Java OOPs Concepts

Simula is considered as the first object-oriented programming language. The programming paradigm where everything is represented as an object, is known as truly object-oriented programming language.

Smalltalk is considered as the first truly object-oriented programming language.

OOPs (Object Oriented Programming System)

Object means a real word entity such as pen, chair, table etc. **Object-Oriented Programming** is a methodology or paradigm to design a program using classes and objects. It simplifies the software development and maintenance by providing some concepts:

- Object
- Class
- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

Object

Any entity that has state and behavior is known as an object. For example: chair, pen, table, keyboard, bike etc. It can be physical and logical.

Class

Collection of objects is called class. It is a logical entity.

Inheritance

When one object acquires all the properties and behaviours of parent object i.e. known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.

Polymorphism

When **one task is performed by different ways** i.e. known as polymorphism. For example: to draw something e.g. shape or rectangle etc.

In java, we use method overloading and method overriding to achieve polymorphism.

Another example can be to speak something e.g. cat speaks meaw, dog barks woof etc.

Java OOPs Concepts

Abstraction

Hiding internal details and showing functionality is known as abstraction. For example: phone call, we don't know the internal processing.

In java, we use abstract class and interface to achieve abstraction.

Encapsulation

Binding (or wrapping) code and data together into a single unit is known as encapsulation. For example: capsule, it is wrapped with different medicines.

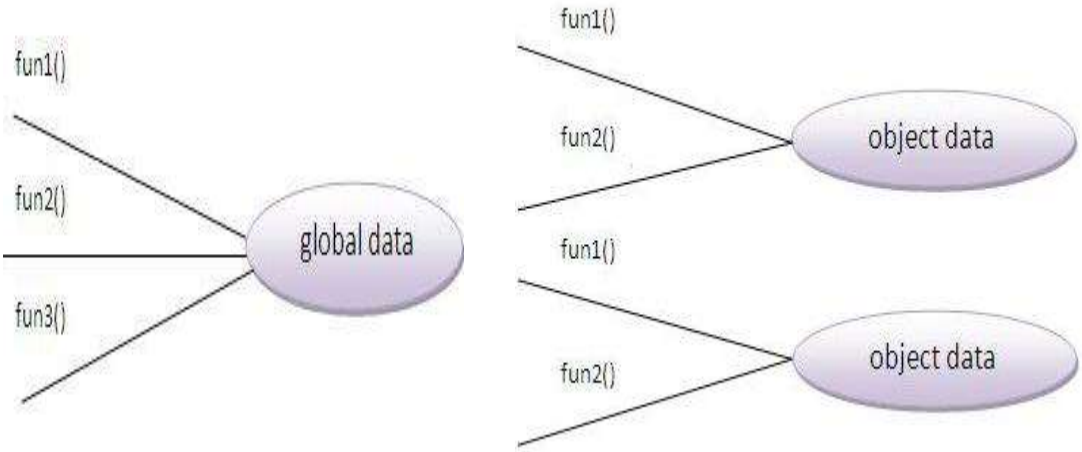
A java class is the example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here.

Advantage of OOPs over Procedure-oriented programming language

1)OOPs makes development and maintenance easier where as in Procedure-oriented programming language it is not easy to manage if code grows as project size grows.

2)OOPs provides data hiding whereas in Procedure-oriented prgramming language a global data can be accessed from anywhere.

3)OOPs provides ability to simulate real-world event much more effectively. We can provide the solution of real word problem if we are using the Object-Oriented Programming language.



Java OOPs Concepts

What is difference between object-oriented programming language and object-based programming language?

Object based programming language follows all the features of OOPs except Inheritance. JavaScript and VBScript are examples of object based programming languages.

Questions:

- Can we overload main method ?
- Constructor returns a value but, what ?
- Can we create a program without main method ?
- What are the 6 ways to use this keyword ?
- Why multiple inheritance is not supported in java ?
- Why use aggregation ?
- Can we override the static method ?
- What is covariant return type ?
- What are the three usage of super keyword?
- Why use instance initializer block?
- What is the usage of blank final variable ?
- What is marker or tagged interface ?
- What is runtime polymorphism or dynamic method dispatch ?
- What is the difference between static and dynamic binding ?
- How downcasting is possible in java ?
- What is the purpose of private constructor?
- What is object cloning ?

Java Naming conventions

Java **naming convention** is a rule to follow as you decide what to name your identifiers such as class, package, variable, constant, method etc.

But, it is not forced to follow. So, it is known as convention not rule.

All the classes, interfaces, packages, methods and fields of java programming language are given according to java naming convention.

Advantage of naming conventions in java

By using standard Java naming conventions, you make your code easier to read for yourself and for other programmers. Readability of Java program is very important. It indicates that **less time** is spent to figure out what the code does.

Name	Convention
class name	should start with uppercase letter and be a noun e.g. String, Color, Button, System, Thread etc.
interface name	should start with uppercase letter and be an adjective e.g. Runnable, Remote, ActionListener etc.
method name	should start with lowercase letter and be a verb e.g. actionPerformed(), main(), print(), println() etc.
variable name	should start with lowercase letter e.g. firstName, orderNumber etc.
package name	should be in lowercase letter e.g. java, lang, sql, util etc.
constants name	should be in uppercase letter. e.g. RED, YELLOW, MAX_PRIORITY etc.

CamelCase in java naming conventions

Java follows camelcase syntax for naming the class, interface, method and variable.

If name is combined with two words, second word will start with uppercase letter always e.g. actionPerformed(), firstName, ActionEvent, ActionListener etc.

Java OOPs Concepts

Object and Class in Java

In object-oriented programming technique, we design a program using objects and classes.

Object is the physical as well as logical entity whereas class is the logical entity only.

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 (tengible and intengible). The example of intangible object is banking system.

An object has three characteristics:

- **state:** represents data (value) of an object.
- **behavior:** represents the behavior (functionality) of an object such as deposit, withdraw etc.
- **identity:** Object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. But, 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 etc. known as its state. It is used to write, so writing is its behavior.

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

Class in Java

A class is a group of objects that has common properties. It is a template or blueprint from which objects are created.

A class in java can contain:

- **data member**
- **method**
- **constructor**
- **block**
- **class and interface**

Java OOPs Concepts

Syntax to declare a class:

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

Simple Example of Object and Class

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

```
1. class Student1{
2.  int id;//data member (also instance variable)
3.  String name;//data member(also instance variable)
4.
5.  public static void main(String args[]){
6.   Student1 s1=new Student1();//creating an object of Student
7.   System.out.println(s1.id);
8.   System.out.println(s1.name);
9.  }
10. }
```

```
Output:0 null
```

Instance variable in Java

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

Method in Java

In java, a method is like function i.e. used to expose behaviour of an object.

Advantage of Method

- Code Reusability
- Code Optimization

new keyword

The new keyword is used to allocate memory at runtime.

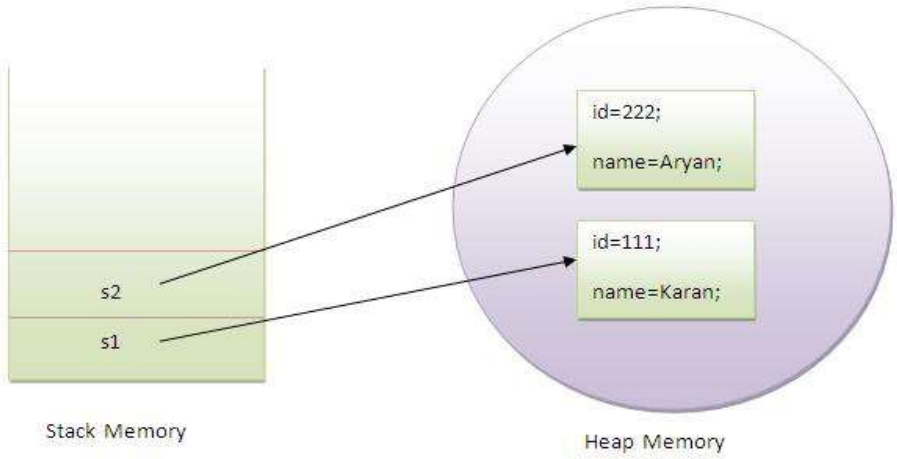
Java OOPs Concepts

Example of Object and class that maintains the records of students

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

```
1.  class Student2{
2.   int rollno;
3.   String name;
4.
5.   void insertRecord(int r, String n){  //method
6.    rollno=r;
7.    name=n;
8.   }
9.
10.  void displayInformation(){System.out.println(rollno+" "+name);}//method
11.
12.  public static void main(String args[]){
13.   Student2 s1=new Student2();
14.   Student2 s2=new Student2();
15.
16.   s1.insertRecord(111,"Karan");
17.   s2.insertRecord(222,"Aryan");
18.
19.   s1.displayInformation();
20.   s2.displayInformation();
21.
22. }
23.}
```

```
111 Karan
222 Aryan
```



As you see in the above figure, object gets the memory in Heap area and 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.

Another Example of Object and Class

There is given another example that maintains the records of Rectangle class. Its exaplanation is same as in the above Student class example.

```
1.  class Rectangle{
2.   int length;
3.   int width;
4.
5.   void insert(int l,int w){
6.    length=l;
7.    width=w;
8.   }
9.
10.  void calculateArea(){System.out.println(length*width);}
11.
12.  public static void main(String args[]){
13.   Rectangle r1=new Rectangle();
14.   Rectangle r2=new Rectangle();
15.
16.   r1.insert(11,5);
17.   r2.insert(3,15);
18.
19.   r1.calculateArea();
20.   r2.calculateArea();
21. }
22. }
```

```
Output:55
       45
```

Java OOPs Concepts

What are the different ways to create an object in Java?

There are many ways to create an object in java. They are:

- By new keyword
- By newInstance() method
- By clone() method
- By factory method etc.

Annonymous object

Annonymous simply means nameless. An object that have no reference is known as annonymous object.

If you have to use an object only once, annonymous object is a good approach.

```
1. class Calculation{
2.
3.  void fact(int  n){
4.   int fact=1;
5.   for(int i=1;i<=n;i++){
6.    fact=fact*i;
7.   }
8.  System.out.println("factorial is "+fact);
9. }
10.
11. public static void main(String args[]){
12.  new Calculation().fact(5);//calling method with annonymous object
13. }
14. }
```

```
Output:Factorial is 120
```

Java OOPs Concepts

Creating multiple objects by one type only

We can create multiple objects by one type only as we do in case of primitives.

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

Let's see the example:

```
1. class Rectangle{
2.  int length;
3.  int width;
4.
5.  void insert(int l,int w){
6.   length=l;
7.   width=w;
8.  }
9.
10. void calculateArea(){System.out.println(length*width);}
11.
12. public static void main(String args[]){
13.  Rectangle r1=new Rectangle(),r2=new Rectangle();//creating two objects
14.
15.  r1.insert(11,5);
16.  r2.insert(3,15);
17.
18.  r1.calculateArea();
19.  r2.calculateArea();
20. }
21. }
```

```
Output:55
        45
```

Method Overloading in Java

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

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

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

Advantage of method overloading?

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

Different ways to overload the method

There are two ways to overload the method in java

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

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

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

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

```
1. class Calculation{
2.   void sum(int a,int 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.   Calculation obj=new Calculation();
7.   obj.sum(10,10,10);
8.   obj.sum(20,20);
9.     }
10. }
```

```
Output:30
        40
```

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

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

```
1. class Calculation2{
2.   void sum(int a,int b){System.out.println(a+b);}
3.   void sum(double a,double b){System.out.println(a+b);}
4.
5.   public static void main(String args[]){
6.   Calculation2 obj=new Calculation2();
7.   obj.sum(10.5,10.5);
8.   obj.sum(20,20);
9.
10.  }
11.}
```

```
Output:21.0
        40
```

Que) Why Method Overloaing is not possible by changing the return type of method?

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

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

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

Can we overload main() method?

Yes, by method overloading. You can have any number of main methods in a class by method overloading. Let's see the simple example:

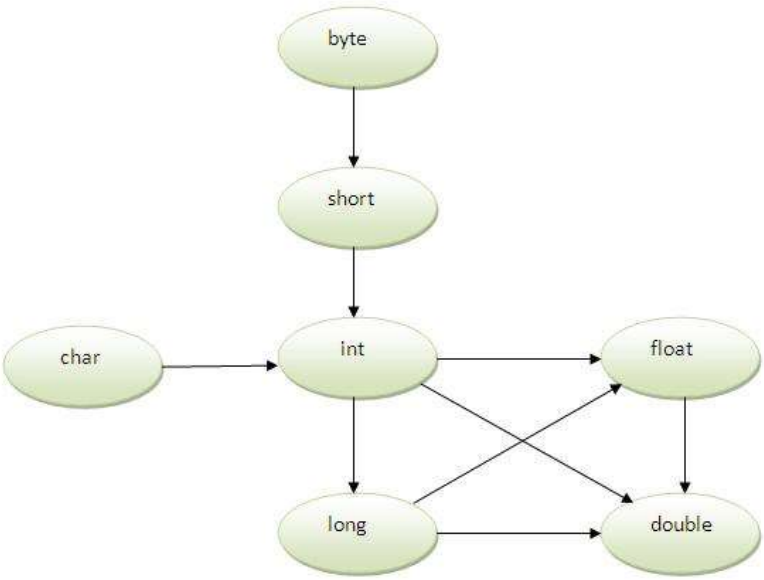
```
class Overloading1{
  public static void main(int a){
  System.out.println(a);
  }

  public static void main(String args[]){
  System.out.println("main() method invoked");
  main(10);
  }
}
```

```
Output:main() method invoked
       10
```

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. }
```

```
Output:40
       60
```

Example of Method Overloading with TypePromotion if matching found

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

```
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. }
```

```
Output:int arg method invoked
```

Example of Method Overloading with TypePromotion in case ambiguity

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

```
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. }
```

```
Output:Compile Time Error
```

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

Constructor in Java

Constructor in java is a *special type of method* that is used to initialize the object.

Java constructor is *invoked at the time of object creation*. It constructs the values i.e. provides data for the object that is why it is known as constructor.

Rules for creating java constructor

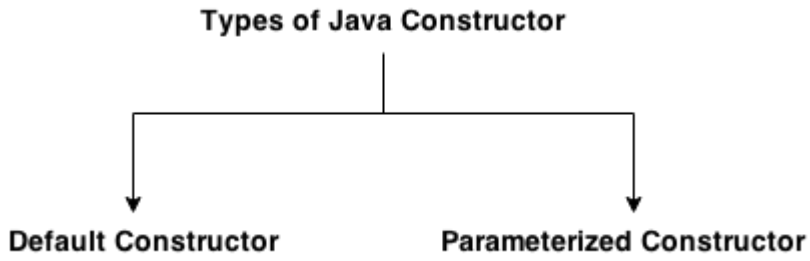
There are basically two rules defined for the constructor.

1. Constructor name must be same as its class name
2. Constructor must have no explicit return type

Types of java constructors

There are two types of constructors:

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



Java Default Constructor

A constructor that have no parameter is known as default constructor.

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. class Bike1{
2. Bike1(){System.out.println("Bike is created");}
3. public static void main(String args[]){
4. Bike1 b=new Bike1();
5. }
6. }
```

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 default constructor?

