IBM. English Sign in (or register)

developerWorks. Technical topics Evaluation software Community Events

JAR files revealed

Explore the power of the JAR file format

Pagadala Suresh (pjsuresh@in.ibm.com), Software Engineer, IBM Global Services India
Palaniyappan Thiagarajan (tpalaniy@in.ibm.com), Software Engineer, IBM Global Services India

Summary: Most Java programmers are familiar with basic operations on JAR files. But few programmers are aware of the *power* of the JAR file format. In this article, the authors explore the many features and benefits of the JAR format, including packaging, executable JAR files, security, and indexing.

Date: 09 Oct 2003
Level: Introductory
Also available in: Chinese

Activity: 100134 views
Comments: 2 (View | Add comment - Sign in)

Average rating (173 votes)
Rate this article

- **Show articles and other content related to my search: jar applet**

What is a JAR file?

The JAR file format is based on the popular ZIP file format, and is used for aggregating many files into one. Unlike ZIP files, JAR files are used not only for archiving and distribution, but also for deployment and encapsulation of libraries, components, and plug-ins, and are consumed directly by tools such as compilers and JVMs. Special files contained in the JAR, such as manifests and deployment descriptors, instruct tools how a particular JAR is to be treated.

A JAR file might be used:

- For distributing and using class libraries
- As building blocks for applications and extensions
- As deployment units for components, applets, or plug-ins
- For packaging auxiliary resources associated with components

The JAR file format provides many benefits and features, many of which are not provided with a traditional archive format such as ZIP or TAR. These include:

- **Security.** You can digitally sign the contents of a JAR file. Tools that recognize your signature can then optionally grant your software security privileges it wouldn't otherwise have, and detect if the code has been tampered with.
- **Decreased download time.** If an applet is bundled in a JAR file, the applet's class files and associated resources can be downloaded by a browser in a single HTTP transaction, instead of opening a new connection for each file.
- **Compression.** The JAR format allows you to compress your files for efficient storage.
- **Transparent platform extension.** The Java Extensions Framework provides a means by which you can add functionality to the Java core platform, which uses the JAR file for packaging of extensions. (Java 3D and JavaMail are examples of extensions developed by Sun.)
- **Package sealing.** Packages stored in JAR files can be optionally *sealed* to enforce version consistency and security. Sealing a package 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.

Compressed and uncompressed JARs

The `jar` tool (see The `jar` tool for details) compresses files by default. Uncompressed JAR files can generally be loaded more quickly than compressed JAR files, because the need to decompress the files during loading is eliminated, but download time over a network may be longer for uncompressed files.

The META-INF directory

Most JAR files contain a META-INF directory, which is used to store package and extension configuration data, such as security and versioning information. The following files or directories in the META-INF directory are recognized and interpreted by the Java 2 platform for configuring applications, extensions, and class loaders:

- **MANIFEST.MF.** The *manifest file* defines the extension- and package-related data.
- **INDEX.LIST.** This file is generated by the new `-i` option of the `jar` tool and contains location information for packages defined in an application or extension. It is part of the JarIndex implementation and used by class loaders to speed up the class loading process.
- ***xxx*.SF.** This is the signature file for the JAR file. The placeholder *xxx* identifies the signer.
- ***xxx*.DSA.** The signature block file associated with the signature file stores the public signature used to sign the JAR file.

The `jar` tool

To perform basic tasks with JAR files, you use the Java Archive Tool (`jar` tool) provided as part of the Java Development Kit. You invoke the `jar` tool with the `jar` command. Table 1 shows some common applications:

Table 1. Common usages of the `jar` tool

Function	Command
Creating a JAR file from individual files	jar cf jar-file input-file...
Creating a JAR file from a directory	jar cf jar-file dir-name
Creating an uncompressed JAR file	jar cf0 jar-file dir-name
Updating a JAR file	jar uf jar-file input-file...
Viewing the contents of a JAR file	jar tf jar-file
Extracting the contents of a JAR file	jar xf jar-file
Extracting specific files from a JAR file	jar xf jar-file archived-file...
Running an application packaged as an executable JAR file	java -jar app.jar

Executable JARs

An *executable jar* file is a self-contained Java application stored in a specially configured JAR file, which can be executed directly by the JVM without having to first extract the files or set up a class path. To run an application stored in a non-executable JAR, you have to add it to your class path and invoke the application's main class by name. But by using executable JAR files, we can run an application without extracting it or needing to know the main entry point. Executable JARs facilitate easy distribution and execution of Java applications.

