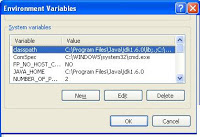
[**How to Set Classpath for Java on Windows Unix and Linux**](http://javarevisited.blogspot.com/2011/01/how-classpath-work-in-java.html)

**What is CLASSPATH in Java**  
Classpath in Java is path to directory or list of directory which is used by ClassLoaders to find and [load class in Java program](http://javarevisited.blogspot.sg/2012/07/when-class-loading-initialization-java-example.html). 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](http://javarevisited.blogspot.sg/2012/03/how-to-create-and-execute-jar-file-in.html).  In this Java tutorial we will learn What is Classpath in Java, how Java resolve classpath and How Classpath works in Java along side How to set classpath for Java in windows and UNIX environment.  I have experience in finance and insurance domain and Java is heavily used in this domain for writing sophisticated Equity, Fixed income trading applications. Most of these investment banks has written test as part of there [core Java interview questions](http://javarevisited.blogspot.sg/2011/04/top-20-core-java-interview-questions.html) and I always find at least one question related to CLASSPATH in Java on those interviews. Java CLASSPATH is one of the most important concepts in Java,  but,  I must say mostly overlooked. This should be the first thing you should learn while writing Java programs because without correct understanding of **Classpath in Java**you can't understand how Java locates your class files. Also don't confuse Classpath with [PATH in Java](http://javarevisited.blogspot.sg/2011/10/how-to-set-path-for-java-unix-linux-and.html), which is another environment variable used to find java binaries located in JDK installation directory, also known as [JAVA\_HOME](http://javarevisited.blogspot.sg/2012/02/how-to-set-javahome-environment-in.html). Main difference between PATH and CLASSPATH is that former is used to locate Java commands while later is used to locate Java class files.  
  
So let’s start with basic and then we will see some example and improvisation of Classpath in Java. In fact CLASSPATH is an [environment variable](http://javarevisited.blogspot.sg/2012/08/how-to-get-environment-variables-in.html) which is used by [Java Virtual Machine](http://javarevisited.blogspot.sg/2011/11/hotspot-jvm-options-java-examples.html) to locate user defined classes. As I said In this tutorial we will see **How to setup classpath for java in windows and Linux** , **java -classpath example** in different scenario and use of java -classpath or java -cp.

**Setting Java Classpath in Windows**

In order to set Classpath for Java in Windows (any version either Windows XP,  Windows 2000 or Windows 7) you need to specify value of environmentvariable ***CLASSPATH***, name of this variable is not case sensitive and it doesn’t matter if name of your *environment variable is*[*Classpath*](http://java67.blogspot.sg/2012/08/what-is-path-and-classpath-in-java-difference.html)*, CLASSPATH or*[*classpath*](http://javabuddy.hubpages.com/hub/What-is-PATH-and-CLASSPATH-in-Java-set-Windows-Unix-Linux)*in Java*.  
  
Here is *Step by Step guide for setting Java Classpath in Windows*:

[](http://javarevisited.blogspot.com/2011/01/how-classpath-work-in-java.html)

1. Go to Environment variable window in Windows by pressing "Windows + Pause “-->Advanced -->Environment variable " or you can go from right click on my computer than choosing properties and then Advanced and then Environment variable this will open Environment variable window in windows.
2. Now specify your environment variable CLASSPATH and put the value of your JAVA\_HOME\lib and also include [current directory](http://javarevisited.blogspot.sg/2012/04/how-to-find-current-directory-in-java.html) by including (dot or period sign).
3. Now *to check the value of Java classpath in windows* type **"echo %CLASSPATH"** in your DOS command prompt and it will show you the value of directory which are included in CLASSPATH.  
     
   You can also set classpath in windows by using DOS command like :  
     
   set CLASSPATH=%CLASSPATH%;JAVA\_HOME\lib;  
   This way you can set classpath in Windows XP, windows 2000 or Windows 7 and 8, as they all come with command prompt.

**Setting Java Classpath in UNIX or Linux**

To set Classpath for Java In Linux you can simply export CLASSPATH="your classpath" from either your ***.***bash\_profile or .bashrc script which will run whenever you login into your Linux or Unix Machine. Now to check value of Java CLASSPATH in Linux type "echo ${CLASSPATH}" this will print value of [Classpath in command prompt](http://javarevisited.blogspot.sg/2011/11/run-java-program-from-command-prompt.html). By using export command you can set classpath for Java in Unix, Linux, Solaris, IBM AIX or any other UNIX operating system. I hope this example for setting classpath in Java will enable to set classpath by yourself let me know if you face any problem while setting up classpath in Java

**Overriding Classpath in Java**

You can override classpath in Java, defined by environment variable CLASSPATH by providing option "-cp" or "-classpath" while [running Java program](http://javarevisited.blogspot.sg/2011/11/run-java-program-from-command-prompt.html) and this is the best way to have different classpath for different Java application running on same Unix or Windows machine, standard way to define classpath for Java application is creating start-up script for Java program and set classpath there as shown below :  
  
CLASSPATH=/home/tester/classes  
java -cp $CLASSPATH Test  
  
By default Java CLASSPATH points to [current directory](http://javarevisited.blogspot.sg/2012/04/how-to-find-current-directory-in-java.html) denoted by "." and it will look for any class only in current directory.

**Different example of using Classpath in Java**

In case you have multiple directories defined in CLASSPATH variable, Java will look for a class starting from first directory and only look second directory in case it did not find the specified class in first directory. This is extremely useful feature of Classpath in java to understand and it’s very useful while [debugging Java application](http://javarevisited.blogspot.sg/2011/07/java-debugging-tutorial-example-tips.html) or  patch release kind of stuff. Let’s see  java -classpath example  
  
I have set my classpath environment variable as CLASSPATH=/home/tester/first:/home/tester/second**.**Now I have Test class of different version in both first and second directory. When I give a command "java Test" What will happen ? Which Test class would be picked? Since JVM search directory in the order they have listed in CLASSPATH variable it will first go to the "first" directory and if it finds Test.class over there it will not go to /home/tester/second directory. Now if you delete Test.classfrom /home/tester/first directory it will go to /home/tester/second directory and will pick  Test.class from there.  
  
I have used this feature of Java Classpath to test my patch releases, we used to have a folder called "patch" listed as first element in Java CLASSPATH and any point of time we want to put any debug statement or want to test any bug we just modify Java source file , compile it and generate class file and put that inside patch folder instead of [creating JAR file](http://javarevisited.blogspot.sg/2012/03/how-to-create-and-execute-jar-file-in.html) and releasing whole new Java application. This is very handy if you are working in a large project where you don't have development environment setup in Windows and your project only runs on Unix server. This approach is much faster than [remote debugging Java application in Eclipse](http://javarevisited.blogspot.sg/2011/02/how-to-setup-remote-debugging-in.html)  
  
Its also worth noting that when you use the  java -jar command line option to run your Java program as an [executable JAR](http://javarevisited.blogspot.sg/2012/03/how-to-create-and-execute-jar-file-in.html), then the CLASSPATH environment variable will be ignored, and also the -cp and -classpath switches will be ignored. In this case you can set your Java classpath in the META-INF/MANIFEST.MF file by using the Class-Path attribute. In short Class-path attribute in manifest file overrides classpath specified by -cp, -classpath or CLASSPATH environment variable.  
  
Now a common question if I have my CLASSPATH variable pointing to *current directory****"."*** and I have class called "Test" inside package "testing" and with below directory structure C:\project\testing\Test.class in my computer.  
  
**What will happen if I run command "java Test" from directory "C:\project\testing\"? will it run?**  
No it will not run it will give you :  
[Exception in thread "main" java.lang.NoClassDefFoundError: Test](http://javarevisited.blogspot.sg/2011/06/noclassdeffounderror-exception-in.html)  
Since name of the class is not Test, instead it’s testing.Test even though your classpath is set to current directory.  
  
Now what will happen if I give command  java testing.Test from C:\project\testing\ it will again not run and give error?  
  
Exception in thread "main" java.lang.NoClassDefFoundError: testing/Test  
  
Why because now it looking for class called Test which is in package testing, starting from current directory "." but don't find it since there is no directory called "testing after this path "C:\project\testing\".  
  
To run it successfully you need to go back to directory  C:\project and now run   
C:\project>java testing.Test  and It will run successfully because of Classpath issues i prefer to use Eclipse rather than [running Java program from command prompt](http://javarevisited.blogspot.sg/2011/11/run-java-program-from-command-prompt.html).

**Errors related to Classpath in Java**

If you are working in Java you must have faced some errors and exception related to classpath in java, two most common issues related to java classpath is [**ClassNotFoundException**](http://javarevisited.blogspot.com/2011/08/classnotfoundexception-in-java-example.html)and [**NoClassDefFoundError**](http://javarevisited.blogspot.com/2011/06/noclassdeffounderror-exception-in.html). I have seen that many Java developer tries to solve this error by trial and error; they just don’t look beyond the hood and try to understand what the reason for these java classpath related errors is. They often misunderstood that these two errors are same also.   
  
Here is the reason of these Java classpath errors:   
  
ClassNotFoundException is an Exception and will be thrown when Java program dynamically tries to [load a Java class at Runtime](http://javarevisited.blogspot.sg/2012/07/when-class-loading-initialization-java-example.html) and don’t find corresponding [class file](http://javarevisited.blogspot.sg/2012/05/10-points-about-class-file-in-java.html) on classpath. Two keyword here “dynamically” and “runtime”. 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. 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.  
  
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. Classic example of this error is using log4j.jar for logging purpose and forgot to include log4j.jar on classpath in java during run-time. to read more about [**logging in Java**](http://javarevisited.blogspot.com/2011/05/top-10-tips-on-logging-in-java.html) see . Keyword here is,  class was present at compile time but not available on run-time.  This is normally occurring due to any method invocation on a particular class which is part of library and not available on classpath in Java. This is also asked as common interview questions as   
“[What is difference between NoClassDefFoundError and ClassNotFoundException Exception in Java](http://javarevisited.blogspot.com/2011/07/classnotfoundexception-vs.html)”   or  
“When do you see NoClassDefFoundError and ClassNotFoundException Exception in Java”. By the way NoClassDefFoundError can also comes due to various other reason like static initializer failure or class not visible to Classloaders in J2EE environment. read [3 ways to resolve NoClassDefFoundError in Java](http://javarevisited.blogspot.sg/2011/06/noclassdeffounderror-exception-in.html) for complete details.

**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 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 same name exist in Java Classpath then **the class which comes earlier in Classpath** will be picked by [Java Virtual Machine](http://javarevisited.blogspot.sg/2011/12/jre-jvm-jdk-jit-in-java-programming.html).

