



Chapter 2

The Class Loader Architecture

This chapter introduces Java's class loader architecture, which provides a dynamic, flexible means for locating resources (both code and data) at runtime. Class loaders allow an application to be built from many disparate resources at runtime, and they provide a namespace mechanism that allows multiple versions of the same resource to coexist in a single virtual machine. Class loaders are also integral to the security architecture because they tell the virtual machine (VM) how and from where classes were loaded.

This chapter will cover finding and loading resources, the class loader delegation model for sharing resources, the standard class loaders provided by the core API, debugging class loading, and some straightforward examples of using class loaders. For information about the relationship between class loaders and the security model and for how to write your own class loaders, see Chapter 5.

2.1 Assembling an Application

Assembling a statically linked, standalone application is easy. In the simplest scenario, all of the binaries that constitute a standalone application are linked into a single file during development. Deploying the application is an all-or-nothing proposition. If you have possession of that single file, then the application is correctly deployed. If you do not have the file, then you do not have the application.

Actual practice is more complex. There are many reasons that you need to split applications into separate components:

- Different applications may share the same components. It is wasteful to deploy a separate copy of a component for each application that uses it.



- Applications change over time. If you make a small change to one component of an application, you should be able to redeploy that component without redeploying the entire application.
- Applications are dynamic. Based on conditions at runtime, applications may incorporate new resources (both data and code) in ways that cannot be fully anticipated during development.
- Applications use data resources such as graphic images. These resources are not managed by programmers and should be configurable without programmer intervention.
- Applications typically have configuration settings that are specified by the deployer of the application. These settings are logically part of the application but are not available until runtime.

All of these issues are variations of the same theme: The components of an application need to be assembled dynamically at runtime, not statically during the development phase.

Assembling the components of an application at runtime poses several challenges for platforms such as Java. A component platform should implement an infrastructure for *locating* various components of an application so that they can be reassembled. When a component is located, the platform should *verify* that the component would actually work with the application. This is a security issue because you do not want rogue components corrupting your applications. It is also a configuration management issue because you may need to choose between multiple versions of a component. If a component cannot be located or verified, then the cause of the failure should be clearly reported to the application. Note that failure to find a single component need not mean that the application as a whole cannot continue. Most applications can operate in a reduced capacity even if some components are missing.

The developer of an application will rarely assert complete control over how components are fitted together. In a server product, it often makes sense to cede some control to administrators. A typical example of this is allowing an administrator to plug in a security subsystem or database driver for the application. Other parties that may want to define and configure components include graphic artists, user interface specialists, problem domain experts, even end

users. (I will use the term *deployment* to describe the process of linking components together and the term *deployer* to describe any person performing deployment tasks.)

Developers will often wear the deployer hat. Developers are *definitely* responsible for building applications that are deployment-friendly—those that give deployment control to the right people at the right times. However, most Java applications will allow some control of deployment at runtime, which poses several interesting challenges.

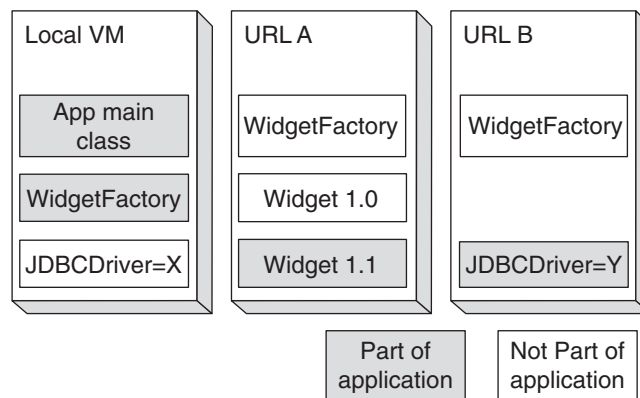


Figure 2–1 Assembling an application at runtime

Figure 2–1 illustrates some of the potential problems that a deployer may encounter when assembling an application. In this example the application's main class needs to find a `WidgetFactory`, the correct version of `Widget`, and some database (JDBC) configuration settings. The shaded components indicate the desired configuration, but there are several potential snags in linking the application correctly:

1. Not all of the resources are colocated with the application's main class, so deployers need some way to specify where the application should look for them.
2. Some resources can be found in more than one location. For example, the `WidgetFactory` is visible within the application, at URL A, and at URL B. A deployer should be able to select a component based on its location.

3. There are two different versions of the `Widget` class, version 1.0 and 1.1. A deployer should be able to automatically access the most recent version or be able to explicitly fall back to older versions when necessary.
4. Finally, the application wants to use some JDBC configuration settings from URL B, *not* the settings that are available locally. A deployer should be able to bypass or ignore local settings.

To make the type resolution process robust, the Java platform provides both reasonable default rules for how resources are located, and a mechanism deployers can use to customize this behavior. Additionally, Java can audit the linking process at runtime and optionally abort if the wrong resource is encountered. These services are the province of class loaders.

The Java class loader architecture provides the services that you need to assemble applications at runtime. Using class loaders, you can find and verify classes, resources, and configuration settings. Class loading services go well beyond support for simple scenarios to handle the challenges of real-world application deployment. For example, you can safely load resources, even code, from untrusted network locations. You can reload changed versions of a component without shutting down your application, or you can even permit multiple versions of a component to coexist at one time.

Unfortunately, most of the details of class loading are hidden from view during application development, and some key services are disabled by default. As a result, many application designs do not make effective use of class loaders. Instead, they rely on abstractions such as the classpath to hide the details of deployment. This approach to deployment leads to applications that cannot evolve and adapt at runtime. Worse yet, they do not even fail gracefully; instead, they terminate with exceptions that are far removed from the actual deployment problems. The remainder of this chapter will explain the class loader architecture and show you how to use and extend it effectively.

2.2 Goals of the Class Loader Architecture

“Design before you code” is a good rule, even when you are merely studying a system already in place. So, before you learn how class loaders work, you need



to understand the goals of the architecture. This section serves to justify, in advance, some of the complexity you will see in the class loader code.

The class loader architecture aims to be transparent, extensible, capable, and configurable. Also, it needs to deal sensibly with name conflicts and version incompatibilities without compromising the ability to share resources loaded from different places or in different ways. Finally, it has to define and enforce some notion of security so that there are specific control mechanisms for what dynamically loaded classes can and cannot do.

2.2.1 Transparency

Transparency is critical. Most Java code is, and should continue to be, written with no explicit awareness that class loaders even exist. Consider the `Simplicity` application shown in Listing 2-1. This code makes no mention of class loaders, yet somehow the `Simplicity` class is loaded into the virtual machine, and later the `RocketShip` class is also loaded. This implies that when the virtual machine begins execution, it knows how to find the application main class, and then somehow it infers where to look for additional classes as necessary. This loading happens implicitly, without any specific coding effort. The principle of least surprise also dictates that once the `RocketShip` class is loaded, it continues to be available and doesn't magically change its characteristics in any way. So, the `println` statement in `main` should output `true`.

Listing 2-1 The Simplicity Class

```
public class Simplicity {  
    public static void main(String [] args) {  
        RocketShip r1 = new RocketShip("Gemini");  
        RocketShip r2 = new RocketShip("Apollo");  
        System.out.println(r1.getClass() == r2.getClass());  
    }  
}
```

2.2.2 Extensibility

Class loading must be extensible. No matter what capabilities are built into the system, some particular user will need something different. Exotic class loader



designs might inject debugging information, smuggle optimization hints to just-in-time (JIT) compilers, pull binary code from source control systems, or enforce Eiffel-like invariants.¹ The class loader architecture should allow these, and other, extensions to be made without requiring modification to the classes to be loaded.

2.2.3 Capability

Class loading must be capable. Specifically, the core API should include class loaders that meet common needs, such as on-the-fly reloading of code that has changed on the file system, loading code from other nodes on the network, and loading code from compressed archives. Additionally, these facilities must not compromise simplicity by surprising the developer; for example, you should not suddenly discover that your application was unknowingly using code downloaded from <http://pureevil.org> instead of from your local hard drive.

2.2.4 Configurability

Class loading must be easily configurable. It is not sufficient to provide a class loading API that developers must explicitly code against. All of the standard scenarios above should be available as configuration options of the virtual machine. They should be accessible for modification by system administrators, not just Java developers. Also, it would be nice if there were an obvious system for adding configuration options for custom class loaders created by third parties, such as Java 2 Enterprise Edition (J2EE) container vendors.

2.2.5 Handling Name and Version Conflicts

Class loading must deal sensibly with name conflicts and with version incompatibilities. Java programmers are taught to avoid name conflicts through careful use of package names. The naming rule is that any classes that you ship must have package names that begin with your dotted domain name in reverse, followed by whatever internal package scheme your organization uses. For example, my company website is <http://www.develop.com>, so my packages begin with `com.develop`.

1. The Eiffel programming language provides constructs to specify *invariants*—conditions that must be true—at the beginning or end of a method's execution. Programs that encode invariants tend to be more readable and have fewer bugs. See [Mey00] for details.