Creating executable JARs

Creating an executable JAR is easy. You begin by placing all your application code in a single directory. Let's say the main class in your application is `com.mycompany.myapp.Sample`. You want to create a JAR file that contains the application code and identifies the main class. To do this, create a file called `manifest` somewhere (not in your application directory), and add the following line to it:

```
Main-Class: com.mycompany.myapp.Sample
```

Then, create the JAR file like this:

```
jar cmf manifest ExecutableJar.jar application-dir
```

That's all there is to it -- now the JAR file ExecutableJar.jar can be executed using `java -jar`.

An executable JAR must reference all the other dependent JARs it requires through the `Class-Path` header of the manifest file. The environment variable CLASSPATH and any class path specified on the command line is ignored by the JVM if the `-jar` option is used.

Launching executable JARs

Now that we've packaged our application into an executable JAR called ExecutableJar.jar, we can launch the application directly from the file using the following command:

```
java -jar ExecutableJar.jar
```

Package sealing

Sealing a package within a JAR file means that all classes defined in that package must be found in the same JAR file. This allows the package author to enforce version consistency among packaged classes. Sealing also provides a security measure to detect code tampering.

To seal a package, add a `Name` header for the package, followed by a `Sealed` header with value "true" to the JAR manifest file. Just as with executable JARs, you can seal a JAR by specifying a manifest file with the appropriate header elements when the JAR is created, as shown here:

```
Name: com/samplePackage/
Sealed: true
```

The `Name` header identifies the package's relative pathname. It ends with a "/" to distinguish it from a filename. Any headers following a `Name` header, without any intervening blank lines, apply to the file or package specified in the `Name` header. In the example above, because the `Sealed` header occurs after the `Name` header without intervening blank lines, the `Sealed` header will be interpreted as applying only to the package `com/samplePackage`.

If you try to load a class in a sealed package from another source other than the JAR file in which the sealed package lives, the JVM will throw a `SecurityException`.

Packaging for extensions

Extensions add functionality to the Java platform, and an extensions mechanism is built into the JAR file format. The Extensions mechanism allows JAR files to specify other required JAR files via the `Class-Path` headers in the manifest file.

Let's say that extension1.jar and extension2.jar are two JAR files in the same directory, with the manifest of extension1.jar containing the following header:

```
Class-Path: extension2.jar
```

This header indicates that the classes in extension2.jar serve as *extension classes* for purposes of the classes in extension1.jar. The classes in extension1.jar can invoke classes in extension2.jar without extension2.jar having to be part of the class path.

The JVM effectively automatically adds JARs referenced in a `Class-Path` header to the class path when loading a JAR that uses the extension mechanism. However, the extension JAR path is interpreted as a relative path, so in general the extension JAR must be stored in the same directory as the JAR referencing it.

For example, assume the class `ExtensionClient`, which references class `ExtensionDemo`, is bundled in a JAR file called ExtensionClient.jar, and that the class `ExtensionDemo` is bundled in ExtensionDemo.jar. In order for ExtensionDemo.jar to be treated as an extension, ExtensionDemo.jar must be listed in the `Class-Path` header in ExtensionClient.jar's manifest, as follows:

```
Manifest-Version: 1.0
Class-Path: ExtensionDemo.jar
```

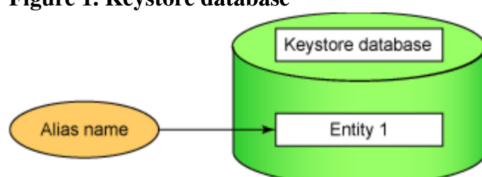
The value of the `Class-Path` header in this manifest is ExtensionDemo.jar with no path specified, indicating that ExtensionDemo.jar is located in the same directory as the ExtensionClient JAR file.