4.      **By default CLASSPATH in Java points to current directory denoted by "."** and it will look for any class only in 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 directory with **colon “:”** separated , On Windows Java Classpath will be  **semi colon “;”** 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 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 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.

I hope you find this **Java Classpath tutorial**useful , please let me know if you have any doubt or any question related to **"How to set classpath for java**" and I would be happy to answer :) keep learning. Your suggestions and comments are always welcome.  If you like to read UNIX command tips you may find  [10 tips of using find command in Linux](http://javarevisited.blogspot.com/2011/03/10-find-command-in-unix-examples-basic.html),  [10 tips to increase speed on Unix command](http://javarevisited.blogspot.com/2011/03/unix-command-tutorial-working-fast-in.html)and  [10 basic networking Commands in Unix](http://javarevisited.blogspot.com/2010/10/basic-networking-commands-in-linuxunix.html) useful. That's all on What is Classpath in Java, How Classpath works in Java, How to set Classpath in Java on Windows and Linux and how to deal with Classpath issues in Java.

Read more: <http://javarevisited.blogspot.com/2011/01/how-classpath-work-in-java.html#ixzz3K0015lYD>

# [When a class is loaded and initialized in JVM - Java](http://javarevisited.blogspot.sg/2012/07/when-class-loading-initialization-java-example.html)

**Classloading and initialization in Java**

Understanding of when a class is loaded and initialized in JVM is one of the fundamental concept of Java programming language. Thanks to Java language specification we have everything clearly documented and explained, but many Java programmer still doesn't know **when a class is loaded** or when a class is **initialized** in Java. Class loading and initialization seems confusing and complex to many beginners and its true until having some experience in belt its not always easy to get into subtle details of How [JVM](http://javarevisited.blogspot.sg/2011/11/hotspot-jvm-options-java-examples.html) works in Java. In this Java tutorial we will see when class loading occurs in Java and when and how [class](http://javarevisited.blogspot.sg/2011/10/class-in-java-programming-general.html) and [interface](http://javarevisited.blogspot.sg/2012/04/10-points-on-interface-in-java-with.html) are initialized in Java. I will not go into detail of ClasLoader or [How ClassLoader works in  Java](http://javarevisited.blogspot.com/2012/12/how-classloader-works-in-java.html), that is subject of another post I am planning. just to keep this article focused and concise. There are several articles on Java fundamentals in Javarevisited like [How HashMap works in Java](http://javarevisited.blogspot.com/2011/02/how-hashmap-works-in-java.html) and [How Garbage collection works in Java](http://javarevisited.blogspot.sg/2011/04/garbage-collection-in-java.html). If you are interested you can also check those.

**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.

That's all on *When a class is initialized and loaded in Java*. We have seen clear guidelines form JLS regarding class initialization. We have also seen the order on which super type and sub type are initialized and order of initialization for both static and non static fields and blocks in Java.

Read more: <http://javarevisited.blogspot.com/2012/07/when-class-loading-initialization-java-example.html#ixzz3K00Dc3aQ>

# [How to resolve java.lang.ClassNotFoundException in Java](http://javarevisited.blogspot.sg/2011/08/classnotfoundexception-in-java-example.html)

**What is ClassNotFoundException in Java**  
**ClassNotFoundException** is one of Java nightmare every Java developer face in there day to day life. [java.lang.NoClassDefFoundError](http://javarevisited.blogspot.sg/2011/06/noclassdeffounderror-exception-in.html)and java.lang.ClassNotFoundException are two errors  which occurs by and now and chew up of your precious time while finding and fixing root cause. From the name java.lang.ClassNotFoundException looks quite simple but underlying cause of it is always different and which classifies it as an environmental issue. In this java tutorial we will see*what is ClassNotFoundException in java*, what is real cause of it and how to fix it along with some more frequent and infamous examples of java.lang.ClassNotFoundException in Java or J2EE, Don’t mistake this exception with [**NoClassDefFoundError in Java**](http://javarevisited.blogspot.com/2011/06/noclassdeffounderror-exception-in.html) which is also due to incorrect [classpath in Java](http://java67.blogspot.sg/2012/08/what-is-path-and-classpath-in-java-difference.html).  Though both of them are related to missing class file when Java tries to load class in Java they are completely different to each other.  Correct understanding of  [When class is loaded in Java](http://javarevisited.blogspot.sg/2012/07/when-class-loading-initialization-java-example.html) and [How Classpath works](http://javarevisited.blogspot.sg/2011/01/how-classpath-work-in-java.html)  is must to troubleshoot and fix this error quickly.

## What is java.lang.classNotFoundException in Java

As the name suggests classNotFoundException in Java is a subclass of java.lang.Exception and Comes when [Java Virtual Machine](http://javarevisited.blogspot.sg/2011/12/jre-jvm-jdk-jit-in-java-programming.html) tries to load a particular class and doesn't found the requested class in classpath. Another important point about this Exception is that, It is a [checked Exception](http://javarevisited.blogspot.sg/2011/12/checked-vs-unchecked-exception-in-java.html) and you need to provide explicitly Exception handling while using methods which can possibly throw classnotfoundexception in java either by using try-catch block or by using [throws clause](http://javarevisited.blogspot.sg/2012/02/difference-between-throw-and-throws-in.html). Though underlying concept of this exception is simple but it always manifest itself in such format that you need to spend some time to figure out what exactly wrong with your classpath. If you want to know nasty [secrets of java classpath](http://javarevisited.blogspot.com/2011/01/how-classpath-work-in-java.html)  which can cause issue see the link.

When ClassNotFoundException occurs in Java:  
  
As per [java doc java.lang.classNotFoundException](http://download.oracle.com/javase/1.4.2/docs/api/java/lang/ClassNotFoundException.html) comes in following cases:

1) When we try to[load a class](http://javarevisited.blogspot.sg/2012/07/when-class-loading-initialization-java-example.html) by using Class.forName() method and [.class file](http://javarevisited.blogspot.sg/2012/05/10-points-about-class-file-in-java.html) or binary of class is not available in classpath.  
2) When Classloader try to load a class by using findSystemClass () method.  
3) While using loadClass() method of class ClassLoader in Java.

[What is ClassNotFoundException in Java - Fix Solution](http://2.bp.blogspot.com/-wrzDeQGAe1I/TWu8pLuLr4I/AAAAAAAAADE/V017G-6Q61w/s1600/java_logo_50_50.jpg)These statements are completely true in terms of theory of ClassNotFoundExcepiton in Java but as per my experience the concept is "ClassNotFoundException will come only when JVM tries to load a class at run-time, nothing related to [compile time](http://javarevisited.blogspot.com/2012/03/what-is-static-and-dynamic-binding-in.html) unlike NoClassDefFoundError". Also since till run time JVM doesn't know about this Class it can only be done by above specified method or by employing Reflection to read the name of class from some configuration file like in case of struts its struts-config.xml file and then load the class specified on those configuration file. [Reflection](http://javarevisited.blogspot.sg/2012/04/how-to-invoke-method-by-name-in-java.html)is great power of Java but you need to be aware of java.lang.classNotFoundException while using it or [loading class in Java](http://javarevisited.blogspot.sg/2012/07/when-class-loading-initialization-java-example.html).

Examples of classnotfoundexception in java  
  
Though java.lang.classNotFoundException is very common and it can come for any classes, I usually see it while doing JDBC connectivity like when I was writing [Java program to connect Oracle database](http://javarevisited.blogspot.sg/2012/04/java-program-to-connect-oracle-database.html). I am going to list some of the most common scenario where you will get classnotfoundexception in java.

**java.lang.classnotfoundexception com.mysql.jdbc.driver**

This is classical and most infamous example of  and also my first encounter with java.lang.ClassNotFoundException and comes when you are writing JDBC connectivity code and trying to load the [JDBC driver](http://javarevisited.blogspot.sg/2012/05/different-types-of-jdbc-drivers-in-java.html). In this particular case of ClassNotFoundException looks like mysql driver jar file is missing from Classpath. If you pay attention you will find that we use method Class.forName (“driver”) to load the driver class which resides in a particular jar in case of this its mysql-connector.jar and if that jar is not in classpath or not accessible to JVM it will throw  [java.lang.ClassNotFoundException: com.mysql.jdbc.Driver](http://javarevisited.blogspot.sg/2012/03/jdbc-javalangclassnotfoundexception.html)

Here are few more infamous examples of java.lang.ClassnotFoundException which comes here and there while doing any Java J2EE project.

java.lang.classnotfoundexception org.hibernate.hql.ast.hqltoken

java.lang.classnotfoundexception org.springframework.web.context.contextloaderlistener

java.lang.classnotfoundexception org.eclipse.core.runtime.adaptor.eclipsestarter

java.lang.classnotfoundexception org.apache.catalina.startup.catalina

java.lang.classnotfoundexception javax.mail.messagingexception

java.lang.classnotfoundexception oracle.jdbc.driver.oracledriver  
  
This ClassNotfoundException comes when you are trying to [connect Oracle database from Java program using JDBC](http://javarevisited.blogspot.sg/2012/04/java-program-to-connect-oracle-database.html) but you don't have corresponding Oracle driver e.g.ojdbc6.jar is not in classpath of your Java program

### More Complicated ClassNotFoundException

With the advent of dynamic library e.g. OSGi and ClassLoader in Java, this exception can be more tricky and hard to find. Thanks to Mr. Anonymous who has summarized this beautifully, here it is what he says

“It can become a bit more complicated than that. In truth a class does not have to be just visible by the [JVM](http://javarevisited.blogspot.sg/2011/11/hotspot-jvm-options-java-examples.html)through its classpath, but be visible by the Classloader being used. When you are in a multi-classloader environment (In a EE environment, for example, but not limited to), each classloader may have its own rules to search for the classes, and this behavior might depend on the dynamic hierarchy of the Classloaders.

For example, in a project that uses an EAR packaging with WARs inside it, libraries in the lib folder of the EAR are visible to classes inside a WAR, but any classes packaged in a jar put in the WEB-INF/lib on the WAR cannot be seen by classes in different modules (other WARs, EJB-JARS, etc).

It can get really complicated as its common for different modules depending on different versions of the same libraries as different modules depend on each other. It can be a challenge to manage this. Sometimes the classloader can see multiple versions of the same class; sometimes they can see no version at all. Sometimes different dependency paths end in different versions of the same class. And many of this cases end in a ClassNotFoundException.

And then we have OSGi... “. If a class is not visible to ClassLoader than it can also throw NoClassDefFoundError in Java as explained in [3 ways to resolve NoClassDefFoundError in Java](http://javarevisited.blogspot.sg/2011/06/noclassdeffounderror-exception-in.html).

