Constructors in Java

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

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

Every time an object is created using the new() keyword, at least one constructor is called.

It calls a default constructor if there is no constructor available in the class. In such case, Java compiler provides a default constructor by default.

There are two types of constructors in Java: no-arg constructor, and parameterized 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.

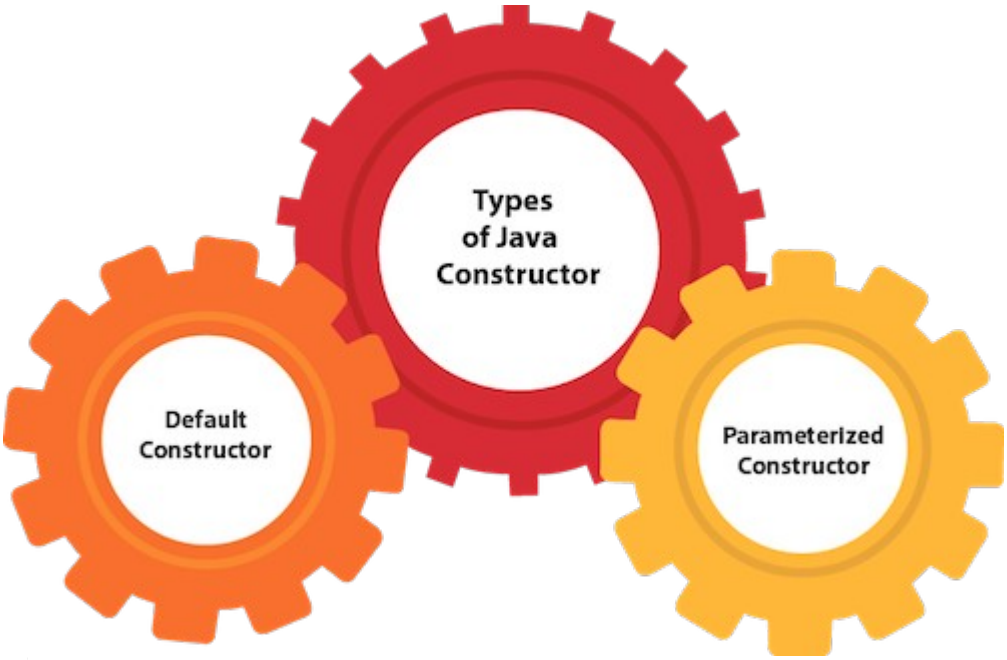
1. Constructor name must be the same as its class name
2. A Constructor must have no explicit return type
3. A Java constructor cannot be abstract, static, final, and synchronized

> 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:

```
<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.

```
//Java Program to create and call a default constructor
class Bike1{
//creating a default constructor
Bike1(){System.out.println("Bike is created");}
//main method
public static void main(String args[]){
//calling a default constructor
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.

Q) 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.         }
```

Test it Now

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 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 the parameterized constructor.
2.          class Student4{
3.              int id;
4.              String name;
5.              //creating a parameterized constructor
6.              Student4(int i,String n){
7.              id = i;
8.              name = n;
9.              }
10.             //method to display the values
11.             void display(){System.out.println(id+" "+name);}
```

```
12.
13.         public static void main(String args[]){
14.         //creating objects and passing values
15.         Student4 s1 = new Student4(111,"Karan");
16.         Student4 s2 = new Student4(222,"Aryan");
17.         //calling method to display the values of object
18.         s1.display();
19.         s2.display();
20.        }
21.       }
```

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
2.      class Student5{
3.         int id;
4.         String name;
5.         int age;
6.         //creating two arg constructor
7.         Student5(int i,String n){
8.         id = i;
9.         name = n;
10.        }
11.        //creating three arg constructor
12.        Student5(int i,String n,int a){
13.        id = i;
```

