Lesson: Packaging Programs in JAR Files

The Java™ Archive (JAR) file format enables you to bundle multiple files into a single archive file. Typically a JAR file contains the class files and auxiliary resources associated with applets and applications.

The JAR file format provides many benefits:

- **Security**: You can digitally sign the contents of a JAR file. Users who recognize your signature can then optionally grant your software security privileges it wouldn't otherwise have.
- **Decreased download time**: If your applet is bundled in a JAR file, the applet's class files and associated resources can be downloaded to a browser in a single HTTP transaction without the need for opening a new connection for each file.
- **Compression**: The JAR format allows you to compress your files for efficient storage.
- **Packaging for extensions**: The extensions framework provides a means by which you can add functionality to the Java core platform, and the JAR file format defines the packaging for extensions. By using the JAR file format, you can turn your software into extensions as well.
- **Package Sealing**: Packages stored in JAR files can be optionally sealed so that the package can enforce version consistency. Sealing a package within a JAR file means that all classes defined in that package must be found in the same JAR file.
- **Package Versioning**: A JAR file can hold data about the files it contains, such as vendor and version information.
- **Portability**: The mechanism for handling JAR files is a standard part of the Java platform's core API.

This lesson has four sections:

1. **Using JAR Files: The Basics** This section shows you how to perform basic JAR-file operations, and how to run software that is bundled in JAR files.
2. **Working with Manifest Files: The Basics** This section explains manifest files and how to customize them so you can do such things as seal packages and set an application's entry point.
3. **Signing and Verifying JAR Files:** This section shows you how to digitally sign JAR files and verify the signatures of signed JAR files.
4. **Using JAR-related APIs** This section introduces you to some of the JAR-handling features of the Java platform. The JAR file format is an important part of the Java platform's extension mechanism. You can learn more about that aspect of JAR files in the The Extension Mechanism trail of this tutorial.

Using JAR Files: The Basics

JAR files are packaged with the ZIP file format, so you can use them for tasks such as lossless data compression, archiving, decompression, and archive unpacking. These tasks are among the most common uses of JAR files, and you can realize many JAR file benefits using only these basic features.

To perform basic tasks with JAR files, you use the Java Archive Tool provided as part of the Java Development Kit (JDK). Because the Java Archive tool is invoked by using the `jar` command, this tutorial refers to it as 'the Jar tool'.

Common JAR file operations

Operation	Command
To create a JAR file	`jar cf` *`jar-file input-file(s)`*
To view the contents of a JAR file	`jar tf` *`jar-file`*
To extract the contents of a JAR file	`jar xf` *`jar-file`*
To extract specific files from a JAR file	`jar xf` *`jar-file archived-file(s)`*
To run an application packaged as a JAR file (requires the `Main-class` manifest header)	`java -jar` *`app.jar`*
To invoke an applet packaged as a JAR file	`<applet code=AppletClassName.class archive="JarFileName.jar" width=width height=height> </applet>`

Creating a JAR File

The basic format of the command for creating a JAR file is:

```
jar cf jar-file input-file(s)
```

The options and arguments used in this command are:

- The **c** option indicates that you want to create a JAR file.
- The **f** option indicates that you want the output to go to a file rather than to stdout.
- **jar-file** is the name that you want the resulting JAR file to have. You can use any filename for a JAR file. By convention, JAR filenames are given a .jar extension, though this is not required.
- The **input-file(s)** argument is a space-separated list of one or more files that you want to include in your JAR file. The input-file(s) argument can contain the wildcard * symbol. If any of the "input-files" are directories, the contents of those directories are added to the JAR archive recursively.

This command will generate a compressed JAR file and place it in the current directory. The command will also generate a default manifest file for the JAR archive.

➢ **Note**: The metadata in the JAR file, such as the entry names, comments, and contents of the manifest, must be encoded in UTF8.

jar command options

v Produces verbose output on stdout while the JAR file is being built. The verbose output tells you the name of each file as it's added to the JAR file.

0 (zero) Indicates that you don't want the JAR file to be compressed. `jar` compresses files by default.

M Indicates that the default manifest file should not be produced.

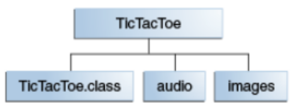
m Used to include manifest information from an existing manifest file. The format for using this option is: `jar cmf jar-file existing-manifest input-file(s)` See Modifying a Manifest File for more information about this option.

