1. [Data Types in java(there are 8)](#DataTypesinjava)
2. [About Class and Instance](#About_Class_instance)
3. [Class loader](#classLoader)
4. [Ways to create Object in Java](#ways_create_object_java)
5. [Methods In Object class](#methods_in_java)
6. [Conditional Operators](#Conditional_operators)
7. [Bitwise Operators](#Bitwise_operators)
8. [Overloading\_static\_dyanmic](#Overloading_static_dyanmic) (Method overloading : (static polymorphism –compile time)
9. [JRE](#JRE)
10. [Why String is Immutable](#why_string_immutable)
11. Ways to create String in Java
12. [Diff String Literal and new String](#String_liternal_newString)
13. [What is String Pool](#StringPool)
14. [How to create immutable class](#how_to_immutable)
15. [Diff between instance & class variables](#diff_instance_class_variables)
16. [Java I/O diagram](#JAVA_io)
17. [InputStream Vs InstreamReader](#InputStream_InputStreamReader)
18. [Marker Interface](#Marker_interface)
19. [Difference between Serializable and Externalizable](#Diff_Serializable_Externalizable)
20. t[ransient keyword](#Transient)
21. [Callback\_interface](#Callback_interface)
22. [How HashMap works in java](#HashMap)
23. [HOW ConcurrentHashMap works](#concurrenthashmap_working)
24. [Diff\_subSequence and Substring](#diff_subSequence_substring)
25. [Var args](#Varargs)
26. [Autoboxing in java](#Autoboxing)
27. [useof\_nestedclasses](#useof_nestedclasses)
28. [diff\_string\_new\_and\_literal](#diff_string_new_and_literal)
29. How [substring\_working](#substring_working)
30. [Comparator\_Vs\_Comparable](#Comparator_Vs_Comparable)
31. [FailSafe\_VS\_Failfast](#FailSafe_VS_FailFast) (Contrary to fail-fast Iterator, **fail-safe iterator** doesn't throw any Exception if Collection is modified structurally
32. [Difference Enumeration & Iterator](#Diff_Enumeration_Iterator) (Iterators allow the caller to remove elements from the underlying collection during the iteration)
33. [Diff\_Runnable\_Callable](#Diff_Runnable_Callable)
34. [**Does code form finally executes if method returns before finally block or JVM exits ?**](#finally_block_ifmethod_returns)
35. [**What is difference in final, finalize and finally keyword in Java?**](#final_finalize_finally) **(**final and finally are keyword, while finalize is method)
36. [Blocking method](#Blocking_methods)(System.in.read(), waits)
37. [Heap and Non Heap Memory](#Heap_NonHeap_memory)
38. [Heap Structure- JVM generations](#Heap_structure)
39. [SHALLOW COPY VS DEEP COPY IN JAVA](#ShallowCopyVsDeepCopy)
40. [Reflection in java](#reflection)
41. [COLLECTION COMPARISION](#Collection_Comparison)
42. [When to use which collection](#When_to_use_List_Set_Map)
43. [Reference Types](#Reference_types_java)
44. [Cyclic Barrier (](#CycliciBarrier)allows multiple threads to wait for each other (barrier) before proceeding.
45. [**When do you override hashcode and equals() ?**](#When_Override_hasCode_equals)
46. Difference between equals and hashcode
47. [diff\_instance\_class\_variables](#diff_instance_class_variables)
48. Single ton
49. [Difference between object ,instance.](#Diff_instance_object)
50. [Generics](#Generics)
51. [generics\_bounderies](#generics_bounderies)
52. Bounded type generics
53. [Enum](#enum)
54. [Double\_Vs\_Bigdecimal.](#double_Vs_BigDecimal)
55. [date\_timestamp\_diff](#date_timestamp_diff)
56. [dto\_vo\_pojo](#dto_vo_pojo)
57. [Double checked locking in java](#double_checked_locking) (in singleton)
58. [Ways to compare String](#CompareString)
59. [Nested classes](#nestedclasses)
60. [How substring works in java](#substring_working)
61. [What is the difference between NoClassDefFoundError and ClassNotFoundException?](#ClassNotFoundException_NoClassDefError)
62. [Differ between equals method and equity](#diff_equals_equity)
63. [Difference between Array and ArrayList](#diff_array_arrayList)
64. [Ways to declare Array in java](#Declare_Array)
65. [Collections.binarySearch() vs. List indexOf()](http://stackoverflow.com/questions/5667427/collections-binarysearch-vs-list-indexof)
66. [Find Bugs vs PMD](#findbugs) vs CheckStyle

If your collection is sorted, binarySearch() will be O(log n) as opposed to indexOf()'s O(n) and you will definitely see an improvement.

1. [Binary Search](#binarysearch)
2. [Overried\_private\_Static](#Overried_private_Static)
3. Diff\_HashMap\_identityHashMap\_WeekHasMap
4. [Diff\_ArrayList\_LinkedList in java](#diff_Arraylist_linkedlist)
5. [Synchronized block and synchrnonized method](#synch_block_method)
6. [Serialization rules](#serialization_Rules)
7. [Bubble sort](#bubble_sort)
8. [String and Final](http://stackoverflow.com/questions/874978/string-and-final)
9. [Difference between Weak Reference and Soft Reference in Java](#weak_)
10. [Why set does not allow duplicates](#_Why_set_is)
11. [Data Structures](#datastructures)
12. [Search algorithms in Java](#searchalgo)
13. [Sorting algorithms](#SortingAlgo)
14. [MergeSort](#mergeSort)
15. Design: Composition vs

Primitive types

<http://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html>

int -32 bit, char -16 bit, double -64 bit ,float 32 bit, long- 64 bit, short -16 bit, byte-8 bit.

**JRE**

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

JVM runs the byte code

JDK is superset of JRE.

## 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:**

Actually, this code is easier to analyze if you break it down into two parts:

Person john;

john = new Person();

In technical speak, the first line 'declares a variable of *type* Person. But what does that mean?? The general explanation is that I now have an empty variable that can only hold a Person object. But wait a minute - its an empty variable! There is nothing in that variables memory location. It turns out that 'types' are mechanically meaningless. Types were originally invented as a way to manage data and nothing else. Even when you declare primitive types such as int, str, chr (w/o initializing it), nothing happens within the computer. This weird syntactical aspect of programming is part of where people get the idea that classes are the blueprint of objects. OOP's have gotten even more confusing with types with delegate types, event handlers, etc. I would try not focus on them too much and just remember that they are all a misnomer. Nothing changes with the variable until its either becomes an object or is set to a memory address of an object.

The second line is also a bit confusing because it does two things at once:

The right side "new Person()" is evaluated first. It creates a new copy of the Person class - that is, it creates a new object.

The left side "john =", is then evaluated after that. It turns john into a reference variable giving it the memory address of the object that was just created on the right side of the same line.

If you want to become a good developer, its important to understand that no computer environment ever works based on philosophic ideals. Computers aren't even that logical - they're really just a big collection of wires that are glued together using basic boolean circuits (mostly NAND and OR).

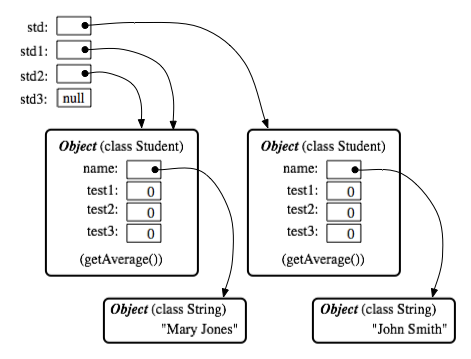
A class defines an object. You can go even further in many languages and say an interface defines common attributes and methods between objects.

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)

evict

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

http://javabeginnerstutorial.com/core-java-tutorial/different-ways-to-create-an-object-in-java/

# [Different ways to create an object in Java](http://javabeginnerstutorial.com/core-java-tutorial/different-ways-to-create-an-object-in-java/)

You must have used the “**new**” operator to create an Object of a Class. But is it the only way to create an Object?

Simple Answers is NO, then in how many ways we can create Object of a Class. There are several like

* Using New keyword
* Using New Instance (Reflection)
* Using Clone
* Using Deserilization
* Using ClassLoader
* … don’t know :)

## Using New Instance (Reflection)

Have you ever tried to connect to any DB using JDBC driver in Java, If your answer is yes then you must have seen “***Class.forName***“. We can also use it to create the object of a class. **Class.forName**actually loads the class in Java but doesn’t create any Object. To Create an Object of the Class you have to use ***newInstance***method of Class class.



1. Ways to create String in Java

String Pool in java

This prints true (even though we don't use equals method: correct way to compare strings)

String s = "a" + "bc";

String t = "ab" + "c";

System.out.println(s == t);

When compiler optimizes your string literals, it sees that both s and t have same value and thus you need only one string object. It's safe because String is immutable in Java.  
As result, both s and t point to the same object and some little memory saved.

Name 'string pool' comes from the idea that all already defined string are stored in some 'pool' and before creating new String object compiler checks if such string is already defined.

1. How to create immutable class
2. Class should be marked as final 2)
3. All fields must be private and final
4. Replace setters with constructor (for assigning a value to a variable).

<http://javarevisited.blogspot.com/2013/03/how-to-create-immutable-class-object-java-example-tutorial.html>

## Immutable Class Example in Java

Here is complete code example of writing immutable class in Java. We have followed simplest approach and all rules for making a class immutable, including it [making class final](http://javarevisited.blogspot.com/2011/12/final-variable-method-class-java.html) to avoid putting immutability at risk due to [Inheritance](http://javarevisited.blogspot.com/2012/10/what-is-inheritance-in-java-and-oops-programming.html) and [Polymorphism](http://javarevisited.blogspot.com/2011/08/what-is-polymorphism-in-java-example.html).

public final class Contacts {

    private final String name;

    private final String mobile;

    public Contacts(String name, String mobile) {

        this.name = name;

        this.mobile = mobile;

    }

    public String getName(){

        return name;

    }

    public String getMobile(){

        return mobile;

    }

}

This Java class is immutable, because its state can not be changed once created. You can see that all of it’s fields are final. This is one of the most simple way of creating immutable class in Java, where all fields of class also remains immutable like String in above case. Some time you may need to write immutable class which includes mutable classes like [java.util.Date](http://javarevisited.blogspot.com/2012/04/difference-between-javautildate-and.html), **despite storing Date into final field it can be modified** **internally,** if internal date is returned to the client. In order to preserve immutability in such cases, its advised to **return copy of original object**, which is also one of the [Java best practice](http://javarevisited.blogspot.co.uk/2012/08/top-10-jdbc-best-practices-for-java.html). here is another example of making a class immutable in Java, which includes mutable member variable.

public final class ImmutableReminder{

    private final Date remindingDate;

    public ImmutableReminder (Date remindingDate) {

        if(remindingDate.getTime() < System.currentTimeMillis()){

            throw new IllegalArgumentException("Can not set reminder” +

                        “ for past time: " + remindingDate);

        }

        this.remindingDate = new Date(remindingDate.getTime());

    }

    public Date getRemindingDate() {

        return (Date) remindingDate.clone();

    }

}

In above example of creating immutable class, [Date](http://javarevisited.blogspot.com/2011/09/convert-date-to-string-simpledateformat.html) is a mutable object. If getRemindingDate() returns actual Date object than despite remindingDate being final variable, internals of Date can be modified by client code. By returning clone() or copy of remindingDate, we avoid that danger and preserves immutability of class.

## Benefits of Immutable Classes in Java

As I said earlier Immutable classes offers several benefits, here are few to mention:

1) Immutable objects are by default [thread safe](http://javarevisited.blogspot.com/2012/01/how-to-write-thread-safe-code-in-java.html), can be shared without synchronization in concurrent environment.

