1. [About Class and Instance](#About_Class_instance)
2. [Conditional Operators](#Conditional_operators)
3. [Bitwise Operators](#Bitwise_operators)
4. [Overloading\_static\_dyanmic](#Overloading_static_dyanmic) (Method overloading : (static polymorphism –compile time)
5. [JRE](#JRE)
6. [Why String is Immutable](#why_string_immutable)
7. [Diff between instance & class variables](#diff_instance_class_variables)
8. [Java I/O diagram](#JAVA_io)
9. [InputStream Vs InstreamReader](#InputStream_InputStreamReader)
10. [Marker Interface](#Marker_interface)
11. [Difference between Serializable and Externalizable](#Diff_Serializable_Externalizable)
12. t[ransient keyword](#Transient)
13. [Callback\_interface](#Callback_interface)
14. [How HashMap works in java](#HashMap)
15. [Diff\_subSequence and Substring](#diff_subSequence_substring)
16. [Var args](#Varargs)
17. [Autoboxing in java](#Autoboxing)
18. [useof\_nestedclasses](#useof_nestedclasses)
19. [diff\_string\_new\_and\_literal](#diff_string_new_and_literal)
20. How [substring\_working](#substring_working)
21. [Comparator\_Vs\_Comparable](#Comparator_Vs_Comparable)
22. [FailSafe\_VS\_Failfast](#FailSafe_VS_FailFast) (Contrary to fail-fast Iterator, **fail-safe iterator** doesn't throw any Exception if Collection is modified structurally
23. [Difference Enumeration & Iterator](#Diff_Enumeration_Iterator) (Iterators allow the caller to remove elements from the underlying collection during the iteration)
24. [Diff\_Runnable\_Callable](#Diff_Runnable_Callable)
25. [**Does code form finally executes if method returns before finally block or JVM exits ?**](#finally_block_ifmethod_returns)
26. [**What is difference in final, finalize and finally keyword in Java?**](#final_finalize_finally) **(**final and finally are keyword, while finalize is method)
27. [Blocking method](#Blocking_methods)(System.in.read(), waits)
28. [Heap and Non Heap Memory](#Heap_NonHeap_memory)
29. [Heap Structure- JVM generations](#Heap_structure)
30. [Reflection in java](#reflection)
31. [COLLECTION COMPARISION](#Collection_Comparison)
32. [When to use which collection](#When_to_use_List_Set_Map)
33. [Reference Types](#Reference_types_java)
34. [Cyclic Barrier (](#CycliciBarrier)allows multiple threads to wait for each other (barrier) before proceeding.
35. [**When do you override hashcode and equals() ?**](#When_Override_hasCode_equals)
36. [diff\_instance\_class\_variables](#diff_instance_class_variables)
37. Single ton
38. [Difference between object ,instance.](#Diff_instance_object)

**JRE**

JRE = JVM + Java Packages Classes(like util, math, lang, awt,swing etc)+runtime libraries.

## Conditional Operators

&& Conditional-AND

|| Conditional-OR

?: Ternary (shorthand for

if-then-else statement)

[**http://www.leepoint.net/notes-java/data/expressions/bitops.html**](http://www.leepoint.net/notes-java/data/expressions/bitops.html)

# *[Java Notes](http://www.leepoint.net/notes-java/index.html): Bitwise Operators*

Java's *bitwise* operators operate on individual bits of integer (int and long) values. If an operand is shorter than an int, it is promoted to int before doing the operations.

It helps to know how integers are represented in binary. For example the decimal number 3 is represented as 11 in binary and the decimal number 5 is represented as 101 in binary. Negative integers are store in *two's complement* form. For example, -4 is 1111 1111 1111 1111 1111 1111 1111 1100.

## The bitwise operators

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Operator | Name | Example | Result | Description |
| *a* & *b* | and | 3 & 5 | 1 | 1 if both bits are 1. |
| *a* | *b* | or | 3 | 5 | 7 | 1 if either bit is 1. |
| ***a* ^ *b*** | **xor** | **3 ^ 5** | **6** | **1 if both bits are different.** |
| ~*a* | not | ~3 | -4 | Inverts the bits. |
| *n* << *p* | left shift | 3 <<< 2 | 12 | Shifts the bits of *n* left *p* positions. Zero bits are shifted into the low-order positions. |
| *n* >> *p* | right shift | 5 >> 2 | 1 | Shifts the bits of *n* right *p* positions. If *n* is a 2's complement signed number, the sign bit is shifted into the high-order positions. |
| *n* >>> *p* | right shift | -4 >>> 28 | 15 | Shifts the bits of *n* right *p* positions. Zeros are shifted into the high-order positions. |

**About class and instance:**

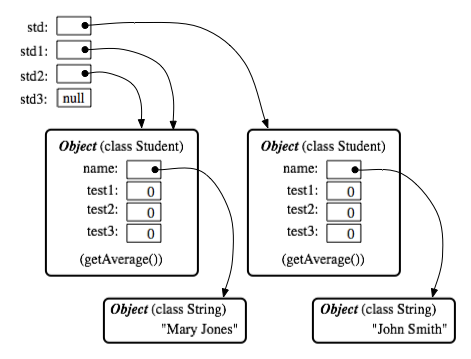
A class defines an object. You can go even further in many languages and say an interface defines common attributes and methods between objects.

An object is something that **can** represent something in the real world. When you want the object to **actually** represent something in the real world that object must be instantiated. Instantiation means you must define the characteristics (attributes) of this specific object, usually through a constructor.

Once you have defined these characteristics you now have an instance of an object.

Example:

A blueprint for a house design is like a class description. All the houses built from that blueprint are objects of that class. A given house is an instance.



Objects are things in memory while instances are things that reference to them. In the above pic: std(instance) -> Student Object (right)

std1(instance) -> Student Object (left)

std2(instance) -> Student Object (left)

std3(instance) -> no object (null)

Why String is immutable or final in Java

String A = "Test"  
String B = "Test"   
  
Now String B called "Test".toUpperCase() which change the same object into "TEST" , so A will also be "TEST" which is not desirable.

2)String has been widely used as parameter for many Java classes e.g. for opening network connection, you can pass hostname and port number as string , you can pass database URL as string for opening database connection, you can [open any file in Java](http://javarevisited.blogspot.sg/2012/07/read-file-line-by-line-java-example-scanner.html) by passing name of file as argument to File I/O classes.  
  
In case, if String is not immutable, this would lead serious security threat , I mean some one can access to any file for which he has authorization, and then can change the file name either deliberately or accidentally and gain access of those file. Because of immutability, you don't need to worry about those kind of threats. This reason also gel with, **Why String is final in Java**, by making java.lang.String final, Java designer ensured that no one overrides any behavior of String class

Examples:

Consider a scenario, in a banking application for money transfer - the beneficiary account number is defined in a string as "0789567345". If by mistake/intentionally this acc. number is changed, money will go to a wrong account.

Another scenario - if someone change the class name anywhere between processing as ..

getClass().getName().subString(0, 5);

The Class loader will simply say 'Class Not Found

[What is the difference between an instance and a class (static) variable in Java](http://stackoverflow.com/questions/15486392/what-is-the-difference-between-an-instance-and-a-class-static-variable-in-java)

A static variable is shared by all instances of the class. and in case of instance variable each instance of class have different copy.

Static variable memory allocate at compile time, They are loaded at load time and initialized at class initialization time and in case of instance variable everything is done at run time.

You can understand by example.

## Example:

An instance variable is one per Object, every object has its own copy of instance variable.

public class Test{

int x = 5;

}

Test t1 = new Test();

Test t2 = new Test();

Both t1 and t2 will have its own copy of x.

A static variable is one per Class, every object of that class shares the same Static variable.

public class Test{

public static int x = 5;

}

Test t1 = new Test();

Test t2 = new Test();

Both t1 and t2 will have the exactly one x to share between them.

‘

JAVA IO

There are byte streams – currently no use ,they use platform specific encoding(read byte by byte, to check end of line stream. Read!=-1

Ex: PrintStream, InputStream,OutputStream

*Character streams : thes are like wrappers to byte streams.*

Reads character by character and portable across the plotforms.

FileWriter and PrinterWriter,InputReader and outputReader. FileReader and FileWriter

Java.io.Scanner:

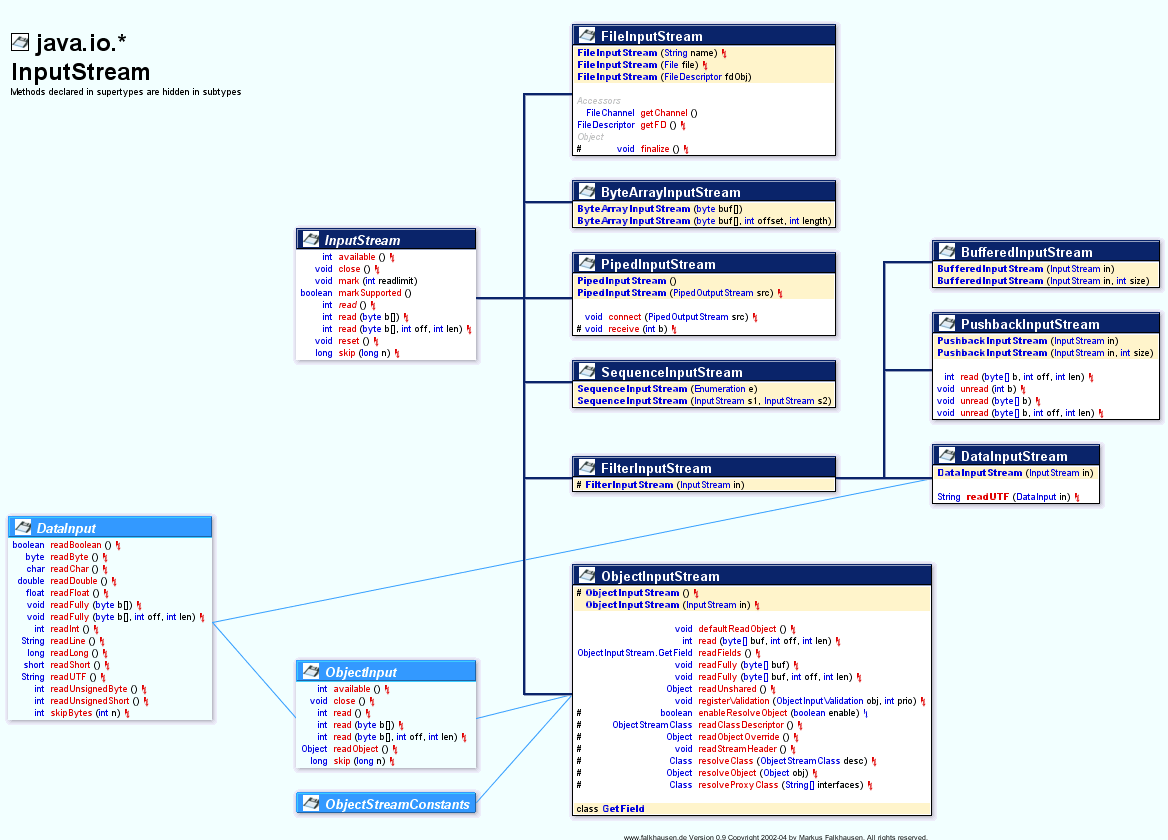
Objects of type [Scanner](http://docs.oracle.com/javase/7/docs/api/java/util/Scanner.html) are useful for breaking down formatted input into tokens and translating individual tokens according to their data type.