### How to fix java.lang.ClassNotFoundException in Java As you have seen from above examples its clear problem of [classpath](http://javarevisited.blogspot.sg/2011/01/how-classpath-work-in-java.html), so here is my approach to fix or resolve java.lang.ClassNotFoundException: 1) First find out the jar file on which problematic class file is present for example in case of "com.mysql.jdbc.driver" its mysql-connector-java.jar. If you don't know how to find which [jar file](http://javarevisited.blogspot.sg/2012/03/how-to-create-and-execute-jar-file-in.html) a particular class you can see [eclipse shortcuts](http://javarevisited.blogspot.com/2010/10/eclipse-tutorial-most-useful-eclipse.html) to do that or you can simply do "Ctrl+T" in Eclipse and type the name of class, It will list all the jar in the order they appear in eclipse classpath.

### 2) Check whether your classpath contains that jar, if your classpath doesn't contain the jar then just add that class in your classpath.

### 3) If it’s present in your classpath then there is high chance that your classpath is getting [overridden](http://javarevisited.blogspot.sg/2011/12/method-overloading-vs-method-overriding.html) or application is using classpath specified in jar file or start-up script and to fix that you need to find the exact classpath used by your application. Live example of reproducing and Fixing ClassNotFoundException in java I think if we are able to reproduce and solve certain problem we become more comfortable dealing with that, that’s why here we will reproduce java.lang.ClassNotFoundException and solve it by following the concept we have discussed so far.  1) Create a Class called StockTrading.java public class StockTrading{    public String getDescription(){    return "StockTrading";   } }  2) create a Class called OnlineStockTranding.java and load the class StockTrading.java as Class.forName ("stocktrading"); public class OnlineStockTrading {    public static void main(String args[]) throws ClassNotFoundException{       Class.forName("StockTrading");       System.out.println("StockTrading class successfully loaded");    } } 3) Compile both Java source file which will create two [class files](http://java67.blogspot.sg/2012/08/what-is-class-file-in-java-how-to-create-class.html) and run the program should run fine. javin@trading~/java: javac \*.java javin@trading ~/java: ls –lrt -rw-r--r-- 1 javin None  90 Aug 21 09:27 StockTrading.java -rw-r--r-- 1 javin None 208 Aug 21 09:28 OnlineStockTrading.java -rwxr-xr-x 1 javin None 282 Aug 21 09:28 StockTrading.class -rwxr-xr-x 1 javin None 638 Aug 21 09:28 OnlineStockTrading.class javin@trading ~/java:$ java OnlineStockTrading StockTrading class successfully loaded 4) Now just remove the .class file for stocktrading.java and [run the Java program](http://javarevisited.blogspot.sg/2011/11/run-java-program-from-command-prompt.html) and it will [throw](http://javarevisited.blogspot.sg/2012/02/difference-between-throw-and-throws-in.html) java.lang.ClassNotFoundException in java. javin@trading ~/java: rm StockTrading.class javin@trading ~/java: java OnlineStockTrading Exception in thread "main" java.lang.ClassNotFoundException: StockTrading at java.net.URLClassLoader$1.run(Unknown Source) at java.security.AccessController.doPrivileged(Native Method) at java.net.URLClassLoader.findClass(Unknown Source) at java.lang.ClassLoader.loadClass(Unknown Source) ClassFoundException vs NoClassDefFoundError vs UnSupportedClassVersionError There are lots of exceptions in java but these three are the one who most haunted the java developer most mainly because these three are mostly related to environment issues and they all depends upon JVM and Classpath behaviour. Though they look similar there is slight [difference between ClassFoundException and NoClassDefFoundError](http://javarevisited.blogspot.com/2011/07/classnotfoundexception-vs.html) and UnSupportedClassVersionError and we will highlight those differences here for easy understanding and differentiating these three: 1) ClassNotFoundException comes on Runtime when requested class is not available in classpath and mainly due to call to Class.forName () or Classloader.loadClass () or ClassLoader.findSystemClass (). 2) NoClassDefFoundError comes when problematic class was present when your compiled your application but they are not in classpath while you running your program. 3) [UnSupportedClassVersionError](http://javarevisited.blogspot.com/2011/07/javalangunsupportedclassversionerror.html)is easy to differentiate because it’s related to version of classpath and usually comes when you compile your code in higher Java version and try to run on lower java version. Can be resolved simply by using one java version for [compiling and running your application](http://javarevisited.blogspot.sg/2011/11/run-java-program-from-command-prompt.html). So that's all on ClassNotFoundException in java for now , please let me know if you have any tip or  any personal experience on solving java.lang.ClassNotFoundException in Java which you would like to share.

Read more: <http://javarevisited.blogspot.com/2011/08/classnotfoundexception-in-java-example.html#ixzz3K015wU7i>

# [How to resolve java.lang.ClassNotFoundException in Java](http://javarevisited.blogspot.sg/2011/08/classnotfoundexception-in-java-example.html)

**What is ClassNotFoundException in Java**  
**ClassNotFoundException** is one of Java nightmare every Java developer face in there day to day life. [java.lang.NoClassDefFoundError](http://javarevisited.blogspot.sg/2011/06/noclassdeffounderror-exception-in.html)and java.lang.ClassNotFoundException are two errors  which occurs by and now and chew up of your precious time while finding and fixing root cause. From the name java.lang.ClassNotFoundException looks quite simple but underlying cause of it is always different and which classifies it as an environmental issue. In this java tutorial we will see*what is ClassNotFoundException in java*, what is real cause of it and how to fix it along with some more frequent and infamous examples of java.lang.ClassNotFoundException in Java or J2EE, Don’t mistake this exception with [**NoClassDefFoundError in Java**](http://javarevisited.blogspot.com/2011/06/noclassdeffounderror-exception-in.html) which is also due to incorrect [classpath in Java](http://java67.blogspot.sg/2012/08/what-is-path-and-classpath-in-java-difference.html).  Though both of them are related to missing class file when Java tries to load class in Java they are completely different to each other.  Correct understanding of  [When class is loaded in Java](http://javarevisited.blogspot.sg/2012/07/when-class-loading-initialization-java-example.html) and [How Classpath works](http://javarevisited.blogspot.sg/2011/01/how-classpath-work-in-java.html)  is must to troubleshoot and fix this error quickly.

## What is java.lang.classNotFoundException in Java

As the name suggests classNotFoundException in Java is a subclass of java.lang.Exception and Comes when [Java Virtual Machine](http://javarevisited.blogspot.sg/2011/12/jre-jvm-jdk-jit-in-java-programming.html) tries to load a particular class and doesn't found the requested class in classpath. Another important point about this Exception is that, It is a [checked Exception](http://javarevisited.blogspot.sg/2011/12/checked-vs-unchecked-exception-in-java.html) and you need to provide explicitly Exception handling while using methods which can possibly throw classnotfoundexception in java either by using try-catch block or by using [throws clause](http://javarevisited.blogspot.sg/2012/02/difference-between-throw-and-throws-in.html). Though underlying concept of this exception is simple but it always manifest itself in such format that you need to spend some time to figure out what exactly wrong with your classpath. If you want to know nasty [secrets of java classpath](http://javarevisited.blogspot.com/2011/01/how-classpath-work-in-java.html)  which can cause issue see the link.

When ClassNotFoundException occurs in Java:  
  
As per [java doc java.lang.classNotFoundException](http://download.oracle.com/javase/1.4.2/docs/api/java/lang/ClassNotFoundException.html) comes in following cases:

1) When we try to[load a class](http://javarevisited.blogspot.sg/2012/07/when-class-loading-initialization-java-example.html) by using Class.forName() method and [.class file](http://javarevisited.blogspot.sg/2012/05/10-points-about-class-file-in-java.html) or binary of class is not available in classpath.  
2) When Classloader try to load a class by using findSystemClass () method.  
3) While using loadClass() method of class ClassLoader in Java.

[What is ClassNotFoundException in Java - Fix Solution](http://2.bp.blogspot.com/-wrzDeQGAe1I/TWu8pLuLr4I/AAAAAAAAADE/V017G-6Q61w/s1600/java_logo_50_50.jpg)These statements are completely true in terms of theory of ClassNotFoundExcepiton in Java but as per my experience the concept is "ClassNotFoundException will come only when JVM tries to load a class at run-time, nothing related to [compile time](http://javarevisited.blogspot.com/2012/03/what-is-static-and-dynamic-binding-in.html) unlike NoClassDefFoundError". Also since till run time JVM doesn't know about this Class it can only be done by above specified method or by employing Reflection to read the name of class from some configuration file like in case of struts its struts-config.xml file and then load the class specified on those configuration file. [Reflection](http://javarevisited.blogspot.sg/2012/04/how-to-invoke-method-by-name-in-java.html)is great power of Java but you need to be aware of java.lang.classNotFoundException while using it or [loading class in Java](http://javarevisited.blogspot.sg/2012/07/when-class-loading-initialization-java-example.html).

Examples of classnotfoundexception in java  
  
Though java.lang.classNotFoundException is very common and it can come for any classes, I usually see it while doing JDBC connectivity like when I was writing [Java program to connect Oracle database](http://javarevisited.blogspot.sg/2012/04/java-program-to-connect-oracle-database.html). I am going to list some of the most common scenario where you will get classnotfoundexception in java.

**java.lang.classnotfoundexception com.mysql.jdbc.driver**

This is classical and most infamous example of  and also my first encounter with java.lang.ClassNotFoundException and comes when you are writing JDBC connectivity code and trying to load the [JDBC driver](http://javarevisited.blogspot.sg/2012/05/different-types-of-jdbc-drivers-in-java.html). In this particular case of ClassNotFoundException looks like mysql driver jar file is missing from Classpath. If you pay attention you will find that we use method Class.forName (“driver”) to load the driver class which resides in a particular jar in case of this its mysql-connector.jar and if that jar is not in classpath or not accessible to JVM it will throw  [java.lang.ClassNotFoundException: com.mysql.jdbc.Driver](http://javarevisited.blogspot.sg/2012/03/jdbc-javalangclassnotfoundexception.html)

Here are few more infamous examples of java.lang.ClassnotFoundException which comes here and there while doing any Java J2EE project.

java.lang.classnotfoundexception org.hibernate.hql.ast.hqltoken

java.lang.classnotfoundexception org.springframework.web.context.contextloaderlistener

java.lang.classnotfoundexception org.eclipse.core.runtime.adaptor.eclipsestarter

java.lang.classnotfoundexception org.apache.catalina.startup.catalina

java.lang.classnotfoundexception javax.mail.messagingexception

java.lang.classnotfoundexception oracle.jdbc.driver.oracledriver  
  
This ClassNotfoundException comes when you are trying to [connect Oracle database from Java program using JDBC](http://javarevisited.blogspot.sg/2012/04/java-program-to-connect-oracle-database.html) but you don't have corresponding Oracle driver e.g.ojdbc6.jar is not in classpath of your Java program

### More Complicated ClassNotFoundException

With the advent of dynamic library e.g. OSGi and ClassLoader in Java, this exception can be more tricky and hard to find. Thanks to Mr. Anonymous who has summarized this beautifully, here it is what he says

“It can become a bit more complicated than that. In truth a class does not have to be just visible by the [JVM](http://javarevisited.blogspot.sg/2011/11/hotspot-jvm-options-java-examples.html)through its classpath, but be visible by the Classloader being used. When you are in a multi-classloader environment (In a EE environment, for example, but not limited to), each classloader may have its own rules to search for the classes, and this behavior might depend on the dynamic hierarchy of the Classloaders.

