## Q= Difference between PATH and CLASSPATH in Java

# Here are some of the common difference between PATH vs CLASSPATH in Java : 1)The main difference between PATH and CLASSPATH is that  PATH is an environment variable which is used to locate JDK binaries like "java" or "javac" command used to run java program and compile java source file. On the other hand, CLASSPATH, an environment variable is used by System or [Application ClassLoader](http://javarevisited.blogspot.com/2012/12/how-classloader-works-in-java.html) to locate and load compile Java bytecodes stored in the .class file. 2) In order to set PATH in Java, you need to include JDK\_HOME/bin directory in PATH environment variable while in order to set CLASSPATH in Java you need to include all those directories where you have put either your .class file or JAR file which is required by your Java application. 3) Another significant difference between PATH and CLASSPATH is that PATH can not be overridden by any Java settings but CLASSPATH can be overridden by providing command line option -classpath or -cp to both "java" and "javac" commands or by using Class-Path attribute in Manifest file inside [JAR archive](http://java67.blogspot.com/2016/01/how-to-run-jar-file-from-command-prompt.html). 4) PATH environment variable is used by operating system to find any binary or command typed in the shell, this is true for both Windows and Linux environment while CLASSPATH is only used by Java ClassLoaders to load class files.

# Q=[Difference between system.gc() and runtime.gc()](https://stackoverflow.com/questions/6197306/difference-between-system-gc-and-runtime-gc)?

Ans=Both are same. System.gc() is effectively equivalent to Runtime.gc(). System.gc()internally calls Runtime.gc().

The only difference is System.gc() is a class method where as Runtime.gc() is an instance method. So, System.gc() is more convenient.

Runtime.gc() is a native method where as System.gc() is non - native method which in turn calls the Runtime.gc()

**System.gc():**

1: It is a class method(static method).

2: Non-Native method.(Code which doesn't directly interacts with Hardware and System Resources).

3: System.gc(), Internally calls Runtime.getRuntime().gc().

**Runtime.gc():**

1: Instance method.

2: Native method(A programming language which directly interacts with Hardware and System Resources.).

Q=JVM architecture and its working?

Ans=**Class Loader Sub system:-**

**Loading;-**when we write java myapp

It loades the our .class file. Using class loaders. There are 3 types of class loader.

Bootstrap class loader: load rt.jar

Extension class loader: load ext folder jar files (jre/lib/exe)

Application class loader: this load the our class and if we write -cp it will load other jars or classes .

**Linking:** It hase three different pages:-

Verify: It verifies byte code (.class ) file is valid or not other wise it will give error.

Prepare: memory allocation happens only for class level variable. Ex:

Public static Boolean flag = true;

It will load the memory for Boolean and assign its default value false.

Resolve:All sysmbolic references are resolved ex,

This class has reference of other class it will resolve here.IT will try to find out other reference class.

Example : x refer to y

If y class not found it throws an exception ClassDefNotfoundException.

**Initialization:** This is the false where idealize static blokes and static variables.

**Run Time Data Areas: It has five types:**

**Method Area:**  It stores class level data like static variables , static blocks , constants , bytecode etc.

Method area is calles param gen space.By default 64 mb memory available for Method area. If param gen will full it throws OutOfmemoryError : PamaGen Space. We can increase using -XX:MaxParamSize.

In java 8 they give this name **MetaSpace**.

**Heap:** Heap is a place where object data is store. All the array , collection everything saved in array.

**Pc Register**: program counter resister.It contains next instruction to be executed per thread.

**JAVA Stacks**: it contains stack frame corresponding to current method execution per thread.Suppose a thread will be execute 3 methods . It will execute first method and then push the method 2 etc.

If any method is recursive and call it self or if we forgot to exit criteria of method calling. Then we got the exception java.lang.StackoverflowError.

For increase memory -xss

**Native Method:** Some times when we use native methods or some dll etc then it will use.

**Execution Engine:**

**Interpretor:** Interpret the bytecode line by line.

**Jit compiler:**

Q=System.out.println(Math.ceil(23.46)); // Prints 24

System.out.println(Math.floor(23.46)); // Prints 23

double val1 = 4.2;

double val2 = 4.5;

System.out.println("Math.rint(" + val1 + ") = " + Math.rint(val1));

System.out.println("Math.round(" + val1 + ") = " + Math.round(val1));

System.out.println("Math.rint(" + val2 + ") = " + Math.rint(val2));

System.out.println("Math.round(" + val2 + ") = " + Math.round(val2));

System.out.println("Math.rint(" + (val2 + 0.001d) + ") = " + Math.rint(val2 + 0.001d));

System.out.println("Math.round(" + (val2 + 0.001d) + ") = " + Math.round(val2 + 0.001d));

**Output:**

Math.rint(4.2) = 4.0

Math.round(4.2) = 4

Math.rint(4.5) = 4.0

Math.round(4.5) = 5

Math.rint(4.501) = 5.0

Math.round(4.501) = 5

Q=difference between put and post method?

Ans=

## even though both PUT and POST can be used to create and update an entity, POST is usually preferred for creating and PUT is preferred for updating an existing entity. For example, to create a new Order you should use: POST /orders and to update an existing order, you should use PUT /orders/13892 which means modify the order with OrderId 13892 If you execute POST request multiple times, it will end up create that many orders, but when you execute PUT it will always produce the same result because of its [idempotent](http://javarevisited.blogspot.com/2016/05/what-are-idempotent-and-safe-methods-of-HTTP-and-REST.html). You should also remember that both PUT and POST are **unsafe methods**. Safe methods in HTTP do not modify the resource in the server e..g GET or HEAD, while Idempotent HTTP methods return same result irrespective of how many times you call them. Idempotency is an important thing while building a fault-tolerant RESTful API. Idempotency is also the reason of why you should use PUT over POST to update a resource in REST. For example, suppose a client wants to update a resource through POST. Since POST is not an idempotent method, calling it multiple times may result in incorrect updates. When to use PUT and POST methods in REST?

Now' it's time for some practical knowledge about when to use the PUT and POST methods to call RESTful WebServices.  
  
1) You should use POST to create new resources and PUT to update existing resources.  
  
2) Use PUT when you know the "id" of the object e.g. Order, Book, Employee  
  
3) Use POST when you need the server to be in control of URL generation of your resources.  
  
4) Examples  
PUT /items/1 update  
POST /items create

<https://stackoverflow.com/questions/630453/put-vs-post-in-rest>

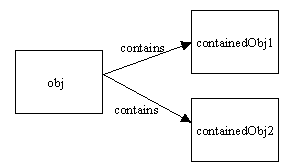
<http://javarevisited.blogspot.in/2016/10/difference-between-put-and-post-in-restful-web-service.html>

**Q1) What are different type of cloning in Java?**

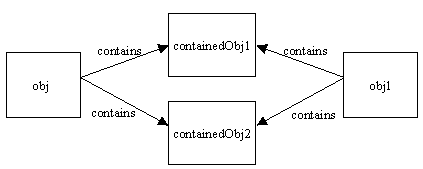
Ans) Java supports two type of cloning: - **Deep and shallow cloning.** By default shallow clone is used in Java. Object class has a method clone() which does shallow cloning.

**Q2) What is Shallow copy?**

Ans) Shallow clone is a copying the reference pointer to the object, which mean the new object is pointing to the same memory reference of the old object. The memory usage is lower.

  
Figure 1: Original java object obj

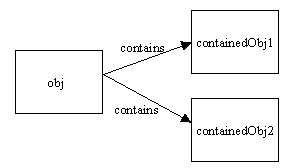
The shallow copy is done for obj and new object obj1 is created but contained objects of obj are not copied.

