<http://javarevisited.blogspot.sg/2012/12/what-is-constructor-in-java-example-chainning-overloading.html>

What is Constructor in Java with Example – Constructor Chaining and Overloading

**What is constructor in Java**

Constructor in Java is block of code which is executed at the time of Object creation. But other than getting called, Constructor is entirely different than methods and has some specific properties like name of constructor must be same as name of Class. Constructor also can not have any return type, constructor’s are automatically chained by using [this keyword](http://javarevisited.blogspot.sg/2012/01/this-keyword-java-example-tutorial.html) and super. Since Constructor is used to create object, object initialization code is normally hosted in Constructor. Similar to method you can also [overload constructor in Java](http://javarevisited.blogspot.sg/2012/01/what-is-constructor-overloading-in-java.html). In this Java tutorial we will some important points about constructor in Java which is worth remembering for any Java programmer. It’s also worth remember that any [static initializer block](http://javarevisited.blogspot.sg/2011/11/static-keyword-method-variable-java.html) is executed before constructor because they are executed [when class is loaded into memory](http://javarevisited.blogspot.sg/2012/07/when-class-loading-initialization-java-example.html) while constructors are executed when you create instance of any object e.g. using new() keyword.

## Constructor in Java – things to remember

[What is Constructor in Java with example ](http://3.bp.blogspot.com/-K6q0DQ1v-tw/TWu8owBtc2I/AAAAAAAAADA/oBoHDBiJ8ag/s1600/17.jpg)Here is some important properties of constructor in Java, these are very specific to constructor only and does not applicable to any other function or method.

1) First and most important rule of declaring constructor is that name of constructor in Java must be exactly same with the class on which you declare constructor, if it doesn't then compiler will flag as error. A [class in Java](http://javarevisited.blogspot.sg/2011/10/class-in-java-programming-general.html) can have as many constructor as it and that is called [constructor overloading in Java](http://javarevisited.blogspot.sg/2012/01/what-is-constructor-overloading-in-java.html) but signature of two constructor must not be same. here is an example of having multiple constructors in Java and how they are called using new() operator:

**public** **class** ConstructorDemo{  
   **public** ConstructorDemo(){  
      **System**.out.println("Inside no argument constructor");  
   }  
        
   **public** ConstructorDemo(**String** name){  
      **System**.out.println("Inside one argument constructor in Java with name: " + name);  
   }  
  
   **public** **static** **void** main(**String** args[]) **throws** **IOException** {  
        
     ConstructorDemo d = **new** ConstructorDemo(); *//calling no argument constructor in java*  
     ConstructorDemo e = **new** ConstructorDemo("Testing"); *//calling one argument constructor in java*  
    
   }  
}  
  
**Output:**  
Inside no argument constructor  
Inside one argument constructor in Java with name: Testing

In above example we have create two separate [object](http://javarevisited.blogspot.sg/2012/12/what-is-object-in-java-or-oops-example.html) by calling two different constructors of class ConstructorDemo. If you notice carefully name of constructor is same as name of class. Also signature of two constructor is different to each other.

2) Another important rule of declaring constructor is that constructor in Java doesn't have return type. As I said constructor is different than methods in Java and doesn't return anything, Java Constructor are by default of type void. Though you can have return statement inside constructor without returning any value but can return control back to caller. See [difference between method and constructor in Java](http://java67.blogspot.sg/2012/11/difference-between-method-vs-constructor-java.html) for more differences.

3) Here comes another interesting property of constructor which is tested in SCJP and various other Java Exams and [Java Interviews](http://javarevisited.blogspot.ca/2011/04/top-20-core-java-interview-questions.html). Every Class in Java has constructor, if no explicit constructor is specified by Programmer, Java Compiler inserts a **no argument constructor** inside class. This is also called **default Constructor in Java**. if you provide any constructor in Java e.g. with one argument or two argument than compiler will not add default constructor or no arguments constructor, which makes your class unusable with framework or library which uses [reflection](http://javarevisited.blogspot.sg/2012/04/how-to-invoke-method-by-name-in-java.html) and follow Java Bean naming convention. So always provide no argument constructor in Java. Another drawback of not providing no argument constructor is chances of having restricted hierarchy. Suppose another sub class is created and you don't add constructor over there than compiler tries to create a default constructor which calls super() at first line. super() means call to no argument constructor of super class and since there is no such constructor in your class it will fail with compilation error. This is like making your class [final in Java](http://javarevisited.blogspot.sg/2011/12/final-variable-method-class-java.html).