For example, in a project that uses an EAR packaging with WARs inside it, libraries in the lib folder of the EAR are visible to classes inside a WAR, but any classes packaged in a jar put in the WEB-INF/lib on the WAR cannot be seen by classes in different modules (other WARs, EJB-JARS, etc).

It can get really complicated as its common for different modules depending on different versions of the same libraries as different modules depend on each other. It can be a challenge to manage this. Sometimes the classloader can see multiple versions of the same class; sometimes they can see no version at all. Sometimes different dependency paths end in different versions of the same class. And many of this cases end in a ClassNotFoundException.

And then we have OSGi... “. If a class is not visible to ClassLoader than it can also throw NoClassDefFoundError in Java as explained in [3 ways to resolve NoClassDefFoundError in Java](http://javarevisited.blogspot.sg/2011/06/noclassdeffounderror-exception-in.html).

### How to fix java.lang.ClassNotFoundException in Java As you have seen from above examples its clear problem of [classpath](http://javarevisited.blogspot.sg/2011/01/how-classpath-work-in-java.html), so here is my approach to fix or resolve java.lang.ClassNotFoundException: 1) First find out the jar file on which problematic class file is present for example in case of "com.mysql.jdbc.driver" its mysql-connector-java.jar. If you don't know how to find which [jar file](http://javarevisited.blogspot.sg/2012/03/how-to-create-and-execute-jar-file-in.html) a particular class you can see [eclipse shortcuts](http://javarevisited.blogspot.com/2010/10/eclipse-tutorial-most-useful-eclipse.html) to do that or you can simply do "Ctrl+T" in Eclipse and type the name of class, It will list all the jar in the order they appear in eclipse classpath.

### 2) Check whether your classpath contains that jar, if your classpath doesn't contain the jar then just add that class in your classpath.

### 3) If it’s present in your classpath then there is high chance that your classpath is getting [overridden](http://javarevisited.blogspot.sg/2011/12/method-overloading-vs-method-overriding.html) or application is using classpath specified in jar file or start-up script and to fix that you need to find the exact classpath used by your application. Live example of reproducing and Fixing ClassNotFoundException in java I think if we are able to reproduce and solve certain problem we become more comfortable dealing with that, that’s why here we will reproduce java.lang.ClassNotFoundException and solve it by following the concept we have discussed so far.  1) Create a Class called StockTrading.java public class StockTrading{    public String getDescription(){    return "StockTrading";   } }  2) create a Class called OnlineStockTranding.java and load the class StockTrading.java as Class.forName ("stocktrading"); public class OnlineStockTrading {    public static void main(String args[]) throws ClassNotFoundException{       Class.forName("StockTrading");       System.out.println("StockTrading class successfully loaded");    } } 3) Compile both Java source file which will create two [class files](http://java67.blogspot.sg/2012/08/what-is-class-file-in-java-how-to-create-class.html) and run the program should run fine. javin@trading~/java: javac \*.java javin@trading ~/java: ls –lrt -rw-r--r-- 1 javin None  90 Aug 21 09:27 StockTrading.java -rw-r--r-- 1 javin None 208 Aug 21 09:28 OnlineStockTrading.java -rwxr-xr-x 1 javin None 282 Aug 21 09:28 StockTrading.class -rwxr-xr-x 1 javin None 638 Aug 21 09:28 OnlineStockTrading.class javin@trading ~/java:$ java OnlineStockTrading StockTrading class successfully loaded 4) Now just remove the .class file for stocktrading.java and [run the Java program](http://javarevisited.blogspot.sg/2011/11/run-java-program-from-command-prompt.html) and it will [throw](http://javarevisited.blogspot.sg/2012/02/difference-between-throw-and-throws-in.html) java.lang.ClassNotFoundException in java. javin@trading ~/java: rm StockTrading.class javin@trading ~/java: java OnlineStockTrading Exception in thread "main" java.lang.ClassNotFoundException: StockTrading at java.net.URLClassLoader$1.run(Unknown Source) at java.security.AccessController.doPrivileged(Native Method) at java.net.URLClassLoader.findClass(Unknown Source) at java.lang.ClassLoader.loadClass(Unknown Source) ClassFoundException vs NoClassDefFoundError vs UnSupportedClassVersionError There are lots of exceptions in java but these three are the one who most haunted the java developer most mainly because these three are mostly related to environment issues and they all depends upon JVM and Classpath behaviour. Though they look similar there is slight [difference between ClassFoundException and NoClassDefFoundError](http://javarevisited.blogspot.com/2011/07/classnotfoundexception-vs.html) and UnSupportedClassVersionError and we will highlight those differences here for easy understanding and differentiating these three: 1) ClassNotFoundException comes on Runtime when requested class is not available in classpath and mainly due to call to Class.forName () or Classloader.loadClass () or ClassLoader.findSystemClass (). 2) NoClassDefFoundError comes when problematic class was present when your compiled your application but they are not in classpath while you running your program. 3) [UnSupportedClassVersionError](http://javarevisited.blogspot.com/2011/07/javalangunsupportedclassversionerror.html)is easy to differentiate because it’s related to version of classpath and usually comes when you compile your code in higher Java version and try to run on lower java version. Can be resolved simply by using one java version for [compiling and running your application](http://javarevisited.blogspot.sg/2011/11/run-java-program-from-command-prompt.html). So that's all on ClassNotFoundException in java for now , please let me know if you have any tip or  any personal experience on solving java.lang.ClassNotFoundException in Java which you would like to share.

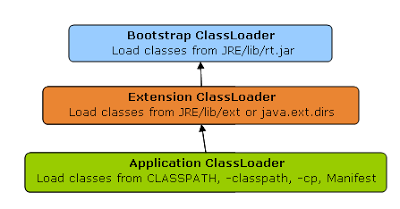
Read more: <http://javarevisited.blogspot.com/2011/08/classnotfoundexception-in-java-example.html#ixzz3K015wU7i>

# [How ClassLoader Works in Java](http://javarevisited.blogspot.com/2012/12/how-classloader-works-in-java.html)

Java class loaders are used to load classes at runtime. ClassLoader in Java works on three principle: delegation, visibility and uniqueness. 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. Visibility principle allows child class loader to see all the classes loaded by parent ClassLoader, but parent class loader can not see classes loaded by child. 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. Correct understanding of class loader is must to resolve issues like [NoClassDefFoundError in Java](http://javarevisited.blogspot.sg/2011/06/noclassdeffounderror-exception-in.html) and [java.lang.ClassNotFoundException](http://javarevisited.blogspot.sg/2011/08/classnotfoundexception-in-java-example.html), which are related to class loading. ClassLoader is also an important topic in advanced Java Interviews, where good knowledge of working of Java ClassLoader and [How classpath works in Java](http://javarevisited.blogspot.ca/2011/01/how-classpath-work-in-java.html) is expected from Java programmer. I have always seen questions like, **Can one class be loaded by two different ClassLoader in Java** on various [Java Interviews](http://javarevisited.blogspot.sg/2011/04/top-20-core-java-interview-questions.html).  In this Java programming tutorial, we will learn what is ClassLoader in Java, How ClassLoader works in Java and some specifics about Java ClassLoader.

**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, **Bootstrap** , **Extension** and **System or Application class loader**. Every class loader has a predefined location, from where they loads class files. Bootstrap ClassLoader is responsible for loading standard JDK class files fromrt.jar and it is parent of all class loaders in Java. Bootstrap class loader don't have any parents, if you call String.class.getClassLoader() it will return null and any code based on that may throw [NullPointerException in Java](http://javarevisited.blogspot.com/2012/06/common-cause-of-javalangnullpointerexce.html). Bootstrap class loader is also known as**Primordial ClassLoader** in Java.  Extension ClassLoader delegates class loading request to its parent, Bootstrap and if unsuccessful, loads class form jre/lib/ext directory or any other directory pointed by java.ext.dirs system property. Extension ClassLoader in JVM is implemented by sun.misc.Launcher$ExtClassLoader. Third default class loader used by JVM to load Java classes is called System or Application class loader and it is responsible for loading application specific classes from [CLASSPATH](http://javarevisited.blogspot.sg/2011/01/how-classpath-work-in-java.html) environment variable, -classpath or -cp command line option, Class-Path attribute of Manifest file inside JAR. Application class loader is a child of Extension ClassLoader and its implemented by sun.misc.Launcher$AppClassLoader class. Also, except Bootstrap class loader, which is implemented in native language mostly in C,  all  Java class loaders are implemented using java.lang.ClassLoader.

[](http://2.bp.blogspot.com/-HCTsr-j_ojw/USTOh1f8JwI/AAAAAAAAAjg/YegPspR5K48/s1600/java_classloader_hierarchy.PNG)

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

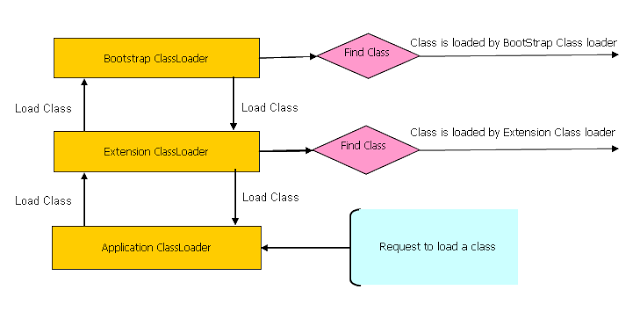
1) Bootstrap ClassLoader - JRE/lib/rt.jar

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 ClassLoader works in Java

[What is ClassLoader in Java, How classloader works in Java](http://3.bp.blogspot.com/-K6q0DQ1v-tw/TWu8owBtc2I/AAAAAAAAADA/oBoHDBiJ8ag/s1600/17.jpg)As I explained earlier Java ClassLoader works in three principles : delegation, visibility and uniqueness. In this section we will see those rules in detail and understand working of Java ClassLoader with example. By the way here is a diagram which explains How ClassLoader load class in Java using delegation.