```
14.          name = n;
15.          age=a;
16.          }
17.          void display(){System.out.println(id+" "+name+" "+age);}
18.
19.          public static void main(String args[]){
20.          Student5 s1 = new Student5(111,"Karan");
21.          Student5 s2 = new Student5(222,"Aryan",25);
22.          s1.display();
23.          s2.display();
24.         }
25.       }
```

Test it Now

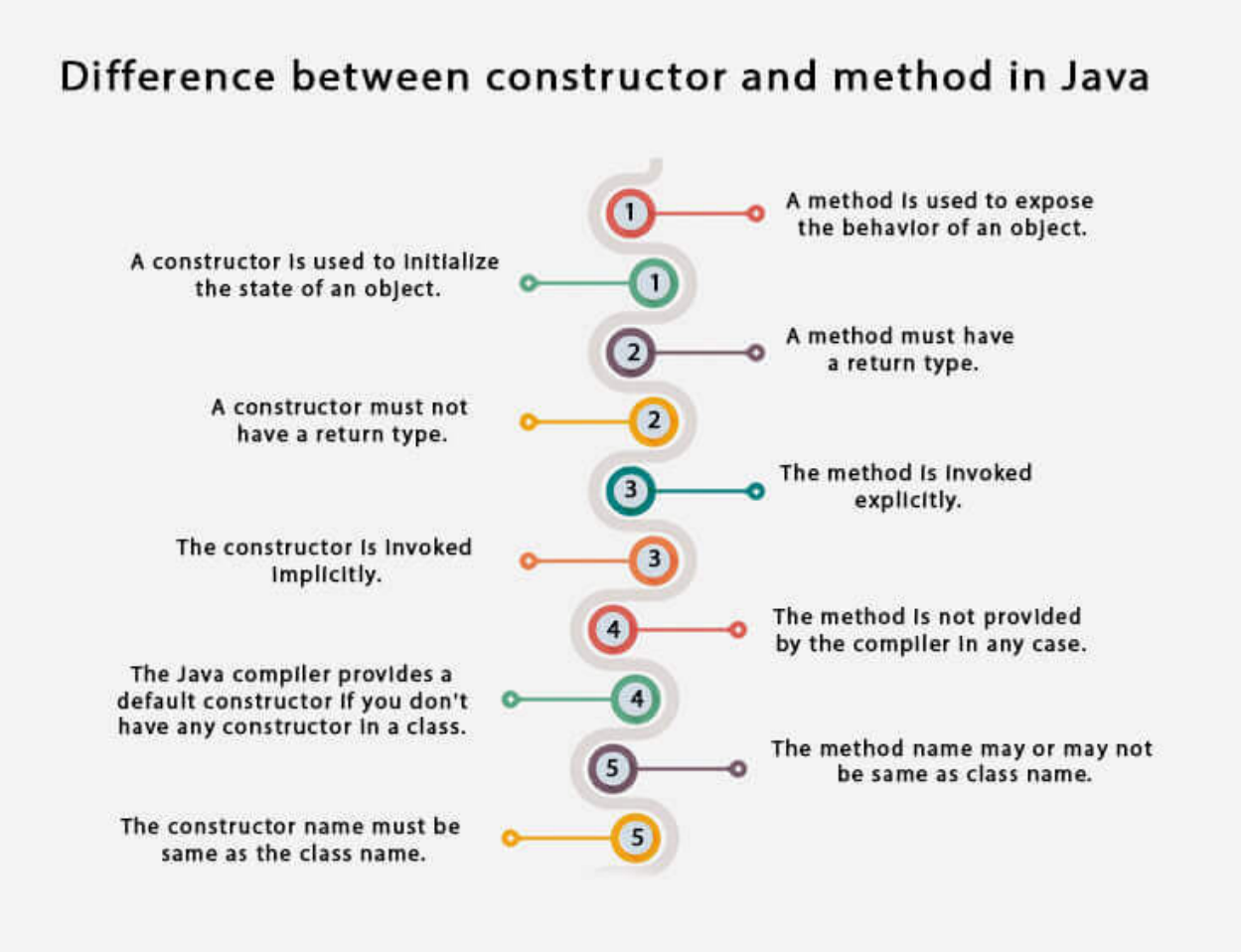
Output:

```
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	Java Method
A constructor is used to initialize the state of an object.	A method is used to expose the beha of an object.
A constructor must not have a return type.	A method must have a return type.
The constructor is invoked implicitly.	The method is invoked explicitly.
The Java compiler provides a default constructor if you don't have any constructor in a class.	The method is not provided by the compiler in any case.
The constructor name must be same as the class name.	The method name may or may not b same as the 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 object.
2.  class Student6{
3.     int id;
```

```
4.          String name;
5.          //constructor to initialize integer and string
6.          Student6(int i,String n){
7.          id = i;
8.          name = n;
9.          }
10.         //constructor to initialize another object
11.         Student6(Student6 s){
12.         id = s.id;
13.         name =s.name;
14.         }
15.         void display(){System.out.println(id+" "+name);}
16.
17.         public static void main(String args[]){
18.         Student6 s1 = new Student6(111,"Karan");
19.         Student6 s2 = new Student6(s1);
20.         s1.display();
21.         s2.display();
22.        }
23.     }
```

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.

```
1.      class Student7{
2.          int id;
3.          String name;
4.          Student7(int i,String n){
5.          id = i;
6.          name = n;
7.          }
8.          Student7(){}
9.          void display(){System.out.println(id+" "+name);}
```

```
10.
11.         public static void main(String args[]){
12.         Student7 s1 = new Student7(111,"Karan");
13.         Student7 s2 = new Student7();
14.         s2.id=s1.id;
15.         s2.name=s1.name;
16.         s1.display();
17.         s2.display();
18.        }
19.       }
```

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.

Is there Constructor class in Java?

Yes.

What is the purpose of Constructor class?

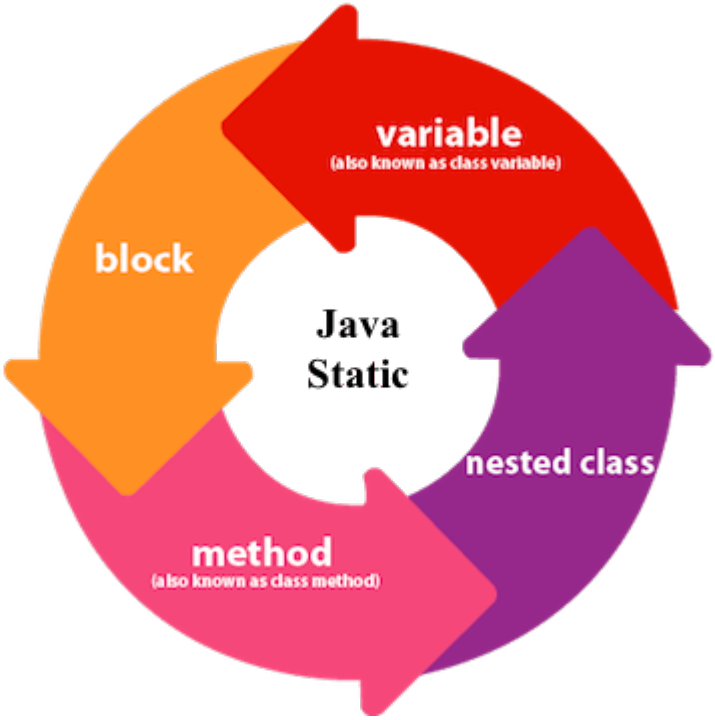
Java provides a Constructor class which can be used to get the internal information of a constructor in the class. It is found in the java.lang.reflect package.

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.      int rollno;
3.      String name;
4.      String college="Education World";
5.  }
```