The naming rule is a good start, but it is insufficient in a dynamic system because it relies on the goodwill and competence of all parties involved. In practice, the naming rule will fail in two ways:

1. Organizations have internal name collisions. Without a very strict top-down policy, this is more difficult to avoid than it might at first seem. Consider a company whose U.S. and European branches each define an interface named `com.myco.FootballPlayer`, for example. Worse still, some organizations might place code in the default package.
2. Organizations need to deploy multiple versions of the same class simultaneously. Consider a mission critical server that needs 24/7 availability. When a new version of a class becomes available, that class should be used to service new clients. Unfortunately, bugs happen and some clients may discover that they have an unintended dependency on an older version. A server should be able to continue to serve the old version of the class to those clients until the problem can be solved.

The runtime must be able to load multiple classes with the exact same name and keep track of them so that developers using the classes are never unpleasantly surprised. Moreover, it is inefficient to create a new virtual machine every time this situation occurs. Classes that change dynamically should be able to share the code from classes that change less frequently.

2.2.6 Security

Finally, the entire architecture must make some security guarantees. This is not often a major problem for monolithic applications where the entire application is viewed as a single whole. Dynamic class loading opens the possibility of multiple classes that do not trust each other, coexisting uneasily in the same virtual machine. The Java class loader architecture is supplemented by a security architecture that provides flexible, administrative security for code loaded from different sources.

2.3 Explicit and Implicit Class Loading

The simplest class loading API is provided by the `ClassLoader` class shown in Listing 2-2.

Listing 2-2 Explicit Loading with ClassLoader

```

package java.lang;
public class ClassLoader {
    public Class loadClass(String n)
        throws ClassNotFoundException;
    //remainder omitted for clarity
}

```

This is fairly straightforward. You pass in the full name of the class, delimiting packages with dots, and you get back a distinguished `Class` object that represents the loaded class. If the class loader fails, it throws the checked `ClassNotFoundException`.

2.3.1 Explicit Loading with URLClassLoader

The easiest way to see `loadClass` in action is to use a concrete `ClassLoader` implementation provided by the core API. By far, the most common and useful standard class loader is `java.net.URLClassLoader`. The class constructor takes an array of URLs, and it can find classes that reside at any of those URLs. The simplest type of URL is a file URL, which simply points to a location on the file system. Consider Listing 2-3.

Listing 2-3 Explicitly Loading from a URL

```

import java.net.*;
public class LoaderDemo {
    public static void main(String [] args) throws Exception {
        URL url = new URL("file:subdir/");
        URL[] urls = new URL[]{url};
        URLClassLoader loader = new URLClassLoader(urls);
        Class cls = loader.loadClass("LoadMe");
        Object o = cls.newInstance();
    }
}
public class LoadMe {
    static {
        System.out.println(LoadMe.class + " loaded.");
    }
}

```


Compile the `LoaderDemo` into one directory, and then compile `LoadMe` into a subdirectory named `subdir`. Navigate to the directory that contains `LoaderDemo.class`, and run `LoaderDemo` with the command line

```
java -classpath . LoaderDemo
```

The dot is shorthand for the current directory, so the Java launcher should be able to load `LoaderDemo`. Also, you will see that `LoaderDemo` successfully loads `LoadMe`, even though the `LoadMe.class` file is not on the classpath. Try deleting the `LoadMe.class` file, and run `LoaderDemo` again. This time you will see a `ClassNotFoundException`. In a real application, you could catch this exception and possibly continue execution, even with some classes missing.

The ability to load classes dynamically via a `URLClassLoader` is so powerful that more than 95 percent of all Java application classes are probably loaded by `URLClassLoader` and its subclasses. However, you will rarely call `loadClass` explicitly as shown above. *Explicit* class loading, in any form, runs counter to one of the primary goals of the class loader architecture: transparency. Explicit class loading requires work by the programmer for each class to be loaded, and it is too tedious for general use.

2.3.2 Implicit Class Loading

Java provides an intuitive mechanism for *implicit* class loading. Every Java class maintains a reference to the class loader that loaded it, accessible via the `getClassLoader()` method. Whenever a class *refers* to another class, the referent is loaded implicitly, using the same class loader that loaded the referring class. Listing 2-4 shows some examples of references.

Listing 2-4 Referenced Classes Are Loaded Implicitly

```
//reference to LoadMeBase
public class LoadMe extends LoadMeBase {
    //reference to LoadMeToo
    static LoadMeToo lmt = new LoadMeToo();
    //reference to LoadMeAlso
    static LoadMeAlso lma;
    static ClassLoader ldr = getSomeLoader();
    //Neither of these refer to LoadMeThree!
```

```
static Class cls = ldr.loadClass("LoadMeThree");  
static Object o = cls.newInstance();  
}
```

This new version of the `LoadMe` class refers to several other classes, all of which will be loaded implicitly when needed. First, `LoadMe` extends another class `LoadMeBase`. In order to verify that `LoadMe` is a legal extension of `LoadMeBase`, the `LoadMeBase` class must be loaded. Whenever a virtual machine loads `LoadMe`, it will attempt to load `LoadMeBase` using the same class loader.

`LoadMe` also has static references to objects of type `LoadMeToo` and `LoadMeAlso`. The virtual machine will load these classes only when they are actually needed to initialize a reference. Since the `LoadMeAlso` reference is initialized only to `null`, the virtual machine will not bother loading this class at all.²

2.3.3 Reference Type versus Referenced Class

Notice that `LoadMeThree` is *not* referenced by `LoadMe`, even though the class will be loaded when the code runs. The difference is that the references `cls` and `obj`, which are used to manipulate `LoadMeThree`, are statically typed as `java.lang.Class` and `java.lang.Object`, respectively. It is the compile-time type of a reference that triggers implicit class loading; so while `Class` and `Object` are *implicitly* loaded, the `LoadMeThree` class object is *explicitly* loaded by `someOtherLoader`. The subtle distinction between reference type and referent class allows implicit and explicit class loading to coexist.

Most Java code uses implicit references, all of which will be transparently loaded by a single `ClassLoader`. On rare occasions, a developer breaks this chain by using explicit class loading and assigning the result to some base-class reference type. Then, life continues as before. If `LoadMeThree` itself makes references, they will be loaded implicitly by the class loader that loaded `LoadMeThree`, that is, `someOtherLoader`.³

2. As an optimization, a virtual machine can preload classes that are not needed yet. However, it cannot make the effects of loading these classes visible to the program. So, if preloading a class caused an exception, that exception would not be thrown until the point in the code where the class was actually first used.

3. This is not completely true but will have to do for now. The delegation mechanism makes things a little more complex, as you will see momentarily.

2.3.4 `ClassLoader.loadClass` versus `Class.forName`

The `loadClass` method shown in Listing 2–2 is actually one of a family of methods in the `Class` and `ClassLoader` classes. The complete list is shown in Listing 2–5. All the methods look similar, so which should you use?

Listing 2–5 Explicit Class Loading APIs

```
//all methods listed throw ClassNotFoundException
package java.lang;

public class Class {
    public Class forName(String name);
    public Class forName(String name, boolean resolve,
                        ClassLoader cl);
    //remainder omitted for clarity
}

public class ClassLoader {
    public Class loadClass(String name);
    public Class loadClass(String name, boolean resolve);
    //remainder omitted for clarity
}
```

The various explicit loading APIs differ in three ways:

1. The APIs that take a `resolve` parameter allow the user to pass in `false` to postpone linking the class. Since the virtual machine will have to link the class before it is used anyway, this option is rarely needed. For more on linking see [Ven99].
2. The single-argument version of `forName` is a shortcut for using the caller's class loader.
3. Some implementations of the `loadClass` method will not load arrays that are not already loaded into the virtual machine. The API docs do not make clear whether this is a bug, but it is “fixed” in SDK version 1.3.

Because of `loadClass`'s odd history of mishandling arrays, `Class.forName` is probably the better choice.



2.3.5 Loading Nonclass Resources

In addition to loading classes, class loaders can also load arbitrary resources. The relevant methods are demonstrated in Listing 2–6.

Listing 2–6 Loading Resources

```
import java.io.*;
import java.net.*;
import java.util.*;
public class LoadResources {
    public static final String res = "config.props";
    public static void main(String [] args)
        throws IOException
    {
        ClassLoader cl = LoadResources.class.getClassLoader();
        System.out.println("All resources at " + res);
        Enumeration enum = cl.getResources(res);
        while (enum.hasMoreElements()) {
            System.out.println(enum.nextElement());
        }
        URL url = cl.getResource(res);
        System.out.println("First resource at " + res);
        System.out.println(url);
        InputStream is = cl.getResourceAsStream(res);
        Properties props = new Properties();
        props.load(is);
        System.out.println("Properties from " + res);
        props.list(System.out);
    }
}
```

The call to `getResources` returns an enumeration of URLs for all the resources found at the specified path. More than one match is possible since a class loader can search multiple locations. The call to `getResource` returns only the first matching URL. The convenience method `getResourceAsStream` uses `getResource` to locate a URL, then connects to the URL and opens an `InputStream` on the data.

