J2EE Project
Survival Guide

Srikanth Shenoy
Viswanath Krishnan
Nithin Mallya
Jagmohan Bhasin

Edited by: Srikanth Shenoy

ObjectSource

Austin

Chapter 21

Classloaders and J2EE

In this chapter:

1. You will understand why classloaders are important in Java and more so in J2EE environments.
2. You will gain an in-depth knowledge of classloaders.
3. You will learn about the J2EE classloader hierarchy and its implications.

Thorough understanding of classloaders is the basis for better partitioning, packaging and deployment of J2EE applications and ultimately for better architecture. So, it is time to get back to the basics of architecture and take it from there.

21.1 Classloader basics

Regular Java applications running from command line involve three classloaders – Bootstrap, Extensions and System-Classpath classloaders, although it will appear as though there is only one classloader to the unsuspecting programmer. The three class loaders have a parent child relationship among themselves. This relationship is similar to the parent and child class relationship in object oriented programming and is shown in Figure 21.1 as a UML diagram.

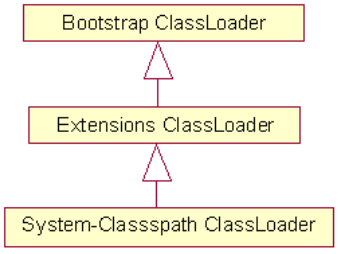

Figure 21.1 Classloader hierarchy for a stand alone Java program.

Bootstrap classloader is the parent of all classloaders and loads the standard JDK classes in lib directory of JRE (*rt.jar* and *i18n.jar*). All the `java.*` classes are loaded by this classloader.

Extensions Classloader is the immediate child of Bootstrap classloader. This classloader loads the classes in *lib\ext* directory of the JRE. For example, JDK1.4.x ships with its own implementation for JCE. The JCE implementation is identified by the *sunjceprovider.jar* and is present in *JRE/lib/ext* directory and is loaded by the extensions classloader.

System-Classpath classloader is the immediate child of Extensions classloader. It loads the classes and jars specified by the CLASSPATH environment variable, `java.class.path` system property, `-cp` or `-classpath` command line settings. If any of the jars specified in one of the above manner have a *MANIFEST.MF* file with a `Class-Path` attribute, the jars specified by the `Class-Path` attribute are also loaded.

Classloader Magic

Things are not so straightforward with J2EE. J2EE does not define a fixed classloader structure, but leaves it to vendors. But generally they follow a hierarchy and you have to understand it clearly to design applications well. The explicit presence of multiple classloaders in J2EE applications poses some unique challenges. For instance, consider the `Person` class in Listing 21.1.

Listing 21.1 Sample class illustrating the influence of Classloaders

```
public class Person  {
  private String firstName;
  private String lastName;

  public boolean equals(Object obj) {
    boolean returnValue = false;
    if (obj.getClass().equals(Person.class)) {
      Person person = (Person) obj;

      if (person.getFirstName().equals(this.firstName) &&
          person.getLastName().equals(this.lastName) )
      {
        returnValue = true;
      }
    }
    return returnValue;
  }
}
```

Let us suppose that you have bundled the `Person` class in both the EJB-JAR and also the WAR. The EJB-JAR contains an EJB – `MyEJB`. `MyEJB` loads this class and initializes the `Person`'s `firstName` and `lastName` to be "John" and "Doe" respectively. The web application also does the same initialization. At some point the Person loaded by web application is passed into the EJB. A comparison occurs when the `equals()` is invoked and Yikes!, you get a `ClassCastException`. And you thought both the `Person` objects are equal and the `equals()` method would return true.

Consider a slightly different situation. Now bundle the `Person` class only in the WAR. The EJB-JAR contains the same EJB – `MyEJB`. It has a method `foo()` on its public interface that has `Person` as an argument. Also assume that you have bundled the home and remote interface for `MyEJB` with the WAR. Now, as soon as the web application invokes `foo()` on `MyEJB`, you get a `NoClassDefFoundError` in the `MyEJB`[1]. Now bundle the `Person` class in the EJB-JAR and repeat the process. Now you get a `NoClassDefFoundError` in the web application[2].

