Appendix 2. Installation using Java Web Start

Java Web Start (JWS) is an installer for Web-based Java applications (see http://java.sun.com/products/javawebstart/). Typically, the user points a browser at a page containing a link to a deployment file; the retrieval of that file triggers the execution of JWS on the client, which takes over from the browser. (This assumes that JWS is already present of the machine.)

JWS uses the information in the deployment file to download the various JAR files making up the application, together with installation icons, a splash screen, and other details. The application is stored in a local cache, and executed inside a JVM sandbox. Subsequent executions of the application utilize the cached copy, unless the original has been modified, in which case the changed JARs are downloaded again.

This appendix shows how the BugRunner and Checkers3D applications of chapters 6 and 8 can be deployed with JWS. BugRunner is the beginnings of a 2D arcade-style game, but uses the Java 3D timer. Checkers3D is a basic Java 3D application that displays a blue sphere floating above a checkboard surface.

Both applications require native libraries, and we consider how JWS installers suitable for Windows can be created. However, JWS is a cross-platform tool, so we'll briefly examine the issues in making BugRunner and Checkers3D work on the Linux and Mac OSes.

Deployment links are usually placed in a *JWS portal page*, a page using JavaScript and VBScript to detect whether the intended client platform actually possesses JWS. If JWS isn't found, then it needs to be downloaded before the application installation can begin. Section 7 describes a portal page for accessing BugRunner and Checkers3D, located at http://fivedots.coe.psu.ac.th/~ad/jws/.

JWS uses digital signing and certificates to secure applications, and section 8 looks at how to use third-party certificates.

1. JWS Benefits

JWS works the same way across multiple platforms, unlike the downloading and execution of applets which is plagued by irritating variations between browsers. The headaches caused by the browsers' non-standard programming frameworks (e.g. differences in JavaScript, HTML) are one reason for the decline in popularity of complex applets, and the growth in thin-clients linked to J2EE-built servers.

Client-side caching avoids an essential problem familiar from applets – the need to download them every time they're used. That's a waste of bandwidth if the applet hasn't changed, and further discourages the development of large applets.

JWS suffers from network overheads during the first download, but the copy of the application in the local cache is used after that (until changes are detected in the original).

The cached copy means that network failure will not prohibit the application from executing, unless it requires network access for specific tasks. By comparison, an applet is out of reach when the network is down.

JWS only retrieves Java software packaged as JARs. However, the JARs may contain native libraries for different OSes and platforms, a feature we'll need in order to utilize the Java 3D libraries. Our examples concentrate on Java applications retrieved by JWS, but applets can be downloaded as well.

JWS prevents hacker attacks by executing the installed code inside a sandbox, stopping anti-social behaviour such as hard disk wiping or spamming from your machine. Sometimes the security restrictions can be too harsh, and JWS offers two ways of relaxing them. The JNLP (Java Network Launching Protocol) API supports controlled ways to interact with the OS, such as reading and writing files, and accessing the clipboard. It's also possible to digitally sign software, permitting its security level to be reduced. We'll investigate this latter approach, which is mandatory if a program uses native libraries.

Since the downloaded application is running free of the browser, there's complete freedom over the kinds of user interaction and GUI elements that can be employed.

2. JWS Downsides

There are a few downsides to using JWS. One is the need to have JWS present on the client machine before the application is downloaded. For existing Java users, this isn't a problem, since JWS is installed as part of J2SE or JRE. But what about games players who don't have any desire to join the Java faithful?

The answer is somewhat messy, since it requires the Web page containing the deployment link to detect whether the client machine has JWS installed. If it hasn't, then a JRE must be downloaded before the application. Unfortunately, the non-standard problems with browsers complicate this detection work. I'll discuss *JWS portal pages* in section 7.

There have been several versions of JWS: JWS 1.0 shipped with JRE 1.4, JWS 1.2 was included with JRE 1.4.1, and JWS 1.4.2 arrived with JRE 1.4.2. The beta version of J2SE/JRE 1.5 comes with a considerably revamped JWS 1.5, which replaces the application manager with a control panel and a cache viewer. The earlier versions of JWS (before 1.4.2) have problems correctly setting up proxy network settings, and placing the cache in an accessible location on multi-user machines.

JWS cannot be (legally) modified or reconfigured prior to its installation. For example, we can't distribute a version of JWS that never places an application icon in the menu list, or always pops up a console window to display output. The location of the cache cannot be preset, and there is no way to setup an uninstallation menu item. Many of these things can be changed, but only after JWS is present on the client machine, which means they must be carried out by the *user*. These tasks may be beyond the ability of novices.

To be fair, the deployment file does allow some installation elements to be configured, including what is displayed during the retrieval process.

The automatic updating of an application requires that JWS checks the home server for changes every time the program is run, causing a (small) delay. Normally, an entire JAR will be downloaded, even if only a single line of it has changed. The solution is the jardiff, which specifies the changes necessary to update a JAR to its current version. jardiffs are much smaller than JARs, since they only need to store modifications. However, jardiffs require specialized server settings before they can be utilized.

JWS deployment files need the JNLP MIME type to be set up in their host server. For example, the mime.types file in Apache must include the line:

```
application/x-java-jnlp-file   JNLP
```

The problem is that the application developer may not have access to the server to do this.

Although JWS is aimed at Web-based downloads, it is possible to a supply a link to a local deployment file (using the file://<path> notation). However, the file must contain a reference to its current location, which will often be unknown to the developer at build time. For instance, at installation time, the file may be stored on a CD created by a third-party, and be mounted on a drive with a name that could be almost anything.

There are add-on JWS tools for building CD installers, including Clio (http://www.vamphq.com/clio.html) and Jess (http://www.vamphq.com/jess.html). Clio adds a built-in Web server to the CD, while Jess writes the application directly to the JWS cache.

3. The JNLP Deployment File

A deployment file is written in XML, and has a .jnlp extension. The file format is defined by the JNLP and API specification (JSR-56), available from http://java.sun.com/products/javawebstart/download-spec.html. A subset is described in the developers guide in the J2SE documentation (see <JAVA_HOME>/docs/guide/jws/developersguide/contents.html).

Most .jnlp files have the following structure:

```
<?xml version="1.0" encoding="UTF-8"?>
<jnlp spec="1.0+"
  codebase="http://www.foo.com/loc/"
  href="appl.jnlp" >

  <information> ... </information>

  <security> ... </security>

  <resources> ... </resources>

  <application-desc> ... </application-desc>
</jnlp>
```


The `codebase` attribute gives the base URL where this file and the other application components are stored. A file:// location may be used instead, if the software is to be loaded locally. The `href` attribute gives the URL reference for this JNLP file, which can be relative to the codebase (as in the example), or be an absolute address.

The `information` tag contains textual information about the application, utilized by JWS at retrieval and execution time. References to icons and a splash screen image are also placed in `<information>`.

The `security` tag is optional. If present, it defines the level of increased access given to the application. Two values are possible: `<all-permissions/>` or the slightly less powerful `<j2ee-application-client-permissions />`. They both require that the application's JARs be digitally signed.