2) Immutable object simplifies development, because its easier to share between multiple threads without external synchronization.

3) Immutable object boost performance of Java application by reducing [synchronization](http://java67.blogspot.com/2013/01/difference-between-synchronized-block-vs-method-java-example.html) in code.  
  
4) Another important benefit of Immutable objects is **reusability**, you can cache Immutable object and reuse them, much like String literals and Integers.  You can use [static factory methods](http://javarevisited.blogspot.it/2011/12/factory-design-pattern-java-example.html) to provide methods like valueOf(), which can return an existing Immutable object from cache, instead of creating a new one.

Apart from above advantages, immutable object has disadvantage of creating garbage as well. Since immutable object can not be reused and they are just a use and throw. String being a prime example, which can create lot of garbage and can potentially slow down application due to [heavy garbage collection](http://javarevisited.blogspot.com/2011/04/garbage-collection-in-java.html), but again that's extreme case and if used properly Immutable object adds lot of value.

Read more: <http://javarevisited.blogspot.com/2013/03/how-to-create-immutable-class-object-java-example-tutorial.html#ixzz3PmmB81uV>

**package nag.java.com;**

**import java.util.Calendar;**

**import java.util.Date;**

**final class ImmutableReminder{**

**private final Date remindingDate;**

**public ImmutableReminder(Date remindingDate) {**

**if(remindingDate.getTime() < System.currentTimeMillis()){**

**throw new IllegalArgumentException("Can not set reminder” + “ for past time: " + remindingDate);**

**}**

**this.remindingDate = new Date(remindingDate.getTime());**

**}**

**public Date getRemindingDate() {**

**try {**

**Thread.sleep(1000);**

**} catch (InterruptedException e) {**

**// TODO Auto-generated catch block**

**e.printStackTrace();**

**}**

**//return (Date) remindingDate.clone();**

**return (Date) remindingDate;**

**}**

**}**

**public class ImmutableReminderMain{**

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

**ImmutableReminder ir1 = new ImmutableReminder(new Date());**

**//ImmutableReminder ir2 = new ImmutableReminder(new Date("12/03/2015"));**

**long upd = ir1.getRemindingDate().getTime() + 1 \* 24 \* 60 \* 60 \* 1000;**

**System.out.println("ir1 :"+ir1.getRemindingDate());**

**System.out.println("Upd :"+upd);**

**ir1.getRemindingDate().setTime(upd);**

**//System.out.println("ir2 :"+ir2.getRemindingDate());**

**System.out.println("ir1 :"+ir1.getRemindingDate());**

**}**

**public static Date addDays(Date date, int days)**

**{**

**Calendar cal = Calendar.getInstance();**

**cal.setTime(date);**

**cal.add(Calendar.DATE, 3); //minus number would decrement the days**

**return cal.getTime();**

**}**

**}**

<http://stackoverflow.com/questions/10130054/why-to-create-a-string-object-using-new>

The basic difference between them is memory allocation.

First option i.e

String s1 = "hello";

When you use this **s1** is called as a **string literal** and memory for s1 is allocated at **compile time**.

But in 2nd case

String s2 = new String("hello");

In this case s2 is called as an object of String representing **hello**

When you tries to create two string literal using the first case, only one memory is referenced by those two literals. I mean String literals are working with a concept of **string pool**. when you create a 2nd string literal with same content, instead of allocating a new space compiler will return the same reference. Hence you will get **true** when you compare those two literals using **==** operator.

But in the 2nd case each time JVM will create a new object for each. and you have to compare their contents using **equals()** method but not with **==** operator.

If you want to create a new string object using 2nd case and also you don't want a new object, then you can use **intern()** method to get the same object.

String s = "hello";

String s1 = new String("hello").intern();

System.out.println(s == s1);

In this case instead of creating a new object, JVM will return the same reference **s**. So the output will be **true**

[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:

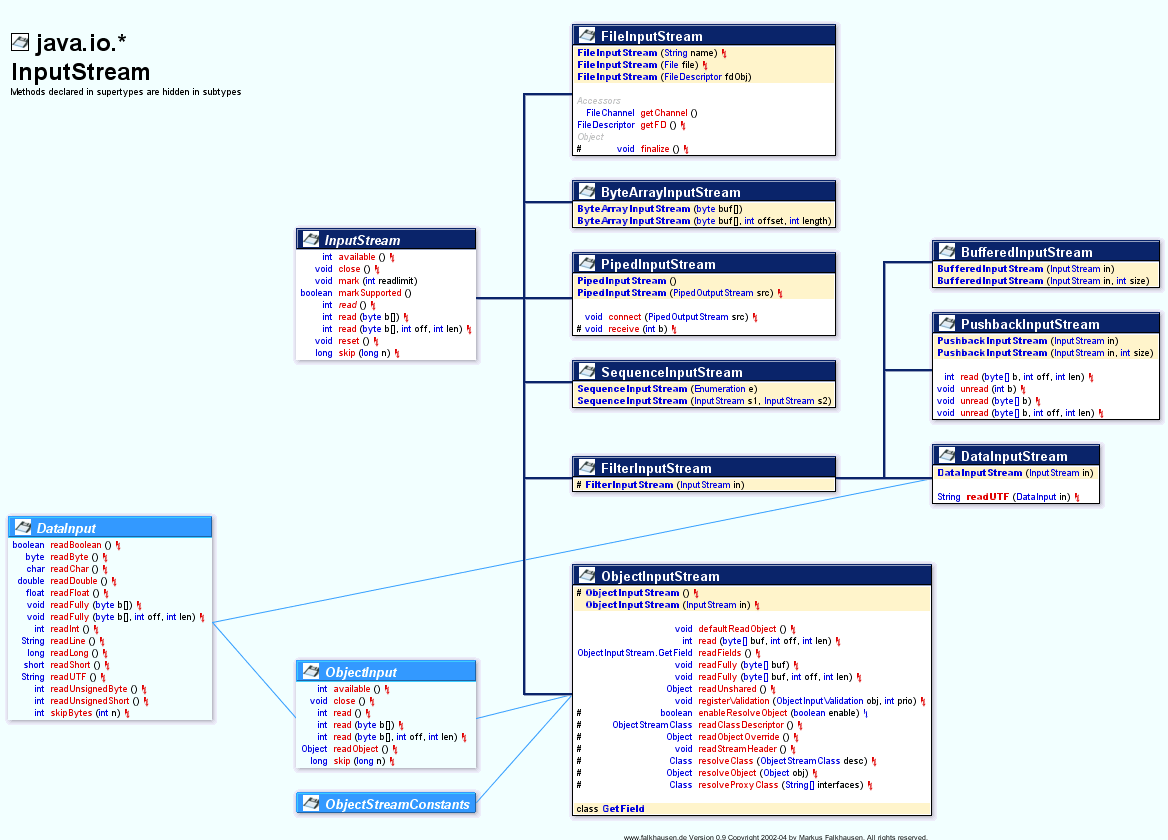
Objects of type [Scanner](http://docs.oracle.com/javase/7/docs/api/java/util/Scanner.html) are useful for breaking down formatted input into tokens and translating individual tokens according to their data type.

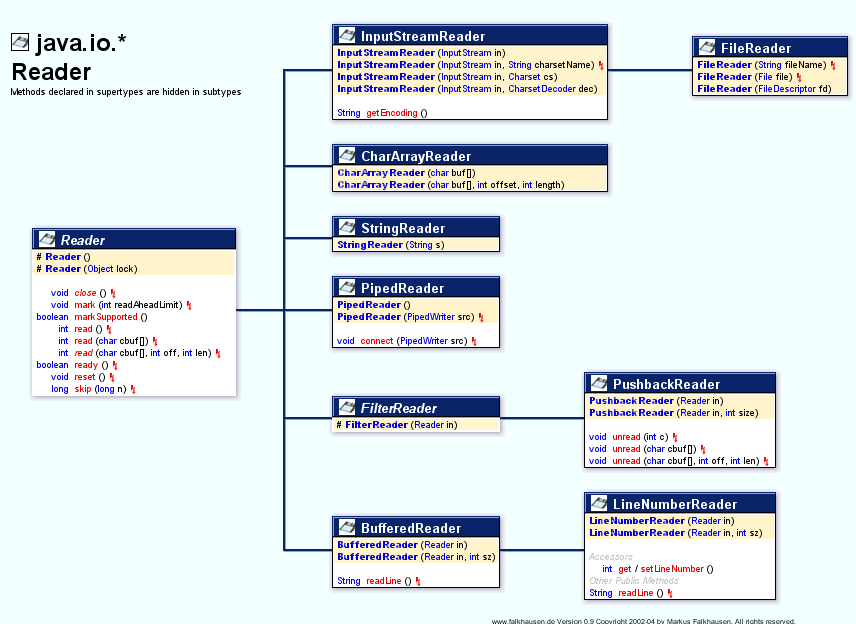
s = new Scanner (new BufferedReader(new FileReader("xanadu.txt"))); //default delimeter space

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 marker 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>

More serialization interview questions

<http://javarevisited.blogspot.com/2011/04/top-10-java-serialization-interview.html>

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>

<https://howtodoinjava.com/java/collections/hashmap/how-hashmap-works-in-java/>

**bucket is nothing Entry<K, V> [] array. Bucket location meaning index of the Entry array . and if two objects have the same hashCode then it will be stored as LinktedList in Entry<> in the bucket**

**How Hashmap works in Java**

A hashmap works like this (this is a little bit simplified, but it illustrates the basic mechanism):

It has a number of "buckets" which it uses to store key-value pairs in. Each bucket has a unique number - that's what identifies the bucket. When you put a key-value pair into the map, the hashmap will look at the hash code of the key, and store the pair in the bucket of which the identifier is the hash code of the key. For example: The hash code of the key is 235 -> the pair is stored in bucket number 235. (Note that one bucket can store more then one key-value pair).

When you lookup a value in the hashmap, by giving it a key, it will first look at the hash code of the key that you gave. The hashmap will then look into the corresponding bucket, and then it will compare the key that you gave with the keys of all pairs in the bucket, by comparing them with equals().

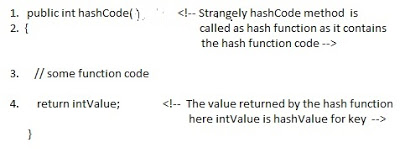
Now you can see how this is very efficient for looking up key-value pairs in a map: by the hash code of the key the hashmap immediately knows in which bucket to look, so that it only has to test against what's in that bucket.

Looking at the above mechanism, you can also see what requirements are necessary on the hashCode() and equals() methods of keys:

* If two keys are the same (equals() returns true when you compare them), their hashCode()method must return the same number. If keys violate this, then keys that are equal might be stored in different buckets, and the hashmap would not be able to find key-value pairs (because it's going to look in the same bucket).
* If two keys are different, then it doesn't matter if their hash codes are the same or not. They will be stored in the same bucket if their hash codes are the same, and in this case, the hashmap will use equals() to tell them apart.

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

**Also the Vararg must be the last argument in the method.  
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>

Another good example

**<http://www.javadb.com/using-varargs-in-java/>**

**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 nobody 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.*

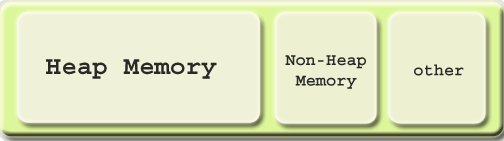
*In this article, we shall try to illuminate these issues to clear up the point*

*.*

#### 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**

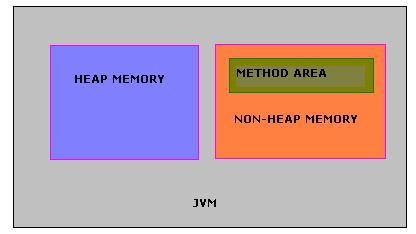
Also, the JVM has memory other than the heap, referred to as *non-heap memory*. It is created at the JVM startup and stores per-class structures such as runtime constant pool, field and method data, and the code for methods and constructors, as well as interned Strings.