[](http://1.bp.blogspot.com/-0gOWex7Pb2E/USTOh2K7zpI/AAAAAAAAAjc/_viQADzxrsk/s1600/Java+classloader+working.PNG)

**Delegation principles**

As discussed on [when a class is loaded and initialized in Java](http://javarevisited.blogspot.sg/2012/07/when-class-loading-initialization-java-example.html), a class is loaded in Java, when its needed. Suppose you have an application specific class called Abc.class, first request of loading this class will come to Application ClassLoader which will delegate to its parent Extension ClassLoader which further delegates to Primordial or Bootstrap class loader. Primordial will look for that class in rt.jar and since that class is not there, request comes to Extension class loader which looks on jre/lib/ext directory and tries to locate this class there, if class is found there than Extension class loader will load that class and Application class loader will never load that class but if its not loaded by extension class-loader than Application class loader loads it from [Classpath in Java](http://java67.blogspot.sg/2012/08/what-is-path-and-classpath-in-java-difference.html). Remember Classpath is used to load class files while [PATH](http://javarevisited.blogspot.ca/2011/10/how-to-set-path-for-java-unix-linux-and.html) is used to locate executable like javac or java command.

**Visibility Principle**

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). as shown in below Example

**package** test;  
  
**import** java.util.logging.Level;  
**import** java.util.logging.Logger;  
  
/\*\*  
 \* Java program to demonstrate How ClassLoader works in Java,

 \* in particular about visibility principle of ClassLoader.

 \*  
 \* @author Javin Paul  
 \*/  
  
**public** **class** ClassLoaderTest {  
    
    **public** **static** **void** main(**String** args[]) {  
        **try** {            
            *//printing ClassLoader of this class*  
            **System**.out.println("ClassLoaderTest.getClass().getClassLoader() : "  
                                 + ClassLoaderTest.**class**.getClassLoader());  
  
            
            *//trying to explicitly load this class again using Extension class loader*  
            **Class**.forName("test.ClassLoaderTest", **true**   
                            ,  ClassLoaderTest.**class**.getClassLoader().getParent());  
        } **catch** (**ClassNotFoundException** ex) {  
            **Logger**.getLogger(ClassLoaderTest.**class**.getName()).log(**Level**.SEVERE, **null**, ex);  
        }  
    }  
  
}  
  
**Output:**  
ClassLoaderTest.getClass().getClassLoader() : sun.misc.Launcher$AppClassLoader@601bb1  
16/08/2012 2:43:48 AM test.ClassLoaderTest main  
SEVERE: **null**  
java.lang.**ClassNotFoundException**: test.ClassLoaderTest  
        at java.net.**URLClassLoader**$1.run(**URLClassLoader**.java:202)  
        at java.security.**AccessController**.doPrivileged(Native **Method**)  
        at java.net.**URLClassLoader**.findClass(**URLClassLoader**.java:190)  
        at sun.misc.Launcher$ExtClassLoader.findClass(Launcher.java:229)  
        at java.lang.**ClassLoader**.loadClass(**ClassLoader**.java:306)  
        at java.lang.**ClassLoader**.loadClass(**ClassLoader**.java:247)  
        at java.lang.**Class**.forName0(Native **Method**)  
        at java.lang.**Class**.forName(**Class**.java:247)  
        at test.ClassLoaderTest.main(ClassLoaderTest.java:29)

**Uniqueness Principle**

According to this principle a class loaded by Parent should not be loaded by Child ClassLoader again. Though its completely possible to write class loader which violates Delegation and Uniqueness principles and loads class by itself, its not something which is beneficial. You should follow all  class loader principle while writing your own ClassLoader.

## 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.

**Where to use ClassLoader in Java**

ClassLoader in Java is a powerful concept and used at many places. One of the *popular example of ClassLoader* is AppletClassLoader which is used to load class by Applet, since Applets are mostly loaded from internet rather than local file system, By using separate ClassLoader you can also loads same class from multiple sources and they will be treated as different class in [JVM](http://javarevisited.blogspot.ca/2011/12/jre-jvm-jdk-jit-in-java-programming.html). J2EE uses multiple class loaders to load class from different location like classes from WAR file will be loaded by Web-app ClassLoader while classes bundled in EJB-JAR is loaded by another class loader. Some web server also supports hot deploy functionality which is implemented using ClassLoader. You can also use ClassLoader to load classes from database or any other persistent store.

That's all about **What is ClassLoader in Java** and **How ClassLoader works in Java**. We have seen delegation, visibility and uniqueness principles which is quite important to debug or troubleshoot any ClassLoader related issues in Java. In summary knowledge of How ClassLoader works in Java is must for any Java developer or architect to design Java application and packaging.

Read more: <http://javarevisited.blogspot.com/2012/12/how-classloader-works-in-java.html#ixzz3KQgk4thI>

# [Why String is immutable or final in Java](http://javarevisited.blogspot.com/2010/10/why-string-is-immutable-in-java.html)

This is one of the most popular [String Interview questions in Java](http://javarevisited.blogspot.com/2012/10/10-java-string-interview-question-answers-top.html), which starts with discussion of,  What is String, How String in Java is different than String in C and C++, and then shifted towards [what is immutable object in Java](http://javarevisited.blogspot.com/2013/03/how-to-create-immutable-class-object-java-example-tutorial.html) , what are the benefits of immutable object , why do you use it and which scenarios do you use it. This is some time also asked as *"Why String is final in Java"* . Though there could be many possible answer for this question, and only designer of String class can answer this , I think below two does make sense  
  
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) Another good reason of Why String is immutable in Java suggested by Dan Bergh Johnsson on comments is: 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"  
  
  
Security and String pool being primary reason of making String immutable, I believe there could be some more very convincing reasons as well, Please post those reasons as comments and I will include those on this post. By the way, above reason holds good to answer, another [Java interview questions](http://javarevisited.blogspot.sg/2011/04/top-20-core-java-interview-questions.html) **"Why String is final in Java".** Also to be immutable you have to be final, so that your subclass doesn't break immutability.  what do you guys think ?

Read more: <http://javarevisited.blogspot.com/2010/10/why-string-is-immutable-in-java.html#ixzz3K01Ua5KS>

# [String vs StringBuffer vs StringBuilder in Java](http://javarevisited.blogspot.com/2011/07/string-vs-stringbuffer-vs-stringbuilder.html)

**Difference between String, Stringbuffer and StringBuilder**

String is one of the most important classes in Java and anyone who starts with Java programming uses String to print something on console by using famous System.out.println() statements. Many Java beginners not aware that [String is immutable and final in Java](http://javarevisited.blogspot.sg/2010/10/why-string-is-immutable-in-java.html) and every modification in String result creates a new String object. So How do you manipulate String in Java without creating String garbage? StringBuilder and StringBuffer is answer of this question. StringBuffer is old class but StringBuilder is newly added in Java 5 along with major improvements like [Enum](http://javarevisited.blogspot.sg/2011/08/enum-in-java-example-tutorial.html), [Generics](http://javarevisited.blogspot.sg/2011/09/generics-java-example-tutorial.html), [varargs methods](http://javarevisited.blogspot.sg/2011/09/variable-argument-in-java5-varargs.html) and [Autoboxing in Java](http://javarevisited.blogspot.sg/2012/07/auto-boxing-and-unboxing-in-java-be.html). No matter which kind of application you are working you will find heavy usage of Java String class but if you do profiling of your application you will find that String is the one class which creates lots of garbage because of many temporary String created in program. In this Java tutorial we will see What is String in Java, some important properties of String in Java, What is StringBuffer in Java , When to use StringBuffer in Java , StringBuilder in Java and how it can be used in place of StringBuffer,  **What are differences between String and StringBuffer and StringBuilder in Java**  which is a [frequently asked core Java question](http://javarevisited.blogspot.sg/2011/04/top-20-core-java-interview-questions.html) and mostly **String vs StringBuilder vs StringBuffer**. Now let's start with String.

## Differences between String, StringBuffer and StringBuilder in Java

### String in Java

Before looking *difference between String and StringBuffer or StringBuilder* let’s see some fundamental properties of String Class in Java  
  
[string and stringbuffer, string vs stringbuffer vs stringbuilder](http://javarevisited.blogspot.com/2010/10/what-is-difference-between-enumeration.html)1) String is immutable in Java:  String is by design immutable in Java you can check [this post](http://javarevisited.blogspot.sg/2010/10/why-string-is-immutable-in-java.html) for reason. Immutability offers lot of benefit to the String class e.g. his hashcode value can be cached which makes it a faster hashmap key and one of the reason [why String is a popular key in HashMap](http://javarevisited.blogspot.com/2011/02/how-hashmap-works-in-java.html). Because String is final it can be safely shared between multiple threads  without any extra [synchronization](http://javarevisited.blogspot.sg/2011/04/synchronization-in-java-synchronized.html).   
  
2)when we represent string in double quotes like "abcd" they are referred as String literal and String literals are created in String pools. When you [compare two String literals](http://javarevisited.blogspot.sg/2012/03/how-to-compare-two-string-in-java.html) using equality operator "==" it returns true because they are actually same instance of String. Anyway comparing object with equality operator is bad practice in Java and you should always use [equals method](http://javarevisited.blogspot.sg/2011/02/how-to-write-equals-method-in-java.html) to check equality.  
  
3) "+" operator is [overloaded](http://javarevisited.blogspot.sg/2011/12/method-overloading-vs-method-overriding.html)for String and used to concatenated two string. Internally "+" operation is implemented using either StringBuffer or StringBuilder.  
  
4) Strings are backed up by character Array and represented in UTF-16 format. By the way this behavior can cause memory leak in String because same character array is shared between source String and SubString which can prevent source String from being garbage collected. See [How SubString works in Java](http://javarevisited.blogspot.sg/2011/10/how-substring-in-java-works.html) for more details.  
  
5) String class overrides equals() and hashcode() method and two Strings are considered to be equal if they contain exactly same character in same order and in same case. If you want ignore case comparison of two strings consider using equalsIgnoreCase() method. See  [how to correctly override equals method in Java](http://javarevisited.blogspot.com/2011/02/how-to-write-equals-method-in-java.html)  to learn more about best practices on equals method. Another worth noting point is that equals method must be consistent with compareTo() method for String because SortedSet and SortedMap e.g. TreeMap uses [compareTo method](http://javarevisited.blogspot.sg/2011/11/how-to-override-compareto-method-in.html) to compare String in Java.  
  
7) toString() method provides String representation of any object and its declared in Object class and its recommended for other class to implement this and provide String representation.  
  
8) String is represented using UTF-16 format in Java.  
  
9) In Java you can create [String from char array](http://javarevisited.blogspot.sg/2012/02/how-to-convert-char-to-string-in-java.html), byte array, another string, from StringBuffer or from StringBuilder. Java String class provides constructor for all of these.

