Class Loaders

Intent

Class loaders let Java dynamically load classes (and linking) on demand, at runtime. Applications may use predefined class loaders, or, if they must rely on specific behavior, may construct their specific class loaders.

Note. Class loaders are tightly coupled with security issues. Security, however, will not be discussed here, and is deferred to the course “Advanced Java Technologies”.

Load-Time Dynamic Loading and Linking

Consider the following program:

```
public class Hello {
   public static void main(String[] args) {
      System.out.println("Hello, world!");
   }
}
```

When the JVM loads class `Hello`, it notices that this class uses classes `String` and `System`, and that it extends class `Object`. If the JVM has not yet loaded one of those classes by this point, the JVM must run off and load these ones before it can finish loading `Hello`.

Run-Time Dynamic Loading and Linking

Run-time dynamic loading and linking loads a class file and resolves its names until run-time, when the name of the class has been determined:

```
public class X implements Runnable {
   public void run() {
      System.out.println("Class X.");
   }
}

public class Y implements Runnable {
   public void run() {
      System.out.println("Class Y.");
   }
}
```

```
public class LoadAndExecute {
   public static void main(String[] args) { // w/o arg checking
      Class cl = Class.forName(args[0]);
      Object o = cl.newInstance();
      Runnable r = (Runnable) o;// assuming a Runnable
      r.run();
}  }
```

Remarks:

- Variable `cl` holds a class object (an instance of class `Class`).
- `Class.forName(String)` loads and links a class file at the point of its use, given the fully qualified class name.
- Method `newInstance` creates an instance of the class the corresponding class object stands for. *The class must have a public null-arg constructor*.
- If you know the type of the object (here `Runnable`) then you can downcast to the known type and apply corresponding methods on it.
- If you don't know the type then you can use *reflection* on the class object to obtain more information about the supported constructors, methods, etc.

Class Loaders

Java uses *class loaders* for the loading and linking classes. Class loaders are ordinary Java objects of subclasses of class `java.lang.ClassLoader`, except the bootstrap class loader. By default, Java application typically has the following class loaders:

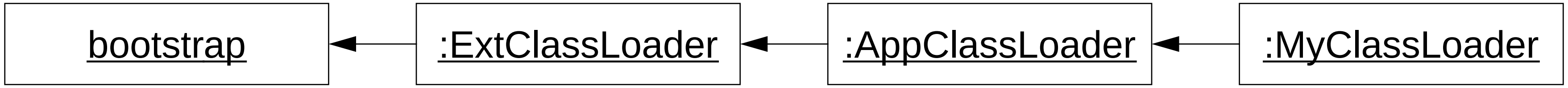
- **Bootstrap class loader:** *native code*, used for loading the classes belonging to the Java system. That is, packages such as `java.*`, `javax.*`, `org.omg.*`, etc.
- **The class loader for the extensions:** e.g. class `sun.misc.Launcher$-ExtClassLoader`, used for loading classes or JAR files added to JRE's `lib/ext` directory. Some JVMs combine this one with the one above.
- **The application class loader:** e.g. class `sun.misc.Launcher$-AppClassLoader`, used for loading the classes for the application. The classes are searched within the file system relative to the definition of the `CLASSPATH` variable.

In addition, an application may install one or more custom class loaders, e.g. `MyClassLoader`.

An applet uses an applet class loader to load applet code relative to a URL.

The Java 2 Parent Class Loader Delegation Model

By convention, when a class loader is asked to load a class, it *first* asks the class loader that loaded *it* (!) to load that class. I.e. the following delegation model exists:



When Not Using the Parent Delegation Model

In some circumstances, you do not want to use the default delegation model. For example, the Apache/Tomcat servlet engine uses class loaders that try to load the requested class directly (and not delegating it to the parent class loader).

Be careful when writing class loaders not using the parent class loader delegation model.

The java.lang.ClassLoader API

The public API:

- The public API:
  `getParent` returns the parent class loader. `getResource`, `getResourceAsStream`, `getResources` return a single resource as an input stream, or an enumeration of all resources. `getSystemClassLoader`, `getSystemResource`, `getSystemResourceAsStream`, `getSystemResources` return the system versions of above. Finally, `loadClass` is used to load and link Java classes. Given a class loader instance, you load a class for example (w/o exception handling):

  ```
  ClassLoader cl = ...;
  Class c = cl.loadClass("example.Hello");
  ```

  By default, `loadClass` performs the following steps:
  - calls `findLoadedClass()` to check if the class has already been loaded
  - calls `loadClass()` on the parent (if any)
  - calls `findClass(String)` to find the class.