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

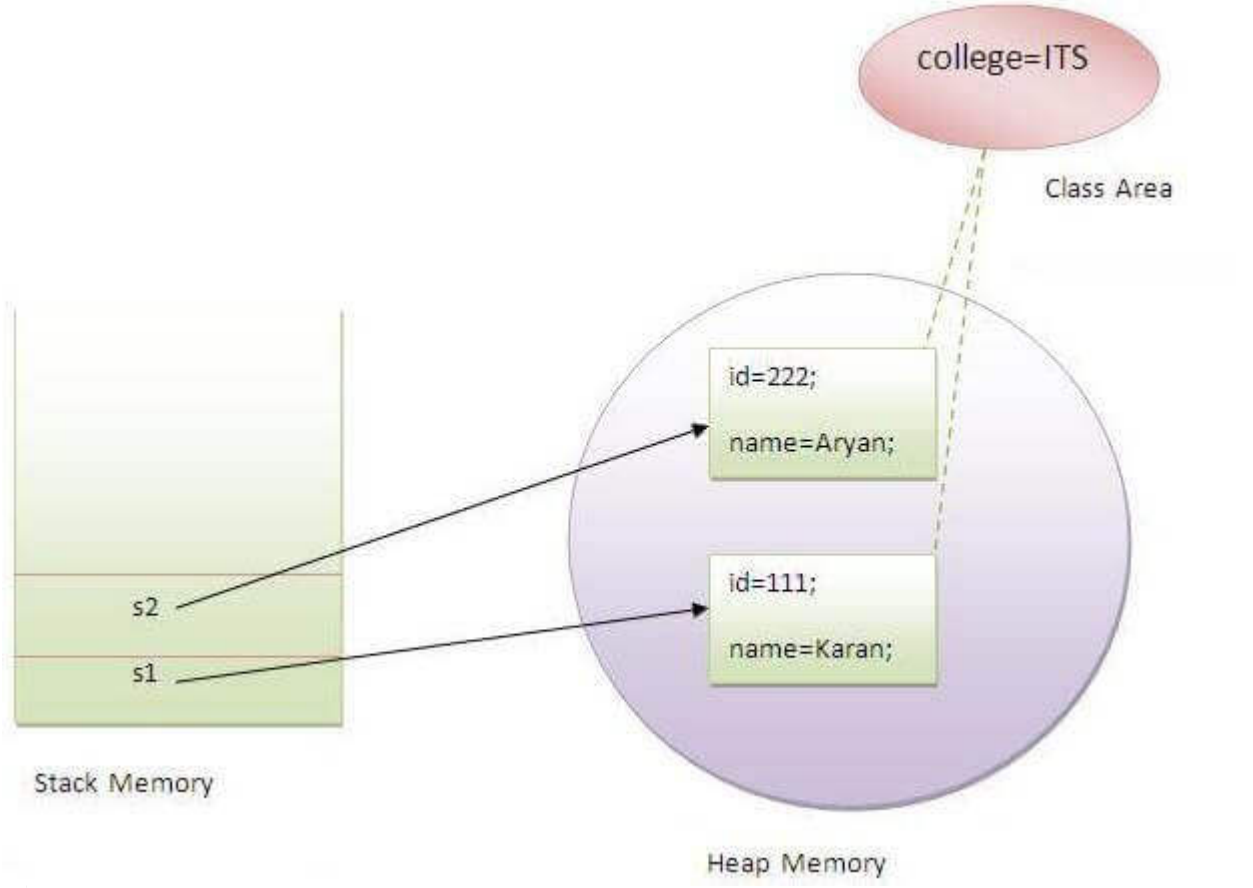
```
1.  //Java Program to demonstrate the use of static variable
2.  class Student{
3.    int rollno;//instance variable
4.    String name;
5.    static String college ="ITS";//static variable
6.    //constructor
7.    Student(int r, String n){
8.    rollno = r;
9.    name = n;
10.   }
```

```
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.
3.	class Counter{
4.	int count=0;//will get memory each time when the instance is created
5.
6.	Counter(){
7.	count++;//incrementing value
8.	System.out.println(count);
9.	}
10.
11.	public static void main(String args[]){
12.	//Creating objects
13.	Counter c1=new Counter();
14.	Counter c2=new Counter();
15.	Counter c3=new Counter();
16.	}
17.	}
```

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.
3.	class Counter2{
4.	static int count=0;//will get memory only once and retain its value
5.
6.	Counter2(){
```

```
7.          count++;//incrementing the value of static variable
8.          System.out.println(count);
9.          }
10.
11.         public static void main(String args[]){
12.         //creating objects
13.         Counter2 c1=new Counter2();
14.         Counter2 c2=new Counter2();
15.         Counter2 c3=new Counter2();
16.         }
17.         }
```

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