### Problem with String in Java

One of its biggest strength Immutability is also biggest problem of Java String if not used correctly. many a times we create a String and then perform a lot of operation on them e.g. converting string into uppercase, lowercase , getting [substring](http://javarevisited.blogspot.sg/2011/10/how-substring-in-java-works.html)out of it , concatenating with other string etc. Since String is an immutable class every time a new String is created and older one is discarded which creates lots of temporary garbage in heap. If String are created using String literal they remain in String pool. To resolve this problem Java provides us two Classes **StringBuffer and StringBuilder**. String Buffer is an older class but StringBuilder is relatively new and added in JDK 5.

### Differences between String and StringBuffer in Java

Main *difference between String and StringBuffer* is String is immutable while StringBuffer is mutable means you can modify a StringBuffer object once you created it without creating any new object. This mutable property makes StringBuffer an ideal choice for dealing with Strings in Java. You can convert a StringBuffer into String by its toString() method. String vs StringBuffer or what is difference between StringBuffer and String is one of the [popular Java interview questions](http://javarevisited.blogspot.sg/2011/04/top-20-core-java-interview-questions.html) for either phone interview or first round. Now days they also include StringBuilder and ask String vs StringBuffer vs StringBuilder. So be preparing for that. In the next section we will see difference between StringBuffer and StringBuilder in Java.

### Difference between StringBuilder and StringBuffer in Java

**StringBuffer** is very good with mutable String but it has one disadvantage all its public methods are [synchronized](http://javarevisited.blogspot.sg/2011/04/synchronization-in-java-synchronized.html)which makes it [thread-safe](http://javarevisited.blogspot.sg/2012/01/how-to-write-thread-safe-code-in-java.html) but same time slow. In JDK 5 they provided similar class called StringBuilder in Java which is a copy of StringBuffer but without synchronization. Try to use ***StringBuilder*** whenever possible it performs better in most of cases than StringBuffer class. You can also use "+" for concatenating two string because "+" operation is internal implemented using either **StringBuffer or StringBuilder in Java**. If you see StringBuilder vs StringBuffer you will find that they are exactly similar and all API methods applicable to StringBuffer are also applicable to StringBuilder in Java. On the other hand String vs StringBuffer is completely different and there API is also completely different, same is true for StringBuilder vs String.

### Summary

In summary here are list of difference between StringBuffer, String and StringBuilder in Java :

1) String is immutable while **StringBuffer and StringBuilder is mutable** object.

2) **StringBuffer is**[**synchronized**](http://www.blogger.com/goog_1642539054)while **StringBuilder is not** which makes StringBuilder faster than StringBuffer.

3) Concatenation operator "+" is internal implemented using either StringBuffer or StringBuilder.

4) Use String if you require [immutability](http://javarevisited.blogspot.sg/2010/10/why-string-is-immutable-in-java.html), use Stringbuffer in java if you need mutable + [thread-safety](http://javarevisited.blogspot.sg/2012/01/how-to-write-thread-safe-code-in-java.html) and use StringBuilder in Java if you require mutable + without thread-safety.

That's all on famous String vs StringBuffer or StringBuffer vs StringBuilder discussion. All these differences helps to avoid common coding mistake of using String in place of StringBuffer in many places. from Java 5 onwards either use + operator of StringBuilder for concatenating String in Java.

Read more: <http://javarevisited.blogspot.com/2011/07/string-vs-stringbuffer-vs-stringbuilder.html#ixzz3KQm40I5C>

# [How to create Immutable Class and Object in Java - Tutorial Example](http://javarevisited.blogspot.com/2013/03/how-to-create-immutable-class-object-java-example-tutorial.html)

Writing or creating immutable classes in Java is becoming popular day by day, because of concurrency and multithreading advantage provided by immutable objects. Immutable objects offers several benefits over conventional mutable object, especially while creating concurrent Java application. Immutable object not only guarantees safe publication of object’s state, but also can be shared among other threads without any external [synchronization](http://javarevisited.blogspot.com/2011/04/synchronization-in-java-synchronized.html). In fact JDK itself contains several immutable classes like [String](http://javarevisited.blogspot.com/2012/03/how-to-compare-two-string-in-java.html), [Integer](http://javarevisited.blogspot.com/2011/08/convert-string-to-integer-to-string.html) and other wrapper classes. For those, who doesn’t know what is immutable class or object, Immutable objects are those, whose state can not be changed once created e.g. java.lang.String, once created can not be modified e.g. trim, uppercase, lowercase. All modification in String result in new object, see [why String is immutable in Java](http://javarevisited.blogspot.com/2010/10/why-string-is-immutable-in-java.html) for more details. In this Java programming tutorial, we will learn, how to write immutable class in Java or how to make a class immutable. By the way making a class immutable is not difficult on code level, but its the decision to make, which class mutable or immutable which makes difference. I also suggest reading, Java Concurrency in Practice to learn more about concurrency benefit offered by Immutable object.

## What is immutable class in Java

[What is Immutable class and object, how to create Immutable in Java with example](http://2.bp.blogspot.com/-wrzDeQGAe1I/TWu8pLuLr4I/AAAAAAAAADE/V017G-6Q61w/s1600/java_logo_50_50.jpg)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).

## How to write immutable class in Java

Despite of few disadvantages, Immutable object still offers several benefits in multi-threaded programming and it’s a great choice to achieve [thread safety](http://javarevisited.blogspot.com/2012/12/how-to-create-thread-safe-singleton-in-java-example.html) in Java code. here are few rules, which helps to make a class immutable in Java :

1. State of immutable object can not be modified after construction, any modification should result in new immutable object.

2. All fields of Immutable class should be final.

3. Object must be properly constructed i.e. object reference must not leak during construction process.

4. Object should be final in order to restrict sub-class for altering immutability of parent class.

By the way, you can still create immutable object by violating few rules, like String has its [hashcode](http://javarevisited.blogspot.com/2011/10/override-hashcode-in-java-example.html) in non final field, but its always guaranteed to be same. No matter how many times you calculate it, because it’s calculated from final fields, which is guaranteed to be same. This required a deep knowledge of Java memory model, and can create subtle [race conditions](http://javarevisited.blogspot.com/2012/02/what-is-race-condition-in.html) if not addressed properly. In next section we will see simple example of writing immutable class in Java. By the way, if your Immutable class has lots of optional and mandatory fields, then you can also use[Builder design pattern](http://javarevisited.blogspot.com/2012/06/builder-design-pattern-in-java-example.html) to make a class Immutable in Java.

## Immutable Class Example in Java

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

public final class Contacts {

    private final String name;

    private final String mobile;

    public Contacts(String name, String mobile) {

        this.name = name;

        this.mobile = mobile;

    }

    public String getName(){

        return name;

    }

    public String getMobile(){

        return mobile;

    }

}

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

public final class ImmutableReminder{

    private final Date remindingDate;

    public ImmutableReminder (Date remindingDate) {

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

            throw new IllegalArgumentException("Can not set reminder” +

                        “ for past time: " + remindingDate);

        }

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

    }

    public Date getRemindingDate() {

        return (Date) remindingDate.clone();

    }

}

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

## Benefits of Immutable Classes in Java

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

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

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

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

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

That's all on **how to write immutable class in Java**. we have seen rules of writing immutable classes, benefits offered by immutable objects and how we can create immutable class in Java which involves mutable fields. Don’t forget to read more about concurrency benefit offered by Immutable object in one of the best Java book recommended to Java programmers, Concurrency Practice in Java.

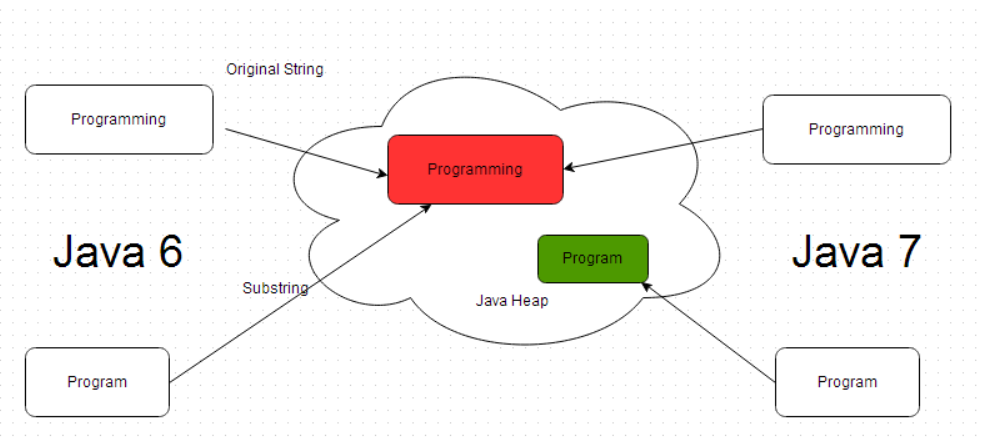
Read more: <http://javarevisited.blogspot.com/2013/03/how-to-create-immutable-class-object-java-example-tutorial.html#ixzz3KQmIjJyK>

# [How SubString method works in Java - Memory Leak Fixed in JDK 1.7](http://javarevisited.blogspot.sg/2011/10/how-substring-in-java-works.html)

**Substring method**from String classis one of most used method in Java, and it's also part of an interesting [String interview question](http://javarevisited.blogspot.com/2012/10/10-java-string-interview-question-answers-top.html)  e.g. How substring works in Java or sometime asked as how does substring creates memory leak in Java. In order to answer these questions, you knowledge of  implementation details is required. Recently one of my friend was drilled on **substring method in Java** during a Java interview, he was using substring() method from long time, and of course all of us has used this, but what surprises him was interviewer's obsession on Java substring, and deep dive till the implementation level. Though String is a special class in Java, and subject of many interview questions e.g. [Why char array is better than String for storing password](http://javarevisited.blogspot.com/2012/03/why-character-array-is-better-than.html) . In this case it was, substring method, which took center stage. Most of us rather just use *substring*(..), and than forgot. Not every Java programmer go into code, and see how exactly it's working. To get a feel of how his interview was let's start .

**Update:**  This issue was actually a bug http://bugs.sun.com/view\_bug.do?bug\_id=6294060,  which is fixed in substring implementation of Java 7. Now, Instead of sharing original character array, substring method creates a copy of it. In short, substring method only retains as much data, as it needed. Thanks to Yves Gillet for pointing this. As some of my readers pointed out, [java.lang.String class](http://javarevisited.blogspot.com/2013/07/java-string-tutorial-and-examples-beginners-programming.html) has also grown into some change in Java 1.7 version and offset and count variable which is used to track positions are removed from String. This may save some bytes with each String instance, but not sharing original array makes substring perform linearly, as compared to constant time previously. Anyway, it's worth to remove any string related memory leak in Java. Having said that, if you have not yet upgraded your Server to Java 7 and still working on Java 1.6 updates, this is one thing, which is worth knowing.