> **Warning**: The manifest must end with a new line or carriage return. The last line will not be parsed properly if it does not end with a new line or carriage return.

Note: When you create a JAR file, the time of creation is stored in the JAR file. Therefore, even if the contents of the JAR file do not change, when you create a JAR file multiple times, the resulting files are not exactly identical. You should be aware of this when you are using JAR files in a build environment. It is recommended that you use versioning information in the manifest file, rather than creation time, to control versions of a JAR file. See the Setting Package Version Information section.

An Example

Let us look at an example. A simple TicTacToe applet. You can see the source code of this applet by downloading the JDK Demos and Samples bundle from Java SE Downloads. This demo contains class files, audio files, and images having this structure:



TicTacToe folder Hierarchy

The audio and images subdirectories contain sound files and GIF images used by the applet. You can obtain all these files from jar/examples directory when you download the entire Tutorial online. To package this demo into a single JAR file named TicTacToe.jar, you would run this command from inside the TicTacToe directory:

```
jar cvf TicTacToe.jar TicTacToe.class audio images
```

The audio and images arguments represent directories, so the Jar tool will recursively place them and their contents in the JAR file. The generated JAR file TicTacToe.jar will be placed in the current directory. Because the command used the v option for verbose output, you would see something similar to this output when you run the command:

```
adding: TicTacToe.class (in=3825) (out=2222) (deflated 41%)
adding: audio/ (in=0) (out=0) (stored 0%)
adding: audio/beep.au (in=4032) (out=3572) (deflated 11%)
adding: audio/ding.au (in=2566) (out=2055) (deflated 19%)
```

You might want to avoid compression, for example, to increase the speed with which a JAR file could be loaded by a browser. Uncompressed JAR files can generally be loaded more quickly than compressed files because the need to decompress the files during loading is eliminated. However, there is a tradeoff in that download time over a network may be longer for larger, uncompressed files.

The Jar tool will accept arguments that use the wildcard * symbol. As long as there weren't any unwanted files in the TicTacToe directory, you could have used this alternative command to construct the JAR file:

```
jar cvf TicTacToe.jar *
```

Though the verbose output doesn't indicate it, **the Jar tool automatically adds a manifest file to the JAR archive** with path name **META-INF/MANIFEST.MF**.

In the above example, the files in the archive retained their relative path names and directory structure. The Jar tool provides the **-C** option that you can use to create a JAR file in which the relative paths of the archived files are not preserved. It's modeled after **TAR's -C** option.

As an example, suppose you wanted to put audio files and gif images used by the TicTacToe demo into a JAR file, and that you wanted all the files to be on the top level, with no directory hierarchy. You could accomplish that by issuing this command from the parent directory of the images and audio directories:

```
jar cf ImageAudio.jar -C images . -C audio .
```

The -C images part of this command directs the Jar tool to go to the images directory, and the . following -C images directs the Jar tool to archive all the contents of that directory. The -C audio . part of the command then does the same with the audio directory. The resulting JAR file would have this table of contents:

```
META-INF/MANIFEST.MF
cross.gif
not.gif
beep.au
ding.au
return.au
yahoo1.au
yahoo2.au
```

By contrast, suppose that you used a command that did not employ the -C option:

```
jar cf ImageAudio.jar images audio
```

The resulting JAR file would have this table of contents:

```
META-INF/MANIFEST.MF
images/cross.gif
images/not.gif
audio/beep.au
audio/ding.au
audio/return.au
audio/yahoo1.au
audio/yahoo2.au
```

Viewing the Contents of a JAR File

This command will display the JAR file's table of contents to **stdout**. The basic format of the command for viewing the contents of a JAR file is: Let's look at the options and argument used in this command:

```
jar tf jar-file
```

- The **t** option indicates that you want to view the table of contents of the JAR file.
- The **f** option indicates that the JAR file whose contents are to be viewed is specified on the command line.
- The **jar-file** argument is the path and name of the JAR file whose contents you want to view.

