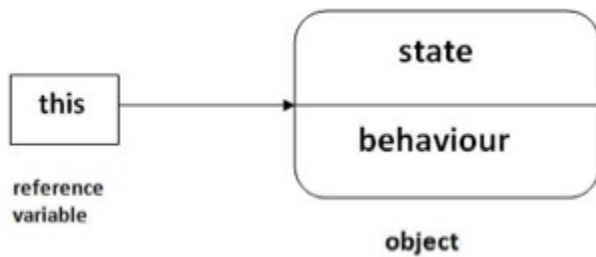
```
class OverloadingCalculation3 {  
  
    void sum(int a,long b){System.out.println("a method invoked");}  
  
    void sum(long a,int b){System.out.println("b method invoked");}  
  
  
    public static void main(String args[]){  
  
        OverloadingCalculation3 obj=new OverloadingCalculation3();  
  
        obj.sum(20,20);//now ambiguity  
  
    }  
  
}
```

[Test it Now](#)

Output:Compile Time Error

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.



Classname reference variable= new classname();

Usage of java this keyword

Here is given the 6 usage of java this keyword.

1. this can be used to refer current class instance variable.
2. this can be used to invoke current class method (implicitly)
3. this() can be used to invoke the current class constructor.
4. this can be passed as an argument in the method call.
5. this can be passed as argument in the constructor call.
6. this can be used to return the current class instance from the method.

Suggestion: If you are beginner to java, lookup only three usage of this keyword.

1) this: to refer current class instance variable

The this keyword can be used to refer current class instance variable. If there is ambiguity between the instance variables and parameters, 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** Student{
2. **int** rollno;
3. String name;
4. **float** fee;

```

5. Student(int rollNo,String name,float fee){
6. rollNo=rollNo;
7. name=name;
8. fee=fee;
9. }
10. void display(){System.out.println(rollNo+" "+name+" "+fee);}
11. }
12. class TestThis1{
13. public static void main(String args[]){
14. Student s1=new Student(111,"ankit",5000f);
15. Student s2=new Student(112,"sumit",6000f);
16. s1.display();
17. s2.display();
18. }}

```

Test it Now

Output:

```

0 null 0.0
0 null 0.0

```

In the above example, parameters (formal arguments) and instance variables are same. So, we are using this keyword to distinguish local variable and instance variable.

Solution of the above problem by this keyword

```

class Student{
int rollNo;
String name;
float fee;
Student(int rollNo,String name,float fee){
this.rollNo=rollNo;
this.name=name;
this.fee=fee;
}
void display(){System.out.println(rollNo+" "+name+" "+fee);}
}

```

```

class TestThis2{
public static void main(String args[]){
Student s1=new Student(111,"ankit",5000f);
Student s2=new Student(112,"sumit",6000f);
s1.display();
s2.display();
}
}

```

Test it Now

Output:

```

111 ankit 5000
112 sumit 6000

```

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 Student{
2.   int rollNo;
3.   String name;
4.   float fee;
5.   Student(int r,String n,float f){
6.     rollNo=r;
7.     name=n;
8.     fee=f;
9.   }
10. void display(){System.out.println(rollNo+" "+name+" "+fee);}
11. }
12.
13. class TestThis3{
14.   public static void main(String args[]){
15.     Student s1=new Student(111,"ankit",5000f);
16.     Student s2=new Student(112,"sumit",6000f);
17.     s1.display();
18.     s2.display();
19.   }}
```

Test it Now

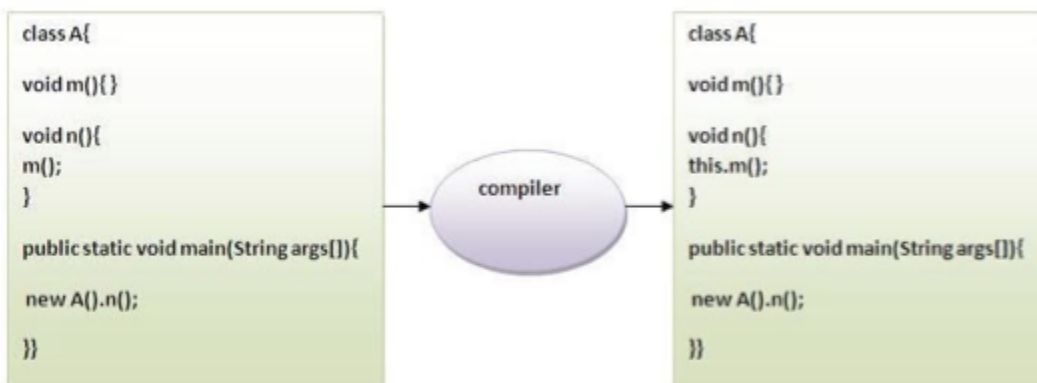
Output:

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

It is better approach to use meaningful names for variables. So we use same name for instance variables and parameters in real time, and always use this keyword.

2) this: to invoke current class method

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