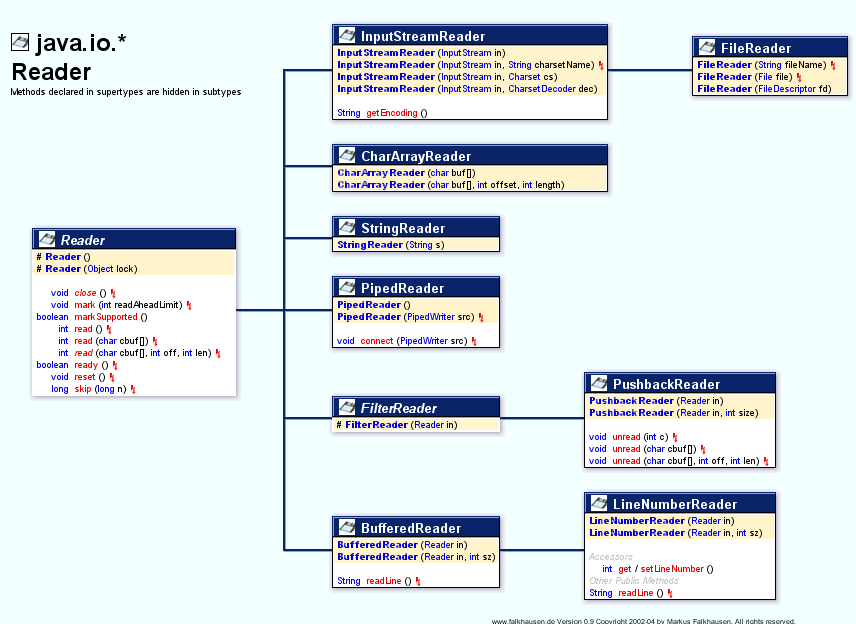
s = new Scanner(new BufferedReader(new FileReader("xanadu.txt"))); //default delimeter space

s.useDelimiter(",\\s\*");

The java.io package contains many classes that your programs can use to read and write data.

The java.nio.file package provides extensive support for file and file system I/O. This is a very comprehensive.





IMP:

**[InputStream vs InputStreamReader](http://stackoverflow.com/questions/3194918/inputstream-vs-inputstreamreader)**

The simple answer is: if you need binary data you can use an InputStream (also a specific one like a DataInputStream), if you need to work with text use an InputStreamReader.

Market interface:

Marker interface is the one which does not have any methods. It is main purpose is to provide meta information about the class .

Ex: Serializable, closeable interface.

**Marker interface in Java** is interfaces with no field or methods or in simple word **empty interface in java is called marker interface**.

**Example of market interface is** Serializable, Clonnable and Remote interface. Now if marker interface doesn't have any field or method or behavior they why would Java needs it?

In summary **marker interface in Java is used to indicate something to compiler, JVM** or any other tool but **Annotation** is better way of doing same thing.  
  
Read more: <http://javarevisited.blogspot.com/2012/01/what-is-marker-interfaces-in-java-and.html#ixzz30Pny9Gpv>

Difference between *Serializable* and *Externalizable*

In case of Serializable Java Virtual machine has full control for serializing object while in case of Externalizable, application gets control for persisting objects. writeExternal() and readExternal() method provides complete **control on format and content of Serialization process** to application which can be leverage to increase performance and speed of serialization process.  
  
Read more: <http://javarevisited.blogspot.com/2012/01/serializable-externalizable-in-java.html#ixzz2woHUJeRa>

transient keyword

In One word **transient keyword** is used in serialization process to prevent any variable from being serialized, so if you have any field which is not making sense to serialize, you can simply declare that as transient and it won't be serialized. In this article we will revise some basics like

What is transient variable in java, why do we need transient variable and most importantly where should we use transient variable or **which fields need to be declared as transient** with example.

### Example of transient variable in java

To understand the concept of transient variables let see a live example in java.

public class **Stock** {

    private **transient** Logger logger = Logger.getLogger(Stock.class); **//will not serialized**

    private String symbol; //will be serialized

    private BigInteger price; //serialized

    private long quantity; //serialized

}

1. Transient keyword can only be applied to fields or member variable. Applying it to method or local variable is compilation error.

2) Another important point is that you can declare an variable static and transient at same time and java compiler doesn't complain but doing that doesn't make any sense because transient is to instruct "do not save this field" and static variables are not saved anyway during serialization.

3) In similar way you can apply transient and final keyword together to a variable compiler will not complain but you will face another problem of reinitializing a final variable during deserialization.

4) Transient variable in java is not persisted or saved when an object gets serialized.

Read more: <http://javarevisited.blogspot.com/2011/09/transient-keyword-variable-in-java.html#ixzz35TliIEj2>

Can Enum extend class in Java? (No, because Java allows a class to only extend one class and enum by default extends java.lang.Enum)

**String is immutable while StringBuffer and StringBuilder is mutable object**.  
  
Read more: <http://javarevisited.blogspot.com/2011/07/string-vs-stringbuffer-vs-stringbuilder.html#ixzz2woKzdsuf>

Callback interface in java

Many are confused by what a callback is because of the name of the damned thing.

A callback method is one which is passed as an argument from another method which is invoked due to some kind of event. The 'call back' nature of the argument is that it returns its result to the method that provided it as an argument - that is to say that it 'calls back' with the return value of the callback method.

//An innocuous looking method which will become known as a callback method

//because of the way in which we will invoke it.

int meaningOfLife() {

return 42;

}

//An innocuous looking method which just takes an int and prints it to screen

void printANumber(int a\_Number) {

System.out.print(a\_Number);

}

//invoking a method which passes another method as an argument in reaction to an event (the 'another' method - meaningOfLife - is therefore called a callback method) and the event - main() - is that the program is starting

void main() {

printANumber(meaningOfLife());

}

**Call After** would be a better name than the stupid name, **callback**. When or if condition gets met within a function, call another function, the **Call After** function, the one received as argument.

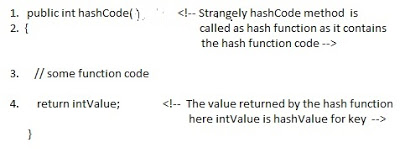
Rather than hard-code an inner function within a function, one writes a function to accept an already-written **Call After** function as argument. The **Call After** might get called based on state changes detected by code in the function receiving the argument.

How HashMap works in java

<http://javahungry.blogspot.com/2013/08/hashing-how-hash-map-works-in-java-or.html>

**How Hashmap works in Java**  
  
HashMap works on the principle of Hashing .  To understand Hashing , we should understand the three terms first   i.e  *Hash Function , Hash Value and Bucket .*  
  
**What is Hash Function , Hash Value  and Bucket ?**  
  
hashCode() function  which returns an integer value is the **Hash function**. The important point to note that ,  this method is present in [Object class ( Mother of all class )](http://javahungry.blogspot.com/2013/06/object-class-and-methods-in-java-example-explanation.html) .  
  
This is the code for the hash function(also known as hashCode method) in Object Class :  
  
    public native int hashCode();

The most important point to note from the above line :  hashCode method return  int value .  
So the **Hash value**isthe int value returned by the hash function**.**  
  
  
    So summarize the terms in the diagram below :  
                   

[](http://3.bp.blogspot.com/-ohpWRYtP3N8/UgVGHwEk3YI/AAAAAAAAAbQ/K5BKPJ8dfmQ/s1600/How+hash+map+works+.jpg)

**What is bucket ?**   
A bucket is used to store key value pairs . A bucket can have multiple key-value pairs . In hash map, bucket used simple linked list to store objects .  
  
After understanding the terms we are ready to move next step , **How hash map works in java** **or How get() works internally in java .**  
  
  
  
**Code inside Java Api (HashMap class internal implementation) for HashMap get(Obejct key) method**

**1**. Public V get(Object key)

{

**2**. **if** (key ==**null**)

**3**. //Some code

**4**. **int** hash = hash(key.hashCode());

**5**. // if key found in hash table then return value

**6**. // else return null

}

**Hash map** **works on the principle of hashing**  
  
HashMap get(Key k) method calls hashCode method on the key object and applies returned hashValue to its own static hash function to find a bucket location(backing array) where keys and values are stored in form of a**nested class called Entry (Map.Entry)**. So you have concluded that from the previous line that**Both key and value is stored in the bucket as a form of  Entry object**. So thinking that Only value is stored  in the bucket is not correct and will not give a good impression on the interviewer .  
  
\* Whenever we call get( Key k )  method on the HashMap object . First it checks that whether key is null or not .  Note that **there can only be one null key in HashMap .**  
 **If key is null , then Null keys always map to hash 0, thus index 0.**  
  
If key is not null then , it will call hashfunction on the key object , see line 4 in above method i.e. key.hashCode()  ,so after key.hashCode() returns hashValue , line 4 looks like  
  
4.                int hash = hash(hashValue)  
  
 , and now ,it applies returned hashValue into its own hashing function .  
  
**We might wonder why we are calculating the hashvalue again using hash(hashValue).** Answer is ,It defends against poor quality hash functions.  
  
Now step 4 final  hashvalue is used to find the bucket location at which the Entry object is stored .**Entry object stores in the bucket like this (hash,key,value,bucketindex) .**  
  
**Interviewer:    What if  when two different keys have the same hashcode ?**  
Solution, [equals() method](http://javahungry.blogspot.com/2013/06/difference-between-equals-and-double-equals-method-with-example-java-collections-interview-question.html) comes to rescue. Here candidate gets puzzled. Since bucket is one and we have two objects with the same *hashcode*. Candidate usually forgets that bucket is a simple linked list.  
  