- All pathnames are displayed with **forward slashes**, regardless of the platform or operating system you're using. Paths in JAR files are always relative; you'll never see a path beginning with C:, for example.

```
META-INF/
META-INF/MANIFEST.MF
UseMe.java
```

Extracting the Contents of a JAR File

The basic command to use for extracting the contents of a JAR file is:

```
jar xf jar-file [archived-file(s)]
```

- The **x** option indicates that you want to extract files from the JAR archive.
- The **f** options indicates that the JAR file from which files are to be extracted is specified on the command line, rather than through stdin.
- The **jar-file** argument is the filename (or path and filename) of the JAR file from which to extract files.
- **archived-file(s)** is an optional argument consisting of a space-separated list of the files to be extracted from the archive. If this argument is not present, the Jar tool will extract all the files in the archive.

When extracting files, the Jar tool makes copies of the desired files and writes them to the current directory, reproducing the directory structure that the files have in the archive. The original JAR file remains unchanged.

Caution: When it extracts files, the Jar tool will overwrite any existing files having the same pathname as the extracted files.

Updating a JAR File

The Jar tool provides a u option which you can use to update the contents of an existing JAR file by modifying its manifest or by adding files.

```
jar uf jar-file input-file(s)
```

- The **u** option indicates that you want to update an existing JAR file.
- **jar-file** is the existing JAR file that is to be updated.
- **input-file(s)** is a space-delimited list of one or more files that you want to add to the JAR file.

Any files already in the archive having the same pathname as a file being added will be overwritten.

Running JAR-Packaged Software

Now that you have learned how to create JAR files, how do you actually run the code you packaged? Consider these scenarios:

- Your JAR file contains an applet that is to be run inside a browser.
- Your JAR file contains an application that is to be started from the command line.
- Your JAR file contains code that you want to use as an extension.

This section will cover the first two situations. A separate trail in the tutorial on the extension mechanism covers the use of JAR files as extensions.

Applets Packaged in JAR Files

To start any applet from an HTML file for running inside a browser, you use the applet tag. For more information, see the Java Applets lesson. If the applet is bundled as a JAR file, the only thing you need to do differently is to use the archive parameter to specify the relative path to the JAR file.

JAR Files as Applications

You can run JAR packaged applications with the Java launcher (java command). The basic command is:

```
java -jar jar-file
```

The **`-jar`** flag tells the launcher that the application is packaged in the JAR file format. You can only specify one JAR file, which must contain all of the application-specific code.

Before you execute this command, make sure that the runtime environment has information about which class within the JAR file is the application's entry point.

To indicate which class is the application's **entry point**, you must add a Main-Class header to the JAR file's manifest. The header takes the form:

```
Main-Class: classname
```

The header's value, classname, is the name of the class that is the application's entry point. When the Main-Class is set in the manifest file, you can run the application from the command line.

```
java -jar app.jar
```

To run the application from the JAR file that is in another directory, you must specify the path of that directory:

```
java -jar path/app.jar
```

```
can@can-System-Product-Name:~/Desktop$ java -jar alo.jar
no main manifest attribute, in alo.jar
```

Working with Manifest Files: The Basics

JAR files support a wide range of functionality, including electronic signing, version control, package sealing, and others. What gives a JAR file this versatility? The answer is the JAR file's manifest.

The manifest is a special file that can contain information about the files packaged in a JAR file. By tailoring this "meta" information that the manifest contains, you enable the JAR file to serve a variety of purposes.

- Understanding the Default Manifest
- Setting an Application's Entry Point
- Setting Package Version Information
- Enhancing Security with Manifest Attributes

Modifying a Manifest File
Adding Classes to the JAR File's Classpath
Sealing Packages within a JAR File

Understanding the Default Manifest

When you create a JAR file, it automatically receives a default manifest file. There can be only one manifest file in an archive, and it **always** has the pathname META-INF/MANIFEST.MF

When you create a JAR file, the default manifest file simply contains the following:

```
Manifest-Version: 1.0
Created-By: 1.7.0_06 (Oracle Corporation)
```

These lines show that a manifest's entries take the form of **"header: value"** pairs. The name of a header is separated from its value by a colon. The default manifest conforms to version 1.0 of the manifest specification and was created by the 1.7.0_06 version of the JDK.

The manifest can also contain information about the other files that are packaged in the archive. Exactly what file information should be recorded in the manifest depends on how you intend to use the JAR file. The default manifest makes no assumptions about what information it should record about other files.

Modifying a Manifest File

You use the **m** command-line option to add custom information to the manifest during creation of a JAR file. This section describes the m option.