[1] You get this `NoClassDefFoundError` because WAR classloader loads the `Person` and the EJB classloader – its parent, cannot see this class. Section 21.2 explains this in more detail

[2] You get this `NoClassDefFoundError` because even though EJB classloader - which is a parent of WAR classloader, loads the `Person` class, the *MANIFEST.MF* file from the WAR does not have an entry for the EJB-JAR. You can solve this problem by removing the Home and Remote interfaces from the WAR and adding the EJB-JAR as a classpath entry in the *MANIFEST.MF* file. Section 21.2 explains this in more detail.

Puzzling isn't it? Both mechanisms of packaging the `Person` have failed. The key thing to remember is that EJB and the web applications have different classloaders. Two objects loaded by different classloaders are never equal even if they carry the same values. Further more, a class is identified uniquely in the context of the associated classloader. By the above definition, if two different classloaders load the same class within their scope, they are treated as two different classes. Hence we get the `ClassCastException` in the first scenario. This also applies to singletons too. If you have implemented a class as a singleton, you will find that each classloader has its own singleton.

Fair enough. What about the second scenario? This is slightly complicated and requires understanding of classloader hierarchies in J2EE. We will cover J2EE classloader hierarchy in Section 2.3. But to understand that and all that is to come, an in-depth knowledge of classloader basics is essential.

21.2 Three principles of Classloader operation

Classloader problems, when they occur are difficult to debug. Good news: There are only three basic principles to understand. If you clearly understand all of the three, you can resolve any classloader problem. Now that I have your attention, let us move on.

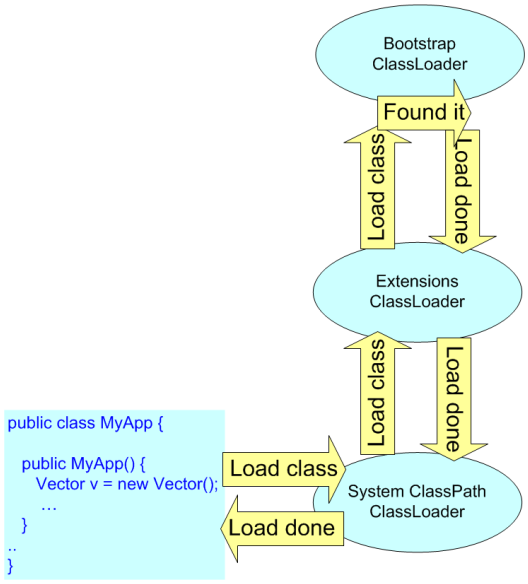

Figure 21.1 Classloader hierarchy illustrating the delegation

The first principle is ***Delegation Principle***. According to this principle, if a particular class is not loaded already, the classloaders delegate the requests to load that class to their parent classloaders. This delegation continues until the top of the hierarchy is reached and the primordial classloader loads the class. Figure 21.1 shows this scenario. The System-ClassPath classloader loads a class called `MyApp`. `MyApp` creates a new instance of `java.util.Vector`. Assume that `java.util.Vector` has not been loaded already. Since System-Classpath classloader loaded the `MyApp` class, it first asks its parent, the extension classloader to load the class. The extension classloader asks the Bootstrap classloader to load `java.util.Vector`. Since `java.util.Vector` is a J2SE class, the bootstrap classloader loads it and returns.

Consider a slightly different scenario in Figure 21.2. In this case, `MyApp` creates a new instance of `MyClass`, another application specific class. Assume that `MyClass` has not been loaded yet. As usual, when the System-Classpath classloader receives the request to load the class, it delegates it to its parent. The request finally reaches the Bootstrap classloader. It cannot find the class. Hence its child, Extensions classloader tries to load it. It cannot find it either. Finally the request

comes back to the System-Classpath classloader. It finds the class and loads it. This explains the alternative path when everything is not a happy day scenario.