**The bucket is the linked list effectively . It’s not a LinkedList as in a java.util.LinkedList - It's a separate (simpler) implementation just for the map .**  
  
So we traverse through linked list , comparing keys in each entries using keys.equals() until it return true.  Then the corresponding entry object Value is returned **.**  
  
One of  our readers Jammy  asked a very good  question   
  
**When the functions 'equals' traverses through the linked list does it traverses from start to end one by one...in other words brute method. Or the linked list is sorted based on key and then it traverses?**  
  
Answer is when an element is added/retrieved, same procedure follows:  
  
  
a. Using key.hashCode() [ see above step 4],determine initial hashvalue for the key  
  
b. Pass intial hashvalue as hashValue  in    hash(hashValue) function, to calculate the final hashvalue.  
  
c. Final hash value is then passed as a first parameter in the indexFor(int ,int )method .  
    The second parameter is length which is a constant in HashMap Java Api , represented by                             DEFAULT\_INITIAL\_CAPACITY  
  
    The default  value of DEFAULT\_INITIAL\_CAPACITY is 16 in HashMap Java Api .  
  
 indexFor(int,int) method  returns the first entry in the appropriate bucket. The linked list in the bucket is then iterated over - (the end is found and the element is added or the key is matched and the value is returned )  
  
  
Explanation about indexFor(int,int) is below :

/\*\*

\* Returns index for hash code h.

\*/

**static** **int** **indexFor**(**int** h, **int** length) {

**return** h & (length-**1**);

}

The above function indexFor() works because Java HashMaps always have a capacity, i.e. number of buckets, as a power of 2.  
 Let's work with a capacity of 256,which is 0x100, but it could work with any power of 2. Subtracting 1  
from a power of 2 yields the exact bit mask needed to bitwise-and with the hash to get the proper bucket index, of range 0 to length - 1.  
256 - 1 = 255  
0x100 - 0x1 = 0xFF  
E.g. a hash of 257 (0x101) gets bitwise-anded with 0xFF to yield a bucket number of 1.  
Difference between subSequence() and subString() in java String

Using str.subSequence(begin, end) returns a [CharSequence](http://docs.oracle.com/javase/6/docs/api/java/lang/CharSequence.html) which is a read only form of the string represented as a sequence of chars. For Example:

String string = "Hello";

CharSequence subSequence = s.subSequence(0, 5);

Its read only in the sense that you can't change the chars within the CharSequence without instantiating a new instance of a CharSequence.

If you have to use str.subSequence(begin, end), you can cast the result to a String:

String string = "Hello";

String subSequence = (String) s.subSequence(0, 5);

and use all the normal String operators like subSequence += " World";

### Difference between HashMap and ConcurrentHashMap in Java Collection

1. As I said earlier first significant difference between HashMap and ConcurrentHashMap is that later is [thread-safe](http://javarevisited.blogspot.sg/2012/01/how-to-write-thread-safe-code-in-java.html) and can be used in concurrent environment without external synchronization. Though it doesn't provide same level of synchronization as achieved by using Hashtable but it’s enough for most practical purpose.  
     
   2)You can make HashMap synchronized by wrapping it on Collections.synchornizedMap(HashMap) which will return a collection which is almost equivalent to Hashtable, where every modification operation on Map is locked on Map object while in case of ConcurrentHashMap, thread-safety is achieved by dividing whole Map into different partition based upon [Concurrency](http://javarevisited.blogspot.sg/2012/07/cyclicbarrier-example-java-5-concurrency-tutorial.html) level and only locking particular portion instead of locking whole Map.  
     
   3) ConcurrentHashMap is more scalable and performs better than Synchronized than HashMap in multi-threaded environment while in Single threaded environment both HashMap and ConcurrentHashMap gives comparable performance, where HashMap only slightly better.

<http://java67.blogspot.sg/2012/08/difference-between-hashmap-and-concurrentHashMap-java-collection.html>

use of transient in java

Google is your friend - [first hit](http://en.wikibooks.org/wiki/Java_Programming/Keywords/transient) - also you might first have a look at what [serialization](http://en.wikipedia.org/wiki/Serialization) is.

It marks a member variable not to be serialized when it is persisted to streams of bytes. When an object is transferred through the network, the object needs to be 'serialized'. Serialization converts the object state to serial bytes. Those bytes are sent over the network and the object is recreated from those bytes. Member variables marked by the java transient keyword are not transferred, they are lost intentionally.

Example from there, slightly modified (thanks @pgras):

public class Foo implements Serializable

{

private String saveMe;

private transient String dontSaveMe;

private transient String password;

//...

}

use of *VOLATILE* keyword in java

**Volatile keyword in Java** is used as an indicator to Java compiler and  [Thread](http://javarevisited.blogspot.com/2011/02/how-to-implement-thread-in-java.html)that do not cache value of this variable and always read it from [main memory](http://javarevisited.blogspot.sg/2011/05/java-heap-space-memory-size-jvm.html). So if you want to share any variable in which read and write operation is atomic by implementation e.g. read and write in int or boolean variable you can declare them as volatile variable. From Java 5 along with major changes like [Autoboxing](http://javarevisited.blogspot.sg/2012/07/auto-boxing-and-unboxing-in-java-be.html), [Enum](http://javarevisited.blogspot.sg/2011/08/enum-in-java-example-tutorial.html),Generics and [Variable arguments](http://javarevisited.blogspot.sg/2011/09/variable-argument-in-java5-varargs.html),  
  
Read more: <http://javarevisited.blogspot.com/2011/06/volatile-keyword-java-example-tutorial.html#ixzz2y3hSTQQg>

[Variable argument or Varargs methods from](http://javarevisited.blogspot.sg/2011/09/variable-argument-in-java5-varargs.html" \o "Variable argument or Varargs methods from Java 5 with Example - Programming Tutorial)

**Variable argument or varargs in Java** allows you to write more flexible methods which can accept as many argument as you need. variable arguments or varargswere added in Java 1.5  
  
**class**  VarargsExample{  
  
  */\*  
   \* @ return multiplication of all numbers in array  
   \* if varargs method accept more than one parameter than varargs arguments  
   \* must be last parameter.  
   \*/*

*//instead of* **public** **int** multiply(**int**[] numbers) \\\\

**public** **int** multiply(**int**... numbers){  
    **int** result = 1;  
      
    **for**(**int** number: numbers){  
      result= result\*number;  
    }  
      
    **return** result  
  }  
}

here is an example of better performance alternative of varargs for 90% of time

public int sum(int a);

public int sum(int a, int b);

public int sum(int... num);

Read more: <http://javarevisited.blogspot.com/2011/09/variable-argument-in-java5-varargs.html#ixzz35UOhC3t3>

Read more: <http://javarevisited.blogspot.com/2011/09/variable-argument-in-java5-varargs.html#ixzz35UOQ1qVU>

**What is Autoboxing in Java**

Autoboxing and unboxing is introduced in Java 1.5 to automatically convert primitive type into boxed primitive( Object or Wrapper class). autoboxing allows you to use primitive and object type interchangeably in Java on many places like assignment, method invocation etc. If you have been using Collections like HashMap or ArrayList before Java 1.5 then you are familiar with the issues like you can not directly put primitives into Collections

Read more: <http://javarevisited.blogspot.com/2012/07/auto-boxing-and-unboxing-in-java-be.html#ixzz2y3iP07l5>

JAVA DECORATOR PATTERN?

Decorator Pattern is one of the famous Gang of Four (GOF) structural design pattern, which provides an alternative way of extending an object's functionality. It's different than traditional way of adding new functionality into object using Inheritance, since it's based on [Composition](http://javarevisited.blogspot.sg/2013/06/why-favor-composition-over-inheritance-java-oops-design.html) and provides additional functionality at runtime, as opposite to Inheritance, which adds new functionalities at compile time. Decorator design pattern is introduced by famous [Gang of Four design pattern book](http://www.amazon.com/dp/0201633612/?tag=javamysqlanta-20), almost 2 decades ago. It's a time tested way of adding new functionalities into object. In this Java design pattern tutorial, we will learn Decorator design pattern by using it in a Java example. This is a best way of learning design pattern, followed you try it yourself to apply in similar scenarios. Decorator pattern is one of the popular design pattern along with Factory method pattern and [Singleton Pattern](http://java67.blogspot.sg/2012/08/what-is-singleton-pattern-in-java.html), and you can see it's usage even in JDK itself. Couple of classes from java.io package e.g. BufferedInputStream, LineNumberInputStream are good example of Decorator design pattern in Java.

### Decorator design pattern in Java

[Real life example of Decorator design pattern in Java](http://3.bp.blogspot.com/-1lzFJzIgaHk/UF2Ci6kY5pI/AAAAAAAAAes/OYiM7r-DHzc/s1600/17.jpg)In order to show you, how to implement Decorator pattern, let me first explain requirements. We need to create software for calculating price for a Sandwich, yummy... no? Since customer can customize sandwich by asking extra cheese or extra fillings, you also need to include cost of those items in final price of Sandwich. Since this customization can vary a lot among different customers and offering from a shop, creating classes for different types of Sandwich with different fillings or extras e.g. BrownBreadSandWithCheese or WhiteBreaSandwitchWithCheeseAndTomato will just clutter code with lots of endless small classes. Now this problem looks a natural fit for applying Decorator pattern, because we have a base object Sandwich, which can be decorated with extra cheese and fillings. By using [Decorator pattern](http://javarevisited.blogspot.com/2011/11/decorator-design-pattern-java-example.html), you can extend functionality of Sandwich at runtime, based upon customer's request, which is impossible with Inheritance until you have a specific class for every possible customer request. **This is one of the reason Why Composition is preferred over Inheritance in Object oriented design** and particularly in Java. Now, let's see our class structure, We have an abstract class Sandwich, with abstract method price() and a concrete implementation class WhiteBreadSandwich, which cost $3.0. Now, in order to provide extra cheese, which obviously incur extra cost, we are going to use *Decorator design pattern*. We have a Decorator abstract class, which will act as base for Decorators called SandwichDecorator, and a concrete implementation of this as CheeseDecorator.

<http://java67.blogspot.com/2013/07/decorator-design-pattern-in-java-real-life-example-tutorial.html>

VARIABLE ARGS IN JAVA

<http://javarevisited.blogspot.sg/2011/09/variable-argument-in-java5-varargs.html>

### Real world Example of varargs in Java

First we look one real world scenario suppose we go one college and take admission on that college now its not really decided that admission will be done for how many student may be 50 student will come or 100 or more than that at a time. So college is one class and Admission is one procedure or method that takes no of student as an argument .So in that method we can use varargs or variable arguments.