You can enable special JAR file functionality, such as **package sealing**, by modifying the default manifest. Typically, modifying the default manifest involves adding special-purpose headers to the manifest that allow the JAR file to perform a particular desired function.

To modify the manifest, **you must first** prepare a text file containing the information you wish to add to the manifest. You then use the Jar tool's m option to add the information in your file to the manifest.

Warning: The text file from which you are creating the manifest must end with a new line or carriage return. The last line will not be parsed properly if it does not end with a new line or carriage return.

The basic command has this format:

```
jar cfm jar-file manifest-addition input-file(s)
```

- The **m** option indicates that you want to merge information from an existing file into the manifest file of the JAR file you're creating.
- **manifest-addition** is the name (or path and name) of the existing text file whose contents you want to add to the contents of JAR file's manifest.

Note: The contents of the manifest must be encoded in UTF-8.

Setting an Application's Entry Point

If you have an application bundled in a JAR file, you need some way to indicate which class within the JAR file is your application's entry point. You provide this information with the Main-Class header in the manifest, which has the general form:

```
Main-Class: classname
```

After you have set the Main-Class header in the manifest, you then run the JAR file using the following form of the java command

```
java -jar JAR-name
```

Warning: The text file must end with a new line or carriage return.

We first create a text file named **text** with the following contents

```
Main-Class: UseMe
```

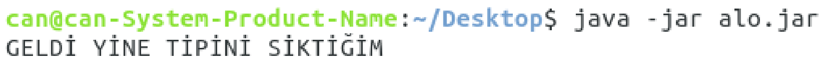
We then create a JAR file named alo.jar by entering the following command:

```
jar cfm alo.jar text UseMe.class
```

This creates the JAR file with a manifest with the following contents:

```
Manifest-Version: 1.0
Created-By: 1.8.0_171 (Oracle Corporation)
Main-Class: UseMe
```

When you run the JAR file with the following command, the main method of MyClass executes:



```
can@can-System-Product-Name:~/Desktop$ java -jar alo.jar
GELDİ YİNE TİPİNİ SİKTİĞİM
```

Setting an Entry Point with the JAR Tool

The **'e'** flag (for 'entrypoint') creates or overrides the manifest's Main-Class attribute. It can be used while creating or updating a JAR file. Use it to specify the application entry point without editing or creating the manifest file. For example, this command creates app.jar where the Main-Class attribute value in the manifest is set to MyApp:

```
jar cfe app.jar MyApp MyApp.class
```

If the entrypoint class name is in a package it may use a '.' (dot) character as the delimiter. For example, if **Main.class** is in a package called **foo** the entry point can be specified in the following ways:

```
jar cfe Main.jar foo.Main foo/Main.class
```

After that you can directly invoke this application by running the following command:

```
java -jar app.jar
```

Adding Classes to the JAR File's Classpath

You may need to reference classes in other JAR files from within a JAR file. For example, in a typical situation an applet is bundled in a JAR file whose manifest references a different JAR file (or several different JAR files) that serves as utilities for the purposes of that applet.

You specify classes to include in the **Class-Path** header field in the manifest file of an applet or application. The Class-Path header takes the following form:

```
Class-Path: jar1-name jar2-name directory-name/jar3-name
```

jar1-name jar2-name should be in the same dir.

By using the Class-Path header in the manifest, you can avoid having to specify a long **-classpath** flag when invoking Java to run the your application.

- **Note**: The Class-Path header points to classes or JAR files on the local network, not JAR files within the JAR file or classes accessible over Internet protocols. To load classes in JAR files within a JAR file into the class path, you must write custom code to load those classes. For example, if MyJar.jar contains another JAR file called MyUtils.jar, you cannot use the Class-Path header in MyJar.jar's manifest to load classes in MyUtils.jar into the class path.

Setting Package Version Information

You may need to include package version information in a JAR file's manifest. You provide this information with the following headers in the manifest:

Headers in a manifest

Header	Definition
`Name`	The name of the specification.
`Specification-Title`	The title of the specification.
`Specification-Version`	The version of the specification.
`Specification-Vendor`	The vendor of the specification.
`Implementation-Title`	The title of the implementation.
`Implementation-Version`	The build number of the implementation.
`Implementation-Vendor`	The vendor of the implementation.