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.

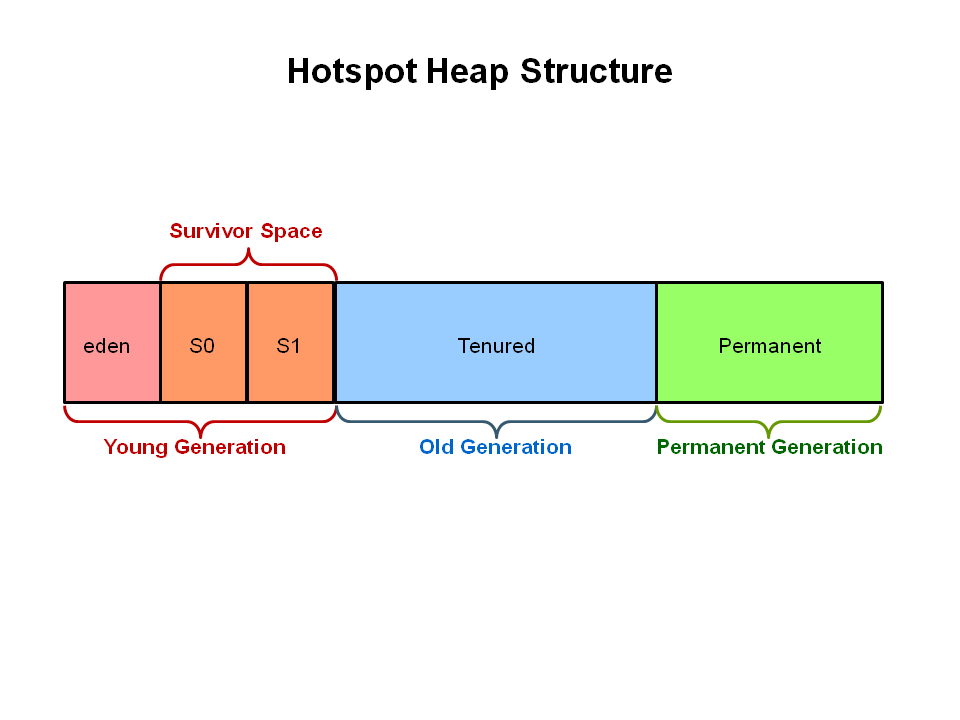
tual machine’s method area.

**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 inform ation 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.

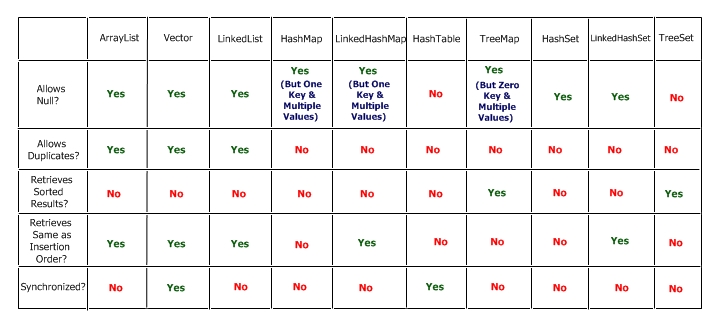
Here are the few other Checked Exceptions -

* 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.

**LinkedList** is faster in add and remove, but slower in get. In brief, LinkedList should be preferred if:

1. there are no large number of random access of element
2. there are a large number of add/remove operations

# 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.

**import** java.util.concurrent.BrokenBarrierException;  
**import** java.util.concurrent.CyclicBarrier;  
**import** java.util.logging.Level;  
**import** java.util.logging.Logger;  
  
/\*\*  
 \* Java program to demonstrate how to use CyclicBarrier in Java. CyclicBarrier is a

 \* new Concurrency Utility added in Java 5 Concurrent package.

 \*  
 \* @author Javin Paul  
 \*/  
**public** **class** CyclicBarrierExample {  
  
    *//Runnable task for each thread*  
    **private** **static** **class** Task **implements** [**Runnable**](http://javarevisited.blogspot.sg/2012/01/difference-thread-vs-runnable-interface.html) {  
  
        **private** **CyclicBarrier** barrier;  
  
        **public** Task(**CyclicBarrier** barrier) {  
            **this**.barrier = barrier;  
        }  
  
        @**Override**  
        **public** **void** run() {  
            **try** {  
                **System**.out.println(**Thread**.currentThread().getName() + " is waiting on barrier");  
                barrier.await();  
                **System**.out.println(**Thread**.currentThread().getName() + " has crossed the barrier");  
            } **catch** (**InterruptedException** ex) {  
                **Logger**.getLogger(CyclicBarrierExample.**class**.getName()).log(**Level**.SEVERE, **null**, ex);  
            } **catch** (**BrokenBarrierException** ex) {  
                **Logger**.getLogger(CyclicBarrierExample.**class**.getName()).log(**Level**.SEVERE, **null**, ex);  
            }  
        }  
    }  
  
    **public** **static** **void** main(**String** args[]) {  
  
        *//creating CyclicBarrier with 3 parties i.e. 3 Threads needs to call await()*  
        **final** **CyclicBarrier** cb = **new** **CyclicBarrier**(3, **new** **Runnable**(){  
            @**Override**  
            **public** **void** run(){  
                *//This task will be executed once all thread reaches barrier*  
                **System**.out.println("All parties are arrived at barrier, lets play");  
            }  
        });  
  
        *//starting each of thread*  
        **Thread** t1 = **new** **Thread**(**new** Task(cb), "Thread 1");  
        **Thread** t2 = **new** **Thread**(**new** Task(cb), "Thread 2");  
        **Thread** t3 = **new** **Thread**(**new** Task(cb), "Thread 3");  
  
        t1.start();  
        t2.start();  
        t3.start();  
        
    }  
}  
  
**Output:**  
**Thread** 1 is waiting on barrier  
**Thread** 3 is waiting on barrier  
**Thread** 2 is waiting on barrier  
All parties are arrived at barrier, lets play  
**Thread** 3 has crossed the barrier  
**Thread** 1 has crossed the barrier  
**Thread** 2 has crossed the barrier

Read more: <http://javarevisited.blogspot.com/2012/07/cyclicbarrier-example-java-5-concurrency-tutorial.html#ixzz3Le9yisJ2>

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.

**Contract is if 2 objects are equal then they should have same hashcode and if 2 objects are not equal then they may or may not have same hash code.**

# As much as is reasonably practical, the hashCode method defined by class Object does return distinct integers for distinct objects. (This is typically implemented by converting the internal address of the object into an integer, but this implementation technique is not required by the JavaTM programming language.)

# [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.

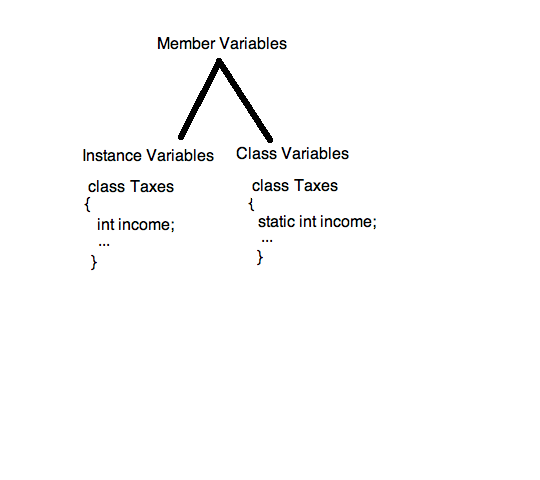
So compiler gives the compile time method no matching method found.

## 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), 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.

Generics

<http://www.journaldev.com/1663/java-generics-tutorial-example-class-interface-methods-wildcards-and-much-more#java-generics-example>

|  |  |
| --- | --- |
| GenericType.java | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | package com.journaldev.generics;    public class GenericsType<T> {        private T t;        public T get(){          return this.t;      }        public void set(T t1){          this.t=t1;      }        public static void main(String args[]){          GenericsType<String> type = new GenericsType<>();          type.set("Pankaj"); //valid            GenericsType type1 = new GenericsType(); //raw type          type1.set("Pankaj"); //valid          type1.set(10); //valid and autoboxing support      }  } |

Notice the use of GenericsType class in the main method. We don’t need to do type-casting and we can remove ClassCastException at runtime. If we don’t provide the type at the time of creation, compiler will produce a warning that “GenericsType is a raw type. References to generic type GenericsType<T> should be parameterized”. When we don’t provide type, the type becomes Object and hence it’s allowing both String and Integer objects but we should always try to avoid this because we will have to use type casting while working on raw type that can produce runtime errors.

**Tip**: We can use @SuppressWarnings("rawtypes") annotation to suppress the compiler warning, check out [**java annotations tutorial**](http://www.journaldev.com/721/java-annotations-tutorial-with-custom-annotation-example-and-parsing-using-reflection).

* E – Element (used extensively by the Java Collections Framework, for example ArrayList, Set etc.)
* K – Key (Used in Map)
* N – Number
* T – Type
* V – Value (Used in Map)
* S,U,V etc. – 2nd, 3rd, 4th types

Question mark (?) is the wildcard in generics and represent an unknown type.

### Generics Unbounded Wildcard

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

|  |  |
| --- | --- |
| 1  2  3  4  5 | public static void printData(List<?> list){          for(Object obj : list){              System.out.print(obj + "::");          }      } |

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

**Use of generics in Java:**

I have mostly used generics to make type safe collections. What are the other uses of generics?

To allow user-created, custom classes to be used with your code.

Say you release an SDK that enables some kind of special functionality. Generics will allow developers to utilize your functionality in many places, with almost any class.

The purpose of generics to reduce code repetition.

# [Understanding upper and lower bounds on ? in Java Generics](http://stackoverflow.com/questions/19795709/understanding-upper-and-lower-bounds-on-in-java-generics)

<http://stackoverflow.com/questions/19795709/understanding-upper-and-lower-bounds-on-in-java-generics>

<http://www.journaldev.com/1663/java-generics-tutorial-example-class-interface-methods-wildcards-and-much-more>

<http://docs.oracle.com/javase/tutorial/java/generics/lowerBounded.html>

<http://docs.oracle.com/javase/tutorial/extra/generics/index.html>

# Lower Bounded Wildcards

A  lower bounded wildcard is expressed using the wildcard character ('?'), following by the super keyword, followed by its lower bound: <? super A>.

Say you want to write a method that puts Integer objects into a list. To maximize flexibility, you would like the method to work on List<Integer>, List<Number>, and List<Object> — anything that can hold Integer values.

The following code adds the numbers 1 through 10 to the end of a list:

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

for (int i = 1; i <= 10; i++) {

list.add(i);

}

}

# Upper Bounded Wildcards

You can use an upper bounded wildcard to relax the restrictions on a variable. For example, say you want to write a method that works on List<Integer>, List<Double>, and List<Number>; you can achieve this by using an upper bounded wildcard.

To declare an upper-bounded wildcard, use the wildcard character ('?'), followed by the extends keyword, followed by its upper bound. Note that, in this context, extends is used in a general sense to mean either "extends" (as in classes) or "implements" (as in interfaces).

To write the method that works on lists of Number and the subtypes of Number, such as Integer, Double, and Float, you would specify List<? extends Number>. The term List<Number> is more restrictive than List<? extends Number> because the former matches a list of type Number only, whereas the latter matches a list of type Number or any of its subclasses.