  
Figure 2: Shallow copy object obj1

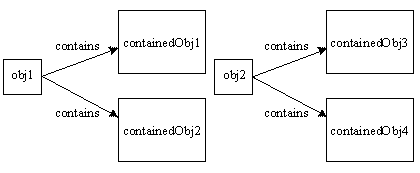
It can be seen that no new objects are created for obj1 and it is referring to the same old contained objects. If either of the containedObj contain any other object no new reference is created.

**Q3) What is deep copy and how it can be acheived?**

Ans) In deep copy is the copy of object itself. A new memory is allocated for the object and contents are copied.

  
Figure 3 : Original Object obj

When a deep copy of the object is done new references are created.

  
Figure 4: obj2 is deep copy of obj1

One solution is to simply implement your own custom method (e.g., deepCopy()) that returns a deep copy of an instance of one of your classes. This may be the best solution if you need a complex mixture of deep and shallow copies for different fields, but has a few significant drawbacks:

* You must be able to modify the class (i.e., have the source code) or implement a subclass. If you have a third-party class for which you do not have the source and which is marked final, you are out of luck.
* You must be able to access all of the fields of the classe's superclasses. If significant parts of the object's state are contained in private fields of a superclass, you will not be able to access them.
* You must have a way to make copies of instances of all of the other kinds of objects that the object references. This is particularly problematic if the exact classes of referenced objects cannot be known until runtime.
* Custom deep copy methods are tedious to implement, easy to get wrong, and difficult to maintain. The method must be revisited any time a change is made to the class or to any of its superclasses.

Other common solution to the deep copy problem is to use **Java Object Serialization**(JOS). The idea is simple: Write the object to an array using **ObjectOutputStream** and then use **ObjectInputStream** to reconsistute a copy of the object. The result will be a completely distinct object, with completely distinct referenced objects. JOS takes care of all of the details: superclass fields, following object graphs, and handling repeated references to the same object within the graph.

* It will only work when the object being copied, as well as all of the other objects references directly or indirectly by the object, are serializable. (In other words, they must implement java.io.Serializable.) Fortunately it is often sufficient to simply declare that a given class implements java.io.Serializable and let Java's default serialization mechanisms do their thing. Java Object Serialization is slow, and using it to make a deep copy requires both serializing and deserializing.

There are ways to speed it up (e.g., by pre-computing serial version ids and defining custom readObject() and writeObject() methods), but this will usually be the primary bottleneck. The byte array stream implementations included in the java.io package are designed to be general enough to perform reasonable well for data of different sizes and to be safe to use in a multi-threaded environment. These characteristics, however, slow down ByteArrayOutputStream and (to a lesser extent) ByteArrayInputStream .

**Q4) What is difference between deep and shallow cloning?**

Ans) The differences are as follows:

Consider a class:

public class MyData{

String id;

Map myData;

}

The shallow copying of this object will be pointing to the same memory reference as the original object. So a change in myData by either original or cloned object will be reflected in other also. But in deep copying there will memory allocated and values assigned to the property will be same. Any change in object will not be reflected in other.

Shallow copying is default cloning in Java which can be achieved using **Object.clone()**method of Object class. For deep copying override the clone method to create new object and copy its values.

**Q5) What are disadvantages of deep cloning ?**

Ans) Disadvantages of using Serialization to achieve deep cloning –

* Serialization is more expensive than using object.clone().
* Not all objects are serializable.
* Serialization is not simple to implement for deep cloned object..

**Q5) What is a transient variable?**

Ans) If some of the properties of a class are not required to be serialized then the varaibles are marked as transient. When an object is deserialized the transient variables retains the default value depending on the type of variable declared and hence lost its original value.

# strictfp keyword in java

**strictfp** is a keyword in java used for restricting floating-point calculations and ensuring same result on every platform while performing operations in the floating-point variable.  
Floating point calculations are platform dependent i.e. different output(floating-point values) is achieved when a class file is run on different platforms(16/32/64 bit processors). To solve this types of issue, strictfp keyword was introduced in JDK 1.2 version by following [IEEE 754](https://en.wikipedia.org/wiki/IEEE_floating_point) standards for floating-point calculations.

**Important points:**

* strictfp modifier is used with classes, interfaces and methods only.
* strictfp class Test
* {
* // all concrete methods here are
* // implicitly strictfp.
* }
* strictfp interface Test
* {
* // all methods here becomes implicitly
* // strictfp when used during inheritance.
* }
* class Car
* {
* // strictfp applied on a concrete method
* strictfp void calculateSpeed(){}
* }

* When a class or an interface is declared with strictfp modifier, then all methods declared in the class/interface, and all nested types declared in the class, are implicitly strictfp.
* strictfp **cannot** be used with abstract methods. However, it can be used with abstract classes/interfaces.
* Since methods of an interface are implicitly abstract, strictfp **cannot** be used with any method inside an interface.
* strictfp interface Test
* {
* double sum();
* strictfp double mul(); // compile-time error here
* }

|  |
| --- |
| //Java program to illustrate strictfp modifier    public class Test  {      // calculating sum using strictfp modifier      public strictfp double sum()      {          double num1 = 10e+10;            double num2 = 6e+08;            return (num1+num2);        }        public static strictfp void main(String[] args)      {          Test t = new Test();            System.out.println(t.sum());      }  } |

Q=**Is array in java is class?**

Ans= yes

|  |
| --- |
| public class Test  {      public static void main(String[] args)      {          int[] x = new int[3];          System.out.println(x.getClass().getName());      }  } |

Run on IDE

Output:

[I

**Q9) Why static methods cannot access non static variables or methods?**

Ans) A static method cannot access non static variables or methods because static methods can be accessed without instantiating the class, so if the class is not instantiated the variables are not intialized and thus cannot be accessed from a static method.

**Q10) What is static class ?**

Ans) A class cannot be declared static except inner class. But a class can be said a static class if all the variables and methods of the class are static and the constructor is private. Making the constructor private will prevent the class to be instantiated. So the only possibility to access is using Class name only

**Q4) Which classes in java are immutable?**

**Ans)** All wrapper classes in java.lang are immutable –   
String, Integer, Boolean, Character, Byte, Short, Long, Float, Double, BigDecimal, BigInteger

**Q5) What are the advantages of immutability?**

Ans)

* Immutable objects are automatically thread-safe, the overhead caused due to use of synchronisation is avoided.
* Once created the state of the immutable object can not be changed so there is no possibility of them getting into an inconsistent state.
* The references to the immutable objects can be easily shared or cached without having to copy or clone them as there state can not be changed ever after construction.
* The best use of the immutable objects is as the keys of a map.

**Q4) Other than Serialization what are the different approach to make object Serializable?**

Ans) Besides the Serializable interface, at least three alternate approaches can serialize Java objects:

* For object serialization, instead of implementing the Serializable interface, a developer can implement the Externalizable interface, which extends Serializable. By implementing Externalizable, a developer is responsible for implementing the writeExternal() and readExternal() methods. As a result, a developer has sole control over reading and writing the serialized objects.
* XML serialization is an often-used approach for data interchange. This approach lags runtime performance when compared with Java serialization, both in terms of the size of the object and the processing time. With a speedier XML parser, the performance gap with respect to the processing time narrows. Nonetheless, XML serialization provides a more malleable solution when faced with changes in the serializable object.
* Finally, consider a "roll-your-own" serialization approach. You can write an object's content directly via either the ObjectOutputStream or the DataOutputStream. While this approach is more involved in its initial implementation, it offers the greatest flexibility and extensibility. In addition, this approach provides a performance advantage over Java serialization.