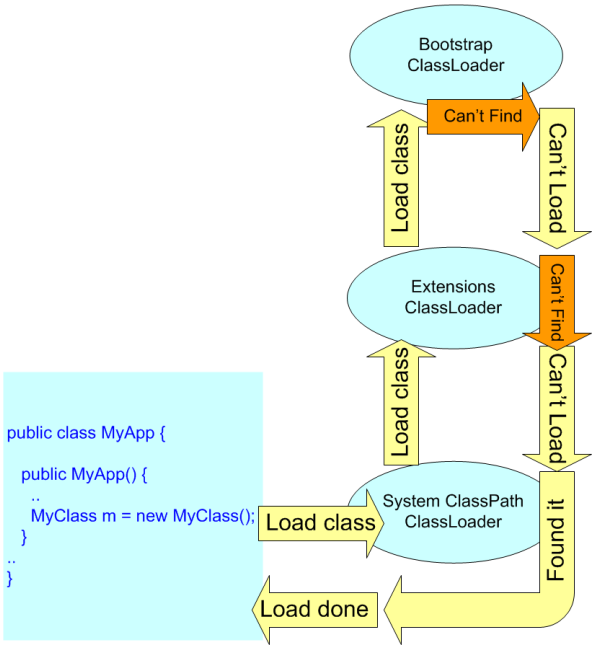

Figure 21.2 Classloader hierarchy illustrating the delegation when classes cannot be found

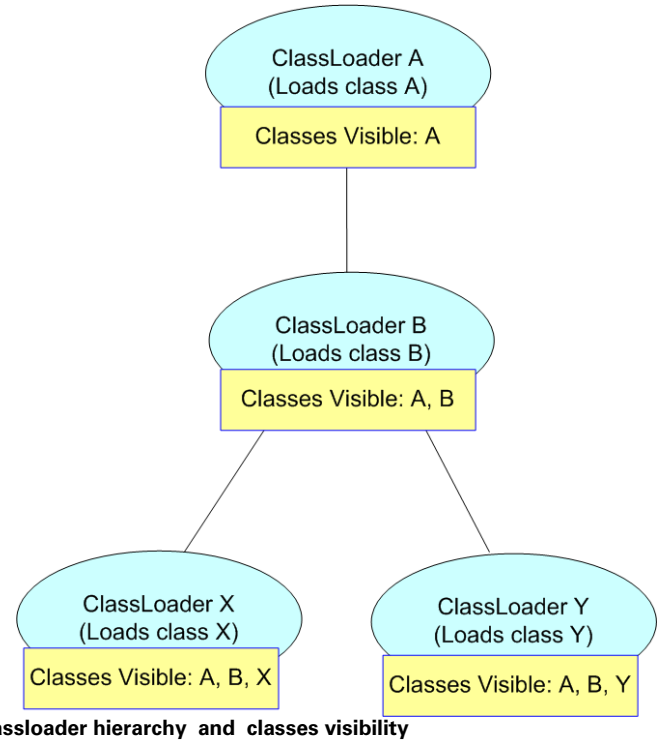

Figure 21.3 Classloader hierarchy and classes visibility

The second principle is the ***Visibility principle***. According to this principle, Classes loaded by parent classloaders are visible to child classloaders but not vice versa. What this means is that a class can only see other classes loaded by the `ClassX`'s classloader or one of its parents. The reverse is not true i.e. a class loaded by `ClassX`'s parent classloader cannot see `ClassX`. An example will make things clearer. Look at Figure 2.3. Four classloaders are shown- ClassLoaders A, B, X and Y. Class A is the topmost Classloader. ClassLoader B is its child. ClassLoaders X and Y are B's siblings. Each of them loads classes with same names i.e. A, B, X and Y. A is the only class visible as far as other classes loaded by ClassLoader A are concerned. As far as classes loaded by ClassLoader B is concerned, A and B are the visible classes. Similarly for classes loaded by ClassLoader X, classes A, B and X are visible, but not class Y. Sibling classloaders cannot see each other's classes.

The third principle is the class ***Uniqueness Principle***. According to this principle, when a classloader loads a class, the child classloaders in the hierarchy will never reload the class. This follows from the delegation principle since a classloader always delegates class loading to its parents. The child classloader will load it (or try to load it) only if the parent hierarchy fails to load the class. Thus the uniqueness of the class is maintained. An interesting scenario emerges when both parent and child classloaders load the same class. You might think how is this feasible after all. Isn't this contradicting the class uniqueness principle?

To answer this question look at Figure 21.3 again. Let us assume that none of the classes have been loaded anywhere in the hierarchy. Let us also suppose that `X`, loaded by ClassLoader X, forcefully uses its classloader to load `B`. This can be done as shown in Listing 21.2 by using an API such as `Class.forName()`. The code shows such a scenario.

