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.

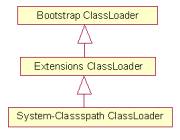


Figure 21.1 Classloader hierarchy for a stand alone Java program.

*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

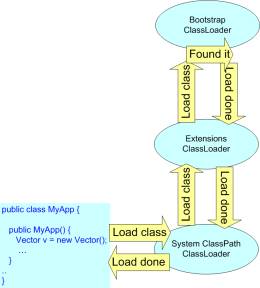
**2** You get this NoClassDefFoundError because even though EJB classloader - which is a parent of WAR classloader, loads the Person class, the *MANIFEST.MF* file from the WAR does not have an entry for the EJB-JAR. You can solve this problem by removing the Home and Remote interfaces from the WAR and 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.

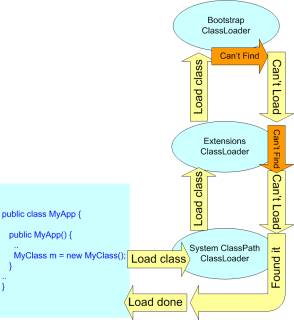


Figure 21.2 Classloader hierarchy illustrating the delegation when classes cannot be found

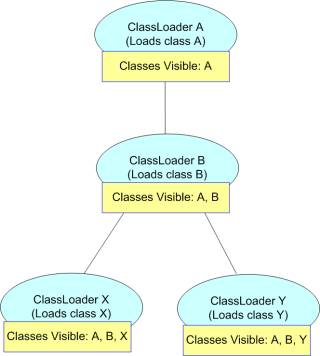


 Figure 21.3 Classloader hierarchy and classes visibility

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.