***/\*\*  
 \* Simple real world example of variable argument methods  
 \*/***  
**public** **class** college {  
  
**public** **void** admission\_method (**int**... no\_of\_student) {  
 *//rest of code for processing*   
  
}  
  
}

**Simple java variable argument example:**

Let consider one simple example of finding the multiplication of n number. First we will try to solve this problem using method overloading

***/\*\*  
 \* Java Program which tries to implement variable argument method using   
 \* method overloading. This started get clumsy once number of parameter exceeds  
 \* five.  
 \*/***  
**class**  VarargsExample{  
  
  **public** **int** multiply(**int** a,**int** b){ **return** a\*b;}  
  
  **public** **int** multiply(**int** a,**int** b,**int** c){ **return** (a\*b)\*c;}   
  
  **public** **int** multiply(**int** a,**int** b,**int** c,**int** d{ **return** (a\*b)\*(c\*d);}  
  
}

If we use method overloading same method will be repeated again and again and its not worth after four or five parameters. now will use array also to solve this problem of variable arguments:

Let see how:

***/\*\*  
 \* Java Program which tries to implement variable argument method using   
 \* method overloading. This started get clumsy once number of parameter exceeds  
 \* five.  
 \*/***  
**class**  VarargsExample{  
  
  */\*  
   \* @return multiplication of all numbers in array  
   \*/*  
  **public** **int** multiply(**int**[] numbers){  
    **int** result = 1;  
      
    **for**(**int** number: numbers){  
      result= result\*number;  
    }  
      
    **return** result  
  }  
}

Here we need to create an integer array and  pass that array to the method and then iterate the array and get result .  
We can simplify this with **variable argument provided by java 5** where creation of array will be done internally and our task become easier.

***/\*\*  
 \* Java Program which uses varargs feature to accept variable number of   
 \* arguments. variable arguments are implemented using anonymous array so if  
 \* another method with exact same signature except array in place of varargs will result  
 \* in compiler error.  
 \*/***  
**class**  VarargsExample{  
  
  */\*  
   \* @ return multiplication of all numbers in array  
   \* if varargs method accept more than one parameter than varargs arguments  
   \* must be last parameter.  
   \*/*  
  **public** **int** multiply(**int**... numbers){  
    **int** result = 1;  
      
    **for**(**int** number: numbers){  
      result= result\*number;  
    }  
      
    **return** result  
  }  
}

use of nested classes in java

There are two type of nested classes in java: inner classes(non-static) and static nested classes

uses of nested classes:

<http://docs.oracle.com/javase/tutorial/java/javaOO/nested.html>

Compelling reasons for using nested classes include the following:

* **It is a way of logically grouping classes that are only used in one place**: If a class is useful to only one other class, then it is logical to embed it in that class and keep the two together. Nesting such "helper classes" makes their package more streamlined.
* **It increases encapsulation**: Consider two top-level classes, A and B, where B needs access to members of A that would otherwise be declared private. By hiding class B within class A, A's members can be declared private and B can access them. In addition, B itself can be hidden from the outside world.
* **It can lead to more readable and maintainable code**: Nesting small classes within top-level classes places the code closer to where it is used.

**3.** **What is the difference between creating String as new() and literal?**

When we create string with new() Operator, it’s created in heap and not added into string pool while String created using literal are created in String pool itself which exists in [PermGen area of heap](http://javarevisited.blogspot.sg/2012/01/tomcat-javalangoutofmemoryerror-permgen.html).

String str = new String("Test");  
   
does not  put the object str in String pool , we need to call String.intern() method which is used to put  them into String pool explicitly. its only when you create String object as String literal e.g. String s = "Test" Java automatically put that into String pool. By the way there is a catch here, Since we are passing arguments as "Test", which is a String literal, it will also create another object as "Test" on string pool.

**.** **How does substring () inside String works?**

Another good Java interview question, I think answer is not sufficient but here it is “Substring creates new object out of source string by taking a portion of original string”.  This question was mainly asked to see if developer is familiar with risk of memory leak, which substring can create. Until Java 1.7, substring holds reference of original character array, which means even a substring of 5 character long, can prevent 1GB character array from garbage collection, by holding a strong reference. This issue is fixed in Java 1.7, where original character array is not referenced any more, but that change also made creation of substring bit costly in terms of time. Earlier it was on the range of O(1), which could be O(n) in worst case on Java 7.

Read more: <http://javarevisited.blogspot.com/2011/04/top-20-core-java-interview-questions.html#ixzz2y8Ds27iM>

## Comparator vs Comparable in Java

[Difference between Comparator vs Comparable in Java](http://2.bp.blogspot.com/-wrzDeQGAe1I/TWu8pLuLr4I/AAAAAAAAADE/V017G-6Q61w/s1600/java_logo_50_50.jpg)Here are some of the common differences, which is worth remembering to answer this question if asked during a telephonic or face to face interview:

**So in Summary if you want to sort objects based on natural order then use Comparable in Java and if you want to sort on some other attribute of object then use Comparator in Java. Now to understand these concepts lets see an example or real life coding:**

1) Comparator in Java is defined in java.util package while Comparable interface in Java is defined in java.lang package, which very much says that Comparator should be used as an utility to sort objects which Comparable should be provided by default.

2) Comparator interface in Java has method public int compare (Object o1, Object o2) which returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second. While Comparable interface has method public int compareTo(Object o) which returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

3) If you see then logical difference between these two is *Comparator in Java* compare two objects provided to him, while Comparable interface compares "this" reference with the object specified. I have shared lot of tips on [how to override compareTo() method](http://javarevisited.blogspot.sg/2011/11/how-to-override-compareto-method-in.html) and avoid some common mistakes programmer makes while implementing Comparable interface.

4) Comparable in Java is used to implement **natural ordering of object**. In Java API String, Date and wrapper classes implements Comparable interface. It’s always good practice to override compareTo() for value objects.

**5) If any class implement Comparable interface in Java then collection of that object either** [**List**](http://javarevisited.blogspot.sg/2012/04/difference-between-list-and-set-in-java.html) **or Array can be sorted automatically by using  Collections.sort() or Arrays.sort() method and object will be sorted based on their natural order defined by CompareTo method.**

6)Objects which implement *Comparable in Java*  can be used as keys in a SortedMap like [TreeMap](http://javarevisited.blogspot.sg/2011/12/treemap-java-tutorial-example-program.html) or elements in a SortedSet  for example TreeSet, without specifying any Comparator.

These were combination of some theoretical and practical differences between Comparator and Comparator interface in Java. It does help you to decide when to use Comparator vs Comparable but things will be more clear when we some best practices around using both of these interfaces. Now let’s see an example of Comparator in Java:

### Example of using Comparator and Comparable in Java

So in Summary if you want to **sort objects based on natural order** then use Comparable in Java and if you want to sort on some other attribute of object then use Comparator in Java. Now to understand these concepts lets see an example or real life coding:

1) There is class called Person, sort the Person based on person\_id, which is primary key in database

2) Sort the Person based on there name.

For a Person class, sorting based on person\_id can be treated as *natural order sorting* and sorting based on name field can be implemented using Comparator interface. To sort based on person\_id we need to implement compareTo() method.

**public** **class** Person **implements** **Comparable** {  
    **private** **int** person\_id;  
    **private** **String** name;  
      
    /\*\*  
     \* Compare current person with specified person  
     \* return zero if person\_id for both person is same   
     \* return negative if current person\_id is less than specified one  
     \* return positive if specified person\_id is greater than specified one  
     \*/  
  @Override   
    **public** **int** compareTo(Object o) {  
        Person p = (Person) o;   
        **return** **this**.person\_id - o.person\_id ;  
    }  
    ….  
}

Generally you should not use difference of integers to decide output of compareTo method as result of **integer subtraction can overflow** but if you are sure that both operands are positive then its one of the quickest way to compare two objects. See my post [things to remember while overriding compareTo in Java](http://javarevisited.blogspot.sg/2011/11/how-to-override-compareto-method-in.html) for more tips on compareTo.

And for sorting based on person name we can implement compare(Object o1, Object o2) method of Java Comparator class.

/\*\*  
 \* Comparator implementation which sorts Person objects on person\_id field  
 \*/  
**public** **class** SortByPerson\_ID **implements** **Comparator**{  
  
    **public** **int** compare(Object o1, Object o2) {  
        Person p1 = (Person) o;  
        Person p2 = (Person) o;   
        **return** p1.getPersonId() - p2.getPersonId();  
    }  
}

Similar guidelines applies while implementing compare() method as well and instead of using subtraction operator, its better to use logical operator to compare whether two integers are equal to, less than or greater than. You can write several types of Java Comparator based upon your need for example  reverseComparator , ANDComparator , ORComparator etc which will return negative or positive number based upon logical results. [String in Java](http://javarevisited.blogspot.sg/2011/07/string-vs-stringbuffer-vs-stringbuilder.html) even provides an special comparator called CASE\_INSENSITIVE\_ORDER, to perform case insensitive comparison of String objects.

Read more: <http://javarevisited.blogspot.com/2011/06/comparator-and-comparable-in-java.html#ixzz2y8RJmE5P>

# [fail-safe vs fail-fast Iterator in Java](http://javarevisited.blogspot.sg/2012/02/fail-safe-vs-fail-fast-iterator-in-java.html" \o "fail-safe vs fail-fast Iterator in Java)

**Difference between fail-safe and fail-fast Iterator** is becoming [favorite core java interview questions](http://javarevisited.blogspot.com/2011/04/top-20-core-java-interview-questions.html) day by day, reason

it touches concurrency a bit and interviewee can go deep on it to ask *how fail-safe or fail-fast behavior is implemented*.

In this article we will see **what is fail-safe and fail fast iterators in java** and differences between fail-fast and fail-safe iterators . Concept of fail-safe iterator are relatively new in Java and first introduced with [Concurrent Collections in Java](http://javarevisited.blogspot.com/2011/04/difference-between-concurrenthashmap.html) 5 like ConcurrentHashMap and CopyOnWriteArrayList.

Read more: <http://javarevisited.blogspot.com/2012/02/fail-safe-vs-fail-fast-iterator-in-java.html#ixzz2y8n4256F>

## fail-safe Iterator in java

Contrary to fail-fast Iterator, **fail-safe iterator** doesn't throw any Exception if Collection is modified structurally

while one thread is Iterating over it because they work on clone of Collection instead of original collection and that’s why they are called as fail-safe iterator. Iterator of CopyOnWriteArrayList is an example of fail-safe Iterator also iterator written by ConcurrentHashMap keySet is also fail-safe iterator and never throw ConcurrentModificationException in Java.

