**Stack vs Heap in Java**

1) 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](http://javarevisited.blogspot.com/2012/12/what-is-object-in-java-or-oops-example.html). No matter, where object is created in code e.g. as 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 there 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 heap and -Xmx is maximum size of java heap

3) If there is no memory left in stack for storing function call or local variable, JVM will throw java.lang.StackOverFlowError, while if there is no more heap space for creating object, JVM will throw java.lang.OutOfMemoryError: Java Heap Space.

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 lot lesser than size of  heap memory in Java.

5) Variables stored in stacks are only visible to the owner Thread, while objects created in 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.

**Why String is immutable or final in Java**

1) Imagine StringPool facility without making string immutable , its not possible at all because in case of string pool one string object/literal e.g. "Test" has referenced by many [reference variables](http://javarevisited.blogspot.sg/2012/02/difference-between-instance-class-and.html) , so if any one of them change the value others will be automatically gets affected i.e. lets say  
  
String A = "Test"  
String B = "Test"   
  
Now String B called "Test".toUpperCase() which change the same object into "TEST" , so A will also be "TEST" which is not desirable.  
  
2)String has been widely used as parameter for many Java classes e.g. for opening network connection, you can pass hostname and port number as string , you can pass database URL as string for opening database connection, you can [open any file in Java](http://javarevisited.blogspot.sg/2012/07/read-file-line-by-line-java-example-scanner.html) by passing name of file as argument to File I/O classes.  
  
In case, if String is not immutable, this would lead serious security threat , I mean some one can access to any file for which he has authorization, and then can change the file name either deliberately or accidentally and gain access of those file. Because of immutability, you don't need to worry about those kind of threats. This reason also gel with, **Why String is final in Java**, by making java.lang.String final, Java designer ensured that no one overrides any behavior of String class.  
  
3)Since String is immutable it can safely shared between many threads ,which is very important for multithreaded programming and to avoid any [synchronization issues in Java](http://javarevisited.blogspot.com/2011/04/synchronization-in-java-synchronized.html), Immutability also makes String instance [thread-safe in Java](http://javarevisited.blogspot.sg/2012/01/how-to-write-thread-safe-code-in-java.html), means you don't need to synchronize String operation externally. Another important point to note about String is[memory leak caused by SubString](http://javarevisited.blogspot.sg/2011/10/how-substring-in-java-works.html), which is not a thread related issues but something to be aware of.  
  
4) Another reason of **Why String is immutable in Java** is to **allow String to cache its hashcode** , being immutable String in Java caches its hashcode, and do not calculate every time we call hashcode method of String, which makes it very fast as hashmap key to be used in [hashmap in Java](http://javarevisited.blogspot.com/2011/02/how-hashmap-works-in-java.html).  This one is also suggested by  Jaroslav Sedlacek in comments below. In short because String is immutable, no one can change its contents once created which guarantees [hashCode](http://javarevisited.blogspot.sg/2011/10/override-hashcode-in-java-example.html)of String to be same on multiple invocation.  
  
5) The absolutely most important reason that String is immutable is that it is used by the [class loading mechanism](http://javarevisited.blogspot.sg/2012/07/when-class-loading-initialization-java-example.html), and thus have profound and fundamental security aspects. Had String been mutable, a request to load "java.io.Writer" could have been changed to load "mil.vogoon.DiskErasingWriter".

**When a class is loaded and initialized in JVM   
  
When Class is loaded in Java**

Class loading is done by ClassLoaders in Java which can be implemented to eagerly load a class as soon as another class references it or [lazy load](http://javarevisited.blogspot.sg/2012/07/why-enum-singleton-are-better-in-java.html) the class until a need of class initialization occurs. If Class is loaded before its actually being used it can sit inside before being initialized. I believe this may vary from JVM to JVM. While its guaranteed by JLS that a class will be loaded when there is a need of [static initialization](http://javarevisited.blogspot.sg/2011/11/static-keyword-method-variable-java.html).

**When a Class is initialized in Java**

After class loading, initialization of class takes place which means initializing all [static members](http://javarevisited.blogspot.sg/2011/11/static-keyword-method-variable-java.html) of class. A Class is initialized in Java when:

1) An [Instance](http://javarevisited.blogspot.sg/2012/02/difference-between-instance-class-and.html) of class is created using either new() keyword or using [reflection](http://javarevisited.blogspot.sg/2012/05/how-to-access-private-field-and-method.html) using class.forName(), which may throw [ClassNotFoundException](http://javarevisited.blogspot.sg/2011/08/classnotfoundexception-in-java-example.html) in Java.

2) An static method of Class is invoked.

3) An static field of Class is assigned.

4) An static field of class is used which is not a constant variable.

5) If Class is a top level class and an [assert statement](http://javarevisited.blogspot.sg/2012/01/what-is-assertion-in-java-java.html) lexically nested within class is executed.

[Reflection](http://javarevisited.blogspot.sg/2012/04/how-to-invoke-method-by-name-in-java.html) can also cause initialization of class. Some methods of java.lang.reflect package may cause class to be initialized. JLS Strictly says that a class should not be initialized by any reason other than above.