One set of such headers can be assigned to each package. The versioning headers should appear directly beneath the Name header for the package. This example shows all the versioning headers:

```
Name: java/util/
Specification-Title: Java Utility Classes
Specification-Version: 1.2
Specification-Vendor: Example Tech, Inc.
Implementation-Title: java.util
Implementation-Version: build57
Implementation-Vendor: Example Tech, Inc.
```

We then create a JAR file named `MyJar.jar` by entering the following command:

```
jar cfm MyJar.jar Manifest.txt MyPackage/*.class
```

This creates the JAR file with a manifest with the following contents:

```
Manifest-Version: 1.0
Created-By: 1.7.0_06 (Oracle Corporation)
Name: java/util/
Specification-Title: Java Utility Classes
Specification-Version: 1.2
Specification-Vendor: Example Tech, Inc.
Implementation-Title: java.util
Implementation-Version: build57
Implementation-Vendor: Example Tech, Inc.
```

Sealing Packages within a JAR File

Packages within JAR files can be optionally sealed, which means that all classes defined in that package must be archived in the same JAR file. You might want to seal a package, for example, to ensure version consistency among the classes in your software.

You seal a package in a JAR file by adding the Sealed header in the manifest, which has the general form: The value myCompany/myPackage/ is the name of the package to seal. Note that the package name **must end** with a "/".

```
Name: myCompany/myPackage/
Sealed: true
```

An Example We want to seal two packages firstPackage and secondPackage in the JAR file MyJar.jar. We first create a text file named Manifest.txt with the following contents:

```
Name: myCompany/firstPackage/
Sealed: true

Name: myCompany/secondPackage/
Sealed: true
```

We then create a JAR file named MyJar.jar by entering the following command: **jar cfm MyJar.jar Manifest.txt MyPackage/*.class** This creates the JAR file with a manifest with the following contents:

```
Manifest-Version: 1.0
Created-By: 1.7.0_06 (Oracle Corporation)
Name: myCompany/firstPackage/
Sealed: true
Name: myCompany/secondPackage/
Sealed: true
```

Sealing JAR Files

If you want to guarantee that all classes in a package come from the same code source, use **JAR sealing**. A sealed JAR specifies that all packages defined by that JAR are sealed unless overridden on a per-package basis.

To seal a JAR file, use the Sealed manifest header with the value true. For example, **Sealed: true** specifies that all packages in this archive are sealed unless explicitly overridden for particular packages with the Sealed attribute in a manifest entry.

Enhancing Security with Manifest Attributes

The following JAR file manifest attributes are available to help ensure the security of your **applet** or **Java Web Start application**.(not so relevant) Only the Permissions attribute is required.

- The **Permissions** attribute is used to ensure that the application requests only the level of permissions that is specified in the applet tag or JNLP file used to invoke the application. Use this attribute to help prevent someone from re-deploying an application that is signed with your certificate and running it at a different privilege level.

  This attribute is required in the manifest for the main JAR file. See Permissions Attribute in the Java Platform, Standard Edition Deployment Guide for more information.
- The **Codebase** attribute is used to ensure that the code base of the JAR file is restricted to specific domains. Use this attribute to prevent someone from re-deploying your application on another website for malicious purposes. See Codebase Attribute in the Java Platform, Standard Edition Deployment Guide for more information.
- The **Entry-Point** attribute is used to identify the classes that are allowed to be used as entry points to your RIA. Use this attribute to prevent unauthorized code from being run from other available entry points in the JAR file. See Entry-Point Attribute in the Java Platform, Standard Edition Deployment Guide for more information.
- The **Trusted-Only** attribute is used to prevent untrusted components from being loaded.

Signing and Verifying JAR Files

You can optionally sign a JAR file with your electronic "signature." Users who verify your signature can grant your JAR-bundled software security privileges that it wouldn't ordinarily have. Conversely, you can verify the signatures of signed JAR files that you want to use.

Understanding Signing and Verification

You digitally sign a file for the same reason you might sign a paper document with pen and ink -- to let readers know that you wrote the document, or at least that the document has your approval.

When you sign a letter, for example, everyone who recognizes your signature can confirm that you wrote the letter. Similarly when you digitally sign a file, anyone who "recognizes" your digital signature knows that the file came from you. **The process of "recognizing" electronic signatures** is called **verification**.