Consider the following process method:

public static void process(List**<? extends Foo>** list) { /\* ... \*/ }

The upper bounded wildcard, <? extends Foo>, where Foo is any type, matches Foo and any subtype of Foo. The process method can access the list elements as type Foo:

public static void process(List<? extends Foo> list) {

for (Foo elem : list) {

// ...

}

}

<https://www.geeksforgeeks.org/bounded-types-generics-java/>

* Sometimes we don’t want whole class to be parameterized, in that case we can create java [generics](https://www.geeksforgeeks.org/generics-in-java/) method. Since constructor is a special kind of method, we can use generics type in constructors too.
* <T extends **superClassName**>
* Note that, in this context, extends is used in a general sense to mean either “extends” (as in classes). Also, This specifies that T can only be replaced by superClassName, or subclasses of superClassName. Thus, superclass defines an inclusive, upper limit.
* Bounded type parameters can be used with methods as well as classes and interfaces.
* Java Generics supports multiple bounds also, i.e . In this case A can be an interface or class. If A is class then B and C should be interfaces. We can’t have more than one class in multiple bounds.

<T extends **superClassName** & **Interface**>

Enum

Enum in Javais a keyword, a feature which is used to represent fixed number of well known values in Java, For example Number of days in Week, Number of planets in Solar system etc. **Enumeration (Enum) in Java** was introduced in JDK 1.5   
  
Read more: <http://javarevisited.blogspot.com/2011/08/enum-in-java-example-tutorial.html#ixzz39XJzF3q6>

<http://javarevisited.blogspot.com/2011/08/enum-in-java-example-tutorial.html>

### How to represent enumerable value without Java enum

[java enum example, enum in java tutorial](http://javarevisited.blogspot.com/2011/08/convert-string-to-integer-to-string.html)Since **Enum in Java** is only available from **Java 1.5** its worth to discuss how we used to represent enumerable values in Java prior JDK 1.5 and without it. I use public static [final constant](http://javarevisited.blogspot.sg/2011/12/final-variable-method-class-java.html) to replicate enum like behavior. Let’s see an Enum example in Java to understand the concept better. In this example we will use US Currency Coin as enumerable which has values like PENNY (1) NICKLE (5), DIME (10), and QUARTER (25).

**public class** CurrencyDenom {

**public** **static** **final** **int** *PENNY* = 1;

**public** **static** **final** **int** *NICKLE* = 5;

**public** **static** **final** **int** *DIME* = 10;

**public** **static** **final** **int** *QUARTER* = 25;

}

**public class** Currency {

**private int** currency; //CurrencyDenom.PENNY,CurrencyDenom.NICKLE,

                         // CurrencyDenom.DIME,CurrencyDenom.QUARTER

}  
  
 Though this can server our purpose it has some serious limitations:  
  
**1) No Type-Safety**: First of all it’s not [type-safe](http://javarevisited.blogspot.sg/2011/09/generics-java-example-tutorial.html); you can assign any valid int value to currency e.g. 99 though there is no coin to represent that value.  
  
**2) No Meaningful Printing**: printing value of any of these constant will print its numeric value instead of meaningful name of coin e.g. when you print NICKLE it will print "5" instead of "NICKLE"  
  
**3) No namespace:** to access the currencyDenom constant we need to prefix class name e.g. CurrencyDenom.PENNY instead of just using PENNY though this can also be achieved by using [static import in JDK 1.5](http://javarevisited.blogspot.sg/2011/11/static-keyword-method-variable-java.html)  
  
**Java Enum** is answer of all this limitation. Enum in Java is type-safe, provides meaningful String names and has there own namespace. Now let's see same example using Enum in Java:

**public** **enum** Currency {PENNY, NICKLE, DIME, QUARTER};

Here Currency is our **enum** and PENNY, NICKLE, DIME, QUARTER are **enum constants**. Notice **curly braces around enum constants** because Enum are type like [class](http://javarevisited.blogspot.sg/2011/10/class-in-java-programming-general.html)and[interface in Java](http://javarevisited.blogspot.sg/2012/04/10-points-on-interface-in-java-with.html). Also we have followed similar naming convention for enum like class and interface (first letter in Caps) and since *Enum constants are implicitly static final* we have used all caps to specify them like Constants in Java.

### What is Enum in Java

Now back to primary questions **“What is Enum in java”** simple *answer Enum is a keyword in java* and on more detail term Java Enum is type like class and interface and can be used to define a set of Enum constants. Enum constants are [implicitly static and final](http://javarevisited.blogspot.sg/2011/12/final-variable-method-class-java.html) and you can not change there value once created. Enum in Java provides type-safety and can be used inside switch statment like int variables. Since enum is a keyword you can not use as variable name and since its only introduced in JDK 1.5 all your previous code which has enum as variable name will not work and needs to be re-factored.

### Benefits of Enums in Java:

1) **Enum is type-safe** you can not assign anything else other than predefined Enum constants to an Enum variable. It is compiler error to assign something else unlike the public static final variables used in Enum int pattern and Enum String pattern.  
  
2) Enum has its own name-space.  
  
3) Best feature of Enum is **you can use Enum in Java inside Switch statement** like int or char primitive data type.we will also see example of using java enum in switch statement in this java enum tutorial.  
  
4) Adding new constants on Enum in Java is easy and you can add new constants without breaking existing code.

## Important points about Enum in Java

1) **Enums in Java are type-safe** and has their own name-space. It means your enum will have a type for example "Currency" in below example and you can not assign any value other than specified in Enum Constants.

**public** **enum** Currency {*PENNY*, *NICKLE*, *DIME*, *QUARTER*};

Currency coin = Currency.PENNY;

coin = 1; //compilation error    
  
  
2**) Enum in Java are reference type**like [class](http://javarevisited.blogspot.sg/2011/10/class-in-java-programming-general.html)or [interface](http://javarevisited.blogspot.sg/2012/04/10-points-on-interface-in-java-with.html)and you can define constructor, methods and variables inside java Enum which makes it more powerful than Enum in C and C++ as shown in next example of Java Enum type.  
  
  
3) You can **specify values of enum constants at the creation time** as shown in below example:  
**public** **enum** Currency {*PENNY*(1), *NICKLE*(5), *DIME*(10), *QUARTER*(25)};  
But for this to work you need to define a member variable and a constructor because PENNY (1) is actually [calling a constructor](http://javarevisited.blogspot.sg/2012/01/what-is-constructor-overloading-in-java.html) which accepts int value , see below example.

**public** **enum** Currency {

*PENNY*(1), *NICKLE*(5), *DIME*(10), *QUARTER*(25);

**private** **int** value;

**private** Currency(**int** value) {

**this**.value = value;

        }

};     
**Constructor of enum in java** must be [**private**](http://javarevisited.blogspot.sg/2012/03/private-in-java-why-should-you-always.html)any other access modifier will result in compilation error. Now to get the value associated with each coin you can define a public getValue() method inside java enum like any normal java class. Also semi colon in the first line is optional.  
  
  
4) Enum constants are implicitly [static](http://javarevisited.blogspot.sg/2012/03/mixing-static-and-non-static.html)and [final](http://javarevisited.blogspot.sg/2010/10/why-string-is-immutable-in-java.html)and cannot be changed once created. For example below code of java enum will result in compilation error:

Currency.PENNY = Currency.DIME;

The final field EnumExamples.Currency.PENNY cannot be re assigned.  
  
    
    
5) **Enum in java can be used as an argument on switch statment** and with "case:" like int or char primitive type. This feature of java enum makes them very useful for switch operations. Let’s see an example of how to use java enum inside switch statement:  

   Currency usCoin = Currency.DIME;

**switch** (usCoin) {

**case** PENNY:

                    System.out.println("Penny coin");

**break**;

**case** NICKLE:

                    System.out.println("Nickle coin");

**break**;

**case** DIME:

                    System.out.println("Dime coin");

**break**;

**case** QUARTER:

                    System.out.println("Quarter coin");

    }  
    
from JDK 7 onwards you can also [String in Switch case in Java](http://javarevisited.blogspot.sg/2011/08/string-switch-case-jdk7-example.html) code.  
  
6) Since **constants defined inside Enum in Java are final you can safely compare them using "==" equality operator** as shown in following example of  Java Enum:

Currency usCoin = Currency.DIME;

**if**(usCoin == Currency.DIME){

  System.*out*.println("enum in java can be compared using ==");

}

By the way comparing objects using == operator is not recommended, Always use [equals() method](http://javarevisited.blogspot.sg/2011/02/how-to-write-equals-method-in-java.html) or [compareTo() method](http://javarevisited.blogspot.sg/2011/11/how-to-override-compareto-method-in.html) to compare Objects.

7) Java compiler automatically generates static values() method for every enum in java. Values() method returns array of Enum constants in the same order they have listed in Enum and you can use values() to [iterate](http://javarevisited.blogspot.sg/2011/10/java-iterator-tutorial-example-list.html)over values of Enum  in Java as shown in below example:

**for**(Currency coin: Currency.values()){

        System.*out*.println("coin: " + coin);

}  
  
And it will print:

**coin: PENNY**

**coin: NICKLE**

**coin: DIME**

**coin: QUARTER**  
                  
Notice the order its exactly same **with defined order in enums**.  
  
Read more: <http://javarevisited.blogspot.com/2011/08/enum-in-java-example-tutorial.html#ixzz39XK5filM>

Wildcards

It's written Collection<?> (pronounced "collection of unknown"), that is, a collection whose element type matches anything. It's called a **wildcard type** for obvious reasons. We can write:

**void** printCollection(Collection<?> c) {

**for** (Object e : c) {

System.out.println(e);

}

}

<http://www.journaldev.com/1663/java-generics-tutorial-example-class-interface-methods-wildcards-and-much-more>

### Generics Type Naming Convention

Naming convention helps us understanding code easily and having a naming convention is one of the best practices of java programming language. So generics also comes with it’s own naming conventions. Usually type parameter names are single, uppercase letters to make it easily distinguishable from java variables. The most commonly used type parameter names are:

* E – Element (used extensively by the Java Collections Framework, for example ArrayList, Set etc.)
* K – Key (Used in Map)
* N – Number
* T – Type
* V – Value (Used in Map)
* S,U,V etc. – 2nd, 3rd, 4th types

# [Double vs. BigDecimal?](http://stackoverflow.com/questions/3413448/double-vs-bigdecimal)

<http://stackoverflow.com/questions/3413448/double-vs-bigdecimal>

A BigDecimal is an exact way of representing numbers. A Double has a certain precision. Working with doubles of various magnitudes (say d1=1000.0 and d2=0.001) could result in the 0.001 being dropped altogether when summing as the difference in magnitude is so large. With BigDecimal this would not happen.

**The disadvantage of BigDecimal is that it's slower, and it's a bit more difficult to program algorithms that way (due to + - \* and / not being overloaded).**

If you are dealing with money, or precision is a must, use BigDecimal. Otherwise Doubles tend to be good enough.

I do recommend reading the [javadoc](http://download.oracle.com/javase/1.5.0/docs/api/java/math/BigDecimal.html) of BigDecimal as they do explain things better than I do here :)

Java.sql.Date and java.util.Date and java.util java.sql.Time vs java.sql.Timestamp

1) First difference on java.sql.Time vs java.sql.Timestamp vs java.sql.Date is about information they represent :

JDBC TIME or java.sql.Time represent only time information e.g. hours, minutes and seconds **without any date information**.

JDBC DATE or java.sql.Date represent only date information e.g. year, month and day **without any time information.**

JDBC TIMESTAMP or java.sql.Timestamp  **represent both date and time information** including nanosecond details.

2) java.sql.Time and java.sql.Timestamp extends [java.util.Date](http://javarevisited.blogspot.in/2011/09/convert-date-to-string-simpledateformat.html) class but java.sql.Date is independent.