4) One more important property of constructor in Java is **constructor chaining**. Calling one constructor from another constructor in Java is called Constructor chaining. you can use [keyword this](http://javarevisited.blogspot.sg/2012/01/this-keyword-java-example-tutorial.html) for calling constructor of same class and keyword super for calling constructor of super class. Anyway **call to constructor must be on the first line of any constructor** or else you will get compilation error. Read more about constructor chaining and constructor overloading [here](http://javarevisited.blogspot.sg/2012/01/what-is-constructor-overloading-in-java.html).

5) You can use any access modifier with Java constructor. they can be [public, protected or private](http://www.blogger.com/javarevisited.blogspot.sg/2012/10/difference-between-private-protected-public-package-access-java.html). Default or no argument

constructor has same access modifier as class. You can also prevent a class from extension by making there constructor private. With private constructor instance of that class can only be created inside declaring class. [Singleton pattern in Java](http://javarevisited.blogspot.ca/2011/03/10-interview-questions-on-singleton.html) is popular example of Class with private constructor.

6) Constructor in Java can not be abstract, [static](http://javarevisited.blogspot.sg/2012/10/what-is-static-import-in-java-5-example-tutorial.html), [final](http://javarevisited.blogspot.sg/2011/12/final-variable-method-class-java.html) or [synchronized](http://javarevisited.blogspot.sg/2011/04/synchronization-in-java-synchronized.html). These modifiers are not allowed for constructor.

7) Since parent class is initialized before child class in Java, Constructor of parent class is executed before constructor of child class, that explains why super() is first statement in default no argument constructor. To understand more about how class is loaded into memory read [How ClassLoader works in Java](http://javarevisited.blogspot.com.au/2012/12/how-classloader-works-in-java.html) and [When class is loaded and initialized in JVM](http://javarevisited.blogspot.sg/2012/07/when-class-loading-initialization-java-example.html).

8) Constructor can throw Exception in Java in fact constructor can declare Exception in there [throws clause](http://javarevisited.blogspot.sg/2012/02/difference-between-throw-and-throws-in.html) but that makes caller to handle or re throw Exception while creating any instance of Class.

9) Creating object using new() keyword and constructor has there pros and cons. Its not good in terms of [Encapsulation](http://javarevisited.blogspot.sg/2012/03/what-is-encapsulation-in-java-and-oops.html) because if you directly create any instance of class you code is tied up with structure of Constructor and any change in constructor will require changes in all places where its object gets created. Standard way is to use [factory design pattern](http://javarevisited.blogspot.sg/2011/12/factory-design-pattern-java-example.html) in Java which encapsulate object creation logic and provides better maintenance over time.

10) Unlike C++ there is no destructor in Java. Though objects has [finalize method](http://javarevisited.blogspot.sg/2012/03/finalize-method-in-java-tutorial.html) which suppose to run before objects gets garbage collected but that is not guaranteed by Java language specification and it may run or may not.

That’s all on What is constructor in Java and important points about constructor in Java. As you see there is lot of rules and specific information around constructor but its an important aspect of Java programming language and you must have good grasp of all constructor specifics in Java. We have also touched concepts like constructor chaining and constructor overloading which is quite popular on various Java exams.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

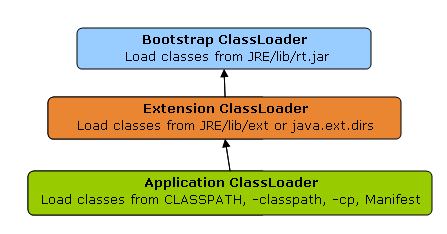
<http://javarevisited.blogspot.com.au/2012/12/how-classloader-works-in-java.html>

How ClassLoader Works in Java

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



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

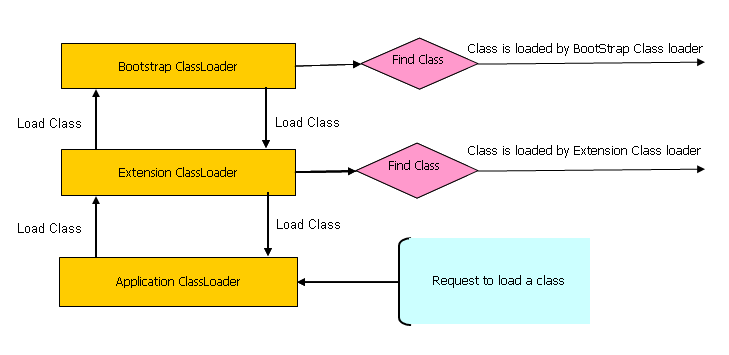
1) Bootstrap ClassLoader - JRE/lib/rt.jar

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.



**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.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

<http://javarevisited.blogspot.sg/2012/07/when-class-loading-initialization-java-example.html>

When a class is loaded and initialized in JVM – Java

**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.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*