Read more: <http://javarevisited.blogspot.com/2012/02/fail-safe-vs-fail-fast-iterator-in-java.html#ixzz2y8nt5Vrm>

**How do I use locales and resource bundles to internationalize my application?**

**Use java.util.Locale and ResourceBundle**

<http://www.avajava.com/tutorials/lessons/how-do-i-use-locales-and-resource-bundles-to-internationalize-my-application.html>

### [InternationalizationTest.java](http://www.avajava.com/tutorials/general-java/how-do-i-use-locales-and-resource-bundles-to-internationalize-my-application/InternationalizationTest.java)

package test;

import java.util.Locale;

import java.util.ResourceBundle;

public class InternationalizationTest {

public static void main(String[] args) throws Exception {

ResourceBundle bundle1 = ResourceBundle.getBundle("TestBundle");

displayValues(bundle1);

Locale defaultLocale = Locale.getDefault();

ResourceBundle bundle2 = ResourceBundle.getBundle("TestBundle", defaultLocale);

displayValues(bundle2);

Locale swedishLocale = new Locale("sv", "SE");

ResourceBundle bundle3 = ResourceBundle.getBundle("TestBundle", swedishLocale);

displayValues(bundle3);

Locale nonexistentLocale = new Locale("xx", "XX");

ResourceBundle bundle4 = ResourceBundle.getBundle("TestBundle", nonexistentLocale);

displayValues(bundle4);

}

public static void displayValues(ResourceBundle bundle) {

System.out.println("hello message:" + bundle.getString("my.hello"));

System.out.println("goodbye message:" + bundle.getString("my.goodbye"));

System.out.println("question message:" + bundle.getString("my.question"));

System.out.println();

}

}

# [Difference between Java Enumeration and Iterator](http://stackoverflow.com/questions/948194/difference-between-java-enumeration-and-iterator)

Looking at the Java API Specification for the [Iterator](http://java.sun.com/javase/6/docs/api/java/util/Iterator.html) interface, there is an explanation of the differences between [Enumeration](http://java.sun.com/javase/6/docs/api/java/util/Enumeration.html):

Iterators differ from enumerations in two ways:

* Iterators allow the caller to remove elements from the underlying collection during the iteration with well-defined semantics.
* Method names have been improved.

The bottom line is, both Enumeration and Iterator will give successive elements, but Iterator is improved in such a way so the method names are shorter, and has an additional remove method. Here is a side-by-side comparison:

Enumeration Iterator

---------------- ----------------

hasMoreElement() hasNext()

nextElement() next()

N/A remove()

As also mentioned in the Java API Specifications, for newer programs, Iterator should be preferred over Enumeration, as "Iterator takes the place of Enumeration in the Java collections framework." (From the [Iterator](http://java.sun.com/javase/6/docs/api/java/util/Iterator.html) specifications.)

# [what is the difference between a portlet and a servlet?](http://stackoverflow.com/questions/1480528/what-is-the-difference-between-a-portlet-and-a-servlet)

Portlets are part of JSR-168 standard that regulates portal containers and components. This is different standard from standards for web containers (and servlets). Though there are definitely strong parallels between these two standards they differ in containers, APIs, life cycle, configuration, deployment, etc.

**The main difference between portlet vs. servlet could be that while servlet always responds to single type of action - request,** **portlet (due to nature of its life cycle and stronger container bindings) has to respond to two types of actions**: render and request. There are of course more to it but I found this as the core difference between the two when I studied portal development.

Source: <http://fanatech.wordpress.com/servlets-vs-portlets/>

**Similarities**

Servlets and Portlets are web based components which use Java for their implementation.

Portlets are managed by a portlet container just like servlet is managed by servlet container.

Both static and dynamic content can be generated by Portlets and Servlets.

The life cycle of portlets and servlets is controlled by the container

The client/server model is used for both servlets and portlets

The packaging and deployment are essentially the same, WAR/EARs.

**.**

**Dissimilarities**

Servlets can render complete web pages, whereas portlets renders html fragments. These fragments are aggregated by the portal into a complete web page.

The content type of JSR 168 portlets can be only cHTML, XHTML, WML. It does not support other content types.

Portlets are not allowed to generate HTML code that contains tags such as body, frame, frameset, head, html, or title.

A Portlet unlike a servlet doesn’t have URL attached to it so it cannot be accessed directly. Access is only through the portal page which holds the portlet.

Portlets can be provided with controls to manipulate its window states or portlet modes.

Multiple instances of a single portlet can be placed onto the same page.

Portlets support persistent configuration and customization, profile information.

**Portlets can have two types of request viz. render request and action request.**

Portlets have two scopes within session; application scope for communication across portlets and portlet scope for intra portlet communication.

Portlet cannot set the character set encoding of the response nor can it set the HTTP response headers.

Portlets doesn’t have access to request URL. So it cannot access the query parameters appended to the URL. Portlets cannot set cookies.

Typical methods of Portlet API are doView(), doEdit(), doHelp() and processAction() while those of servlet are doService(), doPost(), doGet().

Difference between Runnable and Callable interface in Java?

Runnable and Callable interface both are designed to represent task, which can be executed by any thread. Both does same task for the programmer with few difference between each other. In this tutorial we will see about difference between Runnable and Callable interface difference and when we need to use Runnable and Callable interface in our application.

• Runnable interface introduced in JDK 1.0, whereas Callable interface introduced in Java 5 release along with other major changes e.g. Generics, Enum, Static imports and variable argument method.

• Since both are interface when we implement these interface we need to implement run() method from Runnable interface and call() method from Callable interface.

• **run() method didn't not return any value, whereas call() method returns Object where Callable interface is a generic parameterized interface and Type of value is provided at implementation.**

• **Callable interface can throw checked exception because it's call method throws Exception where as run() method has its limitation.**

**Basically if our application needs to return any value from executor method then we need to for Callable interface than Runnable interface.**

Bu keeping all these differences and usage between Runnalbe and Callable interface programmer need to be in a position to decide which interface he needs to choose for his application.

As this is one of the important interview question asked in most of the interviews followed by multi-threading question and mostly asked in Banking domain Java interviews.

**11) Does code form finally executes if method returns before finally block or JVM exits ?**

This Java exception interview question can also be asked in code format, where given a code with System.exit() in try block and something in finally block. It’s worth knowing that**, finally block in Java executes even when return keyword is used in try block. Only time they don’t execute is when you call JVM to exit by executing System.exit(0)f**rom try block in Java.

**12) What is difference in final, finalize and finally keyword in Java?**

Another classic interview question in core Java, this was asked to one of my friend on his telephonic interview for core Java developer with Morgan Stanley. final and finally are keyword, while finalize is method.

final keyword is very useful for creating ad [Immutable class in Java](http://javarevisited.blogspot.com/2013/03/how-to-create-immutable-class-object-java-example-tutorial.html) By making a class final, we prevent it from being extended, similarly by making a method final, we prevent it from being overridden,.

On the other hand, **finalize()** method is called  by garbage collector, before that object is collected, but this is not guaranteed by Java specification.

 finallykeyword is the only one which is related to error and exception handling and you should always have finally block in production code for closing connection and resources. See [here](http://javarevisited.blogspot.com/2012/11/difference-between-final-finally-and-finalize-java.html) for more detailed answer of this question.

Read more: <http://javarevisited.blogspot.com/2013/06/10-java-exception-and-error-interview-questions-answers-programming.html#ixzz2yKd1ndRR>

Why String is immutable or final in Java

String A = "Test"  
String B = "Test"   
  
Now String B called "Test".toUpperCase() which change the same object into "TEST" , so A will also be "TEST" which is not desirable.

2)String has been widely used as parameter for many Java classes e.g. for opening network connection, you can pass hostname and port number as string , you can pass database URL as string for opening database connection, you can [open any file in Java](http://javarevisited.blogspot.sg/2012/07/read-file-line-by-line-java-example-scanner.html) by passing name of file as argument to File I/O classes.  
  
In case, if String is not immutable, this would lead serious security threat , I mean some one can access to any file for which he has authorization, and then can change the file name either deliberately or accidentally and gain access of those file. Because of immutability, you don't need to worry about those kind of threats. This reason also gel with, **Why String is final in Java**, by making java.lang.String final, Java designer ensured that no one overrides any behavior of String class

Examples:

Consider a scenario, in a banking application for money transfer - the beneficiary account number is defined in a string as "0789567345". If by mistake/intentionally this acc. number is changed, money will go to a wrong account.

Another scenario - if someone change the class name anywhere between processing as ..

getClass().getName().subString(0, 5);

The Class loader will simply say 'Class Not Found

[What is the difference between an instance and a class (static) variable in Java](http://stackoverflow.com/questions/15486392/what-is-the-difference-between-an-instance-and-a-class-static-variable-in-java)

A static variable is shared by all instances of the class. and in case of instance variable each instance of class have different copy.

Static variable memory allocate at compile time, They are loaded at load time and initialized at class initialization time and in case of instance variable everything is done at run time.

You can understand by example.

## Example:

An instance variable is one per Object, every object has its own copy of instance variable.

public class Test{

int x = 5;

}

Test t1 = new Test();

Test t2 = new Test();

Both t1 and t2 will have its own copy of x.

A static variable is one per Class, every object of that class shares the same Static variable.

public class Test{

public static int x = 5;

}

Test t1 = new Test();

Test t2 = new Test();

Both t1 and t2 will have the exactly one x to share between them.

## What is Blocking methods in Java

As I said **Blocking methods** are those which blocks the current executing thread from further operation until function returns. So if you have just one thread in your program e.g. [main thread](http://javarevisited.blogspot.com/2011/12/main-public-static-java-void-method-why.html) and you call any blocking method e.g. reading from InputStream, your program will be blocked until reading of file finished. Javadoc clearly mention whether an API call is blocking or not but **most of  java IO methods are blocking**. If you are doing GUI programming in Java using Swing than knowledge of blocking methods becomes even more important for you, because no body likes freezing or non responsive GUI. methods like [invokeAndWait](http://javarevisited.blogspot.com/2011/09/invokeandwait-invokelater-swing-example.html) are blocking in nature and should be used only when you are performing some operation on which user should wait for result. In most simple terms *blocking means your code in next line will not be executed* because Thread which is executing blocking function is waiting for method to return. here is a code example which help you

to understand blocking calls:

public class BlcokingCallTest {

    public static void main(String args[]) throws FileNotFoundException, IOException  {

      System.out.println("Calling blocking method in Java");

      int input = System.in.read();

      System.out.println("Blocking method is finished");

    }

}

*In this code example after executing first print statement your program will be blocked and will not execute second print statement until you enter some characters in console and press enter because* ***read() method blocks*** *until some input is available for reading.*

## Examples of blocking methods in Java:

There are lots of blocking methods in Java API and good thing is that javadoc clearly informs about it and always mention whether a method call is blocking or not. In General methods related to [reading or writing file](http://javarevisited.blogspot.com/2011/12/read-and-write-text-file-java.html), opening network connection, reading from Socket, updating GUI synchronously uses blocking call. here are some of most common methods in Java which are blocking in nature:

1) **InputStream.read()** which blocks until input data is available, an exception is thrown or end of Stream is detected.

2) **ServerSocket.accept()** which listens for incoming socket connection in Java and blocks until a connection is made.

3) **InvokeAndWait()** wait until code is executed from [Event Dispatcher thread](http://javarevisited.blogspot.com/2011/09/swing-interview-questions-answers-in.html).