3) Time information from java.sql.Date and Date information from java.sql.Time is normalized and may set to zero in order to confirm ANSI SQL DATE and TIME types.

So difference between Time, Timestamp and Date of SQL package is clear in terms of what they represent. On contrary java.util.Date also represent Date and time information but **without nanosecond details** and that's why many people prefer to store date as long value (millisecond passed from epoch January 1, 1970 00:00:00.000 GMT). If you compare tojava.sql.Timestamp with [equals() method](http://javarevisited.blogspot.sg/2011/02/how-to-write-equals-method-in-java.html) it will return false as value of nanosecond is unknown.

Read more: <http://javarevisited.blogspot.com/2012/10/difference-between-javasqltime-date-timestamp-jdbc-interview-question.html#ixzz39vKRXen7>

**What is difference between java.sql.Time and java.sql.TimeStamp in Java.**

Answer : This JDBC questions is similar to earlier JDBC interview question java.sql.Date vs java.util.Date. Main difference is that java.sql.Time class doesn't contain any date information on it while java.sql.TimeStamp contains date information. See [4 difference between Time and Timestamp in Java JDBC](http://javarevisited.blogspot.sg/2012/10/difference-between-javasqltime-date-timestamp-jdbc-interview-question.html) for more differences.

### How to convert java.sql.Date to java.util.Date in Java with Example

Difference between DTO VO POJO and Java beans

http://stackoverflow.com/questions/1612334/difference-between-dto-vo-pojo-javabeans

Basically,

DTO: "Data transfer objects " can travel between seperate layers in software architecture.

VO: "Value objects " hold a object such as Integer,Money etc.

POJO: Plain Old Java Object which is not a special object.

Java Beans: requires a Java Class to be serializable, have a no-arg constructor and a getter and setter for each field

Simple Registration form where you have attributes usename,password and email id . when you sumbit this form . In your servlet RegistrationServlet.java file you will get all the attributes from view layer to business layer where you pass the attributes to java beans and then to the DAO or the persistence layer . DTO's helps in transporting the attributes from view layer to bussiness layer and finally to the persistence layer .

DTO was mainly used to get data transportd across the network efficiently , it may be even from JVM to another JVM .

DTOs are often java.io.Serializable - inorder to transfer data across JVM

**VO -** A Value Object [1,2] represents itself a fix set of data and is similar to a Java enum. A Value Object's identity is based on their state rather than on their object identity and is immutable. A real world example would be Color.RED, Color.BLUE, SEX.FEMALE etc.

**POJO V/S JavaBeans**

1. The Java-Beanness of a POJO is that it's public attributes are all accessed via getters and setters that conform to the JavaBeans conventions. e.g. private String foo; public String getFoo(){...} public void setFoo(String foo){...};
2. JavaBeans must implement Serializable and have a no-argument constructor. where as in POJO doesnot have these restrictions .

### Value Object (VO)

Value Object is simple object that holds values. The VO should be entirely **immutable**.

### Data Transfer Object (DTO)

DTO is serializable object, which is used for transferring data between different processes. It shouldn’t be mixed with VO. This is exposed as design pattern.

### Data Access Object (DAO)

This is an object, which provides mechanism for accessing objects from the persistence layer. This is actually one of the most popular design patterns.

## Why Java doesn't support multiple inheritance

1) First reason is **ambiguity around Diamond problem**, consider a class A has foo() method and then B and C derived from A and has there own foo() implementation and now class D derive from B and C using multiple [inheritance](http://javarevisited.blogspot.com/2012/10/what-is-inheritance-in-java-and-oops-programming.html) and if we refer just foo() compiler will not be able to decide which foo() it should invoke. This is also called Diamond problem because structure on this inheritance scenario is similar to 4 edge diamond, see below

           A foo()

           / \

          /   \

   foo() B     C foo()

          \   /

           \ /

            D

           foo()

Read more: <http://javarevisited.blogspot.com/2011/07/why-multiple-inheritances-are-not.html#ixzz3AVVbeq64>

1. Second and more convincing reason to me is that **multiple inheritances does complicate the design and creates problem during casting, constructor chaining etc** and given that there are not many scenario on which you need multiple inheritance its wise decision to omit it for the sake of simplicity.  
     
   Read more: <http://javarevisited.blogspot.com/2011/07/why-multiple-inheritances-are-not.html#ixzz3AVVm4ESI>

### Singleton Design Pattern with Example and Program Code : Design patter Interview Question

*For 6-7 years or above  experience , interviewer may ask about the double checked locking .*

In **double checked locking**  first we check if the object is created , if not then we create one using synchronized block.

**Below  is the best  Optimized code for the Singleton pattern**

**public** **class** **JavaHungrySingleton**

{

**private** **static** **volatile** JavaHungrySingleton uniqueInstance;

**private** **JavaHungrySingleton**(){}

**public** **static** JavaHungrySingleton **getInstance**()

{

**if** (uniqueInstance ==**null** )

{

**synchronized**(JavaHungrySingleton.class)

{

**if** (uniqueInstance ==**null** )

{

uniqueInstance=**new** JavaHungrySingleton();

}

}

}

**return** uniqueInstance ;

}

// other useful methods here

}

How to Compare two String in Java - String Comparison Example  
Here are four examples of comparing String in Java

1) String comparison using equals method

2) String comparison using equalsIgnoreCase method

2) String comparison using compareTo method

4) String comparison using compareToIgnoreCase method

Read more: <http://javarevisited.blogspot.com/2012/03/how-to-compare-two-string-in-java.html#ixzz3AW4IpKgi>

USE OF NESTED CLASSES IN JAVA: When to use which one.

http://stackoverflow.com/questions/18396016/when-to-use-inner-classes-in-java-for-helper-classes

 A static nested class doesn't have access to the members of the enclosing class (unless it is static). If this matches your requirement, make it static. Non-static inner classes, on the other hand, have full access to the members of the enclosing class.

# [Does it make sense to define a final String in Java? [duplicate]](http://stackoverflow.com/questions/10233309/does-it-make-sense-to-define-a-final-string-in-java)

The String object is immutable but what it is is actually a reference to a String object which could be changed.

For example:

String someString = "Lala";

You can reassign the value held by this variable (to make it reference a different string):

someString = "asdf";

However, with this:

final String someString = "Lala";

Then the above reassignment would not be possible and would result in a compile-time error.

How SubString method works in Java - Memory Leak Fixed in JDK 1.7  
  
Read more: <http://javarevisited.blogspot.com/2011/10/how-substring-in-java-works.html#ixzz3Ce8tmX00>

If you look substring method inside String class, you will figure out that it calls String (int offset, int count, char value []) [constructor](http://javarevisited.blogspot.com/2012/12/what-is-constructor-in-java-example-chainning-overloading.html) to create new String object. What is interesting here is, value[], which is the same character array used to represent original string. So **what's wrong with this**?  
  
Read more: <http://javarevisited.blogspot.com/2011/10/how-substring-in-java-works.html#ixzz3Ce9ZpEuj>

# [What is the difference between NoClassDefFoundError and ClassNotFoundException?](http://stackoverflow.com/questions/1457863/what-is-the-difference-between-noclassdeffounderror-and-classnotfoundexception)

|  |  |
| --- | --- |
| down voteaccepted | The difference from the Java API Specifications is as follows.  For [ClassNotFoundException](http://java.sun.com/javase/6/docs/api/java/lang/ClassNotFoundException.html):  Thrown when an application tries to load in a class through its string name using:   * The forName method in class Class. * The findSystemClass method in class ClassLoader. * The loadClass method in class ClassLoader.   but no definition for the class with the specified name could be found.  For [NoClassDefFoundError](http://java.sun.com/javase/6/docs/api/java/lang/NoClassDefFoundError.html):  Thrown if the Java Virtual Machine or a ClassLoader instance tries to load in the definition of a class (as part of a normal method call or as part of creating a new instance using the new expression) and no definition of the class could be found.  The searched-for class definition existed when the currently executing class was compiled, but the definition can no longer be found.  **So, it appears that the NoClassDefFoundError occurs when the source was successfully compiled, but at runtime, the required class files were not found. This may be something that can happen in the distribution or production of JAR files, where not all the required class files were included.**  As for ClassNotFoundException, it appears that it may stem from trying to make reflective calls to classes at runtime, but the classes the program is trying to call is does not exist.  The difference between the two is that one is an Error and the other is an Exception. With NoClassDefFoundError is an Error and it arises from the Java Virtual Machine having problems finding a class it expected to find. A program that was expected to work at compile-time can't run because of class files not being found, or is not the same as was produced or encountered at compile-time. This is a pretty critical error, as the program cannot be initiated by the JVM.  On the other hand, the ClassNotFoundException is an Exception, so it is somewhat expected, and is something that is recoverable. Using reflection is can be error-prone (as there is some expectations that things may not go as expected. There is no compile-time check to see that all the required classes exist, so any problems with finding the desired classes will appear at runtime. |

**java.lang.NoClassDefFoundError is thrown when a particular class referenced by your program is not available in the classpath.**

# [whats the difference between “.equals and ==”](http://stackoverflow.com/questions/1643067/whats-the-difference-between-equals-and)

n Java, == always just compares two references (for non-primitives, that is) - i.e. it tests whether the two operands refer to the same object.

However, the equals method can be overridden - so two distinct objects can still be equal.

For example:

String x = "hello";

String y = new String(new char[] { 'h', 'e', 'l', 'l', 'o' });

System.out.println(x == y); // false

System.out.println(x.equals(y)); // true

Additionally, it's worth being aware that any two equal string constants (primarily string literals, but also combinations of string constants via concatenation) will end up referring to the same string. For example:

String x = "hello";

String y = "he" + "llo";

System.out.println(x == y); // true!

Here x and y are references to the same string, because y is a compile-time constant equal to "hello"

### Difference between Array vs ArrayList in Java

**http://java67.blogspot.com/2012/12/difference-between-array-vs-arraylist-java.html**

First and Major difference between Array and ArrayList in Java is that Array is a **fixed length data structure** while ArrayList is a variable length [Collection class](http://java67.blogspot.sg/2012/09/java-collection-interview-questions.html). You can not change length of Array once created in Java butArrayList re-size itself when gets full depending upon capacity and load factor. Since ArrayList is internally backed by Array in Java, any resize operation in ArrayList will slow down performance as it involves creating new Array and [copying content](http://java67.blogspot.sg/2012/07/copy-elements-from-list-to-set-in-java-collection-example.html) from old array to new array.

One more major difference between ArrayList and Array is that, **you can not store primitives in ArrayList**, it can only contain Objects. While Array can contain both primitives and Objects in Java

# [Declare array in Java?](http://stackoverflow.com/questions/1200621/declare-array-in-java)

You can either use array declaration or array literal (but only when you declare and affect the variable right away, array literals cannot be used for re-assigning an array).

For primitive types:

int[] myIntArray = new int[3];

int[] myIntArray = {1,2,3};

int[] myIntArray = new int[]{1,2,3};

For classes, for example String, it's the same:

String[] myStringArray = new String[3];

String[] myStringArray = {"a","b","c"};

String[] myStringArray = new String[]{"a","b","c"};

There are two types of array.

## One Dimensional Array

Syntax for default values:

int[] num = new int[5];

Or (less preferred)

int num[] = new int[5]

Syntax with values given:

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

Or (less preferred)

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

Note: For convenience int[] num is preferable because it clearly tells that you are talking here about array. Otherwise no difference. Not at all.