**Q7) What happens if an object is serializable but it includes a reference to a non-serializable object?**

Ans- If you try to serialize an object of a class which implements serializable, but the object includes a reference to an non-serializable class then a ‘NotSerializableException’ will be thrown at runtime.

### Generics in Java

Generics was added in Java 5 to provide **compile-time type checking** and removing risk of ClassCastException that was common while working with collection classes.

List list = new ArrayList();

list.add("abc");

list.add(new Integer(5)); //OK

for(Object obj : list){

//type casting leading to ClassCastException at runtime

String str=(String) obj;

}

Above code compiles fine but throws ClassCastException at runtime because we are trying to cast Object in the list to String whereas one of the element is of type Integer.

### Java Generic Interface

Comparable interface is a great example of Generics in interfaces and it’s written as:

package java.lang;

import java.util.\*;

public interface Comparable<T> {

public int compareTo(T o);

}

#### Generic Type Class or Interface

A class is generic if it declares one or more type variables. These type variables are known as the type parameters of the class. Let’s understand with an example.

DemoClass is simple java class, which have one property t (can be more than one also); and type of property is Object.

|  |
| --- |
| class DemoClass {     private Object t;       public void set(Object t) { this.t = t; }       public Object get() { return t; }  } |

Here we want that once initialized the class with a certain type, class should be used with that particular type only. e.g. If we want one instance of class to hold value t of type ‘String‘, then programmer should set and get the only String type. Since we have declared property type to Object, there is no way to enforce this restriction. A programmer can set any object; and can expect any return value type from get method since all java types are subtypes of Object class.

To enforce this type restriction, we can use generics as below:

|  |
| --- |
| class DemoClass<T> {     //T stands for "Type"     private T t;       public void set(T t) { this.t = t; }       public T get() { return t; }  } |

Now we can be assured that class will not be misused with wrong types. A sample usage of DemoClass will look like this:

|  |
| --- |
| DemoClass<String> instance = new DemoClass<String>();  instance.set("lokesh");   //Correct usage  instance.set(1);        //This will raise compile time error |

Above analogy is true for interface as well. Let’s quickly look at an example to understand, how generics type information can be used in interfaces in java.

|  |
| --- |
| //Generic interface definition  interface DemoInterface<T1, T2>  {     T2 doSomeOperation(T1 t);     T1 doReverseOperation(T2 t);  }    //A class implementing generic interface  class DemoClass implements DemoInterface<String, Integer>  {     public Integer doSomeOperation(String t)     {        //some code     }     public String doReverseOperation(Integer t)     {        //some code     }  } |

#### Generic Type Method or Constructor

Generic methods are much similar to generic classes. They are different only in one aspect that scope of type information is inside method (or constructor) only. Generic methods are methods that introduce their own type parameters.

Let’s understand this with an example. Below is code sample of a generic method which can be used to find all occurrences of a type parameters in a list of variables of that type only.

|  |
| --- |
| public static <T> int countAllOccurrences(T[] list, T item) {     int count = 0;     if (item == null) {        for ( T listItem : list )           if (listItem == null)              count++;     }     else {        for ( T listItem : list )           if (item.equals(listItem))              count++;     }     return count;  } |

If you pass a list of String and another string to search in this method, it will work fine. But if you will try to find an Number into list of String, it will give compile time error.

Same as above can be example of generic constructor. Let’s take a separate example for generic constructor as well.

|  |
| --- |
| class Dimension<T>  {     private T length;     private T width;     private T height;       //Generic constructor     public Dimension(T length, T width, T height)     {        super();        this.length = length;        this.width = width;        this.height = height;     }  } |

In this example, Dimension class’s constructor has the type information also. So you can have an instance of dimension with all attributes of a single type only.

## 4) Generic Type Arrays

Array in any language have same meaning i.e. an array is a collection of similar type of elements. In java, pushing any incompatible type in an array on runtime will throw ArrayStoreException. It means array preserve their type information in runtime, and generics use type erasure or remove any type information in runtime. Due to above conflict, instantiating a generic array in java is not permitted.

|  |
| --- |
| public class GenericArray<T> {      // this one is fine      public T[] notYetInstantiatedArray;        // causes compiler error; Cannot create a generic array of T      public T[] array = new T[5];  } |

In the same line as above generic type classes and methods, we can have generic arrays in java. As we know that an array is a collection of similar type of elements and pushing any incompatible type will throw ArrayStoreException in runtime; which is not the case with Collection classes.

|  |
| --- |
| Object[] array = new String[10];  array[0] = "lokesh";  array[1] = 10;      //This will throw ArrayStoreException |

Above mistake is not very hard to make. It can happen anytime. So it’s better to provide the type information to array also so that error is caught at compile time itself.

**Another reason why arrays does not support generics is that arrays are co-variant**, which means that an array of supertype references is a supertype of an array of subtype references. That is, Object[] is a supertype of String[]and a string array can be accessed through a reference variable of type Object[].

|  |
| --- |
| Object[] objArr = new String[10];  // fine  objArr[0] = new String(); |

### Java Generics Wildcards

In generic code, the question mark (?), called the wildcard, represents an unknown type. **A wildcard parameterized type is an instantiation of a generic type where at least one type argument is a wildcard.** Examples of wildcard parameterized types are Collection<?<, List<? extends Number<, Comparator<? super String> and Pair<String,?>.

### Java Generics Upper Bounded Wildcard

Upper bounded wildcards are used to relax the restriction on the type of variable in a method. Suppose we want to write a method that will return the sum of numbers in the list, so our implementation will be something like this.

public static double sum(List<Number> list){

double sum = 0;

for(Number n : list){

sum += n.doubleValue();

}

return sum;

}

Now the problem with above implementation is that it won’t work with List of Integers or Doubles because we know that List<Integer> and List<Double> are not related, this is when upper bounded wildcard is helpful. We use generics wildcard with **extends** keyword and the **upper bound** class or interface that will allow us to pass argument of upper bound or it’s subclasses types.

The above implementation can be modified like below program.

package com.journaldev.generics;

import java.util.ArrayList;

import java.util.List;

public class GenericsWildcards {

public static void main(String[] args) {

List<Integer> ints = new ArrayList<>();

ints.add(3); ints.add(5); ints.add(10);

double sum = sum(ints);

System.out.println("Sum of ints="+sum);

}

public static double sum(List<? extends Number> list){

double sum = 0;

for(Number n : list){

sum += n.doubleValue();

}

return sum;

}

}

It’s similar like writing our code in terms of interface, in above method we can use all the methods of upper bound class Number. Note that with upper bounded list, we are not allowed to add any object to the list except null. If we will try to add an element to the list inside the sum method, the program won’t compile.

Sometimes we have a situation where we want our generic method to be working with all types, in this case unbounded wildcard can be used. Its same as using <? extends Object>.

public static void printData(List<?> list){

for(Object obj : list){

System.out.print(obj + "::");

}

}

We can provide List<String> or List<Integer> or any other type of Object list argument to the printDatamethod. Similar to upper bound list, we are not allowed to add anything to the list.