```
1.          //Java Program to demonstrate the use of a static method.
2.          class Student{
3.              int rollno;
4.              String name;
5.              static String college = "ITS";
6.              //static method to change the value of static variable
7.              static void change(){
8.              college = "BBDIT";
9.              }
10.             //constructor to initialize the variable
11.             Student(int r, String n){
12.             rollno = r;
```

```
13.             name = n;
14.             }
15.             //method to display values
16.             void display(){System.out.println(rollno+" "+name+" "+college);}
17.         }
18.         //Test class to create and display the values of object
19.         public class TestStaticMethod{
20.            public static void main(String args[]){
21.            Student.change();//calling change method
22.            //creating objects
23.            Student s1 = new Student(111,"Karan");
24.            Student s2 = new Student(222,"Aryan");
25.            Student s3 = new Student(333,"Sonoo");
26.            //calling display method
27.            s1.display();
28.            s2.display();
29.            s3.display();
30.            }
31.         }
```

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

```
1.  class A2{
2.    static
3.  {
4.  System.out.println("static block is invoked");
5.  }
6.    public static void main(String args[]){
7.     System.out.println("Hello main");
8.    }
9.  }
```

```
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.  }
```

Output:

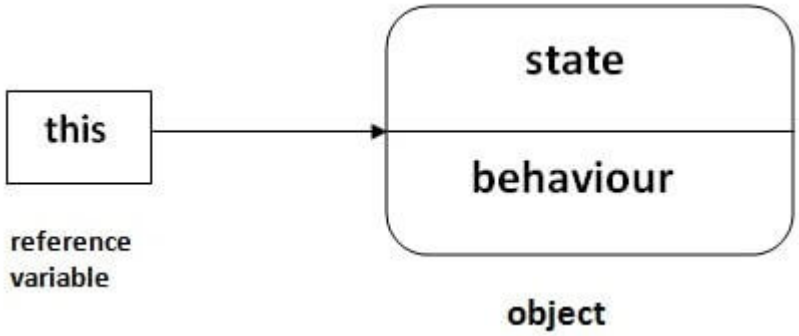
```
static block is invoked
```

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
```

this keyword in java

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

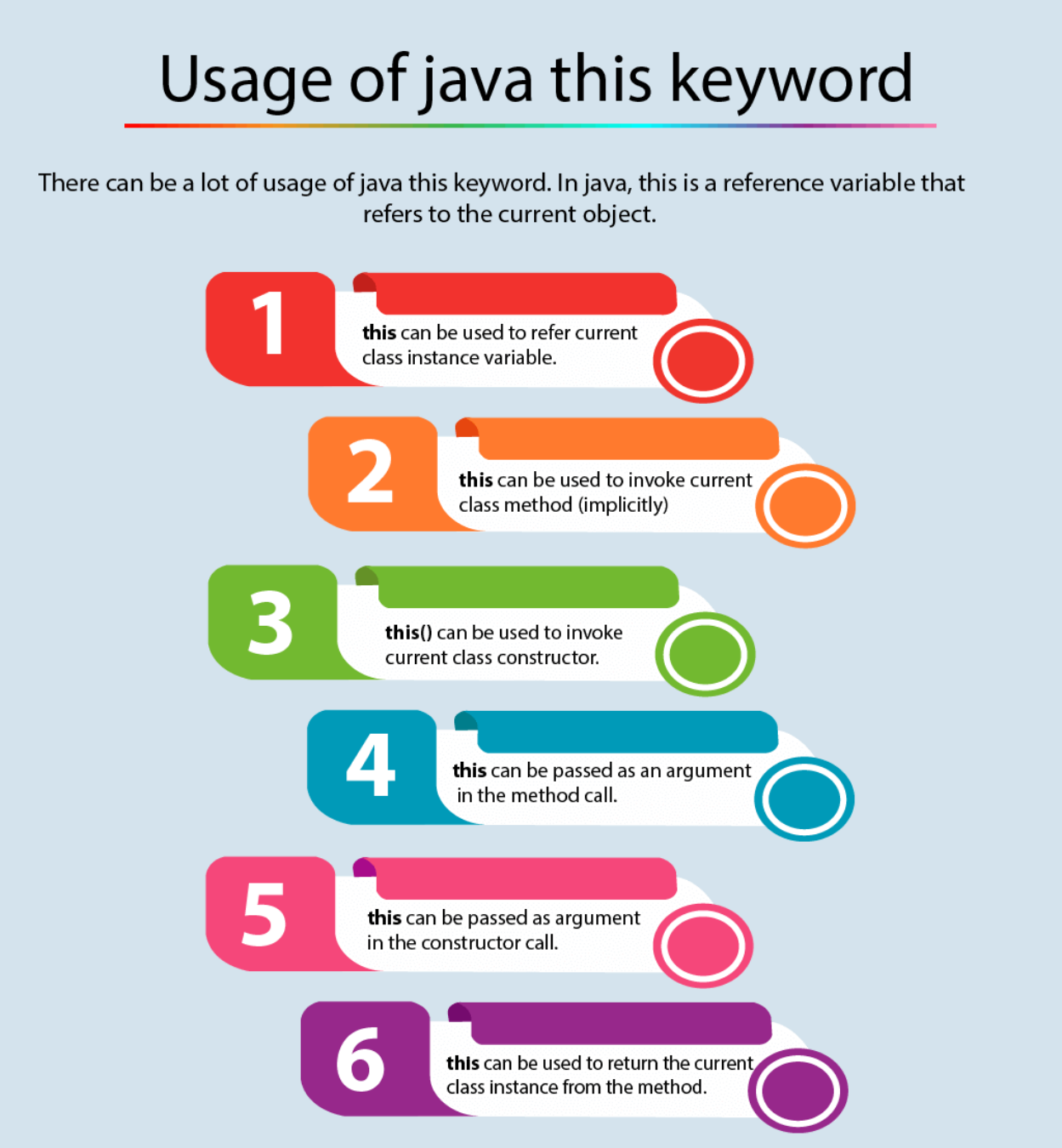


Usage of java this keyword

Here is given the 6 usage of java this keyword.