Question starts with normal chit chat, and Interviewer ask,  "**Have you used substring method in Java**", and my friend proudly said Yes, lot many times, which brings a smile on interviewer's face. He says well, that’s good. Next question was **Can you explain what does substring do?** My friend got an opportunity to show off his talent, and how much he knows about Java API;  He said substring method is used to get parts of String in Java. It’s defined injava.lang.String class, and it's an [overloaded method](http://javarevisited.blogspot.com/2011/12/method-overloading-vs-method-overriding.html). One version of substring method takes just beginIndex, and returns part of String started from beginIndex till end, while other takes two parameters, beginIndex and endIndex, and returns part  of String starting from beginIndex to endIndex-1. He also stressed that every time you call  substring() method in Java,  it will return a new String because [String is immutable in Java](http://javarevisited.blogspot.com/2010/10/why-string-is-immutable-in-java.html).

[](http://2.bp.blogspot.com/-bpVcO5iKthk/VAhfFQLeciI/AAAAAAAAB1Y/hzrzYa2ZmlI/s1600/How+SubString+works+in+Java.png)

Next question was, **what will happen if beginIndex is equal to length in substring(int beginIndex)**, no it won't throw IndexOutOfBoundException instead it will return [empty String](http://javarevisited.blogspot.com/2013/02/5-ways-to-check-if-string-is-empty-in-java-examples.html). Same is the case when beginIndex and endIndex is equal, in case of second method. It will only throw StringIndexBoundException when beginIndex is negative, larger than endIndex or larger than length of String.

So far so good, my friend was happy and interview seems going good, until Interviewee asked him**,  Do you know how substring works in Java**? Most of Java developers fail here, because they don't know how exactly substring method works, until they have not seen the code of java.lang.String. If you look substring method inside String class, you will figure out that it calls String (int offset, int count, char value []) [constructor](http://javarevisited.blogspot.com/2012/12/what-is-constructor-in-java-example-chainning-overloading.html) to create new String object. What is interesting here is, value[], which is the same character array used to represent original string. So **what's wrong with this**?

In case If you have still not figured it out, If the original string is very long, and has array of size 1GB, no matter how small a substring is, it will hold 1GB array.  This will also stop original string to be [garbage collected](http://javarevisited.blogspot.com/2012/10/10-garbage-collection-interview-question-answer.html), in case if doesn't have any live reference. This is clear case of memory leak in Java, where memory is retained even if it's not required. That's how*substring method* creates **memory leak**.

## How SubString in Java works

Obviously next question from interviewer would be,  **how do you deal with this problem?** Though you can not go, and change Java substring method, you can still make some work around, in case you are creating substring of significant longer String. Simple solution is to trim the string, and keep size of character array according to length of substring. Luckilyjava.lang.String has constructor to do this, as shown in below example.

*// comma separated stock symbols from NYSE*

String listOfStockSymbolsOnNYSE = getStockSymbolsForNYSE();

*//calling String(string) constructor*

String apple = **new** String(

listOfStockSymbolsOnNYSE.substring(appleStartIndex, appleEndIndex)

);

If you look code on java.lang.String class, you will see that this [constructor](http://javarevisited.blogspot.sg/2012/01/what-is-constructor-overloading-in-java.html) trim the array, if it’s bigger than String itself.

**public** String(String original) {

...

**if** (originalValue.length > size) {

*// The array representing the String is bigger than the new*

*// String itself. Perhaps this constructor is being called*

*// in order to trim the baggage, so make a copy of the array.*

**int** off = original.offset;

v = Arrays.copyOfRange(originalValue, off, off+size);

} **else** {

*// The array representing the String is the same*

*// size as the String, so no point in making a copy.*

v = originalValue;

}

...

}

Another way to solve this problem is to call intern() method on substring, which will than fetch an [existing string](http://javarevisited.blogspot.sg/2012/03/how-to-compare-two-string-in-java.html) from pool or add it if necessary. Since the String in the pool is a real string it only take space as much it requires. It’s also worth noting that sub-strings are not internalized, when you call intern() method on original String. Most developer successfully answers first three questions, which is related to usage of substring, but they get stuck on last two, How substring creates memory leak or How substring works. It's not completely there fault, because what you know is that every time substring() returns new String which is not exactly true, since it’s backed by same [character array](http://javarevisited.blogspot.com/2012/02/how-to-convert-char-to-string-in-java.html).

This was the only interview question, which bothers my friend little otherwise, its standard service level company  Java interview in India. By the way, he got the call a day after ,even though he struggled little bit on How SubString method works in Java, and that was the reason he shared this interview experience with me.

Read more: <http://javarevisited.blogspot.com/2011/10/how-substring-in-java-works.html#ixzz3K01b3yy3>

# [10 Java String interview Question answers - Advanced](http://javarevisited.blogspot.com/2012/10/10-java-string-interview-question-answers-top.html)

**10 Java String interview Question answers**

String interview questions in Java is one of Integral part of any [Core Java](http://javarevisited.blogspot.sg/2011/04/top-20-core-java-interview-questions.html) or [J2EE interviews](http://javarevisited.blogspot.sg/2011/09/servlet-interview-questions-answers.html). No one can deny importance of String and How much it in any Java application irrespective of whether its core Java desktop application, web application, Enterprise application or Mobile application. [String](http://javarevisited.blogspot.sg/2011/07/string-vs-stringbuffer-vs-stringbuilder.html) is one of the fundamental of Java programming language and correct understanding of String class is must for any Java programmer. What makes *String interview questions* in Java even more interesting is the special status of String in terms of features and privileges it has, like + operator is kind of overloaded to perform String concatenation despite the fact that [Java does not support operator overloading](http://javarevisited.blogspot.sg/2011/08/why-java-does-not-support-operator.html). There is a separate **String pool** to store String literal etc. In this article we will some frequently asked question on String in a Java interview which focuses on range of issues like immutability, [thread-safety](http://javarevisited.blogspot.sg/2012/01/how-to-write-thread-safe-code-in-java.html), Security etc.

## Java String interview Question

[Java String Interview Questions Answers Advanced Experienced](http://3.bp.blogspot.com/-K6q0DQ1v-tw/TWu8owBtc2I/AAAAAAAAADA/oBoHDBiJ8ag/s1600/17.jpg)Here are my list *of frequently asked question on String*, feel free to add any other interesting question which you faced on String during any [Core Java interview](http://java67.blogspot.sg/2012/09/top-10-tricky-java-interview-questions-answers.html) :

**1) What is String in Java ? Is String is data type?**

String in Java is not a primitive data type like int, long or double.  String is a [class](http://javarevisited.blogspot.sg/2011/10/class-in-java-programming-general.html) or in more simple term a user defined type. This is confusing for some one who comes from C background. String is defined in java.lang package and wrappers its content in a character array. String provides [equals() method](http://javarevisited.blogspot.sg/2011/02/how-to-write-equals-method-in-java.html) to compare two String and provides various other method to operate on String like toUpperCase() to convert String into upper case, replace() to [replace String contents](http://javarevisited.blogspot.sg/2011/12/java-string-replace-example-tutorial.html), substring() to get substring, split() to [split long String](http://javarevisited.blogspot.sg/2011/09/string-split-example-in-java-tutorial.html) into multiple String.

**2) Why String is final in Java**

String is final by design in Java, some of the points which makes sense why String is final is Security, optimization and to maintain pool of String in Java. for details on each of this point see [Why String is final in Java](http://javarevisited.blogspot.sg/2010/10/why-string-is-immutable-in-java.html).

**3) What is Difference between String and StringBuffer in Java**

This is probably the most common question on String I have seen in Java interviews. Though String and Stringbuffer are two different class they are used in context of concatenating two Strings, Since String is immutable in Java every operation which changes String produces new String, which can be avoided by using Stringbuffer. See [String vs StringBuffer](http://javarevisited.blogspot.sg/2011/07/string-vs-stringbuffer-vs-stringbuilder.html) for more details.

**4) What is difference in String on C and Java**

If you have mentioned C in your resume, then you are likely to face this *String interview question*. Well C String and Java String are completely different to each other, C String is a **null terminated**[character array](http://javarevisited.blogspot.sg/2012/02/how-to-convert-char-to-string-in-java.html) while String in Java is an Object. Also String is more feature rich in Java than C.

**5) Why char array is better than String for storing password?**

This String interview question is debatable and you might not agree with interviewer but this is also a chance to show that how deep and differently you can think of. One of the reason which people give Why you should store password in char array over String is related to immutability, since its not possible to remove erase contents of String but you can erase contents of char array. See [Why char array preferred over String for password](http://javarevisited.blogspot.sg/2012/03/why-character-array-is-better-than.html) for complete discussion.

**6) How do you compare two String in Java ?**

This is another *common String interview question* which appears on fresher level interviews. There are multiple ways to compare two String like equals() method, equalsIgnoreCase() etc, You can also see [4 ways to compare String in Java](http://javarevisited.blogspot.sg/2012/03/how-to-compare-two-string-in-java.html) for more examples. Main thing which interviewer checks is that whether candidate mentioned equality operator or not "==", comparing String with equality operator is common mistake which works in some case and doesn't work in other. next String interview question is follow-up up of this.

**7) Can we compare String using == operator? What is risk?**

As discussed in previous String question, You can compare String using equality operator but that is not suggested or advised because equality operator is used to compare primitives and [equals() method](http://javarevisited.blogspot.sg/2011/02/how-to-write-equals-method-in-java.html) should be used to compare objects. As we have seen in [pitfall of autoboxing in Java](http://javarevisited.blogspot.sg/2012/07/auto-boxing-and-unboxing-in-java-be.html) that how equality operator can cause subtle issue while comparing primitive to Object, any way String is free from that issue because it doesn't have corresponding primitive type and not participate in autoboxing. Almost all the time comparing String means comparing contents of String i.e. characters and equals() method is used to perform character based comparison. equals() return true if two String points to same object or two String has same contents while == operator returns true if two String object **points to same object** but return false if two different String object contains same contents. That explains why sometime it works and sometime it doesn't. In short [always use equals method in Java](http://javarevisited.blogspot.sg/2011/02/how-to-write-equals-method-in-java.html) to check equality of two String object.

**8) How substring method work in Java**

This is one of the [tricky Java question](http://java67.blogspot.sg/2012/09/top-10-tricky-java-interview-questions-answers.html) relate to String and until you are familiar with internals of String class, its difficult to answer. Substring shares same character array as original String which can create memory leak if original String is quite big and not required to retain in memory but unintentionally retained by substring which is very small in size and prevents large array from begin claimed during [Garbage collection in Java](http://javarevisited.blogspot.sg/2011/04/garbage-collection-in-java.html). See [How Substring works in Java](http://javarevisited.blogspot.sg/2011/10/how-substring-in-java-works.html) for more details.