### Java Generics Unbounded Wildcard

Sometimes we have a situation where we want our generic method to be working with all types, in this case unbounded wildcard can be used. Its same as using <? extends Object>.

public static void printData(List<?> list){

for(Object obj : list){

System.out.print(obj + "::");

}

}

### Java Generics Lower bounded Wildcard

Suppose we want to add Integers to a list of integers in a method, we can keep the argument type as List<Integer> but it will be tied up with Integers whereas List<Number> and List<Object> can also hold integers, so we can use lower bound wildcard to achieve this. We use generics wildcard (?) with **super**keyword and lower bound class to achieve this.

We can pass lower bound or any super type of lower bound as an argument in this case, java compiler allows to add lower bound object types to the list.

public static void addIntegers(List<? super Integer> list){

list.add(new Integer(50));

}

### Subtyping using Generics Wildcard

List<? extends Integer> intList = new ArrayList<>();

List<? extends Number> numList = intList; // OK. List<? extends Integer> is a subtype of List<? extends Number>

### Java Generics Type Erasure

Generics in Java was added to provide type-checking at compile time and it has no use at run time, so java compiler uses **type erasure** feature to remove all the generics type checking code in byte code and insert type-casting if necessary. Type erasure ensures that no new classes are created for parameterized types; consequently, generics incur no runtime overhead.

For example if we have a generic class like below;

public class Test<T extends Comparable<T>> {

private T data;

private Test<T> next;

public Test(T d, Test<T> n) {

this.data = d;

this.next = n;

}

public T getData() { return this.data; }

}

The Java compiler replaces the bounded type parameter T with the first bound interface, Comparable, as below code:

public class Test {

private Comparable data;

private Test next;

public Node(Comparable d, Test n) {

this.data = d;

this.next = n;

}

public Comparable getData() { return data; }

}

### Generics in Java – Further Readings

* Generics doesn’t support sub-typing, so List<Number> numbers = new ArrayList<Integer>(); will not compile, We can’t create generic array, so List<Integer>[] array = new ArrayList<Integer>[10] will not compile,

Q=difference between <?> and <Object> in generics?

Ans=if we write a method

Public void add(List<Object> list){

---

----

}

Then we can pass only list of Object.

But

Public void add(List<?> list){

---

---

}

Then we can pass any type of list in this method.

Q=Stack in java?

Ans=Stack is a subclass of Vector that implements a standard last-in, first-out stack.

Stack only defines the default constructor, which creates an empty stack. Stack includes all the methods defined by Vector, and adds several of its own.

Stack( )

|  |  |
| --- | --- |
| **Sr.No.** | **Method & Description** |
| 1 | **boolean empty()**  Tests if this stack is empty. Returns true if the stack is empty, and returns false if the stack contains elements. |
| 2 | **Object peek( )**  Returns the element on the top of the stack, but does not remove it. |
| 3 | **Object pop( )**  Returns the element on the top of the stack, removing it in the process. |
| 4 | **Object push(Object element)**  Pushes the element onto the stack. Element is also returned. |
| 5 | **int search(Object element)**  Searches for element in the stack. If found, its offset from the top of the stack is returned. Otherwise, .1 is returned. |

# Java Queue Interface

Java Queue interface orders the element in FIFO(First In First Out) manner.

Since Queue is an interface you need to instantiate a concrete implementation of the interface in order to use it. You can choose between the following Queue implementations in the Java Collections API:

* java.util.LinkedList
* java.util.PriorityQueue

LinkedList is a pretty standard queue implementation.

PriorityQueue stores its elements internally according to their natural order (if they implement Comparable), or according to a Comparator passed to the PriorityQueue.

Queue queueA = new LinkedList();

Queue queueB = new PriorityQueue();

### **Methods of Java Queue Interface**

|  |  |
| --- | --- |
| **Method** | **Description** |
| boolean add(object) | It is used to insert the specified element into this queue and return true  upon success. |
| boolean offer(object) | It is used to insert the specified element into this queue. |
| Object remove() | It is used to retrieves and removes the head of this queue. |
| Object poll() | It is used to retrieves and removes the head of this queue, or returns null  if this queue is empty. |
| Object element() | It is used to retrieves, but does not remove, the head of this queue. |
| Object peek() | It is used to retrieves, but does not remove, the head of this queue,  or returns null if this queue is empty. |

Queue queueA = new LinkedList();

queueA.add("element 0");

queueA.add("element 1");

queueA.add("element 2");

//access via Iterator

Iterator iterator = queueA.iterator();

while(iterator.hasNext(){

String element = (String) iterator.next();

}

//access via new for-loop

for(Object object : queueA) {

String element = (String) object;

}

Q=**PriorityQueue** ?

The **java.util.PriorityQueue** class is an unbounded priority queue based on a priority heap.Following are the important points about PriorityQueue:

* The elements of the priority queue are ordered according to their natural ordering, or by a Comparator provided at queue construction time, depending on which constructor is used.
* A priority queue does not permit null elements.
* A priority queue relying on natural ordering also does not permit insertion of non-comparable objects.

## **Class methods**

|  |  |
| --- | --- |
| **S.N.** | **Method & Description** |
| 1 | [**boolean add(E e)**](https://www.tutorialspoint.com/java/util/priorityqueue_add.htm)  This method inserts the specified element into this priority queue. |
| 2 | [**void clear()**](https://www.tutorialspoint.com/java/util/priorityqueue_clear.htm)  This method removes all of the elements from this priority queue. |
| 3 | [**Comparator<? super E> comparator()**](https://www.tutorialspoint.com/java/util/priorityqueue_super.htm)  This method returns the comparator used to order the elements in this queue, or null if this queue is sorted according to the natural ordering of its elements. |
| 4 | [**boolean contains(Object o)**](https://www.tutorialspoint.com/java/util/priorityqueue_contains.htm)  This method returns true if this queue contains the specified element. |
| 5 | [**Iterator<E> iterator()**](https://www.tutorialspoint.com/java/util/priorityqueue_iterator.htm)  This method returns an iterator over the elements in this queue. |
| 6 | [**boolean offer(E e)**](https://www.tutorialspoint.com/java/util/priorityqueue_offer.htm)  This method inserts the specified element into this priority queue. |
| 7 | [**E peek()**](https://www.tutorialspoint.com/java/util/priorityqueue_peek.htm)  This method retrieves, but does not remove, the head of this queue, or returns null if this queue is empty. |
| 8 | [**E poll()**](https://www.tutorialspoint.com/java/util/priorityqueue_poll.htm)  This method retrieves and removes the head of this queue, or returns null if this queue is empty. |
| 9 | [**boolean remove(Object o)**](https://www.tutorialspoint.com/java/util/priorityqueue_remove.htm)  This method removes a single instance of the specified element from this queue, if it is present. |
| 10 | [**int size()**](https://www.tutorialspoint.com/java/util/priorityqueue_size.htm)  This method returns the number of elements in this collection. |
| 11 | [**Object[] toArray()**](https://www.tutorialspoint.com/java/util/priorityqueue_toarray_object.htm)  This method returns an array containing all of the elements in this queue. |
| 12 | [**<T> T[] toArray(T[] a)**](https://www.tutorialspoint.com/java/util/priorityqueue_toarray.htm)  This method returns an array containing all of the elements in this queue; the runtime type of the returned array is that of the specified array. |

Q=**Diffrence between inner class and nested class in java?**

Ans = A non-static class within another class is called Inner class.  
- A static class within another class is called Nested class or static Inner class.

For example:

class Outer{  
 class Inner{  
 }   
 static class Nested {   
 }  
}

- Inner class object must be created using Outer class object.  
- Nested class object is created without using Outer class object.