The resource loading functions are typically used to load data needed by an application class, such as configuration information or images for a graphical application. By using a class loader instead of talking directly to a file system,



you simplify deployment of applications. If you place application resources in the same location as your application classes, you do not have to worry about finding resources.

While standard class loaders such as `URLClassLoader` implement the resource loading methods correctly, custom class loaders (discussed in Chapter 5) may ignore these methods or throw an exception. If you rely on class loaders to load resources, make sure that the class loaders in your application are resource-aware.

2.4 The Class Loader Rules

Implicit class loading provides simplicity, and explicit class loading provides flexibility. However, these mechanisms must have some additional properties to deal with the tricky issue of class visibility. What communication should be possible between classes loaded by two different class loaders? The simple answers to this question have undesirable properties: If there is no communication across class loader boundaries, then there is no real benefit to dynamic class loading, and separate class loaders might just as well be separate virtual machines. On the other hand, complete visibility between class loaders leads to chaos, as there is no way to hide conflicting names or versions from each other. So, the class loader architecture adopts a middle path characterized by the following three rules:

1. The consistency rule: Class loaders never load the same class more than once.
2. The delegation rule: Class loaders always consult a parent class loader before loading a class.
3. The visibility rule: Classes can only “see” other classes loaded by their class loader’s *delegation*, the recursive set of a class’s loader and all its parent loaders.

In combination, these three rules provide a workable solution to all of the design problems listed above.

2.4.1 The Consistency Rule

The consistency rule is the easiest to understand. Once a class is loaded, any future attempts to load the same class from the same class loader must return

the already-loaded class. This rule is necessary to prevent nasty surprises for developers and for the virtual machine. It is also easy to implement; class loaders simply keep a `HashMap` or other data structure of already-loaded classes and consult that structure before attempting to load the class again.

The consequences of breaking the consistency rule are dire:

1. Implicit class loading no longer makes any sense. If more than one version of a class is loaded, you have no way to know which one is used when an implicit reference is made.
2. The integrity of developer code is endangered. For example, imagine that your code assumes that class `Foo` has a field named `bar`. What happens when that field suddenly disappears because a new version of `Foo` has been loaded?
3. The integrity of the virtual machine is endangered. Virtual machine implementations are built based on the assumption that class loaders never reload the same class, and their behavior is undefined if this actually occurs. This is even worse than compromising developer code because a damaged virtual machine might crash, corrupt data, or open a security hole.

Given that following the consistency rule is easy, and the consequences of breaking it are so dire, you may wonder why I belabor the point. The problem is that breaking the consistency rule appears to be a simple way of replacing classes at runtime. On some virtual machines, this can even appear to work for a while. Do not be fooled. The correct way to replace classes on-the-fly (§2.5) has nothing to do with reloading classes into the same `ClassLoader` instance. The delegation and visibility rules provide the necessary framework for replacing classes on-the-fly. When implemented correctly, dynamic replacement is portable to all legal VMs, and it has none of the liabilities associated with trying to shoehorn new classes into old class loaders.

2.4.2 The Delegation Rule

The delegation rule states that class loaders always consult a parent class loader before loading a class. `ClassLoader`'s constructor takes a single argument of type `ClassLoader`, which is the parent of the new class loader. The

parent is available to application code via the `getParent()` method.⁴ The set of a class loader and all of its ancestors is called a class loader *delegation*. Because each class loader checks with its parent, the entire delegation will be checked recursively before a specific class loader is allowed to load a class.

The overall effect of the delegation rule is to allow limited sharing of classes between class loaders. Consider Figure 2–2, which shows a class loading scenario that might occur in a web browser. The browser will provide applets from different domains with unique `URLClassLoaders` to load their own specific classes. However, the applet class loaders must consult their delegations before loading any class. So, when an applet refers to system classes such as `java.lang.Object` or `java.net.URL`, it will get the versions provided by the system class loader. All applets are forced to share the same version of these core classes. As you will see later, this point is critical to security since the core classes include security classes such as `java.lang.SecurityManager`, not to mention classes that can access your hard drive such as `java.io.FileOutputStream`.

2.4.3 The Visibility Rule

The visibility rule states that classes can only see other classes loaded by their class loader's delegation. Consider Figure 2–2 again. Both `good.org` and `pureevil.org` have a class with the same name, `org.good.Main`. Assuming that you browse to `good.org` first, the following sequence of events ensues:

1. The browser creates a class loader `CLgood` for `good.org`.
2. `CLgood` is asked to load `Main`.
3. `CLgood` first delegates to the system class loader.
4. The system class loader fails to find `Main`.
5. `CLgood` loads `Main` from `good.org`. Any classes that `Main` refers to will be implicitly loaded by `CLgood`'s delegation.
6. The browser creates a class loader `CLevil` for `pureevil.org`.

4. `ClassLoader` also has a convenience constructor that takes no arguments. If you use this version, then the new class loader will take the result of `ClassLoader.getSystemClassLoader()` as its parent. The system class loader finds classes from various locations including the classpath and is discussed in detail shortly.

7. CL_{evil} executes steps 2 through 4 again.
8. CL_{evil} loads `Main` from `pureevil.org`. As far as the VM is concerned, this class is *entirely different* from the class of the same name loaded in step 5.

The fact that `pureevil.org` introduces its own private version of `org.good.Main` has no effect on the execution of the “real” `org.good.Main` loaded from `good.org`. Each applet gets its own copy of any application-specific classes. This helps to guarantee that applets do not accidentally or maliciously manipulate code loaded by other applets. At the same time, delegation guarantees that all applets share the core API classes such as `java.lang.String` and `java.io.FileOutputStream`. This helps to protect the integrity of the core API and is far more efficient than loading multiple copies of exactly the same code.

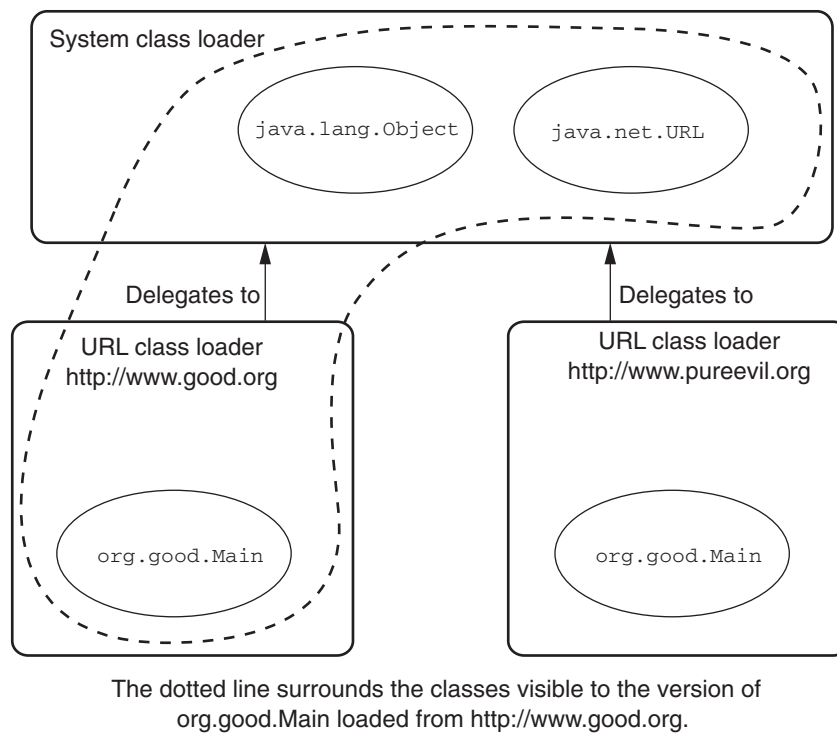


Figure 2–2 Class loader delegations

2.4.4 Delegations as Namespaces

The class loading rules have several surprising implications. The first is that a class loader delegation forms a namespace. The Java language already has namespaces in the form of packages; for example, `com.develop.String` is a totally different class than `java.lang.String`. Once you factor in class loading, the idea of namespace includes not only a class's package name, but also which class loader loaded the class. This fact is easily verified by executing Listing 2–7.

Listing 2–7 The Same Binary Class Loaded by Two Different Loaders

```
import java.net.*;
public class LoaderDemo2 {
    public static void main(String [] args) throws Exception {
        URL url = new URL("file:subdir/");
        URL[] urls = new URL[]{url};
        URLClassLoader loader = new URLClassLoader(urls);
        URLClassLoader loader2 = new URLClassLoader(urls);
        Class cls = loader.loadClass("LoadMe");
        Class cls2 = loader2.loadClass("LoadMe");
        Object o = cls.newInstance();
        //cls == cls2 is false
        System.out.println("(cls == cls2) is " + (cls == cls2));
    }
}
```

As far as the VM is concerned, `cls` and `cls2` are just as different as classes with different names. In this example, the classes happen to be loaded from the same location, so the bytecodes are exactly the same.⁵ However, the classes might just as well have been loaded from different locations, or at different times. The classes might contain different fields, different method signatures, or different method implementations. Since the VM considers the classes to be totally different anyway, it does not have to know, or care.