How Class is initialized in Java

[class loading and initialization in Java - When example](http://javarevisited.blogspot.sg/2011/02/how-to-setup-remote-debugging-in.html)Now we know what triggers initialization of a class in Java, which is precisely documented in [Java language specification](http://docs.oracle.com/javase/specs/). Its also important to know in which **order** various fields ([static and non static](http://javarevisited.blogspot.sg/2012/03/mixing-static-and-non-static.html)), block (static an non static), various classes (sub class and super class) and various interfaces (sub interface, implementation class and super interface) is initialized in Java. Infact many[Core Java interview question](http://javarevisited.blogspot.sg/2011/04/top-20-core-java-interview-questions.html) and SCJP question based on this concept because it affect final value of any variable if its initialized on multiple places. Here are some of the **rules of class initialization in Java**:

1) Classes are initialized from *top to bottom* so field declared on top initialized before field declared in bottom

2) Super Class is initialized before Sub Class or derived class in Java

3) If Class initialization is triggered due to access of [static field](http://javarevisited.blogspot.sg/2011/11/static-keyword-method-variable-java.html), only Class which has declared static field is initialized and it doesn't trigger initialization of super class or sub class even if static field is referenced by Type  of Sub Class, [Sub Interface](http://javarevisited.blogspot.sg/2012/04/10-points-on-interface-in-java-with.html) or by implementation class of interface.

4) [interface initialization](http://javarevisited.blogspot.sg/2012/04/10-points-on-interface-in-java-with.html) in Java doesn't cause super interfaces to be initialized.

5) Static fields are initialized during static initialization of class while non static fields are initialized when instance of class is created. It means **static fields are initialized before non static fields in Java**.

6)non static fields are initialized by [constructors in Java](http://javarevisited.blogspot.sg/2012/01/what-is-constructor-overloading-in-java.html). sub class constructor implicitly call super class constructor before doing any initialization, which guarantees that non static or instance variables of super class is initialized before sub class.

**Examples of  class initialization in Java:**

Here is an example of when class is initialized in Java. In this example we will see which classes are initialized in Java.

/\*\*  
 \* Java program to demonstrate **class loading and initialization** in Java.  
 \*/ **public** **class** ClassInitializationTest {  
  
    **public** **static** **void** main(**String** args[]) **throws** **InterruptedException** {  
    
        NotUsed o = **null**; *//this class is not used, should not be initialized*  
        Child t = **new** Child(); *//initializing sub class, should trigger super class initialization*  
        **System**.out.println((**Object**)o == (**Object**)t);  
    }  
}  
  
/\*\*  
 \* Super class to demonstrate that Super class is loaded and initialized before Subclass.  
 \*/  
**class** Parent {  
    **static** { **System**.out.println("static block of Super class is initialized"); }  
    {**System**.out.println("non static blocks in super class is initialized");}  
}  
  
/\*\*  
 \* Java class which is not used in this program, consequently not loaded by JVM  
 \*/  
**class** NotUsed {  
    **static** { **System**.out.println("NotUsed Class is initialized "); }  
}  
  
***/\*\*  
 \* Sub class of Parent, demonstrate when exactly sub class loading and initialization occurs.  
 \*/***  
**class** Child **extends** Parent {  
    **static** { **System**.out.println("static block of Sub class is initialized in Java "); }  
    {**System**.out.println("non static blocks in sub class is initialized");}  
}  
  
**Output:**  
**static** block of Super **class** is initialized  
**static** block of Sub **class** is initialized in Java  
non **static** blocks in **super** **class** is initialized  
non **static** blocks in sub **class** is initialized  
**false**

**Observation:**

1) Super class is initialized before sub class in Java.

2) [Static variables or blocks](http://javarevisited.blogspot.sg/2012/03/what-is-static-and-dynamic-binding-in.html) are initialized before non static blocks or fields.

3) Not used class is not initialized at all because its not been used, none of the cases mentioned on JLS or above which triggers initialization of class is not happened here.

Let's have a look on another example of class initialization in Java:

/\*\*  
 \* Another Java program example to demonstrate class initialization and loading in Java.  
 \*/  
**public** **class** ClassInitializationTest {  
  
    **public** **static** **void** main(**String** args[]) **throws** **InterruptedException** {  
    
       *//accessing static field of Parent through child, should only initialize Parent*  
       **System**.out.println(Child.familyName);  
    }  
}  
  
**class** Parent {  
    *//compile time constant, accessing this will not trigger class initialization*  
    *//protected static final String familyName = "Lawson";*  
    
    **protected** **static** **String** familyName = "Lawson";  
    
    **static** { **System**.out.println("static block of Super class is initialized"); }  
    {**System**.out.println("non static blocks in super class is initialized");}  
}  
  
**Output:**  
**static** block of Super **class** is initialized  
Lawson

**Observation**

1. Here class initialization occurs because **static field is accessed** which is not a [compile time](http://javarevisited.blogspot.sg/2012/03/what-is-static-and-dynamic-binding-in.html) constant. had you declare "familyName" compile time constant using [final keyword in Java](http://javarevisited.blogspot.sg/2011/12/final-variable-method-class-java.html) (as shown in commented section) class initialization of super class would not have occurred.

2) Only super class is initialized even though static field is referenced using sub type.

There is another **example of class initialization** related to interface on JLS which explains clearly that initialization of sub interfaces does not trigger initialization of super interface. I highly recommend reading JLS 14.4 for understating class loading and initialization in more detail.