Q: How does class loading work when the same class exists in different applications on the same server?

Ans:

A Java classloader typically works by looking for classes in one or more places in a fixed sequence. For instance, the classloader that loads your application when you run it from the command line looks first in the rt.jar file (and others on the bootclasspath), and then in the directories and JAR files specified by your classpath.

A webapp classloading is similar in principle, but a bit more complicated in practice. For a particular webapp, a webapp's classloader looks for classes in the following order. For example Tomcat 6 looks for classes in this order:

1. Bootstrap classes of your JVM
2. System class loader classes (described [here](http://tomcat.apache.org/tomcat-6.0-doc/class-loader-howto.html))
3. /WEB-INF/classes of the webapp
4. /WEB-INF/lib/\*.jar of the webapp
5. $CATALINA\_HOME/lib
6. $CATALINA\_HOME/lib/\*.jar

Of course, once the classloader has found the class it is looking for, it looks no further. So classes with the same name later in the order won't get loaded.

The complication is that the web container has one classloader for each webapp, and these classloaders delegate to other classloaders that manage the common classes. In practice, this means that some classes will only ever be loaded once for the entire container (e.g. 1. and 2.) and others may get loaded multiple times by different classloaders.

(When a class is loaded more than once, it results in distinct Class objects and distinct class statics. The versions of the class are different types as far as the JVM is concerned and you cannot typecast from one version to the other.)

Finally, Tomcat can be configure to allow individual webapps to be "hot loaded". This entails stopping a webapp, creating a new classloader for it, and restarting it.

**FOLLOWUP**

So ... synchronizing a static method will not protect access to a shared resource where the class has been loaded multiple times?

It depends on the details, but it probably won't. (Or to look at if another way, if a class has *actually*been loaded multiple times, then a static method of each "load" of the class will access a different set of static fields.)

If you really want a singleton application class instance to be shared by multiple webapps in the same container, it is simplest if you put the class into $CATALINA\_HOME/lib or the equivalent. But you also should ask yourself if this is good system design. Consider combining the webapps, or to using request forwarding etc instead of a shared data structure. The singleton pattern tends to be troublesome in webapps, and this flavor is even more so.

Q: What is a Java ClassLoader?

Taken from this nice [tutorial](http://www.oracle.com/technetwork/articles/javase/classloaders-140370.html) from Sun:

Motivation

Applications written in statically compiled programming languages, such as C and C++, are compiled into native, machine-specific instructions and saved as an executable file. The process of combining the code into an executable native code is called linking - the merging of separately compiled code with shared library code to create an executable application. This is different in dynamically compiled programming languages such as Java. In Java, the .class files generated by the Java compiler remain as-is until loaded into the Java Virtual Machine (JVM) -- in other words, the linking process is performed by the JVM at runtime. Classes are loaded into the JVM on an 'as needed' basis. And when a loaded class depends on another class, then that class is loaded as well.

When a Java application is launched, the first class to run (or the entry point into the application) is the one with public static void method called main(). This class usually has references to other classes, and all attempts to load the referenced classes are carried out by the class loader.

To get a feeling of this recursive class loading as well as the class loading idea in general, consider the following simple class:

public class HelloApp {

public static void main(String argv[]) {

System.out.println("Aloha! Hello and Bye");

}

}

If you run this class specifying the -verbose:class command-line option, so that it prints what classes are being loaded, you will get an output that looks as follows. Note that this is just a partial output since the list is too long to show here.

prmpt>java -verbose:class HelloApp

[Opened C:\Program Files\Java\jre1.5.0\lib\rt.jar]

[Opened C:\Program Files\Java\jre1.5.0\lib\jsse.jar]

[Opened C:\Program Files\Java\jre1.5.0\lib\jce.jar]

[Opened C:\Program Files\Java\jre1.5.0\lib\charsets.jar]

[Loaded java.lang.Object from shared objects file]

[Loaded java.io.Serializable from shared objects file]

[Loaded java.lang.Comparable from shared objects file]

[Loaded java.lang.CharSequence from shared objects file]

[Loaded java.lang.String from shared objects file]

[Loaded java.lang.reflect.GenericDeclaration from shared objects file]

[Loaded java.lang.reflect.Type from shared objects file]

[Loaded java.lang.reflect.AnnotatedElement from shared objects file]

[Loaded java.lang.Class from shared objects file]

[Loaded java.lang.Cloneable from shared objects file]

[Loaded java.lang.ClassLoader from shared objects file]

[Loaded java.lang.System from shared objects file]

[Loaded java.lang.Throwable from shared objects file]

.

.

.

[Loaded java.security.BasicPermissionCollection from shared objects file]

[Loaded java.security.Principal from shared objects file]

[Loaded java.security.cert.Certificate from shared objects file]

[Loaded HelloApp from file:/C:/classes/]

Aloha! Hello and Bye

[Loaded java.lang.Shutdown from shared objects file]

[Loaded java.lang.Shutdown$Lock from shared objects file]

As you can see, the Java runtime classes required by the application class (HelloApp) are loaded first.

Class Loaders in the Java 2 Platform

The Java programming language keeps evolving to make the life of applications developers easier everyday. This is done by providing APIs that simplify your life by allowing you to concentrate on business logic rather than implementation details of fundamental mechanisms. This is evident by the recent change of J2SE 1.5 to J2SE 5.0 in order to reflect the maturity of the Java platform.

As of JDK 1.2, a bootstrap class loader that is built into the JVM is responsible for loading the classes of the Java runtime. This class loader only loads classes that are found in the boot classpath, and since these are trusted classes, the validation process is not performed as for untrusted classes. In addition to the bootstrap class loader, the JVM has an extension class loader responsible for loading classes from standard extension APIs, and a system class loader that loads classes from a general class path as well as your application classes.