The delegation model introduces another wrinkle into the namespace issue. Try running the `LoaderDemo` example, but this time, install the `LoadMe` class

5. The one exception to this is if the `LoadMe.class` file is replaced while this program is running. In this case, it is possible that two different files would be loaded. This is discussed in more detail later in this chapter in the Hot Deployment example.

into the same directory instead of into `subdir`. Now, you will see a totally different result—`LoaderDemo` will report that `cls` and `cls2` are the same, even though different loaders were asked to load each. What has happened? Both `loader` and `loader2` implicitly delegate to the system class loader. When you moved `LoadMe` into the same directory as `LoaderDemo`, you made `LoadMe` visible to the system class loader. Neither `loader` nor `loader2` actually loads the `LoadMe` class. So, the first run reported that

```
LoadMe_loader != LoadMe_loader2
```

and the second run reported that

```
LoadMe_system == LoadMe_system
```

Though this delegation behavior is efficient and allows controlled sharing of code, it is also the cause of most class loading errors. The bugs all follow this general pattern: A developer designs a class to be loaded dynamically. Then, the class is unintentionally copied onto the classpath. Because all class loaders delegate to the system class loader,⁶ the classpath version of the class is always the one found. The symptom of this bug is that dynamic class loading does not occur. If the purpose of dynamic class loading was to centralize deployment to a website, new versions of the class never seem to work on client machines. This bug is difficult to detect because no exception is thrown. The virtual machine succeeds in finding a class, but it loads the “wrong” version of the class that is visible to a parent class loader.

2.4.5 Static Fields Are Not Singletons

The singleton design pattern models an entity that appears only a single time in a system. A simple approach to implementing the singleton pattern in Java is to create a class with a private constructor to prevent accidental creation of multiple instances. Then, the class can provide a static method that returns the single instance of the class, as shown in Listing 2–8.

6. This is an example of being economical with the truth. It is possible to use the single-argument `ClassLoader` constructor to deliberately bypass the classpath, but this usage is very rare in practice.

Listing 2-8 A Naive Singleton Implementation

```
public class NaiveSingleton {
    private static NaiveSingleton onlyOne;
    private NaiveSingleton() {}
    static {
        onlyOne = new NaiveSingleton();
    }
    public NaiveSingleton getInstance() { return onlyOne; }
}
```

Dynamic class loading thwarts this approach because over time, multiple versions of a class might be loaded, each with its own copy of the static field. As a result, designing a singleton in Java requires some additional thought about how to maintain the singleton identity in a dynamic environment. Some solutions include keeping the singleton in a class that is not loaded dynamically, or storing the singleton state outside of a particular Java instance using JDBC or EJB.

2.4.6 Implicit Loading Hides Most Details

“Normal” code that relies on implicit class loading will never see any of these issues. Implicit class loading is kept simple by the consistency and visibility rules. The consistency rule says that there will be only one version of a class loaded by a particular class loader, and the visibility rule says that the class loader delegation defines the set of all visible classes. So, there may be multiple different versions of classes floating around your VM, but unless you explicitly use class loaders, you will only ever see one of them.

Even when you do make explicit use of class loaders, you can use abstraction to hide this complexity from the bulk of your code. The next section will show you how to divide your application into two distinct parts, a simple client part that doesn’t worry about class loader issues, and a server that does the grunt work to provide a useful service based on class loaders.

2.5 Hot Deployment

Class loaders can be used to load multiple versions of a class from different locations in space, but they can also be used to load different versions of a class from the same location at different times. This ability, often called *hot deployment*, is

useful for redeploying incremental changes to a class without having to shut down the virtual machine. In the example that follows, client code⁷ will use multiple versions of a class, without writing any explicit class loading code. The server code will instantiate new class loaders as necessary to load new versions of the class.

The example will use that classic OO staple, the two-dimensional `Point`, as seen in Listing 2–9.

Listing 2–9 The Point Interface

```
public interface Point {
    public int getX();
    public int getY();
    public void move(int dx, int dy);
}
```

The initial `PointImpl` implementation is defective, as shown in Listing 2–10. It will later be replaced without shutting down the VM.

Listing 2–10 A Defective PointImpl

```
public class PointImpl implements Point {
    private int x;
    private int y;
    public int getX() { return x; }
    public int getY() { return y; }
    public void move(int dx, int dy) {
        x += dx;
        //oops! forgot to move y
    }
    public String toString() {
        return "Point at " + x + ", " + y;
    }
}
```

The client code is a `PointClient` class that creates a `Point` and moves it around. However, `PointClient` should not reference the `PointImpl` implementation class through a variable of type `PointImpl`. If it does, the rules of implicit class loading will take over, and the `PointImpl` class will be loaded by

7. Throughout the book, I will use the terms “client” and “server” in their most generic sense. Client code is code that utilizes some service provided by server code.

the same class loader that loads `PointClient`. This defeats the purpose of the example. A single version of `PointClient` should be able to load and use new versions of `PointImpl`. This is impossible if `PointClient` is loaded from the same loader because the consistency rule forbids unloading and reloading `PointImpl` from a single loader. Therefore, the creation of `PointImpl` is hidden behind a `PointServer` object that handles the details of explicitly loading new versions of `PointImpl`, as shown in Listing 2-11.

Listing 2-11 The PointServer Class

```
import java.net.*;
public class PointServer {
    static ClassLoader cl;
    static Class ptClass;
    public static synchronized Point
        createPoint(Point template) throws Exception
    {
        if (ptClass == null) reloadImpl();
        Point newPt = (Point) ptClass.newInstance();
        if (template != null) {
            newPt.move(template.getX(), template.getY());
        }
        return newPt;
    }
    public static synchronized void reloadImpl()
        throws Exception
    {
        URL[] serverURLs = new URL[]{new URL("file:subdir/")};
        cl = new URLClassLoader(serverURLs);
        ptClass = cl.loadClass("PointImpl");
    }
}
```

The `PointServer` object acts as a factory, providing a `createPoint` method that returns a `Point`. The `template` parameter can be used to initialize the new `Point` to match an existing `Point`. If `templatePoint` is null the new `Point` will start at the origin. `PointServer` also provides the helper method `reloadImpl` to load a new version of `PointImpl`.

The `PointClient` class provides a simple command-line interface to move a `Point` or reload the `PointImpl` implementation, as shown in Listing 2-12.

Listing 2-12 PointClient

```

import java.io.*;
public class PointClient {
    static Point pt;
    public static void main(String [] args) throws Exception
    {
        BufferedReader br = new BufferedReader(
            new InputStreamReader(System.in));
        pt = PointServer.createPoint(null);
        System.out.println(pt);
        while (true) {
            System.out.println("MOVE1, RELOAD, or EXIT");
            String cmdRead = br.readLine();
            String cmd = cmdRead.toUpperCase();
            if (cmd.equals("EXIT")) {
                return;
            } else if (cmd.equals("RELOAD")) {
                PointServer.reloadImpl();
                pt = PointServer.createPoint(pt);
                System.out.println(pt);
            } else if (cmd.equals("MOVE1")) {
                pt.move(1,1);
                System.out.println(pt);
            }
        }
    }
}

```

In order to run this example, you will need to compile the `PointImpl` class into a directory named `subdir`, and you will need to compile `Point`, `PointClient`, and `PointServer` into your top-level directory. Then, when you run the application and enter the appropriate commands, you will see a session similar to Listing 2-13.

Listing 2-13 A Session with PointClient

```

Point at 0, 0
MOVE1, RELOAD, or EXIT
MOVE1
Point at 1, 0
MOVE1, RELOAD, or EXIT
RELOAD

```

```
Point at 1, 0
MOVE1, RELOAD, or EXIT
MOVE1
Point at 2, 0
MOVE1, RELOAD, or EXIT
```

The `move` implementation is broken; although the `PointClient` is trying to increment both `x` and `y`, only `x` is changing. Simply reloading the class doesn't fix the problem. Even though the `PointServer` is loading a new version of `PointImpl`, it contains the same defective code as the original. You need to repair the bug in Java and then recompile the fixed version. Make sure that you leave the `PointClient` process running while you make these changes. Fix and recompile `PointImpl` to correctly increment `y`. Now, try again to reload and move the `Point`.

Listing 2-14 Moving the Corrected `PointImpl`

```
RELOAD
Point at 2, 0
MOVE1, RELOAD, or EXIT
MOVE1
Point at 3, 1
MOVE1, RELOAD, or EXIT
EXIT
```

Listing 2-14 shows that after you reload the changed `PointImpl`, the `move` operation now works correctly. You can use this technique to replace code in the field without forcing clients to shut down their applications.

2.5.1 Using Hot Deployment

There are several points to remember when you apply hot deployment in your own applications. The first three have to do with how the classes get loaded. First, clients must not reference the type that will need to be dynamically replaced. If they do, one of two bad things will happen: The client will implicitly load the classes with its class loader, or it will fail to load the classes at all. Either way, there will be no way to get a new version of the server class without shutting down the client.