1. this 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 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

```
1.  class Student{
2.  int rollno;
3.  String name;
4.  float fee;
5.  Student(int rollno,String name,float fee){
6.  this.rollno=rollno;
7.  this.name=name;
8.  this.fee=fee;
9.  }
10. void display(){System.out.println(rollno+" "+name+" "+fee);}
11. }
```

```
12.
13.         class TestThis2{
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
```

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:

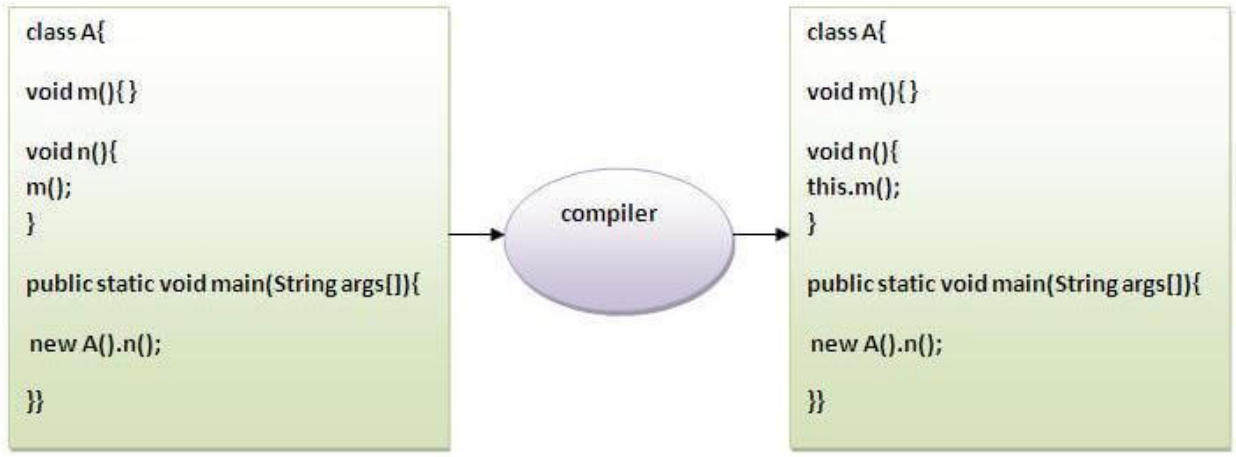
```
111 ankit 5000
```

```
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




```
1.  class A{
2.  void m(){System.out.println("hello m");}
3.  void n(){
4.  System.out.println("hello n");
5.  //m();//same as this.m()
6.  this.m();
7.  }
8.  }
9.  class TestThis4{
10. public static void main(String args[]){
11. A a=new A();
12. a.n();
13. }}
```

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:

```
1.  class A{
2.  A(){System.out.println("hello a");}
3.  A(int x){
4.  this();
5.  System.out.println(x);
6.  }
7.  }
8.  class TestThis5{
9.  public static void main(String args[]){
10. A a=new A(10);
11. }}
```

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. public static void main(String args[]){
12. A a=new A();
13. }}
```

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.

```
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(rollno,name,course);//reusing constructor
12. this.fee=fee;
13. }
14. void display(){System.out.println(rollno+" "+name+" "+course+" "+fee);}
15. }
16. class TestThis7{
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

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:

```
1.      class S2{
2.        void m(S2 obj){
3.        System.out.println("method is invoked");
4.        }
5.        void p(){
6.        m(this);
7.        }
8.        public static void main(String args[]){
9.        S2 s1 = new S2();
10.       s1.p();
11.       }
12.     }
```

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:

```
1.  class B{
2.    A4 obj;
3.    B(A4 obj){
4.      this.obj=obj;
5.    }
6.    void display(){
7.      System.out.println(obj.data);//using data member of A4 class
8.    }
9.  }
10.
11.  class A4{
12.    int data=10;
13.    A4(){
14.     B b=new B(this);
15.     b.display();
16.    }
17.    public static void main(String args[]){
18.     A4 a=new A4();
19.    }
20.  }
```

Test it Now

```
Output:10
```

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

We can return this keyword as an 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

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

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 printir reference variable and this, output of both variables are same.