Default constructor provides the default values to the object like 0, null etc. depending on the type.

Example of default constructor that displays the default values

```
1. class Student3{
2. int id;
3. String name;
4. void display(){System.out.println(id+" "+name);}
5. public static void main(String args[]){
6. Student3 s1=new Student3();
7. Student3 s2=new Student3();
8. s1.display();
9. s2.display();
10. }
11. }
```

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 that have parameters is known as parameterized constructor.

Why use parameterized constructor?

Parameterized constructor is used to provide different values to the distinct objects.

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.  class Student4{
2.      int id;
3.      String name;
4.
5.      Student4(int i,String n){
6.      id = i;
7.      name = n;
8.      }
9.      void display(){System.out.println(id+" "+name);}
10.
11.     public static void main(String args[]){
12.     Student4 s1 = new Student4(111,"Karan");
13.     Student4 s2 = new Student4(222,"Aryan");
14.     s1.display();
15.     s2.display();
16.    }
17. }
```

Output:

```
111 Karan
222 Aryan
```

Java OOPs Concepts

Constructor Overloading in Java

Constructor overloading is a technique in Java in which a class can have any number of constructors that differ in parameter lists.The compiler differentiates these constructors by taking into account the number of parameters in the list and their type.

Example of Constructor Overloading

```
1.  class Student5{
2.      int id;
3.      String name;
4.      int age;
5.      Student5(int i,String n){
6.      id = i;
7.      name = n;
8.      }
9.      Student5(int i,String n,int a){
10.     id = i;
11.     name = n;
12.     age=a;
13.     }
14.     void display(){System.out.println(id+" "+name+" "+age);}
15.
16.     public static void main(String args[]){
17.     Student5 s1 = new Student5(111,"Karan");
18.     Student5 s2 = new Student5(222,"Aryan",25);
19.     s1.display();
20.     s2.display();
21.    }
22. }
```

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
Constructor is used to initialize the state of an object.	Method is used to expose behaviour of an object.
Constructor must not have return type.	Method must have return type.
Constructor is invoked implicitly.	Method is invoked explicitly.
The java compiler provides a default constructor if you don't have any constructor.	Method is not provided by compiler in any case.
Constructor name must be same as the class name.	Method name may or may not be same as class name.

Java Copy Constructor

There is no copy constructor in java. But, we can copy the values of 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. class Student6{
2.     int id;
3.     String name;
4.     Student6(int i,String n){
5.     id = i;
6.     name = n;
7.     }
8.
9.     Student6(Student6 s){
10.    id = s.id;
11.    name =s.name;
12.    }
13.    void display(){System.out.println(id+" "+name);}
14.
15.    public static void main(String args[]){
16.    Student6 s1 = new Student6(111,"Karan");
17.    Student6 s2 = new Student6(s1);
18.    s1.display();
19.    s2.display();
20.   }
21. }
```

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. }
```

Output:`111 Karan    111 Karan`

Java OOPs Concepts

Q) Does constructor return any value?

Ans: yes, that is 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 method etc. You can perform any operation in the constructor as you perform in the method.

Java static keyword

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

The static can be:

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

1) Java static variable

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

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

Advantage of static variable

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

Understanding problem without static variable

```
1. class Student{
2.     int rollno;
3.     String name;
4.     String college="ITS";
5. }
```

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

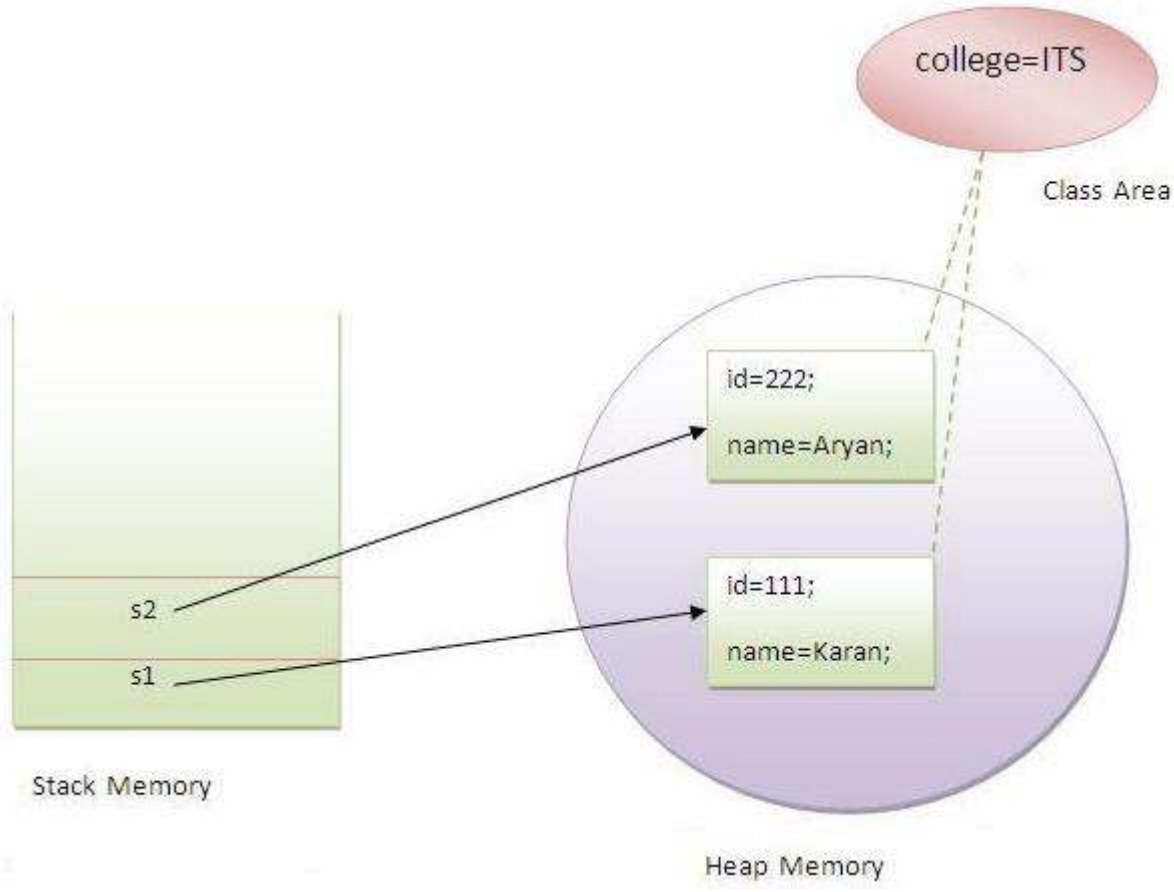
Java static property is shared to all objects.

Java OOPs Concepts

Example of static variable

```
1. //Program of static variable
2.
3. class Student8{
4.    int rollno;
5.    String name;
6.    static String college ="ITS";
7.
8.    Student8(int r,String n){
9.    rollno = r;
10.   name = n;
11.   }
12. void display (){System.out.println(rollno+" "+name+" "+college);}
13.
14. public static void main(String args[]){
15. Student8 s1 = new Student8(111,"Karan");
16. Student8 s2 = new Student8(222,"Aryan");
17.
18. s1.display();
19. s2.display();
20. }
21. }
```

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



Java OOPs Concepts

Program of 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 to other objects. So each objects will have the value 1 in the count variable.

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

```
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. class Counter2{
2. static int count=0;//will get memory only once and retain its value
3.
4. Counter2(){
5. count++;
6. System.out.println(count);
7. }
8.
9. public static void main(String args[]){
10.
11. Counter2 c1=new Counter2();
12. Counter2 c2=new Counter2();
13. Counter2 c3=new Counter2();
14.
15. }
16. }
```

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

Example of static method

```
1. //Program of changing the common property of all objects(static field).
2.
3. class Student9{
4.     int rollno;
5.     String name;
6.     static String college = "ITS";
7.
8.     static void change(){
9.     college = "BBDIT";
10.    }
11.
12.    Student9(int r, String n){
13.    rollno = r;
14.    name = n;
15.    }
16.
17.    void display (){System.out.println(rollno+" "+name+" "+college);}
18.
19.   public static void main(String args[]){
20.   Student9.change();
21.
22.   Student9 s1 = new Student9 (111,"Karan");
23.   Student9 s2 = new Student9 (222,"Aryan");
24.   Student9 s3 = new Student9 (333,"Sonoo");
25.
26.   s1.display();
27.   s2.display();
28.   s3.display();
29.   }
30. }
```

```
Output:111 Karan BBDIT
       222 Aryan BBDIT
       333 Sonoo BBDIT
```

Another example of static method that performs normal calculation

```
1. //Program to get cube of a given number by 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. }
```

```
Output:125
```

Restrictions for static method

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

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

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

```
Output:Compile Time Error
```

Q) why java main method is static?

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

3) Java static block

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

Example of static block

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

```
Output:static block is invoked
       Hello main
```

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

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

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

```
Output:static block is invoked (if not JDK7)
```

In JDK7 and above, output will be:

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

this keyword in java

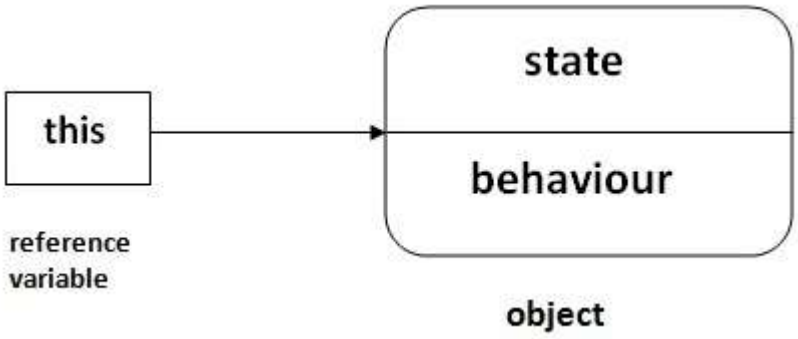
There can be a lot of usage of **java this keyword**. In java, this is a **reference variable** that refers to the current object.

Usage of java this keyword

Here is given the 6 usage of java this keyword.

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

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



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

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

Java OOPs Concepts

Understanding the problem without this keyword

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

```
1. class Student10{
2.     int id;
3.     String name;
4.
5.     Student10(int id,String name){
6.     id = id;
7.     name = name;
8.     }
9.     void display(){System.out.println(id+" "+name);}
10.
11.    public static void main(String args[]){
12.    Student10 s1 = new Student10(111,"Karan");
13.    Student10 s2 = new Student10(321,"Aryan");
14.    s1.display();
15.    s2.display();
16.    }
17. }
```

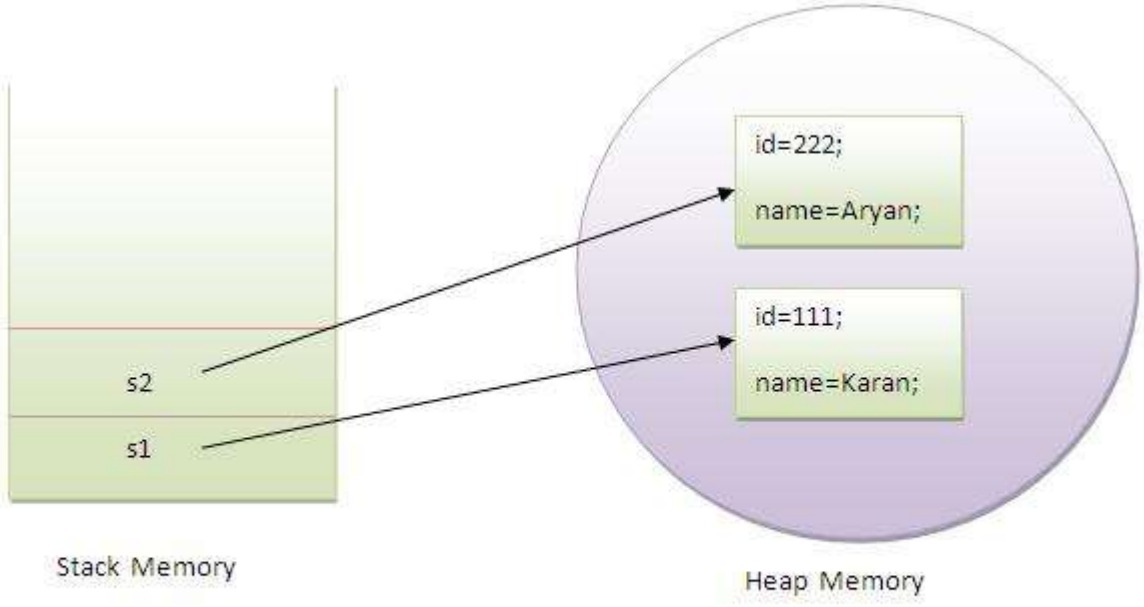
```
Output:0 null
       0 null
```

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

Solution of the above problem by this keyword

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

