

*This section was originally published in the first edition of Learning Java as part of the Applets chapter. In Java 1.4, the Java Plug-in makes it possible for users to accept signed applets based on certificates without requiring local installation of keys. We include this section covering the importation of public keys and certificates for support of legacy environments that may still use these techniques.*

### The TestWrite example

Let's take a look at an example, just to prove that you can actually sign objects with the older releases of the SDK and the Java Plug-in. In the process of discussing the example, we'll point out some things that were lacking.

Use your browser to navigate to <http://www.oreilly.com/catalog/learnjava/TestWrite/Unsigned.html>. When you push the button, this applet attempts to write a harmless file on the local host. Give it a try. The applet should fail with a security exception and display a message.

At this point, load a web page that displays the same applet: <http://www.oreilly.com/catalog/learnjava/TestWrite/Signed.html>. The only difference is that I've wrapped the applet in a JAR file that I have signed, with a key and certificate that I generated using `keytool`. When it loads the applet's JAR file, the Java Plug-in also retrieves the signature. However, nothing has changed for the applet just yet. When you click the button, you still get a security exception when the applet tries to write a file.

But now that an identity is associated with the applet, we can do more. You can download my certificate from <http://www.oreilly.com/catalog/learnjava/TestWrite/Jonathan.cer>. Enabling the signed applet to write a file takes two steps. First, you'll have to import this certificate file into a keystore, which is simply a collection of keys and certificates. The second step is to create a policy file that tells the Java Plug-in that this applet is allowed to write a file.

To import the certificate file, use the `keytool` utility that comes with the SDK. For this example, we'll create an entirely new keystore. The following command imports the Jonathan.cer certificate file and puts it in a keystore called client.keystore. If the keystore file doesn't exist, it will be created.

```
C:\> keytool -import -file Jonathan.cer -alias Jonathan -keystore client.keystore -storepass  
buendia
```

The `-alias` option tells `keytool` what alias to use for the imported certificate. This is just a name you can use to refer to the keys in the certificate. The integrity of the keystore is protected using a password, specified with the `-storepass` option. Whenever you access the keystore, you'll need to supply the same password to verify that the keystore hasn't been tampered with.

After you type this command, `keytool` prints out some information about the certificate you are about to import. Then it asks if this certificate should be trusted. This is an important question, as anyone can generate this kind of self-signed certificate. (I've decided to be my own certificate authority of one, for now.)

```
Owner: CN=Jonathan Knudsen  
Issuer: CN=Jonathan Knudsen
```

```
Serial number: 3804d2c1
Valid from: Wed Oct 13 14:43:13 EDT 1999 until:
  Tue Jan 11 13:43:13 EST 2000
Certificate fingerprints:
  MD5:  A6:56:C2:52:2B:20:69:26:67:A2:0B:78:D3:3C:AC:45
  SHA1: A3:AF:41:93:73:1F:48:EE:6E:F5:8B:93:66:3E:73:F5:99:5D:54:AF
Trust this certificate? [no]:  yes
Certificate was added to keystore

C:\>
```

What can we do? We might simply get a certificate signed by a more popular authority than myself. But you would still have to register the CA's certificate somehow, following this same process.

For now, you can verify this certificate by hand. We'll do this not only to get through this example, but because it presents a useful analogy to the process of certification. The SHA1 "thumbprint" that `keytool` displays should match this:

```
| A3:AF:41:93:73:1F:48:EE:6E:F5:8B:93:66:3E:73:F5:99:5D:54:AF |
```

If everything looks good, you can type "yes" to accept this certificate. By doing so, you are saying that you accept my identity, and believe that I have in fact signed certificates that match this thumbprint. You have effectively chained your trust to this printed page rather than an online certificate authority. You can be reasonably sure that no one has gone to the effort of issuing counterfeit O'Reilly books.

You are now ready to assign greater privileges to this applet. For example, you can allow it to write files. To grant privileges in this way, you need to create a security policy. Use the SDK's `policytool` utility (which we introduced in chapter 3, "Tools of the Trade"). Start the utility from a command line:

```
| C:\> policytool |
```

The `policytool` interface isn't pretty, but it gets the job done. When you first run `policytool`, you may get a message that it can't find the default policy file. Just click **OK**. (If you don't get this message, choose **New** from the **File** menu to start with a clean slate.)

Our eventual goal is to tell the Plug-in that any applet we have signed is allowed to write a file on your computer. The policy file needs to reference the certificate in the keystore we just created. The first thing we need to do, therefore, is tell `policytool` where to find the keystore. Select the **Change KeyStore** option from the **Edit** method. Type the URL of the keystore file you just created. (On a Windows system, it looks something like this: `file://c:/SignTest/client.keystore`.) You can leave the keystore type blank. Then click **OK**.

Now that we're pointed at the right keystore, we need to add a permission for file writing. Click on the **Add Policy Entry** button. In the window that comes up, leave the CodeBase field blank. Fill in the SignedBy

field with `Jonathan`. This means that we're setting a policy for all code signed by `Jonathan`. (Remember, `Jonathan` is the alias we used for the certificate when it was imported to the keystore.)

To give code signed by Jonathan permission to write files, click on the **Add Permission** button. In the next dialog, choose **FilePermission** from the top combo box. In the second combo box, choose `<<ALL FILES>>`. In the third combo box, choose **write**. Finally, click on **OK** to finish creating the permission. Then click on **Done** to finish creating the policy.

Having set this up, you now need to save the policy file. Choose **Save** from the **File** menu; save this policy file as `client.policy`. You can exit the `policytool` utility now.