The second point is implied by the first one. Clients cannot use a reference of the implementation type; they must use a reference of a base class or interface type. As the `PointClient` example shows, the client uses a reference of type `Point`. This base interface is loaded implicitly by the client's class loader. Of course, this does imply that you have to shut down the client if you want to change the *interface* that the client is programming against. This is reasonable since the client would have to write some new code to use a new interface anyway.

The third point is that the implementation class must be able to find the same version of the base interface that the client is using. In other words, the implementation's class loader must delegate to the client's class loader. This is implicit in the `Point` example because the `PointClient` is loaded by the system class loader, which is the default parent for new class loaders.

The remaining issues deal with the relationship between old and new versions of a class. The VM recognizes no relationship between old and new versions of a class, so you must manufacture any necessary relationship in your code. In the example, the `PointServer` class manages the relationship between old and new versions of `PointImpl`. When a new `PointImpl` is instantiated, the `PointServer` uses the old `PointImpl` as a template, thereby maintaining the state that the client had already accumulated. This requires two specific actions on the part of the client. First, the client must make the state of the original object available to the server factory so that it can correctly instantiate the new version. Second, the client must drop any references to the old version of the `PointImpl`, both to preserve resources and to take advantage of the new version of the code. The client executes both of these actions via the single line of code⁸

```
pt = PointServer.createPoint(pt)
```

8. With one more level of indirection, you could hide these details from the client as well. The server gives the client a forwarding proxy to the actual object, which allows the server to control the object itself and swap it out at any time, possibly without consent or even awareness on the part of the client.



2.6 Unloading Classes

Once you switch all of your applications to use dynamic class loading, what happens to old versions of classes that nobody is using anymore? Class loaders and classes require a noticeable amount of memory, and if this memory is not reclaimed, your highly available server application will eventually fail. Of course, in Java you cannot explicitly destroy an object. The absence of an explicit `delete` operator is one of the primary safety features of the language. So, class loaders and classes must be reclaimed just like any other object, by the garbage collector (GC).

The VM specification does not make any special distinction about how class or class loader objects will be collected. This means that virtual machines may garbage collect these objects exactly like they would recover any other Java objects. On the other hand, the spec does not *prevent* a VM from handling these objects separately or even from refusing to unload them from memory.

The SDK implementation of Java treats classes and class loaders slightly differently from other objects. The SDK garbage collector defaults to treating classes and loaders just like any other objects, and it can collect them when there are no existing references. You can disable this behavior with the non-standard launcher flag `-Xnoclassgc`. You will need to check the documentation, or possibly even write test cases, to determine the behavior of other virtual machines.

2.6.1 Making Sure Classes Are Collectable

If you are writing server applications, it is probable that you will deploy them on a VM that can garbage collect classes. The only thing you have to worry about is making sure that your code drops all references to classes that you want the GC to reclaim. This is easier said than done. Unintended references are one of the major memory problems in Java, and they can strike any kind of code.

The consequences of unintended references are particularly severe when class loaders are involved. Consider these facts. Every instance of a class maintains a reference to its `Class` object, accessible through the `getClass()` method. Moreover, every instance of `Class` maintains a reference to its `ClassLoader`, accessible via `getClassLoader()`. In turn, class loaders hold



a cache of every `Class` that they have ever loaded. The net result of all of these references is that *a single instance of any class loaded by a class loader will keep the class loader, and every class that it loaded, in memory.*

In a simple example like the `PointClient`, references are easy to track down. The three references that might keep an old class loader in memory are `PointClient.pt`, `PointServer.cl`, and `PointServer.ptClass`. Each of these references is reset whenever a new version is loaded, so there is no danger that an unintended reference will hold the old classes in memory. In a larger application, it is important to keep track of these “problem” references during the design phase to make sure that none of them cause trouble later.

2.7 Bootclasspath, Extensions Path, and Classpath

Java provides several mechanisms for configuring how classes are loaded, which do not require any explicit class loader code. One of these mechanisms, the classpath, is the only interaction with class loaders that some Java applications will ever need. The classpath provides a simple mechanism to specify a set of locations where your code can be found. Despite its simplicity, the classpath is the source of a great many headaches, even for experienced Java developers. By understanding the classpath in terms of the class loader architecture, though, you can avoid these problems or quickly troubleshoot them when they occur.

The classpath is only one part of Java’s class loader configuration options. When you run an application with the standard Java launcher `java`, your application begins life with a set of three class loaders already in place: the bootstrap, extensions, and system class loaders, as is shown in Figure 2–3. The bootstrap class loader loads the core system packages from the bootclasspath, the extensions class loader loads extensions to the core API from the extensions path, and the system class loader loads application code from the classpath. The three loaders are often treated as a single loader named the system class loader.⁹ This three-tiered design accomplishes two objectives: simplicity for the

9. It is confusing that the term “system class loader” can be used to mean either (1) the loaders that consult the classpath, or (2) the loader plus its delegation. In practice, this distinction rarely matters. The API call `ClassLoader.getSystemClassLoader()` returns the classpath class loader, but because of the delegation model this loader can also load bootstrap or extensions classes.

common case, and great flexibility for the rare cases where you need to make sweeping changes to deployment, security, or how the core classes are loaded.

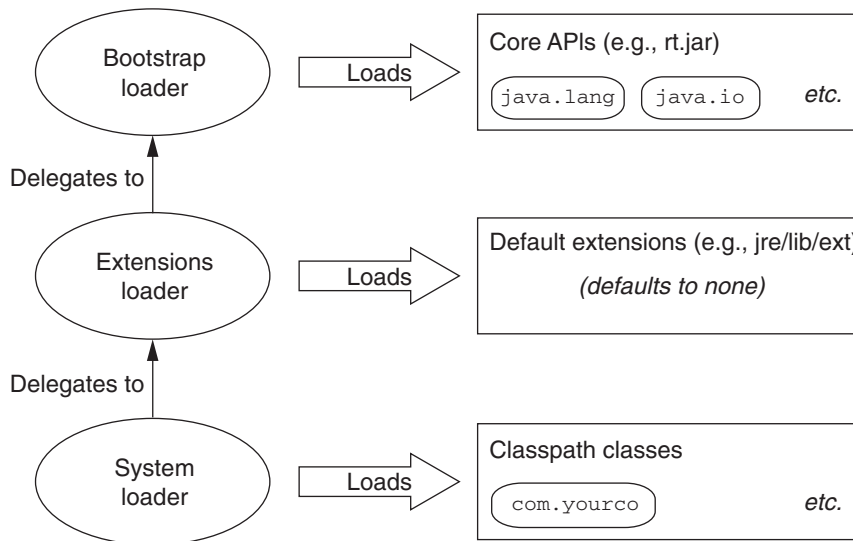


Figure 2-3 The three standard class loaders

2.7.1 The Classpath

The common case mentioned previously involves only the system class loader. This class loader is variously called the application class loader, the classpath class loader, or the system class loader. Although “classpath loader” is the most informative, the API functions that deal with this loader use the term “system,” therefore, system will be preferred here. Regardless of how you name this loader, it simply loads classes from directories and Java Archive (JAR) files listed on the classpath.

There are several ways to control the initial setting of the classpath. In the absence of any other information, the classpath is set to “.”, the directory from which the `java` launcher is run. This setting can be overridden by specifying a `CLASSPATH` environment variable. The environment variable, in turn, can be overridden via a command-line switch. The `java` launcher recognizes either `-cp` or `-classpath`, but note that many other Java SDK command-line tools

recognize only the longer `-classpath` form. Finally, the `-classpath` command-line switch can be overridden by yet another command-line switch, `-jar`. If you use the `-jar` switch, then user classes will only be loaded from a single specified JAR file. These options are summarized in Table 2-1.

Time and experience have proven that using the command line is the cleanest, most reliable way to set the classpath. If you set the `CLASSPATH` environment variable, then you always run the chance of another application or user resetting it to point somewhere else. Similarly, you cannot rely on the implicit use of the current directory because if any other actor on your system has set the `CLASSPATH`, the current directory will be ignored. When you set the classpath from the command line, you have complete control. The command line is the one place to look in case of problems, and changes elsewhere on the machine will not cause mysterious failures loading classes. For this reason, choose the command-line method of setting the classpath over any other method whenever you are using the `java` launcher.

The value of the classpath is a delimited list of directories or JAR files. The delimiter is platform specific, and it is available on each platform as the constant `java.io.File.pathSeparatorChar`. Examples in the text will use a semicolon, which is the delimiter for Win32 operating systems. If a class is not in the default package, then the class's package name must be combined with the classpath to locate the class. For instance, the command line

```
java -cp MyJar.jar;MyClasses com.develop.Test
```

Table 2-1 Setting the Classpath

Setting	Comments
<code>-jar</code> switch	List only a single JAR
<code>-cp</code> switch	List a mix of JARs and directories
<code>CLASSPATH</code> environment variable	Not recommended
Current directory	Not recommended



will only be able to locate the file `Test.class` if it is located in the

```
MyClasses/com/develop/
```

directory, or if it is stored in `MyJar.jar` with directory information included, as

```
com/develop/Test.class
```

The conversion of the package name into a directory hierarchy is necessary to distinguish classes with the same base name but different package names.