The `resources` tag lists the JARs comprising the program.

The `application-desc` tag shows how the program is to be called, along with optional input arguments.

4. Steps in Developing a JWS Application

The following six steps outline the development process for a JWS installer. In Sections 5 and 6, they'll be explained in greater detail, as the BugRunner and Checkers3D applications are converted into JWS applications.

1. Write and test the application on a stand-alone machine, packaging it (and all its resources) as JAR files.
2. Modify the application code to make it suitable for deployment. The necessary changes will be minimal unless native libraries are used. In that case, each library must be wrapped up inside a JAR, and loadLibrary() calls to the application's top-level.
3. Create a new public/private keypair for signing the application and its component JARs. At this stage, a third-party certificate may be obtained from a certificate authority (CA). We'll delay talking about this until section 8.
4. Sign everything with the private key: the application JAR, the extension JARs, and any native library JARs.
5. Create a deployment file (a .jnlp file) using a file:// codebase so that the installation can be tested locally. This stage will require the creation of application icons and a splash screen image, used by JWS.
6. Change the deployment file to use the host server's URL, and place everything on that server. The deployment file will usually be accessed through a JWS portal page.

   Test the installer on a variety of client platforms, OSes, and browsers. Some of the test clients should not possess JWS.

5. A JWS Installer for BugRunner

Step 1. Write the Application.

The installer version of the application shouldn't rely on non-standard extensions or native libraries being present on the client machine. However, BugRunner uses the Java 3D timer, which is part of the Java 3D extension. The OpenGL Windows version of Java 3D is implemented across seven files:

- JAR files: j3daudio.jar, j3dcore.jar, j3dutils.jar, and vecmath.jar
- Native libraries: J3D.dll, j3daudio.dll, J3DUtils.dll

The native libraries will vary across different platforms, and the JAR versions may vary slightly as well.

Only j3dutils.jar and J3DUtils.dll are needed for the timer functionality, as explained in Appendix 1. They should be placed in the BugRunner directory, to be locally accessible to the application. Java 3D should not be installed.

Figure 1 shows the BugRunner directory prior to compilation. It contains all the Java files (unchanged from chapter 6), and j3dutils.jar and J3DUtils.dll. The batch files are optional, but reduce the tedium of typing long command lines.

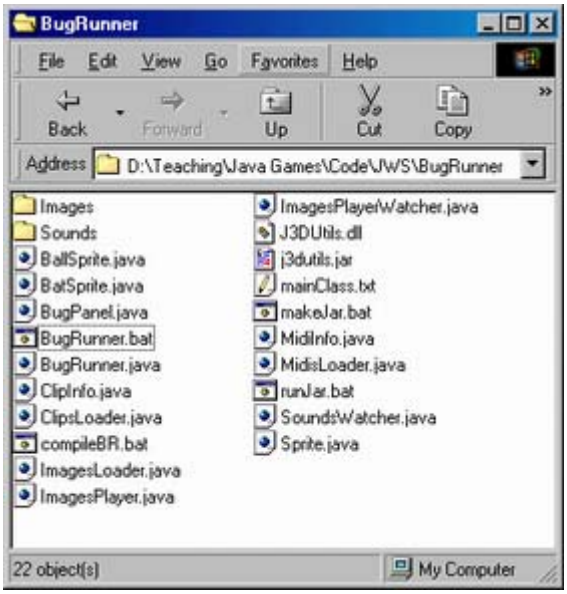

Figure 1. The BugRunner Directory.

Since Java 3D isn't installed in a standard location checked by javac and java, the calls to the compiler and JVM must include additional classpath information.
The compileBR.bat batch file contains:

```
javac -classpath "%CLASSPATH%;j3dutils.jar" *.java
```

The BugRunner.bat batch file has:

```
java -cp "%CLASSPATH%;j3dutils.jar" BugRunner
```

There's no need to mention J3DUtils.dll, which will be found by the JAR so long as it's in the local directory.

Once the program has been fully debugged, it should be packaged as a JAR. The BugRunner application consists of various classes, and the subdirectories Images/ and Sounds/. These should be thrown together into a single BugRunner.jar file. The makeJar.bat batch file contains the line:

```
jar cvmf mainClass.txt BugRunner.jar *.class Images Sounds
```

The manifest details in mainClass.txt are:

```
Main-Class: BugRunner
Class-Path: j3dutils.jar
```

The manifest specifies the class location of main(), and adds j3dutils.jar to the classpath used when BugRunner.jar is executed. We assume that it's in the same directory as BugRunner.jar.

The DLLs (only J3DUtils.dll in this case) are not added to BugRunner.jar.

The application now consists of three files: BugRunner.jar, j3dutils.jar, and J3DUtils.dll. These should be moved to a different directory on a different machine and tested again. Double clicking on BugRunner.jar should start it running. Alternatively, type:

```
java –jar BugRunner.jar
```

Step 2. Modify the Application for Deployment

Since the native library J3DUtils.dll is utilized by BugRunner, there are two tasks to be carried out. The DLL must be placed inside its own JAR:

```
jar cvf J3DUtilsDLL.jar J3DUtils.dll
```

There's no need for additional manifest information.

Also, the main() method of BugRunner, in BugRunner.java, must be modified to call loadLibrary() for each DLL:

```
public static void main(String args[])
{
  // DLL used by Java 3D J3DTimer extension
  String os = System.getProperty("os.name");
  if (os.startsWith("Windows")) {
    System.out.println("Loading '" + os + "' native libraries...");
    System.out.print("  J3DUtils.dll... ");
    System.loadLibrary("J3DUtils");  // drop ".dll"
    System.out.println("OK");
  }
  else {
    System.out.println("Sorry, OS '" + os + "' not supported.");
    System.exit(1);
  }

  long period = (long) 1000.0/DEFAULT_FPS;
  new BugRunner(period*1000000L);    // ms --> nanosecs
}
```


If several libraries are loaded, the load order will matter if there are dependencies between them.

The checking of the os.name property string gives the program a chance to report an error if the application is started by an OS that doesn't support the library. This coding style also allows the application to load different libraries depending on the OS name. For instance:

```
String os = System.getProperty("os.name");
System.out.println("Loading " + os + " native libraries...");
if (os.startsWith("Windows")) {
  System.loadLibrary("J3DUtils");  // drop ".dll"
       :
}
else if (os.startsWith("Linux")) {
  System.loadLibrary("J3DUtils"); // drop "lib" prefix & ".so"
       :
}
else if (os.startsWith("Mac")) {
  System.loadLibrary("J3DUtils"); // drop ".jnilib"
       :
}
else {
  System.out.println("Sorry, OS '" + os + "' not supported.");
  System.exit(1);
}
```

A longer example of this kind can be found in "Marc's Web Start Kamasutra", a JWS and JNLP forum thread at http://forum.java.sun.com/thread.jsp?forum=38&thread=166873.

A lengthy list of the possible os.name values is presented in http://www.vamphq.com/os.html.