**10)What is String pool in Java**

Another [tough Java question](http://java67.blogspot.sg/2012/09/top-10-tough-core-java-interview-questions-answers.html) asked in  String interview. String pool is a special storage area in Java heap, mostly located on PerGen space, to store String literals like "abc". When Java program creates a new String using String literal, [JVM](http://javarevisited.blogspot.sg/2011/12/jre-jvm-jdk-jit-in-java-programming.html) checks for that String in pool and if String literal is already present in pool than same object is returned instead of creating a whole new object. String pool check is only performed when you create String as literal, if you create String using new() operator, a new String object will be created even if String with same content is available in pool.

**9) What does intern() method do in Java**

As discussed in previous String interview question, String object crated by new() operator is by default not added in String pool as opposed to String literal. intern() method allows to put an [String object](http://javarevisited.blogspot.sg/2012/08/convert-collection-to-string-in-java.html) into pool.

**11) Does String is thread-safe in Java**

If you are familiar with the concept of immutability and [thread-safety](http://javarevisited.blogspot.sg/2011/07/java-multi-threading-interview.html) you can easily answer this String interview question in Java. Since [String is immutable](http://javarevisited.blogspot.sg/2010/10/why-string-is-immutable-in-java.html), it is thread-safe and it can be shared between multiple thread without external synchronization.

That's all on *Java String interview question*. In Summary there are lot of specifics about String which needs to be know for any one who has started Java programming and these String question will not just help to perform better on Java Interviews but also opens new door of learning about String. I didn't know many String related concepts until I come across these question which motivated to research and learn more about String in Java.

Read more: <http://javarevisited.blogspot.com/2012/10/10-java-string-interview-question-answers-top.html#ixzz3K01k1dxm>

[**Difference between instance class and local variables in Java**](http://javarevisited.blogspot.com/2012/02/difference-between-instance-class-and.html)

There are lot of **differences between instance variable, class variable and local variable in Java** and knowing them will help you to write correct and bug free Java programs. Java is full featured programming language and provides different kind of variables like **static variable** also called**Class variable** since it belongs to whole Class, **non static** also called **instance variable** and **local variables** which varies in scope and value. Thank god Java doesn't have any register variable or auto scope like C otherwise it would have so much detail to remember. static variables are common source of error in may multi-threaded [java program](http://javarevisited.blogspot.com/2011/11/run-java-program-from-command-prompt.html) and does require a bit of carefulness while using it. On the other hand instance variable and local variable has less sharing visibility than static variable. 

**Instance Variable vs Class Variable in Java**

[Difference between local, instance and class variable in Java](hhttp://javarevisited.blogspot.com/2011/10/how-to-set-path-for-java-unix-linux-and.html)

let's first see *difference between instance variable and class variable* also known as **non static vs static variable in java**. Instance variable are per instance (object) basis. If you have 5 instance of one class you will have five copies of instance variable. these are also referred as *non static variable* and initialized when you create instance of any object using new() operator or by using other methods like reflection e.g. Class.newInstance(). On the other hand Class variables are declared using static keyword and they have exact same value for every instance. static or class variable are initialized when class is first loaded into JVM memory unlike instance variable which initialized when instance is created. *Static variables are similar to global variable* in C and can be used to store data which is static in nature and has same value for all instance, but at same static variable also cause subtle concurrency bugs if updated by multiple threads. you can read more about static keywords in my post [secrets of static keyword in Java](http://javarevisited.blogspot.com/2011/11/static-keyword-method-variable-java.html).

**Instance variable vs local variable in Java**

Now let's see **difference between instance variable and local variable**. local variables are local in scope and they are not visible or accessible outside there scope which is determined by {} while instance variables are visible on all part of code based on there access modifier e.g. **public** , **private** or **protected**. only public can be accessed from outside while protected and private can be accessed from subclass and class itself. Access modifier can not be applied to local variable and you can not even make them static. only modifier which is applicable to local variable is final and only [final local variables](http://javarevisited.blogspot.com/2011/12/final-variable-method-class-java.html) are visible inside anonymous class. value of instance variable is limited to instance, one instance can not access value of other instance in Java.

Best practices related to local, instance and static variable in Java

Its good to know some of the best practices related to declaration and use of Variables in Java while learning differences among different type of variable in Java:

1. Always **name your variable as per Java Bean naming convention**.
2. By default **give private access to your member variables** (both static and instance) and provide more access step by step e.g. from private to protected to package to public. This way you will be following encapsulation principle.
3. Always **declare local variable where you use**instead of declaring it on top of method or block.
4. **Don't hide instance or static variable** by giving same name to local variable. this may result in subtle programming bugs.
5. Be **consistent** with your variable naming convention **don't mix different convention from different language** e.g. some programming language use first word to denote type of variable e.g. bExit to denote boolean Exit variable or iNumber to denote integer Number variable. Though they are good but mixing simple names as per Java Bean naming convention with this will only lead confusion.

Read more: <http://javarevisited.blogspot.com/2012/02/difference-between-instance-class-and.html#ixzz3K01uVXH7>

# [What is final in Java? Final variable , Method and Class Example](http://javarevisited.blogspot.com/2011/12/final-variable-method-class-java.html)

**Final in java** is very important keyword and can be applied to class, method, and variables in Java. In this java final tutorial we will see **what is final keyword in Java**, *what does it mean by making final variable*, final method and final class in java and *what are primary benefits of using final keywords in Java* and finally some examples of final in Java. Final is often used along with [static keyword in Java](http://javarevisited.blogspot.com/2011/11/static-keyword-method-variable-java.html) to make static final constant and you will see how final in Java can increase performance of Java application.

## Example of Final variable, Final method and Class in Java

### What is final keyword in Java?

[Final keyword, final variable, final method and class in java example](http://javarevisited.blogspot.com/2011/10/how-substring-in-java-works.html)**Final is a keyword** or reserved word in java and can be applied to member variables, methods, class and local variables in Java. Once you make a reference final you are not allowed to change that reference and compiler will verify this and raise **compilation error** if you try to re-initialized **final variables in java**.

**What is final variable in Java?**

Any variable either member variable or local variable (declared inside method or block) modified by final keyword is called final variable. Final variables are often declare with static keyword in java and treated as constant. Here is an example of final variable in Java

public static final String LOAN = "loan";

LOAN = new String("loan") **//invalid compilation error**

Final variables are by default read-only.

**What is final method in Java**

Final keyword in java can also be applied to methods. A java method with final keyword is called final method and it can not be overridden in sub-class. You should make a method final in java if you think it’s complete and its behavior should remain constant in sub-classes. Final methods are faster than non-final methods because they are not required to be resolved during run-time and they are bonded on compile time. Here is an *example of final method in Java***:**

class **PersonalLoan**{

 public final String getName(){

     return "personal loan";

 }

}

class **CheapPersonalLoan** extends **PersonalLoan**{

    @Override

    public final String getName(){

        return "cheap personal loan"; **//compilation error: overridden method is final**

    }

}

### What is final Class in Java

Java class with final modifier is called final [class in Java](http://javarevisited.blogspot.com/2011/10/class-in-java-programming-general.html). Final class is complete in nature and can not be sub-classed or inherited. Several classes in Java are final e.g. String, Integer and other wrapper classes. Here is an *example of final class in java*

final class **PersonalLoan**{

}

class **CheapPersonalLoan** extends **PersonalLoan**{  //compilation error: cannot inherit from final class

}

### Benefits of final keyword in Java

Here are few benefits or advantage of using final keyword in Java:

1. Final keyword improves performance. Not just JVM can cache **final variable** but also application can cache frequently use final variables.

2. Final variables are safe to share in [multi-threading](http://javarevisited.blogspot.com/2011/02/how-to-implement-thread-in-java.html) environment without additional synchronization overhead.

3. **Final keyword**allows [JVM](http://javarevisited.blogspot.com/2011/12/jre-jvm-jdk-jit-in-java-programming.html) to optimized method, variable or class.

### Final and Immutable Class in Java

Final keyword helps to write immutable class. Immutable classes are the one which can not be modified once it gets created and String is primary example of immutable and final class which I have discussed in detail on [Why String is final or immutable in Java](http://javarevisited.blogspot.com/2010/10/why-string-is-immutable-in-java.html). Immutable classes offer several benefits one of them is that they are effectively read-only and can be safely shared in between multiple threads without any synchronization overhead. You can not make a class immutable without making it final and hence final keyword is required to make a class immutable in java.

### Example of Final in Java

Java has several system classes in JDK which are final, some example of final classes are String, Integer, Double and other wrapper classes. You can also use final keyword to make your code better whenever it required. See relevant section of **java final tutorial** for *example of final variable***,** *final method* and *final class in Java*.

### Important points on final in Java

1. **Final keyword** can be applied to member variable, local variable, method or [class in Java](http://javarevisited.blogspot.com/2011/10/class-in-java-programming-general.html).

2. **Final member variable** must be initialized at the time of declaration or inside constructor, failure to do so will result in compilation error.

3. You can not reassign value to *final variable in Java*.

4. **Local final variable** must be initializing during declaration.

5. Only final variable is accessible inside anonymous class in Java.

6. **Final method** can not be [overridden in Java](http://javarevisited.blogspot.com/2011/12/method-overloading-vs-method-overriding.html).

7. **Final class** can not be inheritable in Java.

8. **Final** is different than **finally** keyword which is used on [Exception handling in Java](http://javarevisited.blogspot.com/2011/12/checked-vs-unchecked-exception-in-java.html).

9. Final should not be confused with finalize() method which is declared in object class and called before an object is garbage collected by JVM.

10. All variable declared inside java interface are implicitly final.

11. **Final and abstract** are two opposite keyword and a final class can not be [abstract in java](http://javarevisited.blogspot.com/2010/10/abstraction-in-java.html).

12. Final methods are bonded during compile time also called static binding.

13. *Final variables* which is not initialized during declaration are called blank final variable and must be initialized on all constructor either explicitly or by calling this(). Failure to do so compiler will complain as "*final variable (name) might not be initialized*".

14. Making a class, method or variable final in Java helps to improve performance because JVM gets an opportunity to make assumption and optimization.

15. As per Java code convention **final variables are treated as constant** and written in all Caps e.g.

private final int COUNT=10;

16. Making a collection reference variable final means only reference can not be changed but you can add, remove or change object inside collection. For example:

private final List Loans = new ArrayList();

list.add(“home loan”);  **//valid**

list.add("personal loan"); **//valid**

loans = new Vector();  **//not valid**

That’s all on **final in Java**. We have seen what final variable, final method is and final class in Java and what does those mean. In Summary whenever possible start using final in java it would result in better and faster code.

Read more: <http://javarevisited.blogspot.com/2011/12/final-variable-method-class-java.html#ixzz3K020QQUK>