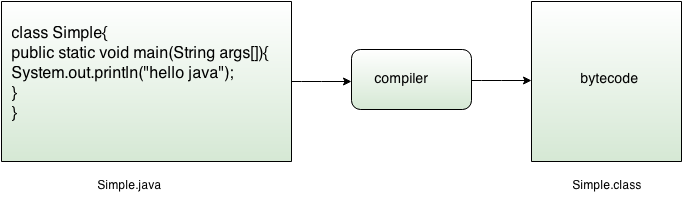
# Java Basics

## 1. Difference between Javac and java

**What happens at compile time?**

At compile time, java file is compiled by Java Compiler (It does not interact with OS) and converts the java code into bytecode.

**What happens at runtime?**

|  |
| --- |
| At runtime, following steps are performed: |
| what happens at runtime when simple java program runs |

|  |
| --- |
| **Classloader:**is the subsystem of JVM that is used to load class files. |
| **Bytecode Verifier:**checks the code fragments for illegal code that can violate access right to objects. |
| **Interpreter:**read bytecode stream then execute the instructions. |

Q1 .**Can you save a java source file by other name than the class name?**

-> Yes, if the class is not public. It is explained in the figure given below:

**Q2. Can you have multiple classes in a java source file?**

**-> Yes we can have**

# Difference between JDK, JRE and JVM

1. [Brief summary of JVM](http://www.javatpoint.com/difference-between-jdk-jre-and-jvm)
2. [Java Runtime Environment (JRE)](http://www.javatpoint.com/difference-between-jdk-jre-and-jvm#jre)
3. [Java Development Kit (JDK)](http://www.javatpoint.com/difference-between-jdk-jre-and-jvm#jdk)

Understanding the difference between JDK, JRE and JVM is important in Java. We are having brief overview of JVM here.

If you want to get the detailed knowledge of Java Virtual Machine, move to the next page. Firstly, let's see the basic differences between the JDK, JRE and JVM.

### JVM

JVM (Java Virtual Machine) is an abstract machine. It is a specification that provides runtime environment in which java bytecode can be executed.

JVMs are available for many hardware and software platforms. JVM, JRE and JDK are platform dependent because configuration of each OS differs. But, Java is platform independent.

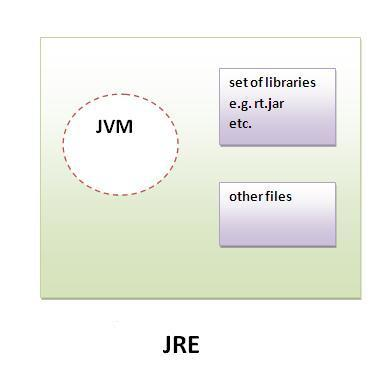
The JVM performs following main tasks:

* Loads code
* Verifies code
* Executes code
* Provides runtime environment

### JRE

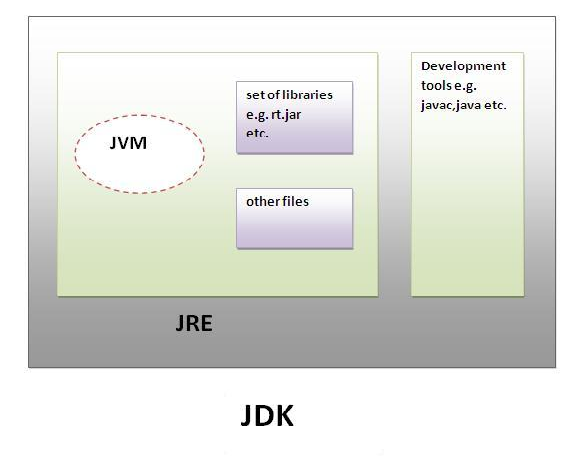
JRE is an acronym for Java Runtime Environment.It is used to provide runtime environment.It is the implementation of JVM. It physically exists. It contains set of libraries + other files that JVM uses at runtime.

Implementation of JVMs are also actively released by other companies besides Sun Micro Systems.



### JDK

JDK is an acronym for Java Development Kit.It physically exists.It contains JRE + development tools.



**JVM (Java Virtual Machine)**

1. [Java Virtual Machine](http://www.javatpoint.com/internal-details-of-jvm)
2. [Internal Architecture of JVM](http://www.javatpoint.com/internal-details-of-jvm#jvminternalarch)

JVM (Java Virtual Machine) is an abstract machine. It is a specification that provides runtime environment in which java bytecode can be executed.

JVMs are available for many hardware and software platforms (i.e. JVM is platform dependent).

### What is JVM

It is:

1. **A specification** where working of Java Virtual Machine is specified. But implementation provider is independent to choose the algorithm. Its implementation has been provided by Sun and other companies.
2. **An implementation** Its implementation is known as JRE (Java Runtime Environment).
3. **Runtime Instance** Whenever you write java command on the command prompt to run the java class, an instance of JVM is created.

### What it does

The JVM performs following operation:

* Loads code
* Verifies code
* Executes code
* Provides runtime environment

JVM provides definitions for the:

* Memory area
* Class file format
* Register set
* Garbage-collected heap
* Fatal error reporting etc.

### Internal Architecture of JVM

|  |
| --- |
| Let's understand the internal architecture of JVM. It contains classloader, memory area, execution engine etc. |

### 1) Classloader

Classloader is a subsystem of JVM that is used to load class files.

### 2) Class(Method) Area

Class(Method) Area stores per-class structures such as the runtime constant pool, field and method data, the code for methods.

### 3) Heap

It is the runtime data area in which objects are allocated.

### 4) Stack

|  |
| --- |
| Java Stack stores frames.It holds local variables and partial results, and plays a part in method invocation and return. |
| Each thread has a private JVM stack, created at the same time as thread. |
| A new frame is created each time a method is invoked. A frame is destroyed when its method invocation completes. |

### 5) Program Counter Register

PC (program counter) register. It contains the address of the Java virtual machine instruction currently being executed.

### 6) Native Method Stack

It contains all the native methods used in the application.

### 7) Execution Engine

|  |
| --- |
| It contains: |
| **1) A virtual processor** |
| **2) Interpreter:** Read bytecode stream then execute the instructions. |

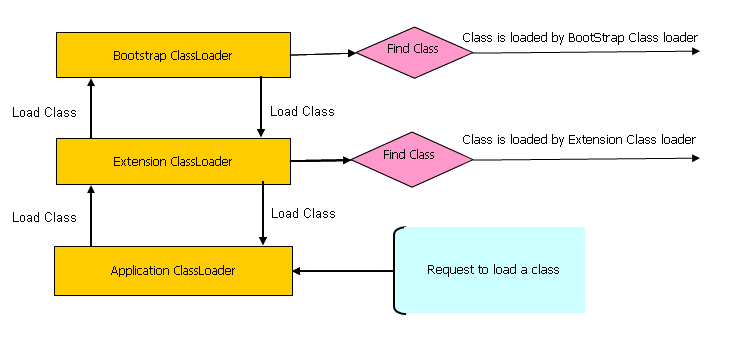
--------------CLASS LOADER--------------------------------

Java class loaders are used **to load class at run time**.

**How Class Loader Works in Java**

Class loader in java works in three principle:-

1. **Delegation** :- Delegation principle forward request of class loading to parent class loader and only loads the class, if parent is not able to find or load class
2. **Visibility:-** According to visibility principle, Child ClassLoader can see class loaded by Parent ClassLoader but vice-versa is not true. Which mean if class Abc is loaded by Application class loader than trying to load class ABC explicitly using extension ClassLoader will throw either [java.lang.ClassNotFoundException](http://javarevisited.blogspot.ca/2011/08/classnotfoundexception-in-java-example.html)
3. **Uniqueness:-** Uniqueness principle allows to load a class exactly once, which is basically achieved by delegation and ensures that child ClassLoader doesn't reload the class already loaded by parent



Can one class be loaded by two different ClassLoader in Java ?

**What is CLASSPATH in Java**  
Classpath in Java is the path to directory or list of the directory which is used by ClassLoaders to find and load class in Java program. Classpath can be specified using CLASSPATH environment variable which is case insensitive, -cp or -classpath command line option or Class-Path attribute in manifest.mf file inside JAR file in Java  
  
**Errors related to Classpath in Java**

**1.ClassNotFoundException** : is an Exception and will be thrown when Java

Program dynamically tries to load a Java class at Runtime and don’t find the corresponding class file on the classpath. Two keywords here “dynamically” and “runtime”. A classic example of these errors is whey you try to load JDBC driver by using Class.forname(“driver name”) and greeted with[java.lang.ClassNotFoundException: com.mysql.jdbc.Driver](http://javarevisited.blogspot.sg/2012/03/jdbc-javalangclassnotfoundexception.html). So this error essentially comes when Java try to load a class using forName() or by loadClass() method of ClassLoader. The key thing to note is that presence of that class on Java classpath is not checked on compile time. So even if those classes are not present on Java classpath your program will compile successfully and only fail when you try to run.

2. **NoClassDefFoundError:** On the other hand, NoClassDefFoundError is

an Error and more critical than ClassNotFoundException which is an exception and recoverable. NoClassDefFoundError comes when a particular class was present in Java Classpath during compile time but not available during run-time. A classic example of this error is using log4j.jar for logging purpose and forgot to include log4j.jar on the classpath in java during run-time.

**Summary of CLASSPATH in Java**

1.      **Classpath i**n Java is **an environment variable** used by Java Virtual machine to locate or find  [class files in Java](http://java67.blogspot.sg/2012/08/what-is-class-file-in-java-how-to-create-class.html) during class loading.

2.      You can **override the value of Classpath in Java** defined by environment variable CLASSPATH by providing **JVM command line option –cp or –classpath** while running your application.

3.      If two classes with the same name exist in Java Classpath then **the class which comes earlier in Classpath** will be picked by Java Virtual Machine.

4.      **By default CLASSPATH in Java points to current directory denoted by "."** and it will look for any class only in the current directory.

5.      When you use the **-jar command line  option** to [run your program as an executable JAR](http://javarevisited.blogspot.sg/2012/03/how-to-create-and-execute-jar-file-in.html), then the Java **CLASSPATH environment variable will be ignored**, and also the **-cp and -classpath switches will be ignored** and, In this case, you can set your java classpath in the *META-INF/MANIFEST.MF file by using the Class-Path attribute*.

6.      **In Unix of Linux, Java Classpath** contains names of the directory with **colon “:”** separated, On Windows Java Classpath will be  **semicolon “;”** separated while if you defined java classpath in Manifest file those will be **space** separated.

7.       You can check value of classpath in java inside your application by looking at following system property **“java.class.path**”  System.getProperty("java.class.path")  
  
**Class-Path attribute is used to contain classpath inside manifest file**. Also, make sure that your manifest file must end with a blank line (carriage return or new line), here is an example of java classpath in the manifest file.  
  
**Main-Class: com.classpathexample.Demo\_Classpath  
Class-Path: lib/tibco.jar lib/log4j.jar**

8.       It’s also important to note that **path specified in the manifest file is not absolute** instead **they are relative from application jar’s path**. For example, in above if your application jar file is in C:\test directory you must need a lib directory inside test and tibco.jar and log4j.jar inside that.

9.       **ClassNotFoundException**is an Exception and will be thrown when Java program dynamically tries to load a particular Class at Runtime and don’t find that on Java classpath due to result of Class.forName() or loadClass() method invocation.

10. **NoClassDefFoundError**comes when a particular class was present in Java Classpath during compile time but not available during runtime on Classpath in Java.

**What is ClassLoader in Java:**

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 three default class loader used in Java

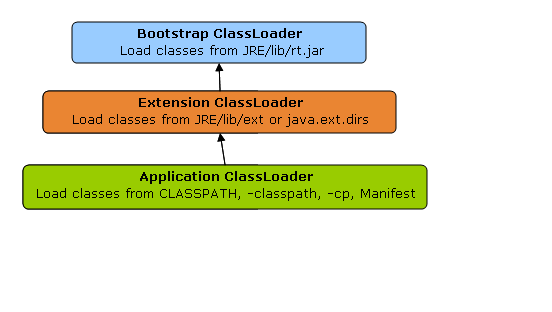
1. **Bootstrap** :- is responsible for loading standard JDK class files from rt.jar and it is parent of all class loaders in Java. Bootstrap class loader is also known as**Primordial ClassLoader** in Java
2. **Extension** : Bootstrap class loader don't have any parents, if you call String.class.getClassLoader() 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)
3. and **System or Application class loader** : 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.

In short here is the location from which Bootstrap, Extension and Application ClassLoader load Class files.

1) Bootstrap ClassLoader - JRE/lib/rt.jar

2) Extension ClassLoader - JRE/lib/ext or any directory denoted by java.ext.dirs

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).



## How to load class explicitly in Java

Java provides API to explicitly load a class by Class.forName(classname) and Class.forName(classname, initialized, classloader), remember JDBC code which is used to load JDBC drives we have seen in [Java program to Connect Oracle database](http://javarevisited.blogspot.ca/2012/04/java-program-to-connect-oracle-database.html). As shown in above example you can pass name of ClassLoader which should be used to load that particular class along with binary name of class. Class is loaded by calling loadClass() method of java.lang.ClassLoader class which calls findClass() method to locate bytecodes for corresponding class. In this example Extension ClassLoader uses java.net.URLClassLoader which search for class files and resources in [JAR](http://javarevisited.blogspot.ca/2012/10/5-ways-to-add-multiple-jar-to-classpath-java.html) and directories. any search path which is ended using "/" is considered directory. If findClass() does not found the class than it throws[java.lang.ClassNotFoundException](http://javarevisited.blogspot.de/2012/03/jdbc-javalangclassnotfoundexception.html) and if it finds it calls defineClass() to convert bytecodes into a .class instance which is returned to the caller.

Read more: <http://javarevisited.blogspot.com/2012/12/how-classloader-works-in-java.html#ixzz4CNP0BRiA>

## JVM) Memory Model – Memory Management in Java :

## Java (JVM) Memory Model : The below diagram show JVM memory model.  JVM Heap memory is physically divided into two parts – **Young Generation** and **Old Generation**.

## C:\Users\prasanna\Downloads\Java-Memory-Model.png

**Young Generation:**

Young generation is the place where all the new objects are created. When young generation is filled, garbage collection is performed. This garbage collection is called **Minor GC**. Young Generation is divided into three parts – **Eden Memory** and two **Survivor Memory** spaces.

Important Points about Young Generation Spaces:

* Most of the newly created objects are located in the Eden memory space.
* When Eden space is filled with objects, Minor GC is performed and all the survivor objects are moved to one of the survivor spaces.
* Minor GC also checks the survivor objects and move them to the other survivor space. So at a time, one of the survivor space is always empty.
* Objects that are survived after many cycles of GC, are moved to the Old generation memory space. Usually it’s done by setting a threshold for the age of the young generation objects before they become eligible to promote to Old generation.

## Old Generation :

Old Generation memory contains the objects that are long lived and survived after many rounds of Minor GC. Usually garbage collection is performed in Old Generation memory when it’s full. Old Generation Garbage Collection is called **Major GC** and usually takes longer time.

### Stop the World Event:

All the Garbage Collections are “Stop the World” events because all application threads are stopped until the operation completes.

Since Young generation keeps short-lived objects, Minor GC is very fast and the application doesn’t get affected by this.

However Major GC takes longer time because it checks all the live objects. Major GC should be minimized because it will make your application unresponsive for the garbage collection duration. So if you have a responsive application and there are a lot of Major Garbage Collection happening, you will notice timeout errors.

The duration taken by garbage collector depends on the strategy used for garbage collection. That’s why it’s necessary to monitor and tune the garbage collector to avoid timeouts in the highly responsive applications.