The launcher's command-line processing is relatively forgiving. The launcher accepts relative paths and forgives your choice of a directory delimiter; for example, the names `MyClasses`, `MyClasses/`, and `MyClasses\` all correctly locate classes in the `MyClasses` subdirectory of the current directory. However, there are other places in the Java world where the only legal choice is the `'/'` delimiter, including a trailing `'/'` at the end of a string specifying a directory. If you develop the habit of using `foo/` to specify a directory name on the command line, you will be using a consistent syntax that will also work in other places such as file URLs.

2.7.2 The Extensions Path

The system class loader delegates to the extensions class loader. The extensions class loader loads *installed optional packages*, which the Java Extension Mechanism [Ext] defines as “packages housed in one or more JAR files that implement an API that extends the Java platform [...] in the sense that the virtual machine can load them without their being on the class path, much as if they were classes in the platform's core API.” This characterization implies that the extensions class loader should be used to share packages that are common to a large number of applications for which it would be inconvenient to repeatedly specify locations on the classpath. Examples of such packages include XML tools and custom security providers.

In short, installed optional packages are JAR files that are placed in a set of well-known directories and made available to all applications. Placing classes on





the extensions path differs from placing classes on the classpath in two major ways:

1. It is easy to specify a large number of JAR files at once.
2. Extensions pass all security checks.

Directories in the extensions path are automatically scanned for JAR files. Directories in the classpath are not scanned, so you must list each JAR file individually. Note, however, that the extensions loader loads *only* from JAR files, never from class files. The reason for this is the assumption that the code for common libraries should be deployed as JAR files. When they are deployed this way the danger of loading mismatched versions of classes is minimized.

By default, installed optional packages are loaded as needed from the

```
${JAVA_HOME}/lib/ext
```

directory. It is possible to override this by setting the `java.ext.dirs` property on the command line. For example,

```
java -Djava.ext.dirs=myext1;myext2 MyMainClass
```

would allow any JAR files under the `myext1` or `myext2` directories to be loaded as installed optional packages. In practice, you are unlikely to override the location of the extensions directory. Since the purpose of extensions is to be omnipresent, it usually makes little sense to override the extensions path on a per-application basis.

Some developers use the extensions path in place of the classpath because they like the convenience of not listing each JAR individually. I would discourage this practice if you are concerned about security, however, because it operates against the intention that extensions are exempt from security checks. Per-application class settings are better accomplished by setting a command-line classpath.

The second major difference between installed optional packages and classpath classes has to do with security. When Java 2 security is enabled, like it is with the `-Djava.security.manager` launcher flag, installed optional packages default to being *completely* trusted by the virtual machine. This high level



of trust is enabled by the following entry in the default `java.policy` file, which resides at `${JAVA_HOME}/lib/security/java.policy`:

```
grant codeBase "file:${java.home}/lib/ext/*" {  
    permission java.security.AllPermission;  
};
```

On the other hand, classes loaded from the classpath run with a minimal set of permissions, which does not include general access to the network, local files, or critical virtual machine subsystems. Both the classpath and extensions security settings can be changed, augmented, or replaced on a per-VM basis with custom policies. Nevertheless, the design assumes that classes in installed optional packages will normally be trusted parts of the system and that application classes from the classpath will not.

2.7.3 The Bootclasspath

The extensions class loader delegates to the bootstrap class loader, sometimes also known as the primordial class loader. The bootstrap class loader checks the bootclasspath and loads the core API packages such as `java.lang`, `java.io`, and `java.util`, usually from a file named `rt.jar`. The bootstrap class loader is almost certain to be implemented in native code, and unlike all other class loaders, it does not present itself as a Java identity. When you call `getClassLoader()` for a bootstrap class, the result will be `null`.

The bootstrap class loader has two significant properties that separate it from all other class loaders; both are related to security:

1. Classes loaded by the bootstrap class loader are not verified, that is, the virtual machine assumes that they are well-formed binary classes.
2. Classes loaded by the bootstrap loader are not subject to security checks. This is subtly different from installed optional packages loaded by the extensions class loader: Bootstrap classes are never even asked for their permissions, while installed optional packages are asked but default to having all permissions anyway.

You will rarely want to place your own classes on the bootclasspath. There is no need to do so for security reasons or to make deployment convenient because

installed optional packages work well for these purposes. There are only two situations in which it makes sense to deploy on the bootclasspath: if you wish to replace the core classes with custom versions, or if the core API classes need to have direct access to your code.

Replacing core classes is a useful debugging trick. In its simplest form, you might want to recompile the core classes with the `javac -g` flag to generate debugging information, which the default `rt.jar` lacks. Many Java integrated development environments (IDEs) include an `rtd.jar` file, which has debugging information included, and they automatically set the bootclasspath to this version of the core API when they start a debugging session by calling

```
java -Xbootclasspath:rtd.jar {vmArgs} YourMainClass
```

You might also want to recompile a subset of the core classes to insert debugging information. In this case, you can use the

```
-Xbootclasspath/a: or -Xbootclasspath/p:
```

flags, which append or prepend your classpath to the normal bootclasspath, respectively. The following section (§2.8) demonstrates one use of this technique.

The other situation that demands changes to the bootclasspath is when the core API classes need direct access to your code. The only case where this might occur is when the core API provides a factory method for installing some service that you define. In this situation, your service class must be visible to the class loader expected by the factory method.

One situation that requires changes to the bootclasspath is loading a custom security policy. The security policy implements the mapping between a class and the security permissions that are granted to that class. The exact details of how the policy works are unimportant here; suffice it to say that you can configure the virtual machine to replace its standard policy with a class of your own choosing.

However, there is a hitch. Look at Listing 2–15, which shows pseudocode for loading a custom policy. Notice the single-argument version of `Class.forName`, which relies on implicit class loading. Since the system code resides in

rt.jar, it is loaded by the bootstrap class loader. In order to be visible, a custom policy implementation must also be loaded by the bootstrap loader. If you write a custom policy, you will need to set the bootclasspath to point to your custom policy class.

Listing 2–15 Loading a Custom Security Policy

```
String custPolicy = System.getProperty(  
    "java.security.policy");  
Class cls = Class.forName(custPolicy);  
Object o = cls.newInstance();  
Policy.setPolicy((java.security.Policy) o);
```

Taken together, the bootstrap, extensions, and system loaders provide a large variety of options for deployment and security settings on a single machine. This is often sufficient for configuring simple applications. If you want to dynamically deploy and redeploy classes, the standard class loaders won't help you. You, or some other code in your process, will need to install some additional class loader instances, probably of `java.net.URLClassLoader`.

2.8 Debugging Class Loading

The flexibility of the class loader architecture creates the potential for extreme confusion when something goes wrong. Consider the simple situation of an application failing with a `ClassNotFoundException` for class `Foo`. This could be caused by the `Foo.class` file being in the wrong directory, by an incorrect `-cp` parameter on the command line, by a problem with the `CLASSPATH` environment variable, or by any of the bootclasspath or extensions path settings. Even worse, the failure to load `Foo` could be the result of a chain reaction caused by some other class `Quux` being loaded from an unexpected location. Once `Quux` is loaded, all of the classes that `Quux` references are implicitly loaded from `Quux`'s class loader delegation, so if `Quux` is in an unexpected place (but still visible), it may cause bizarre loading failures at distant locations in the code.

All of these problems can happen even with the standard class loaders. When you start instantiating your own class loaders, the situation becomes even



more bewildering. This section will show you three tricks for debugging class loading problems:

1. Instrumenting your application
2. Using the `-verbose:class` flag
3. Instrumenting the core API

2.8.1 Instrumenting an Application

You can often diagnose class loading problems by instrumenting your application near the trouble spot. For example, consider Listing 2–16, which you might use to troubleshoot the case of class `Quux` being unable to implicitly load class `Foo`.

Listing 2–16 Instrumenting an Application

```
public class Quux {
    public void useFoo() {
        //assume this call is failing, and you don't know why
        Foo f = new Foo();
    }
    //add this static block to tell where Quux is coming from
    static {
        ClassLoader cl = Quux.class.getClassLoader();
        System.out.println("Delegation for Quux");
        while (cl != null) {
            System.out.println(cl);
            cl = cl.getParent();
        }
        System.out.println("{bootstrap loader}");
    }
}
```

The static block added to the `Quux` class simply logs the entire delegation for `Quux` by recursively calling `getParent()` until the bootstrap loader is reached. If `Quux` came from the classpath, you would see output similar to this:

```
sun.misc.Launcher$AppClassLoader@404536
sun.misc.Launcher$ExtClassLoader@7d8483
{bootstrap loader}
```



Here you can see the three installed class loaders in action. The instance of the nested class `sun.misc.Launcher.AppClassLoader` is the system class loader, and the instance of `sun.misc.Launcher.ExtClassLoader` is the extensions loader. By inserting a block of code like this near the site of a class loading failure, you will be able to verify that the classes that *did* load came from the right place.