```
Output111 Karan
       222 Aryan
```



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

Program where this keyword is not required

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

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

Java OOPs Concepts

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

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

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

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

Java OOPs Concepts

Where to use this() constructor call?

The this() constructor call should be used to reuse the constructor in 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 Student14{
2.      int id;
3.      String name;
4.      String city;
5.
6.      Student14(int id,String name){
7.      this.id = id;
8.      this.name = name;
9.      }
10.     Student14(int id,String name,String city){
11.     this(id,name);//now no need to initialize id and name
12.     this.city=city;
13.     }
14.     void display(){System.out.println(id+" "+name+" "+city);}
15.
16.     public static void main(String args[]){
17.     Student14 e1 = new Student14(111,"karan");
18.     Student14 e2 = new Student14(222,"Aryan","delhi");
19.     e1.display();
20.     e2.display();
21.    }
22. }
```

```
Output:111 Karan null
       222 Aryan delhi
```

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

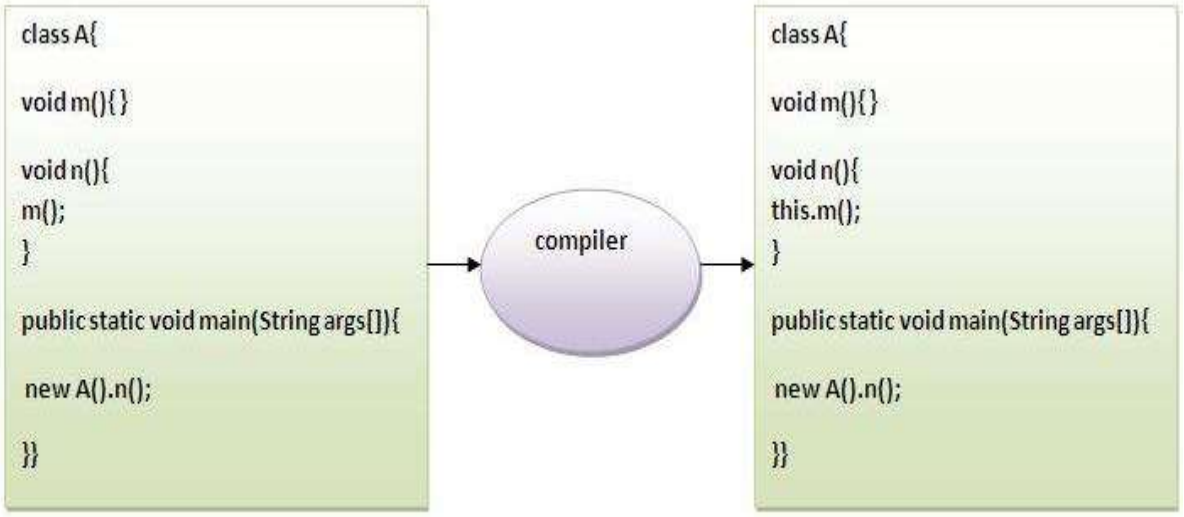
```
1.  class Student15{
2.      int id;
3.      String name;
4.      Student15(){System.out.println("default constructor is invoked");}
5.      Student15(int id,String name){
6.      id = id;
7.      name = name;
8.      this ();//must be the first statement
9.      }
10.     void display(){System.out.println(id+"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

```
Output:Compile Time Error
```

Java OOPs Concepts

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

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




```
1.  class S{
2.    void m(){
3.    System.out.println("method is invoked");
4.    }
5.    void n(){
6.    this.m();//no need because compiler does it for you.
7.    }
8.    void p(){
9.    n();//complier will add this to invoke n() method as this.n()
10.   }
11.   public static void main(String args[]){
12.   S s1 = new S();
13.   s1.p();
14.   }
15. }
```

```
Output:method is invoked
```

4) this keyword can be passed 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.}
```

```
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.

5) The this keyword can be passed 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.class A4{
11.  int data=10;
12.  A4(){
13.   B b=new B(this);
14.   b.display();
15.  }
16.  public static void main(String args[]){
17.   A4 a=new A4();
18.  }
```

```
}
```

```
Output:10
```

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

We can return the this keyword as an statement from the method. In such case, return type of the method must be the class type (non-primitive).

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();
}
}
```

```
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.

```
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();
}
}
```

```
Output:A5@22b3ea59
       A5@22b3ea59
```

Java OOPs Concepts

Inheritance in Java

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

The idea behind inheritance in java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of parent class, and you can add new methods and fields also.

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

Why use inheritance in java

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

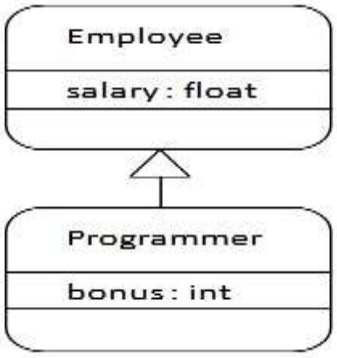
Syntax of Java Inheritance

```
1. class Subclass-name extends Superclass-name
2. {
3.   //methods and fields
4. }
```

The **extends keyword** indicates that you are making a new class that derives from an existing class.

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

Understanding the simple example of inheritance



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

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

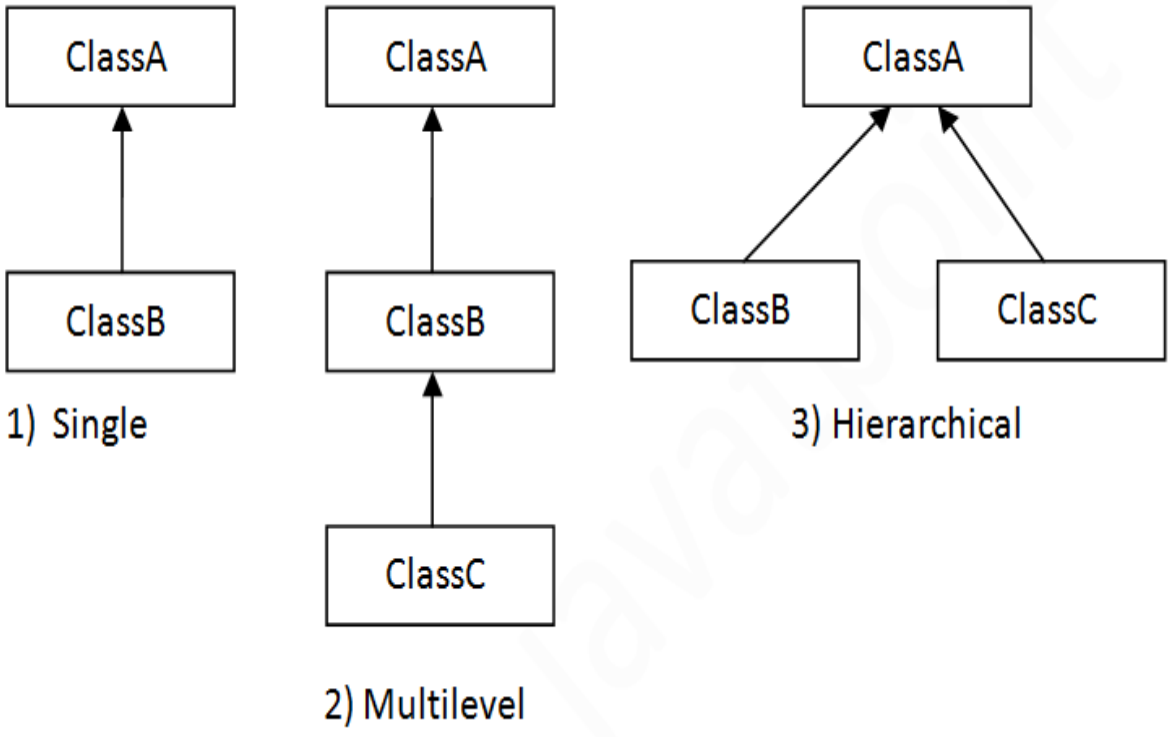
```
Programmer salary is:40000.0
Bonus of programmer is:10000
```

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

Types of inheritance in java

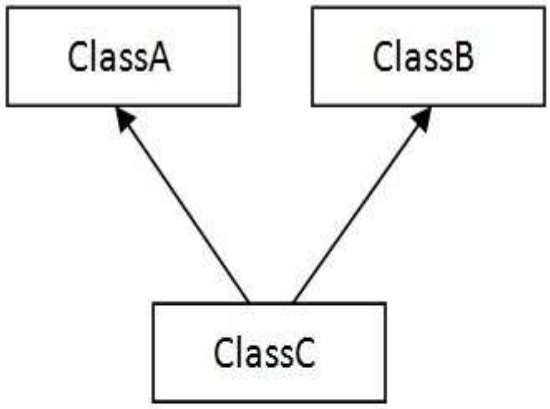
On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.

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

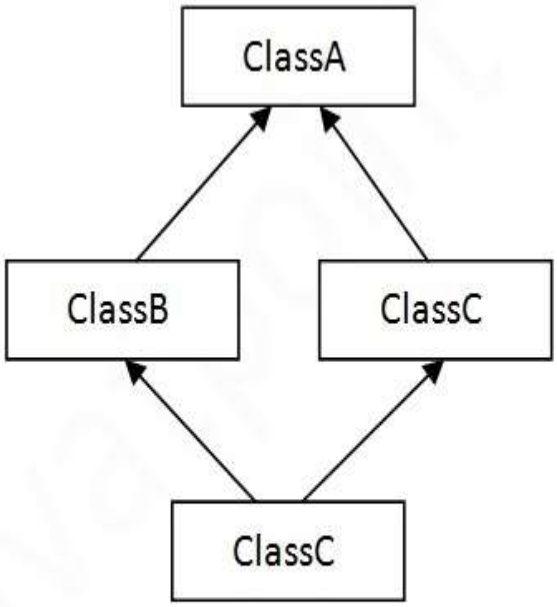


Note: Multiple inheritance is not supported in java through class.

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



4) Multiple



5) Hybrid

Q) Why multiple inheritance is not supported in java?

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

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

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

Java OOPs Concepts

```
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. }
```

```
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.

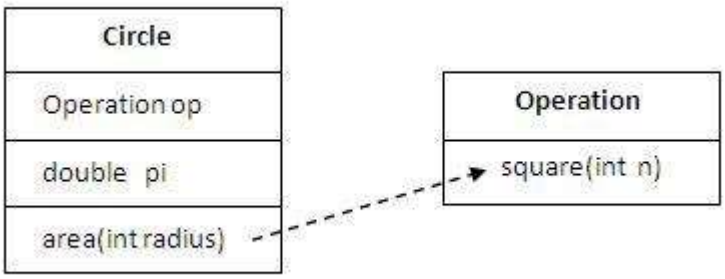
```
1. class Employee{
2. int id;
3. String name;
4. Address address;//Address is a class
5. ...
6. }
```

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. public static void main(String args[]){
18.   Circle c=new Circle();
19.   double result=c.area(5);
20.   System.out.println(result);
21. }
22. }
```

```
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. }
```

Java OOPs Concepts

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. }
```

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

Method Overriding in Java

If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in java**.

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

Usage of Java Method Overriding

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

Rules for Java Method Overriding

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

Understanding the problem without method overriding

Let's understand the problem that we may face in the program if we don't use method overriding.

```
1. class Vehicle{
2.   void run(){System.out.println("Vehicle is running");}
3. }
4. class Bike extends Vehicle{
5.
6.   public static void main(String args[]){
7.   Bike obj = new Bike();
8.   obj.run();
9.   }
10. }
```

```
Output:Vehicle is running
```

Problem is that I have to provide a specific implementation of run() method in subclass that is why we use method overriding.

Example of method overriding

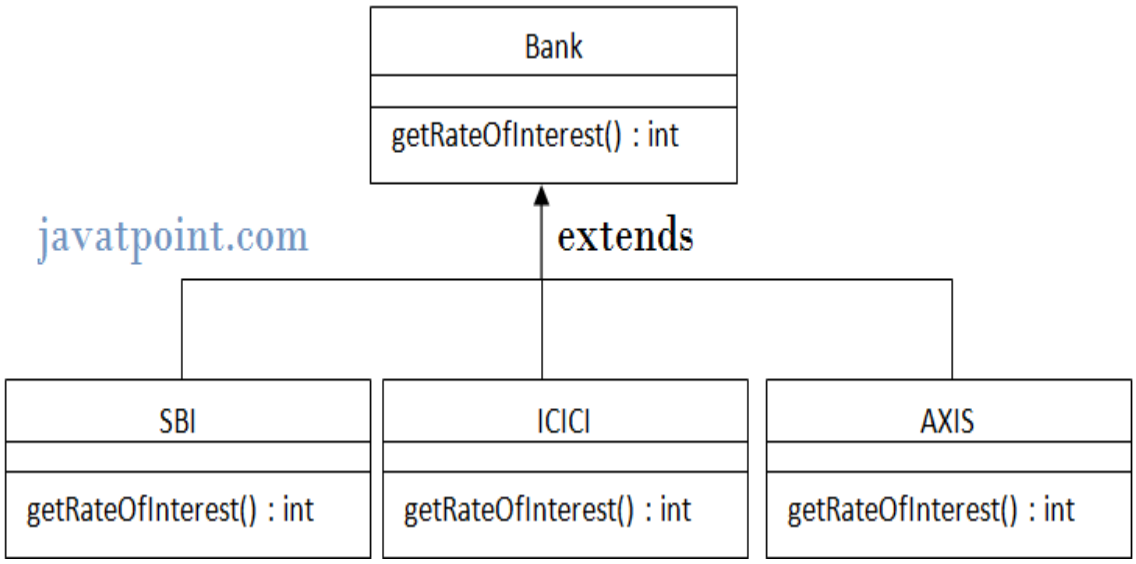
In this example, we have defined the run method in the subclass as defined in the parent class but it has some specific implementation. The name and parameter of the method is same and there is IS-A relationship between the classes, so there is method overriding.