**Permanent Generation (PremGen) :**

Permanent Generation or “Perm Gen” contains the application metadata required by the JVM to describe the classes and methods used in the application. Note that Perm Gen is not part of Java Heap memory.

Perm Gen is populated by JVM at runtime based on the classes used by the application. Perm Gen also contains Java SE library classes and methods. Perm Gen objects are garbage collected in a full garbage collection.

### Method Area:

Method Area is part of space in the Perm Gen and used to store class structure (runtime constants and static variables) and code for methods and constructors.

### Memory Pool:

Memory Pools are created by JVM memory managers to create a pool of immutable objects, if implementation supports it. String Pool is a good example of this kind of memory pool. Memory Pool can belong to Heap or Perm Gen, depending on the JVM memory manager implementation.

### Runtime Constant Pool:

Runtime constant pool is per-class runtime representation of constant pool in a class. It contains class runtime constants and static methods. Runtime constant pool is the part of method area.

### Java Stack Memory:

Java Stack memory is used for execution of a thread. They contain method specific values that are short-lived and references to other objects in the heap that are getting referred from the method. You should read [Difference between Stack and Heap Memory](http://www.journaldev.com/4098/java-heap-space-vs-stack-memory).

### Java Heap Memory Switches:

Java provides a lot of memory switches that we can use to set the memory sizes and their ratios. Some of the commonly used memory switches are:

|  |  |
| --- | --- |
| VM SWITCH | VM SWITCH DESCRIPTION |
| -Xms | For setting the initial heap size when JVM starts |
| -Xmx | For setting the maximum heap size. |
| -Xmn | For setting the size of the Young Generation, rest of the space goes for Old Generation. |
| -XX:PermGen | For setting the initial size of the Permanent Generation memory |
| -XX:MaxPermGen | For setting the maximum size of Perm Gen |
| -XX:SurvivorRatio | For providing ratio of Eden space and Survivor Space, for example if Young Generation size is 10m and VM switch is -XX:SurvivorRatio=2 then 5m will be reserved for Eden Space and 2.5m each for both the Survivor spaces. The default value is 8. |
| -XX:NewRatio | For providing ratio of old/new generation sizes. The default value is 2. |

Most of the times, above options are sufficient, but if you want to check out other options too then please check [JVM Options Official Page](http://www.oracle.com/technetwork/java/javase/tech/vmoptions-jsp-140102.html).

### Java Garbage Collection:

Java Garbage Collection is the process to identify and remove the unused objects from the memory and free space to be allocated to objects created in the future processing. One of the best feature of java programming language is the **automatic garbage collection**, unlike other programming languages such as C where memory allocation and deallocation is a manual process.

**Garbage Collector** is the program running in the background that looks into all the objects in the memory and find out objects that are not referenced by any part of the program. All these unreferenced objects are deleted and space is reclaimed for allocation to other objects.

One of the basic way of garbage collection involves three steps:

1. **Marking**: This is the first step where garbage collector identifies which objects are in use and which ones are not in use.
2. **Normal Deletion**: Garbage Collector removes the unused objects and reclaim the free space to be allocated to other objects.
3. **Deletion with Compacting**: For better performance, after deleting unused objects, all the survived objects can be moved to be together. This will increase the performance of allocation of memory to newer objects.

There are two problems with simple mark and delete approach.

1. First one is that it’s not efficient because most of the newly created objects will become unused
2. Secondly objects that are in-use for multiple garbage collection cycle are most likely to be in-use for future cycles too.

The above shortcomings with the simple approach is the reason that **Java Garbage Collection is Generational** and we have **Young Generation** and **Old Generation** spaces in the heap memory. I have already explained above how objects are scanned and moved from one generational space to another based on the Minor GC and Major GC.

### Java Garbage Collection Types

There are five types of garbage collection types that we can use in our applications. We just need to use JVM switch to enable the garbage collection strategy for the application. Let’s look at each of them one by one.

1. **Serial GC (-XX:+UseSerialGC)**: Serial GC uses the simple **mark-sweep-compact** approach for young and old generations garbage collection i.e Minor and Major GC. Serial GC is useful in client-machines such as our simple stand alone applications and machines with smaller CPU. It is good for small applications with low memory footprint.
2. **Parallel GC (-XX:+UseParallelGC)**: Parallel GC is same as Serial GC except that is spawns N threads for young generation garbage collection where N is the number of CPU cores in the system. We can control the number of threads using -XX:ParallelGCThreads=n JVM option. Parallel Garbage Collector is also called throughput collector because it uses multiple CPUs to speed up the GC performance. Parallel GC uses single thread for Old Generation garbage collection.
3. **Parallel Old GC (-XX:+UseParallelOldGC)**: This is same as Parallel GC except that it uses multiple threads for both Young Generation and Old Generation garbage collection.
4. **Concurrent Mark Sweep (CMS) Collector (-XX:+UseConcMarkSweepGC)**: CMS Collector is also referred as concurrent low pause collector. It does the garbage collection for Old generation. CMS collector tries to minimize the pauses due to garbage collection by doing most of the garbage collection work concurrently with the application threads.

CMS collector on young generation uses the same algorithm as that of the parallel collector. This garbage collector is suitable for responsive applications where we can’t afford longer pause times. We can limit the number of threads in CMS collector using -XX:ParallelCMSThreads=n JVM option.

1. **G1 Garbage Collector (-XX:+UseG1GC)**: The Garbage First or G1 garbage collector is available from Java 7 and it’s long term goal is to replace the CMS collector. The G1 collector is a parallel, concurrent, and incrementally compacting low-pause garbage collector.

Garbage First Collector doesn’t work like other collectors and there is no concept of Young and Old generation space. It divides the heap space into multiple equal-sized heap regions. When a garbage collection is invoked, it first collects the region with lesser live data, hence “Garbage First”. You can find more details about it at [Garbage-First Collector Oracle Documentation](http://docs.oracle.com/javase/7/docs/technotes/guides/vm/G1.html).

## Garbage Collection :

Garbage collection is the process of freeing space in the heap or the nursery for allocation of new objects. This section describes the garbage collection in the JRockit JVM.

* + [The Mark and Sweep Model](https://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/garbage_collect.html" \l "wp1085786)
  + [Generational Garbage Collection](https://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/garbage_collect.html" \l "wp1086786)
  + [Dynamic and Static Garbage Collection Modes](https://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/garbage_collect.html" \l "wp1086732)
  + [Compaction](https://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/garbage_collect.html" \l "wp1086917)

### The Mark and Sweep Model :

The JRockit JVM uses the mark and sweep garbage collection model for performing garbage collections of the whole heap. A mark and sweep garbage collection consists of two phases, the mark phase and the sweep phase.

During the mark phase all objects that are reachable from Java threads, native handles and other root sources are marked as alive, as well as the objects that are reachable from these objects and so forth. This process identifies and marks all objects that are still used, and the rest can be considered garbage.

During the sweep phase the heap is traversed to find the gaps between the live objects. These gaps are recorded in a free listand are made available for new object allocation.

The JRockit JVM uses two improved versions of the mark and sweep model. One is mostly concurrent mark and sweep and the other is parallel mark and sweep. You can also mix the two strategies, running for example mostly concurrent mark and parallel sweep.

#### Mostly Concurrent Mark and Sweep:

*The*mostly concurrent mark and sweep strategy*(often simply called*concurrent garbage collection*)* allows the Java threads to continue running during large portions of the garbage collection. The threads must however be stopped a few times for synchronization.

The mostly **concurrent mark** phase is divided into four parts:

* + **Initial marking**, where the root set of live objects is identified. This is done while the Java threads are paused.
  + **Concurrent marking**, where the references from the root set are followed in order to find and mark the rest of the live objects in the heap. This is done while the Java threads are running.
  + **Precleaning**, where changes in the heap during the concurrent mark phase are identified and any additional live objects are found and marked. This is done while the Java threads are running.
  + **Final marking**, where changes during the precleaning phase are identified and any additional live objects are found and marked. This is done while the Java threads are paused.

The mostly **concurrent sweep** phase consists of four parts:

* + **Sweeping of one half of the heap**. This is done while the Java threads are running and are allowed to allocate objects in the part of the heap that isn’t currently being swept.
  + **A short pause to switch halves.**
  + **Sweeping of the other half of the heap**. This is done while the Java threads are running and are allowed to allocate objects in the part of the heap that was swept first.
  + A short pause for synchronization and recording statistics.

#### Parallel Mark and Sweep :

The parallel mark and sweep strategy (also called the parallel garbage collector) uses all available CPUs in the system for performing the garbage collection as fast as possible. All Java threads are paused during the entire parallel garbage collection.

### Generational Garbage Collection:

The nursery, when it exists, is garbage collected with a special garbage collection called a young collection. A garbage collection strategy which uses a nursery is called a generational garbage collection strategy, or simplygenerational garbage collection.

The young collector used in the JRockit JVM identifies and promotes all live objects in the nursery that are outside the keep area to the old space. This work is done in parallel using all available CPUs. The Java threads are paused during the entire young collection.

### Dynamic and Static Garbage Collection Modes:

By default, the JRockit JVM uses a dynamic garbage collection mode that automatically selects a garbage collection strategy to use, aiming at optimizing the application throughput. You can also choose between two other dynamic garbage collection modes or select the garbage collection strategy statically. The following dynamic modes are available:

* + ***throughput***, which optimizes the garbage collector for maximum application throughput. This is the default mode.
  + ***pausetime***, which optimizes the garbage collector for short and even pause times.
  + ***deterministic***, which optimizes the garbage collector for very short and deterministic pause times. This mode is only available as a part of Oracle JRockit Real Time.

The major static strategies are:

* + ***singlepar***, which is a single-generational parallel garbage collector (same as parallel)
  + ***genpar***, which is a two-generational parallel garbage collector
  + ***singlecon***, which is a single-generational mostly concurrent garbage collector
  + ***gencon***, which is a two-generational mostly concurrent garbage collector

For more information on how to select the best mode or strategy for your application, see [Selecting and Tuning a Garbage Collector](https://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/memman.html#wp1087125).

### Compaction:

Objects that are allocated next to each other will not necessarily become unreachable (“die”) at the same time. This means that the heap may become fragmented after a garbage collection, so that the free spaces in the heap are many but small, making allocation of large objects hard or even impossible. Free spaces that are smaller than the minimum thread local area (TLA) size can not be used at all, and the garbage collector discards them as dark matter until a future garbage collection frees enough space next to them to create a space large enough for a TLA.

To reduce fragmentation, the JRockit JVM compacts a part of the heap at every garbage collection (old collection). Compaction moves objects closer together and further down in the heap, thus creating larger free areas near the top of the heap. The size and position of the compaction area as well as the compaction method is selected by advanced heuristics, depending on the garbage collection mode used.

Compaction is performed at the beginning of or during the sweep phase and while all Java threads are paused.

For information on how to tune compaction, see [Tuning the Compaction of Memory](https://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/memman.html#wp1088193).

#### External and Internal Compaction

The JRockit JVM uses two compaction methods called external compaction and internal compaction. External compaction moves (evacuates) the objects within the compaction area to free positions outside the compaction area and as far down in the heap as possible. Internal compaction moves the objects within the compaction area as far down in the compaction area as possible, thus moving them closer together.

The JVM selects a compaction method depending on the current garbage collection mode and the position of the compaction area. External compaction is typically used near the top of the heap, while internal compaction is used near the bottom where the density of objects is higher.

#### Sliding Window Schemes :

The position of the compaction area changes at each garbage collection, using one or two sliding windows to determine the next position. Each sliding window moves a notch up or down in the heap at each garbage collection, until it reaches the other end of the heap or meets a sliding window that moves in the opposite direction, and starts over again. Thus the whole heap is eventually traversed by compaction over and over again.

#### Compaction Area Sizing:

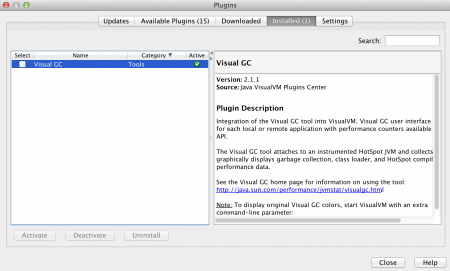
The size of the compaction area depends on the garbage collection mode used. In throughput mode the compaction area size is static, while all other modes, including the static mode, adjust the compaction area size depending on the compaction area position, aiming at keeping the compaction times equal throughout the run. The compaction time depends on the number of objects moved and the number of references to these objects. Thus the compaction area will be smaller in parts of the heap where the object density is high or where the amount of references to the objects within the area is high. Typically the object density is higher near the bottom of the heap than at the top of the heap, except at the very top where the latest allocated objects are found. Thus the compaction areas are usually smaller near the bottom of the heap than in the top half of the heap.

### Memory Management in Java – Java Garbage Collection Monitoring

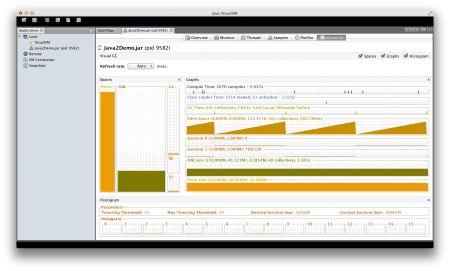
#### Java VisualVM with Visual GC:

If you want to see memory and GC operations in GUI, then you can use jvisualvm tool. Java VisualVM is also part of JDK, so you don’t need to download it separately.

Just run jvisualvm command in the terminal to launch the Java VisualVM application. Once launched, you need to install **Visual GC** plugin from Tools -< Plugins option, as shown in below image.

[](http://cdn.journaldev.com/wp-content/uploads/2014/05/VisualVM-Visual-GC-Plugin.png)

After installing **Visual GC**, just open the application from the left side column and head over to **Visual GC**section. You will get an image of JVM memory and garbage collection details as shown in below image.

[](http://cdn.journaldev.com/wp-content/uploads/2014/05/Serial-GC-VisualGC.png)

**Java Garbage Collection Tuning** :

Should be the last option you should use for increasing the throughput of your application and only when you see drop in performance because of longer GC timings causing application timeout.

If you see java.lang.OutOfMemoryError: PermGen space errors in logs, then try to monitor and increase the Perm Gen memory space using -XX:PermGen and -XX:MaxPermGen JVM options. You might also try using -XX:+CMSClassUnloadingEnabled and check how it’s performing with CMS Garbage collector.

If you are see a lot of Full GC operations, then you should try increasing Old generation memory space.

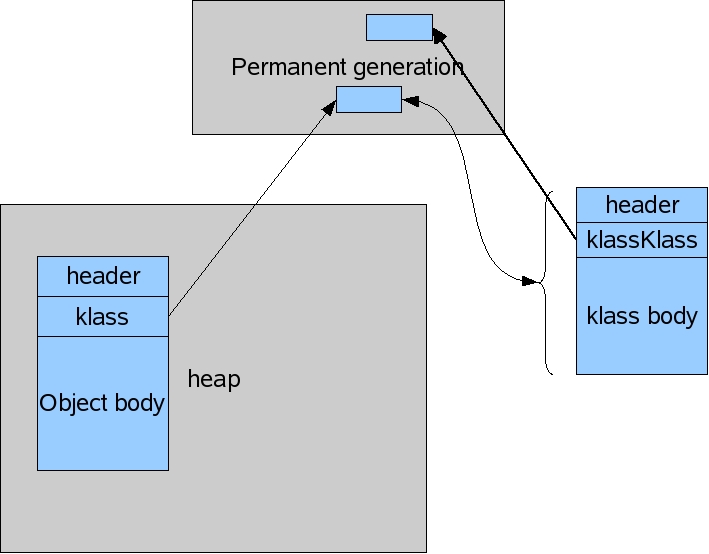
Overall garbage collection tuning takes a lot of effort and time and there is no hard and fast rule for that. You would need to try different options and compare them to find out the best one suitable for your application.