For example:

public class Program{  
 public static void main(String[] args){

Outer o = new Outer() ;

Outer.Inner i = o.new Inner() ;

Outer.Nested n = new Outer.Nested() ;

}  
}

- Inner class object holds the reference of Outer class object that has created it.  
- Nested class object do not hold reference of Outer class object, unless explicitly passed and stored.  
  
- Hence Inner class object can access instance data members and methods of its own as well as of the Outer class object's whose reference it holds.

# Q=Java - Inner classes

## **Nested Classes**

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

**Syntax**

Following is the syntax to write a nested class. Here, the class **Outer\_Demo**is the outer class and the class **Inner\_Demo** is the nested class.

class Outer\_Demo {

class Nested\_Demo {

}

}

Nested classes are divided into two types −

* **Non-static nested classes** − These are the non-static members of a class.
* **Static nested classes** − These are the static members of a class.



## **Inner Classes (Non-static Nested Classes)**

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

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

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

### **Inner Class**

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

Following is the program to create an inner class and access it. In the given example, we make the inner class private and access the class through a method.

**Example**

class Outer\_Demo {

int num;

// inner class

private class Inner\_Demo {

public void print() {

System.out.println("This is an inner class");

}

}

// Accessing he inner class from the method within

void display\_Inner() {

Inner\_Demo inner = new Inner\_Demo();

inner.print();

}

}

public class My\_class {

public static void main(String args[]) {

// Instantiating the outer class

Outer\_Demo outer = new Outer\_Demo();

// Accessing the display\_Inner() method.

outer.display\_Inner();

}

}

Here you can observe that **Outer\_Demo** is the outer class, **Inner\_Demo** is the inner class, **display\_Inner()** is the method inside which we are instantiating the inner class, and this method is invoked from the **main**method.

If you compile and execute the above program, you will get the following result −

**Output**

This is an inner class.

### **Accessing the Private Members**

As mentioned earlier, inner classes are also used to access the private members of a class. Suppose, a class is having private members to access them. Write an inner class in it, return the private members from a method within the inner class, say, **getValue()**, and finally from another class (from which you want to access the private members) call the getValue() method of the inner class.

To instantiate the inner class, initially you have to instantiate the outer class. Thereafter, using the object of the outer class, following is the way in which you can instantiate the inner class.

Outer\_Demo outer = new Outer\_Demo();

Outer\_Demo.Inner\_Demo inner = outer.new Inner\_Demo();

The following program shows how to access the private members of a class using inner class.

**Example**

class Outer\_Demo {

// private variable of the outer class

private int num = 175;

// inner class

public class Inner\_Demo {

public int getNum() {

System.out.println("This is the getnum method of the inner class");

return num;

}

}

}

public class My\_class2 {

public static void main(String args[]) {

// Instantiating the outer class

Outer\_Demo outer = new Outer\_Demo();

// Instantiating the inner class

Outer\_Demo.Inner\_Demo inner = outer.new Inner\_Demo();

System.out.println(inner.getNum());

}

}

If you compile and execute the above program, you will get the following result −

**Output**

The value of num in the class Test is: 175

We define the term "inner class" to the nested class that is:

* + declared inside the body of another class.
  + not declared inside a method of another class.
  + not a static nested class.
  + not an anonymous inner class.
* An example:

class Outer{

class Inner{

}

}

* When we compile the above code we get 2 class files:
  + Outer.class
  + Outer$Inner.class
* Notice that inner class is tied to its outer class though it is still a separate class.
* An inner class cannot have any kind of static code including the public static void main(String[] args).
* Only classes with "public static void main(String[] args)" can be called using "java" command.
* In our earlier example, Inner class didn't have a static main method. So, we can't call java Outer$Inner!
* The inner class is just like any other member of its enclosing class.
* It has access to all of its enclosing class' members including private.

There are 2 ways to instantiate an instance of inner class:

* From within its outer class
* From outside its outer class
* class Outer{
* private String s = "Outer string"; //Outer instance variable
* Inner i1 = new Inner();
* void getS(){
* System.out.println(s);
* }
* void getInnerS(){
* System.out.println(i1.s);
* }
* class Inner{
* private String s = "Inner string"; //Inner instance variable, uninitialized
* void getS(){
* System.out.println(s);
* }
* void getOuterS(){
* System.out.println(Outer.this.s);
* }
* }
* public static void main(String[] args){
* Outer o = new Outer();
* Outer.Inner oi = o.new Inner();//can also be new Outer().new Inner();
* o.getS();
* oi.getS();
* o.getInnerS();
* oi.getOuterS();
* }
* }
* Output:
* Outer string
* Inner string
* Inner string
* Outer string

## **Method-local Inner Class**

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

A method-local inner class can be instantiated only within the method where the inner class is defined. The following program shows how to use a method-local inner class.

**Example**

public class Outerclass {

// instance method of the outer class

void my\_Method() {

int num = 23;

// method-local inner class

class MethodInner\_Demo {

public void print() {

System.out.println("This is method inner class "+num);

}

} // end of inner class

// Accessing the inner class

MethodInner\_Demo inner = new MethodInner\_Demo();

inner.print();

}

public static void main(String args[]) {

Outerclass outer = new Outerclass();

outer.my\_Method();

}

}

If you compile and execute the above program, you will get the following result −

**Output**

This is method inner class 23

## **Anonymous Inner Class**

* Inner classes that are declared wtihout a name are called anonymous inner classes.
* Anonymous inner classes have no name, and their type must be either a subclass of the named type or an implementer of the named interface.
* An anonymous inner class is always created as part of a statement; don't forget to close the statement after the class definition with a curly brace. This is a rare case in Java, a curly brace followed by a semicolon.
* Because of polymorphism, the only methods you can call on an anonymous inner class reference are those defined in the reference variable class (or interface), even though the anonymous class is really a subclass or implementer of the reference variable type.
* An anonymous inner class can extend one subclass or implement one interface, Unlike non-anonymous classes (inner or otherwise), an anonymous inner class cannot do both. In other words, it cannot both extend a class and implement an interface, nor can it implement more than one interface.
* An argument-local inner class is declared, defined, and automatically instantiated as part of a method invocation. The key to remember is that the class is being defined within a method argument, so the syntax will end the class definition with a curly brace, followed by a closing parenthesis to end the method call, followed by a semicolon to end the statement: });

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

**Syntax**

AnonymousInner an\_inner = new AnonymousInner() {

public void my\_method() {

........

........

}

};

The following program shows how to override the method of a class using anonymous inner class.

**Example**

abstract class AnonymousInner {

public abstract void mymethod();

}

public class Outer\_class {

public static void main(String args[]) {

AnonymousInner inner = new AnonymousInner() {

public void mymethod() {

System.out.println("This is an example of anonymous inner class");

}

};

inner.mymethod();

}

}

If you compile and execute the above program, you will get the following result −

**Output**

This is an example of anonymous inner class

In the same way, you can override the methods of the concrete class as well as the interface using an anonymous inner class.

## **Anonymous Inner Class as Argument**

Generally, if a method accepts an object of an interface, an abstract class, or a concrete class, then we can implement the interface, extend the abstract class, and pass the object to the method. If it is a class, then we can directly pass it to the method.

But in all the three cases, you can pass an anonymous inner class to the method. Here is the syntax of passing an anonymous inner class as a method argument −

obj.my\_Method(new My\_Class() {

public void Do() {

.....

.....

}

});

The following program shows how to pass an anonymous inner class as a method argument.