The changes to BugRunner.java mean it must be recompiled and re-JARed:

```
javac -classpath "%CLASSPATH%;j3dutils.jar" *.java
jar cvmf mainClass.txt BugRunner.jar *.class Images Sounds
```

Step 3. Create a Public/Private Keypair for Signing the Application

The digital signing of a JAR requires two of Java's security tools: keytool and jarsigner. They are described in the security tools section of the J2SE documentation in <JAVA HOME>/docs/tooldocs/tools.html#security.

keytool generates and manages keypairs collected together in a keystore. Each keypair is made up of a public key and private key, and an associated public-key certificate. A greatly simplified diagram showing a typical keypair is shown in Figure 2.

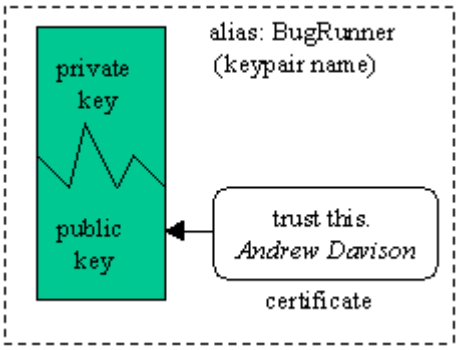

Figure 2. The Elements of a Keypair.

The two keys can be used to encrypt documents. Something encrypted with a private key can only be decrypted with the corresponding public key; if the encryption uses the public key, then only the private key can unlock it.

The intention is that the public key is widely distributed, but the user keeps his private key secret. There is no way of determining the private key from examining the public key.

A message sent to the user can be encrypted with the public key, so that only he/she can read it by applying their private key.

A message from the user to another person can be encrypted with the private key. The fact that the user's public key can decrypt the message means that it must have come from that user. The private key is being used as a *digital signature*.

One of the problems with the public/private keys approach is how to safely distribute a public key to other people. For instance, if I receive an e-mail from "Stan Lee" giving me his public key, how do I know that it really is from the famous Atlas/Timely editor. This is where the public-key certificate comes into play.

A certificate is a digitally signed statement from a third party, perhaps my respected friend "Alan Moore", that this really is "Stan Lee"s public key. Of course, the question of authenticity still applies, but now to the "Alan Moore" signature, which can be combated by signing it with the certificate of yet another person. This process leads to a chain of certificates, ending with a certificate which can be proved genuine in some non-forgeable way (for example, by visiting the person and asking them).

Whenever keytool generates a new keypair it adds a self-signed certificate to the public key. In effect, all my public keys contain certificates signed by me saying they're genuine. This is pretty useless in a real situation, but it's sufficient for our demos. We'll see that JWS issues a dire warning when it sees a self-signed certificate, but will let it pass if the client gives the okay. We discuss how to obtain better certificates in section 8.

A new keypair is generated in the keystore called MyKeyStore by typing:

```
keytool -genkey -keystore MyKeyStore -alias BugRunner
```


The user is prompted for the keystore password, a lengthy list of personal information, and a password for the new keypair (see Figure 3). The keypair's alias (name) is BugRunner in this example, although any name could be used.

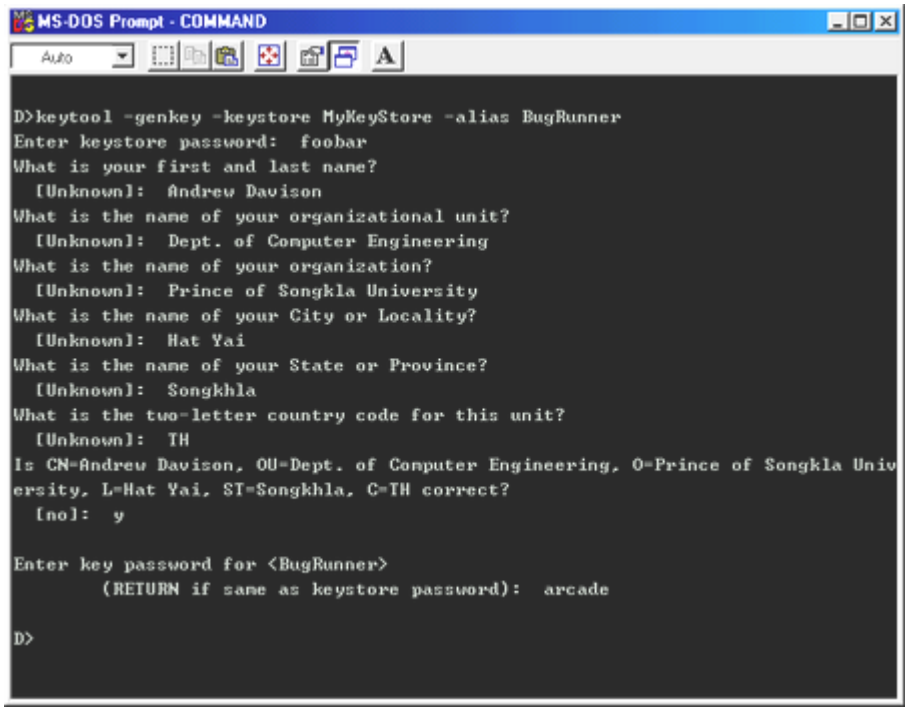

Figure 3. Generate a New Keypair.

Better passwords should be thought up than those used in my examples; a good password uses letters, numbers, and punctuation symbols, and should be at least eight characters long.

The keystore's contents can be examined:

```
keytool -list -keystore MyKeyStore
```

Step 4. Sign Everything with the Private Key

We are now ready to use the jarsigner tool to start signing the JARs in the BugRunner application. Figure 4 presents a simple diagram of what jarsigner does to a JAR.

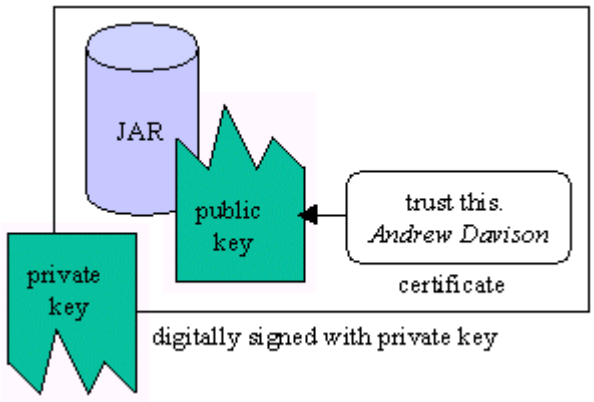

Figure 4. A JAR File Signed with jarsigner.

The jarsigner digitally signs a JAR with a private key. A digital signature has many useful characteristics:

- Its authenticity can be checked by seeing if it 'matches' the public key stored with the JAR. This relies on the public key being trusted, which depends on the certificates attached to it.
- The digital signature cannot be forged, since the private key is only known to the sender of the JAR.
- Unlike a real signature, the digital signature is partly derived from the data it is attached to (i.e. the JAR file). This means that it can't be removed from its original JAR, stuck on a different one, and still be authenticated successfully.