```
1. class Vehicle{
2. void run(){System.out.println("Vehicle is running");}
3. }
4. class Bike2 extends Vehicle{
5. void run(){System.out.println("Bike is running safely");}
6.
7. public static void main(String args[]){
8. Bike2 obj = new Bike2();
9. obj.run();
10. }
```

```
Output:Bike is running safely
```

Real example of Java Method Overriding

Consider a scenario, Bank is a class that provides functionality to get rate of interest. But, rate of interest varies according to banks. For example, SBI, ICICI and AXIS banks could provide 8%, 7% and 9% rate of interest.




```
1. class Bank{
2. int getRateOfInterest(){return 0;}
3. }
4.
5. class SBI extends Bank{
6. int getRateOfInterest(){return 8;}
7. }
8.
```

```
9.  class ICICI extends Bank{
10. int getRateOfInterest(){return 7;}
11. }
12. class AXIS extends Bank{
13. int getRateOfInterest(){return 9;}
14. }
15.
16. class Test2{
17. public static void main(String args[]){
18. SBI s=new SBI();
19. ICICI i=new ICICI();
20. AXIS a=new AXIS();
21. System.out.println("SBI Rate of Interest: "+s.getRateOfInterest());
22. System.out.println("ICICI Rate of Interest: "+i.getRateOfInterest());
23. System.out.println("AXIS Rate of Interest: "+a.getRateOfInterest());
24. }
25. }
```

```
Output:
SBI Rate of Interest: 8
ICICI Rate of Interest: 7
AXIS Rate of Interest: 9
```

Can we override static method?

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

Why we cannot override static method?

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

Can we override java main method?

No, because main is a static method.

Java OOPs Concepts

Difference between method Overloading and Method Overriding in java

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

No.	Method Overloading	Method Overriding
1)	Method overloading is used *to increase the readability* of the program.	Method overriding is used *to provide the specific implementation* of the method that is already provided by its super class.
2)	Method overloading is performed *within class*.	Method overriding occurs *in two classes* that have IS-A (inheritance) relationship.
3)	In case of method overloading, *parameter must be different*.	In case of method overriding, *parameter must be same*.
4)	Method overloading is the example of *compile time polymorphism*.	Method overriding is the example of *run time polymorphism*.
5)	In java, method overloading can't be performed by changing return type of the method only. *Return type can be same or different* in method overloading. But you must have to change the parameter.	*Return type must be same or covariant* in method overriding.

Java Method Overloading example

```
1. class OverloadingExample{
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. }
```

Java Method Overriding example

```
1. class Animal{
2. void eat(){System.out.println("eating...");}
3. }
4. class Dog extends Animal{
5. void eat(){System.out.println("eating bread...");}
6.
```

Covariant Return Type

The covariant return type specifies that the return type may vary in the same direction as the subclass.

Before Java5, it was not possible to override any method by changing the return type. But now, since Java5, it is possible to override method by changing the return type if subclass overrides any method whose return type is Non-Primitive but it changes its return type to subclass type. Let's take a simple example:

Note: If you are beginner to java, skip this topic and return to it after OOPs concepts.

Simple example of Covariant Return Type

```
1. class A{
2. A get(){return this;}
3. }
4.
5. class B1 extends A{
6. B1 get(){return this;}
7. void message(){System.out.println("welcome to covariant return type");}
8.
9. public static void main(String args[]){
10. new B1().get().message();
11. }
12. }
```

```
Output:welcome to covariant return type
```

As you can see in the above example, the return type of the get() method of A class is A but the return type of the get() method of B class is B. Both methods have different return type but it is method overriding. This is known as covariant return type.

super keyword in java

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

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

Usage of java super Keyword

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

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

Problem without super keyword

```
1. class Vehicle{
2.   int speed=50;
3. }
4. class Bike3 extends Vehicle{
5.   int speed=100;
6.   void display(){
7.    System.out.println(speed);//will print speed of Bike
8.   }
9.   public static void main(String args[]){
10.   Bike3 b=new Bike3();
11.   b.display();
12. }
13. }
```

```
Output:100
```

In the above example Vehicle and Bike both class have a common property speed. Instance variable of current class is refered by instance bydefault, but I have to refer parent class instance variable that is why we use super keyword to distinguish between parent class instance variable and current class instance variable.

Solution by super keyword

```
1. //example of super keyword
2.
3. class Vehicle{
4.   int speed=50;
5. }
6.
7. class Bike4 extends Vehicle{
8.   int speed=100;
```

```
9.
10.  void display(){
11.   System.out.println(super.speed);//will print speed of Vehicle now
12.  }
13.  public static void main(String args[]){
14.   Bike4 b=new Bike4();
15.   b.display();
16.
17. }
18. }
```

```
Output:50
```

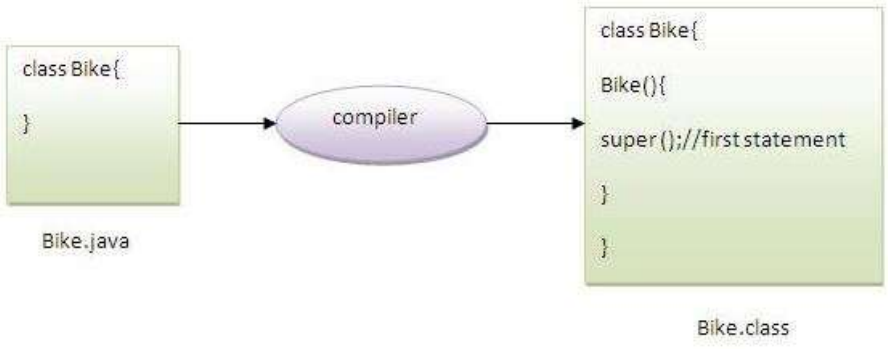
2) super is used to invoke parent class constructor.

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

```
1.  class Vehicle{
2.    Vehicle(){System.out.println("Vehicle is created");}
3.  }
4.
5.  class Bike5 extends Vehicle{
6.    Bike5(){
7.     super();//will invoke parent class constructor
8.     System.out.println("Bike is created");
9.    }
10.  public static void main(String args[]){
11.   Bike5 b=new Bike5();
12.
13. }
14. }
```

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

Note: super() is added in each class constructor automatically by compiler.



As we know well that default constructor is provided by compiler automatically but it also adds super() for the first statement.If you are creating your own constructor and you don't have either this() or super() as the first statement, compiler will provide super() as the first statement of the constructor.

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

```
1. class Vehicle{
2.   Vehicle(){System.out.println("Vehicle is created");}
3. }
4.
5. class Bike6 extends Vehicle{
6.   int speed;
7.   Bike6(int speed){
8.     this.speed=speed;
9.     System.out.println(speed);
10.  }
11.  public static void main(String args[]){
12.   Bike6 b=new Bike6(10);
13. }
14. }
```

```
Output:Vehicle is created
       10
```

3) super can be used to invoke parent class method

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

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

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

In the above example Student and Person both classes have message() method if we call message() method from Student class, it will call the message() method of Student class not of Person class because priority is given to local.

In case there is no method in subclass as parent, there is no need to use super. In the example given below message() method is invoked from Student class but Student class does not have message() method, so you can directly call message() method.

Program in case super is not required

```
1. class Person{
2. void message(){System.out.println("welcome");}
3. }
4.
5. class Student17 extends Person{
6.
7. void display(){
8. message();//will invoke parent class message() method
9. }
10.
11. public static void main(String args[]){
12. Student17 s=new Student17();
13. s.display();
14. }
15. }
```

```
Output:welcome
```

Instance initializer block:

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

The initialization of the instance variable can be directly but there can be performed extra operations while initializing the instance variable in the instance initializer block.

Que) What is the use of instance initializer block while we can directly assign a value in instance data member? For example:

```
1. class Bike{
2.     int speed=100;
3. }
```

Why use instance initializer block?

Suppose I have to perform some operations while assigning value to instance data member e.g. a for loop to fill a complex array or error handling etc.

Example of instance initializer block

Let's see the simple example of instance initializer block the performs initialization.

```
1. class Bike7{
2.     int speed;
3.
4.     Bike7(){System.out.println("speed is "+speed);}
5.
6.     {speed=100;}
7.
8.     public static void main(String args[]){
9.     Bike7 b1=new Bike7();
10.    Bike7 b2=new Bike7();
11.    }
12. }
```

```
Output:speed is 100
       speed is 100
```

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

1. method
2. constructor
3. block

Java OOPs Concepts

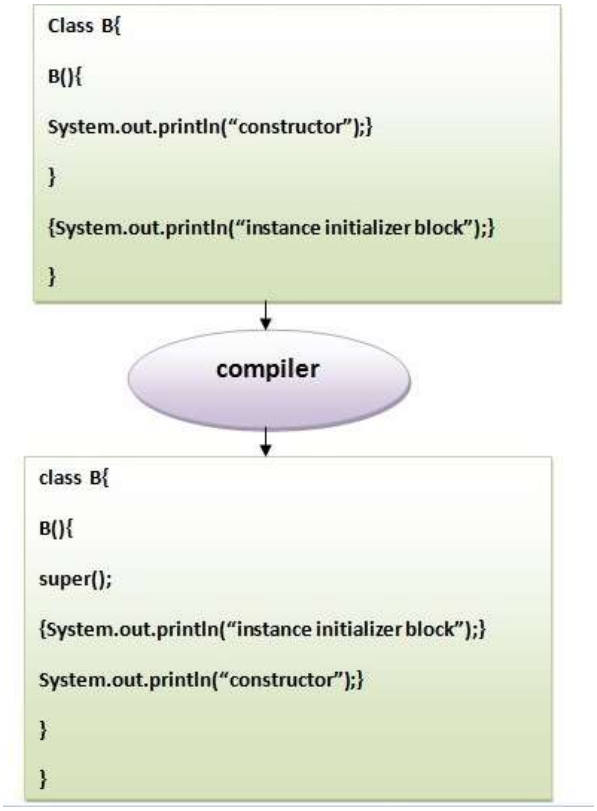
What is invoked firstly instance initializer block or constructor?

```
1. class Bike8{
2.     int speed;
3.
4.     Bike8(){System.out.println("constructor is invoked");}
5.
6.     {System.out.println("instance initializer block invoked");}
7.
8.     public static void main(String args[]){
9.     Bike8 b1=new Bike8();
10.    Bike8 b2=new Bike8();
11.    }
12. }
```

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

In the above example, it seems that instance initializer block is firstly invoked but NO. Instance intializer block is invoked at the time of object creation. The java compiler copies the instance initializer block in the constructor after the first statement super(). So firstly, constructor is invoked. Let's understand it by the figure given below:

Note: The java compiler copies the code of instance initializer block in every constructor.



Rules for instance initializer block :

There are mainly three rules for the instance initializer block. They are as follows:

1. The instance initializer block is created when instance of the class is created.
2. The instance initializer block is invoked after the parent class constructor is invoked (i.e. after super() constructor call).
3. The instance initializer block comes in the order in which they appear.

Program of instance initializer block that is invoked after super()

```
1. class A{
2. A(){
3. System.out.println("parent class constructor invoked");
4. }
5. }
6. class B2 extends A{
7. B2(){
8. super();
9. System.out.println("child class constructor invoked");
10. }
11. {System.out.println("instance initializer block is invoked");}
12.
13. public static void main(String args[]){
14. B2 b=new B2();
15. }
16. }
```

```
Output:parent class constructor invoked
       instance initializer block is invoked
       child class constructor invoked
```

Another example of instance block

```
1. class A{
2. A(){
3. System.out.println("parent class constructor invoked");
4. }
5. }
6.
7. class B3 extends A{
8. B3(){
9. super();
10. System.out.println("child class constructor invoked");
11. }
12.
13. B3(int a){
14. super();
15. System.out.println("child class constructor invoked "+a);
16. }
17.
18. {System.out.println("instance initializer block is invoked");}
19.
20. public static void main(String args[]){
21. B3 b1=new B3();
22. B3 b2=new B3(10);
23. }
24. }
```

```
Output:parent class constructor invoked
       instance initializer block is invoked
       child class constructor invoked
       parent class constructor invoked
       instance initializer block is invoked
       child class constructor invoked 10
```

Final Keyword In Java

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

1. variable
2. method
3. class

The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only. We will have detailed learning of these. Let's first learn the basics of final keyword.

1) Java final variable

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

Example of final variable

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

```
1. class Bike9{
2.  final int speedlimit=90;//final variable
3.  void run(){
4.   speedlimit=400;
5.  }
6.  public static void main(String args[]){
7.  Bike9 obj=new  Bike9();
8.  obj.run();
9.  }
10. }//end of class
```

```
Output:Compile Time Error
```

2) Java final method

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

Example of final method

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

```
Output:Compile Time Error
```

3) Java final class

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

Example of final class

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

```
Output:Compile Time Error
```

Q) Is final method inherited?

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

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

```
Output:running...
```

Q) What is blank or uninitialized final variable?

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

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

It can be initialized only in constructor.

Example of blank final variable

```
1. class Student{
2. int id;
3. String name;
4. final String PAN_CARD_NUMBER;
5. ...
6. }
```

Que) Can we initialize blank final variable?

Yes, but only in constructor. For example:

```
1.  class Bike10{
2.   final int speedlimit;//blank final variable
3.
4.   Bike10(){
5.   speedlimit=70;
6.   System.out.println(speedlimit);
7.   }
8.
9.   public static void main(String args[]){
10.    new Bike10();
11. }
12.}
```