When the JAR file is signed, you also have the option of time stamping the signature. Similar to putting a date on a paper document, time stamping the signature identifies when the JAR file was signed. The time stamp can be used to verify that the certificate used to sign the JAR file was valid at the time of signing.

The ability to sign and verify files is an important part of the Java platform's security architecture. Security is controlled by the security policy that's in force at runtime. You can configure the policy to grant security privileges to applets and to applications. For example, you could grant permission to an applet to perform normally forbidden operations such as reading and writing local files or running local executable programs. If you have downloaded some code that's signed by a trusted entity, you can use that fact as a criterion in deciding which security permissions to assign to the code.

Once you (or your browser) have verified that an applet is from a trusted source, you can have the platform relax security restrictions to let the applet perform operations that would ordinarily be forbidden. A trusted applet can have freedoms as specified by the policy file in force.

The Java platform enables signing and verification by **using special numbers** called **public** and **private keys**. Public keys and private keys come in pairs, and they play complementary roles.

The private key is the electronic "pen" with which you can sign a file. As its name implies, your private key is known only to you so that no one else can "forge" your signature. A file signed with your private key can be verified only by the corresponding public key.

Public and private keys alone, however, **aren't enough to truly verify a signature**. Even if you've verified that a signed file contains a matching key pair, you still need some way to confirm that the public key actually comes from the signer that it purports to come from.

One more element, therefore, is required to make signing and verification work. That additional element is the **certificate** that the signer includes in a signed JAR file. A certificate is a digitally signed statement from a **recognized certification authority** that indicates who owns a particular public key. Certification authorities are entities (typically firms specializing in digital security) that are trusted throughout the industry to sign and issue certificates for keys and their owners. In the case of signed JAR files, the certificate indicates **who owns the public key contained in the JAR file**.

When you sign a JAR file your public key is placed inside the archive along with an associated certificate so that it's easily available for use by anyone wanting to verify your signature.

To summarize digital signing:

- The signer signs the JAR file using a private key.
- The corresponding public key is placed in the JAR file, together with its certificate, so that it is available for use by anyone who wants to verify the signature.

Digests and the Signature File

When you sign a JAR file, each file in the archive is given a digest entry in the archive's manifest. Here's an example of what such an entry might look like:

```
Name: test/classes/ClassOne.class
SHA1-Digest: TD1GZt8G11dXY2p4olSZPc5Rj64=
```

The digest values are hashes or encoded representations of the contents of the files as they were at the time of signing. <u>A file's digest will change if and only if the file itself changes</u>.

When a JAR file is signed, a signature file is automatically generated and placed in the JAR file's META-INF directory, the same directory that contains the archive's manifest. Signature files have filenames with an **.SF extension**. Here is an example of the contents of a signature file:

```
Signature-Version: 1.0
SHA1-Digest-Manifest: h1yS+K9T7DyHtZrtI+LxvgqaMYM=
Created-By: 1.7.0_06 (Oracle Corporation)

Name: test/classes/ClassOne.class
SHA1-Digest: fcav7ShIG6i86xPepmitOVo4vWY=

Name: test/classes/ClassTwo.class
SHA1-Digest: xrQem9snnPhLySDiZyclMlsFdtM=

Name: test/images/ImageOne.gif
SHA1-Digest: kdHbE7kL9ZHLgK7akHttYV4XIa0=

Name: test/images/ImageTwo.gif
SHA1-Digest: mF0D5zpk68R4oaxEqoS9Q7nhm60=
```

As you can see, the signature file contains digest entries for the archive's files that look similar to the digest-value entries in the manifest. However, while the digest values in the manifest are computed from the files themselves, the digest values in the signature file are computed from the corresponding entries in the manifest. Signature files also contain a digest value for the entire manifest (see the SHA1-Digest-Manifest header in the above example).

When a signed JAR file is being verified, the digests of each of its files are re-computed and compared with the digests recorded in the manifest to ensure that the contents of the JAR file haven't changed since it was signed. As an additional check, digest values for the manifest file itself are re-computed and compared against the values recorded in the signature file.

The Signature Block File

In addition to the signature file, a signature block file is automatically placed in the META-INF directory when a JAR file is signed. Unlike the manifest file or the signature file, signature block files are not human-readable.