The actual mechanics of creating a signed JAR are very simple:

```
jarsigner -keystore MyKeyStore foo.jar BugRunner
```

This signs foo.jar using the BugRunner keypair stored in MyKeyStore. jarsigner will prompt the user for the keystore and keypair passwords.

A variant of this is to create a new signed JAR file rather than modify the existing one:

```
jarsigner -keystore MyKeyStore -signedjar foo_signed.jar
                                          foo.jar BugRunner
```

This leaves foo.jar unchanged, creating a signed version called foo_signed.jar. The name of the new JAR can be anything.

For the BugRunner application, there are three JARs: BugRunner.jar, j3dutils.jar, and J3DUtilsDLL.jar. The latter two are signed like so:

```
jarsigner -keystore MyKeyStore -signedjar j3dutils_signed.jar
                                          j3dutils.jar  BugRunner

jarsigner -keystore MyKeyStore J3DUtilsDLL.jar BugRunner
```

The creation of a new JAR for the signed version of j3dutils.jar is to avoid any confusion with the original JAR created by Sun.

This process means yet another reJARing of BugRunner, since the manifest information in mainClass.txt must be changed to:

```
Main-Class: BugRunner
Class-Path: j3dutils_signed.jar
```

The jar command in makeJar.bat is unchanged:

```
jar cvmf mainClass.txt BugRunner.jar *.class Images Sounds
```

After BugRunner.jar is regenerated, then it is signed:

```
jarsigner -keystore MyKeyStore BugRunner.jar BugRunner
```


Step 5. Create a Deployment File

The deployment file for the BugRunner application, BugRunner.jnlp, is:

```
<?xml version="1.0" encoding="utf-8"?>
<jnlp spec="1.0+"
  <!-- codebase="http://fivedots.coe.psu.ac.th/~ad/jws/BugRunner/"-->
  codebase="file:///D:/Teaching/Java Games/Code/JWS/BugRunner/"
  href="BugRunner.jnlp"
  >

  <information>
    <title>BugRunner</title>
    <vendor>Andrew Davison</vendor>
    <homepage href="http://fivedots.coe.pcu.ac.th/~ad/jg"/>
    <description>BugRunner</description>
    <description kind="short">BugRunner: a 2D arcade-style game,
                            using the J3DTimer class</description>
    <icon href="bug32.gif"/>
    <icon kind="splash" href="BRBanner.gif"/>
    <offline-allowed/>
  </information>

  <security>
    <all-permissions/>
  </security>

  <resources os="Windows">
    <j2se version="1.4+"/>
    <jar href="BugRunner.jar" main="true"/>
    <jar href="j3dutils_signed.jar"/>
    <nativelib href="J3DUtilsDLL.jar"/>
  </resources>

  <application-desc main-class="BugRunner"/>
</jnlp>
```

The URL `codebase` value is commented out at this stage, instead we use a path to the local development directory.

The `information` tag contains two forms of textual description, a one-line message and a longer paragraph (confusingly labeled with the attribute value `'short'`). The icon and splash screen images are named; they should be located in the BugRunner directory. The icon is the default size of 32x32 pixels, but other sizes are possible, and several icons with different resolutions can be supplied. GIF or JPEG images can be used. Unfortunately, transparent GIF are rendered with black backgrounds, at least on Windows.

The `offline-allowed` tag states that the application is still able to run when JWS detects that the network is unavailable.

`all-permissions` security is used, which requires that all the JARs named in the `resources` section are signed.

The resources will only be downloaded if the client side OS matches the `os` attribute. There is also an optional `arch` attribute to further constrain the installation.

For example, the following is able to retrieve any one of five different versions of j3daudio.jar depending on the OS and architecture.

```
<resources os="Windows">
  <jar href="jars/j3d/windows/j3daudio.jar"/>
</resources>

<resources os="Linux" arch="x86">     <!-- Linux IBM -->
  <jar href="jars/j3d/linux/i386/j3daudio.jar"/>
</resources>

<resources os="Linux" arch="i386">    <!-- Linux Sun -->
  <jar href="jars/j3d/linux/i386/j3daudio.jar"/>
</resources>

<resources os="Solaris" arch="sparc">
  <jar href="jars/j3d/solaris/j3daudio.jar"/>
</resources>

<resources os="Mac OS X" arch="ppc">
  <jar href="jars/j3d/osx/j3daudio.jar"/>
</resources>
```

The `jars/` directory should be in the same directory as the deployment file.

It may seem rather silly to have five different JARs when their contents should be identical because they're coded in Java. In practice however, this approach avoids incompatibles that may have crept into the different versions.

More details on how the Java 3D libraries can be divided into multiple resource tags are given in the "Marc's Web Start Kamasutra" forum thread, http://forum.java.sun.com/thread.jsp?forum=38&thread=166873.

The `j2se` version tag in BugRunner.jnlp specifies that any version of J2SE or JRE after 1.4.0 can execute the application. JWS will abort if it detects an earlier version when it starts the program. It is possible to specify initial settings for the JRE when it starts, and to trigger an automatic download of a JRE if the client's version is incompatible. The following tags illustrate these features:

```
<j2se version="1.4.2" initial-heap-size="64m"/>
<j2se version="1.4.2-beta"
      href="http://java.sun.com/products/autodl/j2se"/>
```

BugRunner.jnlp specifies that three JARs should be retrieved:

```
<jar href="BugRunner.jar" main="true"/>
<jar href="j3dutils_signed.jar"/>
<nativelib href="J3DUtilsDLL.jar"/>
```

BugRunner.jar contains the application's main() method, and J3DUtilsDLL.jar holds the native library. The JARs must be in the same directory as the .jnlp file, and must be signed.

The `resources` tag is considerably more versatile than our example shows. For instance, it's possible to request that resources be downloaded lazily. This *may* mean that a resource is only retrieved when the application requires it at run time.

```
<jar href="sound.jar" download="lazy"/>
```

Resources may be grouped together into parts, and subdivided into extensions. Each extension is in its own deployment file.

Property name/value pairs, which can be accessed in code with System.getProperty() and System.getProperties() calls, may appear inside the resources section. For instance:

```
<property name="key" value="overwritten"/>
```

The `application-desc` tag states how the application is to be called, and may include argument tags:

```
<application-desc main-class="Foo">
  <argument>arg1</argument>
  <argument>arg2</argument>
</application-desc>
```

The development guide in the J2SE documentation explains many of these tags (<JAVA_HOME>/docs/guide/jws/developersguide/contents.html). An extensive JNLP tag reference page is located at http://lopica.sourceforge.net/ref.html.

Deployment testing should be carried out by moving the relevant files to a different directory on a different machine. For BugRunner, there are six files:

- BugRunner.jnlp: the deployment file;
- bug32.gif and BRBanner.gif: the installer icon and splash;
- BugRunner.jar, j3dutils_signed.jar, and J3DUtilsDLL.jar: the resource JARs.

The chosen directory must match the one used in the codebase attribute at the start of the deployment file.

Double clicking on BugRunner.jnlp should initiate JWS and the installation process, which is shown in Figure 5.