Since there is more than one class loader, they are represented in a tree whose root is the bootstrap class loader. Each class loader has a reference to its parent class loader. When a class loader is asked to load a class, it consults its parent class loader before attempting to load the item itself. The parent in turn consults its parent, and so on. So it is only after all the ancestor class loaders cannot find the class that the current class loader gets involved. In other words, a delegation model is used.

The java.lang.ClassLoader Class

The java.lang.ClassLoader is an abstract class that can be subclassed by applications that need to extend the manner in which the JVM dynamically loads classes. Constructors in java.lang.ClassLoader (and its subclasses) allow you to specify a parent when you instantiate a new class loader. If you don't explicitly specify a parent, the virtual machine's system class loader will be assigned as the default parent. In other words, the ClassLoader class uses a delegation model to search for classes and resources. Therefore, each instance of ClassLoader has an associated parent class loader, so that when requested to find a class or resources, the task is delegated to its parent class loader before attempting to find the class or resource itself. The loadClass() method of the ClassLoader performs the following tasks, in order, when called to load a class:

If a class has already been loaded, it returns it. Otherwise, it delegates the search for the new class to the parent class loader. If the parent class loader doesn't find the class, loadClass() calls the method findClass() to find and load the class. The finalClass() method searches for the class in the current class loader if the class wasn't found by the parent class loader.

There's more in the original article, which also shows you how to implement your own network class loaders, which answers your question as to why (and how). See also the [API docs](http://java.sun.com/javase/6/docs/api/java/lang/ClassLoader.html).

Java Classloader:

The Java Classloader is a part of the [Java Runtime Environment](https://en.wikipedia.org/wiki/Java_Runtime_Environment) that [dynamically loads](https://en.wikipedia.org/wiki/Dynamic_loading) [Java classes](https://en.wikipedia.org/wiki/Java_class) into the [Java Virtual Machine](https://en.wikipedia.org/wiki/Java_Virtual_Machine).

Each Java class must be loaded by a class loader.

When the JVM is started, three class loaders are used:[[3]](https://en.wikipedia.org/wiki/Java_Classloader#cite_note-3)[[4]](https://en.wikipedia.org/wiki/Java_Classloader#cite_note-4)

1. Bootstrap class loader
2. Extensions class loader
3. System class loader

The bootstrap class loader loads the core Java libraries[[5]](https://en.wikipedia.org/wiki/Java_Classloader#cite_note-5) located in the <JAVA\_HOME>/jre/lib directory. This class loader, which is part of the core JVM, is written in native code.

The extensions class loader loads the code in the extensions directories (<JAVA\_HOME>/jre/lib/ext,[[6]](https://en.wikipedia.org/wiki/Java_Classloader#cite_note-6) or any other directory specified by the java.ext.dirs system property). It is implemented by the sun.misc.Launcher$ExtClassLoader class.

The system class loader loads code found on java.class.path, which maps to the [CLASSPATH](https://en.wikipedia.org/wiki/Classpath_(Java)) [environment variable](https://en.wikipedia.org/wiki/Environment_variable). This is implemented by the sun.misc.Launcher$AppClassLoaderclass.

# Java VisualVM - Monitoring Application Threads

Java VisualVM presents data for local and remote applications in a tab specific for that application. Application tabs are displayed in the main window to the right of the Applications window. You can have multiple application tabs open at one time. Each application tab contains sub-tabs that display different types of information about the application.

## Monitoring Thread Activity

Java VisualVM displays real-time, high-level data on thread activity in the Threads tab.  
**Note:** The information displayed in the Threads tab is based on Java Management Extensions (JMX). The Threads tab is visible if Java VisualVM can make a JMX technology-based connection (JMX connection) with the target application and retrieve JMX instrumentation from the Java Virtual Machine (JVM). If the target application is a local application running on Java Development Kit (JDK) version 1.6, the JMX connection is made automatically. If the target application is not running on JDK version 1.6, you may need to explicitly establish a JMX connection with the JVM software. For more on establishing a JMX connection, see the following document:

* [Connecting to JMX Agents Explicitly](https://docs.oracle.com/javase/8/docs/technotes/guides/visualvm/jmx_connections.html)

By default the Threads tab displays a timeline of current thread activity. You can click a thread in the timeline to view details about that thread in the Details tab.

### Timeline Tab

This tab displays a timeline with real-time thread states. Use the buttons in the Timeline toolbar to zoom in/out on the current view and to switch to the Scale to Fit mode. The drop-down list enables you to select which threads are displayed. You can choose to view all threads, live threads or finished threads. You can also select a single thread or multiple threads to display a subset of the threads. You can double-click on a thread timeline to open that thread in the Details tab.

A timeline for each thread provides a quick overview of the thread's activity.

## Taking a Thread Dump

You can use Java VisualVM to take a thread dump (stack trace) while a local application is running. Taking a thread dump does not stop the application. When you print the thread dump you get a printout of the thread stack that includes thread states for the Java threads.

When you print a thread dump in Java VisualVM, the tool prints a stack trace of the active threads of the application. Using Java VisualVM to take a thread dump can be very convenient in cases where you do not have a command-line console for the application. You can use a stack trace to help diagnose a number of issues such as deadlocks or when an application hangs.

Screenshot of thread dump (stack trace) in thread dump sub-tab.

Q: How to identify stuck thread in java? How to resolve issue?

1. By using Visual VM take Thread dump, and kill the threads which are stuck.
2. Use interrupt(), instead of thread.stop().

Q: prevent deadlocks in java?

Some quick tips out of my head

* don't use multiple threads (like Swing does, for example, by mandating that everything is done in the EDT)
* don't hold several locks at once. If you do, always acquire the locks in the same order
* don't execute foreign code while holding a lock
* use interruptible locks