## Multidimensional array

### Declaration

int[][] num = new int[5][2];

Or

int num[][] = new int[5][2];

Or

int[] num[] = new int[5][2];

### Initialization

num[0][0]=1;

num[0][1]=2;

num[1][0]=1;

num[1][1]=2;

num[2][0]=1;

num[2][1]=2;

num[3][0]=1;

num[3][1]=2;

num[4][0]=1;

num[4][1]=2;

Or

int[][] num={ {1,2}, {1,2}, {1,2}, {1,2}, {1,2} };

### Ragged Array (or Non-rectangular Array)

int[][] num = new int[5][];

num[0] = new int[1];

num[1] = new int[5];

num[2] = new int[2];

num[3] = new int[3];

So here we are defining columns explicitly.  
**Another Way:**

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

## For Accessing:

for (int i=0; i<(num.length); i++ ) {

for (int j=0;j<num[i].length;j++)

System.out.println(num[i][j]);

}

Alternatively:

for (int[] a : num) {

for (int i : a) {

System.out.println(i);

}

}

**Easiest way to convert a List to a Set?**

There are two ways to do so, depending on how you want equal defined. The first piece of code puts a list into a [**HashSet**](http://docs.oracle.com/javase/7/docs/api/java/util/HashSet.html). Duplication is then identified mostly by hashCode(). In most cases, it will work. But if you need to specify the way of comparison, it is better to use the second piece of code where you can define your own comparator.

|  |
| --- |
| Set<Integer> set = **new** HashSet<Integer>(list); |
| Set<Integer> set = **new** TreeSet<Integer>(aComparator);  set.addAll(list); |

**Can you override private or static method in Java ?**

Another popular Java tricky question, As I said method overriding is a good topic to ask trick questions in Java.  Anyway, you can not override private or static method in Java, if you create similar method with same return type and same method arguments that's called method hiding.

**If a method throws NullPointerException in super class, can we override it with a method which throws RuntimeException?**

One more tricky Java questions from overloading and overriding concept. Answer is you can very well throw super class of RuntimeException in overridden method but you can not do same if its checked Exception.

# [How does LinkedList work internally in Java?](http://stackoverflow.com/questions/8239310/how-does-linkedlist-work-internally-in-java)

So, is LinkedList internally an array-like sequence?

No. It's a series of instances of a private nested class Entry, which has next, previous and element references. Note that you could have found this out yourself by looking at the source code, which comes with the JDK.

The reason why this internal structure is not exposed is that it prevents the strcture from becoming corrupted and e.g. containing a loop. And the unifrom access via the List and Deque interfaces allows polymorphic use.

**Comparision between**[**ArrayList**](https://www.geeksforgeeks.org/arraylist-in-java/)**and**[**LinkedList**](http://geeksquiz.com/linked-list-in-java/)**:-**

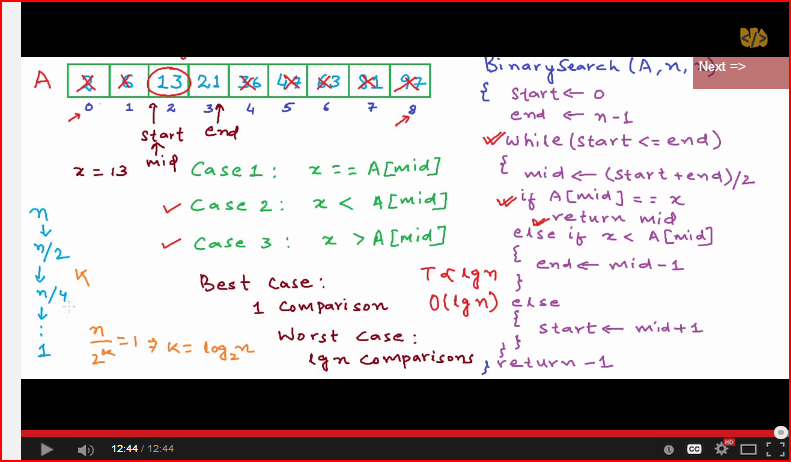
1. Insertions are easy and fast in LinkedList as compared to ArrayList because there is no  
   risk of resizing array and copying content to new array if array gets full which makes  
   adding into ArrayList of O(n) in worst case, while adding is O(1) operation in LinkedList  
   in Java. ArrayList also needs to be update its index if you insert something anywhere except  
   at the end of array.
2. Removal also better in LinkedList than ArrayList due to same reasons as insertion.
3. LinkedList has more memory overhead than ArrayList because in ArrayList each index only  
   holds actual object (data) but in case of LinkedList each node holds both data and address  
   of next and previous node.
4. Both LinkedList and ArrayList require O(n) time to find if an element is present or not. However we can do Binary Search on ArrayList if it is sorted and therefore can search in O(Log n) time.

**How Binary Search works:**

**There are two important search algorithms:**

**Linear Search and Binary Search**

**<https://www.youtube.com/watch?v=JQhciTuD3E8>**

****

[**http://www.programmingsimplified.com/java/source-code/java-program-for-binary-search**](http://www.programmingsimplified.com/java/source-code/java-program-for-binary-search)

**import** java.util.Scanner;

**class** BinarySearch

{

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

{

**int** c, first, last, middle, n, search, array[];

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

System.out.println("Enter number of elements");

n = in.nextInt();

array = **new** **int**[n];

System.out.println("Enter " + n + " integers");

**for** (c = 0; c < n; c++)

array[c] = in.nextInt();

System.out.println("Enter value to find");

search = in.nextInt();

first = 0;

last = n - 1;

middle = (first + last)/2;

**while**( first <= last )

{

**if** ( array[middle] < search )

first = middle + 1;

**else** **if** ( array[middle] == search )

{

System.out.println(search + " found at location " + (middle + 1) + ".");

**break**;

}

**else**

last = middle - 1;

middle = (first + last)/2;

}

**if** ( first > last )

System.out.println(search + " is not present in the list.**\n**");

}

}

**The other program.**

int[] data;

2 int size;

3

4 public boolean binarySearch(int key)

5 {

6 int low = 0;

7 int high = size - 1;

8

9 while(high >= low) {

10 int middle = (low + high) / 2;

11 if(data[middle] == key) {

12 return true;

13 }

14 if(data[middle] < key) {

15 low = middle + 1;

16 }

17 if(data[middle] > key) {

18 high = middle - 1;

19 }

20 }

21 return false;

22 }

## How ConcurrentHashMap is implemented in Java

* Multiple partitions which can be locked independently. (16 by default)
* Using concurrent Locks operations for thread safety instead of synchronized.
* Has thread safe Iterators. synchronizedCollection's iterators are not thread safe.
* Does not expose the internal locks. synchronizedCollection does.

;

Uses HashEntry and Segments

ConcurrentHashMap is introduced as an alternative of Hashtable and provided all functions supported by Hashtable with additional feature called "concurrency level", which allows ConcurrentHashMap to partition Map. ConcurrentHashMap allows multiple readers to read concurrently without any [blocking](http://javarevisited.blogspot.com/2012/02/what-is-blocking-methods-in-java-and.html). This is achieved by partitioning Map into different parts based on concurrency level and locking only a portion of Map during updates. Default concurrency level is 16, and accordingly Map is divided into 16 part and each part is governed with different lock. This means, 16 thread can operate on Map simultaneously, until they are operating on different part of Map. This makes ConcurrentHashMap high performance despite keeping thread-safety intact.  Though, it comes with caveat. Since update operations like put(), remove(), putAll() or clear() is not [synchronized](http://javarevisited.blogspot.com/2011/04/synchronization-in-java-synchronized.html), **concurrent retrieval may not reflect most recent change on Map**.

In case of putAll() or clear(), which operates on whole Map, concurrent read may reflect insertion and removal of only some entries. Another important point to remember is iteration over CHM, [Iterator](http://javarevisited.blogspot.com/2011/10/java-iterator-tutorial-example-list.html) returned by keySet of ConcurrentHashMap are weekly consistent and they only reflect state of ConcurrentHashMap and certain point and may not reflect any recent change. Iterator of ConcurrentHashMap's keySet area also [fail-safe](http://javarevisited.blogspot.in/2012/02/fail-safe-vs-fail-fast-iterator-in-java.html) and doesn’t throw ConcurrentModificationExceptoin..

Default concurrency level is 16 and can be changed, by providing a number which make sense and work for you while creating ConcurrentHashMap. Since concurrency level is used for internal sizing and indicate number of concurrent update without contention, so, if you just have few writers or thread to update Map keeping it low is much better.ConcurrentHashMap also uses ReentrantLock to internally lock its segments.

Read more: <http://javarevisited.blogspot.com/2013/02/concurrenthashmap-in-java-example-tutorial-working.html#ixzz3PsrFdqfj>

***ConcurrentHashMap advantages summary***

* ConcurrentHashMap allows concurrent read and thread-safe update operation. All operations of ConcurrentHashMap are thread-safe.
* With update operation, ConcurrentHashMap only lock a specific part of Map instead of whole Map.
* Concurrent update is achieved by internally dividing Map into small segments which is defined using concurrency level.
* Iterator returned by ConcurrentHashMap is fail safe and never throw ConcurrentModificationException.
* ConcurrentHashMap doesn’t allow null as key or value.

Also

[**http://www.javatechtipssharedbygaurav.com/2013/12/how-concurrenthashmap-works-internally.html**](http://www.javatechtipssharedbygaurav.com/2013/12/how-concurrenthashmap-works-internally.html)

## Difference between synchronized method vs block in Java

<http://java67.blogspot.com/2013/01/difference-between-synchronized-block-vs-method-java-example.html>

Here are Some more differences between synchronized method and block in Java based upon experience and syntactical rules of synchronized keyword in Java. Though both block and method can be used to provide highest degree of synchronization in Java, use of synchronized block over method is considered as better [Java coding practices](http://javarevisited.blogspot.sg/2013/01/java-best-practices-method-overloading-constructor.html).

1) One significant difference between synchronized method and block is that, Synchronized block generally **reduce scope of lock**. As scope of lock is inversely proportional to performance, its always better to lock only critical section of code. One of the best example of using synchronized block is [double checked locking in Singleton pattern](http://javarevisited.blogspot.com/2012/07/why-enum-singleton-are-better-in-java.html) where instead of locking wholegetInstance() method we only lock critical section of code which is used to create Singleton instance. This improves performance drastically because locking is only required one or two times.

2) Synchronized block provide **granular control over lock**, as you can use arbitrary any lock to provide mutual exclusion to criticalsection code. On the other hand synchronized method always lock either on current object represented by [this keyword](http://javarevisited.blogspot.com/2012/01/this-keyword-java-example-tutorial.html) or class level lock, if its static synchronized method.

3) Synchronized block can throw throw [java.lang.NullPointerException](http://java67.blogspot.sg/2012/09/what-is-nullpointerexception-in-java.html) if expression provided to block as parameter evaluates to null, which is not the case with synchronized methods.

4) In case of synchronized method, lock is acquired by thread when it enter method and released when it leaves method, either normally or by throwing Exception. On the other hand in case of synchronized block, thread acquires lock when they enter synchronized block and release when they leave synchronized block.