  Notice that `loadClass` is a template method.

- The factory method API:
  `findClass`, `findLibrary`, `findResource`, and `findResources` are factory methods. These will have to be overridden by subclasses.

- The base API:
  `defineClass` provides a convenience method for translating a bytecode array to a class object, performing all the loading, linking, and initialization. Other methods are here, too, plus another `loadClass` method for internal use.

Important note:

- **A class always remembers the class loader that loaded it, and any classes referenced by that class that haven't been loaded will be loaded (if possibly) by that class loader (however, by respecting the delegation model).**

Java Name Spaces, Protection Domains, and Code Sources

This is an advanced topic. Will be discussed in course “Advanced Java Technologies”.

Java’s Built-In Class Loaders

`java.security.SecureClassLoader`:

Intended to be the base class for all custom class loaders. When writing your own class loader, either extend from class `URLClassLoader` (see next) or from this class.

`java.net.URLClassLoader`:

Flexible class loader that uses URLs for loading classes. Since there are `file` URLs, `http` URLs, and `ftp` URLs, this class loader can load classes from the file system, from HTTP servers, and from FTP servers.

To load a class from a file:

```
// w/o exception handling
URL[] urls = { new File("some_path").toURL() };
ClassLoader cl = new URLCassLoader(urls);
Object o = cl.loadClass("example.Hello").newInstance();
```

`URLClassLoader` deals also with ZIP and JAR archives:

```
// w/o exception handling
URL[] urls = { new File("library.jar").toURL() };
ClassLoader cl = new URLCassLoader(urls);
Object o = cl.loadClass("example.Hello").newInstance();
```

To load a class from a HTTP server:

```
// w/o exception handling
URL[] urls = {
   new URL("http", "www.hta-bi.bfh.ch", "/~due/");
};
ClassLoader cl = new URLCassLoader(urls);
Object o = cl.loadClass("Hello").newInstance();
```

To load a class from an FTP server:

```
// w/o exception handling
URL[] urls = {
   new URL("ftp", "username:passwd@some_ftp_server:", "/");
};
ClassLoader cl = new URLCassLoader(urls);
Object o = cl.loadClass("Hello").newInstance();
```

`sun.applet.AppletClassLoader`:

Extends `URLClassLoader`, used for loading applet classes.

`java.rmi.server.RMIClassLoader`:

Actually a wrapper class for an instance of a subclass of `URLClassLoader`.
Used to deal with RMI's class annotations.

`sun.misc.Launcher$ExtClassLoader`,
`sun.misc.Launcher$AppClassLoader`, and bootstrap loader:

Mentioned above.

Custom Class Loaders

A Java 2 class loader needs to override one of three `find` methods:

- `Class findClass(String)`: This method is expected to obtain the byte-code from a fully qualified class name. Called from the public `loadClass` method. Once the bytecode is obtained, the method must call one of the `defineClass()` methods.

- `URL findResource(String name)`: This method is expected to obtain the bytes from a given arbitrary name. Called from the public `getResource` method. If you implement this method, be sure to implement the `findResources` method, too.

- `String findLibrary(String)`: Returns the absolute path name of a native library.

In the following, we'll concentrate on the loading of classes only,

Custom Class Loader Skeleton

Here is a skeleton for a custom class loader (security issues omitted):

```
import java.security.SecureClassLoader;
public class CustomClassLoader extends SecureClassLoader {
   ...
}
```

By convention, the null-argument constructor should ensure that the class loader associated by the calling thread becomes the parent:

```
public CustomClassLoader() {
   super(Thread.currentThread().getContextClassLoader());
}
```

If you want to have more control over the class loader hierarchy, you could provide another constructor, too:

```
public CustomClassLoader(ClassLoader parent) {
   super(parent);
}
```

Finally, you provide your custom `findClass` method. This method should use one of the three forms of the `defineClass` methods defined in the `SecureClassLoader` class or its ancestors. Here, the simplest on is used:

```
protected Class findClass(String name)
   throws ClassNotFoundException
{
   ...
   byte[] bytecode = retrieveClass(className);
   return defineClass(name, bytecode, 0, bytecode.length);
}

private byte[] retrieveClass(String classname)
   // perhaps add "throws"-clause here
{
   // Get byte array of the class from some place.
}
```

Add methods for obtaining resources if necessary. No code examples shown.