**Example**

// interface

interface Message {

String greet();

}

public class My\_class {

// method which accepts the object of interface Message

public void displayMessage(Message m) {

System.out.println(m.greet() +

", This is an example of anonymous inner class as an argument");

}

public static void main(String args[]) {

// Instantiating the class

My\_class obj = new My\_class();

// Passing an anonymous inner class as an argument

obj.displayMessage(new Message() {

public String greet() {

return "Hello";

}

});

}

}

If you compile and execute the above program, it gives you the following result −

**Output**

Hello This is an example of anonymous inner class as an argument

## **Static Nested Class**

* Nested classes that are declared "static" are called static nested classes.
* Static nested classes are inner classes marked with the static modifier.
* A static nested class is not an inner class, it's a top-level nested class.
* Because the nested class is static, it does not share any special relationship with an instance of the outer class. In fact, you don't need an instance of the outer class to instantiate a static nested class.
* Instantiating a static nested class requires using both the outer and nested class names as follows:

BigOuter.Nested n = new BigOuter.Nested();

* A static nested class cannot access non-static members of the outer class, since it does not have an implicit reference to any outer instance (in other words, the nested class instance does not get an outer this reference).

A static inner class is a nested class which is a static member of the outer class. It can be accessed without instantiating the outer class, using other static members. Just like static members, a static nested class does not have access to the instance variables and methods of the outer class. The syntax of static nested class is as follows −

**Syntax**

class MyOuter {

static class Nested\_Demo {

}

}

Instantiating a static nested class is a bit different from instantiating an inner class. The following program shows how to use a static nested class.

**Example**

public class Outer {

static class Nested\_Demo {

public void my\_method() {

System.out.println("This is my nested class");

}

}

public static void main(String args[]) {

Outer.Nested\_Demo nested = new Outer.Nested\_Demo();

nested.my\_method();

}

}

If you compile and execute the above program, you will get the following result −

**Output**

This is my nested class

Q**= generics important points?**

Ans=

List<?> list = new ArrayList<?>(); X (unexpected type found:? required class or interface without bounds)

List<?> list = new ArrayList<? Extends Number>(); X(unexpected type found:? Extends Number required class or interface without bound)

List<?> list = new ArrayList<String>(); Right

List<?> list = new ArrayList<Integer>(); Right

List<? Extends Number> list = new ArrayList<String>(); X (compile time error incompatible type found AL<String> required AL<? extends Number> )

List<? Super String> list = new ArrayList<Object>(); Right

Public void addList(List<?> list){

list.add(“A”); X

list.add(1); X

list.add(23.33); X

list.add(null); Y (because null is correct value of any type)

}

Than question is what is the use of above method is we can write any thing.Ans is these types of method is used for read only operation only.

Public void addList(List<?> list){

Sop(list); Y

}

->M1(List<? Extends x> list){

}

In above method we can call this method using interface and class or using its sub classes.

But with in the method we can’t add anything to list except null b/c we don’t know the type X exactly.

This type of methos also best suitable for read only operation.

->M1(List<? Super x) l){}

X can be either class or interface, if x is a class then we call this method by passing arraylist of either x type or its super classes if x is an interface then we can call this method by passing arrayList of either x type or super class of implementation class of x.

We can add null or x or its super classes.

Runnable --- > Thread class -🡪 Object

**Declaring type parameter yet class level:**

Class <T>{

We can use T with in this class based on our requirement.

}

**Declaring type parameter yet method level:**

We have to declare type parameter just before return type.

**public** <T> **boolean** add(T t){

we can use T any where within this method based on our requirement.

**return** **true**;

}

Bounded type methods in java:-

Public <T> void m1(); Y

Public <T extends Number > void m1(); Y

Public <T extends Runnable> void m1(); Y

Public <T extends Number & Runnable> void m1(); Y

Public <T extends cumparable & Runnable> void m1(); Y

Public <T extends Number & cumparable & Runnable> void m1(); Y

Public <T extends Runnable & Number> void m1(); X (first we have to take class then interface.)

Public <T extends Thread & Number> void m1(); X (We can’t extends more than one class)

* + - If we send non generic object to generic area then it start behaving like generic object same vice versa.

Ex- main(){

List<String> list = new Arraylist<>();

List.add(“a”);

List.add(“b”);

Show(list);

}

Public void show(ArrayList l){

L.add(12); // here we can use any thing it behaves like non generic

l.add(12.324);

}

* + - Then main purpose of generics is to provide type safty and to resolve type casting problems.
    - Type safety and type casting both are applicable yet compile time hence generics concept also applicable only at compile time but not yet run time.
    - Yet the time of compilation at the last step generic syntax will be removed and hence for the jvm generics systex won’t be available.

Ex:

Public void m1(List<String> list){}

Public void m1(List<Integer> list){} // give compile time error b/c its know at run time generics will remove and bonth method will become same.

Q=stack in java?

Ans=

Ans=Stack is a subclass of Vector that implements a standard last-in, first-out stack.

Stack only defines the default constructor, which creates an empty stack. Stack includes all the methods defined by Vector, and adds several of its own.

* 1. **Object push(Object element)** : Pushes an element on the top of the stack.  
       
     2. **Object pop()** : Removes and returns the top element of the stack. An ‘EmptyStackException’ exception is thrown if we call pop() when the invoking stack is empty.  
       
     3. **Object peek( )** : Returns the element on the top of the stack, but does not remove it.  
       
     4. **boolean empty()** : It returns true if nothing is on the top of the stack. Else, returns false.  
       
     5. **int search(Object element)** : It determines whether an object exists in the stack. If the element is found, it returns the position of the element from the top of the stack. Else, it returns -1.

**Q=BitSet in java?**

Ans=The BitSet class creates a special type of array that holds bit values. The BitSet array can increase in size as needed. This makes it similar to a vector of bits. This is a legacy class but it has been completely re-engineered in Java 2, version 1.4.

**Constructors:**

**BitSet class Constructors**

/ \

BitSet() BitSet(int no\_Of\_Bits)

* **BitSet() :**A no-argument constructor to create an empty BitSet object.
* **BitSet(int no\_Of\_Bits) :**A one-constructor with an integer argument to create an instance of the BitSet class with an initial size of the integer argument representing the number of bits.

# Q=Queue Interface In Java?

# Ans=The java.util.Queue is a subtype of java.util.Collection interface. Java Queue interface orders the element in FIFO(First In First Out) manner. In FIFO, first element is removed first and last element is removed at last.

### **Methods of Java Queue Interface**

|  |  |
| --- | --- |
| **Method** | **Description** |
| boolean add(object) | It is used to insert the specified element into this queue and return true  upon success. |
| boolean offer(object) | It is used to insert the specified element into this queue. |
| Object remove() | It is used to retrieves and removes the head of this queue. |
| Object poll() | It is used to retrieves and removes the head of this queue, or returns null if this queue is empty. |
| Object element() | It is used to retrieves, but does not remove, the head of this queue. |
| Object peek() | It is used to retrieves, but does not remove, the head of this queue, or returns null if this queue is empty. |

You can choose between the following Queue implementations in the Java Collections API:

* java.util.LinkedList
* java.util.PriorityQueue

LinkedList is a pretty standard queue implementation.

PriorityQueue stores its elements internally according to their natural order (if they implement Comparable), or according to a Comparator passed to the PriorityQueue.