```

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

```

Test it Now

Output:

```

hello n
hello m

```

3) this() : to invoke current class constructor

The this() constructor call can be used to invoke the current class constructor. It is used to reuse the constructor. In other words, it is used for constructor chaining.

Calling default constructor from parameterized constructor:

```

class A{
{System.out.println("hello a");}
A(int x){
this();
System.out.println(x);
}
}

```

```

class TestThis5{
public static void main(String args[]){
=new A(10);
}
}

```

Test it Now

Output:

```

hello a
10

```

Calling parameterized constructor from default constructor:

1. **class** A{
2. A(){
3. **this**(5);
4. System.out.println("hello a");
5. }
6. A(int x){


```

7. System.out.println(x);
8. }
9. }
10. class TestThis6{
11.
12. public static void main(String args[]){
13. A a=new A();
14. }}

```

Test it Now

Output:

```

5
hello a

```

Real usage of this() constructor call

The this() constructor call should be used to reuse the constructor from the constructor. It maintains the chain between the constructors i.e. it is used for constructor chaining. Let's see the example given below that displays the actual use of this keyword.

```

class Student{
int rollno;
String name,course;
float fee;
Student(int rollno,String name,String course){
this.rollno=rollno;
this.name=name;
this.course=course;
}
Student(int rollno,String name,String course,float fee){
this(rollno,name,course);//reusing constructor
this.fee=fee;
}
void display(){System.out.println(rollno+" "+name+" "+course+" "+fee);}
}
class TestThis7{
public static void main(String args[]){
Student s1=new Student(111,"ankit","java");
Student s2=new Student(112,"sumit","java",6000f);
s1.display();
s2.display();
}}

```

Test it Now

Output:

```

111 ankit java null
112 sumit java 6000

```

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

```

1. class Student{
2. int rollno;
3. String name,course;
4. float fee;

```

```

5. Student(int rollNo,String name,String course){
6.     this.rollNo=rollNo;
7.     this.name=name;
8.     this.course=course;
9. }
10. Student(int rollNo,String name,String course,float fee){
11.     this.fee=fee;
12.     this(rollNo,name,course);//C.T.Error
13. }
14. void display(){System.out.println(rollNo+" "+name+" "+course+" "+fee);}
15. }
16. class TestThis8{
17.     public static void main(String args[]){
18.         Student s1=new Student(111,"ankit","java");
19.         Student s2=new Student(112,"sumit","java",6000f);
20.         s1.display();
21.         s2.display();
22.     }}

```

Test it Now

Compile Time Error: Call to this must be first statement in constructor

4) this: to pass as an argument in the method

The this keyword can also be passed as an argument in the method. It is mainly used in the event handling. Let's see the example:

```

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

```

Test it Now

Output:

method is invoked

Application of this that can be passed as an argument:

In event handling (or) in a situation where we have to provide reference of a class to another one. It is used to reuse one object in many methods.

5) this: to pass as argument in the constructor call

We can pass the this keyword in the constructor also. It is useful if we have to use one object in multiple classes. Let's see the example:

```
class B{
    A4 obj;
    B(A4 obj){
        this.obj=obj;
    }
    void display(){
        System.out.println(obj.data);//using data member of A4 class
    }
}

class A4{
    int data=10;
    A4(){
        B b=new B(this);
        b.display();
    }
    public static void main(String args[]){
        A4 a=new A4();
    }
}
```

Test it Now

Output:10

6) this keyword can be used to return current class instance

We can return this keyword as a statement from the method. In such case, return type of the method must be the class type (non-primitive). Let's see the example:

Syntax of this that can be returned as a statement

1. return_type method_name(){
2. **return this;**
3. }

Example of this keyword that you return as a statement from the method

```
class A{
    A getA(){
        return this;
    }
}
```

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

Test it Now

Output:

```
Hello java
```

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.

1. **class** A5{
2. **void** m(){
3. System.out.println(**this**); //prints same reference ID
4. }
5. **public static void** main(String args[]){
6. A5 obj=**new** A5();
7. System.out.println(obj); //prints the reference ID
8. obj.m();
9. }
10. }

Test it Now

Output:

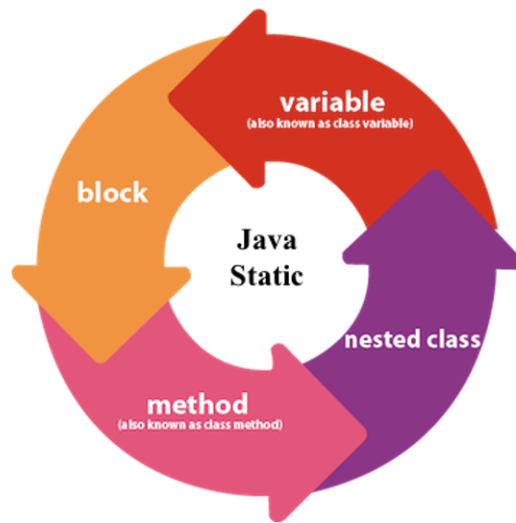
```
A5@22b3ea59  
A5@22b3ea59
```

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 a class variable)
2. Method (also known as a class method)
3. Block
4. Nested class



1) Java static variable

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

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

Advantages of static variable

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

Understanding the problem without static variable

1. **class** Student
2. {
3. **int** rollno;
4. String name;

```
5.     String college="ITS";
6. }
```

Suppose there are 500 students in my college, now all instance data members will get memory each time when the object is created. All students have its unique rollno and name, so instance data member is good in such case. Here, "college" refers to the common property of all objects. If we make it static, this field will get the memory only once.

Java static property is shared to all objects.

Example of static variable

```
1. //Java Program to demonstrate the use of static variable
2. class Student{
3.     int rollno;//instance variable
4.     String name;
5.     static String college ="ITS";//static variable
6.     //constructor
7.     Student(int r, String n){
8.         rollno = r;
9.         name = n;
10.    }
11.    //method to display the values
12.    void display (){System.out.println(rollno+" "+name+" "+college);}
13.}
14.//Test class to show the values of objects
15. public class TestStaticVariable1{
16.     public static void main(String args[]){
17.         Student s1 = new Student(111,"Karan");
18.         Student s2 = new Student(222,"Aryan");
19.         //we can change the college of all objects by the single line of code
20.         //Student.college="BBDIT";
21.         s1.display();
22.         s2.display();
23.     }
24. }
```

Test it Now

Output:

```
111 Karan ITS
222 Aryan ITS
```

Program of the counter without static variable

In this example, we have created an instance variable named count which is incremented in the constructor. Since instance variable gets the memory at the time of object creation, each object will have the copy of the instance variable. If it is incremented, it won't reflect other objects. So each object will have the value 1 in the count variable.

1. //Java Program to demonstrate the use of an instance variable
2. //which get memory each time when we create an object of the class.

```
class Counter{  
int count=0;//will get memory each time when the instance is created
```

```
Counter()  
{  
count++; //incrementing value  
System.out.println(count);  
}
```

```
public static void main(String args[]){  
//Creating objects  
Counter c1=new Counter();  
Counter c2=new Counter();  
Counter c3=new Counter();  
}  
}
```

Test it Now

Output:

```
1  
1  
1
```

Program of counter by static variable

As we have mentioned above, static variable will get the memory only once, if any object changes the value of the static variable, it will retain its value.

1. //Java Program to illustrate the use of static variable which
2. //is shared with all objects.

```
class Counter2{  
static int count=0;//will get memory only once and retain its value
```

```
Counter2(){  
count++; //incrementing the value of static variable  
System.out.println(count);
```

```
}
```

```
public static void main(String args[]){
```

```
//creating objects
```

```
Counter2 c1=new Counter2();
```

```
Counter2 c2=new Counter2();
```

```
Counter2 c3=new Counter2();
```

```
}
```

```
}
```

Test it Now

Output:

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

Example of static method

```
//Java Program to demonstrate the use of a static method.
```

```
class Student{
```

```
    int rollno;
```

```
    String name;
```

```
    static String college = "ITS";
```

```
//static method to change the value of static variable
```

```
    static void change(){
```

```
        college = "BBDIT";
```

```
    }
```

```
//constructor to initialize the variable
```

```
    Student(int r, String n){
```

```
        rollno = r;
```

```
        name = n;
```

```
    }
```

```
//method to display values
```

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

```
}
```

```
//Test class to create and display the values of object
```

```
public class TestStaticMethod{
```

```
    public static void main(String args[]){
```



```
Student.change();//calling change method
```

```
//creating objects
```

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