That’s all for Java Memory Model, Memory Management in Java and Garbage Collection, I hope it helps you in understanding JVM memory and garbage collection process.

### Permanent Generation:

Have you ever wondered how the permanent generation fits into our generational system? Ever been curious about what's in the permanent generation. Are objects ever promoted into it? Ever promoted out? We'll you're not alone. Here are some of the answers.

Java objects are instantiations of Java classes. Our JVM has an internal representation of those Java objects and those internal representations are stored in the heap (in the young generation or the tenured generation). Our JVM also has an internal representation of the Java classes and those are stored in the permanent generation. That relationship is shown in the figure below.



The internal representation of a Java object and an internal representation of a Java class are very similar. From this point on let me just call them Java objects and Java classes and you'll understand that I'm referring to their internal representation. The Java objects and Java classes are similar to the extent that during a garbage collection both are viewed just as objects and are collected in exactly the same way. So why store the Java objects in a separate permanent generation? Why not just store the Java classes in the heap along with the Java objects?

Well, there is a philosophical reason and a technical reason. The philosophical reason is that the classes are part of our JVM implementation and we should not fill up the Java heap with our data structures. The application writer has a hard enough time understanding the amount of live data the application needs and we shouldn't confuse the issue with the JVM's needs.

The technical reason comes in parts. Firstly the origins of the permanent generation predate my joining the team so I had to do some code archaeology to get the story straight (thanks Steffen for the history lesson).

Originally there was no permanent generation. Objects and classes were just stored together.

Back in those days classes were mostly static. Custom class loaders were not widely used and so it was observed that not much class unloading occurred. As a performance optimization the permanent generation was created and classes were put into it. The performance improvement was significant back then. With the amount of class unloading that occur with some applications, it's not clear that it's always a win today.

It might be a nice simplification to not have a permanent generation, but the recent implementation of the parallel collector for the tenured generation (aka parallel old collector) has made a separate permanent generation again desirable. The issue with the parallel old collector has to do with the order in which objects and classes are moved. If you're interested, I describe this at the end.

So the Java classes are stored in the permanent generation. What all does that entail? Besides the basic fields of a Java class there are

* Methods of a class (including the bytecodes)
* Names of the classes (in the form of an object that points to a string also in the permanent generation)
* Constant pool information (data read from the class file, see chapter 4 of the JVM specification for all the details).
* Object arrays and type arrays associated with a class (e.g., an object array containing references to methods).
* Internal objects created by the JVM (java/lang/Object or java/lang/exception for instance)
* Information used for optimization by the compilers (JITs)

That's it for the most part. There are a few other bits of information that end up in the permanent generation but nothing of consequence in terms of size. All these are allocated in the permanent generation and stay in the permanent generation. So now you know.

This last part is really, really extra credit. During a collection the garbage collector needs to have a description of a Java object (i.e., how big is it and what does it contain). Say I have an object X and X has a class K. I get to X in the collection and I need K to tell me what X looks like. Where's K? Has it been moved already? With a permanent generation during a collection we move the permanent generation first so we know that all the K's are in their new location by the time we're looking at any X's.

How do the classes in the permanent generation get collected while the classes are moving? Classes also have classes that describe their content. To distinguish these classes from those classes we spell the former klasses. The classes of klasses we spell klassKlasses. Yes, conversations around the office can be confusing. Klasses are instantiation of klassKlasses so the klassKlass KZ of klass Z has already been allocated before Z can be allocated. Garbage collections in the permanent generation visit objects in allocation order and that allocation order is always maintained during the collection. That is, if A is allocated before B then A always comes before B in the generation. Therefore if a Z is being moved it's always the case that KZ has already been moved.

And why not use the same knowledge about allocation order to eliminate the permanent generations even in the parallel old collector case? The parallel old collector does maintain allocation order of objects, but objects are moved in parallel. When the collection gets to X, we no longer know if K has been moved. It might be in its new location (which is known) or it might be in its old location (which is also known) or part of it might have been moved (but not all of it). It is possible to keep track of where K is exactly, but it would complicate the collector and the extra work of keeping track of K might make it a performance loser. So we take advantage of the fact that classes are kept in the permanent generation by collecting the permanent generation before collecting the tenured generation. And the permanent generation is currently collected serially.

# ---------------------Equals and Hash Code-----------------------------------

*Author: Manish Hatwalne*

## Introduction

The Java super class **java.lang.Object** has two very important methods defined in it. They are -

* public boolean equals(Object obj)
* public int hashCode()

These methods prove very important when user classes are confronted with other Java classes, when objects of such classes are added to collections etc. These two methods have become part of Sun Certified Java Programmer 1.4 exam (SCJP 1.4) objectives. This article intends to provide the necessary information about these two methods that would help the SCJP 1.4 exam aspirants. Moreover, this article hopes to help you understand the mechanism and general contracts of these two methods; irrespective of whether you are interested in taking the SCJP 1.4 exam or not. This article should help you while implementing these two methods in your own classes.

## public boolean equals(Object obj)

This method checks if some other object passed to it as an argument is *equal* to the object on which this method is invoked. The default implementation of this method in Object class simply checks if two object references x and y refer to the same object. i.e. It checks if x == y. This particular comparison is also known as "shallow comparison". However, the classes providing their own implementations of the equals method are supposed to perform a "deep comparison"; by actually comparing the relevant data members. Since Object class has no data members that define its state, it simply performs shallow comparison.  
  
This is what the JDK 1.4 API documentation says about the equals method of Object class-

Indicates whether some other object is "equal to" this one.

The equals method implements an equivalence relation:

* It is **reflexive**: for any reference value x, x.equals(x) should return true.
* It is **symmetric**: for any reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true.
* It is **transitive**: for any reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.
* It is **consistent**: for any reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the object is modified.
* For any non-null reference value x, x.equals(null) should return false.

The equals method for class Object implements the most discriminating possible equivalence relation on objects; that is, for any reference values x and y, this method returns true if and only if x and y refer to the same object (x==y has the value true).  
  
Note that it is generally necessary to override the hashCode method whenever this method is overridden, so as to maintain the general contract for the hashCode method, which states that equal objects must have equal hash codes.

The contract of the equals method precisely states what it requires. Once you understand it completely, implementation becomes relatively easy, moreover it would be correct. Let's understand what each of this really means.

1. **Reflexive -**It simply means that the object must be equal to itself, which it would be at any given instance; unless you intentionally override the equals method to behave otherwise.
2. **Symmetric -**It means that if object of one class is equal to another class object, the other class object must be equal to this class object. In other words, one object can not unilaterally decide whether it is equal to another object; two objects, and consequently the classes to which they belong, must bilaterally decide if they are equal or not. They BOTH must agree.  
   Hence, it is improper and incorrect to have your own class with equals method that has comparison with an object of java.lang.String class, or with any other built-in Java class for that matter. It is very important to understand this requirement properly, because it is quite likely that a naive implementation of equals method may violate this requirement which would result in undesired consequences.
3. **Transitive -**It means that if the first object is equal to the second object and the second object is equal to the third object; then the first object is equal to the third object. In other words, if two objects agree that they are equal, and follow the symmetry principle, one of them can not decide to have a similar contract with another object of different class. All three must agree and follow symmetry principle for various permutations of these three classes.  
   Consider this example - A, B and C are three classes. A and B both implement the equals method in such a way that it provides comparison for objects of class A and class B. Now, if author of class B decides to modify its equals method such that it would also provide equality comparison with class C; he would be violating the transitivity principle. Because, no properequals comparison mechanism would exist for class A and class C objects.
4. **Consistent -**It means that if two objects are equal, they must remain equal as long as they are not modified. Likewise, if they are not equal, they must remain non-equal as long as they are not modified. The modification may take place in any one of them or in both of them.
5. **null comparison -**It means that any instantiable class object is not equal to null, hence the equals method must return *false* if a null is passed to it as an argument. You have to ensure that your implementation of the equals method returns false if a null is passed to it as an argument.
6. **Equals & Hash Code relationship -**The last note from the API documentation is very important, it states the relationship requirement between these two methods. It simply means that if two objects are equal, then they must have the same hash code, however the opposite is NOT true. This is discussed in details later in this article.

The details about these two methods are interrelated and how they should be overridden correctly is discussed later in this article.

## public int hashCode()

This method returns the hash code value for the object on which this method is invoked. This method returns the hash code value as an integer and is supported for the benefit of hashing based collection classes such as Hashtable, HashMap, HashSet etc. This method must be overridden in every class that overrides the equals method.  
  
This is what the JDK 1.4 API documentation says about the hashCode method of Object class-

Returns a hash code value for the object. This method is supported for the benefit of hashtables such as those provided by java.util.Hashtable.

The general contract of hashCode is:

* Whenever it is invoked on the same object more than once during an execution of a Java application, the hashCode method must consistently return the same integer, provided no information used in equals comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
* If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result.
* It is not required that if two objects are unequal according to the equals(java.lang.Object) method, then calling the hashCode method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hashtables.

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.)

As compared to the general contract specified by the equals method, the contract specified by the hashCode method is relatively simple and easy to understand. It simply states two important requirements that must be met while implementing the hashCode method. The third point of the contract, in fact is the elaboration of the second point. Let's understand what this contract really means.

1. **Consistency during same execution -**Firstly, it states that the hash code returned by the hashCode method must be consistently the same for multiple invocations during the same execution of the application as long as the object is not modified to affect the equals method.
2. **Hash Code & Equals relationship -**The second requirement of the contract is the hashCode counterpart of the requirement specified by the equals method. It simply emphasizes the same relationship - equal objects must produce the same hash code. However, the third point elaborates that unequal objects *need not* produce distinct hash codes.

After reviewing the general contracts of these two methods, it is clear that the relationship between these two methods can be summed up in the following statement -

**Equal objects must produce the same hash code as long as they are equal, however unequal objects need not produce distinct hash codes.**

The rest of the requirements specified in the contracts of these two methods are specific to those methods and are not directly related to the relationship between these two methods. Those specific requirements are discussed earlier. This relationship also enforces that whenever you override the equals method, you must override the hashCode method as well. Failing to comply with this requirement usually results in undetermined, undesired behavior of the class when confronted with Java collection classes or any other Java classes.

## Correct Implementation Example

The following code exemplifies how all the requirements of equals and hashCode methods should be fulfilled so that the class behaves correctly and consistently with other Java classes. This class implements the equals method in such a way that it only provides equality comparison for the objects of the same class, similar to built-in Java classes like String and other wrapper classes.

1. public class Test

2. {

3. private int num;

4. private String data;

5.

6. public boolean equals(Object obj)

7. {

8. if(this == obj)

9. return true;

10. if((obj == null) || (obj.getClass() != this.getClass()))

11. return false;

12. // object must be Test at this point

13. Test test = (Test)obj;

14. return num == test.num &&

15. (data == test.data || (data != null && data.equals(test.data)));

16. }

17.

18. public int hashCode()

19. {

20. int hash = 7;

21. hash = 31 \* hash + num;

22. hash = 31 \* hash + (null == data ? 0 : data.hashCode());

23. return hash;

24. }

25.

26. // other methods

27. }

Now, let's examine why this implementation is the correct implementation. The class Test has two member variables - num and data. These two variables define state of the object and they also participate in the equals comparison for the objects of this class. Hence, they should also be involved in calculating the hash codes of this class objects.  
  
Consider the equals method first. We can see that at line 8, the passed object reference is compared with this object itself, this approach usually saves time if both the object references are referring to the same object on the heap and if the equals comparison is expensive. Next, the if condition at line 10 first checks if the argument is null, if not, then (due to the short-circuit nature of the OR || operator) it checks if the argument is of type Test by comparing the classes of the argument and this object. This is done by invoking the getClass() method on both the references. If either of these conditions fails, then false is returned. This is done by the following code -  
if((obj == null) || (obj.getClass() != this.getClass())) return false; **// prefer**  
This conditional check should be preferred instead of the conditional check given by -  
if(!(obj instanceof Test)) return false; **// avoid**  
This is because, the first condition (code in blue) ensures that it will return false if the argument is a subclass of the class Test. However, in case of the second condition (code in red) it fails. Theinstanceof operator condition fails to return false if the argument is a subclass of the class Test. Thus, it might violate the symmetry requirement of the contract. The instanceof check is correct only if the class is final, so that no subclass would exist. The first condition will work for both, final and non-final classes. Note that, both these conditions will return false if the argument isnull. The instanceof operator returns false if the left hand side (LHS) operand is null, irrespective of the operand on the right hand side (RHS) as specified by [JLS 15.20.2](http://java.sun.com/docs/books/jls/second_edition/html/expressions.doc.html#80289). However, the first condition should be preferred for better type checking.  
  
This class implements the equals method in such a way that it provides equals comparison only for the objects of the same class. Note that, this is not mandatory. But, if a class decides to provide equals comparison for other class objects, then the other class (or classes) must also agree to provide the same for this class so as to fulfill the symmetry and reflexivity requirements of the contract. This particular equals method implementation does not violate both these requirements. The lines 14 and 15 actually perform the equality comparison for the data members, and return true if they are equal. Line 15 also ensures that invoking the equals method on String variable data will not result in a NullPointerException.  
While implementing the equals method, primitives can be compared directly with an equality operator (==) after performing any necessary conversions (Such as float to Float.floatToIntBits or double to Double.doubleToLongBits). Whereas, object references can be compared by invoking their equals method recursively. You also need to ensure that invoking the equals method on these object references does not result in a NullPointerException.  
  
Here are some useful guidelines for implementing the equals method correctly.

1. Use the equality == operator to check if the argument is the reference to this object, if yes. return true. This saves time when actual comparison is costly.
2. Use the following condition to check that the argument is not null and it is of the correct type, if not then return false.  
   if((obj == null) || (obj.getClass() != this.getClass())) return false;  
   Note that, correct type does not mean the same type or class as shown in the example above. It could be any class or interface that one or more classes agree to implement for providing the comparison.
3. Cast the method argument to the correct type. Again, the correct type may not be the same class. Also, since this step is done after the above type-check condition, it will not result in aClassCastException.
4. Compare significant variables of both, the argument object and this object and check if they are equal. If \*all\* of them are equal then return true, otherwise return false. Again, as mentioned earlier, while comparing these class members/variables; primitive variables can be compared directly with an equality operator (==) after performing any necessary conversions (Such as float to Float.floatToIntBits or double to Double.doubleToLongBits). Whereas, object references can be compared by invoking their equals method recursively. You also need to ensure that invoking equals method on these object references does not result in a NullPointerException, as shown in the example above (Line 15).  
   It is neither necessary, nor advisable to include those class members in this comparison which can be calculated from other variables, hence the word "significant variables". This certainly improves the performance of the equals method. Only you can decide which class members are significant and which are not.
5. Do not change the type of the argument of the equals method. It takes a java.lang.Object as an argument, do not use your own class instead. If you do that, you will not be overriding theequals method, but you will be overloading it instead; which would cause problems. It is a very common mistake, and since it does not result in a compile time error, it becomes quite difficult to figure out why the code is not working properly.
6. Review your equals method to verify that it fulfills all the requirements stated by the general contract of the equals method.
7. Lastly, do not forget to override the hashCode method whenever you override the equals method, that's unpardonable. ;)

Now, let's examine the hashCode method of this example. At line 20, a non-zero constant value 7 (arbitrary) is assigned to an int variable hash. Since the class members/variables num and data do participate in the equals method comparison, they should also be involved in the calculation of the hash code. Though, this is not mandatory. You can use subset of the variables that participate in the equals method comparison to improve performance of the hashCode method. Performance of the hashCode method indeed is very important. But, you have to be very careful while selecting the subset. The subset should include those variables which are most likely to have the greatest diversity of the values. Sometimes, using all the variables that participate in the equals method comparison for calculating the hash code makes more sense.  
This class uses both the variables for computing the hash code. Lines 21 and 22 calculate the hash code values based on these two variables. Line 22 also ensures that invoking hashCode method on the variable data does not result in a NullPointerException if data is null. This implementation ensures that the general contract of the hashCode method is not violated. This implementation will return consistent hash code values for different invocations and will also ensure that equal objects will have equal hash codes.  
While implementing the hashCode method, primitives can be used directly in the calculation of the hash code value after performing any necessary conversions, such as float toFloat.floatToIntBits or double to Double.doubleToLongBits. Since return type of the hashCode method is int, long values must to be converted to the integer values. As for hash codes of the object references, they should be calculated by invoking their hashCode method recursively. You also need to ensure that invoking the hashCode method on these object references does not result in a NullPointerException.  
  