Read more: <http://javarevisited.blogspot.com/2012/02/what-is-blocking-methods-in-java-and.html#ixzz2yPtpGEjW>

<http://www.yourkit.com/docs/kb/sizes.jsp>

*as experience has shown, sometimes a sort of uncertainty may arise on the subject of Java Virtual Machine (JVM) memory structure and other related aspects such as sizes of various kinds of memory, live and dead objects, etc.*

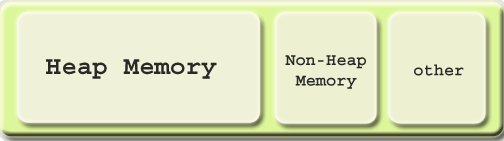
*In this article, we shall try to illuminate these issues to clear up the point*

*.*

#### Heap and Non-Heap Memory

The JVM memory consists of the following segments:

* Heap Memory, which is the storage for Java objects
* Non-Heap Memory, which is used by Java to store loaded classes and other meta-data
* JVM code itself, JVM internal structures, loaded profiler agent code and data, etc.



**Heap**

The JVM has a *heap* that is the runtime data area from which memory for all class instances and arrays are allocated. It is created at the JVM start-up.

The heap size may be configured with the following VM options:

* -Xmx<size> - to set the maximum Java heap size
* -Xms<size> - to set the initial Java heap size

Example:

It is possible to increase heap size allocated by the JVM by using command line options Here we have 3 options

-Xms<size> set initial Java heap size

-Xmx<size> set maximum Java heap size

-Xss<size> set java thread stack size

java -Xms16m -Xmx64m ClassName

In the above line we can set minimum heap to 16mb and maximum heap 64mb

By default, the maximum heap size is 64 Mb.

Heap memory for objects is reclaimed by an automatic memory management system which is known as a garbage collector. The heap may be of a fixed size or may be expanded and shrunk, depending on the garbage collector's strategy.

**Non-Heap**

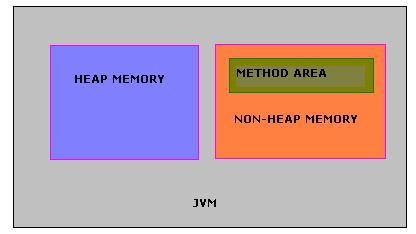
Also, the JVM has memory other than the heap, referred to as *non-heap memory*. It is created at the JVM startup and stores per-class structures such as runtime constant pool, field and method data, and the code for methods and constructors, as well as interned Strings.

Unfortunately, **the only information JVM provides on non-heap memory is its overall size.** No detailed information on non-heap memory content is available.

The abnormal growth of non-heap memory size may indicate a potential problem, in this case you may check up the following:

* If there are class loading issues such as leaked loaders. In this case, the problem may be solved with the help of [Class loaders](http://www.yourkit.com/docs/java/help/class_loaders.jsp) view.
* If there are strings being massively interned. For detection of such problem, [Object allocation recording](http://www.yourkit.com/docs/java/help/allocations.jsp) may be used.

A run time constant pool is a per-class or per-interface run time representation of the constant\_pool table in a class file. Each runtime constant pool is allocated from the Java virtual machine’s method area.

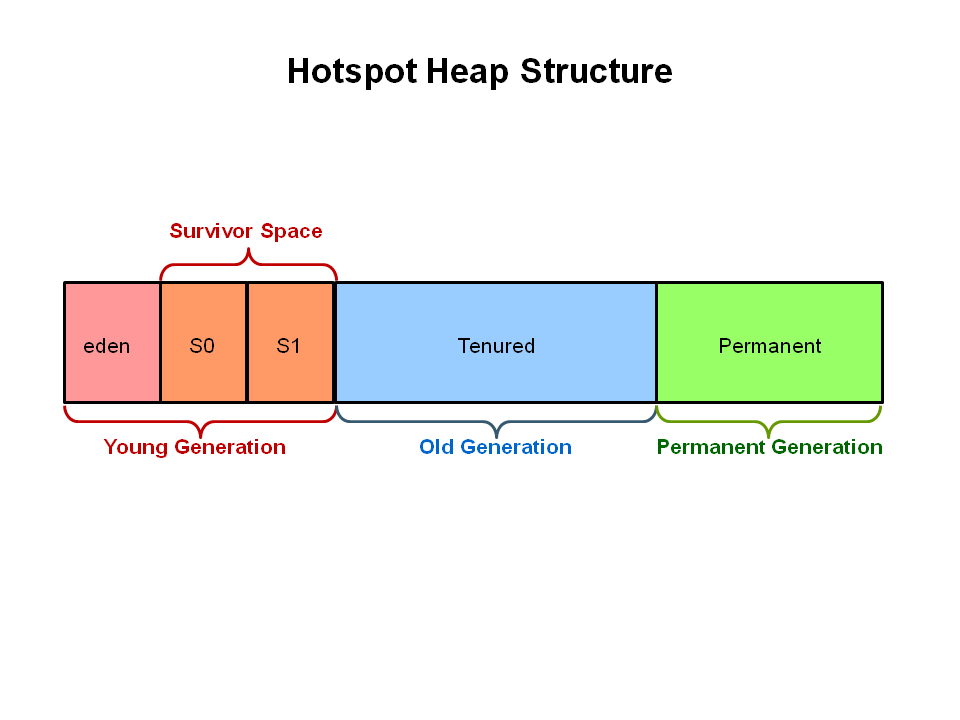
tual machine’s method area.

**Java Stacks or Frames**

Java stacks are created private to a thread. Every thread will have a program counter (PC) and a java stack. PC will use the java stack to store the intermediate values, dynamic linking, return values for methods and dispatch exceptions. This is used in the place of registers.

#### JVM Generations

The information learned from the object allocation behavior can be used to enhance the performance of the JVM. Therefore, the heap is broken up into smaller parts or generations. The heap parts are: Young Generation, Old or Tenured Generation, and Permanent Generation



The **Young Generation** is where all new objects are allocated and aged. When the young generation fills up, this causes a **minor garbage collection**. Minor collections can be optimized assuming a high object mortality rate. A young generation full of dead objects is collected very quickly. Some surviving objects are aged and eventually move to the old generation.

**Stop the World Event** - All minor garbage collections are "Stop the World" events. This means that all application threads are stopped until the operation completes. Minor garbage collections are always Stop the World events.

The **Old Generation** is used to store long surviving objects. Typically, a threshold is set for young generation object and when that age is met, the object gets moved to the old generation. Eventually the old generation needs to be collected. This event is called a **major garbage collection**.

Major garbage collection are also Stop the World events. Often a major collection is much slower because it involves all live objects. So for Responsive applications, major garbage collections should be minimized. Also note, that the length of the Stop the World event for a major garbage collection is affected by the kind of garbage collector that is used for the old generation space.

The **Permanent generation** contains metadata required by the JVM to describe the classes and methods used in the application. The permanent generation is populated by the JVM at runtime based on classes in use by the application. In addition, Java SE library classes and methods may be stored here.

Classes may get collected (unloaded) if the JVM finds they are no longer needed and space may be needed for other classes. The permanent generation is included in a full garbage collection

Reflection in java

<http://www.programmerinterview.com/index.php/java-questions/java-reflection-example/\\>

Similarly, Reflection in Java is the ability to examine and/or modify the properties or behavior of an ***object*** at run-time. It’s important to note that reflection specifically applies to ***objects*** – so we need an object of a class to get information for that particular class.

Reflection in Java consists of 2 primary things that you should remember:

1. Metadata. Metadata literally means data about the data. In this case, metadata means extra data that has to do with your Java program – like data about your Java classes, constructors, methods, fields, etc.

2. Functionality that allows you to *manipulate* the metadata as well. So, functionality that would allow you to manipulate those fields, methods, constructors, etc. You can actually call methods and constructors using Java reflection – which is an important fact to remember.

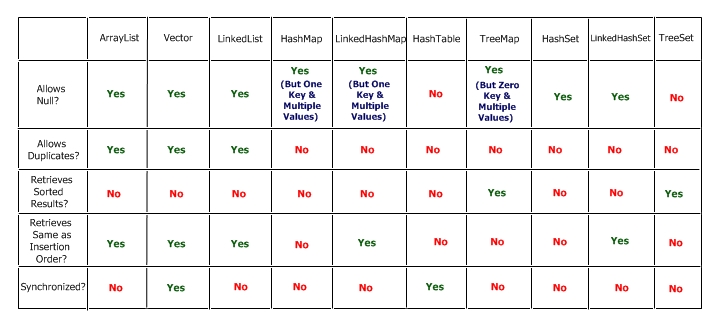
Here are the few other Checked Exceptions -

* SQLException
* IOException
* DataAccessException
* ClassNotFoundException
* InvocationTargetException

unchecked exceptions are not checked at compile-time, they are being checked at runtime. Lets see another example.

Here are the few most frequently seen unchecked exceptions -

* NullPointerException
* ArrayIndexOutOfBoundsException
* ArithmeticException
* IllegalArgumentException

**Comparison Chart between different Collection**  
   


═══════════════╦═══════════════════╦═══════════════════╦═════════════════════╗

**Popular implementation**

List - ArrayList, LinkedList and Vector

Set - HashSet, TreeSet and LinkedHashSet

Map - HashMap, Hashtable and TreeMap

**When to use List, Set and Map in Java**

Based upon our understanding of *difference between Set, List and Map* we can now decide when to use List, Set or Map in Java.

1) If you need to access elements frequently by using **index**, than List is a way to go. Its implementation e.g. [ArrayList](http://javarevisited.blogspot.com/2011/05/example-of-arraylist-in-java-tutorial.html) provides faster access if you know index.

2) If you want to store elements and want them to maintain an **order** on which they are inserted into collection then go for List again, as[List](http://javarevisited.blogspot.com/2012/03/how-to-loop-arraylist-in-java-code.html) is an ordered collection and maintain insertion order.

1. If you want to create collection of unique elements and **don't want any duplicate** than choose any Set implementation e.g. [HashSet](http://javarevisited.blogspot.com/2012/06/hashset-in-java-10-examples-programs.html),LinkedHashSet or TreeSet.

All Set implementation follow there general contract e.g. uniqueness but also add addition feature e.g. TreeSet is a SortedSet and elements stored on TreeSet can be sorted by using [Comparator or Comparable in Java](http://javarevisited.blogspot.com/2011/06/comparator-and-comparable-in-java.html).

1. LinkedHashSet also maintains insertion order.

# Reference Types In Java

<http://java.dzone.com/articles/reference-types-java-part-1>

Lately, I have been learning a thing or two about the JVM internals. And one of the most interesting things that I came to know about was, the existence of different types of references in Java.   
there are actually 4 kinds of reference types in Java. 