The last thing you need to do is tell the Java Plug-in to use the policy file we've just so laboriously created. To do this, use a text editor to edit the `java.security` file. This file is found underneath the installation directory for the JRE, in the `lib/security` subdirectory. (If you have both the SDK and the JRE installed, you will actually have two copies of this file in different locations. For example, you might have one JRE installed in `C:\sdk1.3\jre` and another in `C:\Program Files\JavaSoft\JRE`. Make sure you edit the `java.security` file that is used by the Plug-in, which is probably the one that's not in the SDK installation directory.)

Once you've found the right `java.security` file, you need to find the section that contains entries for `policy.url`. Add a line to reference the policy file we just created, like this:

```
| policy.url.3=file:/c:/SignTest/client.policy |
```

Finally, you're ready to test the signed applet. (You may need to restart your browser.) Navigate to the signed applet page. When you press the button, the file is written successfully.

So, how did we create this certificate and sign the JAR file? Let's move on.

## Keystores, Keys, and Certificates

The SDK supports keystores that hold identities along with their public keys, private keys, and certificates. It includes a utility called `keytool` that manages keystores. You just saw how Jonathan's certificate could be contained in a keystore. The identities in keystores are visible to Java. We'll discuss what that means in a bit, but for the most part, we'll only use this database as a repository while we create and work with our identity locally.

An identity can be a person, an organization, or perhaps a logical part of an organization. Before it can be used, an identity must have a public key and at least one certificate validating its public key. `keytool` refers to entities in the local database by IDs or aliases. These names are arbitrary and are not used outside of the keystore and any policy files that reference it. Identities that have a private key stored locally in the keystore, along with their public key, can be used to sign JAR files. These identities are also called *signers*.

The default location for a keystore is the file `.keystore` in the user's home directory. On a single user system, the Java installation directory is used instead of the user's home directory.

Beware if you have both the SDK and HotJava installed in separate locations on a single user system, such as a PC running Windows. You will have to inform HotJava where to find the SDK package using the `JDK_HOME` environment variable. The default keystore location is used by `keytool` unless you specify another keystore with the `-keystore` option.

If you are going to maintain any private keys in a keystore (if you will have any signers), you must take special care to keep the keystore file safe (and not publicly readable). Private keys must be kept private.

## Public and private keys

We can create a new entry in the default keystore, complete with a key pair, with the following `keytool` command:

```
C:\> keytool -genkey -alias Jonathan -keyalg DSA -keysize 1024 -dname "CN=Jonathan Knudsen,  
OU=Technical Publications, O=O'Reilly & Associates, C=US" -keypass gianpi -storepass buendia
```

There are a lot of options to explain. The most important one is `-genkey`, which tells `keytool` to create a new key pair for this entry. A key pair enables this entry to sign code. The `-alias` option supplies an alias for this entry, Jonathan. The `-keyalg` argument, DSA, is the algorithm for which we are going to generate the keys. The current release of Java only supports one: DSA, the Digital Signature Algorithm, which is a U.S. government standard for signing. The `-keysize` argument is the key length in bits. For most algorithms, larger key sizes provide stronger encryption. DSA supports keys of either 512 or 1024 bits. You should use the latter, unless you have a specific reason to do otherwise.

`keytool` generates the keys and places them in the default keystore. Private keys are specially protected using the `-keypass` option. To retrieve Jonathan's private key, you will have to know the correct key password. The integrity of the keystore as a whole is protected by the `-storepass` option. You need to supply the same keystore password to retrieve data from this keystore later.

Once we've created a keystore entry, we can display it with the command:

```
C:\> keytool -list -alias Jonathan -storepass  
buendia
```

To see more detail, add the `-v` option (for "verbose"):

```
C:\> keytool -list -alias Jonathan -v  
-storepass buendia
```

Or we can list the entire contents of the database:

```
C:\> keytool -list -storepass buendia
```

## Certificates

Now that we have keys, we want a certificate in which to wrap our public key for distribution. Ideally, at this point, we'd send a public key to a trusted certificate authority and receive a certificate in return. `keytool` can generate such a request, called a Certificate Signing Request (CSR). To generate a signing request for the entry we just created, you would do this:

```
| C:\> keytool -csr -alias Jonathan -file Jonathan.csr -keypass firenze -storepass buendia
```

You need to specify the alias for the entry you want, a filename where the CSR will be written, and the password for the private key. Once you've generated the CSR file, you can send it off to your favorite Certificate Authority. Once they've performed some identity checks on you, and once you pay them, they will send a certificate back to you. Suppose they send it back in a file called Jonathan.x509. You can use `keytool` to import this certificate as follows:

```
| C:\> keytool -import -alias Jonathan -file Jonathan.x509 -keypass firenze -storepass buendia
```

To demonstrate the features of `keytool`, we will serve as our own authority (as we did in the example) and use our own self-signed certificate. It turns out that `keytool` already did this for us when we created keys! A self-signed certificate already exists in the keystore; all we have to do is export it as follows:

```
| C:\> keytool -export -alias Jonathan -file Jonathan.cer -storepass buendia
```

## Signing JARs

If we have a signer keystore entry, initialized with its private and public keys, we are ready to sign JAR files. This is accomplished using another command-line utility, `jarsigner`. All we need to do is specify which keystore entry should do the signing, which JAR needs to be signed, and the keystore password.

```
| C:\> jarsigner -storepass buendia testwrite.jar Jonathan
```

If we now list the archive, we will see that `jarsigner` has added two files to the *META-INF* directory: *JONATHAN.SF* and *JONATHAN.DSA*. *JONATHAN.SF* is the signature file—it is like the manifest file for this particular signature. It lists the objects that were signed and the signature algorithms. *JONATHAN.DSA* is the actual binary signature.