Security in JAR files

A JAR file can be signed by using the `jarsigner` tool or directly through the `java.security` API. A signed JAR file is exactly the same as the original JAR file, except that its manifest is updated, and two additional files are added to the META-INF directory, a signature file and a signature block file.

A JAR file is signed using a certificate stored in the *Keystore* database. Certificates stored in the keystore are protected with a password, which must be provided to the `jarsigner` tool to sign a JAR file.

Figure 1. Keystore database



Each signer of a JAR is represented by a signature file with the extension .SF within the META-INF directory of the JAR file. The format of the file is similar to the manifest file -- a set of RFC-822 headers. As shown below, it consists of a main section, which includes information

supplied by the signer but not specific to any particular JAR file entry, followed by a list of individual entries which also must be present in the manifest file. To validate a file from a signed JAR, a digest value in the signature file is compared against a digest calculated against the corresponding entry in the JAR file.

Listing 1. Manifest and signature files in signed JARs

```
Contents of signature file META-INF/MANIFEST.MF

Manifest-Version: 1.0
Created-By: 1.3.0 (Sun Microsystems Inc.)

Name: Sample.java
SHA1-Digest: 3+DdYW8INICtyG8ZarHlFxX0W6g=

Name: Sample.class
SHA1-Digest: YJ5yQHBZBJ3SsTNcHJFqUkfWEmI=

Contents of signature file META-INF/JAMES.SF

Signature-Version: 1.0
SHA1-Digest-Manifest: HBstZOJBuuTJ6QMIdB90T8sjaOM=
Created-By: 1.3.0 (Sun Microsystems Inc.)

Name: Sample.java
SHA1-Digest: qipMDrkurQcKwnyIlI3Jtrnia8Q=

Name: Sample.class
SHA1-Digest: pT2DYby8QXPcCzv2NwpLxd8p4G4=
```

Digital signatures

A digital signature is a signed version of the `.SF` signature file. Digital signature files are binary files and have the same filename as the `.SF` file but a different extension. The extension varies depending on the type of digital signature -- RSA, DSA, or PGP -- and on the type of certificate used to sign the JAR.

Keystore

To sign a JAR file, you must first have a private key. Private keys and their associated public-key certificates are stored in password-protected databases called `keystores`. The JDK contains tools for creating and modifying keystores. Each key in the keystore can be identified by an alias, which is typically the name of the signer who owns the key.

All keystore entries (key and trusted certificate entries) are accessed with unique aliases. An alias is specified when you add an entity to the keystore using the `keytool -genkey` command to generate a key pair (public and private key). Subsequent `keytool` commands must use this same alias to refer to the entity.

For example, to generate a new public/private key pair with the alias "james" and wrap the public key into a self-signed certificate, you would use with the following command:

```
keytool -genkey -alias james -keypass jamespass
        -validity 80 -keystore jamesKeyStore
        -storepass jamesKeyStorePass
```

This command sequence specifies an initial password of "jamespass" required by subsequent commands to access the private key associated with the alias "james" in the keystore "jamesKeyStore." If the keystore "jamesKeyStore" does not exist, `keytool` will automatically create it.

The jarsigner tool

The `jarsigner` tool uses keystore to generate or verify digital signatures for JAR files.

Assuming you've created the keystore "jamesKeyStore" as in the example above, and it contains a key with alias "james," you can sign a JAR file with the following command:

```
jarsigner -keystore jamesKeyStore -storepass jamesKeyStorePass
          -keypass jamespass -signedjar SSample.jar Sample.jar james
```

This command fetches the key whose alias is "james" and whose password is "jamespass" from the keystore named "jamesKeyStore" with the password "jamesKeyStorePass," and signs the Sample.jar file, creating a signed JAR, SSample.jar.

The `jarsigner` tool can also verify a signed JAR file; this operation is considerably simpler than signing the JAR file. Just execute the following command:

```
jarsigner -verify SSample.jar
```

If the signed JAR file has not been tampered with, the `jarsigner` tool will tell you the JAR has been verified; otherwise, it will throw a `SecurityException` indicating which files could not be verified.