2.8.2 Using `-verbose:class`

You can obtain similar logging without writing any code by running the virtual machine with the `-verbose:class` flag for tracing class loading. Listing 2-17 shows a partial example of the output from running the `LoaderDemo` example (see Listing 2-2) with the `-verbose:class` flag.

Listing 2-17 Output from `-verbose:class`

```
[Opened d:\java\jdk1.3\jre\lib\rt.jar]
[Opened d:\java\jdk1.3\jre\lib\i18n.jar]
[Opened d:\java\jdk1.3\jre\lib\sunrsasign.jar]
[Loaded java.lang.Object from d:\java\jdk1.3\jre\lib\rt.jar]
[Loaded java.io.Serializable from d:\java\jdk1.3\jre\lib\rt.jar]
[Loaded java.lang.Comparable from d:\java\jdk1.3\jre\lib\rt.jar]
    {more lines like this }
[Loaded LoaderDemo]
[Loaded sun.misc.URLClassPath$JarLoader from
d:\java\jdk1.3\jre\lib\rt.jar]
    {more lines again }
[Loaded LoadMe]
```

The output shows the order in which classes are loaded, and for the core API classes, it also shows the JAR file they were loaded from. The complete listing of loaded classes from an application run can be very helpful in many debugging scenarios. In addition to helping with class loading problems, it also gives some hints as to what the application was doing. This information can also help you tune the application footprint. Sometimes a single class can pull in a large number of other classes via implicit class loading. The `-verbose:class` output makes this situation painfully clear, and it gives you the opportunity to locate and perhaps discontinue using classes that incur this hidden expense.

2.8.3 Instrumenting the Core API

The `-verbose:class` feature is easy to use, but the format and quantity of output is controlled by the virtual machine. In some situations, you might want to log even more information, such as failure to load classes or the order in which loaders are consulted to load a class. If you control the source code for the class loaders used in an application, you can create a special logging subclass for each class loader. If you do not control the source code, your only recourse may be to modify the core API classes to add custom logging.

The Java SDK provides a nonstandard flag that makes it easy to replace core API classes. To take advantage of this, specify an alternate location that is consulted before the normal bootclasspath via the `-Xbootclasspath/p:` flag. Even if a VM does not support this nonstandard flag, you can still replace core classes by building a version of `rt.jar` that contains the modified classes. The VM flag simply makes experimentation easier by eliminating the need to keep track of multiple custom versions of `rt.jar`.

Listing 2-18 shows a modified version of `URLClassLoader` that logs class loader creation, classes found, and classes not found. The `logConstructor` method logs the creation of all `URLClassLoaders`, including the URLs that the loader will consult and the parent loader. The log output also includes the `this` reference, which you can use to cross-reference with the class loading portion of the log to determine which specific instance loaded a class. The `logConstructor` method also logs the call stack at the time the loader was created by instantiating an `Exception` and then extracting its stack trace (without ever throwing the `Exception`). This is a common trick for logging call stacks.¹⁰

Listing 2-18 Adding Logging to `URLClassLoader`

```
//extract java.net.URLClassLoader from src.jar in your SDK
//directory to a "boot" directory. Insert the following
//methods adding other code as instructed in the comments:
/**
 * add a call to logConstructor after the call to super
```

10. You could also use `Thread.dumpStack()`, which does the same trick internally. I prefer the direct instantiation of an exception because the `printStackTrace` can be redirected to a `PrintStream` other than `out`.

```

    * in each URLClassLoader constructor
    */
private void logConstructor(URL[] urls, ClassLoader parent) {
    if (parent == null) {
        parent = getSystemClassLoader();
    }
    System.out.println("Created URLClassLoader " + this);
    System.out.println("\tparent: " + parent);
    for (int n=0; n<urls.length; n++) {
        System.out.println("\turl: " + urls[n]);
    }
    System.out.println("created at ");
    new Exception().printStackTrace();
    System.out.println();
}

protected Class loadClass(String name, boolean resolve)
    throws ClassNotFoundException
{
    Class cls = null;
    try {
        cls = super.loadClass(name, resolve);
        return cls;
    }
    finally {
        System.out.print("Class " + name);
        if (cls == null) {
            System.out.println(" could not be loaded by " + this);
        } else {
            ClassLoader cl = cls.getClassLoader();
            if (cl == this) {
                System.out.println(" loaded by " + cl);
            } else {
                System.out.println(" requested by " + this +
                    ", loaded by " + cl);
            }
        }
    }
}

```

The `loadClass` method logs whether class loading was successful and the identity of both the requesting loader and the loader that was actually used. (Remember that the delegation model implies that the requesting loader may

delegate instead of doing the loading itself.) If you compile this version of `URLClassLoader` to a boot directory and then prepend that directory to the `bootclasspath`, you will get quite a large output, similar to Listing 2–19.

Listing 2–19 Output from the Logging Version of `URLClassLoader`

```
{Note: this output has been drastically clipped to show only a few
classes being loaded}
>java -Xbootclasspath/p:boot -cp classes LoaderDemo
Created URLClassLoader sun.misc.Launcher$ExtClassLoader@100d7a
  parent: null
  created at {stack trace omitted}
Created URLClassLoader sun.misc.Launcher$AppClassLoader@ac738
  parent: sun.misc.Launcher$ExtClassLoader@100d7a
  url: file:/D:/halloway/JavaCode/v1tests/classes/
  created at {stack trace omitted}
Class LoaderDemo could not be loaded by
  sun.misc.Launcher$ExtClassLoader@100d7a
Class LoaderDemo loaded by
  sun.misc.Launcher$AppClassLoader@ac738
Created URLClassLoader java.net.URLClassLoader@3179c3
  parent: sun.misc.Launcher$AppClassLoader@ac738
  url: file:subdir/
  created at {stack trace omitted}
Class LoadMe could not be loaded by
  sun.misc.Launcher$ExtClassLoader@100d7a
Class LoadMe could not be loaded by
  sun.misc.Launcher$AppClassLoader@ac738
Class LoadMe loaded by java.net.URLClassLoader@3179c3
```

This output demonstrates many of the points made in this chapter. First, the VM creates the extensions class loader, which delegates to the bootstrap class loader. Then, the VM creates the system class loader, which delegates to the extensions class loader. The log output clearly shows that both the system and extensions class loaders are implemented as subclasses of `URLClassLoader`. When `LoaderDemo` creates an instance of `URLClassLoader`, it does not specify a parent, but you can see that it implicitly delegates to the classpath class loader. Finally, you can see that `LoaderDemo` is loaded from the classpath but that `LoadMe` is loaded by the `URLClassLoader` that points to `file:subdir/`.

You will find that replacing `java.net.URLClassLoader` is far more useful for debugging than either using the `-verbose:class` flag or adding ad hoc code to your own classes. Of course, you may not like the specific information generated by the `URLClassLoader` modifications listed above. Good! The entire point of replacing the class is to generate exactly the output you want, so start with the example code and tweak it to meet your needs.

Replacing core classes is a trick for in-house debugging only. You should never *ship* a custom version of a core API class to customers without specifically verifying that what you are doing does not violate the license agreement. For more information see the LICENSE file that is in the root directory of your SDK installation.

You might find similar approaches useful in logging Swing, or Remote Method Invocation (RMI), or just about any Java technology. In theory, any core API class is fair game. In practice, you need to be very careful not to break anything. If you are not exactly sure what you are doing, you may introduce catastrophic bugs. Even simply adding logging code could cause concurrency problems and deadlocks. Modifying the core APIs is like recompiling your operating system. It can be entertaining and highly educational, but you should not do it in a production environment.

2.9 Inversion and the Context Class Loader

Thanks to the delegation model, one class can reference another without both classes having to come from the same class loader. In particular, a class `A` can reference any class `B` that is visible to `A`'s class loader's delegation. More concretely, an application class `Main` can refer to `java.lang.String` even though `Main` comes from the classpath class loader and `String` comes from the bootstrap loader. However, this relationship is *not* bidirectional. The `String` class cannot refer to the application `Main` class, because the classpath class loader is not part of the bootstrap loader's delegation. The problem is one of *inversion*—a class from a parent loader cannot reference a class from a child loader. Some legal permutations that do not cause inversion are listed in Table 2-2.



Table 2-2 Legal and Illegal References across Class Loaders

Relationship between Classes A and B	Legal Class Loader Relationships
No relationship	Any
A has field/variable of type B	CL _A equals/descends from CL _B
A extends B	CL _A equals/descends from CL _B
A has field/variable of type B and B has field/variable of type A	CL _A <i>must</i> equal CL _B

In most situations, inversion problems are easily avoided. Simply put base classes, superinterfaces, and referenced classes *at least as high* in the class loader hierarchy as the classes that reference them. This is trivially accomplished by placing all of your application code on the classpath. For hot deployment scenarios, install interfaces on the classpath and load implementations with `URLClassLoaders` that are children of the classpath loader.