```
class A5{
void m(){
System.out.println(this);//prints same reference ID
}
public static void main(String args[]){
A5 obj=new A5();
System.out.println(obj);//prints the reference ID
obj.m();
}
}
```

Test it Now

Output:

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

Inheritance in Java

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs (Object Oriented programming system).

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 the parent class. Moreover, you can add new methods and fields in your current class also.

Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.

Why use inheritance in java

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

Terms used in Inheritance

- **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
- **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

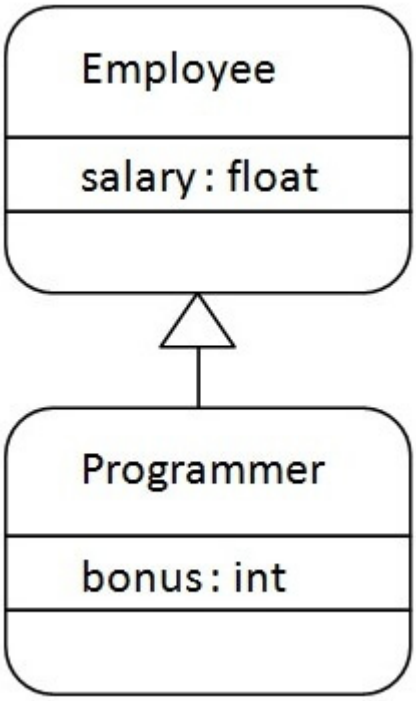
The syntax of Java Inheritance

```
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. The meaning of "extends" is to increase the functionality.

In the terminology of Java, a class which is inherited is called a parent or superclass, and the new class is called child or subclass.

Java Inheritance Example



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

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

Test it Now

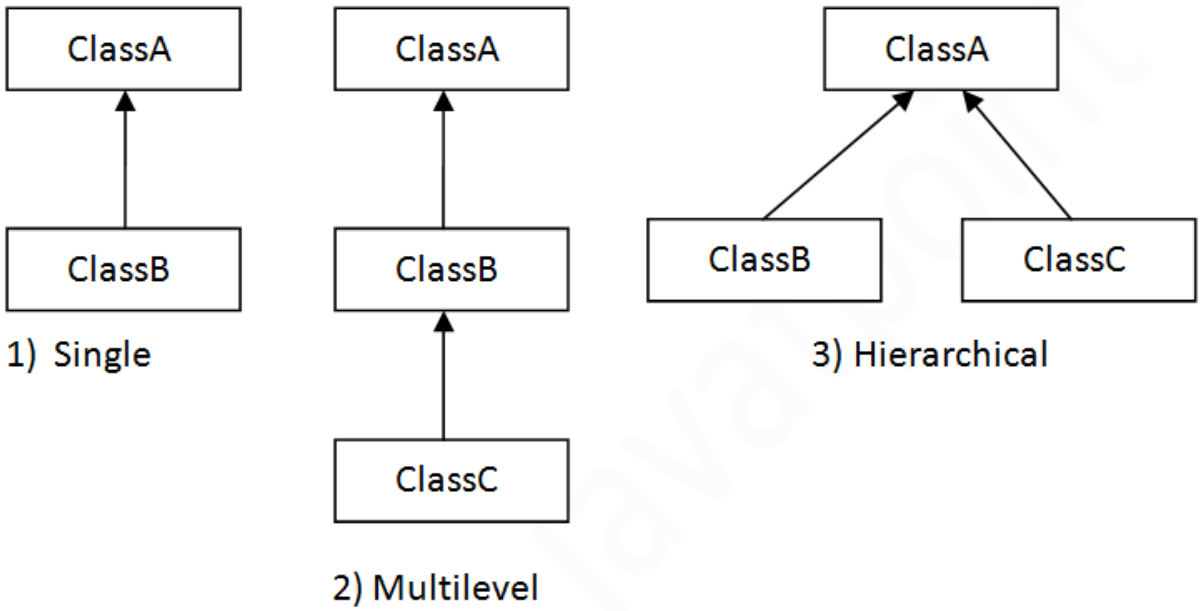
```
Programmer salary is:40000.0
Bonus of programmer is:10000
```

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

Types of inheritance in java

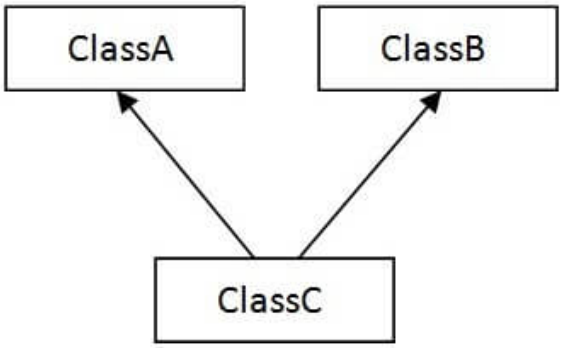
On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.

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

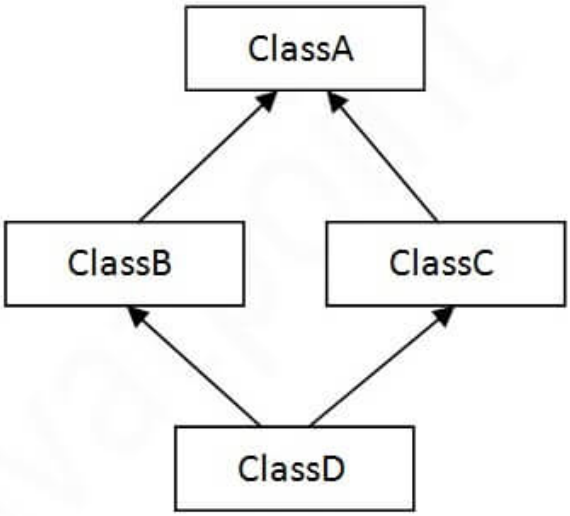


Note: Multiple inheritance is not supported in Java through class.

When one class inherits multiple classes, it is known as multiple inheritance. For Example:



4) Multiple



5) Hybrid

.