```
Output:70
```

static blank final variable

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

Example of static blank final variable

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

Q) What is final parameter?

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

```
1. class Bike11{
2.   int cube(final int n){
3.    n=n+2;//can't be changed as n is final
4.    n*n*n;
5.   }
6.   public static void main(String args[]){
7.     Bike11 b=new Bike11();
8.     b.cube(5);
9.  }
10. }
```

```
Output:Compile Time Error
```

Q) Can we declare a constructor final?

No, because constructor is never inherited.

Polymorphism in Java

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

There are two types of polymorphism in java: compile time polymorphism and runtime polymorphism. We can perform polymorphism in java by method overloading and method overriding.

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

Runtime Polymorphism in Java

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

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

Let's first understand the upcasting before Runtime Polymorphism.

Upcasting

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


```
1. class A{}
2. class B extends A{}
```

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

Example of Java Runtime Polymorphism

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

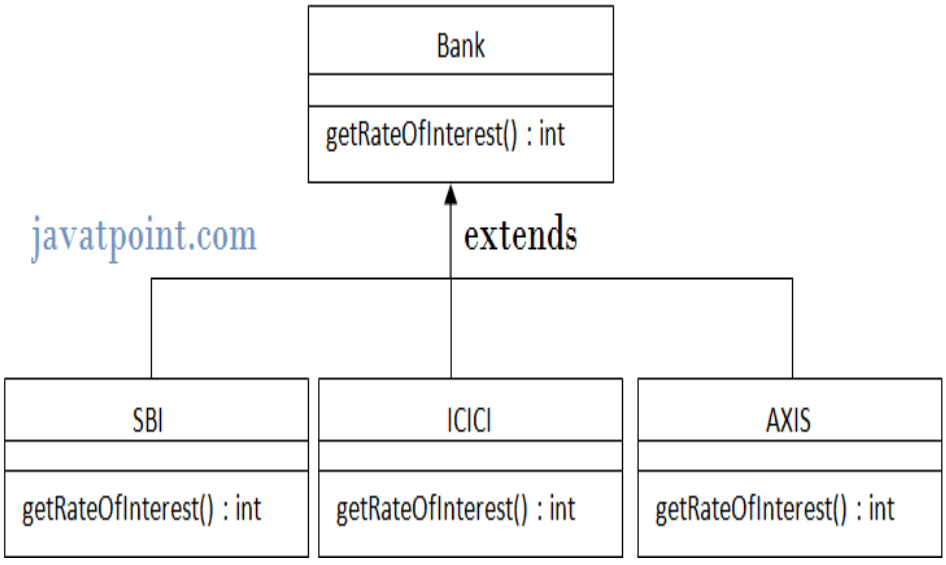
Since method invocation is determined by the JVM not compiler, it is known as runtime polymorphism.

```
1. class Bike{
2.   void run(){System.out.println("running");}
3. }
4. class Splender extends Bike{
5.   void run(){System.out.println("running safely with 60km");}
6.
7.   public static void main(String args[]){
8.     Bike b = new Splender();//upcasting
9.     b.run();
10.  }
11. }
```

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

Real example of Java Runtime Polymorphism

Consider a scenario, Bank is a class that provides method to get the rate of interest. But, rate of interest may differ according to banks. For example, SBI, ICICI and AXIS banks could provide 8%, 7% and 9% rate of interest.



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

```
1. class Bank{
2. int getRateOfInterest(){return 0;}
3. }
4.
5. class SBI extends Bank{
6. int getRateOfInterest(){return 8;}
7. }
8.
9. class ICICI extends Bank{
10. int getRateOfInterest(){return 7;}
11. }
12. class AXIS extends Bank{
13. int getRateOfInterest(){return 9;}
14. }
15.
16. class Test3{
17. public static void main(String args[]){
18. Bank b1=new SBI();
19. Bank b2=new ICICI();
20. Bank b3=new AXIS();
21. System.out.println("SBI Rate of Interest: "+b1.getRateOfInterest());
22. System.out.println("ICICI Rate of Interest: "+b2.getRateOfInterest());
23. System.out.println("AXIS Rate of Interest: "+b3.getRateOfInterest());
24. }
25. }
```

```
Output:
SBI Rate of Interest: 8
ICICI Rate of Interest: 7
AXIS Rate of Interest: 9
```

Java Runtime Polymorphism with data member

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

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

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

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

```
Output:90
```

Java Runtime Polymorphism with Multilevel Inheritance

Let's see the simple example of Runtime Polymorphism with multilevel inheritance.

```
1. class Animal{
2. void eat(){System.out.println("eating");}
3. }
4.
5. class Dog extends Animal{
6. void eat(){System.out.println("eating fruits");}
7. }
8.
9. class BabyDog extends Dog{
10. void eat(){System.out.println("drinking milk");}
11.
12. public static void main(String args[]){
13. Animal a1,a2,a3;
14. a1=new Animal();
15. a2=new Dog();
16. a3=new BabyDog();
17.
18. a1.eat();
19. a2.eat();
20. a3.eat();
21. }
22. }
```

```
Output: eating
        eating fruits
        drinking Milk
```

Try for Output

```
1. class Animal{
2. void eat(){System.out.println("animal is eating...");}
3. }
4.
5. class Dog extends Animal{
6. void eat(){System.out.println("dog is eating...");}
7. }
8.
9. class BabyDog1 extends Dog{
10. public static void main(String args[]){
11. Animal a=new BabyDog1();
12. a.eat();
13. }}
```

```
Output: Dog is eating
```

Since, BabyDog is not overriding the eat() method, so eat() method of Dog class is invoked.

Static Binding and Dynamic Binding

Connecting a method call to the method body is known as binding.

There are two types of binding

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

Understanding Type

Let's understand the type of instance.

1) variables have a type

Each variable has a type, it may be primitive and non-primitive.

```
1. int data=30;
```

Here data variable is a type of int.

2) References have a type

```
1. class Dog{
2.  public static void main(String args[]){
3.   Dog d1;//Here d1 is a type of Dog
4.  }
5. }
```

3) Objects have a type

An object is an instance of particular java class,but it is also an instance of its superclass.

```
1. class Animal{}
2. class Dog extends Animal{
3.  public static void main(String args[]){
4.   Dog d1=new Dog();
5.  }
6. }
```

Here d1 is an instance of Dog class, but it is also an instance of Animal.

static binding

When type of the object is determined at compiled time(by the compiler), it is known as static binding. If there is any private, final or static method in a class, there is static binding.

Java OOPs Concepts

Example of static binding

```
1. class Dog{
2.  private void eat(){System.out.println("dog is eating...");}
3.
4.  public static void main(String args[]){
5.   Dog d1=new Dog();
6.   d1.eat();
7.  }
8. }
```

Dynamic binding

When type of the object is determined at run-time, it is known as dynamic binding or **Dynamic Method Dispatch**. In other words, it is the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed. That is, if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed.

Example of dynamic binding

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

```
Output:dog is eating...
```

In the above example object type cannot be determined by the compiler, because the instance of Dog is also an instance of Animal.So compiler doesn't know its type, only its base type.

Java instanceof

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

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

Simple example of java instanceof

Let's see the simple example of instance operator where it tests the current class.

```
1. class Simple1{
2.  public static void main(String args[]){
3.  Simple1 s=new Simple1();
4.  System.out.println(s instanceof Simple);//true
5.  }
6. }
```

```
Output:true
```

An object of subclass type is also a type of parent class. For example, if Dog extends Animal then object of Dog can be referred by either Dog or Animal class.

Another example of java instanceof operator

```
1. class Animal{}
2. class Dog1 extends Animal{//Dog inherits Animal
3.
4.  public static void main(String args[]){
5.  Dog1 d=new Dog1();
6.  System.out.println(d instanceof Animal);//true
7.  }
8. }
```

```
Output:true
```

instanceof in java with a variable that have null value

If we apply instanceof operator with a variable that have null value, it returns false. Let's see the example given below where we apply instanceof operator with the variable that have null value.

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

```
5.  }
6. }
```

```
Output:false
```

Downcasting with java instanceof operator

When Subclass type refers to the object of Parent class, it is known as downcasting. If we perform it directly, compiler gives Compilation error. If you perform it by typecasting, ClassCastException is thrown at runtime. But if we use instanceof operator, downcasting is possible.

```
1. Dog d=new Animal();//Compilation error
```

If we perform downcasting by typecasting, ClassCastException is thrown at runtime.

```
1. Dog d=(Dog)new Animal();
2. //Compiles successfully but ClassCastException is thrown at runtime
```

Possibility of downcasting with instanceof

Let's see the example, where downcasting is possible by instanceof operator.

```
1. class Animal { }
2.
3. class Dog3 extends Animal {
4.   static void method(Animal a) {
5.     if(a instanceof Dog3){
6.       Dog3 d=(Dog3)a;//downcasting
7.       System.out.println("ok downcasting performed");
8.     }
9.   }
10.
11.  public static void main (String [] args) {
12.    Animal a=new Dog3();
13.    Dog3.method(a);
14.  }
15.
16. }
```

```
Output:ok downcasting performed
```

Java OOPs Concepts

Downcasting without the use of java instanceof

Downcasting can also be performed without the use of instanceof operator as displayed in the following example:

```
1. class Animal { }
2. class Dog4 extends Animal {
3.   static void method(Animal a) {
4.        Dog4 d=(Dog4)a;//downcasting
5.        System.out.println("ok downcasting performed");
6.   }
7.    public static void main (String [] args) {
8.     Animal a=new Dog4();
9.     Dog4.method(a);
10.  }
11. }
```

```
Output:ok downcasting performed
```

Let's take closer look at this, actual object that is referred by a, is an object of Dog class. So if we downcast it, it is fine. But what will happen if we write:

```
1. Animal a=new Animal();
2. Dog.method(a);
3. //Now ClassCastException but not in case of instanceof operator
```

Java OOPs Concepts

Understanding Real use of instanceof in java

Let's see the real use of instanceof keyword by the example given below.

```
1. interface Printable{}
2. class A implements Printable{
3. public void a(){System.out.println("a method");}
4. }
5. class B implements Printable{
6. public void b(){System.out.println("b method");}
7. }
8.
9. class Call{
10. void invoke(Printable p){//upcasting
11. if(p instanceof A){
12. A a=(A)p;//Downcasting
13. a.a();
14. }
15. if(p instanceof B){
16. B b=(B)p;//Downcasting
17. b.b();
18. }
19.
20. }
21. }//end of Call class
22.
23. class Test4{
24. public static void main(String args[]){
25. Printable p=new B();
26. Call c=new Call();
27. c.invoke(p);
28. }
29. }
```

```
Output: b method
```

Java OOPs Concepts

Abstract class in Java

A class that is declared with abstract keyword, is known as abstract class in java. It can have abstract and non-abstract methods (method with body).

Before learning java abstract class, let's understand the abstraction in java first.

Abstraction in Java

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

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

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

Ways to achieve Abstaction

There are two ways to achieve abstraction in java

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

Abstract class in Java

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

Example abstract class

```
1. abstract class A{}
```

abstract method

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

Example abstract method

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

Example of abstract class that has abstract method

In this example, Bike the abstract class that contains only one abstract method run. It implementation is provided by the Honda class.

```
1. abstract class Bike{
2.   abstract void run();
3. }
4.
5. class Honda4 extends Bike{
6. void run(){System.out.println("running safely..");}
7.
8. public static void main(String args[]){
9.  Bike obj = new Honda4();
10. obj.run();
11. }
12. }
```

```
running safely..
```

Understanding the real scenario of abstract class

In this example, Shape is the abstract class, its implementation is provided by the Rectangle and Circle classes. Mostly, we don't know about the implementation class (i.e. hidden to the end user) and object of the implementation class is provided by the **factory method**.

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

In this example, if you create the instance of Rectangle class, draw() method of Rectangle class will be invoked.

File: TestAbstraction1.java

```
1. abstract class Shape{
2. abstract void draw();
3. }
4. //In real scenario, implementation is provided by others i.e. unknown by end user
5. class Rectangle extends Shape{
6. void draw(){System.out.println("drawing rectangle");}
7. }
8.
9. class Circle1 extends Shape{
10. void draw(){System.out.println("drawing circle");}
11. }
12.
13. //In real scenario, method is called by programmer or user
14. class TestAbstraction1{
15. public static void main(String args[]){
```

```
16. Shape s=new Circle1();//In real scenario, object is provided through method e.g. get
    Shape() method
17. s.draw();
18. }
19. }
```

```
drawing circle
```

Another example of abstract class in java

File: TestBank.java

```
1.  abstract class Bank{
2. abstract int getRateOfInterest();
3. }
4.
5. class SBI extends Bank{
6. int getRateOfInterest(){return 7;}
7. }
8. class PNB extends Bank{
9. int getRateOfInterest(){return 7;}
10. }
11.
12. class TestBank{
13. public static void main(String args[]){
14. Bank b=new SBI();//if object is PNB, method of PNB will be invoked
15. int interest=b.getRateOfInterest();
16. System.out.println("Rate of Interest is: "+interest+" %");
17. }}
```

```
Rate of Interest is: 7 %
```

Abstract class having constructor, data member, methods etc.

An abstract class can have data member, abstract method, method body, constructor and even main() method.

File: TestAbstraction2.java

```
1. //example of abstract class that have method body
2.  abstract class Bike{
3.    Bike(){System.out.println("bike is created");}
4.    abstract void run();
5.    void changeGear(){System.out.println("gear changed");}
6.  }
7.
8.  class Honda extends Bike{
9.  void run(){System.out.println("running safely..");}
10. }
11. class TestAbstraction2{
12. public static void main(String args[]){
13.  Bike obj = new Honda();
14.  obj.run();
```

```
15.  obj.changeGear();
16. }
17. }
```

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

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

```
1. class Bike12{
2. abstract void run();
3. }
```

```
compile time error
```

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

Another real scenario of abstract class

The abstract class can also be used to provide some implementation of the interface. In such case, the end user may not be forced to override all the methods of the interface.

Note: If you are beginner to java, learn interface first and skip this example.