### **Q=*difference between element(), peek(), poll() and remove() methods of the Queue interface?***

* Ans=The peek() method retrieves the value of the first element of the queue**without removing**it from the queue. For each invocation of the method we always get the same value and its execution  
  does not affect the size of the queue. **If the queue is empty the peek() method returns null.**
* The element() method behaves like peek(), so it again retrieves the value of the first element **without removing it.**Unlike peek ), however, **if the list is empty element() throws a NoSuchElementException**
* The poll() method retrieves the value of the first element of the queue **by removing it from the queue.**. At each invocation it removes the first element of the list and if the list is already empty **it returns null but does not throw any exception**
* The remove() method behaves as the poll() method, so it **removes the first element**of the list and **if the list is empty it throws a NoSuchElementException**

# Q=[Difference between offer() and add() in priority queue in java? [duplicate]](https://stackoverflow.com/questions/15591431/difference-between-offer-and-add-in-priority-queue-in-java)

Ans=The two functions come from two different interfaces that PriorityQueue implements:

* add() comes from Collection.
* offer() comes from Queue.

For a capacity-constrained queue, the difference is that add() always returns true and throws an exception if it can't add the element, whereas offer() is allowed to return false if it can't add the element.

However, this doesn't apply to PriorityQueue; the two functions are synonymous.

# Q=System.exit()

# Ans=The **java.lang.System.exit()** method exits current program by terminating running Java virtual machine. This method takes a status code. A non-zero value of status code is generally used to indicate abnormal termination.

# **exit(0)** : Generally used to indicate successful termination. **exit(1) or exit(-1) or any other non-zero value** – Generally indicates unsuccessful termination.

# Q=Different ways for Integer to String Conversions In Java

public static String toString(int i)

# String str3 = String.valueOf(1234);

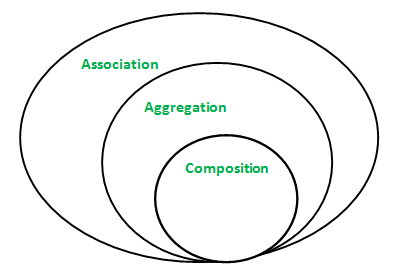
# String str4 = new Integer(d).toString();

int e = 12345;

    DecimalFormat df = new DecimalFormat("#");

    String str5 = df.format(e);

# Q=Association, Composition and Aggregation in Java

Association is relation between two separate classes which establishes through their Objects. Association can be one-to-one, one-to-many, many-to-one, many-to-many.  
In Object-Oriented programming, an Object communicates to other Object to use functionality and services provided by that object. **Composition** and **Aggregation** are the two forms of association.  
[](http://cdncontribute.geeksforgeeks.org/wp-content/uploads/AssociationAggregation-and-Composition.png)

**Aggregation**

It is a special form of Association where:

* It represents **Has-A** relationship.
* It is a **unidirectional association** i.e. a one way relationship. For example, department can have students but vice versa is not possible and thus unidirectional in nature.
* In Aggregation,**both the entries can survive individually** which means ending one entity will not effect the other entity

**Composition**

Composition is a restricted form of Aggregation in which two entities are highly dependent on each other.

* It represents **part-of** relationship.
* In composition, both the entities are dependent on each other.
* When there is a composition between two entities, the composed object **cannot exist**without the other entity.

Lets take example of**Library**.

|  |
| --- |
| // Java program to illustrate  // the concept of Composition  import java.io.\*;  import java.util.\*;    // class book  class Book  {        public String title;      public String author;        Book(String title, String author)      {            this.title = title;          this.author = author;      }  }    // Libary class contains  // list of books.  class Library  {        // reference to refer to list of books.      private final List<Book> books;        Library (List<Book> books)      {          this.books = books;      }        public List<Book> getTotalBooksInLibrary(){           return books;      }    }    // main method  class GFG  {      public static void main (String[] args)      {            // Creating the Objects of Book class.          Book b1 = new Book("EffectiveJ Java", "Joshua Bloch");          Book b2 = new Book("Thinking in Java", "Bruce Eckel");          Book b3 = new Book("Java: The Complete Reference", "Herbert Schildt");            // Creating the list which contains the          // no. of books.          List<Book> books = new ArrayList<Book>();          books.add(b1);          books.add(b2);          books.add(b3);            Library library = new Library(books);            List<Book> bks = library.getTotalBooksInLibrary();          for(Book bk : bks){                System.out.println("Title : " + bk.title + " and "              +" Author : " + bk.author);          }      }  } |

Run on IDE

Output

Title : EffectiveJ Java and Author : Joshua Bloch

Title : Thinking in Java and Author : Bruce Eckel

Title : Java: The Complete Reference and Author : Herbert Schildt

In above example a library can have no. of **books** on same or different subjects. So, If Library gets destroyed then All books within that particular library will be destroyed. i.e. book can not exist without library. That’s why it is composition.

**Aggregation vs Composition**

1. **Dependency:** Aggregation implies a relationship where the child **can exist independently**of the parent. For example, Bank and Employee, delete the Bank and the Employee still exist. whereas Composition implies a relationship where the child **cannot exist independent** of the parent. Example: Human and heart, heart don’t exist separate to a Human
2. **Type of Relationship:** Aggregation relation is **“has-a”** and composition is **“part-of”**relation.
3. **Type of association:**Composition is a **strong** Association whereas Aggregation is a **weak**Association.

**Q=what is solid design principle?**

**Ans=https://howtodoinjava.com/best-practices/5-class-design-principles-solid-in-java/**

Classes are the building blocks of your java application. If these blocks are not strong, your building (i.e. application) is going to face the tough time in future. This essentially means that not so well-written can lead to very difficult situations when the application scope goes up or application faces certain design issues either in production or maintenance.

On the other hand, set of well designed and written classes can speed up the coding process by leaps and bounds, while reducing the number of bugs in comparison.

In this post, I will list down 5 most recommended design principles, you should keep in mind, while writing your classes. These design principles are called SOLID, in short. They also form the [**best practices**](https://howtodoinjava.com/category/best-practices/) to be followed for designing your application classes.

**Table Of Contents**

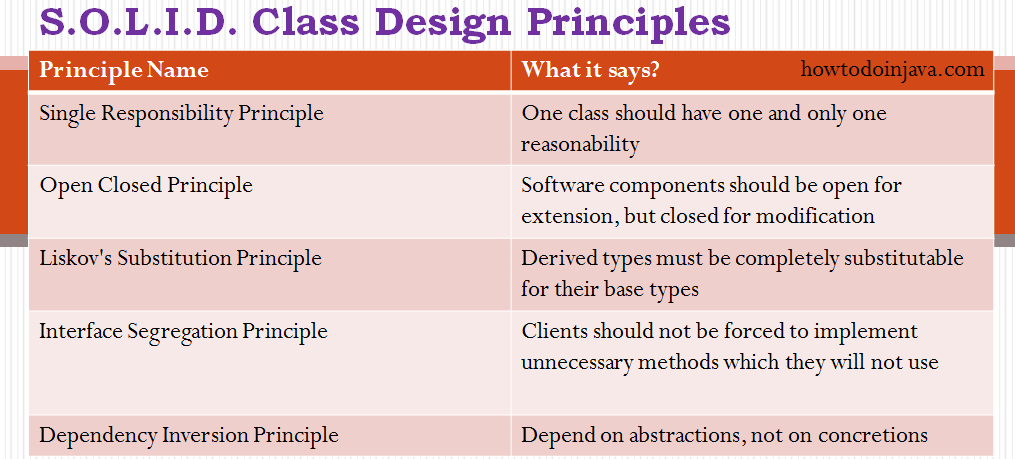
[Single Responsibility Principle](https://howtodoinjava.com/best-practices/5-class-design-principles-solid-in-java/#SRP)

[Open Closed Principle](https://howtodoinjava.com/best-practices/5-class-design-principles-solid-in-java/#OCP)

[Liskov's Substitution Principle](https://howtodoinjava.com/best-practices/5-class-design-principles-solid-in-java/#LSP)

[Interface Segregation Principle](https://howtodoinjava.com/best-practices/5-class-design-principles-solid-in-java/#ISP)

[Dependency Inversion Principle](https://howtodoinjava.com/best-practices/5-class-design-principles-solid-in-java/#DI)

5 java class design principles

Lets drill down all of them one by one.