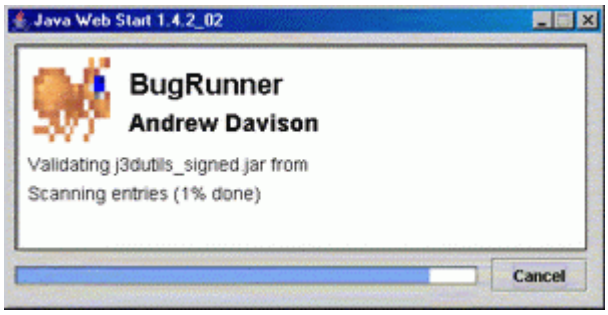

Figure 5. BugRunner Installation.

The dialog box in Figure 6 appears at the point when the application should start executing.

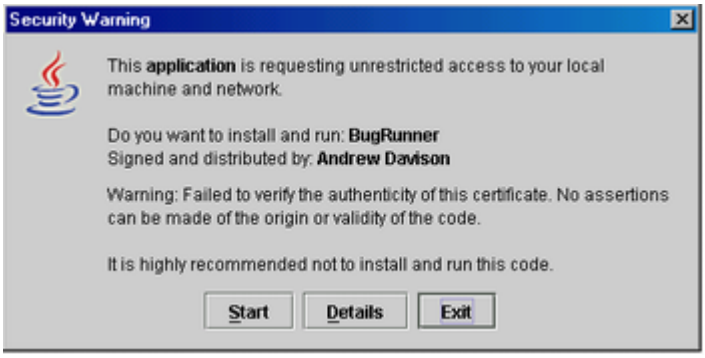

Figure 6. Execute at Your Peril.

Since BugRunner has requested <all-permissions/> access *and* the digital signature uses a self-signed public key certificate, then JWS reports that "it is highly recommended not to install and run this code". This sort of message is deeply worrying to novice users. It can only be removed if we replace the self-signed certificate by a third party certificate, as outlined in section 8.

Clicking on 'Details" will show information obtained from the certificate, or chain of certificates, attached to the JAR. Clicking on "Exit" will stop JWS without executing the application. "Start" will start the application and, depending on how JWS is configured, add a BugRunner item to the Windows Start menu and a BugRunner icon to the desktop.

For more details about the application, and to configure JWS, the application manager should be started. Figure 7 shows the manager for JWS 1.4.2.

Figure 7. The JWS 1.4.2. Application Manager.

The manager has been pensioned off in version 1.5, replaced by a control panel and a cache manager.

Step 6. Place Everything on a Server

It's finally time to move the six BugRunner files to a server:

- BugRunner.jnlp;
- bug32.gif, BRBanner.gif;
- BugRunner.jar, j3dutils_signed.jar, J3DUtilsDLL.jar.

BugRunner.jnlp must be modified to have its codebase use the server's URL:

```
codebase="http://fivedots.coe.psu.ac.th/~ad/jws/BugRunner/"
```


The BugRunner/ directory is placed below jws/ which holds a JWS portal page called index.html. The directory structure is shown in Figure 8.

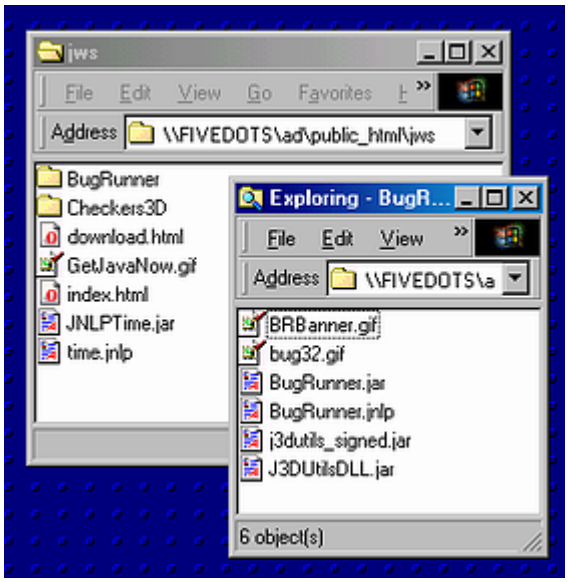

Figure 8. The Server Directories used for JWS.

The portal page (loaded from http://fivedots.coe.psu.ac.th/~ad/jws/) appears as shown in Figure 9.

Figure 9. The JWS Portal Page.

Clicking on the BugRunner link will cause it to be downloaded, with JWS showing the same dialogs as in Figures 5 and 6.

Before starting this phase of the testing, any previous client-side installation of BugRunner should be removed, via the JWS application manager.

6. A JWS Installer for Checkers3D

We'll go through the six installer development steps again, this time for the Checkers3D application.

Step 1. Write the Application

Checkers3D uses the OpenGL Windows version of Java 3D, so requires:

- JAR files: j3daudio.jar, j3dcore.jar, j3dutils.jar, and vecmath.jar
- Native libraries: J3D.dll, j3daudio.dll, J3DUtils.dll

They must be copied into the Checkers3D/ directory, resulting in Figure 10.

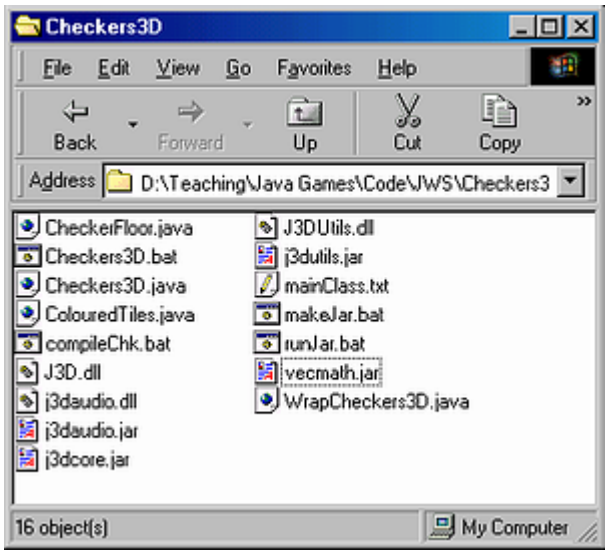

Figure 10. The Initial Checker3D/ Directory.

The compileChk.bat batch file contains:

```
javac -classpath "%CLASSPATH%;vecmath.jar;j3daudio.jar;
                          j3dcore.jar;j3dutils.jar" *.java
```

The Checkers3D.bat batch file:

```
java -cp "%CLASSPATH%;vecmath.jar;j3daudio.jar;
                       j3dcore.jar;j3dutils.jar" Checkers3D
```

Once the program has been tested, the application should be packaged as a JAR. The makeJar.bat batch file has the line:

```
jar cvmf mainClass.txt Checkers3D.jar *.class
```


The manifest details in mainClass.txt are:

```
Main-Class: Checkers3D
Class-Path: vecmath.jar j3daudio.jar j3dcore.jar j3dutils.jar
```

The application now consists of eight files:

- the application JAR file: Checkers3D.jar;
- four Java 3D JAR files: j3daudio.jar, j3dcore.jar, j3dutils.jar, and vecmath.jar;
- three native libraries: J3D.dll, j3daudio.dll, J3DUtils.dll;
- and a partridge in a pear tree (no, I'm joking about that one).