```
1. interface A{
2. void a();
3. void b();
4. void c();
5. void d();
6. }
7.
8. abstract class B implements A{
9. public void c(){System.out.println("I am C");}
10. }
11.
12. class M extends B{
13. public void a(){System.out.println("I am a");}
14. public void b(){System.out.println("I am b");}
15. public void d(){System.out.println("I am d");}
16. }
17.
18. class Test5{
19. public static void main(String args[]){
20. A a=new M();
21. a.a();
22. a.b();
23. a.c();
24. a.d();
25. }}
```

Java OOPs Concepts

```
Output:I am a
       I am b
       I am c
       I am d
```

Interface in Java

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

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

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

It cannot be instantiated just like abstract class.

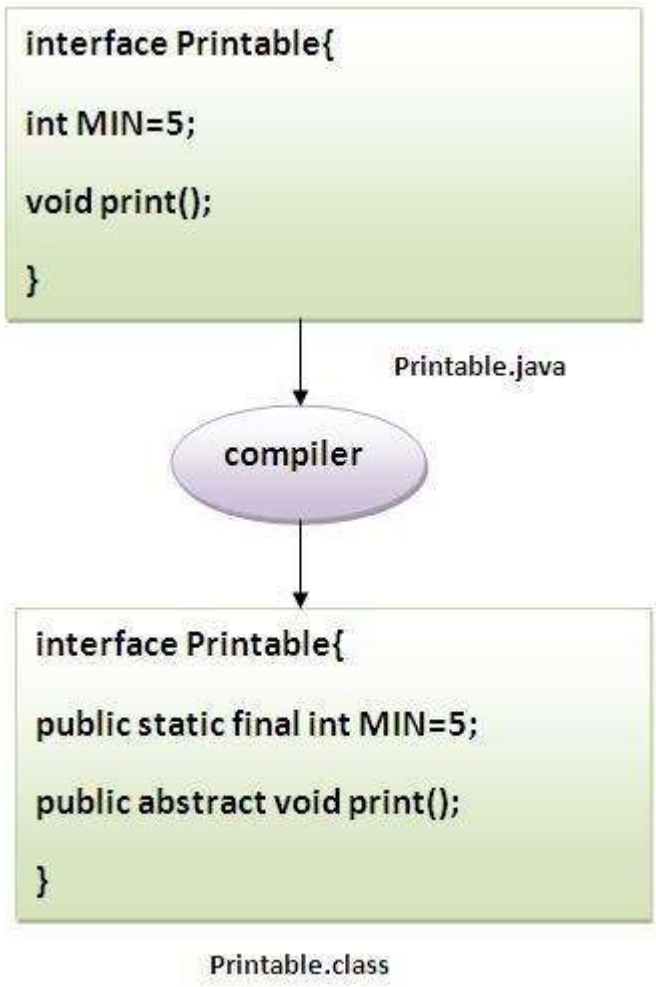
Why use Java interface?

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

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

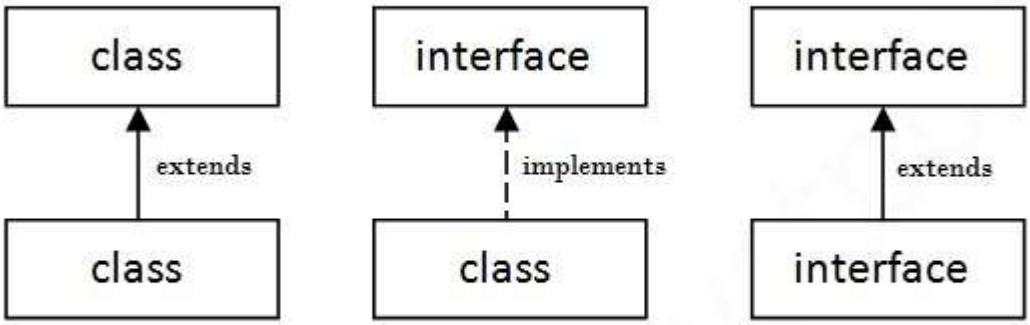
The java compiler adds public and abstract keywords before the interface method and public, static and final keywords before data members.

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



Understanding relationship between classes and interfaces

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



Simple example of Java interface

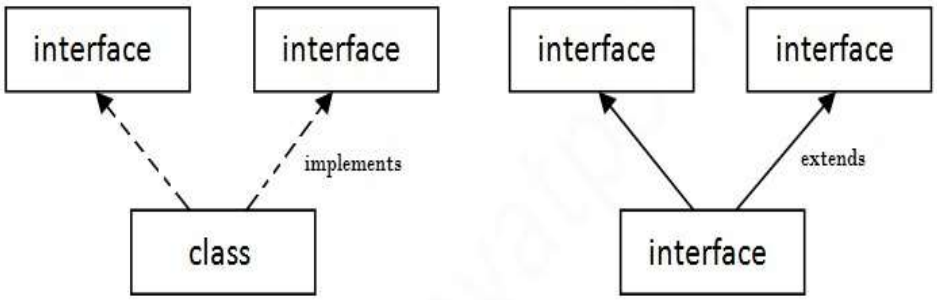
In this example, Printable interface have only one method, its implementation is provided in the A class.

```
1. interface printable{
2. void print();
3. }
4.
5. class A6 implements printable{
6. public void print(){System.out.println("Hello");}
7.
8. public static void main(String args[]){
9. A6 obj = new A6();
10. obj.print();
11. }
12. }
```

```
Output:Hello
```

Multiple inheritance in Java by interface

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



Multiple Inheritance in Java

```
1. interface Printable{
2. void print();
3. }
4.
5. interface Showable{
6. void show();
7. }
8.
9. class A7 implements Printable,Showable{
10.
11. public void print(){System.out.println("Hello");}
12. public void show(){System.out.println("Welcome");}
13.
14. public static void main(String args[]){
15. A7 obj = new A7();
16. obj.print();
17. obj.show();
18. }
19. }
```

```
Output:Hello
       Welcome
```

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

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

```
1. interface Printable{
2. void print();
3. }
4. interface Showable{
5. void print();
6. }
7.
8. class TestTnterface1 implements Printable,Showable{
9. public void print(){System.out.println("Hello");}
10. public static void main(String args[]){
11. TestTnterface1 obj = new TestTnterface1();
12. obj.print();
13. }
14. }
```

```
Hello
```

As you can see in the above example, Printable and Showable interface have same methods but its implementation is provided by class TestTnterface1, so there is no ambiguity.

Interface inheritance

A class implements interface but one interface extends another interface .

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

```
Hello
Welcome
```

Java OOPs Concepts

Q) What is marker or tagged interface?

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

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

Nested Interface in Java

Note: An interface can have another interface i.e. known as nested interface. We will learn it in detail in the nested classes chapter. For example:

```
1. interface printable{
2.  void print();
3.  interface MessagePrintable{
4.    void msg();
5.  }
6. }
```

Difference between abstract class and interface

Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated.

But there are many differences between abstract class and interface that are given below.

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

7) **Example:**
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

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

Java OOPs Concepts

Example of abstract class and interface in Java

Let's see a simple example where we are using interface and abstract class both.

```
1. //Creating interface that has 4 methods
2. interface A{
3. void a();//bydefault, public and abstract
4. void b();
5. void c();
6. void d();
7. }
8.
9. //Creating abstract class that provides the implementation of one method of A interfa
   ce
10. abstract class B implements A{
11. public void c(){System.out.println("I am C");}
12. }
13.
14. //Creating subclass of abstract class, now we need to provide the implementation of r
    est of the methods
15. class M extends B{
16. public void a(){System.out.println("I am a");}
17. public void b(){System.out.println("I am b");}
18. public void d(){System.out.println("I am d");}
19. }
20.
21. //Creating a test class that calls the methods of A interface
22. class Test5{
23. public static void main(String args[]){
24. A a=new M();
25. a.a();
26. a.b();
27. a.c();
28. a.d();
29. }}
```

Output:

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

Java Package

A **java package** is a group of similar types of classes, interfaces and sub-packages.

Package in java can be categorized in two form, built-in package and user-defined package.

There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

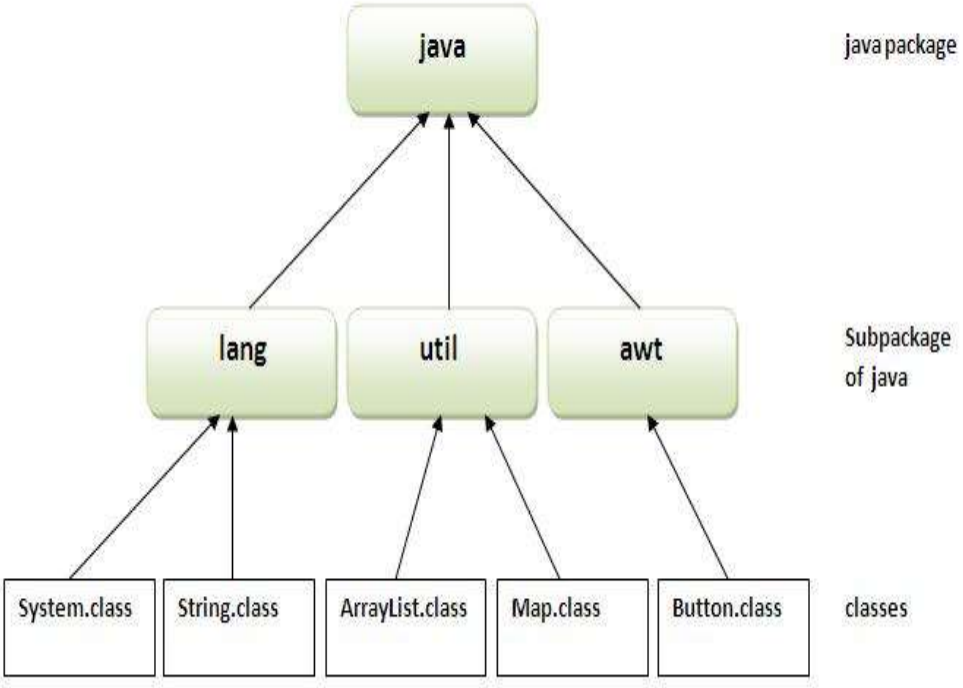
Here, we will have the detailed learning of creating and using user-defined packages.

Advantage of Java Package

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.

2) Java package provides access protection.

3) Java package removes naming collision.



Java OOPs Concepts

Simple example of java package

The **package keyword** is used to create a package in java.

```
1. //save as Simple.java
2. package mypack;
3. public class Simple{
4.  public static void main(String args[]){
5.     System.out.println("Welcome to package");
6.    }
7. }
```

How to compile java package

If you are not using any IDE, you need to follow the **syntax** given below:

```
1. javac -d directory javafilename
```

For **example**

```
1. javac -d . Simple.java
```

The -d switch specifies the destination where to put the generated class file. You can use any directory name like /home (in case of Linux), d:/abc (in case of windows) etc. If you want to keep the package within the same directory, you can use . (dot).

How to run java package program

You need to use fully qualified name e.g. mypack.Simple etc to run the class.

To Compile: javac -d . Simple.java

To Run: java mypack.Simple

```
Output:Welcome to package
```

The -d is a switch that tells the compiler where to put the class file i.e. it represents destination. The . represents the current folder.

How to access package from another package?

There are three ways to access the package from outside the package.

1. import package.*;
2. import package.classname;
3. fully qualified name.

Java OOPs Concepts

1) Using packagename.*

If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages.

The import keyword is used to make the classes and interface of another package accessible to the current package.

Example of package that import the packagename.*

```
1. //save by A.java
2.
3. package pack;
4. public class A{
5.   public void msg(){System.out.println("Hello");}
6. }
```

```
1. //save by B.java
2.
3. package mypack;
4. import pack.*;
5.
6. class B{
7.   public static void main(String args[]){
8.    A obj = new A();
9.    obj.msg();
10.  }
11. }
```

```
Output:Hello
```

2) Using packagename.classname

If you import package.classname then only declared class of this package will be accessible.

Example of package by import package.classname

```
1. //save by A.java
2.
3. package pack;
4. public class A{
5.   public void msg(){System.out.println("Hello");}
6. }
```

```
1. //save by B.java
2.
3. package mypack;
4. import pack.A;
5.
6. class B{
7.   public static void main(String args[]){
8.    A obj = new A();
9.    obj.msg();
10.  }
11. }
```

```
Output:Hello
```

3) Using fully qualified name

If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.

It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

Example of package by import fully qualified name

```
1. //save by A.java
2.
3. package pack;
4. public class A{
5.   public void msg(){System.out.println("Hello");}
6. }
```

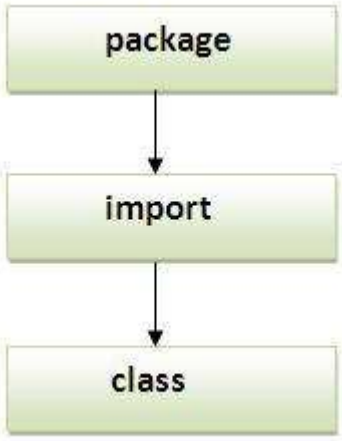
```
1. //save by B.java
2.
3. package mypack;
4. class B{
5.   public static void main(String args[]){
6.    pack.A obj = new pack.A();//using fully qualified name
7.    obj.msg();
8.   }
9. }
```