Writing a very good implementation of the hashCode method which calculates hash code values such that the distribution is uniform is not a trivial task and may require inputs from mathematicians and theoretical computer scientist. Nevertheless, it is possible to write a decent and correct implementation by following few simple rules.  
  
Here are some useful guidelines for implementing the hashCode method correctly.

1. Store an arbitrary non-zero constant integer value (say 7) in an int variable, called hash.
2. Involve significant variables of your object in the calculation of the hash code, all the variables that are part of equals comparison should be considered for this. Compute an individual hash code int var\_code for each variable var as follows -
   1. If the variable(var) is byte, char, short or int, then var\_code = (int)var;
   2. If the variable(var) is long, then var\_code = (int)(var ^ (var >>> 32));
   3. If the variable(var) is float, then var\_code = Float.floatToIntBits(var);
   4. If the variable(var) is double, then -  
      long bits = Double.doubleToLongBits(var);  
      var\_code = (int)(bits ^ (bits >>> 32));
   5. If the variable(var) is boolean, then var\_code = var ? 1 : 0;
   6. If the variable(var) is an object reference, then check if it is null, if yes then var\_code = 0; otherwise invoke the hashCode method recursively on this object reference to get the hash code. This can be simplified and given as -  
      var\_code = (null == var ? 0 : var.hashCode());
3. Combine this individual variable hash code var\_code in the original hash code hash as follows -   
   hash = 31 \* hash + var\_code;
4. Follow these steps for all the significant variables and in the end return the resulting integer hash.
5. Lastly, review your hashCode method and check if it is returning equal hash codes for equal objects. Also, verify that the hash codes returned for the object are consistently the same for multiple invocations during the same execution.

The guidelines provided here for implementing equals and hashCode methods are merely useful as guidelines, these are not absolute laws or rules. Nevertheless, following them while implementing these two methods will certainly give you correct and consistent results.

**public** **class** Equals1 {

String s="Test";

//int i = 10;

**public** Equals1() {}

**public** Equals1(String st) {

**this**.s = st;

//this.i = b;

}

**public** **boolean** equals(Equals1 othobj) {

**return** **this**.s.equals(othobj.s);

}

**public** **final** **int** hashcode() {

**return** s.hashCode();

}

}

**public** **class** MainEquals {

/\*\*

\* **@param** args

\*/

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

// **TODO** Auto-generated method stub

//Equals1 e1=new Equals1("s", 10);

//Equals1 e2=new Equals1("s", 10);

//--------------EQUALS--------------------

// 1. validating CHARACTER inside object.

// 2.By default its using == operator.

// 3. if compare object ,need to override equals method in object class (like Person class).

//-----------------(==)-----------------

// 1. compare two object referance to see whether they are refer the same instance

String s1 = **new** String("Hello");

String s2 = **new** String("Hello");

String s4 = "Hello";

String s3 = "Hello";

System.out.println("s1 equals s2:"+s1.equals(s2)); // TRUE

System.out.println("s1 equals s4:"+s1.equals(s4)); // TRUE

System.out.println("s2 equals s4:"+s2.equals(s4)); // TRUE

System.out.println("s1 == s4:"+(s1==s4)); //False

System.out.println("s1 == s2:"+(s1==s2)); //False

System.out.println("s3 == s4:"+(s3==s4)); //TRUE

System.out.println("s3 equals s4:"+(s3.equals(s4))); //TRUE

//----------------------WRAPPER CLASS------------------

Integer i = **new** Integer(10);

Integer i2 = **new** Integer(10);

**int** i3 = 10;

**int** i4 = 10;

**int** i5 = 11;

System.out.println("--------------------WRAPPER------------");

System.out.println("i equals i2:"+ (i.equals(i2))); // TRUE

System.out.println("i == i2:"+ (i==i2)); // FALSE

System.out.println("i2 == i3:"+ (i3==i2)); // TRUE

System.out.println("i4 == i3:"+ (i4==i3)); // TRUE

System.out.println("i == i3:"+ (i==i3)); // TRUE

System.out.println("i2 equals i3:"+ (i2.equals(i3))); // TRUE

System.*out*.println("----------Class equals--------------------");

Equals1 e1=**new** Equals1("test");

Equals1 e2=**new** Equals1("test");

Equals1 e4=**new** Equals1("test4");

Equals1 e3=e1;

System.*out*.println("e1 Euals e2:"+e1.equals(e2)); // TRUE

System.*out*.println("e1 == e2:"+(e1==e2)); //FALSE

System.*out*.println("e3 Euals e2:"+e3.equals(e1)); // TRUE

System.*out*.println("e3 == e1:"+(e3==e1)); // TRUE

System.*out*.println("e4 Euals e1:"+e4.equals(e1)); // TRUE

System.*out*.println("e4 == e1:"+ (e4==e1)); // FALSE

System.*out*.println("e3 :"+e3.s + " e1 :" +e1.s); //e3 :test4 e1:test4

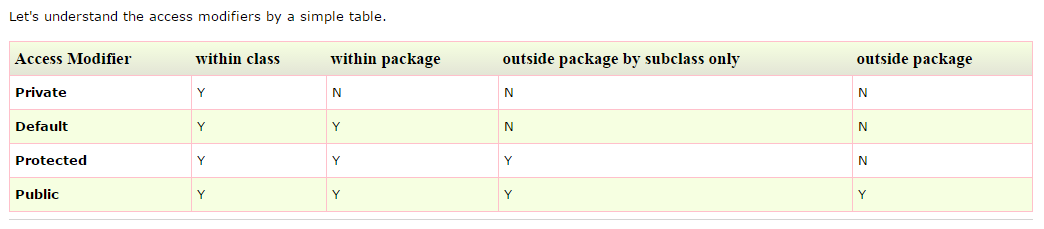
e3.s="test4";

System.*out*.println("e1 Euals e2:"+e1.equals(e2)); // FALSE

}

}

**Access Modifiers**



Method access from super class method access modifier to sub class access modifier

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Super Class Method Type/**  **SubClass method Type** | **Private** | **Default** | **Protected** | **Public** |
| **Private** | Yes | Yes | Yes | Yes |
| **Default** | No | Yes | Yes | Yes |
| **Protected** | No | No | Yes | Yes |
| **Public** | No | No | No | Yes |

# private access modifier

The private access modifier is accessible only within class.

**Simple example of private access modifier**

|  |
| --- |
| In this example, we have created two classes A and Simple. A class contains private data member and private method. We are accessing these private members from outside the class, so there is compile time error. |
|  |

**class** A{

**private** **int** data=40;

**private** **void** msg(){System.out.println("Hello java");}

}

**public** **class** Simple{

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

   A obj=**new** A();

   System.out.println(obj.data);//Compile Time Error

   obj.msg();//Compile Time Error

    }

}

**Role of Private Constructor**

|  |
| --- |
| If you make any class constructor private, you cannot create the instance of that class from outside the class. For example: |

**class** A{

**private** A(){}//private constructor

**void** msg(){System.out.println("Hello java");}

}

**public** **class** Simple{

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

    A obj=**new** A();//Compile Time Error

  }

}

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

# 2) default access modifier

|  |
| --- |
| If you don't use any modifier, it is treated as **default** bydefault. The default modifier is accessible only within package. |

**Example of default access modifier**

|  |
| --- |
| In this example, we have created two packages pack and mypack. We are accessing the A class from outside its package, since A class is not public, so it cannot be accessed from outside the package. |

//save by A.java

**package** pack;

**class** A{

**void** msg(){System.out.println("Hello");}

}

//save by B.java

**package** mypack;

**import** pack.\*;

**class** B{

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

    A obj = **new** A();//Compile Time Error

    obj.msg();//Compile Time Error

  }

}

In the above example, the scope of class A and its method msg() is default so it cannot be accessed from outside the package.

# 3) protected access modifier

The **protected access modifier** is accessible within package and outside the package but through inheritance only.

The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.

**Example of protected access modifier**

In this example, we have created the two packages pack and mypack. The A class of pack package is public, so can be accessed from outside the package. But msg method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

//save by A.java

**package** pack;

**public** **class** A{

**protected** **void** msg(){System.out.println("Hello");}

}

//save by B.java

**package** mypack;

**import** pack.\*;

**class** B **extends** A{

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

   B obj = **new** B();

   obj.msg();

   }

}

Output:Hello

# 4) public access modifier

|  |
| --- |
| The **public access modifier** is accessible everywhere. It has the widest scope among all other modifiers. |

**Example of public access modifier**

//save by A.java

**package** pack;

**public** **class** A{

**public** **void** msg(){System.out.println("Hello");}

}

//save by B.java

**package** mypack;

**import** pack.\*;

**class** B{

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

   A obj = **new** A();

    obj.msg();

  }

}

 Output:Hello

**Java access modifiers with method overriding**

If you are overriding any method, overridden method (i.e. declared in subclass) must not be more restrictive.

**class** A{

**protected** **void** msg(){System.out.println("Hello java");}

}

**public** **class** Simple **extends** A{

**void** msg(){System.out.println("Hello java");}//C.T.Error

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

   Simple obj=**new** Simple();

    obj.msg();

   }

}

The default modifier is more restrictive than protected. That is why there is compile time error

# Overloadding And Overrrding

|  |
| --- |
| **Overloading:** There are two ways to overload the method in java |

1. By changing number of arguments
2. By changing the data type
3. Method Overloading is not possible by changing the return type of the method.

Q1. Why Method Overloaing is not possible by changing the return type of method?

**Due to ambiguity of the method.**

Q2. Can we overload main() method?

**Yes we can overload main method.**

# Constructor in Java :

There are two types of constructor

1.Default Constructor

2.Parametrized constructor

**Q) What is the purpose of default constructor?**

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

**Constructor Overloading in Java**

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

## Difference between constructor and method in java

There are many differences between constructors and methods. They are given below.

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

# Final Keyword Question

**1 What is the use of final keyword in java?**    
 By using final keyword we can make

Final class

Final method

Final variables

If we declare any class as final we can not extend that class

If we declare any method as final it can not be overridden in sub class

If we declare any variable as final its value unchangeable once assigned.

**2. What is the main difference between abstract method and final method?**

* Abstract methods must be overridden in sub class where as final methods can not be overridden in sub class

**3. What is the actual use of final class in java?**

If a class needs some security and it should not participate in inheritance in this scenario we need to use final class. We can not extend final class.

**4. What will happen if we try to extend final class in java?**

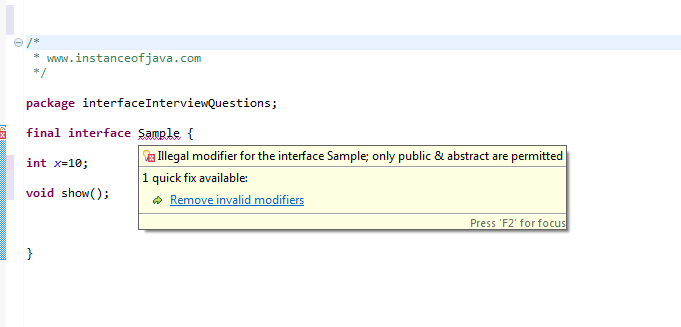
Compile time error will come.

1. package com.finalkeywordintweviewprograms;
2. public final  Class SuperDemo{
3. int a,b;
5. public void show() {
6. System.out.println(a);
7. System.out.println(b);
9. }
10. }

1. package com.finalkeywordintweviewprograms;
2. public Class Sample  extends SuperDemo{  //The type Sample cannot subclass the final class
3. SuperDemo
4. }

**5.Can we declare interface as final?**

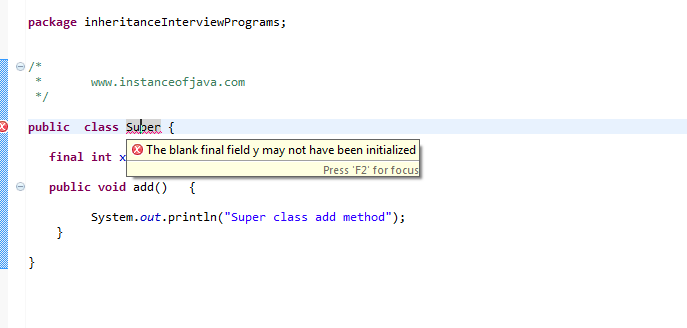
No We can not declare interface as final because interface should be implemented by some class so its not possible to declare interface as final.

[](https://3.bp.blogspot.com/-Os3PkMKXd4g/VtWhsViGPQI/AAAAAAAAAn0/e6onSIYwGrE/s1600/interface+final.png)

**6. Is it possible to declare final variables without initialization?**

No. Its not possible to declare a final variable without initial value assigned.

While declaring itself we need to initialize some value and that value can not be change at any time.

[](https://2.bp.blogspot.com/-f-ycwOdkLeg/VtgRQGt_JuI/AAAAAAAAAoQ/XZMEbuBWHTU/s1600/final+variable.png)

1. package com.finalkeywordintweviewprograms;
2. public final  Class Sample{
4. final int x=12,y=13;
6. public void Method() {
7. x=25;// compile time error:The final field Super.x cannot be assigned
8. y=33;// compile time error: The final field Super.y cannot be assigned
10. }
12. }

**7. Can we declare constructor as final?**  
No . Constructors can not be final.

**8.What will happen if we try to override final methods in sub classes?**  
Compile time error will come :Cannot override the final method from Super class

**9.Can we create object for final class?**  
Yes we can create object for final class.

**10.What is the most common predefined final class object you used in your code?**  
 String (for example)

# STATIC Keyword Questions

#### 1.what is static in java?

Static is a keyword in java.

One of the Important keyword in java.

Clear understanding of static keyword is required to build projects.

We have static variables, static methods , static blocks.

Static means class level.

#### 2.Why we use static keyword in java?

Static keyword is mainly used for memory management.

Static variables get memory when class loading itself.

Static variables can be used to point common property all objects.

#### 3.What is static variable in java?

Variables declared with static keyword is known as static variables.

Static variables gets memory on class loading.

Static variables are class level.

If we change any static variable value using a particular object then its value changed for all objects means it is common to every object of that class.

static int a,b;

1. package com.instanceofjavastatic;
2. class StaticDemo{
4. static int a=40;
5. static int b=60;
7. }

* We can not declare local variables as static it leads to compile time error "illegal start of expression".
* Because being static variable it must get memory at the time of class loading, which is not possible to provide memory to local variable at the time of class loading.

1. package com.instanceofjava;
2. class StaticDemo{
4. static int a=10;
5. static int b=20;
6. public static void main(String [] args){
8. //local variables should not be static
9. static int a=10;// compile time error: illegal start of expression
10. }
11. }

 Read more : [Explain about static variables in java?](http://www.instanceofjava.com/2015/04/static-variable-method-block-in-java.html)

**4. *what is static method in java with example***

* Method which is having static in its method definition is known as static method.