Single Inheritance Example

File: TestInheritance.java

```
1.  class Animal{
2.  void eat(){System.out.println("eating...");}
3.  }
4.  class Dog extends Animal{
5.  void bark(){System.out.println("barking...");}
6.  }
7.  class TestInheritance{
8.  public static void main(String args[]){
9.  Dog d=new Dog();
10. d.bark();
11. d.eat();
12. }}
```

Output:

```
barking...
eating...
```

Multilevel Inheritance Example

File: TestInheritance2.java

```
1.  class Animal{
2.  void eat(){System.out.println("eating...");}
3.  }
4.  class Dog extends Animal{
5.  void bark(){System.out.println("barking...");}
6.  }
7.  class BabyDog extends Dog{
8.  void weep(){System.out.println("weeping...");}
9.  }
10. class TestInheritance2{
11. public static void main(String args[]){
12. BabyDog d=new BabyDog();
13. d.weep();
14. d.bark();
15. d.eat();
16. }}
```

Output:

```
weeping...
barking...
eating...
```

Hierarchical Inheritance Example

File: TestInheritance3.java

```
1.  class Animal{
2.  void eat(){System.out.println("eating...");}
3.  }
4.  class Dog extends Animal{
5.  void bark(){System.out.println("barking...");}
6.  }
7.  class Cat extends Animal{
8.  void meow(){System.out.println("meowing...");}
9.  }
10. class TestInheritance3{
11. public static void main(String args[]){
```

```
12.         Cat c=new Cat();
13.         c.meow();
14.         c.eat();
15.         //c.bark();//C.T.Error
16.         }}
```

Output:

```
meowing...
eating...
```

Q) Why multiple inheritance is not supported in java?

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

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

Since compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error.

```
1.          class A{
2.          void msg(){System.out.println("Hello");}
3.          }
4.          class B{
5.          void msg(){System.out.println("Welcome");}
6.          }
7.          class C extends A,B{//suppose if it were
8.
9.           public static void main(String args[]){
10.            C obj=new C();
11.            obj.msg();//Now which msg() method would be invoked?
12.         }
13.         }
```

Test it Now

```
Compile Time Error
```

Aggregation in Java

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

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

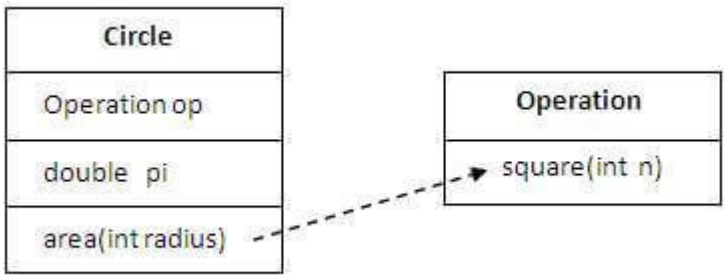
```
1.  class Employee{
2.  int id;
3.  String name;
4.  Address address;//Address is a class
5.  ...
6.  }
```

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

Why use Aggregation?

- For Code Reusability.

Simple Example of Aggregation



In this example, we have created the reference of Operation class in the Circle class.

```
1.  class Operation{
2.   int square(int n){
3.    return n*n;
4.   }
5.  }
```

```
6.
7.          class Circle{
8.           Operation op;//aggregation
9.           double pi=3.14;
10.
11.          double area(int radius){
12.            op=new Operation();
13.            int rsquare=op.square(radius);//code reusability (i.e. delegates the method call).
14.            return pi*rsquare;
15.          }
16.
17.
18.
19.          public static void main(String args[]){
20.            Circle c=new Circle();
21.            double result=c.area(5);
22.            System.out.println(result);
23.          }
24.         }
```

Test it Now

```
Output:78.5
```

When use Aggregation?

- Code reuse is also best achieved by aggregation when there is no is-a relationship.
- Inheritance should be used only if the relationship is-a is maintained throughout the lifetime of the objects involved; otherwise, aggregation is the best choice.

Understanding meaningful example of Aggregation

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

Address.java

```
1.          public class Address {
2.          String city,state,country;
3.
```

```
4.          public Address(String city, String state, String country) {
5.             this.city = city;
6.             this.state = state;
7.             this.country = country;
8.          }
9.
10.         }
```

Emp.java

```
1.          public class Emp {
2.          int id;
3.          String name;
4.          Address address;
5.
6.          public Emp(int id, String name,Address address) {
7.             this.id = id;
8.             this.name = name;
9.             this.address=address;
10.         }
11.
12.         void display(){
13.         System.out.println(id+" "+name);
14.         System.out.println(address.city+" "+address.state+" "+address.country);
15.         }
16.
17.         public static void main(String[] args) {
18.         Address address1=new Address("gzb","UP","india");
19.         Address address2=new Address("gno","UP","india");
20.
21.         Emp e=new Emp(111,"varun",address1);
22.         Emp e2=new Emp(112,"arun",address2);
23.
24.         e.display();
25.         e2.display();
26.
27.         }
28.         }
```

Test it Now

```
Output:111 varun
       gzb UP india
       112 arun
```

```
gno UP india
```

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.

```
1.  class Adder{
2.  static int add(int a,int b){return a+b;}
3.  static int add(int a,int b,int c){return a+b+c;}
4.  }
5.  class TestOverloading1{
6.  public static void main(String[] args){
7.  System.out.println(Adder.add(11,11));
8.  System.out.println(Adder.add(11,11,11));
9.  }}
```