Sometimes code that is very high in the class loader hierarchy will need to access code that is an arbitrary distance lower in the class hierarchy. Consider the hypothetical `StuffedAnimal` API (a.k.a. SAPI) in Listing 2-20. The API consists of two classes, `StuffedAnimal` and `StuffedAnimalFactory`. The `StuffedAnimal` interface defines the contract between the client and the implementation, and the `StuffedAnimalFactory` provides a configurable way for clients to request a `StuffedAnimal` implementation.

Listing 2-20 The `StuffedAnimal` API (SAPI)

```
public interface StuffedAnimal {
    public void snuggle();
    public void sleep();
    public void getMisplaced();
}

public class StuffedAnimalFactory {
    public static StuffedAnimal newAnimal() {
        String name = System.getProperty("stuffed.animal");
        if (name == null) {
            throw new Error("stuffed.animal not specified");
        }
    }
}
```




```

        //see StuffedAnimalFactory3 for better approach
    try {
        Class cls = Class.forName(name);
        return (StuffedAnimal) cls.newInstance();
    } catch (Exception e) {
        e.printStackTrace();
        throw new Error("Unable to create " + name);
    }
}
}

```

Listing 2-21 A StuffedAnimal Provider

```

public class TeddyBear implements StuffedAnimal {
    public void snuggle() {
        System.out.println("I love you");
    }
    public void sleep() {
        System.out.println("ZZZ");
    }
    public void getMisplaced() {
        throw new Error("child very unhappy");
    }
}

```

Listing 2-22 A StuffedAnimal Client

```

public class StuffedAnimalClient {
    public static void main(String [] args) {
        StuffedAnimal sa = StuffedAnimalFactory.newAnimal();
        sa.snuggle();
        sa.sleep();
    }
}

```

Imagine that `StuffedAnimals` become so popular that almost all Java applications want to use them. Because the API is stable and widely used, you can make the SAPI available to all. Simply add `StuffedAnimal` and `StuffedAnimalFactory` to a `SAPI.jar` file in the extensions directory. Now, assume that the `StuffedAnimalClient` wants to define and use a particular `TeddyBear` provider implementation as shown in Listings 2-21 and 2-22. Assuming that

the client and provider code are in the classes subdirectory, the `java` command line would look like this:

```
java -cp classes/ -Dstuffed.animal=TeddyBear \
  StuffedAnimalClient
```

This usage has several advantages:

1. The client does not have to worry about the location of the API code because it is picked up automatically from the extensions directory.
2. The API definition, client, and server are all free from worrying about explicit class loading—none of the code even mentions a class loader.
3. The client is the deployer. Clients can select different implementations without changing a line of code by setting the `stuffed.animal` property.

In fact, the only problem with this example is that it simply doesn't work. The API is loaded as an extension, but the server code (Teddy Bear) is on the classpath. The API's reference to the server code is a class loader inversion and the factory's call to `Class.forName` is unable to see the `TeddyBear` class.

There are a couple of workarounds to this problem that you should avoid. Obviously, you could dodge the issue by installing *all* the classes under the same class loader, either the extensions or the system loader. Application architectures that take this approach often end up installing classes in several different places just to make sure things load. Never do this. This haphazard approach defeats the purpose of having a *hierarchy* of different loaders, and it almost guarantees deployment problems.

A slightly better idea would be to add a `ClassLoader` argument where necessary in the API, as shown in Listing 2–23. This version of SAPI allows the client to specify a class loader on each call to the factory, and it uses the three-argument version of `Class.forName` to reach the correct class loader. This explicit approach will work, but it is tedious to ask the client to pass in a class loader every time. This strategy becomes even more tedious if the class loader must be passed through dozens of intermediate methods before it needs to be used.

Listing 2–23 SAPI with Explicit Class Loading

```
public class StuffedAnimalFactory2 {
    public static StuffedAnimal newAnimal(ClassLoader cl) {
```

```

        String name = System.getProperty("stuffed.animal");
        if (name == null) {
            throw new Error("stuffed.animal not specified");
        }
        //see StuffedAnimalFactory3 for best approach
        try {
            Class cls = Class.forName(name, true, cl);
            return (StuffedAnimal) cls.newInstance();
        } catch (Exception e) {
            e.printStackTrace();
            throw new Error("Unable to create " + name);
        }
    }
}

public class StuffedAnimalClient2 {
    public static void main(String [] args) {
        Class cls = StuffedAnimalClient2.class;
        ClassLoader cl = cls.getClassLoader();
        StuffedAnimal sa = StuffedAnimalFactory2.newAnimal(cl);
        sa.snuggle();
        sa.sleep();
    }
}

```

An easier approach than using an explicit argument is to define a *context class loader* to use in potential inversion situations. A context loader is not passed as an explicit parameter; instead, it is available at any time through a special API.

The thread context class loader is designed precisely for this purpose. New as of SDK version 1.2, the context loader is implemented by a pair of methods on the thread class, shown in Listing 2–24. Functionally, the context class loader methods are simply a wrapper around a single object reference kept in thread local storage. When you write a provider API, such as the `StuffedAnimal` API, you should use the thread-specific context loader on each thread, thereby avoiding the need to pollute your API with extra class loader parameters. This is particularly important if your code is going to execute in a container environment, such as a J2EE implementation, in which the instantiation of class loaders is typically controlled by the J2EE container vendor, not the application author.

Listing 2-24 The Context Class Loader API

```

package java.lang;

public class Thread implements Runnable {
    public void setContextClassLoader(ClassLoader cl);
    public ClassLoader getContextClassLoader();
    //rest of class omitted for clarity
}

```

Listing 2-25 shows a version of SAPI that uses the thread context class loader. In this listing, the context loader is used only once, so the code is actually a little longer than the explicit version shown in Listing 2-23. In a larger application, setting the context loader in a centralized location would greatly reduce the amount of class loader-related code.

Listing 2-25 SAPI Using the Thread Context Class Loader

```

public class StuffedAnimalFactory3 {
    public static StuffedAnimal newAnimal() {
        String name = System.getProperty("stuffed.animal");
        if (name == null) {
            throw new Error("stuffed.animal not specified");
        }
        ClassLoader cl = Thread.currentThread()
            .getContextClassLoader();

        try {
            Class cls = Class.forName(name, true, cl);
            return (StuffedAnimal) cls.newInstance();
        } catch (Exception e) {
            e.printStackTrace();
            throw new Error("Unable to create " + name);
        }
    }
}

public class StuffedAnimalClient3 {
    public static void main(String [] args) {
        Class cls = StuffedAnimalClient2.class;
        ClassLoader cl = cls.getClassLoader();
        Thread.currentThread().setContextClassLoader(cl);
        StuffedAnimal sa = StuffedAnimalFactory3.newAnimal();
        sa.snuggle();
        sa.sleep();
    }
}

```

In the versions of the `StuffedAnimalClient` that set a class loader, I chose the class loader using the code

```
ClassLoader cl = StuffedAnimalClient.class.getClassLoader();
```

instead of the arguably simpler

```
ClassLoader cl = ClassLoader.getSystemClassLoader();
```

These two formulations mean entirely different things. The former says “Give me the class loader that loaded `StuffedAnimal`,” while the latter says “Give me the class loader that loads from the classpath.” It is a coincidence that these evaluate to the same loader in this example. If you used the latter formulation and later switched to using hot deployment, someone besides the system loader might load the `StuffedAnimalClient` and `TeddyBear`, and you would have another inversion problem. Do not make assumptions about what class loader will load your class. Always query for your class’s actual loader at runtime with `YourClass.class.getClassLoader`.

2.10 Onward

Class loaders enable flexible, dynamic deployment of Java applications. Classes can be loaded from a variety of different sources, which are selected at runtime. Class loaders provide controlled sharing of code. The delegation model of class loading allows some classes to be shared widely, while other classes are kept local to class loaders far down the delegation hierarchy.

Class loading is often felt but not seen. Most Java code takes no explicit account of class loading, allowing implicit class loading to deal with many common loading scenarios. The core API provides a set of built-in class loaders: the bootstrap, extensions, and system class loaders. These loaders provide for different levels of security and different policies of sharing between and within applications. The combination of implicit class loading and the configuration options for the standard class loaders meets many class loading needs without requiring any explicit class loading code.

You can gain additional flexibility by instantiating your own class loaders to explicitly load classes. The `java.net.URLClassLoader` class can load

classes from any supported URL protocol. One use of `URLClassLoader` is hot deployment, whereby old and new versions of a class can coexist in the same process. Debugging class loading can be tricky, but with the `-verbose:class` flag and customized implementations of `URLClassLoader`, you can quickly eliminate most problems.

The context class loader is a thread local class loader reference. Use the context loader as an out-of-band mechanism to communicate which class loader needs to be used when related components are loaded by different class loaders.

2.11 Resources

If you want to learn more about the topics covered in this chapter, the following references may prove useful. [New00] covers class loading in detail, with an emphasis on how to use class loading to support different deployment strategies. [JavaGeeks] includes several free white papers related to class loading. “Understanding `Class.forName()`” has a good explanation of the thread context class loader, and “Using the `BootClasspath`” discusses replacing the core classes. [Ven99] explains the class loading process, with an emphasis on the actual structure of class files and how the virtual machine loads and verifies classes.