Listing 21.2 Using Class.forName()

```
01 public class X {
02
03   public X() {
04      ClassLoader cl = this.getClass().getClassLoader();
05      Class B = Class.forName("B", true, cl);
06   }
07 }
```

In the constructor for `X`, the class `B` is loaded by explicitly using `Person`'s parent classloader, i.e. the parent of the classloader that loaded `Person`. By doing so, the delegation is overridden and `B` is loaded by ClassLoaderX – the classloader of `X`. Now suppose that another class loaded by ClassLoader B tries to access `B`, it cannot find it and hence follows the delegation principle. Since the delegation principle only consults the parents, ClassLoader B also eventually loads Class B. When some other code tries to compare two objects of type `B` - each loaded by

different classloaders, it gets a `ClassCastException`. In fact this is the reason why `ClassCastException` was thrown in the first example in Section 21.1.

> ***Delegation Principle:*** If a class is not loaded already, the classloaders delegate the request to load that class to their parent classloaders.
> ***Visibility Principle:*** Classes loaded by parent classloaders are visible to child classloaders but not vice versa.
> ***Uniqueness Principle:*** When a classloader loads a class, the child classloaders in the hierarchy will never reload that class.

These three principles are key to untangling and debugging any classloader issues you will ever face. Till now we described classloader hierarchy using arbitrary names. In the next section you will see that such classloader hierarchy is very real in J2EE applications. Most of the application code should never explicitly reference classloaders. It is in fact a bad thing. It is mostly the framework code that might want to explicitly use classloaders once in a while. However every developer and architect should understand the classloader hierarchy to write elegantly packaged code.

21.3 J2EE classloader hierarchy and its implications

In J2EE, each application is packaged as an Enterprise ARchive (EAR). The EAR is a self-contained deployment unit having minimal dependencies on external classes (with the exception of application server classes). Each EAR gets its own classloader. Packaging and deploying classes and resources as EAR make the software evolution non-intrusive.

Structure of an EAR

An EAR file is composed of any number of following components.

1. EJB-JAR – Each EJB-JAR can contain one or more EJBs
2. WAR – Each WAR contains exactly one web application.
3. RAR – Resource Adapter Archive implementing Java Connector Architecture (JCA).
4. Dependency JAR – Normal JAR file containing classes that are shared between web and ejb application. It can also be a third party library used by both web and ejb application.
5. *application.xml* – XML file describing the contents of the EAR.

More information on the structure of the EAR, WAR and EJB-JAR can be obtained from the J2EE specification. Figure 2.4 shows a sample EAR. The EAR, named *sample.ear* contains the following artifacts.

1. Two dependency libraries *util-a.jar* and *util-b.jar*.
2. Two EJB-JARs *loanejb.jar* and *personejb.jar*
3. A WAR named *sampleweb.war*
4. application.xml

As we go along in this chapter, we will explain the individual classloaders that load these artifacts.

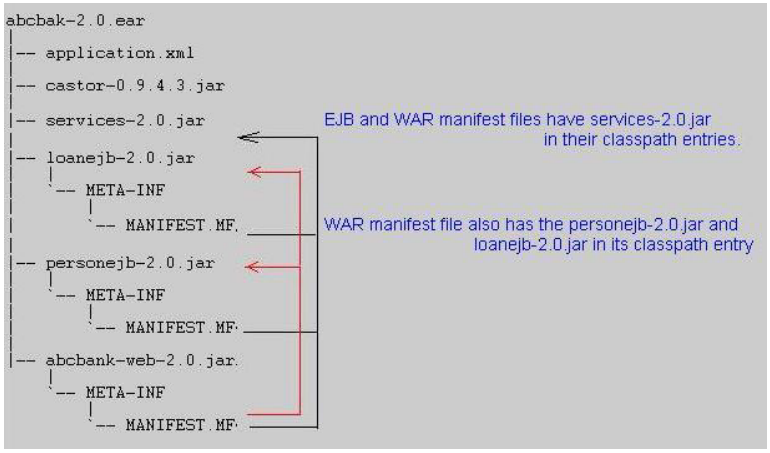

Figure 21.4 Sample EAR structure

The software that you develop typically depends on several third party libraries over which you do not have control. Suppose that a certain third party library such as Log4J is in version 1.0.2 when you deployed your first J2EE application into production. Time flies and you are ready to deploy your second application that uses Log4J version 1.2.8 and is heavily dependent on certain features in Log4J 1.2.8. By using the EAR approach for deployment, you have ensured that the older application can continue to use the older Log4J classes, while newer application can take advantage of the latest features in Log4J. In other words you have created a sandbox for your application in the shared application server environment.