## **Single Responsibility Principle**

The name of the principle says it all:

**"One class should have one and only one responsibility"**

In other words, you should write, change and maintain a class for only one purpose. If it is model class then it should strictly represent only one actor/ entity. This will give you the flexibility to make changes in future without worrying the impacts of changes for another entity.

Similarly, If you are writing service/manager class then it should contain only that part of method calls and nothing else. Not even utility global functions related to module. Better separate them in another globally accessible class file. This will help in maintaining the class for that particular purpose, and you can decide the visibility of class to specific module only.

## **Open Closed Principle**

This is second important rule which you should keep in mind while designing your application. It says:

**"Software components should be open for extension, but closed for modification"**

What does it mean?? It means that your classes should be designed such a way that whenever fellow developers wants to change the flow of control in specific conditions in application, all they need to extend your class and override some functions and that’s it.

If other developers are not able to design desired behavior due to constraints put by your class, then you should reconsider changing your class. I do not mean here that anybody can change the whole logic of your class, but he/she should be able to override the options provided by software in unharmful way permitted by software.

For example, if you take a look into any good framework like struts or spring, you will see that you can not change their core logic and request processing, BUT you modify the desired application flow just by extending some classes and plugin them in configuration files.

## **Liskov’s Substitution Principle**

This principle is a variation of previously discussed open closed principle. It says:

**"Derived types must be completely substitutable for their base types"**

It means that the classes fellow developer created by extending your class should be able to fit in application without failure. I.e. if a fellow developer poorly extended some part of your class and injected into framework/ application then it should not break the application or should not throw fatal [**exceptions**](https://howtodoinjava.com/best-practices/java-exception-handling-best-practices/).

This can be insured by using strictly following first rule. If your base class is doing one thing strictly, the fellow developer will override only one feature incorrectly in worst case. This can cause some errors in one area, but whole application will not do down.

## **Interface Segregation Principle**

This principle is my favorite one. It is applicable to interfaces as single responsibility principle holds to classes. It says:

**"Clients should not be forced to implement unnecessary methods which they will not use"**

Take an example. Developer Alex created an interface Reportable and added two methods generateExcel()and generatedPdf(). Now client ‘A’ wants to use this interface but he intend to use reports only in PDF format and not in excel. Will he achieve the functionality easily.

NO. He will have to implement two methods, out of which one is extra burden put on him by designer of software. Either he will implement another method or leave it blank. So are not desired cases, right??

So what is the solution? Solution is to create two interfaces by breaking the existing one. They should be like PdfReportable and ExcelReportable. This will give the flexibility to user to use only required functionality only.

## **Dependency Inversion Principle**

Most of us are already familiar with the words used in principle’s name. It says:

**"Depend on abstractions, not on concretions"**

In other words. you should design your software in such a way that various modules can be separated from each other using an abstract layer to bind them together. The classical use of this principle of **[BeanFactory](https://howtodoinjava.com/spring/spring-core/different-spring-3-ioc-containers-with-example/" \t "_blank" \o "Different spring 3 IoC containers with example)** in [**spring framework**](https://howtodoinjava.com/java-spring-framework-tutorials/). In spring framework, all modules are provided as separate components which can work together by simply injected dependencies in other module. They are so well closed in their boundaries that you can use them in other software modules apart from spring with same ease.

This has been achieved by dependency inversion and open closed principles. All modules expose only abstraction which is useful in extending the functionality or plugin in another module.

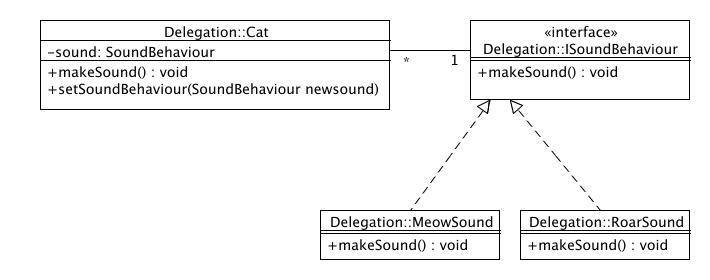
## **Q=**DRY (Don't repeat yourself)

Our first object oriented design principle is DRY, as name suggest **DRY (don't repeat yourself)** means don't write duplicate code, instead use [Abstraction](http://javarevisited.blogspot.com/2010/10/abstraction-in-java.html) to abstract common things in one place. If you have block of code in more than two place consider making it a separate method, or if you use a hard-coded value more than one time make them [public final constant](http://javarevisited.blogspot.com/2011/12/final-variable-method-class-java.html). Benefit of this Object oriented design principle is in maintenance. It's important  not to abuse it, duplication is not for code, but for functionality . It means, if you used common code to validate OrderID and SSN it doesn’t mean they are same or they will remain same in future. By using common code for two different functionality or thing you closely couple them forever and when your OrderID changes its format , your SSN validation code will break. So beware of such coupling and just don’t combine anything which uses similar code but are not related.

Q=**Delegate Pattern (how to delegate a task in class)?**

Ans=It is a technique where an object expresses certain behavior to the outside but in reality delegates responsibility for implementing that behaviour to an associated object. This sounds at first very similar to the proxy pattern, but it serves a much different purpose. Delegation is an abstraction mechanism which centralizes object (method) behaviour.

This example is actually more advanced than just plain delegation, it shows how delegation makes it easy to compose behaviors at run-time.



### **Sample**

First lets create an Interface for our Delegates, since Delegates can be interchanged they all must implement the same interface:

public interface ISoundBehaviour {

public void makeSound();

}

public class MeowSound implements ISoundBehaviour {

public void makeSound() {

System.out.println("Meow");

}

}

public class RoarSound implements ISoundBehaviour {

public void makeSound() {

System.out.println("Roar!");

}

}

Now lets create a cat class which uses the MeowSound per default:

public class Cat {

private ISoundBehaviour sound = new MeowSound();

public void makeSound() {

this.sound.makeSound();

}

public void setSoundBehaviour(ISoundBehaviour newsound) {

this.sound = newsound;

}

}

Finally a Main class to test our delegation pattern:

public class Main {

public static void main(String args[]) {

Cat c = new Cat();

// Delegation

c.makeSound(); // Output: Meow

// now to change the sound it makes

ISoundBehaviour newsound = new RoarSound();

c.setSoundBehaviour(newsound);

// Delegation

c.makeSound(); // Output: Roar!

}

}

Q=What is auto boxing and auto unboxing in java?Suppose we have one variable Integer I = null;

And I write int value = I; what happens?

Ans=The automatic conversion of primitive data types into its equivalent Wrapper type is known as boxing and opposite operation is known as unboxing.

In above scenario it gives null pointer exception.

**Q=HAshMAp**

Ans=

* + 1. When hashcode and eqals will not implement then always value will go to new bucket and we can not fetch that value because hashcode will call Object’s hashcode and it will always different.
    2. When hashcode return any constant value then always it goes to same bucket in linked list if eqals is different. If eqals will same then it will override as usual.
    3. When eqals will not implement then hashcode

Q=what is bridge method in java generics?