1. *Strong references.*
2. *Soft references.*
3. *Weak references.*
4. *Phantom references.*

If you want to make a copy of

int[] a = {1,2,3,4,5};

this is the way to go

int[] b = Arrays.copyOf(a, a.length);

Arrays.copy may be faster than a.clone() on small arrays. Both copy elements equally fast but clone() returns Object so compiler has to insert an implicit cast, you can see it in the bytecode, something like this

ALOAD 1

INVOKEVIRTUAL [I.clone ()Ljava/lang/Object;

CHECKCAST [I

ASTORE 2

**Example of volatile keyword in Java:**

To Understand example of volatile keyword in java let’s go back to [Singleton pattern in Java](http://javarevisited.blogspot.com/2011/03/10-interview-questions-on-singleton.html) and see [double checked locking in Singleton](http://javarevisited.blogspot.gr/2012/07/why-enum-singleton-are-better-in-java.html) with Volatile and without volatile keyword in java.

/\*\*  
 \* Java program to demonstrate **where to use Volatile keyword in Java**.  
 \* In this example Singleton Instance is declared as volatile variable to ensure  
 \* every thread see updated value for \_instance.  
 \*   
 \* @author Javin Paul  
 \*/  
**public** **class** Singleton{  
**private** **static** **volatile** Singleton \_instance; *//volatile variable*  
  
**public** **static** Singleton getInstance(){  
  
   if(\_instance == **null**){  
            **synchronized**(Singleton.**class**){  
              if(\_instance == **null**)  
              \_instance = **new** Singleton();  
            }  
  
   }  
   **return** \_instance;  
  
}

**Marker interface in Java** is interfaces with no field or methods or in simple word **empty interface in java is called marker interface**. Example of market interface is Serializable, Clonnable and Remote interface. Now if marker interface doesn't have any field or method or behavior they why would Java needs it?  
  
Read more: <http://javarevisited.blogspot.com/2012/01/what-is-marker-interfaces-in-java-and.html#ixzz30Pnls0Dp>

In summary **marker interface in Java is used to indicate something to compiler, JVM** or any other tool but **Annotation** is better way of doing same thing.  
  
Read more: <http://javarevisited.blogspot.com/2012/01/what-is-marker-interfaces-in-java-and.html#ixzz30Pny9Gpv>

**What is CyclicBarrier in Java**

CyclicBarrier in Java is a synchronizer introduced in JDK 5 on java.util.Concurrent package along with other concurrent utility like [Counting Semaphore](http://javarevisited.blogspot.sg/2012/05/counting-semaphore-example-in-java-5.html), [BlockingQueue](http://javarevisited.blogspot.sg/2012/02/producer-consumer-design-pattern-with.html), [ConcurrentHashMap](http://javarevisited.blogspot.sg/2011/04/difference-between-concurrenthashmap.html) etc. CyclicBarrier is similar to CountDownLatch which we have seen in last article  [What is CountDownLatch in Java](http://javarevisited.blogspot.sg/2012/07/countdownlatch-example-in-java.html) and allows multiple threads to wait for each other (barrier) before proceeding. Difference between CountDownLatch and CyclicBarrier is a also very [popular multi-threading interview question](http://javarevisited.blogspot.sg/2011/07/java-multi-threading-interview.html) in Java. CyclicBarrier is a natural requirement for concurrent program because it can be used to perform final part of task once individual tasks  are completed. All threads which [wait](http://javarevisited.blogspot.sg/2011/05/wait-notify-and-notifyall-in-java.html) for each other to reach barrier are called parties, CyclicBarrier is initialized with number of parties to be wait and threads wait for each other by calling CyclicBarrier.await() method which is a [blocking method in Java](http://javarevisited.blogspot.sg/2012/02/what-is-blocking-methods-in-java-and.html) and  blocks until all Thread or parties call await(). In general calling await() is shout out that Thread is waiting on barrier. await() is a blocking call but can be timed out or Interrupted by other thread. In this Java concurrency tutorial we will see *What is CyclicBarrier in Java*  and  an example of CyclicBarrier on which three Threads will wait for each other before proceeding further.

Read more: <http://javarevisited.blogspot.com/2012/07/cyclicbarrier-example-java-5-concurrency-tutorial.html#ixzz30PoXJSZc>

**3. What is the difference between creating String as new() and literal?**

When we create string with new() Operator, it’s created in heap and not added into string pool while String created using literal are created in String pool itself which exists in PermGen area of heap.

String s = new String("Test");  
   
does not  put the object in String pool , we need to call String.intern() method which is used to put  them into String pool explicitly. its only when you create String object as String literal e.g. String s = "Test" Java automatically put that into String pool.

System.out.println(" Please enter the input string :" );

Scanner in = **new** Scanner (System.in);

String s=in.nextLine();

**13.** **When do you override hashcode and equals() ?**  
Whenever necessary especially if you want to do equality check or want to use your object as key in HashMap.

# [Java Class that implements Map and keeps insertion order?](http://stackoverflow.com/questions/683518/java-class-that-implements-map-and-keeps-insertion-order)

LinkedHashMap

class OuterClass {

...

static class StaticNestedClass {

...

}

class InnerClass {

...

}

}

A nested class is a member of its enclosing class. Non-static nested classes (inner classes) have access to other members of the enclosing class, even if they are declared private. Static nested classes do not have access to other members of the enclosing class. As a member of the OuterClass, a nested class can be declared private, public, protected, or *package private*. (Recall that outer classes can only be declared public or *package private*.)

Static nested classes are accessed using the enclosing class name:

OuterClass.StaticNestedClass

For example, to create an object for the static nested class, use this syntax:

OuterClass.StaticNestedClass nestedObject =

new OuterClass.StaticNestedClass();

# is overloading example of static or dynamic polymorphism

Polymorphsim is of two types: static binding(compile time polymorphism) and dynamic binding(run time polymorphism

**VIMP:**compile time [polymorphism](http://www.javaranch.com/campfire/StoryPoly.jsp) works on the **ref type not Object Type(instance)** so compiler is trying to find and match the method with name method(with String type) only but it is defined in class Example.

**Polymorphism**

Method overloading : (static polymorphism –compile time)

Method overriding : (dynamic polymorphism –run time)

Method overloading : (static polymorphism –compile time)

**Method overloading** is defining several methods in the same class, that accept different numbers and types of parameters. In this case, the actual method called is decided at compile-time, based on the number and types of arguments. For instance, the method System.out.println() is overloaded, so that you can pass ints as well as Strings, and it will call a different version of the method.

Method overriding : (dynamic polymorphism –run time)

For example, the standard Java class java.util.LinkedHashSet extends java.util.HashSet. The method add() is overridden in LinkedHashSet. If you have a variable that is of type HashSet, and you call its add() method, it will call the appropriate implementation of add(), based on whether it is a HashSet or a LinkedHashSet. This is called polymorphism.

<http://javarevisited.blogspot.com/2012/03/what-is-static-and-dynamic-binding-in.html>

The term "static polymorphism" **has been used to refer to overloading.** "static" here means that the decision about which method to call is made when the program is compiled. This is in contrast to "dynamic polymorphism", what you get from **overriding methods** in Java, in which that decision is deferred until runtime.

generally our understanding is that in overloading , which method is to be called is decided during compile time... but we are passing the values to these overloaded methods during runtime. depending on the values that we give at runtime the proper method will be invoked...

**package** test.main;

**public** **class** TestOverloading {

**void** method(**int** n) {

System.*out*.println("Number: " + n);

}

}

**class** ExampleWithString **extends** TestOverloading {

**void** method(String s) {

System.*out*.println("Text: " + s);

}

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

TestOverloading e = **new** ExampleWithString();

e.method(23);

e.method("Hello"); // Error. Even though instance has method.

}

}

Notice that at line 16, it is a compile error, even though the actual instance has the method that takes a string. The reason is that the signature checking is only done at compile time.

Object is checked at run time and class is compile time.

e is the instance of **ExampleWithString**,but compile time [polymorphism](http://www.javaranch.com/campfire/StoryPoly.jsp) works on the **ref type not Object Type(instance)** so compiler is trying to find and match the method with name method(with String type) only but it is defined in class Example.

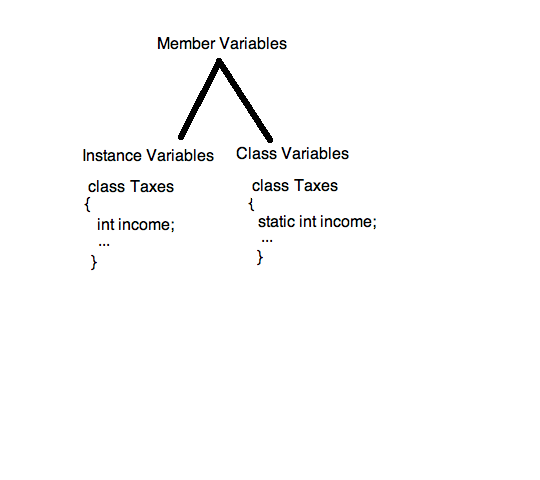
So compiler gives the compile time method no matching method found.

## Difference between class and instance variables

Now, it should be clear what the difference between instance and class variables is. Class variables only have one copy that is shared by all the different objects of a class, whereas every object has it’s own personal copy of an instance variable. So, instance variables across different objects can have different values whereas class variables across different objects can have only one value.

## Class and Instance variables are both Member variables

Here’s a little diagram to help you remember the differences between instance and class variables:



Looking at the Java API Specification for the [Iterator](http://java.sun.com/javase/6/docs/api/java/util/Iterator.html) interface, there is an explanation of the differences between [Enumeration](http://java.sun.com/javase/6/docs/api/java/util/Enumeration.html):

Iterators differ from enumerations in two ways:

* Iterators allow the caller to remove elements from the underlying collection during the iteration with well-defined semantics.
* Method names have been improved.

The bottom line is, both Enumeration and Iterator will give successive elements, but Iterator is improved in such a way so the method names are shorter, and has an additional remove method. Here is a side-by-side comparison:

Enumeration Iterator

---------------- ----------------

hasMoreElement() hasNext()

nextElement() next()

N/A remove()

As also mentioned in the Java API Specifications, for newer programs, Iterator should be preferred over Enumeration, as "Iterator takes the place of Enumeration in the Java collections framework." (From the [Iterator](http://java.sun.com/javase/6/docs/api/java/util/Iterator.html) specifications.)

**How will the below literal value be internally represented?  
float f = 21.22;**

It will be represented as a double value. Floating point literals are always double by default. If you want a float, you must append an F or f to the literal.

**Give your observation on the below statement.  
double d = 10.12/0;**

This will compile and execute fine. The result will be Infinity

**Can a final variable be declared inside a method?**

No. Local variables cannot be declared as final. YES-NAGa but no use.