J2EE classloader hierarchy

J2EE specifies that a hierarchy of classloaders is needed to achieve the isolation between applications, but leaves it to the vendors to define the exact structure. However to comply with the J2EE specification, most vendors have classloaders for each of the J2EE application components depending on its location. Further, these classloaders have a hierarchy among themselves, i.e. they have a parent-child relationship. Figure 2.5 shows a sample hierarchy of classloaders. Note that each

application server's classloader hierarchy might slightly differ. Application server vendors sometimes tend to treat two or more of these classloaders as one. For instance, a certain application server might treat Application classloader and EJB classloader to be the same. But the general concepts behind the hierarchy remain the same.

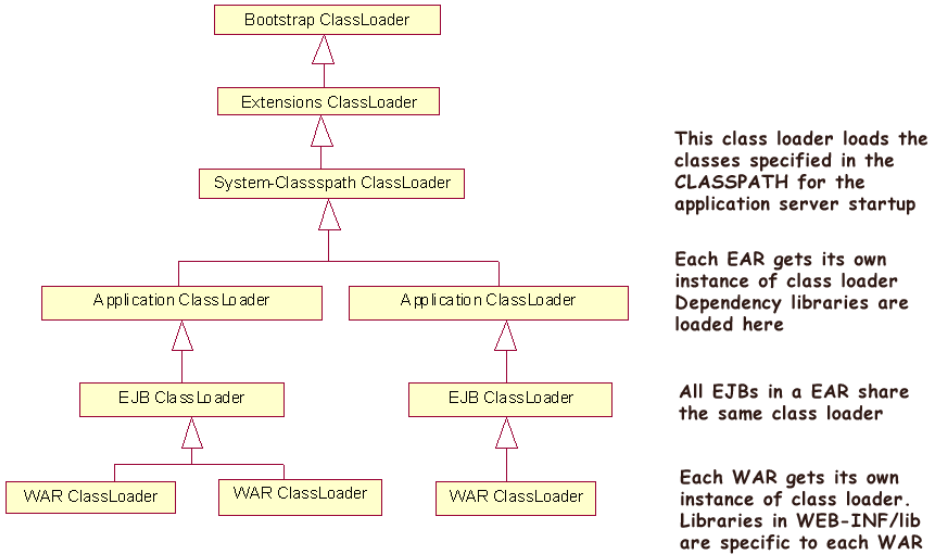

Figure 21.5 Sample Classloader Hierarchy in J2EE Application Servers.

Standard UML notations are used to show the parent child relationship among the classloaders in Figure 21.5. The extensions classloader is a child of the bootstrap classloader. The system classpath classloader extends the JDK extensions classloader. The system classpath classloader loads the classes from the classpath of the JVM. Application-specific classloaders are children of the system classpath classloader. Since each Child classloaders implicitly have visibility to the classes loaded by System-Classpath Classloader or its parents. When the parent classloader is below the System-Classpath Classloader in hierarchy, the child classloader can "see" the class only when it is specified in the manifest file for the child classloader. For instance, if an EJB application wants to reference *util-a.jar* and *util-b.jar*, then we can add the following entry into the EJB-JAR's manifest file, `MANIFEST.MF`.

```
Class-Path: util-a.jar util-b.jar
```

The *util-a.jar* and *util-b.jar* are located within the same EAR containing the EJB-JAR. The EAR classloader loads them, but they become visible to the EJB classloader (a child of EAR classloader), when the EJB-JAR references these jars in its manifest file.

Let us go back to Figure 21.5 again. Bootstrap classloader is the primordial classloader for all applications. This is the classloader that loads standard J2SE classes. The Extensions classloader is used to load the classes under *jre/lib/ext*

directory. These classes are not part of the core Java platform but extend it in some way.