**Synchronized method vs synchronized block Example in Java**

Here is an example of  sample class which shows on which object synchronized method and block are locked and how to use them :

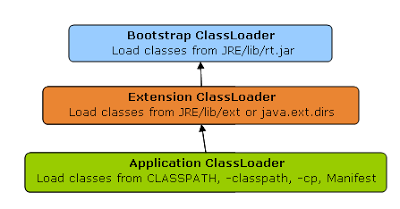
/\*\*  
  \* Java class to demonstrate use of synchronization method and block in Java  
  \*/  
**public** **class** SycnronizationExample{  
    
    
    **public** **synchronized** **void** lockedByThis(){  
        **System**.out.println(" This synchronized method is locked by current" instance of object i.e. this");  
    }  
    
    **public** **static** **synchronized** **void** lockedByClassLock(){  
        **System**.out.println("This static synchronized method is locked by class level lock of this class i.e. SychronizationExample.class");  
  
    }  
    
    **public** **void** lockedBySynchronizedBlock(){  
        **System**.err.println("This line is executed without locking");  
        
        **Object** obj = **String**.**class**; *//class level lock of Stirng class*  
        
        **synchronized**(obj){  
            **System**.out.println("synchronized block, locked by lock represented using obj variable");  
        }  
    }  
        
}

# What are rules of serialization in Java?

Here is a summary of some rules of Java serialization:

* An object is serializable only if its class or its superclass implements the Serializable(or  Externalizable) interface.
* An object is serializable (itself implements the Serializable interface) even if its superclass is not. However, the first superclass in the hierarchy of the serializable class, that does not implements Serializable interface, MUST have a no-arg constructor. If this is violated, readObject() will produce a java.io.InvalidClassException in runtime.
* The no-arg constructor of every non-serializable superclass will run when an object is deserialized. However, the deserialized objects? constructor does not run when it is deserialized.
* The class must be visible at the point of serialization.
* All primitive types are serializable.
* Transient fields (with transient modifier) are NOT serialized, (i.e., not saved or restored). A class that implements Serializable must mark transient fields of classes that do not support serialization (e.g., a file stream).
* Static fields (with static modifier) are Not serialized.
* If member vairiables of a serializable object reference to a non-serializable object, the code will compile but a RumtimeException will be thrown.

**What is ClassLoader in Java**

1. Class loader
2. ClassLoader in Java is a class which is used to load [class files in Java](http://javarevisited.blogspot.ca/2012/05/10-points-about-class-file-in-java.html). Java code is compiled into class file by [javac](http://javarevisited.blogspot.sg/2012/12/javac-is-not-recognized-as-internal-or-external-command.html)compiler and [JVM](http://javarevisited.blogspot.sg/2011/12/jre-jvm-jdk-jit-in-java-programming.html)executes Java program, by executing byte codes written in class file. ClassLoader is responsible for loading class files from file system, network or any other source. There are ddfthree default class loader used in Java, **Bootstrap** , **Extension** and **System or Application class loader**. Every class loader has a predefined location, from where they loads class files. Bootstrap ClassLoader is responsible for loading standard JDK class files from rt.jar and it is parent of all class loaders in Java. Bootstrap class loader don't have any parents, if you call String.class.getClassLoader() it will return null and any code based on that may throw [NullPointerException in Java](http://javarevisited.blogspot.com/2012/06/common-cause-of-javalangnullpointerexce.html). Bootstrap class loader is also known as**Primordial ClassLoader** in Java.  Extension ClassLoader delegates class loading request to its parent, Bootstrap and if unsuccessful, loads class form jre/lib/ext directory or any other directory pointed by java.ext.dirs system property. Extension ClassLoader in JVM is implemented by  sun.misc.Launcher$ExtClassLoader. Third default class loader used by JVM to load Java classes is called System or Application class loader and it is responsible for loading application specific classes from [CLASSPATH](http://javarevisited.blogspot.sg/2011/01/how-classpath-work-in-java.html) environment variable, -classpath or -cp command line option, Class-Path attribute of Manifest file inside JAR. Application class loader is a child of Extension ClassLoader and its implemented by sun.misc.Launcher$AppClassLoader class. Also, except Bootstrap class loader, which is implemented in native language mostly in C,  all  Java class loaders are implemented using java.lang.ClassLoader.
3. [](http://2.bp.blogspot.com/-HCTsr-j_ojw/USTOh1f8JwI/AAAAAAAAAjg/YegPspR5K48/s1600/java_classloader_hierarchy.PNG)
4. In short here is the location from which Bootstrap, Extension and Application ClassLoader load Class files.
5. 1) Bootstrap ClassLoader - JRE/lib/rt.jar
6. 2) Extension ClassLoader - JRE/lib/ext or any directory denoted by java.ext.dirs
7. 3) Application ClassLoader - CLASSPATH environment variable, -classpath or -cp option, Class-Path attribute of Manifest inside [JAR file](http://javarevisited.blogspot.sg/2012/03/how-to-create-and-execute-jar-file-in.html).
8. Read more: <http://javarevisited.blogspot.com/2012/12/how-classloader-works-in-java.html#ixzz3SPUIr0vb>

Bubble sort

public static void bubbleSort(int[] numArray) {

int n = numArray.length;

int temp = 0;

for (int i = 0; i < n; i++) {

for (int j = 1; j < (n - i); j++) {

if (numArray[j - 1] > numArray[j]) {

temp = numArray[j - 1];

numArray[j - 1] = numArray[j];

numArray[j] = temp;

}

}

}

}

What is difference between in the following statements

<http://stackoverflow.com/questions/874978/string-and-final>

String name = "Tiger";

final String name ="Tiger";

Although the String class is final class, why do we need to create a String "CONSTANT" variable as final?

final in this context means that the variable name can only be assigned once. Assigning a different String object to it again results in a compile error.

I think the source of the confusion here is that the final keyword can be used in several different contexts:

* final class: The class cannot be subclassed.
* final method: The method cannot be overridden.
* final variable: The variable can only be assigned once.

[**http://programmers.stackexchange.com/questions/250804/why-to-declare-a-string-as-final-and-then-use-it**](http://programmers.stackexchange.com/questions/250804/why-to-declare-a-string-as-final-and-then-use-it)

At runtime, it does not make a difference.

The point is readability - as a member variable, it likely is declared at the beginning of the class' source code, and making it static means it doesn't have to be allocated for each new instance of the class.

Making it final signifies to the reader that the value will not change (to the compiler too, but that's less important here).

This way, there are no "magic values" buried in the implementation, and if a change is desired to the "constant", it only needs to be changed in one place.

# C[omparing strings with == which are declared final in Java](http://stackoverflow.com/questions/19418427/comparing-strings-with-which-are-declared-final-in-java)

[**http://stackoverflow.com/questions/19418427/comparing-strings-with-which-are-declared-final-in-java**](http://stackoverflow.com/questions/19418427/comparing-strings-with-which-are-declared-final-in-java)

**Difference between Week Reference and Strong reference in java**

The only real difference between a soft reference and a weak reference is that the garbage collector uses algorithms to decide whether or not to reclaim a softly reachable object, but always reclaims a weakly reachable object.

**<http://stackoverflow.com/questions/299659/what-is-the-difference-between-a-soft-reference-and-a-weak-reference-in-java>**

**Weak Reference** <http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/ref/WeakReference.html>

**Principle:** weak reference is related to garbage collection. Normally, object having one or more reference will not be eligible for garbage collection.  
The above principle is not applicable when it is weak reference. If an object has only weak reference with other objects, then its ready for garbage collection.

Let's look at the below example: We have an Map with Objects where Key is reference a object.

import java.util.HashMap;

public class Test {

public static void main(String args[]) {

HashMap<Employee, EmployeeVal> aMap = new

HashMap<Employee, EmployeeVal>();

Employee emp = new Employee("Vinoth");

EmployeeVal val = new EmployeeVal("Programmer");

aMap.put(emp, val);

emp = null;

System.gc();

System.out.println("Size of Map" + aMap.size());

}

}

Now, during the execution of the program we have made emp = null. The Map holding the key makes no sense here as it is null. In the above situation, the object is not garbage collected.

**WeakHashMap**

WeakHashMap is one where the entries (key-to-value mappings) will be removed when it is no longer possible to retrieve them from the Map.

Let me show the above example same with **WeakHashMap**

import java.util.WeakHashMap;

public class Test {

public static void main(String args[]) {

WeakHashMap<Employee, EmployeeVal> aMap =

new WeakHashMap<Employee, EmployeeVal>();

Employee emp = new Employee("Vinoth");

EmployeeVal val = new EmployeeVal("Programmer");

aMap.put(emp, val);

emp = null;

System.gc();

int count = 0;

while (0 != aMap.size()) {

++count;

System.gc();

}

System.out.println("Took " + count

+ " calls to System.gc() to result in weakHashMap size of : "

+ aMap.size());

}

}

**Output:** Took 20 calls to System.gc() to result in aMap size of : 0.

WeakHashMap has only weak references to the keys, not strong references like other Map classes. There are situations which you have to take care when the value or key is strongly referenced though you have used WeakHashMap. This can avoided by wrapping the object in a **WeakReference**.

import java.lang.ref.WeakReference;

import java.util.HashMap;

public class Test {

public static void main(String args[]) {

HashMap<Employee, EmployeeVal> map =

new HashMap<Employee, EmployeeVal>();

WeakReference<HashMap<Employee, EmployeeVal>> aMap =

new WeakReference<HashMap<Employee, EmployeeVal>>(

map);

map = null;

while (null != aMap.get()) {

aMap.get().put(new Employee("Vinoth"),

new EmployeeVal("Programmer"));

System.out.println("Size of aMap " + aMap.get().size());

System.gc();

}

System.out.println("Its garbage collected");

}

}

# [Why set is not allowed duplicate value, which kind of mechanism used behind them?](http://stackoverflow.com/questions/20870879/why-set-is-not-allowed-duplicate-value-which-kind-of-mechanism-used-behind-them)

Internally **SET** store element using **HASHTABLE** ...**HASHTABLE** is a structure of Key value pairs..Here what the values passed by the **SET** is treated as Keys of **HASHTABLE** Internally. keys are unique cannot be duplicated. That is the reason if you pass any duplicate value it return false and does not added to the **SET** ...

If the adding element return true it will added into **SET**...Else it return False, that why it won't give any compilation or runtime error and it wont be added to **SET**

# What’s the difference between equals() and ==?

## The “==” operator compares the objects’ location(s) in memory

<http://www.programmerinterview.com/index.php/java-questions/java-whats-the-difference-between-equals-and/>

Are you confused? Well, let us explain further: as we mentioned earlier, the “==” operator is actually checking to see if the string objects (obj1 and obj2) refer to the exact same memory location. In other words, if both obj1 and obj2 are just different names for the **same** object then the “==” operator will return true when comparing the 2 objects. Another example will help clarify this:

String obj1 = new String("xyz");

// now obj2 and obj1 reference the same place in memory

String obj2 = obj1;

if(obj1 == obj2)

System.out.printlln("obj1==obj2 is TRUE");

else

System.out.println("obj1==obj2 is FALSE");

Note in the code above that obj2 and obj1 both reference the same place in memory because of this line: “String obj2 = obj1;”. And because the “==” compares the memory reference for each object, it will return true. And, the output of the code above will be:

obj1==obj2 is TRUE

## The equals() method

Now that we’ve gone over the “==” operator, let’s discuss the equals() method and how that compares to the “==” operator. The equals method is defined in the Object class, from which every class is either a direct or indirect descendant. By *default*, the equals() method actually behaves the same as the “==” operator – meaning it checks to see if both objects reference the same place in memory. But, the equals method is actually meant to compare the contents of 2 objects, and not their location in memory.

<https://www.sparkred.com/blog/open-source-java-static-code-analyzers/>

**FindBugs** is another static code analyzer very similar to PMD.  The biggest difference between PMD and FindBugs is that FindBugs works on byte code, whereas PMD works on source code. FindBugs can find things like:

– Improper use of .equals() and .hashCode()  
– Unsafe casts  
– When something will always be null  
– Possible StackOverflows  
– Possible ignored exceptions

There is a lot of overlap between FindBugs and PMD. Because of the limitations of working with byte code or source code, each excels in their own area. They compliment each other, but are not the same thing.