These should be moved to a different directory on a different machine and tested. Double clicking on Checkers3D.jar should start the application.

Step 2. Modify the Application for Deployment

The three DLLs must be placed inside their own JARs:

```
jar cvf J3DDLL.jar J3D.dll
jar cvf j3daudioDLL.jar j3daudio.dll
jar cvf J3DUtilsDLL.jar J3DUtils.dll
```

The main() method of Checkers3D must be modified to call loadLibrary() for the three DLLs.

```
public static void main(String[] args)
{
  // DLLs used by Java 3D extension
  String os = System.getProperty("os.name");
  if (os.startsWith("Windows")) {
    System.out.println("Loading '" + os + "' native libraries...");
    System.out.print("  J3D.dll... ");
    System.loadLibrary("J3D");  // drop ".dll"
    System.out.println("OK");

    System.out.print("  j3daudio.dll... ");
    System.loadLibrary("j3daudio");
    System.out.println("OK");

    System.out.print("  J3DUtils.dll... ");
    System.loadLibrary("J3DUtils");
    System.out.println("OK");
  }
  else {
    System.out.println("Sorry, OS '" + os + "' not supported.");
    System.exit(1);
  }

  new Checkers3D();
} // end of main()
```


Step 3. Create a Public/Private Keypair for Signing the Application

A new keypair is generated in the keystore:

```
keytool -genkey -keystore MyKeyStore -alias Checkers3D
```

Step 4. Sign Everything with the Private Key

For the Checkers application, there are eight JAR files:

- the application JAR file: Checkers3D.jar;
- four Java 3D JAR files: j3daudio.jar, j3dcore.jar, j3dutils.jar, vecmath.jar;
- three native libraries JARs : J3DDLL.jar, j3daudioDLL.jar, J3DUtilsDLL.jar.

The Sun JAR files are copied and signed:

```
jarsigner -keystore MyKeyStore -signedjar j3daudio_signed.jar
                                          j3daudio.jar  Checkers3D

jarsigner -keystore MyKeyStore -signedjar j3dcore_signed.jar
                                          j3dcore.jar Checkers3D

jarsigner -keystore MyKeyStore -signedjar j3dutils_signed.jar
                                          j3dutils.jar Checkers3D

jarsigner -keystore MyKeyStore -signedjar vecmath_signed.jar
                                          vecmath.jar Checkers3D
```

The DLL JAR files are signed in place:

```
jarsigner -keystore MyKeyStore J3DDLL.jar Checkers3D

jarsigner -keystore MyKeyStore j3daudioDLL.jar Checkers3D

jarsigner -keystore MyKeyStore J3DUtilsDLL.jar Checkers3D
```

The manifest information in mainClass.txt is changed to:

```
Main-Class: Checkers3D
Class-Path: vecmath_signed.jar j3daudio_signed.jar
                  j3dcore_signed.jar j3dutils_signed.jar
```

After Checkers3D.jar is regenerated, it is signed:

```
jarsigner -keystore MyKeyStore Checkers3D.jar Checkers3D
```

Step 5. Create a Deployment File

The deployment file for the Checkers3D application, Checkers3D.jnlp, is:


```
<?xml version="1.0" encoding="utf-8"?>
<!-- Checkers3D Deployment -->

<jnlp spec="1.0+"
  codebase="file:///D:/Teaching/Java Games/Code/JWS/Checkers3D/"
  href="Checkers3D.jnlp"
  >

  <information>
    <title>Checkers3D</title>
    <vendor>Andrew Davison</vendor>
    <homepage href="http://fivedots.coe.pcu.ac.th/~ad/jg"/>
    <description>Checkers3D</description>
    <description kind="short">Checkers3D: a simple java 3D example
        showing a blue sphere above a checkboard.</description>
    <icon href="chess32.gif"/>
    <icon kind="splash" href="startBanner.gif"/>
    <offline-allowed/>
  </information>

  <security>
    <all-permissions/>
  </security>

  <resources os="Windows">
    <j2se version="1.4+"/>
    <jar href="Checkers3D.jar" main="true"/>

    <jar href="j3daudio_signed.jar"/>
    <jar href="j3dcore_signed.jar"/>
    <jar href="j3dutils_signed.jar"/>
    <jar href="vecmath_signed.jar"/>

    <nativelib href="J3DDLL.jar"/>
    <nativelib href="j3daudioDLL.jar"/>
    <nativelib href="J3DUtilsDLL.jar"/>
  </resources>

  <application-desc main-class="Checkers3D"/>
</jnlp>
```

At this stage, codebase is pointing to a local directory.

The icons and splash screen images, chess32.gif and startBanner.gif, must be placed in the Checkers3D/ directory. The resources section lists eight JAR files.

Double clicking on Checkers3D.jnlp should initiate JWS and the installation process, as shown in Figure 11.

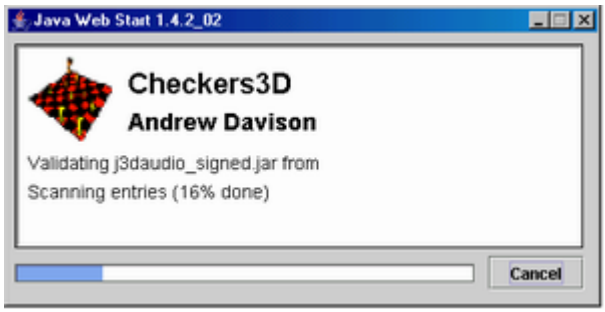

Figure 11. Checkers3D Installation.

The dialog box in Figure 12 appears at the point when the application should start executing.

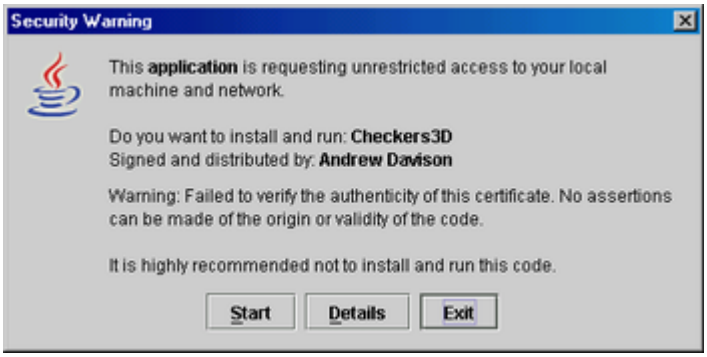

Figure 12. Execute at Your Peril Again.

Checkers3D suffers from the "highly recommended not to install and run this code" message, since it uses a self-signed certificate just like BugRunner.

Step 6. Place Everything on a Server

It's finally time to move the 11 Checkers3D files to a server:

- Checkers3D.jnlp;
- Checkers3D.jar;
- j3daudio.jar, j3dcore.jar, j3dutils.jar, vecmath.jar;
- J3DDLL.jar, j3daudioDLL.jar, J3DUtilsDLL.jar;
- chess32.gif, startBanner.gif.