```
Output:Hello
```

Note: If you import a package, subpackages will not be imported.

If you import a package, all the classes and interface of that package will be imported excluding the classes and interfaces of the subpackages. Hence, you need to import the subpackage as well.

Note: Sequence of the program must be package then import then class.



Subpackage in java

Package inside the package is called the **subpackage**. It should be created **to categorize the package further**.

Let's take an example, Sun Microsystem has definded a package named java that contains many classes like System, String, Reader, Writer, Socket etc. These classes represent a particular group e.g. Reader and Writer classes are for Input/Output operation, Socket and ServerSocket classes are for networking etc and so on. So, Sun has subcategorized the java package into subpackages such as lang, net, io etc. and put the Input/Output related classes in io package, Server and ServerSocket classes in net packages and so on.

> ***The standard of defining package is domain.company.package e.g. com.javatpoint.bean or org.sssit.dao.***

Example of Subpackage

```
1. package com.javatpoint.core;
2. class Simple{
3.   public static void main(String args[]){
4.    System.out.println("Hello subpackage");
5.   }
6. }
```

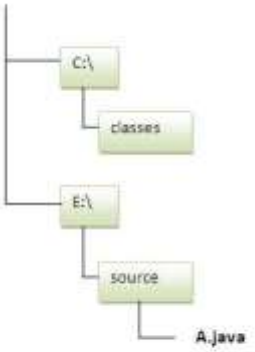
To Compile: javac -d . Simple.java

To Run: java com.javatpoint.core.Simple

```
Output:Hello subpackage
```

How to send the class file to another directory or drive?

There is a scenario, I want to put the class file of A.java source file in classes folder of c: drive. For example:




```
1. //save as Simple.java
2.
3. package mypack;
4. public class Simple{
5.  public static void main(String args[]){
6.    System.out.println("Welcome to package");
7.   }
8. }
```

To Compile:

e:\sources> javac -d c:\classes Simple.java

To Run:

To run this program from e:\source directory, you need to set classpath of the directory where the class file resides.

e:\sources> set classpath=c:\classes;.;

e:\sources> java mypack.Simple

Another way to run this program by -classpath switch of java:

The -classpath switch can be used with javac and java tool.

To run this program from e:\source directory, you can use -classpath switch of java that tells where to look for class file. For example:

e:\sources> java -classpath c:\classes mypack.Simple

```
Output:Welcome to package
```

Java OOPs Concepts

Ways to load the class files or jar files

There are two ways to load the class files temporary and permanent.

- Temporary
  - By setting the classpath in the command prompt
  - By -classpath switch
- Permanent
  - By setting the classpath in the environment variables
  - By creating the jar file, that contains all the class files, and copying the jar file in the jre/lib/ext folder.

Rule: There can be only one public class in a java source file and it must be saved by the public class name.

```
1. //save as C.java otherwise Compilte Time Error
2.
3. class A{}
4. class B{}
5. public class C{}
```

How to put two public classes in a package?

If you want to put two public classes in a package, have two java source files containing one public class, but keep the package name same. For example:

```
1. //save as A.java
2.
3. package javatpoint;
4. public class A{}
```

```
1. //save as B.java
2.
3. package javatpoint;
4. public class B{}
```

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

There are many non-access modifiers such as static, abstract, synchronized, native, volatile, transient etc. Here, we will learn access modifiers.

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.

```
1. class A{
2. private int data=40;
3. private void msg(){System.out.println("Hello java");}
4. }
5.
6. public class Simple{
7.  public static void main(String args[]){
8.    A obj=new A();
9.    System.out.println(obj.data);//Compile Time Error
10.   obj.msg();//Compile Time Error
11.   }
12. }
```

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:

```
1. class A{
2. private A(){}//private constructor
3. void msg(){System.out.println("Hello java");}
4. }
5. public class Simple{
6.  public static void main(String args[]){
7.    A obj=new A();//Compile Time Error
8.  }
9. }
```

Note: A class cannot be private or protected except nested class.

2) default access modifier

If you don't use any modifier, it is treated as **default** bydefault. 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.

```
1. //save by A.java
2. package pack;
3. class A{
4.   void msg(){System.out.println("Hello");}
5. }
```

```
1. //save by B.java
2. package mypack;
3. import pack.*;
4. class B{
5.   public static void main(String args[]){
6.    A obj = new A();//Compile Time Error
7.    obj.msg();//Compile Time Error
8.   }
9. }
```

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.

```
1. //save by A.java
2. package pack;
3. public class A{
4. protected void msg(){System.out.println("Hello");}
5. }
```

```
1. //save by B.java
2. package mypack;
3. import pack.*;
4.
5. class B extends A{
6.   public static void main(String args[]){
7.    B obj = new B();
8.    obj.msg();
9.   }
10. }
```

```
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

```
1. //save by A.java
2.
3. package pack;
4. public class A{
5. public void msg(){System.out.println("Hello");}
6. }
```

```
1. //save by B.java
2.
3. package mypack;
4. import pack.*;
5.
6. class B{
7.   public static void main(String args[]){
8.    A obj = new A();
9.    obj.msg();
10.  }
11. }
```

```
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.

Encapsulation in Java

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

We can create a fully encapsulated class in java by making all the data members of the class private. Now we can use setter and getter methods to set and get the data in it.

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

Advantage of Encapsulation in java

By providing only setter or getter method, you can make the class **read-only or write-only**.

It provides you the **control over the data**. Suppose you want to set the value of id i.e. greater than 100 only, you can write the logic inside the setter method.

Simple example of encapsulation in java

Let's see the simple example of encapsulation that has only one field with its setter and getter methods.

```
1. //save as Student.java
2. package com.javatpoint;
3. public class Student{
4. private String name;
5. public String getName(){
6. return name;
7. }
8. public void setName(String name){
9. this.name=name
10. }
11. }
```

```
1. //save as Test.java
2. package com.javatpoint;
3. class Test{
4. public static void main(String[] args){
5. Student s=new Student();
6. s.setname("vijay");
7. System.out.println(s.getName());
8. }
9. }
```

```
Compile By: javac -d . Test.java
Run By: java com.javatpoint.Test     Output: vijay
```

Object class in Java

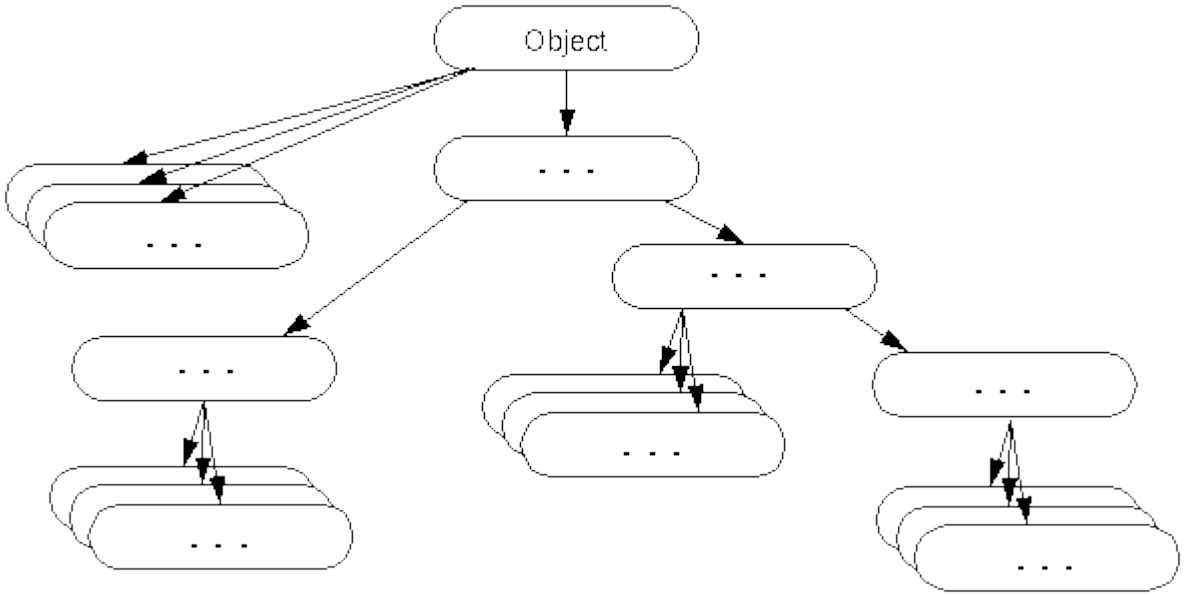
The **Object class** is the parent class of all the classes in java by-default. In other words, it is the topmost class of java.

The Object class is beneficial if you want to refer any object whose type you don't know. Notice that parent class reference variable can refer the child class object, known as upcasting.

Let's take an example, there is getObject() method that returns an object but it can be of any type like Employee, Student etc, we can use Object class reference to refer that object. For example:

1. Object obj=getObject();//we don't what object would be returned from this method

The Object class provides some common behaviour to all the objects such as object can be compared, object can be cloned, object can be notified etc.



Java OOPs Concepts

Methods of Object class

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

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

Object Cloning in Java

The **object cloning** is a way to create exact copy of an object. For this purpose, clone() method of Object class is used to clone an object.

The **java.lang.Cloneable interface** must be implemented by the class whose object clone we want to create. If we don't implement Cloneable interface, clone() method generates **CloneNotSupportedException**.

The **clone() method** is defined in the Object class. Syntax of the clone() method is as follows:

1. **protected** Object clone() **throws** CloneNotSupportedException

Why use clone() method ?

The **clone() method** saves the extra processing task for creating the exact copy of an object. If we perform it by using the new keyword, it will take a lot of processing to be performed that is why we use object cloning.

Advantage of Object cloning

Less processing task.

Example of clone() method (Object cloning)

Let's see the simple example of object cloning

```
1. class Student18 implements Cloneable{
2. int rollno;
3. String name;
4.
5. Student18(int rollno,String name){
6. this.rollno=rollno;
7. this.name=name;
8. }
9.
10. public Object clone()throws CloneNotSupportedException{
11. return super.clone();
12. }
13.
14. public static void main(String args[]){
15. try{
16. Student18 s1=new Student18(101,"amit");
17.
18. Student18 s2=(Student18)s1.clone();
19.
20. System.out.println(s1.rollno+" "+s1.name);
```

```
21. System.out.println(s2.rollno+" "+s2.name);
22.
23. }catch(CloneNotSupportedException c){}
24.
25. }
26. }
```

```
Output:101 amit
       101 amit
```

download the example of object cloning

As you can see in the above example, both reference variables have the same value. Thus, the clone() copies the values of an object to another. So we don't need to write explicit code to copy the value of an object to another.

If we create another object by new keyword and assign the values of another object to this one, it will require a lot of processing on this object. So to save the extra processing task we use clone() method.

Demerits of clone

Cloning can cause unintentional side effects. For example, if the object cloned contains a reference variable called *objRef*, then when the clone is made, *objRef* in the clone will refer to the same object as does *objRef* in the original. If the clone makes a change to the contents of the object referred to by objRef, then it will be changed for the original object too.

For example, if an object opens an I?O stream and is then cloned, two objects will be capable of operating on the same stream. And, if one of these objects closes the stream, the othr object might still attempt to write to it, causing an error.

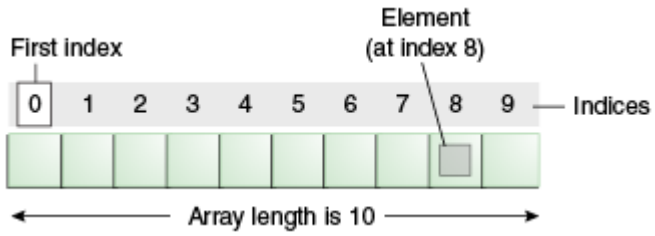
To handle these types of problems, it requires to override the clone() method defined by Object class.

Java Array

Normally, array is a collection of similar type of elements that have contiguous memory location.

Java array is an object the contains elements of similar data type. It is a data structure where we store similar elements. We can store only fixed set of elements in a java array.

Array in java is index based, first element of the array is stored at 0 index.



Advantage of Java Array

- **Code Optimization:** It makes the code optimized, we can retrieve or sort the data easily.
- **Random access:** We can get any data located at any index position.

Disadvantage of Java Array

- **Size Limit:** We can store only fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in java.

Types of Array in java

There are two types of array.

- Single Dimensional Array
- Multidimensional Array

Single Dimensional Array in java

Syntax to Declare an Array in java

```
1. dataType[] arr; (or)
2. dataType []arr; (or)
3. dataType arr[];
```

Java OOPs Concepts

Instantiation of an Array in java

1. arrayRefVar=**new** datatype[size];

Example of single dimensional java array

Let's see the simple example of java array, where we are going to declare, instantiate, initialize and traverse an array.

```
1. class Testarray{
2. public static void main(String args[]){
3.
4. int a[]=new int[5];//declaration and instantiation
5. a[0]=10;//initialization
6. a[1]=20;
7. a[2]=70;
8. a[3]=40;
9. a[4]=50;
10.
11. //printing array
12. for(int i=0;i<a.length;i++)//length is the property of array
13. System.out.println(a[i]);
14.
15. }}
```

```
Output: 10
        20
        70
        40
        50
```

Declaration, Instantiation and Initialization of Java Array

We can declare, instantiate and initialize the java array together by:

```
1. int a[]={33,3,4,5};//declaration, instantiation and initialization
```

Let's see the simple example to print this array.

```
1. class Testarray1{
2. public static void main(String args[]){
3.
4. int a[]={33,3,4,5};//declaration, instantiation and initialization
5.
6. //printing array
7. for(int i=0;i<a.length;i++)//length is the property of array
8. System.out.println(a[i]);
9.
10. }}
```

```
Output:33
       3
       4
       5
```

Passing Array to method in java

We can pass the java array to method so that we can reuse the same logic on any array.

Let's see the simple example to get minimum number of an array using method.