**I don't want my class to be inherited by any other class. What should i do?**

You should declared your class as final. A class declared as final can't be inherited by any other class.

**107.When will you declare a class as final?**

When a class is independent and completely concrete in nature, then the class has to be marked as final.

**Can you give few examples of final classes defined in Java API?**

java.lang.String,java.lang.Math are final classes.

**How to define a constant variable in Java?**

The variable should be declared as static and final. So only one copy of the variable exists for all instances of the class and the value can't be changed also.static final int PI = 3.14; is an example for constant.

**When will you define a method as static?**

When a method needs to be accessed even before the creation of the object of the class then we should declare the method as static.

**I want to print "Hello" even before main is executed. How will you acheive that?**

Print the statement inside a static block of code. Static blocks get executed when the class gets loaded into the memory and even before the creation of an object. Hence it will be executed before the main method.

**Cant you use the constructor for initialisation rather than static block?**

Constructors are used for object level initialisation whereas the static block are used for class level initialisation ie to initialise constants.

**When overriding a static method, can it be converted to a non-static method?**

No. It should be static only.

**Can we declare a static variable inside a method?**

Static varaibles are class level variables and they can't be declared inside a method. If declared, the class will not compile.

**Can a abstract class be defined without any abstract methods?**

Yes it's possible. This is basically to avoid instance creation of the class.

**What happens if a class has implemented an interface but has not provided implementation for a method in a interface?**

Its the same as the earlier answer. The class has to be marked as abstract. This will be enforced by the compiler.

**an a class be marked as native?**

No. Only methods can be marked as native.

**130.What is the use of native methods?**

When a java method accesses native library written in some other programming language then the method has to be marked as native.

**131.What is the disadvantage of native methods?**

By using native methods, the java program loses platform independence - the accessed platform might be tightly coupled with a operating system hence java program also loses OS independence.

**132.What is the purpose of transient modifier?**

Only variables can be marked as transient. Variables marked as transient will not be persisted during object persistence.

**133.What is the purpose of volatile modifier?**

Only variables can be marked as volatile. Volatile variables might be modified asynchronously.

**What modifiers are allowed for methods in an Interface?**

Only public and abstract modifiers are allowed for methods in interfaces.

Can abstract class have constructor in Java? (Yes, detailed answer is [here](http://java67.blogspot.sg/2013/02/can-abstract-class-have-constructor-in-java.html))

Difference between wait and sleep in Java?(wait release lock, sleep keep it, for details see [here](http://javarevisited.blogspot.sg/2011/12/difference-between-wait-sleep-yield.html))

Difference between checked and unchecked exception in Java? (former is checked by compiler and it's handling is enforced by mandating try-catch or try-finally block. Later is not checked by compiler but can be caught using try-catch or try-finally block. For example, java.io.IOException, java.sql.SQLException are checked exception, while java.lang.NullPointerException and java.lang.ArrayIndexOutOfBoundsException are example of unchecked exception in Java, for better answer see [here](http://javarevisited.blogspot.sg/2011/12/checked-vs-unchecked-exception-in-java.html))  
  
Read more: <http://javarevisited.blogspot.com/2014/02/top-30-java-phone-interview-questions.html#ixzz32ql0NUfI>

Comparator and Comparable example:

<http://www.digizol.com/2008/07/java-sorting-comparator-vs-comparable.html>

If we need to sort using other fields of the employee, we’ll have to change the Employee class’s compareTo() method to use those fields. But then we’ll loose this empId based sorting mechanism. This is not a good alternative if we need to sort using different fields at different occasions. But no need to worry; Comparator is there to save us. By writing a class that implements the java.util.Comparator interface, you can sort Employees using any field as you wish even without touching the Employee class itself; Employee class does not need to implement java.lang.Comparable or java.util.Comparator interface. - See more at: <http://www.digizol.com/2008/07/java-sorting-comparator-vs-comparable.html#sthash.9zntwwvR.dpuf>

### Difference between Abstraction and Encapsulation in Java - OOPS

**Abstraction vs Encapsulation – Java OOPS**

<http://java67.blogspot.sg/2012/08/difference-between-abstraction-and-encapsulation-java-oops.html>

Abstraction and Encapsulation in Java are two important [Object oriented programming concept](http://javarevisited.blogspot.sg/2012/03/10-object-oriented-design-principles.html) and they are completely different to each other. Only similarity between Abstraction and Encapsulation is that they are OOPS concept, other than that they mean two different things. Abstraction represent taking out the behavior from How exactly its implemented, one example of[abstraction in Java](http://javarevisited.blogspot.sg/2010/10/abstraction-in-java.html) is interface while Encapsulation means hiding details of implementation from outside world so that when things change no body gets affected. One example of [Encapsulation in Java](http://javarevisited.blogspot.sg/2012/03/what-is-encapsulation-in-java-and-oops.html) is private methods; clients don't care about it, You can change, amend or even remove that method  if that method is not encapsulated and it were public all your clients would have been affected. Apart from this main difference in behavior here are couple of more*differences between Abstraction and Encapsulation in Java*.

there are some of the main differences between Abstraction vs Encapsulation in Java and OOPS(Object Oriented programming) concept. Abstraction and Encapsulation along with [Inheritance](http://java67.blogspot.sg/2012/08/what-is-inheritance-in-java-oops-programming-example.html) and [polymorphism](http://javarevisited.blogspot.sg/2011/08/what-is-polymorphism-in-java-example.html) forms basis of Object oriented programming in Java.

1) First difference between Abstraction and Encapsulation is that, Abstraction is implemented in Java using [interface](http://javarevisited.blogspot.sg/2012/04/10-points-on-interface-in-java-with.html) and abstract class while Encapsulation is implemented using [private](http://javarevisited.blogspot.sg/2012/03/private-in-java-why-should-you-always.html), package-private and protected access modifier.

2) Encapsulation is also called data hiding.

3) Design principles "[programming for interface than implementation](http://javarevisited.blogspot.sg/2012/06/20-design-pattern-and-software-design.html)" is based on abstraction and "*encapsulate whatever changes*" is based upon Encapsulation.

That's all from my side on differences between Abstraction and Encapsulation in Java. Correct understanding of Encapsulation and Abstraction is must for any Java developer. Head first Object oriented Analysis and design is a great book to learn more about Abstraction, Encapsulation and other OOPS concept.

<http://javarevisited.blogspot.com/2011/03/10-interview-questions-on-singleton.html>

Double checked locking is a technique

public class EagerSingleton {

    private static volatile EagerSingleton instance = null;

    // private constructor

    private EagerSingleton() {

    }

    public static EagerSingleton getInstance() {

        if (instance == null) {

            synchronized (EagerSingleton.class) {

                // Double check

                if (instance == null) {

                    instance = new EagerSingleton();

                }

            }

        }

        return instance;

    }

}

;

Double checked locking should only be used when you have requirement for lazy initialization otherwise [use Enum to implement singleton](http://javarevisited.blogspot.com/2012/07/why-enum-singleton-are-better-in-java.html) or simple static final variable.

**Singleton pattern with static factory method**

This is one of my favorite method to impelemnt Singleton pattern in Java, Since Singleton instance is [static](http://javarevisited.blogspot.sg/2011/11/static-keyword-method-variable-java.html) and [final variable](http://javarevisited.blogspot.sg/2011/12/final-variable-method-class-java.html) it initialized when class is first loaded into memeory so creation of instance is inherently thread-safe.

***/\*\*  
\* Singleton pattern example with static factory method  
\*/***  
  
**public** **class** Singleton{  
    *//initailzed during class loading*  
    **private** **static** **final** Singleton INSTANCE = **new** Singleton();  
    
    *//to prevent creating another instance of Singleton*  
    **private** Singleton(){}  
  
    **public** **static** Singleton getSingleton(){  
        **return** INSTANCE;  
    }  
}

You can call Singleton.getSingleton() to get access of this class.

Read more: <http://javarevisited.blogspot.com/2012/07/why-enum-singleton-are-better-in-java.html#ixzz36WmgBsG5>

}  
  
Read more: <http://javarevisited.blogspot.com/2011/03/10-interview-questions-on-singleton.html#ixzz36WlZP5HM>

**2) Enum Singletons handled Serialization by themselves**

Another problem with conventional Singletons are that once you implement [serializable interface](http://javarevisited.blogspot.sg/2011/04/top-10-java-serialization-interview.html) they are no longer remain Singleton because readObject() method always return a new instance just like constructor in Java. you can avoid that by using readResolve() method and discarding newly created instance by replacing with Singeton as shwon in below example :

*//readResolve to prevent another instance of Singleton*  
    **private** Object readResolve(){  
        **return** INSTANCE;  
    }

This can become even more complex if your Singleton Class maintain state, as you need to make them [transient](http://javarevisited.blogspot.sg/2012/03/difference-between-transient-and.html), but witn **Enum Singleton**, Serialization is guarnateed by JVM.

**3) Creation of Enum instance is thread-safe**

As stated in point 1 since creatino of Enum instance is thread-safe by default you don't need to worry about double checked locking.

In summary, given the **Serialzation and thraead-safety guaranteed** and with couple of line of code enum Singleton pattern is best way to create Singleton in Java 5 world. you can still use other popular methods if you feel so but I still have to find a *convincing reason not to use Enum as Singleto*n, let me know if you got any.

Read more: <http://javarevisited.blogspot.com/2012/07/why-enum-singleton-are-better-in-java.html#ixzz36WoI5XIv>

Example:

### Using Enum

This type of implementation recommend the use of enum. [Enum](http://docs.oracle.com/javase/tutorial/java/javaOO/enum.html" \o "enum in java), as written in java docs, provide implicit support for thread safety and only one instance is guaranteed. This is also a good way to have singleton with minimum effort.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public enum EnumSingleton {      INSTANCE;      public void someMethod(String param) {          // some class member      }  } |

the truth is that object oriented programming often creates confusion by creating a disconnect between the philosophical side of development and the actual mechanical workings of the computer. I'll try to contrast the two for you:

The basic concept of OOP is this: Class >> Object >> Instance.

The class = the blue print. The Object is an actual thing that is built based on the 'blue print' (like the house). An instance is a virtual copy (but not a real copy) of the object.

The more technical explanation of an 'instance' is that it is a 'memory reference' or a reference variable. This means that an 'instance' is a variable in memory that only has a memory address of an object in it. The object it addresses is the same object the instance is said to be 'an instance of'. If you have many instances of an object, you really just have many variables in difference places in your memory that all have the same exact memory address in it - which are all the address of the same exact object. You can't ever 'change' an instance, although it looks like you can in your code. What you really do when you 'change' an instance is you change the original object directly. Electronically, the processor goes through one extra place in memory (the reference variable/instance) before it changes the data of the original object.

The process is: processor >> memory location of instance >> memory location of original object.