Checkers3D.jnlp must be modified to have its codebase use the server's URL:

```
codebase="http://fivedots.coe.psu.ac.th/~ad/jws/Checkers3D/"
```


The Checkers3D/ directory is placed below jws/ on the server, as is shown in Figure 13.

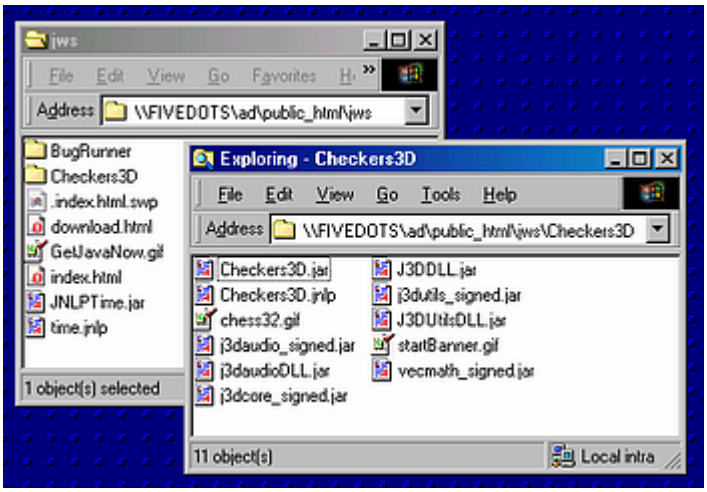

Figure 13. The Cherckers3D Directory on the server.

Clicking on the Checkers3D link on the JWS portal page (see Figure 9) will cause it to be downloaded, with JWS showing the same dialogs as in Figures 11 and 12.

Before starting this phase, any previous installation of Checkers3D should be removed, via the JWS application manager.

7. The JWS Portal Page

The portal page (shown in Figure 9) contains a mix of JavaScript and VBScript which attempts to detect if JWS is present on the client machine. If it isn't, then the user is given the option of downloading a JRE. The task is considerably complicated by the variations between browsers, OSes, and platforms.

A simplified version of the page:

```
<HTML>
<HEAD><TITLE>Java Web Start Portal</TITLE>
<script language="JavaScript" type="text/javascript">

  // check in various ways if JWS is installed;
  // decide if the browser is IE on Windows

  function insertLink(url, name) {...}
  // add a link to the named URL if JWS is present

  function installInfo() {...}
  // report whether JWS has been found

  function nativeReport() {...}
  // warn about the use of Windows DLLs by BugRunner and Checkers3D
</script>

<script language="VBScript">
  // check for various JWS objects in the Windows registry
```



```
</script>

</HEAD>
<BODY><H1>Java Web Start Portal</H1>

<script language="JavaScript" type="text/javascript">
  installInfo();
  nativeReport();
</script>

<h2>Installation Links</h2>
<P><ul>
  <li><script language="JavaScript" type="text/javascript">
    insertLink("time.jnlp", "Official US Time");
  </script>

  <li><script language="JavaScript" type="text/javascript">
    insertLink("BugRunner/BugRunner.jnlp", "BugRunner");
  </script>

  <li><script language="JavaScript" type="text/javascript">
    insertLink("Checkers3D/Checkers3D.jnlp", "Checkers3D");
  </script>
</ul></P></BODY>
</HTML>
```

The JavaScript part of the page sets a flag, javawsInstalled, to 1 if it thinks that JWS is (probably) installed. It also determines whether Windows and Internet Explorer are being used. Three functions are defined, insertLink(), installInfo(), and nativeReport (), which are called in the body of the page.

If the client's browser is Internet Explorer on Windows, then some VBScript code also has a go at setting javawsInstalled.

7.1. Setting javawsInstalled in JavaScript

We try four tests:

1. Check if the JNLP or applet MIME type is set in the browser:

```
if (navigator.mimeTypes && navigator.mimeTypes.length) {
  if (navigator.mimeTypes['application/x-java-jnlp-file'])
    javawsInstalled = 1;
  if (navigator.mimeTypes['application/x-java-applet'])
    javawsInstalled = 1;
}
```

2. Check if Java is enabled on non-windows OSes:

```
if (!isWin && navigator.javaEnabled())
  javawsInstalled = 1;
```

3. Check for the presence of LiveConnect, an indicator of JWS' presence:

```
if (window.java != null)
  javawsinstalled = 1;
```


4. Check for the Java plug-in:

```
var numPlugs=navigator.plugins.length;
if (numPlugs) {
  for (var i=0; i < numPlugs; i++) {
    var plugNm = navigator.plugins[i].name.toLowerCase();
    if (plugNm.indexOf('java plug-in') != -1) {
      javawsinstalled = 1;
      break;
    }
  }
}
```

Many of these tests only suggest that JWS is available.

None of these approaches is guaranteed to work on every platform, OS, or browser. For example, the JNLP MIME type must be set by an Opera user manually, otherwise it is not detected, even on a machine with JWS. Only Mozilla-based browsers and Opera support LiveConnect. The javaEnabled() call will always return true on a Windows machine because it detects the less-than-useful Microsoft JVM. Several browser do not store plug-in information, and even if the Java plug-in is found, it may not be the JVM utilized by the browser.

7.2. The JavaScript Functions

installInfo() prints a link to one of Sun's download pages for JREs if javawsInstalled is 0. The section in the JWS developers guide called "Creating the Web Page that Launches the Application" (<JAVA_HOME>/docs/guide/jws/developersguide/launch.html) gives an example of the auto-installation of a JRE for Windows from a portal page. I haven't used this approach since it seems better to leave the download choice up to the user.

nativeReport() prints a warning if the client OS is not Windows, since BugRunner and Checkers3D rely on DLLs.

insertLink() adds a JNLP link to the page only if javawsInstalled is 1.

7.3. The VBScript Code

The VBScript code is a long multi-way branch which checks the Windows registry for different versions of the JWS. A typical test:

```
If (IsObject(CreateObject("JavaWebStart.isInstalled.1.4.2"))) Then
  javawsInstalled = 1
```

The numerous JWS versions necessitate several branches. The developers guide, up to version 1.4.2, refers to JavaWebStart.isInstalled.1, JavaWebStart.isInstalled.2, and JavaWebStart.isInstalled.3, which in all likelihood never existed.

There are also tests for the JWS-related MIME types, application/x-java-jnlp-file and application/x-java-applet.

7.4. More Information on Portal Pages

The portal page example in the developers guide for version 1.5.0 uses a similar approach to the code here; the main difference, as mentioned above, is the auto-installation of a Windows JRE.

Crucial to portal page code is the need to accurately detect the browser and OS. A good discussion of these problems can be found at the QuirksMode JavaScript site, http://www.quirksmode.org/index.html?/js/detect.html.

A more rigorous way of determining the browser's JRE is to load a simple applet which interrogates the JVM's "java.version" property. This approach is described in the Java Plug-in forum thread "How to Detect a Java Plugin from JavaScript", http://forum.java.sun.com/thread.jsp?thread=168544&forum=30&message=527124.