```
Student s2 = new Student(222,"Aryan");
```

```
Student s3 = new Student(333,"Sonoo");
```

```
//calling display method
```

```
s1.display();
```

```
s2.display();
```

```
s3.display();
```

```
}
```

```
}
```

Test it Now

```
Output:111 Karan BBDIT
```

```
222 Aryan BBDIT
```

```
333 Sonoo BBDIT
```

Another example of a static method that performs a normal calculation

```
1. //Java Program to get the cube of a given number using the static method
```

```
2.
```

```
3. class Calculate{
```

```
4. static int cube(int x){
```

```
5. return x*x*x;
```

```
6. }
```

```
7.
```

```
8. public static void main(String args[]){
```

```
9. int result=Calculate.cube(5);
```

```
10. System.out.println(result);
```

```
11. }
```

```
12. }
```

Test it Now

```
Output:125
```

Restrictions for the 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 is the Java main method static?

Ans) It is because the object is not required to call a static method. If it were a non-static method, JVM creates an 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 the main method at the time of classloading.

Example of static block

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

Test it Now

Output:static block is invoked
Hello main

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

Ans) No, one of the ways was the static block, but it was possible till JDK 1.6. Since JDK 1.7, it is not possible to execute a java class without the main method.

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

Since JDK 1.7 and above, output would be:

Error: Main method not found in class A3, please define the main method as:
public static void main(String[] args)
or a JavaFX application class must extend javafx.application.Application

Access Modifiers in java

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

The access modifiers in java specifies accessibility(scope) of a data member, method, constructor or class.

There are 4 types of java access modifiers:

1. private
2. default
3. protected
4. public

1) private access modifier

The private access modifier is accessible only within class.

Simple example of private access modifier

In this example, we have created two classes A and Simple. A class contains private data member and private method. We are accessing these private members from outside the class, so there is compile time error.

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

Role of Private Constructor

If you make any class constructor private, you cannot create the instance of that class from outside the class. For example:

```
class A
{
    private A()
    {}//private constructor
    void msg(){System.out.println("Hello java");}
}

public class Simple{
    public static void main(String args[])
    {
        A obj=new A();//Compile Time Error
    }
}
```

2) default access modifier

If you don't use any modifier, it is treated as **default** by default. The default modifier is accessible only within package.

Example of default access modifier

In this example, we have created two packages pack and mypack. We are accessing the A class from outside its package, since A class is not public, so it cannot be accessed from outside the package.

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

//save by B.java
package mypack;
import pack.*;
class B{
    public static void main(String args[]){
        A obj = new A();//Compile Time Error
        obj.msg();//Compile Time Error
    }
}
```

In the above example, the scope of class A and its method msg() is default so it cannot be accessed from outside the package.

3) protected access modifier

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

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

Example of protected access modifier

In this example, we have created the two packages pack and mypack. The A class of pack package is public, so can be accessed from outside the package. But msg method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

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

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

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

Output:Hello

4) public access modifier

The **public access modifier** is accessible everywhere. It has the widest scope among all other modifiers.

Example of public access modifier

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

//save by B.java
package mypack;
import pack.*;
class B
{
    public static void main(String args[]){
        A obj = new A();
        obj.msg();
    }
}
```

Output:Hello

Understanding all java access modifiers

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

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

Java access modifiers with method overriding

If you are overriding any method, overridden method (i.e. declared in subclass) must not be more restrictive.

```
1. class A{
2. protected void msg(){System.out.println("Hello java");}
3. }
4.
5. public class Simple extends A{
6. void msg(){System.out.println("Hello java");} //C.T.Error
7. public static void main(String args[]){
8.     Simple obj=new Simple();
9.     obj.msg();
10. }
11. }
```

The default modifier is more restrictive than protected. That is why there is compile time error

Inheritance can be defined as the process where one class acquires the properties (methods and fields) of another. With the use of inheritance the information is made manageable in a hierarchical order.

The class which inherits the properties of other is known as subclass (derived class, child class) and the class whose properties are inherited is known as superclass (base class, parent class).

extends Keyword

extends is the keyword used to inherit the properties of a class. Following is the syntax of extends keyword.

Syntax

```
class Super {
    .....
    .....
}
class Sub extends Super {
    .....
    .....
}
```

Sample Code

Following is an example demonstrating Java inheritance. In this example, you can observe two classes namely Calculation and My_Calculation.

Using extends keyword, the My_Calculation inherits the methods addition() and Subtraction() of Calculation class.

Copy and paste the following program in a file with name My_Calculation.java

Example

[Live Demo](#)