The signature block file contains two elements essential for verification:

- The digital signature for the JAR file that was generated with the signer's private key
- The certificate containing the signer's public key, to be used by anyone wanting to verify the signed JAR file

Signature block filenames typically will have a **.DSA extension** indicating that they were created by the default **Digital Signature Algorithm**. Other filename extensions are possible if keys associated with some other standard algorithm are used for signing.

Signing JAR Files

You use the JAR Signing and Verification Tool to sign JAR files and time stamp the signature. You invoke the JAR Signing and Verification Tool by using the **jarsigner** command, so we'll refer to it as "Jarsigner" for short.

To sign a JAR file, you must first have a private key. <u>Private keys and their associated public-key certificates are stored in password-protected databases called</u> **keystores**. A keystore can hold the keys of many potential signers. Each key in the keystore can be identified by an alias which is typically the name of the signer who owns the key. The key belonging to Rita Jones might have the alias "rita", for example.

The basic form of the command for signing a JAR file is **jarsigner jar-file alias**

The Jarsigner tool will prompt you for the passwords for the keystore and alias.

This basic form of the command assumes that the keystore to be used is in a file named .keystore in your home directory. It will create signature and signature block files with names x.SF and x.DSA respectively, where x is the first eight letters of the alias, all converted to upper case. This basic command will overwrite the original JAR file with the signed JAR file.

In practice, you might want to use one or more of the command options that are available. For example, time stamping the signature is encouraged so that any tool used to deploy your application can verify that the certificate used to sign the JAR file was valid at the time that the file was signed. A warning is issued by the Jarsigner tool if a time stamp is not included.

Jarsigner Command Options

Option	Description
`-keystore` *url*	Specifies a keystore to be used if you don't want to use the `.keystore` default database.
`-sigfile` *file*	Specifies the base name for the .SF and .DSA files if you don't want the base name to be taken from your alias. *file* must be composed only of upper case letters (A-Z), numerals (0-9), hyphen (-), and underscore (_).
`-signedjar` *file*	Specifies the name of the signed JAR file to be generated if you don't want the original unsigned file to be overwritten with the signed file.
`-tsa` *url*	Generates a time stamp for the signature using the Time Stamping Authority (TSA) identified by the URL.
`-tsacert` *alias*	Generates a time stamp for the signature using the TSA's public key certificate identified by *alias*.
`-altsigner` *class*	Indicates that an alternative signing mechanism be used to time stamp the signature. The fully-qualified class name identifies the class used.
`-altsignerpath` *classpathlist*	Provides the path to the class identified by the `altsigner` option and any JAR files that the class depends on.

Example Let's look at a couple of examples of signing a JAR file with the Jarsigner tool. In these examples, we will assume the following:

- Your alias is "johndoe".
- The keystore you want to use is in a file named "mykeys" in the current working directory.
- The TSA that you want to use to time stamp the signature is located at `http://tsa.url.example.com`.

Under these assumptions, you could use this command to sign a JAR file named app.jar:

```
jarsigner -keystore mykeys -tsa http://tsa.url.example.com app.jar johndoe
```

You will be prompted to enter the passwords for both the keystore and your alias. Because this command doesn't make use of the -sigfile option, the .SF and .DSA files it creates would be named JOHNDOE.SF and JOHNDOE.DSA. Because the command doesn't use the -signedjar option, the resulting signed file will overwrite the original version of app.jar.

Let's look at what would happen if you used a different combination of options:

```
jarsigner -keystore mykeys -sigfile SIG -signedjar SignedApp.jar
          -tsacert testalias app.jar johndoe
```

The signature and signature block files would be named SIG.SF and SIG.DSA, respectively, and the signed JAR file SignedApp.jar would be placed in the current directory. The original unsigned JAR file would remain unchanged. Also, the signature would be time stamped with the TSA's public key certificate identified as testalias.

Verifying Signed JAR Files

Typically, verification of signed JAR files will be the responsibility of your Java™ Runtime Environment. Your browser will verify signed applets that it downloads. Signed applications invoked with the **-jar** option of the interpreter **will be verified by the runtime environment**.

However, you can verify signed JAR files yourself by using the **jarsigner** tool. You might want to do this, for example, to test a signed JAR file that you've prepared.

The basic command to use for verifying a signed JAR file is: **jarsigner -verify jar-file**

This command will verify the JAR file's signature and ensure that the files in the archive haven't changed since it was signed. You'll see the following message if the verification is successful: **jar verified**.