**PMD** is an extremely useful tool in analyzing source code. According to the project website, it ‘scans source code and looks for potential problems, possible bugs, unused and suboptimal code, over-complicated expressions and duplicate code’. PMD comes with a huge set of rules that can analyze many different things in java code. To name a few:

– Empty try/catch blocks

– Over-complicated expressions

– Using .equals() instead of ‘==’

– Unused variables and imports

– Unnecessary loops and if statements

– Enforce naming conventions

**Checkstyle** is a tool for analyzing coding style and conventions. It’s not going to stop any rouge exceptions, but it will give feedback on how the code is put together. Checkstyle is useful to ensure java code is being written right. Here are some things Checkstyle will catch:

– Missing/improper javadoc  
– Whitespace  
– Placement of braces and parentheses  
– Line length  
– Naming conventions

[**https://medium.freecodecamp.org/the-top-data-structures-you-should-know-for-your-next-coding-interview-36af0831f5e3**](https://medium.freecodecamp.org/the-top-data-structures-you-should-know-for-your-next-coding-interview-36af0831f5e3)

### Commonly used Data Structures

Let’s first list the most commonly used data structures, and then we’ll cover them one by one:

1. Arrays
2. Stacks
3. Queues
4. Linked Lists
5. [Trees](#tree)
6. Graphs
7. Tries (they are effectively trees, but it’s still good to call them out separately).
8. Hash Tables

**Trees**

[**https://medium.freecodecamp.org/all-you-need-to-know-about-tree-data-structures-bceacb85490c**](https://medium.freecodecamp.org/all-you-need-to-know-about-tree-data-structures-bceacb85490c)

### Terminology summary

* **Root**is the topmost node of the tree
* **Edge**is the link between two nodes
* **Child**is a node that has a parent node
* **Parent**is a node that has an edge to a child node
* **Leaf**is a node that does not have a child node in the tree
* **Height**is the length of the longest path to a leaf
* **Depth**is the length of the path to its root

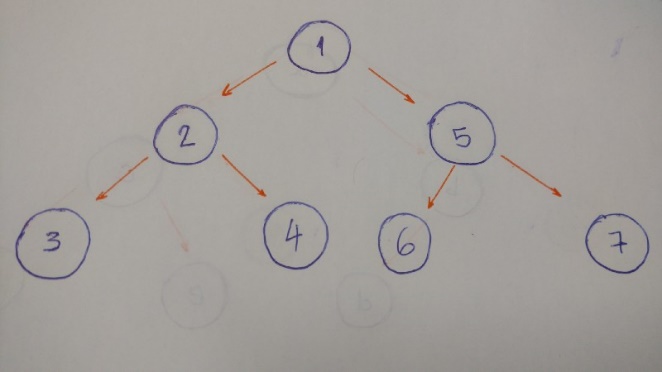
**Binary tree**

*in computer science, a binary tree is a tree data structure in which each node has at the most two children, which are referred to as the left child and the right child.”*

The first thing we need to keep in mind when we implement a binary tree is that it is a collection of nodes. Each node has three attributes: value, left\_child, and right\_child.

 think about tree traversal.

We have **two options** here: **Depth-First Search (DFS)** and **Breadth-First Search (BFS)**.



**DFS:**

When we go deep to the leaf and backtrack, this is called **DFS** algorithm.

**Pre order**

**1-2-3-4-5-6-7**

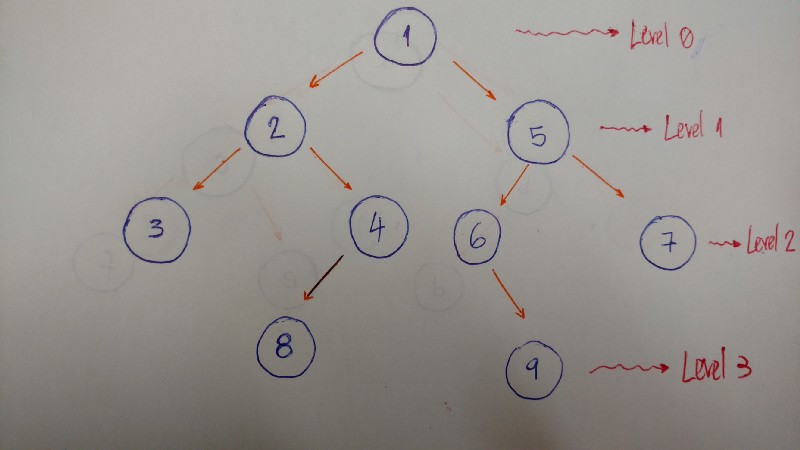
**In order**

**3-2-4-1-6-5-7**

**post order**

**3–4–2–6–7–5–1.**

**BFS**



To implement a **BFS** algorithm, we use the queue data structure to help.

**1 2 5 3** 4 6 7 8 7

**JAVA SEARCH ALGORITHMS**

**<http://www.vogella.com/tutorials/JavaAlgorithmsSearch/article.html>**

* 1. **Sequential SEARCH:**

Given a collection you try every element in the collection until you have found the element or until you reach the end of the collection

* 1. Binary search

Binary search requires that the collection is already sorted. For example by [Quicksort](http://www.vogella.com/tutorials/JavaAlgorithmsQuicksort/article.html) or [Mergesort](http://www.vogella.com/tutorials/JavaAlgorithmsMergesort/article.html). Binary search checks the element in the middle of the collection. If the search element is smaller or greater than the found element, then a sub-array is defined which is then searched again. If the searched element is smaller than the found element, then the sub-array is searched from the start of the array until the found element. If the searched element is larger than the found element, then the sub-array is searched from the found element until the end of the array. Once the searched element is found or the collection is empty then the search is over

**public** **class** BinarySearch {

**public** **static** **boolean** contains(**int**[] a, **int** b) {

**if** (a.length == 0) {

**return** **false**;

}

**int** low = 0;

**int** high = a.length-1;

**while**(low <= high ) {

**int** middle = (low+high) /2;

System.***out***.println("middle"+middle);

**if** (b> a[middle] ){

low = middle +1;

} **else** **if** (b< a[middle]){

high = middle -1;

} **else** { // The element has been found

**return** **true**;

}

}

**return** **false**;

}

}

<https://www.geeksforgeeks.org/searching-algorithms/>



**Methods In Object class**

public toString()

hasCode()

equals()

**finalize()** –This method is called just before an object is garbage collected

**clone()**

The remaining three methods **wait()**, **notify()** **notifyAll()** are related to Concurrency. Refer [Inter-thread Communication in Java](https://www.geeksforgeeks.org/inter-thread-communication-java/)for details.

**Sorting Algorithms**

**Mergesort** is up there with the fastest standard sort algorithms. The default Collections.sort() implementation in Java 7 is a **Mergesort** algorithm adapted from 'TimSort.' The default **Arrays.**sort() implementation in Java 7 is a dual pivot**Quicksort**.

**<https://www.geeksforgeeks.org/sorting-algorithms/>**

* [Selection Sort](https://www.geeksforgeeks.org/selection-sort/)
* [Bubble Sort](https://www.geeksforgeeks.org/bubble-sort/)
* [Recursive Bubble Sort](https://www.geeksforgeeks.org/recursive-bubble-sort/)
* [Insertion Sort](https://www.geeksforgeeks.org/insertion-sort/)
* [Recursive Insertion Sort](https://www.geeksforgeeks.org/recursive-insertion-sort/)
* [Merge Sort](https://www.geeksforgeeks.org/merge-sort/)
* [Iterative Merge Sort](https://www.geeksforgeeks.org/iterative-merge-sort/)
* [Quick Sort](https://www.geeksforgeeks.org/quick-sort/)
* [Iterative Quick Sort](https://www.geeksforgeeks.org/iterative-quick-sort/)
* [Heap Sort](https://www.geeksforgeeks.org/heap-sort/)
* [Counting Sort](https://www.geeksforgeeks.org/counting-sort/)
* [Radix Sort](https://www.geeksforgeeks.org/radix-sort/)
* [Bucket Sort](https://www.geeksforgeeks.org/bucket-sort-2/)
* [ShellSort](https://www.geeksforgeeks.org/shellsort/)
* [TimSort](https://www.geeksforgeeks.org/timsort/)
* [Comb Sort](https://www.geeksforgeeks.org/comb-sort/)
* [Pigeonhole Sort](https://www.geeksforgeeks.org/pigeonhole-sort/)
* [Cycle Sort](https://www.geeksforgeeks.org/cycle-sort/)
* [Cocktail Sort](https://www.geeksforgeeks.org/cocktail-sort/)
* [Strand Sort](https://www.geeksforgeeks.org/strand-sort/)
* [Bitonic Sort](https://www.geeksforgeeks.org/bitonic-sort/)
* [Pancake sorting](https://www.geeksforgeeks.org/pancake-sorting/)
* [Binary Insertion Sort](https://www.geeksforgeeks.org/binary-insertion-sort/)
* [BogoSort or Permutation Sort](https://www.geeksforgeeks.org/bogosort-permutation-sort/)
* [Gnome Sort](https://www.geeksforgeeks.org/gnome-sort-a-stupid-one/)
* [Sleep Sort – The King of Laziness / Sorting while Sleeping](https://www.geeksforgeeks.org/sleep-sort-king-laziness-sorting-sleeping/)
* [Structure Sorting (By Multiple Rules) in C++](https://www.geeksforgeeks.org/structure-sorting-in-c/)
* [Stooge Sort](https://www.geeksforgeeks.org/stooge-sort/)
* [Tag Sort (To get both sorted and original)](https://www.geeksforgeeks.org/tag-sort/)
* [Tree Sort](https://www.geeksforgeeks.org/tree-sort/)
* [Cartesian Tree Sorting](https://www.geeksforgeeks.org/cartesian-tree-sorting/)
* [Odd-Even Sort / Brick Sort](https://www.geeksforgeeks.org/odd-even-sort-brick-sort/)
* [QuickSort on Singly Linked List](https://www.geeksforgeeks.org/quicksort-on-singly-linked-list/)
* [QuickSort on Doubly Linked List](https://www.geeksforgeeks.org/quicksort-for-linked-list/)
* [3-Way QuickSort (Dutch National Flag)](https://www.geeksforgeeks.org/3-way-quicksort-dutch-national-flag/)
* [Merge Sort for Linked Lists](https://www.geeksforgeeks.org/merge-sort-for-linked-list/)
* [Merge Sort for Doubly Linked List](https://www.geeksforgeeks.org/merge-sort-for-doubly-linked-list/)
* [3-way Merge Sort](https://www.geeksforgeeks.org/3-way-merge-sort/)

# Merge Sort

Like [QuickSort](https://www.geeksforgeeks.org/quick-sort/), Merge Sort is a [Divide and Conquer](https://www.geeksforgeeks.org/divide-and-conquer-introduction/) algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. **The merge () function** is used for merging two halves. The merge(arr, l, m, r) is key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one. See following C implementation for details.

**Data Types in java (there are 8)**

Primitive data are only single values; they have no special capabilities. There are 8 primitive data types

primitive data types  
[](https://cdncontribute.geeksforgeeks.org/wp-content/uploads/data-types-in-java.jpg)

1. SHALLOW COPY VS DEEP COPY IN JAVA

https://www.geeksforgeeks.org/deep-shallow-lazy-copy-java-examples/

A call to super.clone() performs a shallow copy where all the fields values of the original object are copied to the new object. If a field value is a primitive type, a shallow copy copies the value of the primitive type. But, if a field value is a reference type, then only the reference is copied, and not the referred object itself. Therefore, both the original and its clone refer to the same object and if either one modifies the referred object, the modification will be visible to the other. This might result in unexpected behavior in an application. In such situation, you should perform a deep copy that makes copies of dynamically allocated memory pointed to by the reference type fields. In a deep copy, the original and the copied objects are independent of each other and therefore the objects can update their own fields without worrying about any referencing problems.