```
class Calculation {
    int z;

    public void addition(int x, int y) {
        z = x + y;
        System.out.println("The sum of the given numbers:"+z);
    }

    public void Subtraction(int x, int y) {
        z = x - y;
        System.out.println("The difference between the given numbers:"+z);
    }
}

public class My_Calculation extends Calculation {
    public void multiplication(int x, int y) {
        z = x * y;
        System.out.println("The product of the given numbers:"+z);
    }

    public static void main(String args[]) {
        int a = 20, b = 10;
```



```

    My_Calculation demo = new My_Calculation();
    demo.addition(a, b);
    demo.Subtraction(a, b);
    demo.multiplication(a, b);
}
}

```

Compile and execute the above code as shown below.

```

javac My_Calculation.java
java My_Calculation

```

After executing the program, it will produce the following result –

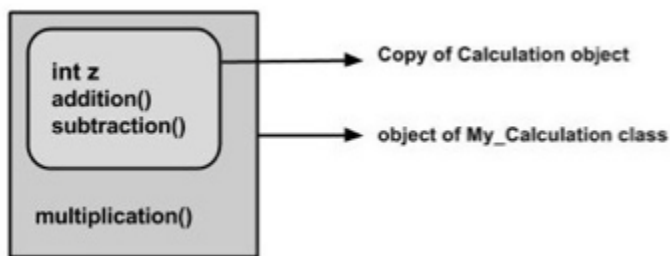
Output

```

The sum of the given numbers:30
The difference between the given numbers:10
The product of the given numbers:200

```

In the given program, when an object to **My_Calculation** class is created, a copy of the contents of the superclass is made within it. That is why, using the object of the subclass you can access the members of a superclass.



The Superclass reference variable can hold the subclass object, but using that variable you can access only the members of the superclass, so to access the members of both classes it is recommended to always create reference variable to the subclass.

If you consider the above program, you can instantiate the class as given below. But using the superclass reference variable (**cal** in this case) you cannot call the method **multiplication()**, which belongs to the subclass **My_Calculation**.

```

Calculation demo = new My_Calculation();
demo.addition(a, b);
demo.Subtraction(a, b);

```

Note – A subclass inherits all the members (fields, methods, and nested classes) from its superclass. Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.

The super keyword

The **super** keyword is similar to **this** keyword. Following are the scenarios where the super keyword is used.

- It is used to **differentiate the members** of superclass from the members of subclass, if they have same names.
- It is used to **invoke the superclass** constructor from subclass.

Differentiating the Members

If a class is inheriting the properties of another class. And if the members of the superclass have the names same as the sub class, to differentiate these variables we use super keyword as shown below.

```
super.variable  
super.method();
```

Sample Code

This section provides you a program that demonstrates the usage of the **super** keyword.

In the given program, you have two classes namely *Sub_class* and *Super_class*, both have a method named display() with different implementations, and a variable named num with different values. We are invoking display() method of both classes and printing the value of the variable num of both classes. Here you can observe that we have used super keyword to differentiate the members of superclass from subclass.

Copy and paste the program in a file with name Sub_class.java.

Example

[Live Demo](#)

```
class Super_class {  
    int num = 20;  
  
    // display method of superclass  
    public void display() {  
        System.out.println("This is the display method of superclass");  
    }  
}  
  
public class Sub_class extends Super_class {  
    int num = 10;  
  
    // display method of sub class  
    public void display() {  
        System.out.println("This is the display method of subclass");  
    }  
  
    public void my_method() {  
        // Instantiating subclass  
        Sub_class sub = new Sub_class();  
  
        // Invoking the display() method of sub class  
        sub.display();  
  
        // Invoking the display() method of superclass  
        super.display();  
  
        // printing the value of variable num of subclass  
        System.out.println("value of the variable named num in sub class:"+ sub.num);  
  
        // printing the value of variable num of superclass  
        System.out.println("value of the variable named num in super class:"+ super.num);  
    }  
  
    public static void main(String args[]) {
```

```

        Sub_class obj = new Sub_class();
        obj.my_method();
    }
}

```

Compile and execute the above code using the following syntax.

```

javac Super_Demo
java Super

```

On executing the program, you will get the following result –

Output

```

This is the display method of subclass
This is the display method of superclass
value of the variable named num in sub class:10
value of the variable named num in super class:20

```

Invoking Superclass Constructor

If a class is inheriting the properties of another class, the subclass automatically acquires the default constructor of the superclass. But if you want to call a parameterized constructor of the superclass, you need to use the super keyword as shown below.