1. static void show(){
3. }

* JVM will not call these static methods automatically. Develioper needs to call these static methods from main method or static block or variable initialization.
* Only Main method will b called by JVM automatically.
* We can call these static methods by using class name itself no need to create object.

Read more @[what is static method in java](http://www.instanceofjava.com/2015/04/static-methods-in-java-example.html)

#### 5.what is static block in java?

* Static blocks are the blocks which will have braces and with static keyword.
* These static blocks will be called when JVM loads the class.
* Static blocks are the blocks with static keyword.
* Static blocks wont have any name in its prototype.
* Static blocks are class level.
* Static block will be executed only once.
* No return statements.
* No arguments.
* No this or super keywords supported.

* 1. static{
  3. }

 Read more @ [Static blocks in java with example programs](http://www.instanceofjava.com/2015/05/static-block-in-java-example.html)

**6.What is the need of static block?**

* Static blocks will be executed at the time of class loading.
* So if you want any logic that needs to be executed at the time of class loading that logic need to place inside the static block so that it will be executed at the time of class loading.

**7.Why main method is static in java?**

* To execute main method without creating object then the main method should be static so that JVM will call main method by using class name itself.

**8.Can we overload static methods in java?**

* Yes we can overload static methods in java.
* Its always possibles static method overloading.
* Defining multiple static methods with same name and different arguments will possible.
* By this we can define multiple main methods in our class with different arguments but only default main method will be called by the JVM remaining methods we need to call explicitly.
* Lets see an example java program which explains static method overloading.

1. class StaticMethodOverloading{
3. public static void staticMethod(){
5. System.out.println("staticMethod(): Zero arguments");
7. }
9. public static void staticMethod(int a){
11. System.out.println("staticMethod(int a): one argument");
13. }
15. public static void staticMethod(String str, int x){
17. System.out.println("staticMethod(String str, int x): two arguments");
19. }
21. public static void main(String []args){
23. StaticMethodOverloading.staticMethod();
24. StaticMethodOverloading.staticMethod(12);
25. StaticMethodOverloading.staticMethod("Static method overloading",10);
27. }
28. }

 **Output:**

1. staticMethod(): Zero arguments
2. staticMethod(int a): one argument
3. staticMethod(String str, int x): two arguments

**9.Can we override static methods in java / Method Hiding?**

* NO we can not override static methods in java.
* Method overriding : in method overriding we will define super class method in sub class with same signature and these methods will be called at run time based on the object.
* Method Hiding: In method hiding even though super class methods are accesible in sub class they are not the part of sub class so the methods and these methods will be called based on the class.
* In method Hiding if we define same static method in super class and sub class they are not same they are unique and distinct from the other.

**10.Can we write static public void main(String [] args)?**

* Yes we can define main method like static public void main(String[] args){}
* Order of modifiers we can change.

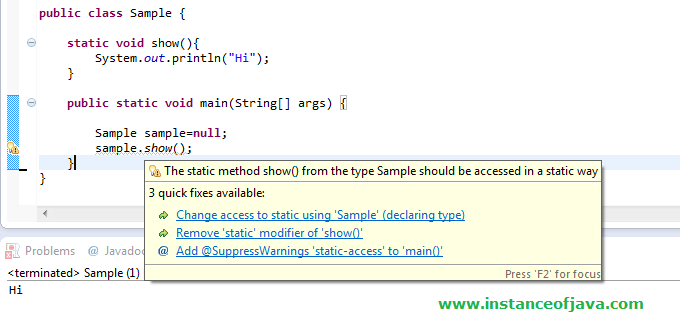
1. package instanceofjava;
2. public MainDemo{
3. **static public**void main(String [] args){
5. }
6. }

**11.Can we call super class static methods from sub class?**

* Yes we can call super class static methods from sub class.
* If you want to call static method of a class we can call directly from another static method or by using its class name we can call static method of that class.
* let us see a program on how to call a static method of a class

1. package com.instanceofjava;
2. public class Sample{
3. public static void show(){
5. System.out.println("show() method called");
7. }
8. public static void main(String args[]){
10. show();
11. Sample.show();
13. }
14. }

* We can also call static method like this but this is not recommended.

[](https://4.bp.blogspot.com/-z8iqYaXuRYU/Vsat-b_gN6I/AAAAAAAAAlk/vSdqsrhuz0g/s1600/static+in+java.png)

**Output:**

1. show() method called
2. show() method called

* Now our question is can we call super class static method from sub class?
* Yes we can call super class static method inside sub class using super\_class\_method();
* We can also call super class static method using Sub\_class\_name.superclass\_staticMethod()

1. package com.instanceofjava;
3. public class SuperDemo{
4. public static void show(){
6. System.out.println("Super class show() method called");
8. }
10. }

1. package com.instanceofjava;
2. public class SubDemo extends SuperDemo{
3. public void print(){
5. System.out.println("Sub class print() method called");
7. }
8. public static void main(String args[]){
10. SuperDemo.show();
11. SubDemo.show();
12. }
13. }

**Output:**

1. Super class show() method called
2. Super class show() method called

* If the same static method defined in sub class also then we can not call super class method using sub class name if we call them sub class static method will be executed.

1. package com.instanceofjava;
3. public class SuperDemo{
4. public static void show(){
6. System.out.println("Super class show() method called");
8. }
10. }
11. package com.instanceofjava;
12. public class SubDemo extends SuperDemo{
13. public static void show(){
15. System.out.println("Sub class show() method called");
17. }
18. public static void main(String args[]){
20. SuperDemo.show();
21. SubDemo.show();
22. }
23. }

**Output:**

1. Super class show() method called
2. Sub class show() method called

**Java static constructor – Is it really Possible to have them in Java?**

Have you heard of **static constructor in Java**? I guess yes but the fact is that they are not allowed in Java. A constructor can not be static in Java.  Before I explain the reason let’s have a look at the below piece of code:

public class StaticTest

{

/\* See below - I have marked the constructor as static \*/

public static StaticTest()

{

System.out.println("Static Constructor of the class");

}

public static void main(String args[])

{

/\*Below: I'm trying to create an object of the class

which would intern call the constructor\*/

StaticTest obj = new StaticTest();

}

}

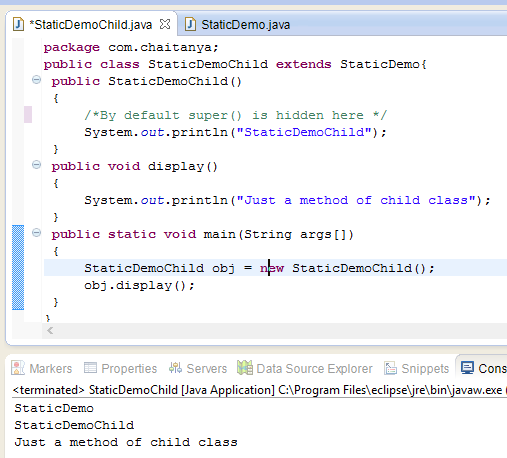
**Output:** You would get the below error message when you try to run the above java code.

**“modifier static not allowed here”**

**Why java doesn’t support static constructor?**

It’s actually pretty simple to understand – Everything that is marked static belongs to the class only, for example **static method cannot be inherited in the sub class because they belong to the class in which they have been declared. Refer**[**static keyword**](http://beginnersbook.com/2013/04/java-static-class-block-methods-variables/)**.**  
Lets back to constructors, Since each constructor is being called by its subclass during creation of the object of its subclass, so if you mark constructor as static the subclass will not be able to access the constructor of its parent class because it is marked static and thus belong to the class only. This will violate the whole purpose of inheritance concept and that is reason why a constructor cannot be static.

Let’s understand this with the help of an example –



public class StaticDemo

{

public StaticDemo()

{

/\*Constructor of this class\*/

System.out.println("StaticDemo");

}

}

public class StaticDemoChild extends StaticDemo

{

public StaticDemoChild()

{

/\*By default this() is hidden here \*/

System.out.println("StaticDemoChild");

}

public void display()

{

System.out.println("Just a method of child class");

}

public static void main(String args[])

{

StaticDemoChild obj = new StaticDemoChild();

obj.display();

}

}

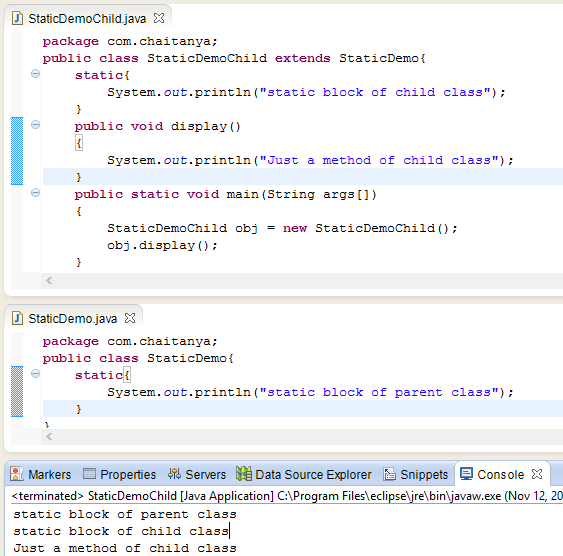
**Output:**  
StaticDemo  
StaticDemoChild  
Just a method of child class

**Did you notice?**We just created the object of child class and as a result it first called the constructor of parent class and then the constructor of it’s own class. It happened because the object creation calls constructor implicitly and every **child class constructor by default has super()** as first statement which calls it’s parent class’s constructor. The statement super() is used to call the parent class(base class) constructor.

Above explanation is the reason why constructor cannot be static – Because if we make them static they cannot be called from child class thus object of child class couldn’t be created.

**Static Constructor Alternative – Static Blocks**

Java has static blocks which can be treated as static constructor. Let’s consider the below program –



public class StaticDemo{

static{

System.out.println("static block of parent class");

}

}

public class StaticDemoChild extends StaticDemo{

static{

System.out.println("static block of child class");

}

public void display()

{

System.out.println("Just a method of child class");

}

public static void main(String args[])

{

StaticDemoChild obj = new StaticDemoChild();

obj.display();

}

}

**Output:**  
static block of parent class  
static block of child class  
Just a method of child class

In the above example we have used static blocks in both the classes which worked perfectly. We cannot use static constructor so it’s a good alternative if we want to perform a static task during object creation.

--------------------------------I**mmutable class--------------------------------------**

As said earlier, Immutable classes are those class, whose [object](http://javarevisited.blogspot.com/2012/12/what-is-object-in-java-or-oops-example.html) can not be modified once created, it means any modification on immutable object will result in another immutable object. best example to understand immutable and mutable objects are, [String and StringBuffer](http://javarevisited.blogspot.com/2011/07/string-vs-stringbuffer-vs-stringbuilder.html). Since String is immutable class, any change on existing string object will result in another string e.g. replacing a character into String, [creating substring from String](http://javarevisited.blogspot.in/2011/10/how-substring-in-java-works.html), all result in a new objects. While in case of mutable object like StringBuffer, any modification is done on object itself and no new objects are created. Some times this immutability of String can also cause security hole, and that the reason [why password should be stored on char array instead of String](http://javarevisited.blogspot.com/2012/03/why-character-array-is-better-than.html).  
  
  
**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.

## 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

------------------**Observer And Observable**-------------------------------

**Observable:** This class represents an observable object, or "data" in the model-view paradigm. It can be subclassed to represent an object that the application wants to have observed. An observable object can have one or more observers. An observer may be any object that implements interface Observer. After an observable instance changes, an application calling the Observable's notifyObservers method causes all of its observers to be notified of the change by a call to their update method. The order in which notifications will be delivered is unspecified. The default implementation provided in the Observable class will notify Observers in the order in which they registered interest, but subclasses may change this order, use no guaranteed order, deliver notifications on separate threads, or may guarantee that their subclass follows this order, as they choose. Note that this notification mechanism has nothing to do with threads and is completely separate from the wait and notify mechanism of class Object. When an observable object is newly created, its set of observers is empty. Two observers are considered the same if and only if the equals method returns true for them.

* Is normal call and which provide following methods

1. *Void addObserver(Observer obj)*
2. *Void clearChanged().*
3. *deleteObserver(Observer obj)*
4. *setChange()*
5. *notifyObserver() and notifyObserver(Object obg);*

**Observer:**

*Void update(Observable objservobj, Object obj)*

Observobj – is the object observed and arg is the value passed by notyObservers()

**Following**

**package** com.prodconsum;

**import** java.util.Observable;

**import** java.util.Observer;

**public** **class** WatcherObserver **implements** Observer{

@Override

**public** **void** update(Observable o, Object arg) {

// **TODO** Auto-generated method stub

System.*out*.println(" Update called :"+ ((Integer)arg).intValue());

}

}

**package** com.prodconsum;

**import** java.util.Observable;

**public** **class** BeingObserable **extends** Observable{

**void** counter(**int** period) {

**for**(; period >=0; period--) {

setChanged();

System.*out*.println(" has changed:"+hasChanged());

notifyObservers(**new** Integer(period));

**try** {

Thread.*sleep*(100);

}

**catch**(InterruptedException e) {

System.*out*.println("Sleep Interrupted");

}

}

}

}

**package** com.prodconsum;

**public** **class** MainClass {

/\*\*

\* **@param** args

\*/

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

// **TODO** Auto-generated method stub

BeingObserable obserable = **new** BeingObserable();

WatcherObserver observer = **new** WatcherObserver();

WatcherObserver observer2 = **new** WatcherObserver();

obserable.addObserver(observer);

obserable.addObserver(observer2);

obserable.counter(10);

}

}

# -------------------------------Serialization-------------------------------------------------

Is a mechanism of writing the **state of an object into a byte stream**. It is mainly used in Hibernate, RMI, JPA, EJB, JMS technologies. The reverse operation of serialization is called ***deserialization***. The **String** class and all the **wrapper classes** implements *java.io.Serializable* interface by default. It is mainly used to travel object's state on the network (known as marshaling).

**import** java.io.Serializable;

**public** **class** Student **implements** Serializable{

**int** id;

 String name;

**public** Student(**int** id, String name) {

**this**.id = id;

**this**.name = name;

 }

}

**import** java.io.\*;

**class** Persist{

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

   Student s1 =**new** Student(211,"ravi");

   FileOutputStream fout=**new** FileOutputStream("f.txt");

   ObjectOutputStream out=**new** ObjectOutputStream(fout);

   out.writeObject(s1);

   out.flush();

    System.out.println("success");

  }

}

**import** java.io.\*;

**class** Depersist{

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

   ObjectInputStream in=**new** ObjectInputStream(**new** FileInputStream("f.txt"));

   Student s=(Student)in.readObject();

   System.out.println(s.id+" "+s.name);

   in.close();

 }

}

## Java Serialization with Inheritance (IS-A Relationship) :

If a class implements serializable then all its sub classes will also be serializable. Let's see the example given below:

**import** java.io.Serializable;

**class** Person **implements** Serializable{

**int** id;

  String name;

  Person(**int** id, String name) {

**this**.id = id;

**this**.name = name;

  }

}

**class** Student **extends** Person{

 String course;

**int** fee;

**public** Student(**int** id, String name, String course, **int** fee) {

**super**(id,name);

**this**.course=course;

**this**.fee=fee;

}

}

Now you can serialize the Student class object that extends the Person class which is Serializable.Parent class properties are inherited to subclasses so if parent class is Serializable, subclass would also be.

## Java Serialization with Aggregation (HAS-A Relationship) :

If a class has a reference of another class, all the references must be Serializable otherwise serialization process will not be performed. In such case, *NotSerializableException* is thrown at runtime.