```
1. class Testarray2{
2. static void min(int arr[]){
3. int min=arr[0];
4. for(int i=1;i<arr.length;i++)
5.  if(min>arr[i])
6.   min=arr[i];
7.
8. System.out.println(min);
9. }
10.
11. public static void main(String args[]){
12.
13. int a[]={33,3,4,5};
14. min(a);//passing array to method
15.
16. }}
```

```
Output:3
```

Multidimensional array in java

In such case, data is stored in row and column based index (also known as matrix form).

Syntax to Declare Multidimensional Array in java

```
1. dataType[][] arrayRefVar; (or)
2. dataType [][]arrayRefVar; (or)
3. dataType arrayRefVar[][]; (or)
4. dataType []arrayRefVar[];
```

Example to instantiate Multidimensional Array in java

```
1. int[][] arr=new int[3][3];//3 row and 3 column
```

Example to initialize Multidimensional Array in java

```
1. arr[0][0]=1;
2. arr[0][1]=2;
3. arr[0][2]=3;
4. arr[1][0]=4;
5. arr[1][1]=5;
6. arr[1][2]=6;
7. arr[2][0]=7;
8. arr[2][1]=8;
9. arr[2][2]=9;
```

Example of Multidimensional java array

Let's see the simple example to declare, instantiate, initialize and print the 2Dimensional array.

```
1. class Testarray3{
2. public static void main(String args[]){
3.
4. //declaring and initializing 2D array
5. int arr[][]={{1,2,3},{2,4,5},{4,4,5}};
6.
7. //printing 2D array
8. for(int i=0;i<3;i++){
9.  for(int j=0;j<3;j++){
10.   System.out.print(arr[i][j]+" ");
11. }
12. System.out.println();
13. }
14.
15. }}
```

```
Output:1 2 3
       2 4 5
       4 4 5
```

What is the class name of java array?

In java, array is an object. For array object, an proxy class is created whose name can be obtained by getClass().getName() method on the object.

```
1. class Testarray4{
2. public static void main(String args[]){
3. int arr[]={4,4,5};
4. Class c=arr.getClass();
5. String name=c.getName();
6.
7. System.out.println(name);
8. }}
```

```
Output:I
```

Java OOPs Concepts

Copying a java array

We can copy an array to another by the arraycopy method of System class.

Syntax of arraycopy method

```
1. public static void arraycopy(
2. Object src, int srcPos,Object dest, int destPos, int length
3. )
```

Example of arraycopy method

```
1. class TestArrayCopyDemo {
2.     public static void main(String[] args) {
3.         char[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e',
4.                 'i', 'n', 'a', 't', 'e', 'd' };
5.         char[] copyTo = new char[7];
6.
7.         System.arraycopy(copyFrom, 2, copyTo, 0, 7);
8.         System.out.println(new String(copyTo));
9.     }
10. }
```

```
Output:caffein
```

Addition of 2 matrices in java

Let's see a simple example that adds two matrices.

```
1. class Testarray5{
2. public static void main(String args[]){
3. //creating two matrices
4. int a[][]={{1,3,4},{3,4,5}};
5. int b[][]={{1,3,4},{3,4,5}};
6.
7. //creating another matrix to store the sum of two matrices
8. int c[][]=new int[2][3];
9.
10. //adding and printing addition of 2 matrices
11. for(int i=0;i<2;i++){
12. for(int j=0;j<3;j++){
13. c[i][j]=a[i][j]+b[i][j];
14. System.out.print(c[i][j]+" ");
15. }
16. System.out.println();//new line
17. }
18.
19. }}
```

```
Output:2  6  8
       6  8  10
```

Wrapper class in Java

Java uses primitive types, such as int and char for performance reasons. These data types are not part of the object hierarchy. They are passed by value to methods and cannot be directly passed by reference. Also, there is no way for two methods to refer to the same instance of an int. At times, it is required to create an object representation for one of these primitive types as there are collection classes that deal only with objects (compareTo, equals); to store a primitive type in one of these classes, the primitive type is wrap in a class.

Wrapper class in java provides the mechanism *to convert primitive into object and object into primitive*.

Since J2SE 5.0, **autoboxing** and **unboxing** feature converts primitive into object and object into primitive automatically. The automatic conversion of primitive into object is known and autoboxing and vice-versa unboxing.

One of the eight classes of *java.lang* package are known as wrapper class in java. The list of eight wrapper classes are given below:

Primitive Type	Wrapper class
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

Wrapper class Example: Primitive to Wrapper

```
1. public class WrapperExample1{
2. public static void main(String args[]){
3. //Converting int into Integer
4. int a=20;
5. Integer i=Integer.valueOf(a);//converting int into Integer
6. Integer j=a;//autoboxing, now compiler will write Integer.valueOf(a) internally
7.
8. System.out.println(a+" "+i+" "+j);
9. }}
```

Output:

```
20 20 20
```

Wrapper class Example: Wrapper to Primitive

```
1. public class WrapperExample2{
2. public static void main(String args[]){
3. //Converting Integer to int
4. Integer a=new Integer(3);
5. int i=a.intValue();//converting Integer to int
6. int j=a;//unboxing, now compiler will write a.intValue() internally
7.
8. System.out.println(a+" "+i+" "+j);
9. }}
```

Output:

```
3 3 3
```

Call by Value and Call by Reference in Java

There is only call by value in java, not call by reference. If we call a method passing a value, it is known as call by value. The changes being done in the called method, is not affected in the calling method.

Example of call by value in java

In case of call by value original value is not changed. Let's take a simple example:

```
1. class Operation{
2.  int data=50;
3.
4.  void change(int data){
5.  data=data+100;//changes will be in the local variable only
6.  }
7.
8.  public static void main(String args[]){
9.    Operation op=new Operation();
10.
11.   System.out.println("before change "+op.data);
12.   op.change(500);
13.   System.out.println("after change "+op.data);
14. }
15.}
```

```
Output:before change 50
       after change 50
```

Another Example of call by value in java

In case of call by reference original value is changed if we made changes in the called method. If we pass object in place of any primitive value, original value will be changed. In this example we are passing object as a value.

```
1. class Operation2{
2.  int data=50;
3.  void change(Operation2 op){
4.  op.data=op.data+100;//changes will be in the instance variable
5.  }
6.  public static void main(String args[]){
7.    Operation2 op=new Operation2();
8.    System.out.println("before change "+op.data);
9.    op.change(op);//passing object
10.   System.out.println("after change "+op.data);
11. }
12.}
```

```
Output:before change 50
after change 150
```

Java Strictfp Keyword

Java strictfp keyword ensures that you will get the same result on every platform if you perform operations in the floating-point variable. The precision may differ from platform to platform that is why java programming language have provided the strictfp keyword, so that you get same result on every platform. So, now you have better control over the floating-point arithmetic.

Legal code for strictfp keyword

The strictfp keyword can be applied on methods, classes and interfaces.

```
1. strictfp class A{}//strictfp applied on class
```

```
1. strictfp interface M{}//strictfp applied on interface
```

```
1. class A{
2. strictfp void m(){}//strictfp applied on method
3. }
```

Illegal code for strictfp keyword

The strictfp keyword can be applied on abstract methods, variables or constructors.

```
1. class B{
2. strictfp abstract void m();//Illegal combination of modifiers
3. }
```

```
1. class B{
2. strictfp int data=10;//modifier strictfp not allowed here
3. }
```

```
1. class B{
2. strictfp B(){}//modifier strictfp not allowed here
3. }
```

Creating API Document | javadoc tool

We can create document api in java by the help of **javadoc** tool. In the java file, we must use the documentation comment /**... */ to post information for the class, method, constructor, fields etc.

Let's see the simple class that contains documentation comment.

```
1. package com.abc;
2. /** This class is a user-defined class that contains one methods cube.*/
3. public class M{
4.
5. /** The cube method prints cube of the given number */
6. public static void  cube(int n){System.out.println(n*n*n);}
7. }
```

To create the document API, you need to use the javadoc tool followed by java file name. There is no need to compile the javafile.

On the command prompt, you need to write:

```
javadoc M.java
```

to generate the document api. Now, there will be created a lot of html files. Open the index.html file to get the information about the classes.

Java Command Line Arguments

1. Command Line Argument
2. Simple example of command-line argument
3. Example of command-line argument that prints all the values

The java command-line argument is an argument i.e. passed at the time of running the java program.

The arguments passed from the console can be received in the java program and it can be used as an input.

So, it provides a convenient way to check the behavior of the program for the different values. You can pass **N** (1,2,3 and so on) numbers of arguments from the command prompt.

Simple example of command-line argument in java

In this example, we are receiving only one argument and printing it. To run this java program, you must pass at least one argument from the command prompt.

```
1. class CommandLineExample{
2. public static void main(String args[]){
3. System.out.println("Your first argument is: "+args[0]);
4. }
5. }
```

```
1. compile by > javac CommandLineExample.java
2. run by > java CommandLineExample sonoo
```

```
Output: Your first argument is: sonoo
```

Java OOPs Concepts

Example of command-line argument that prints all the values

In this example, we are printing all the arguments passed from the command-line. For this purpose, we have traversed the array using for loop.

```
1. class A{
2. public static void main(String args[]){
3. 
4. for(int i=0;i<args.length;i++)
5. System.out.println(args[i]);
6. 
7. }
8. }
```

1. compile by > javac A.java
2. run by > java A sonoo jaiswal 1 3 abc

```
Output: sonoo
        jaiswal
        1
        3
        abc
```

Varargs (Variable-arity) in Java: Variable argument method

Basic Syntax and Uses

Variable arguments language feature makes it possible to call a method with a variable number of arguments. Before making that call, the rightmost parameter in the method's parameter list must conform to the following syntax:

type ... *variableName*

A variable-length argument is specified by three periods (…). The ellipsis (...) identifies a variable number of arguments, and is demonstrated in the following summation method.

```
static int sum (int ... numbers)
{
  int total = 0;
  for (int i = 0; i < numbers.length; i++)   // for ( i : numbers)
      total += numbers [i];
  return total;
}
```

Note: There must be only one varargs parameter.

Summing a list of integers is one basic use for variable arguments. Computing the average of a list of floating-point numbers, concatenating a list of strings into a single string, and finding the maximum/minimum values in lists of floating-point numbers are some other basic uses. The following BasicUses.java source code demonstrates these basic uses, to give you more exposure to variable arguments.

```
// BasicUses.java
class BasicUses
{
  public static void main (String [] args)
  {
    System.out.println ("Average: " +
                        avg (20.3, 3.1415, 32.3));
    System.out.println ("Concatenation: " +
                        concat ("Hello", " ", "World"));
    System.out.println ("Maximum: " +
                        max (30, 22.3, -9.3, 173.2));
    System.out.println ("Minimum: " +
                        min (30, 22.3, -9.3, 173.2));
    System.out.println ("Sum: " +
                        sum (20, 30));
  }
  static double avg (double ... numbers)
  {
    double total = 0;
    for (int i = 0; i < numbers.length; i++)
        total += numbers [i];
    return total / numbers.length;
  }
  static String concat (String ... strings)
  {
```

```
    StringBuilder sb = new StringBuilder ();
    for (int i = 0; i < strings.length; i++)
        sb.append (strings [i]);
    return sb.toString ();
  }
  static double max (double ... numbers)
  {
    double maximum = Double.MIN_VALUE;
    for (int i = 0; i < numbers.length; i++)
        if (numbers [i] > maximum)
            maximum = numbers [i];
    return maximum;
  }
  static double min (double ... numbers)
  {
    double minimum = Double.MAX_VALUE;
    for (int i = 0; i < numbers.length; i++)
        if (numbers [i] < minimum)
            minimum = numbers [i];
    return minimum;
  }
  static int sum (int ... numbers)
  {
    int total = 0;
    for (int i = 0; i < numbers.length; i++)
        total += numbers [i];
    return total;
  }
}
```

Look closely at BasicUses.java and you'll discover a second innovation: java.lang.StringBuilder. This class is used, instead of java.lang.StringBuffer, in the concatenation method because StringBuilder's lack of synchronization offers better performance than StringBuffer. (Why call a synchronized method and obtain the resulting performance hit, no matter how small, when it's not necessary?)

Difference between object and class

There are many differences between object and class. A list of differences between object and class are given below:

No.	Object	Class
1)	Object is an **instance** of a class.	Class is a **blueprint or template** from which objects are created.
2)	Object is a **real world entity** such as pen, laptop, mobile, bed, keyboard, mouse, chair etc.	Class is a **group of similar objects**.
3)	Object is a **physical** entity.	Class is a **logical** entity.
4)	Object is created through **new keyword** mainly e.g. Student s1=new Student();	Class is declared using **class keyword** e.g. class Student{}
5)	Object is created **many times** as per requirement.	Class is declared **once**.
6)	Object **allocates memory when it is created**.	Class **doesn't allocated memory when it is created**.
7)	There are **many ways to create object** in java such as new keyword, newInstance() method, clone() method, factory method and deserialization.	There is only **one way to define class** in java using class keyword.

Difference between method overloading and method overriding in java

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

No.	Method Overloading	Method Overriding
1)	Method overloading is used *to increase the readability* of the program.	Method overriding is used *to provide the specific implementation* of the method that is already provided by its super class.
2)	Method overloading is performed *within class*.	Method overriding occurs *in two classes* that have IS-A (inheritance) relationship.
3)	In case of method overloading, *parameter must be different*.	In case of method overriding, *parameter must be same*.
4)	Method overloading is the example of *compile time polymorphism*.	Method overriding is the example of *run time polymorphism*.
5)	In java, method overloading can't be performed by changing return type of the method only. *Return type can be same or different* in method overloading. But you must have to change the parameter.	*Return type must be same or covariant* in method overriding.

Java Method Overloading example

```
1. class OverloadingExample{
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. }
```

Java Method Overriding example

```
1. class Animal{
2. void eat(){System.out.println("eating...");}
3. }
4. class Dog extends Animal{
5. void eat(){System.out.println("eating bread...");}
6. }
```