```

super(values);

```

Sample Code

The program given in this section demonstrates how to use the super keyword to invoke the parametrized constructor of the superclass. This program contains a superclass and a subclass, where the superclass contains a parameterized constructor which accepts a integer value, and we used the super keyword to invoke the parameterized constructor of the superclass.

Copy and paste the following program in a file with the name Subclass.java

Example

[Live Demo](#)

```

class Superclass {
    int age;

    Superclass(int age) {
        this.age = age;
    }

    public void getAge() {
        System.out.println("The value of the variable named age in super class is: " +age);
    }
}

public class Subclass extends Superclass {
    Subclass(int age) {
        super(age);
    }

    public static void main(String argd[]) {

```

```
        Subclass s = new Subclass(24);  
        s.getAge();  
    }  
}
```

Compile and execute the above code using the following syntax.

```
javac Subclass  
java Subclass
```

On executing the program, you will get the following result –

Output

The value of the variable named age in super class is: 24

IS-A Relationship

IS-A is a way of saying: This object is a type of that object. Let us see how the **extends** keyword is used to achieve inheritance.

```
public class Animal {  
}  
  
public class Mammal extends Animal {  
}  
  
public class Reptile extends Animal {  
}  
  
public class Dog extends Mammal {  
}
```

Now, based on the above example, in Object-Oriented terms, the following are true –

- Animal is the superclass of Mammal class.
- Animal is the superclass of Reptile class.
- Mammal and Reptile are subclasses of Animal class.
- Dog is the subclass of both Mammal and Animal classes.

Now, if we consider the IS-A relationship, we can say –

- Mammal IS-A Animal
- Reptile IS-A Animal
- Dog IS-A Mammal
- Hence: Dog IS-A Animal as well

With the use of the extends keyword, the subclasses will be able to inherit all the properties of the superclass except for the private properties of the superclass.

We can assure that Mammal is actually an Animal with the use of the instance operator.

Example

[Live Demo](#)

```
class Animal {  
}  
  
class Mammal extends Animal {
```

```

}

class Reptile extends Animal {
}

public class Dog extends Mammal {

    public static void main(String args[]) {
        Animal a = new Animal();
        Mammal m = new Mammal();
        Dog d = new Dog();

        System.out.println(m instanceof Animal);
        System.out.println(d instanceof Mammal);
        System.out.println(d instanceof Animal);
    }
}

```

This will produce the following result –

Output

```

true
true
true

```

Since we have a good understanding of the **extends** keyword, let us look into how the **implements** keyword is used to get the IS-A relationship.

Generally, the **implements** keyword is used with classes to inherit the properties of an interface. Interfaces can never be extended by a class.

Example

```

public interface Animal {
}

public class Mammal implements Animal {
}

public class Dog extends Mammal {
}

```

The instanceof Keyword

Let us use the **instanceof** operator to check determine whether Mammal is actually an Animal, and dog is actually an Animal.

Example

[Live Demo](#)

```

interface Animal{}
class Mammal implements Animal{}

public class Dog extends Mammal {

    public static void main(String args[]) {
        Mammal m = new Mammal();
        Dog d = new Dog();
    }
}

```

```
        System.out.println(m instanceof Animal);  
        System.out.println(d instanceof Mammal);  
        System.out.println(d instanceof Animal);  
    }  
}
```

This will produce the following result –

Output

```
true  
true  
true
```

HAS-A relationship

These relationships are mainly based on the usage. This determines whether a certain class **HAS-A** certain thing. This relationship helps to reduce duplication of code as well as bugs.

Lets look into an example –

Example

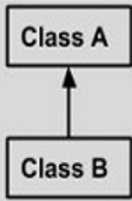
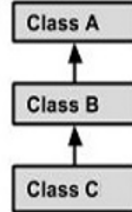
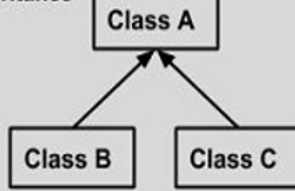
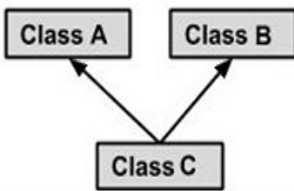
```
public class Vehicle{}  
public class Speed{}  
  