**class** Address{

 String addressLine,city,state;

**public** Address(String addressLine, String city, String state) {

**this**.addressLine=addressLine;

**this**.city=city;

**this**.state=state;

}

}

**import** java.io.Serializable;

**public** **class** Student **implements** Serializable{

**int** id;

 String name;

 Address address;//HAS-A

**public** Student(**int** id, String name) {

**this**.id = id;

**this**.name = name;

 }

}

Since Address is not Serializable, you can not serialize the instance of Student class.

#### Note: All the objects within an object must be Serializable.

## Java Serialization with static data member :

**I**f there is any static data member in a class, it will not be serialized because static is the part of class not object.

**class** Employee **implements** Serializable{

**int** id;

 String name;

**static** String company="SSS IT Pvt Ltd";//it won't be serialized

**public** Student(**int** id, String name) {

**this**.id = id;

**this**.name = name;

 }

}

## Java Serialization with array or collection:

Rule: In case of array or collection, all the objects of array or collection must be serializable. If any object is not serialiizable, serialization will be failed.

## Externalizable in java

The Externalizable interface provides the facility of writing the state of an object into a byte stream in compress format. It is not a marker interface.

The Externalizable interface provides two methods:

* **public void writeExternal(ObjectOutput out) throws IOException**
* **public void readExternal(ObjectInput in) throws IOException**

-------------------------------------Cloning or copying class---------------------------

There are two types of cloning

1. **Shallow cloning**: which means that the parent object or the top level objects and its member are duplicated, but any lower level objects are not duplicated. Rather refrence of these object are copied. So when an object is shallow cloned and modify any of it child classes it will affect the original copy.
2. **Deep cloning:** when an object is deep cloned the entire object and its child objects are alos copied.

**Shallow Cloning Implementation:**

In order to implement shallow cloning we need to implement **clone able** interface and override the clone method with implementation.

import java.util.\*;

public class clsExampleClone implements Cloneable

{

int myNum;

// Array aggregated inside the object

Integer [] myArray;

// Constructor loads array of iteger values

clsExampleClone (int nElements)

{

myNum = nElements;

myArray = new Integer[myNum];

Random ran = new Random ();

for (int i=0; i < myNum; i++)

{myArray[i] = new Integer (ran.nextInt (10000));}

}

// Override the clone object with Implementation

public Object clone ()

{

try

{

return super.clone ();

}

catch (CloneNotSupportedException e)

{throw new Error ("Clone Error");}

}

}

---------------------------COLLECTION CLASS-------------------------------

**Difference between Stack vs Heap in Java**

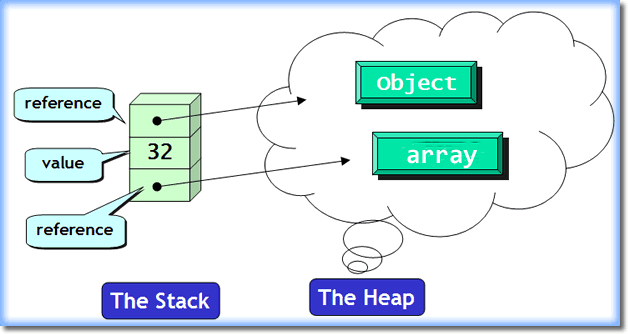
Here are few differences between stack and heap memory in Java:

1) The main difference between heap and stack is that stack memory is used to store [local variables](http://javarevisited.blogspot.com/2012/02/difference-between-instance-class-and.html) and function call while heap memory is used to store objects in Java. No matter, where the object is created in code e.g. as a member variable, local variable or class variable,  they are always created inside heap space in Java.

2) Each [Thread in Java](http://javarevisited.blogspot.com/2011/02/how-to-implement-thread-in-java.html) has their own stack which can be specified using -Xss JVM parameter, similarly, you can also specify heap size of Java program using JVM option -Xms and -Xmx where -Xms is starting size of the heap and -Xmx is a maximum size of java heap. to learn more about JVM options see my post [10 JVM option Java programmer should know](http://javarevisited.blogspot.com/2011/11/hotspot-jvm-options-java-examples.html).

3) If there is no memory left in the stack for storing function call or local variable, JVM will throw java.lang.StackOverFlowError, while if there is no more heap space for creating an object, JVM will throw java.lang.OutOfMemoryError: Java Heap Space. Read more about how to deal with java.lang.OutOfMemoryError  in my post [2 ways to solve OutOfMemoryError in Java](http://javarevisited.blogspot.com/2011/09/javalangoutofmemoryerror-permgen-space.html).

4) If you are using [Recursion](http://javarevisited.blogspot.com/2012/12/recursion-in-java-with-example-programming.html), on which method calls itself, You can quickly fill up stack memory. Another difference between stack and heap is that size of stack memory is a lot lesser than the size of  heap memory in Java.

[](https://1.bp.blogspot.com/-gKWUcwIKWWU/VvPtKUAIFjI/AAAAAAAAFRc/WLCqWfSxlZ4ioocmBuFS3KaRhzs0I13OA/s1600/Difference+between+stack+and+heap+memory+in+Java.gif)

5) Variables stored in stacks are only visible to the owner Thread while objects created in the heap are visible to all thread. In other words, stack memory is kind of private memory of Java Threads while heap memory is shared among all threads.

**-------------------------------Exception Handling -----------------------**

## Java Multi catch block

#### Rule: At a time only one Exception is occured and at a time only one catch block is executed.

#### Rule: All catch blocks must be ordered from most specific to most general i.e. catch for ArithmeticException must come before catch for Exception .

1. **public** **class** TestMultipleCatchBlock{
2. **public** **static** **void** main(String args[]){
3. **try**{
4. **int** a[]=**new** **int**[5];
5. a[5]=30/0;
6. }
7. **catch**(ArithmeticException e){System.out.println("task1 is completed");}
8. **catch**(ArrayIndexOutOfBoundsException e){System.out.println("task 2 completed");}
9. **catch**(Exception e){System.out.println("common task completed");}
10. System.out.println("rest of the code...");
11. }  }

**Output:task1 completed**

**rest of the code...**

1. **try**{
2. **int** a[]=**new** **int**[5];
3. a[5]=30/0;
4. }
5. **catch**(Exception e){System.out.println("common task completed");}
6. **catch**(ArithmeticException e){System.out.println("task1 is completed");}
7. **catch**(ArrayIndexOutOfBoundsException e){System.out.println("task 2 completed");}
8. System.out.println("rest of the code...");
9. }

**Compile-time error**

1. Method Overriding in Java.

There are many rules if we talk about methodoverriding with exception handling. The Rules are as follows:

**If the superclass method does not declare an exception**

If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but it can declare unchecked exception.

**If the superclass method declares an exception**

If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception.

### If the superclass method does not declare an exception

#### 1) Rule: If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception.

1. **import** java.io.\*;
2. **class** Parent{
3. **void** msg(){System.out.println("parent");}
4. }
6. **class** TestExceptionChild **extends** Parent{
7. **void** msg()**throws** IOException{  //CT : Compile time error
8. System.out.println("TestExceptionChild");
9. }
10. **public** **static** **void** main(String args[]){
11. Parent p=**new** TestExceptionChild();
12. p.msg();
13. }
14. }

#### 2) Rule: If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but can declare unchecked exception.

1. **import** java.io.\*;
2. **class** Parent{
3. **void** msg(){System.out.println("parent");}
4. }
5. **class** TestExceptionChild1 **extends** Parent{
6. **void** msg()**throws** ArithmeticException{
7. System.out.println("child");
8. }
9. **public** **static** **void** main(String args[]){
10. Parent p=**new** TestExceptionChild1();
11. p.msg();
12. }
13. }

If the superclass method declares an exception

#### 1) Rule: If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception.

1. **import** java.io.\*;
2. **class** Parent{
3. **void** msg()**throws** ArithmeticException{System.out.println("parent");}
4. }
6. **class** TestExceptionChild2 **extends** Parent{
7. **void** msg()**throws** Exception{System.out.println("child"); //CT: Compileer time error
8. }
10. **public** **static** **void** main(String args[]){
11. Parent p=**new** TestExceptionChild2();
12. **try**{
13. p.msg();
14. }**catch**(Exception e){}
15. }
16. }

Example in case subclass overridden method declares no exception

1. **import** java.io.\*;
2. **class** Parent{
3. **void** msg()**throws** Exception{System.out.println("parent");}
4. }
5. **class** TestExceptionChild5 **extends** Parent{
6. **void** msg(){System.out.println("child");}
8. **public** **static** **void** main(String args[]){
9. Parent p=**new** TestExceptionChild5();
10. **try**{
11. p.msg();
12. }**catch**(Exception e){}
13. }
14. }

### Method Overriding with Exception Handling

There are few things to remember when overriding a method with exception handling. If super class method does not declare any exception, then sub class overriden method cannot declare checked exception but it can declare unchecked exceptions.

#### Example of Subclass overriden Method declaring Checked Exception

import java.io.\*;

class Super

{

void show() { System.out.println("parent class"); }

}

public class Sub extends Super

{

void show() throws IOException //Compile time error

{ System.out.println("parent class"); }

public static void main( String[] args )

{

Super s=new Sub();

s.show();

}

}

As the method **show()** doesn't throws any exception while in Super class, hence its overriden version can also not throw any checked exception.

#### Example of Subclass overriden Method declaring Unchecked Exception

import java.io.\*;

class Super

{

void show(){ System.out.println("parent class"); }

}

public class Sub extends Super

{

void show() throws ArrayIndexOutOfBoundsException //Correct

{ System.out.println("child class"); }

public static void main(String[] args)

{

Super s=new Sub();

s.show();

}

}

Output : child class

Because *ArrayIndexOutOfBoundsException* is an unchecked exception hence, overrided **show()** method can throw it.

#### More about Overriden Methods and Exceptions

If Super class method throws an exception, then Subclass overriden method can throw the same exception or no exception, but must not throw parent exception of the exception thrown by Super class method.

It means, if Super class method throws object of **NullPointerException** class, then Subclass method can either throw same exception, or can throw no exception, but it can never throw object of **Exception** class (parent of NullPointerException class).

#### Example of Subclass overriden method with same Exception

import java.io.\*;

class Super

{

void show() throws Exception

{ System.out.println("parent class"); }

}

public class Sub extends Super {

void show() throws Exception //Correct

{ System.out.println("child class"); }

public static void main(String[] args)

{

try {

Super s=new Sub();

s.show();

}

catch(Exception e){}

}

}

Output : child class

#### Example of Subclass overriden method with no Exception

import java.io.\*;

class Super

{

void show() throws Exception

{ System.out.println("parent class"); }

}

public class Sub extends Super {

void show() //Correct

{ System.out.println("child class"); }

public static void main(String[] args)

{

try {

Super s=new Sub();

s.show();

}

catch(Exception e){}

}

}

Output : child class

#### Example of Subclass overriden method with parent Exception

import java.io.\*;

class Super

{

void show() throws ArithmeticException

{ System.out.println("parent class"); }

}

public class Sub extends Super {

void show() throws Exception //Cmpile time Error

{ System.out.println("child class"); }

public static void main(String[] args)

{

try {

Super s=new Sub();

s.show();

}

catch(Exception e){}

}

}

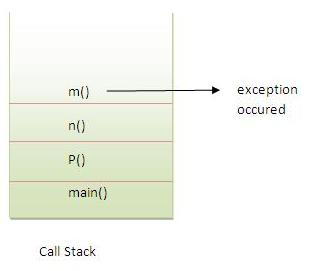
# Java Exception propagation

#### Rule: By default Unchecked Exceptions are forwarded in calling chain (propagated).

1. **class** TestExceptionPropagation1{
2. **void** m(){
3. **int** data=50/0;
4. }
5. **void** n(){
6. m();
7. }
8. **void** p(){
9. **try**{
10. n();
11. }**catch**(Exception e){System.out.println("exception handled");}
12. }
13. **public** **static** **void** main(String args[]){
14. TestExceptionPropagation1 obj=**new** TestExceptionPropagation1();
15. obj.p();
16. System.out.println("normal flow...");
17. }
18. }

Output:exception handled

normal flow...



In the above example exception occurs in m() method where it is not handled,so it is propagated to previous n() method where it is not handled, again it is propagated to p() method where exception is handled.

#### Rule: By default, Checked Exceptions are not forwarded in calling chain (propagated).

1. **class** TestExceptionPropagation2{
2. **void** m(){
3. **throw** **new** java.io.IOException("device error");//checked exception
4. }
5. **void** n(){
6. m();
7. }
8. **void** p(){
9. **try**{
10. n();
11. }**catch**(Exception e){System.out.println("exception handeled");}
12. }
13. **public** **static** **void** main(String args[]){
14. TestExceptionPropagation2 obj=**new** TestExceptionPropagation2();
15. obj.p();
16. System.out.println("normal flow");
17. }
18. }

Sort an Array

String[] fruits = new String[] {"Pineapple","Apple", "Orange", "Banana"};

Arrays.sort(fruits);

int i=0;

for(String temp: fruits){

System.out.println("fruits " + ++i + " : " + temp);

}

fruits 1 : Apple

fruits 2 : Banana

fruits 3 : Orange

fruits 4 : Pineapple

# Sort an ArrayList:

To sort an ArrayList, use the **Collections.sort()**.

List<String> fruits = new ArrayList<String>();

fruits.add("Pineapple");

fruits.add("Apple");

fruits.add("Orange");

fruits.add("Banana");

Collections.sort(fruits);

int i=0;

for(String temp: fruits){

System.out.println("fruits " + ++i + " : " + temp);

}

Result: fruits 1 : Apple, fruits 2 : Banana, fruits 3 : Orange, fruits 4 : Pineapple

# Sort an Object with Comparable

To sort an Object by its property, you have to make the Object implement the **Comparable** interface and override the**compareTo()** method.

public class Fruit implements Comparable<Fruit>{

private String fruitName;

private String fruitDesc;

private int quantity;

public Fruit(String fruitName, String fruitDesc, int quantity) {

super();

this.fruitName = fruitName;

this.fruitDesc = fruitDesc;

this.quantity = quantity;

}

public String getFruitName() {

return fruitName;

}

public void setFruitName(String fruitName) {

this.fruitName = fruitName;

}

public String getFruitDesc() {

return fruitDesc;

}

public void setFruitDesc(String fruitDesc) {

this.fruitDesc = fruitDesc;

}

public int getQuantity() {

return quantity;

}

public void setQuantity(int quantity) {

this.quantity = quantity;

}

public int compareTo(Fruit compareFruit) {

int compareQuantity = ((Fruit) compareFruit).getQuantity();

//ascending order

return this.quantity - compareQuantity;

//descending order

//return compareQuantity - this.quantity;

}

}

The **compareTo()** method is hard to explain, in integer sorting, just remember

1. this.quantity – compareQuantity is ascending order.
2. compareQuantity – this.quantity is descending order.

To understand more about compareTo() method,

<https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html>

# 4.Sort an Object with Comparator

How about sorting with Fruit’s “fruitName” or “Quantity”? The Comparable interface is only allow to sort a single property. To sort with multiple properties, you need **Comparator.**

import java.util.Comparator;

public class Fruit implements Comparable<Fruit>{

private String fruitName;

private String fruitDesc;

private int quantity;

public Fruit(String fruitName, String fruitDesc, int quantity) {

super();

this.fruitName = fruitName;

this.fruitDesc = fruitDesc;

this.quantity = quantity;

}

public String getFruitName() {

return fruitName;

}

public void setFruitName(String fruitName) {

this.fruitName = fruitName;

}

public String getFruitDesc() {

return fruitDesc;

}

public void setFruitDesc(String fruitDesc) {

this.fruitDesc = fruitDesc;

}

public int getQuantity() {

return quantity;

}

public void setQuantity(int quantity) {

this.quantity = quantity;

}

public int compareTo(Fruit compareFruit) {

int compareQuantity = ((Fruit) compareFruit).getQuantity();

//ascending order

return this.quantity - compareQuantity;

//descending order

//return compareQuantity - this.quantity;

}

public static Comparator<Fruit> FruitNameComparator

= new Comparator<Fruit>() {

public int compare(Fruit fruit1, Fruit fruit2) {

String fruitName1 = fruit1.getFruitName().toUpperCase();

String fruitName2 = fruit2.getFruitName().toUpperCase();

//ascending order

return fruitName1.compareTo(fruitName2);

//descending order

//return fruitName2.compareTo(fruitName1);

}

};

}

The Fruit class contains a static **FruitNameComparator** method to compare the “fruitName”. Now the Fruit object is able to sort with either “quantity” or “fruitName” property. Run it again.