Two online example of this approach, containing many other good Java detection techniques as well, are at http://members.toast.net/4pf/javasniff.html and http://cvs.sdsc.edu/cgi-bin/cvsweb.cgi/mbt/mbt/apps/Explorer/Detect.js?rev=1.2

8. Third-Party Certificates

Figures 6 and 12 show the problem with using self-signed certificates in an application – JWS issues a scary message.

The solution is to replace the certificate by one generated by a trusted third party, a Certification Authority (CA). Popular CAs include Verisign (http://www.verisign.com/), Thawte (http://www.thawte.com/) and Entrust (http://www.entrust.com). These companies charge money for their services, but a free alternative is CACert.org (https://www.cacert.org/).

Beefing up the certificate for a keypair consists of the following steps:

1. Extract a Certificate Signing Request (CSR) from the keypair.
2. Send the CSR to the CA, requesting a certificate.
3. After checking the returned certificate, import it into the keystore, replacing the keypair's self-signed certificate.
4. Start signing JARs with the keypair.

Step 1. Generate a CSR with the -certreq keytool option:

```
keytool -certreq -keystore MyKeyStore -alias BugRunner
                                          -file BugRunner.csr
```

This generates a CSR for the BugRunner keypair, stored in BugRunner.csr, which is a text file of the form:

```
-----BEGIN NEW CERTIFICATE REQUEST-----
MIICoDCCAl4C.....
.....
-----END NEW CERTIFICATE REQUEST-----
```


Step 2. The CSR is sent to the CA, usually by pasting its text into a Web form accessed via a secure link (a https URL). At CACert.org, this step requires some preliminary work. The user must first join CACert.org, which is free, and send in details about the Web domain that he/she controls. This information is double checked with the site's Web administrator by e-mail. Only then can CSRs be submitted. The certificate generated in response to a CSR is called a *server certificate* by CACert.org.

Step 3. The server certificate is received in an ordinary e-mail, and so should be examined before being added to the keystore.

```
keytool -printcert -file certfile.cer
```

We assume that the text message is stored in certfile.cer. If the .cer extension is used, then many browsers will be able to open and interpret the file's certificate contents. The text should look like:

```
-----BEGIN CERTIFICATE-----
MIICxDCCAi0....
....
-----END CERTIFICATE-----
```

Although we often talk about a server certificate, the data may actually consist of a chain of certificates rather than just one.

Once the user is happy that the details match those supplied in the original CSR, the server certificate can be imported into the keystore. In our case, it replaces the self-signed certificate for the BugRunner keypair:

```
keytool -import -trustcacerts -keystore MyKeyStore
                        -alias BugRunner -file certfile.cer
```

The server certificate is automatically verified: in a chain, the current certificate is trusted because of the certificate at the next level up. This continues until the certificate for the CA is reached. This may already be a *trusted* (or root) certificate, stored in JWS's cacerts keystore. cacerts comes pre-built with Verisign, Thawte, and Entrust trusted certificates, but doesn't have any from CACert.org.

CACert.org offers a root certificate for download, which can be added to cacert (if you have write permissions). Alternatively, it can be placed in the local MyKeyStore keystore as a trusted certificate:

```
keytool -import -alias cacertOrg -keystore MyKeyStore
                                          -file caRoot.cer
```

Here, we assume that the root certificate is stored in caRoot.cer, and is saved under the name (alias) cacertOrg. Since no keypair exists for cacertOrg, keytool assumes it is a trusted certificate. This can be verified by listing the contents of the keystore:


```
keytool -list -keystore MyKeyStore
```

The root certificate should be imported before the server certificate, or the server certificate's authentication process will end with the warning "Failed to establish chain from reply".

An alternative is to 'glue' the root and server certificates together as a single entity, then import the result into the keystore as a replacement for the self-signed certificate. This process is described by Chris W. Johnson at http://gargravarr.cc.utexas.edu/chrisj/misc/java-cert-parsing.html.

Step 4. Signing can now commence with the third-party certified BugRunner keypair:

```
jarsigner -keystore MyKeyStore foo.jar BugRunner
```

9. More Information

If you've installed the J2SE documentation, then there's a lot of JWS information already available locally in <JAVA_HOME>/docs/guide/jws/index.html, including the developers guide. In J2SE 1.5.0, there are several JWS examples in the Java demos. In earlier JWS versions, the code was scattered through the developers guide.

The JWS home page at Sun (http://java.sun.com/products/javawebstart/) contains links to an official FAQ, technical articles, and installer demos.

In J2SE/JRE 1.4.2. or earlier, the JWS materials did not contain the developers pack, which includes the complete JWS specification, JNLP API documentation, jardiff tool, and additional libraries. These have been folded into the main JWS release since 1.5. The pack is available from http://java.sun.com/products/javawebstart/download-jnlp.html.

The Java Web Start and JNLP developers forum is quite active (http://forum.java.sun.com/forum.jsp?forum=38).

The best unofficial JWS site is http://lopica.sourceforge.net/, with a lengthy FAQ, useful reference sections and links. However, it hasn't been updated in some time, and a lot of the FAQ is about earlier versions of JWS.

The installer examples from Sun are at http://java.sun.com/products/javawebstart/demos.html. Another great source is Up2Go.Net (http://www.up2go.net), with installer categories including multimedia, communications, and over 40 games.

9.1. JWS and Java 3D

As mentioned a few times, issues concerning JWS and Java 3D are described in the forum thread "Marc's Web Start Kamasutra" at http://forum.java.sun.com/thread.jsp?forum=38&thread=166873.

The problems that arise when utilizing native libraries in stand-alone applications, JWS installers, and applets are discussed in:

Transparent Java Standard Extensions with Native Libraries on Multiple Platforms
Pierre A.I. Wijkman, Mitra Wijkman, Suru Dissanaike
PPPJ'03: 2nd Int. Conf. on the Principles and Practice of Prog. in Java, 2000
`http://atlas.dsv.su.se/~pierre/a/papers/nativelibs.pdf`

This paper is of particular interest since it uses Java 3D as its example extension with native libraries. The code is available from http://www.dsv.su.se/~adapt/im/t_nativelibs/, with an explanation in Swedish.

The FlyingGuns game/simulation is a real-life use of Java 3D and JWS (http://www.flyingguns.com), using JNLP resource extensions to position the Java 3D libraries in a separate deployment file. The direct link to its top-level deployment file is http://www.hardcode.de/fg/webstart/flyingguns.jnlp.

Xith3D is a scene graph-based 3D programming API, similar to Java 3D. However, Xith3D runs on top of JOGL, a set of Java bindings for OpenGL. William Denniss has a detailed example of how to use JWS to install Xith3D at http://www.xith.org/tutes/GettingStarted/html/deploying_xith3d_games_with.html.

A brief introduction to JWS by Kevin Glass, for a space game using JOGL, is at http://www.cokeandcode.com/info/webstart-howto.html.

The installers for JOGL and Xith3D handle similar native library problems to those found in Java 3D.