This is followed by the System-Classpath classloader, which loads the classes in the CLASSPATH or as specified by the command line switch `-classpath`. This is the classloader that loads all the application server classes and libraries. A class specified during the application server startup is then available to all applications underneath no matter what. Hence it is very tempting to load the J2EE application class files from this location when a `NoClassDefFoundError` is encountered. In fact that solves the problem (temporarily at least) and is exactly the reason why many developers new to J2EE make the mistake of specifying the classes here. By putting the classes here, they are also visible to every J2EE application running on the application server. Normally this is a very bad thing to do. Unless there is a compelling reason to do so you should avoid this approach. Such compelling reasons include you being a service provider for the application server – for instance, if you are providing the CMP framework implementation, which will be used by the application server, then your classes have to go in here so that your classes are visible to the application server classes (Recall the first principle specified above).

The next in hierarchy is the EAR classloader. Every J2EE EAR gets its own classloader. This is the starting point for isolating application classes from one another. This classloader loads all the dependency libraries in the EAR. Dependency libraries are generally those classes common between the web and EJB application. For instance, an application exception thrown from EJB module has to be caught in the web module. The throws clause in the EJB module and the catch clause in the web module have to refer to the same class loaded by the same classloader at run time. Similarly a Value Object [Core J2EE Patterns – Deepak Alur et al] By applying the first principle of classloaders, both EJB and the web module can see the class loaded by the EAR classloader, since it is their common parent. By applying the second principle, since the exception class is already loaded by the parent classloader and is visible to the child classloader, it will not be reloaded.

Usually all the jars in an EAR are dependency libraries. An interesting question arises. Aren't EJBs packaged as jars? How does the application server know to load the EJBs in a separate classloader? The simple answer is that it doesn't, unless you tell it so. You have to explicitly specify which of the jars are EJB modules in *application.xml* – the deployment descriptor for the EAR. The two dependency libraries from the sample.ear – util-a.jar and util-b.jar are loaded by EAR classloader.

WAR classloaders are the siblings of EJB classloader. The key difference between these classloaders is that while all EJB modules whether in the single or different EJB-JARs, share the same EJB classloader, each WAR gets its own classloader. All of the WAR classloaders however inherit from the same parent viz. EJB classloader. The rationale behind this hierarchy is that EJBs contain the core of the business logic and web applications have to "see" them to invoke their business

methods. Of course to "see" them the WAR manifest file has to have an entry as shown earlier. In the sample.ear, the *loanejb.jar* and *personejb.jar* are loaded by the EJB classloader, while the *sampleweb.war* is loaded by the WAR classloader.

An important rule of thumb is that classes should be no higher in the hierarchy than they are supposed to exist. When you put the classes at a level higher than where it is supposed to reside, you are introducing an avalanche of classes moving to the higher level. Consider the class dependency shown in Figure 21.6. At runtime, Class A depends on B, which in turn depends on C. Le t us suppose that all of these classes reside in the WAR classloader. If, for some reason, Class A is moved into EJB-JAR, but the rest of them are left in the WAR, Class A gets loaded by the EJB classloader. At runtime, when the Class A is loaded, it looks for all dependent classes (Class B, in this case). It cannot find Class B since Class B is loaded by the child classloader. To prevent this problem you are forced to move Class B into EJB-JAR, which in turn forces to move Class C into EJB-JAR. Consequently the entire class dependency graph has to be moved into EJB-JAR. The solution to this problem is application partitioning (Refer to Chapter 25 on Deployment Considerations).

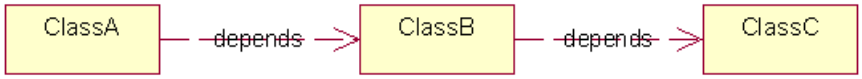

Figure 21.6 Runtime class dependency.

Classes should be no higher in the hierarchy than they are supposed to exist. When you put the classes at a level higher than where it is supposed to reside, you are introducing an avalanche of classes moving to the higher level.

21.4 Summary

In this chapter you gained an in-depth knowledge about classloaders. You also understood the three key principles in classloader operation. You looked into the hierarchy in J2EE classloaders and its variants. You also understood which artifacts are loaded by which classloaders. This information is critical while packaging your application into EARs, WARs and JARs. This knowledge, when applied during design time results in applications that are better "partitioned" – which we will deal in the next chapter.