public class Van extends Vehicle {  
    private Speed sp;  
}
```

This shows that class Van HAS-A Speed. By having a separate class for Speed, we do not have to put the entire code that belongs to speed inside the Van class, which makes it possible to reuse the Speed class in multiple applications.

In Object-Oriented feature, the users do not need to bother about which object is doing the real work. To achieve this, the Van class hides the implementation details from the users of the Van class. So, basically what happens is the users would ask the Van class to do a certain action and the Van class will either do the work by itself or ask another class to perform the action.

Types of Inheritance

There are various types of inheritance as demonstrated below.

Single Inheritance  <pre> graph BT B[Class B] --> A[Class A] </pre>	<pre> public class A { } public class B extends A { } </pre>
Multi Level Inheritance  <pre> graph BT C[Class C] --> B[Class B] B --> A[Class A] </pre>	<pre> public class A { } public class B extends A { } public class C extends B { } </pre>
Hierarchical Inheritance  <pre> graph BT B[Class B] --> A[Class A] C[Class C] --> A </pre>	<pre> public class A { } public class B extends A { } public class C extends A { } </pre>
Multiple Inheritance  <pre> graph BT C[Class C] --> A[Class A] C --> B[Class B] </pre>	<pre> public class A { } public class B { } public class C extends A,B { } // Java does not support multiple inheritance </pre>

A very important fact to remember is that Java does not support multiple inheritance. This means that a class cannot extend more than one class. Therefore following is illegal –

Example

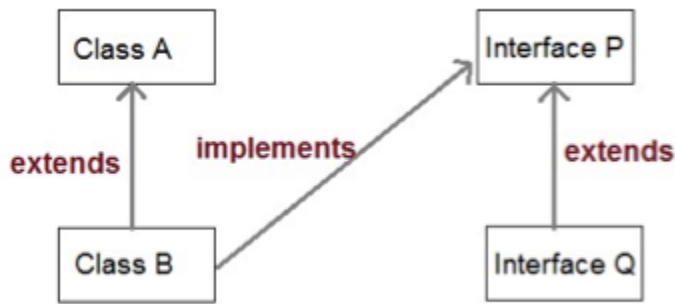
```
public class extends Animal, Mammal{}
```

However, a class can implement one or more interfaces, which has helped Java get rid of the impossibility of multiple inheritance.

Inheritance (IS-A)

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

Inheritance defines **is-a** relationship between a Super class and its Sub class. `extends` and `implements` keywords are used to describe inheritance in Java.



Let us see how **extends** keyword is used to achieve Inheritance.

```
class Vehicle.
{
    .....
}
class Car extends Vehicle
{
    ..... //extends the property of vehicle class.
}
```

Now based on above example. In OOPs term we can say that,

- **Vehicle** is super class of **Car**.
- **Car** is sub class of **Vehicle**.
- Car IS-A Vehicle.

Purpose of Inheritance

1. It promotes the code reusability i.e the same methods and variables which are defined in a parent/super/base class can be used in the child/sub/derived class.
2. It promotes polymorphism by allowing method overriding.

Disadvantages of Inheritance

Main disadvantage of using inheritance is that the two classes (parent and child class) get **tightly coupled**.

This means that if we change code of parent class, it will affect to all the child classes which are inheriting/deriving the parent class, and hence, **it cannot be independent of each other**.

Simple example of Inheritance

```
class Parent
{
    public void p1()
    {
        System.out.println("Parent method");
    }
}
```



```

public class Child extends Parent {
    public void c1()
    {
        System.out.println("Child method");
    }
    public static void main(String[] args)
    {
        Child cobj = new Child();
        cobj.c1();    //method of Child class
        cobj.p1();    //method of Parent class
    }
}

```

Child method Parent method

Another example of Inheritance

```

class Vehicle
{
    String vehicleType;
}
public class Car extends Vehicle {

    String modelType;
    public void showDetail()
    {
        vehicleType = "Car";    //accessing Vehicle class member
        modelType = "sports";
        System.out.println(modelType+" "+vehicleType);
    }
    public static void main(String[] args)
    {
        Car car =new Car();
        car.showDetail();
    }
}