1. Sort Fruit array based on its “fruitName” property in ascending order.

Java

Arrays.sort(fruits, Fruit.FruitNameComparator);

Output

Bash

fruits 1 : Apple, Quantity : 100

fruits 2 : Banana, Quantity : 90

fruits 3 : Orange, Quantity : 80

fruits 4 : Pineapple, Quantity : 70

2. Sort Fruit array based on its “quantity” property in ascending order.

Java

Arrays.sort(fruits)

Output

Bash

fruits 1 : Pineapple, Quantity : 70

fruits 2 : Orange, Quantity : 80

fruits 3 : Banana, Quantity : 90

fruits 4 : Apple, Quantity : 100

The **java.lang.Comparable** and **java.util.Comparator** are powerful but take time to understand and make use of it, may be it’s due to the lacking of detail example.

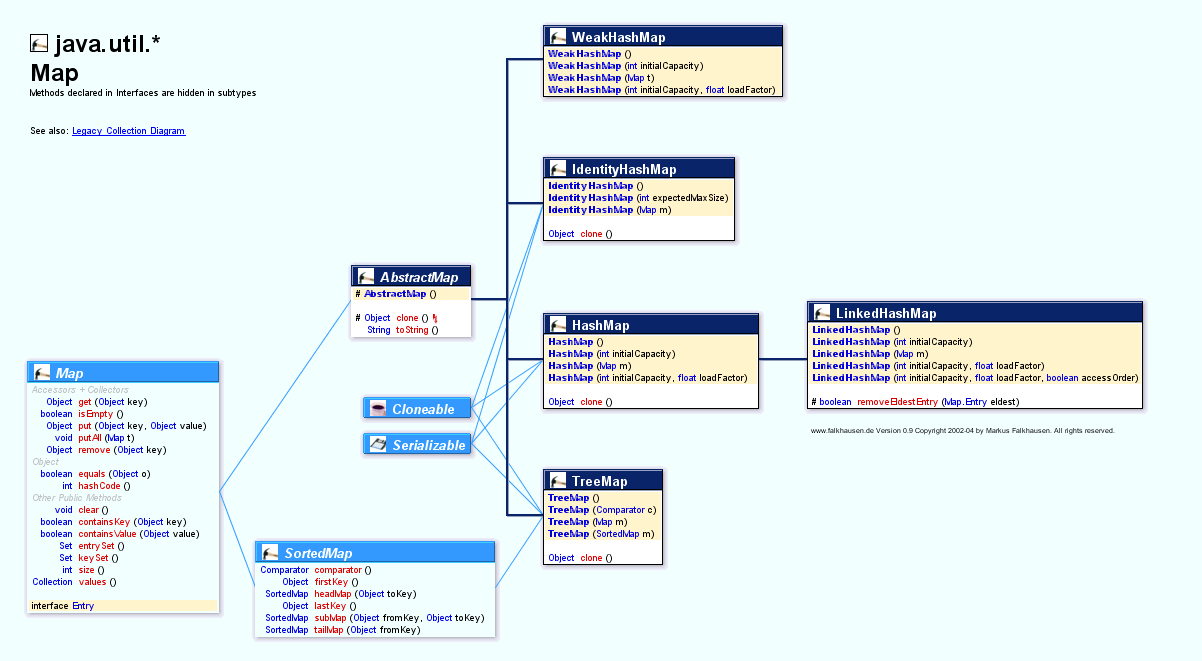
**CompareTo:**

1. A string is less than another if it comes before the other is dictionary order.
2. A string is greater than another if it comes before the other in dictionary order.
3. A string is equal than it will come order.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| slno | | Class name/type | Descrption | |
| 1 | | **Queue(I) Extends Collection**  *Throw Exp –Ret spcl val*  Add(e) – offer(e)  Remove()– pull()  Element() – peek() | Is inserting element in **FIFO** manner, it provide additional insertion, extraction, and inspection operations. Each of these methods exists in two forms:   1. one throws an exception if the operation fails. 2. the other returns a special value (either null or false, depending on the operation) | |
| 2 | | **Blocking Queue(I) Extends Queue**  Same as Queue except, introducing two more methods for **blocking**.  Put(e).  Take(); | That wait for the queue to become non-empty when retrieving an element, and wait for space to become available in the queue when storing an element.   1. One throws an exception. 2. The second returns a special value (either null or false, depending on the operation). 3. The third blocks the current thread indefinitely until the operation can succeed. 4. And the fourth blocks for only a given maximum time limit before giving up. | |
| 3 | | **TransferQueue(I) Extands Blocking Queue.**  Same as Blocking queue but its introduce some more methods.  [**transfer**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/TransferQueue.html#transfer-E-)**(**[**E**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/TransferQueue.html)**e)**  **try**[**transfer**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/TransferQueue.html#transfer-E-)**(**[**E**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/TransferQueue.html)**e)**  **try**[**transfer**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/TransferQueue.html#transfer-E-)([**E**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/TransferQueue.html) e, timeslice) | In short, BlockingQueue guarantees that the element made by producer must be in the queue, while TransferQueue gets one step further, it guarantees that the element "consumed" by some consumer.  A TransferQueue may be useful for example in message passing applications in which producers sometimes (using method[transfer(E)](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/TransferQueue.html#transfer-E-)) await receipt of elements by consumers invoking take or poll, while at other times enqueue elements (via method put) without waiting for receipt. [Non-blocking](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/TransferQueue.html#tryTransfer-E-) and [time-out](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/TransferQueue.html#tryTransfer-E-long-java.util.concurrent.TimeUnit-) versions of tryTransferare also available. A TransferQueue may also be queried, via [hasWaitingConsumer()](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/TransferQueue.html#hasWaitingConsumer--), whether there are any threads waiting for items, which is a converse analogy to a peek operation | |
|  | Class | | | Deception | |
| 4 | public class ArrayBlockingQueue<E>  extends [AbstractQueue](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractQueue.html)<E>  implements [BlockingQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html)<E>, [Serializable](https://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html) | | | This queue orders elements FIFO (first-in-first-out), | |
| 5 | public class LinkedBlockingQueue<E>  extends [AbstractQueue](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractQueue.html)<E>  implements [BlockingQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html)<E>, [Serializable](https://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html) | | | The LinkedBlockingQueue keeps the elements internally in a linked structure (linked nodes). This linked structure can optionally have an upper bound if desired. If no upper bound is specified, Integer.MAX\_VALUEis used as the upper bound. | |
| 6 | public class **PriorityBlockingQueue<E>**  extends [AbstractQueue](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractQueue.html)<E>  implements [BlockingQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html)<E>, [Serializable](https://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html) | | | The PriorityBlockingQueue is an unbounded concurrent queue. It uses the same ordering rules as thejava.util.PriorityQueue class. You cannot insert null into this queue. ssAll elements inserted into the PriorityBlockingQueue must implement the java.lang.Comparable interface. The elements thus order themselves according to whatever priority you decide in your Comparableimplementation. | |
| 7 | public class **SynchronousQueue<E>**  extends [AbstractQueue](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractQueue.html)<E>  implements [BlockingQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html)<E>, [Serializable](https://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html) | | | The SynchronousQueue is a queue that can only contain a single element internally. A thread inseting an element into the queue is blocked until another thread takes that element from the queue. Likewise, if a thread tries to take an element and no element is currently present, that thread is blocked until a thread insert an element into the queue. | |

## Deque(Double Ended queue):

|  |  |  |
| --- | --- | --- |
| Slno | Class | Deception |
| 1 | public interface **Deque<E>**  extends [Queue](https://docs.oracle.com/javase/8/docs/api/java/util/Queue.html)<E>  addFirst(), addLast()  pull()First(), pullLast(), removeFirst(), removeLast()  offerFirst(), offerLast()  getFirst(), getLast() | A linear collection that supports element insertion and removal at both ends |
| 2 | public class LinkedBlockingQueue<E>  extends [AbstractQueue](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractQueue.html)<E>  implements [BlockingQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html)<E>, [Serializable](https://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html) | The LinkedBlockingQueue keeps the elements internally in a linked structure (linked nodes). This linked structure can optionally have an upper bound if desired. If no upper bound is specified, Integer.MAX\_VALUEis used as the upper bound. |
| 3 | public class **PriorityBlockingQueue<E>**  extends [AbstractQueue](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractQueue.html)<E>  implements [BlockingQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html)<E>, [Serializable](https://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html) | The PriorityBlockingQueue is an unbounded concurrent queue. It uses the same ordering rules as thejava.util.PriorityQueue class. You cannot insert null into this queue. ssAll elements inserted into the PriorityBlockingQueue must implement the java.lang.Comparable interface. The elements thus order themselves according to whatever priority you decide in your Comparableimplementation. |
| 4 | public class **SynchronousQueue<E>**  extends [AbstractQueue](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractQueue.html)<E>  implements [BlockingQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html)<E>, [Serializable](https://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html) | The SynchronousQueue is a queue that can only contain a single element internally. A thread inseting an element into the queue is blocked until another thread takes that element from the queue. Likewise, if a thread tries to take an element and no element is currently present, that thread is blocked until a thread insert an element into the queue. |

****

1. **Hash Map:**

**package** com.collection;

**public** **class** HashMapCustom<K, V> {

**private** Entry<K,V>[] table; //Array of Entry.

**private** **int** capacity= 4; //Initial capacity of HashMap

**static** **class** Entry<K, V> {

K key;

V value;

Entry<K,V> next;

**public** Entry(K key, V value, Entry<K,V> next){

**this**.key = key;

**this**.value = value;

**this**.next = next;

}

}

@SuppressWarnings("unchecked")

**public** HashMapCustom(){

table = **new** Entry[capacity];

}

/\*\*

\* Method allows you put key-value pair in HashMapCustom.

\* If the map already contains a mapping for the key, the old value is replaced.

\* Note: method does not allows you to put null key though it allows null values.

\* Implementation allows you to put custom objects as a key as well.

\* Key Features: implementation provides you with following features:-

\* >provide complete functionality how to override equals method.

\* >provide complete functionality how to override hashCode method.

\* **@param** newKey

\* **@param** data

\*/

**public** **void** put(K newKey, V data){

**if**(newKey==**null**)

**return**; //does not allow to store null.

//calculate hash of key.

**int** hash=hash(newKey);

//create new entry.

Entry<K,V> newEntry = **new** Entry<K,V>(newKey, data, **null**);

//if table location does not contain any entry, store entry there.

**if**(table[hash] == **null**){

table[hash] = newEntry;

}**else**{

Entry<K,V> previous = **null**;

Entry<K,V> current = table[hash];

**while**(current != **null**){ //we have reached last entry of bucket.

**if**(current.key.equals(newKey)){

**if**(previous==**null**){ //node has to be insert on first of bucket.

newEntry.next=current.next;

table[hash]=newEntry;

**return**;

}

**else**{

newEntry.next=current.next;

previous.next=newEntry;

**return**;

}

}

previous=current;

current = current.next;

}

previous.next = newEntry;

}

}

/\*\*

\* Method returns value corresponding to key.

\* **@param** key

\*/

**public** V get(K key){

**int** hash = hash(key);

**if**(table[hash] == **null**){

**return** **null**;

}**else**{

Entry<K,V> temp = table[hash];

**while**(temp!= **null**){

**if**(temp.key.equals(key))

**return** temp.value;

temp = temp.next; //return value corresponding to key.

}

**return** **null**; //returns null if key is not found.

}

}

/\*\*

\* Method removes key-value pair from HashMapCustom.

\* **@param** key

\*/

**public** **boolean** remove(K deleteKey){

**int** hash=hash(deleteKey);

**if**(table[hash] == **null**){

**return** **false**;

}**else**{

Entry<K,V> previous = **null**;

Entry<K,V> current = table[hash];

**while**(current != **null**){ //we have reached last entry node of bucket.

**if**(current.key.equals(deleteKey)){

**if**(previous==**null**){ //delete first entry node.

table[hash]=table[hash].next;

**return** **true**;

}

**else**{

previous.next=current.next;

**return** **true**;

}

}

previous=current;

current = current.next;

}

**return** **false**;

}

}

/\*\*

\* Method displays all key-value pairs present in HashMapCustom.,

\* insertion order is not guaranteed, for maintaining insertion order

\* refer LinkedHashMapCustom.

\* **@param** key

\*/

**public** **void** display(){

**for**(**int** i=0;i<capacity;i++){

**if**(table[i]!=**null**){

Entry<K, V> entry=table[i];

**while**(entry!=**null**){

System.*out*.print("{"+entry.key+"="+entry.value+"}" +" ");

entry=entry.next;

}

}

}

}

/\*\*

\* Method implements hashing functionality, which helps in finding the appropriate

\* bucket location to store our data. This is very important method, as performance of HashMapCustom is very much

\* dependent on this method's implementation.

\* **@param** key

\*/

**private** **int** hash(K key){

**return** Math.*abs*(key.hashCode()) % capacity;

}

}

**package** com.collection;

**public** **class** HashMapCustomMain {

/\*\*

\* **@param** args

\*/

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

// **TODO** Auto-generated method stub

HashMapCustom<Integer, Integer> hashMapCustom = **new** HashMapCustom<Integer, Integer>();

hashMapCustom.put(21, 12);

hashMapCustom.put(25, 121);

hashMapCustom.put(30, 151);

hashMapCustom.put(33, 15);

hashMapCustom.put(35, 89);

System.*out*.println("value corresponding to key 21="

+ hashMapCustom.get(21));

System.*out*.println("value corresponding to key 51="

+ hashMapCustom.get(51));

System.*out*.print("Displaying : ");

hashMapCustom.display();

System.*out*.println("\n\nvalue corresponding to key 21 removed: "

+ hashMapCustom.remove(21));

System.*out*.println("value corresponding to key 51 removed: "

+ hashMapCustom.remove(51));

System.*out*.print("Displaying : ");

hashMapCustom.display();

}

}

Linked List :

**package** com.collection;

**public** **class** OwnLinkedList {

**private** **static** **int** *counter*;

**private** Node head;

// Default constructor

**public** OwnLinkedList() {}

// appends the specified element to the end of this list.

**public** **void** add(Object data) {

// Initialize Node only incase of 1st element

**if** (head == **null**) {

head = **new** Node(data);

}

Node crunchifyTemp = **new** Node(data);

Node crunchifyCurrent = head;

// Let's check for NPE before iterate over crunchifyCurrent

**if** (crunchifyCurrent != **null**) {

**while** (crunchifyCurrent.getNext() != **null**) {

crunchifyCurrent = crunchifyCurrent.getNext();

}

// the last node's "next" reference set to our new node

crunchifyCurrent.setNext(crunchifyTemp);

}

// increment the number of elements variable

*incrementCounter*();

}

**public** Object get(**int** index)

// returns the element at the specified position in this list.