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.

```
1.  class Adder{
2.  static int add(int a, int b){return a+b;}
3.  static double add(double a, double b){return a+b;}
4.  }
5.  class TestOverloading2{
```

```
6.      public static void main(String[] args){
7.      System.out.println(Adder.add(11,11));
8.      System.out.println(Adder.add(12.3,12.6));
9.      }}
```

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:

```
1.      class Adder{
2.      static int add(int a,int b){return a+b;}
3.      static double add(int a,int b){return a+b;}
4.      }
5.      class TestOverloading3{
6.      public static void main(String[] args){
7.      System.out.println(Adder.add(11,11));//ambiguity
8.      }}
```

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.  }
```

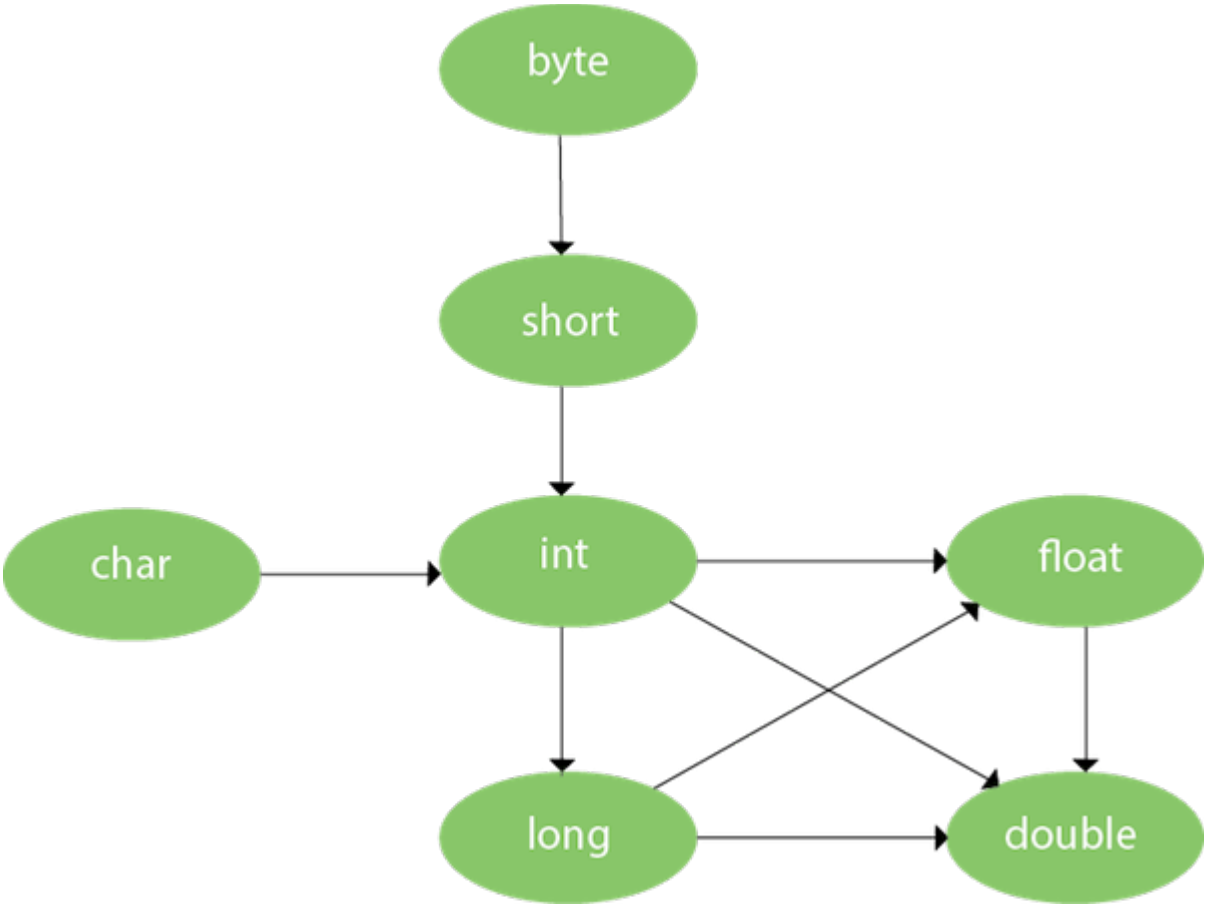
Test it Now

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

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

Test it Now

```
Output:40
       60
```

Example of Method Overloading with Type Promotion if matching found

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

```
1.          class OverloadingCalculation2{
2.            void sum(int a,int b){System.out.println("int arg method invoked");}
3.            void sum(long a,long b){System.out.println("long arg method invoked");}
4.
5.            public static void main(String args[]){
6.            OverloadingCalculation2 obj=new OverloadingCalculation2();
7.            obj.sum(20,20);//now int arg sum() method gets invoked
8.            }
9.          }
```

Test it Now

```
Output:int arg 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.

```
1.          class OverloadingCalculation3{
2.            void sum(int a,long b){System.out.println("a method invoked");}
3.            void sum(long a,int b){System.out.println("b method invoked");}
4.
5.            public static void main(String args[]){
6.            OverloadingCalculation3 obj=new OverloadingCalculation3();
7.            obj.sum(20,20);//now ambiguity
8.            }
9.          }
```

Test it Now

```
Output:Compile Time Error
```

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