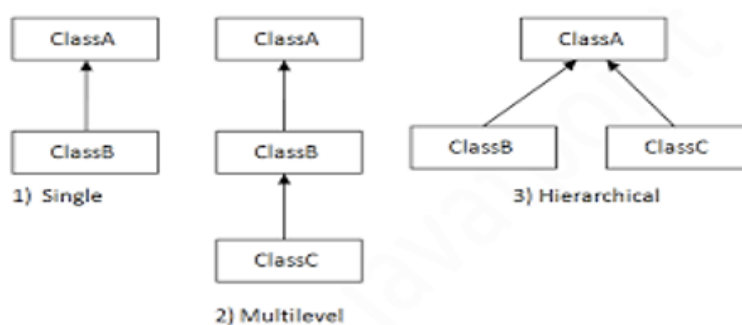
```

sports Car

Types of Inheritance

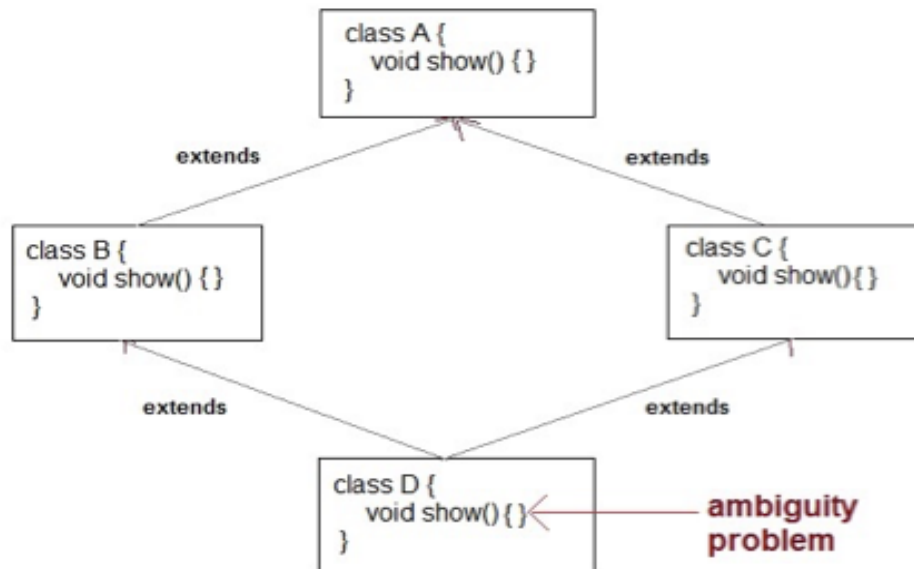
1. Single Inheritance
2. Multilevel Inheritance
3. Heirarchical Inheritance

NOTE :Multiple inheritance is not supported in java



Why multiple inheritance is not supported in Java

- To remove ambiguity.
- To provide more maintainable and clear design.



super keyword

In Java, `super` keyword is used to refer to immediate parent class of a child class. In other words **super** keyword is used by a subclass whenever it need to refer to its immediate super class.

```
class Parent
{
    String name;
}
class Child extends Parent {
    String name;
    void detail()
    {
        super.name = "Parent";
        name = "Child";
    }
}
```

Blue arrows indicate the relationship between the `super.name` access in the `detail()` method of the `Child` class and the `String name` attribute in the `Parent` class. A red oval highlights the `super.name` expression.

Example of Child class reffering Parent class property using super keyword

```
class Parent
{
    String name;
}

public class Child extends Parent {
    String name;
    public void details()
    {
        super.name = "Parent";    //refers to parent class member
        name = "Child";
        System.out.println(super.name+" and "+name);
    }
    public static void main(String[] args)
    {
        Child cobj = new Child();
        cobj.details();
    }
}
```

Parent and Child

Example of Child class refering Parent class methods using super keyword

```
class Parent
{
    String name;
    public void details()
    {
        name = "Parent";
        System.out.println(name);
    }
}

public class Child extends Parent {
    String name;
    public void details()
    {
        super.details();          //calling Parent class details() method
        name = "Child";
        System.out.println(name);
    }
    public static void main(String[] args)
    {
        Child cobj = new Child();
        cobj.details();
    }
}
```

Parent Child

Example of Child class calling Parent class constructor using super keyword

```
class Parent
{
    String name;

    public Parent(String n)
    {
        name = n;
    }
}

public class Child extends Parent {
    String name;

    public Child(String n1, String n2)
    {

        super(n1);          //passing argument to parent class constructor
        this.name = n2;
    }
    public void details()
    {
        System.out.println(super.name+" and "+name);
    }
    public static void main(String[] args)
    {
        Child cobj = new Child("Parent","Child");
        cobj.details();
    }
}
```

Parent and Child

Note: When calling the parent class constructor from the child class using super keyword, super keyword should always be the first line in the method/constructor of the child class.

Super class reference pointing to Sub class object.

In context to above example where Class B extends class A.

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

is legal syntax because of IS-A relationship is there between class A and Class B.

Q. Can you use both this() and super() in a Constructor?

NO, because both super() and this() must be first statement inside a constructor. Hence we cannot use them together.