{

// index must be 1 or higher

**if** (index <= 0)

**return** **null**;

Node crunchifyCurrent = **null**;

**if** (head != **null**) {

crunchifyCurrent = head.getNext();

**for** (**int** i = 0; i < index; i++) {

**if** (crunchifyCurrent.getNext() == **null**)

**return** **null**;

crunchifyCurrent = crunchifyCurrent.getNext();

}

**return** crunchifyCurrent.getData();

}

**return** crunchifyCurrent;

}

// removes the element at the specified position in this list.

**public** **boolean** remove(**int** index) {

// if the index is out of range, exit

**if** (index < 1 || index > size())

**return** **false**;

Node crunchifyCurrent = head;

**if** (head != **null**) {

**for** (**int** i = 0; i < index; i++) {

**if** (crunchifyCurrent.getNext() == **null**)

**return** **false**;

crunchifyCurrent = crunchifyCurrent.getNext();

}

crunchifyCurrent.setNext(crunchifyCurrent.getNext().getNext());

// decrement the number of elements variable

decrementCounter();

**return** **true**;

}

**return** **false**;

}

**public** **int** size() {

**return** *getCounter*();

}

**private** **static** **int** getCounter() {

**return** *counter*;

}

**private** **static** **void** incrementCounter() {

*counter*++;

}

**private** **void** decrementCounter() {

*counter*--;

}

}

**Node.java**

**package** com.collection;

**public** **class** Node {

// reference to the next node in the chain, or null if there isn't one.

Node next;

Object data;

// Node constructor

**public** Node(Object dataValue) {

next = **null**;

data = dataValue;

}

// another Node constructor if we want to specify the node to point to.

@SuppressWarnings("unused")

**public** Node(Object dataValue, Node nextValue) {

next = nextValue;

data = dataValue;

}

// these methods should be self-explanatory

**public** Object getData() {

**return** data;

}

@SuppressWarnings("unused")

**public** **void** setData(Object dataValue) {

data = dataValue;

}

**public** Node getNext() {

**return** next;

}

**public** **void** setNext(Node nextValue) {

next = nextValue;

}

}

**Main.java**

**public** **class** OwnLinkedListMain {

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

OwnLinkedList list = **new** OwnLinkedList();

list.add("10");

list.add("20");

System.*out*.println("list value"+list.get(1));

}

}

**Difference between IdentityHashMap and HashMap**

1) The main difference between *HashMap vs IdentityHashMap* is that IdentityHashMap uses equality operator "==" for comparing keys and values inside Map while HashMap uses [equals method](http://javarevisited.blogspot.com/2012/12/difference-between-equals-method-and-equality-operator-java.html) for comparing keys and values.

2) Unlike HashMap, who uses hashcode to find bucket location, IdentityHashMap also doesn't use hashCode() instead it uses System.identityHashCode(object).

3) Another key difference between IdentityHashMap and HashMap in Java is Speed. Since IdentityHashMap doesn't useequals() its comparatively faster than HashMap for object with expensive [equals() and hashCode().](http://java67.blogspot.sg/2012/11/difference-between-operator-and-equals-method-in.html)

4) One more difference between HashMap and IdentityHashMap is Immutability of the key. One of the basic requirement to safely store [Objects](http://javarevisited.blogspot.com/2012/12/what-is-object-in-java-or-oops-example.html) in HashMap is keys needs to be immutable, IdentityHashMap doesn't require keys to be immutable as it is not relied on equals and hashCode.

There is also a class called IdentityHashtable which is analogous to [Hashtable in Java](http://javarevisited.blogspot.com/2012/01/java-hashtable-example-tutorial-code.html) but it’s not part of standard JDK and available in com.sun... package.

**Example of IdentityHashMap in Java**

Here is an example of IdentityHashMap in Java which shows the key difference between HashMap and IdentityHashMap in comparing Objects.  IdentityHashMap uses equality operator for comparison instead of equals method in Java :

**import** java.util.IdentityHashMap;  
  
/\*\*  
 \* Java program to show difference between HashMap and IdentityHashMap in Java  
 \* @author Javin Paul  
 \*/  
**public** **abstract** **class** Testing {  
  
   **public** **static** **void** main(**String** args[]) {  
        **IdentityHashMap**<**String**, **String**> identityMap = **new** **IdentityHashMap**<**String**, **String**>();  
        
        identityMap.put("sony", "bravia");  
        identityMap.put(**new** **String**("sony"), "mobile");  
        
       *//size of identityMap should be 2 here because two strings are different objects*  
        **System**.out.println("Size of IdentityHashMap: " + identityMap.size());  
        **System**.out.println("IdentityHashMap: " + identityMap);  
        
        identityMap.put("sony", "videogame");  
        
         *//size of identityMap still should be 2 because "sony" and "sony" is same object*  
        **System**.out.println("Size of IdentityHashMap: " + identityMap.size());  
        **System**.out.println("IdentityHashMap: " + identityMap);  
      
   }  
}  
  
Output  
Size of **IdentityHashMap**: 2  
**IdentityHashMap**: {sony=bravia, sony=mobile}  
Size of **IdentityHashMap**: 2  
**IdentityHashMap**: {sony=videogame, sony=mobile}

That’s all on the difference between IdentityHashMap and HashMap in Java.  As I said IdentityHashMap violates Map interface general contract and should only be used when reference equality makes sense. As per Javadoc, IdentityHashMap is suitable to keep object reference during Serialization and deep copy and can also be used  to maintain as a proxy object.

**Data Structure Top Questions**

**1: How to find middle element of linked list in one pass?**

In order to find middle element of linked list in one pass, you need to maintain two-pointer, one increment at each node while other increments after two nodes at a time, by having this arrangement, when first pointer reaches end, second pointer will point to middle element of linked list. See this trick to [find middle element of linked list in single pass](http://javarevisited.blogspot.com/2012/12/how-to-find-middle-element-of-linked-list-one-pass.html) for more details.

### import test.LinkedList.Node; /\*\*  \* Java program to find middle element of linked list in one pass.  \* In order to find middle element of linked list we need to find length first  \* but since we can only traverse linked list one time, we will use two pointers  \* one which we will increment on each iteration while other which will be  \* incremented every second iteration. so when first pointer will point to the  \* end of linked list, second will be pointing to the middle element of linked list  \* @author  \*/ public class LinkedListTest {           public static void main(String args[]) {         *//creating LinkedList with 5 elements including head*       LinkedList linkedList = new LinkedList();       LinkedList.Node head = linkedList.head();       linkedList.add( new LinkedList.Node("1"));       linkedList.add( new LinkedList.Node("2"));       linkedList.add( new LinkedList.Node("3"));       linkedList.add( new LinkedList.Node("4"));            *//finding middle element of LinkedList in single pass*       LinkedList.Node current = head;       int length = 0;       LinkedList.Node middle = head;            while(current.next() != null){           length++;           if(length%2 ==0){               middle = middle.next();           }           current = current.next();       }            if(length%2 == 1){           middle = middle.next();       }       System.out.println("length of LinkedList: " + length);       System.out.println("middle element of LinkedList : " + middle);            }     } class LinkedList{     private Node head;     private Node tail;        public LinkedList(){         this.head = new Node("head");         tail = head;     }        public Node head(){         return head;     }        public void add(Node node){         tail.next = node;         tail = node;     }        public static class Node{         private Node next;         private String data;         public Node(String data){             this.data = data;         }                public String data() {             return data;         }         public void setData(String data) {             this.data = data;         }         public Node next() {             return next;         }         public void setNext(Node next) {             this.next = next;         }                public String toString(){             return this.data;         }     } } Output: length of LinkedList: 4 middle element of LinkedList : 2 2: How to find if linked list has a loop ?

This question has bit of similarity with earlier algorithm and data structure interview question. I mean we can use two pointer approach to solve this problem. If we maintain two pointers, and we increment one pointer after processing two nodes and other after processing every node, we are likely to find a situation where both the pointers will be pointing to same node. This will only happen if linked list has loop.

### 3 : How to find 3rd element from end in a linked list in one pass?

This is another frequently asked linked list interview question. This question is exactly similar to[finding middle element of linked list in single pass](http://javarevisited.blogspot.sg/2012/12/how-to-find-middle-element-of-linked-list-one-pass.html). If we apply same trick of maintaining two pointers and increment other pointer, when first has moved up to 3rd element, than when first pointer reaches to the end of linked list, second pointer will be pointing to the 3rd element from last in a linked list.

### 4: In an integer array, there is 1 to 100 number, out of one is duplicate, how to find?

This is a rather simple data structures question, especially for this kind of. In this case you can simply add all numbers stored in array, and total sum should be equal to n(n+1)/2. Now just subtract actual sum to expected sum, and that is your duplicate number. Of course there is a brute force way of checking each number against all other numbers, but that will result in performance of O(n^2) which is not good. By the way this trick will not work if array have multiple duplicates or its not numbers forming arithmetic progression. Here is example of one way to [find duplicate number in array](http://javarevisited.blogspot.com/2012/02/how-to-check-or-detect-duplicate.html).

In this Java tutorial, we will see a couple of ways to find if an array contains duplicates or not in Java. We will use the unique property of Java collection class Set which doesn’t allow duplicates to check java array for duplicate elements.  Here are five ways we can check if an array has duplicates or not:

1) **brute force method**which compares each element of Array to all other elements and returns true if it founds duplicates. Though this is not an efficient choice it is the one which first comes to mind.

2) Another quick way of checking if a Java array contains duplicates or not is to**convert that array into Set**. Since Set doesn’t allow duplicates size of  the corresponding Set will be smaller than original Array if Array contains duplicates otherwise the size of both Array and Set will be same.

3) One more way to detect duplication in java array is adding every element of the array into HashSet which is a Set implementation. Since the add(Object obj) method of Set returns false if Set already contains an element to be added, it can be used to find out if the array contains duplicates in Java or not.

In next section, we will complete code example of all three ways of **duplicate detection on Array in java**. Remember this discussion is just confirming whether an array contains duplicate or not , it's not finding out actual duplicate elements from Array though you can easily extend example Java program to accomplish that task based on your requirement.   
  
  
This is also one of the popular programming interviews questions, asked in several interviews. I also suggest you to solves problems from [Cracking the Coding Interview: 189 Programming Questions and Solutions](http://www.amazon.com/dp/098478280X/?tag=javamysqlanta-20). One of the best book to prepare for software developer interviews.

## Code Example of checking duplicate on Array in Java

Here is complete code sample of all above methods to check if your array contains duplicates or not.

**import** java.util.Arrays;

**import** java.util.HashSet;

**import** java.util.List;

**import** java.util.Set;

**public** **class** **CheckDuplicatesInJavaArray** {

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

       String[] withDuplicates = **new** String[] {"one","two","three","one"};

        String[] withoutDuplicates = **new** String[] {"one","two","three"};

    System.*out*.println("Checking array with duplicate using brute force: " + *bruteforce*(withDuplicates));

       System.*out*.println("Checking array without any duplicate using brute force: " + *bruteforce*(withoutDuplicates));

     System.*out*.println("Checking array with duplicate using Set and List: " + *checkDuplicateUsingSet*(withDuplicates));

        System.*out*.println("Checking array without any duplicate using Set and List: " + *checkDuplicateUsingSet*(withoutDuplicates));

       System.*out*.println("Checking array with duplicate using Set and List: " + *checkDuplicateUsingAdd*(withDuplicates));

        System.*out*.println("Checking array without any duplicate using Set and List: " + *checkDuplicateUsingAdd*(withoutDuplicates));

    }

    /\*

     \* brute force way of checking if array contains duplicates in Java

     \* comparing each element to all other elements of array

     \* complexity on order of O(n^2) not advised in production

     \*/

**public** **static** **boolean** bruteforce(String[] input) {

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

**for** (**int** j = 0; j < input.length; j++) {

**if** (input[i].equals(input[j]) && i != j) {

**return** **true**;

                }

            }

        }

**return** **false**;

    }

    /\*

     \* detect duplicate in array by comparing size of List and Set

     \* since Set doesn't contain duplicate, size must be less for an array which contains duplicates

     \*/

**public** **static** **boolean** checkDuplicateUsingSet(String[] input){

        List inputList = Arrays.*asList*(input);

        Set inputSet = **new** HashSet(inputList);

**if**(inputSet.size()< inputList.size())

**return** **true**;

        }

**return** **false**;

    }

    /\*

     \* Since Set doesn't allow duplicates add() return false

     \* if we try to add duplicates into Set and this property

     \* can be used to check if array contains duplicates in Java

     \*/

**public** **static** **boolean** checkDuplicateUsingAdd(String[] input) {

        Set tempSet = **new** HashSet();

**for** (String str : input) {

**if** (!tempSet.add(str)) {

**return** **true**;

            }

        }

**return** **false**;

    }

}

**Output:**

Checking array with duplicate using brute force: true

Checking array without any duplicate using brute force: false

Checking array with duplicate using Set and List: true

Checking array without any duplicate using Set and List: false

Checking array with duplicate using Set and List: true

Checking array without any duplicate using Set and List: false

## 4: Write a Java program to sort an array using Bubble Sort algorithm?

I have always send couple of questions from searching and sorting in data structure interviews. Bubble sort is one of the simplest sorting algorithm but if you ask anyone to implement on the spot it gives you an opportunity to gauge programming skills of a candidate. See [How to sort array using Bubble Sort in Java](http://java67.blogspot.com/2012/12/bubble-sort-in-java-program-to-sort-integer-array-example.html) for complete solution of this datastrucutre interview question.

public class BubbleSort {  
    
    
    public static void main(String args[]) {  
         int[] unsorted = {32, 39,21, 45, 23, 3};  
        bubbleSort(unsorted);  
        
        int[] test = { 5, 3, 2, 1};  
        bubbleSort(test);  
        
    }     
    
    public static void bubbleSort(int[] unsorted){  
        System.out.println("unsorted array before sorting : " + Arrays.toString(unsorted));  
       
        for(int i=0; i<unsorted.length -1; i++){  
            
             for(int j= 1; j<unsorted.length -i; j++){  
                
                 if(unsorted[j-1] > unsorted[j]){  
                    int temp = unsorted[j];  
                    unsorted[j] = unsorted[j-1];  
                    unsorted[j-1] = temp;  
                }  
            }  
            System.out.printf("unsorted array after %d pass %s: %n", i+1,Arrays.toString(unsorted));  
        }  
    }  
}

### 6: How do you find duplicates in an array if there is more than one duplicate?

Sometime this is asked as follow-up question of earlier data structure interview question, related to finding duplicates in Array. One way of solving this problem is using a [Hashtable or HashMap](http://javarevisited.blogspot.com/2013/02/how-to-get-key-from-value-in-hashtable.html) data structure. You can traverse through array, and store each number as key and number of occurrence as value. At the end of traversal you can find all duplicate numbers, for which occurrence is more than one. In Java if a number already exists in[HashMap](http://java67.blogspot.com/2013/02/10-examples-of-hashmap-in-java-programming-tutorial.html) then calling get(index) will return number otherwise it return null. this property can be used to insert or update numbers in HashMap.

**7 : Write Java program to print Fibonacci series ?**

public static int fibonacci2(int number){

if(number == 1 || number == 2){

return 1;

}

int fibo1=1, fibo2=1, fibonacci=1;

for(int i= 3; i<= number; i++){

//Fibonacci number is sum of previous two Fibonacci number

fibonacci = fibo1 + fibo2;

fibo1 = fibo2;

fibo2 = fibonacci;

}

return fibonacci; //Fibonacci number

}

### 8: What is binary search tree?

This is a data structure question from Tree data structures. Binary Search Tree has some special properties e.g. left nodes contains items whose value is less than root , right sub tree contains keys with higher node value than root, and there should not be any duplicates in the tree. Apart from definition, interview can ask you to implement binary search tree in Java and questions on tree traversal e.g. IN order, preorder, and post order traversals are quite popular data structure question.