JARs can also be signed programmatically using the `java.util.jar` and `java.security` APIs. Alternatively, you can use tools such as Netscape Object Signing Tool.

You can also sign JARs programmatically using the `java.util.jar` and `java.security` APIs. (See Resources for details). Alternatively, you can use tools such as the Netscape Object Signing Tool.

JAR indexing

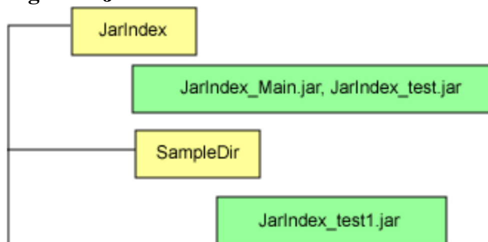
If an application or applet is bundled into multiple JAR files, the class loader uses a simple linear search algorithm to search each element of the class path, which may entail the class loader downloading and opening many JAR files until the class or resource is found. If the class loader tries to find a nonexistent resource, all the JAR files within the application or applet will have to be downloaded. For large network applications and applets this could result in slow start up, sluggish response, and wasted network bandwidth.

Since JDK 1.3, the JAR file format has supported indexing to optimize the process of searching for classes in network applications, especially applets. The JarIndex mechanism collects the contents of all the JAR files defined in an applet or application and stores the information in an index file in the first JAR file. After the first JAR file is downloaded, the applet class loader will use the collected content information for efficient downloading of JAR files. This directory information is stored in a simple text file named INDEX.LIST in the META-INF directory of the root JAR file.

Creating a JarIndex

You can create a JarIndex by specifying the `-i` option to the `jar` command. Suppose we have a directory structure as depicted in the following diagram:

Figure 2. JarIndex



You would use the following command to create an index file for JarIndex_Main.jar, JarIndex_test.jar, and JarIndex_test1.jar:

```
jar -i JarIndex_Main.jar JarIndex_test.jar SampleDir/JarIndex_test1.jar
```

The INDEX.LIST file has a simple format, containing the names of the packages or classes contained in each JAR file indexed, as shown in Listing 2:

Listing 2. Example JarIndex INDEX.LIST file

```
JarIndex-Version: 1.0

JarIndex_Main.jar
sp

JarIndex_test.jar
Sample

SampleDir/JarIndex_test1.jar
org
org/apache
org/apache/xerces
org/apache/xerces/framework
org/apache/xerces/framework/xml4j
```

Summary

The JAR format is much more than an archive format; it has many features for improving the efficiency, security, and organization of Java

applications. Because these features are built into the core platform, including the compiler and classloader, developers can leverage the power of the JAR file format to simplify and improve their development and deployment processes.

Resources

- See the documentation for the command-line options of the jar utility.
- Raffi Krikorian offers assistance on programmatically signing a JAR file in this article from ONJava.
- The article "Java Web Start" (*developerWorks*, September 2001) describes how this technology allows applications to specify what JAR files they need and dynamically download them.
- You'll find hundreds of articles about every aspect of Java programming in the *developerWorks* Java technology zone.

About the authors

Pagadala J. Suresh is a Software Engineer for IBM Global Services India. His areas of expertise include Java technology, WebSphere Application Server, and WebSphere Studio Application Developer (WSAD), Ariba Buyer. He participated in the IBM Redbook program on WebSphere. Contact Pagadala at pjsuresh@in.ibm.com.

Palaniyappan Thiagarajan is a Software Engineer for IBM Global Services India in Bangalore, India. He is an IBM Certified Specialist for IBM WebSphere Application Server V3.5 and IBM DB2 UDB V7.1 Family Fundamentals. Contact Palaniyappan at tpalaniy@in.ibm.com.

Close [x]

developerWorks: Sign in

IBM ID:
Need an IBM ID?
Forgot your IBM ID?

Password:
Forgot your password?
Change your password

Keep me signed in.

By clicking **Submit**, you agree to the developerWorks terms of use.

Submit Cancel

The first time you sign into developerWorks, a **profile** is created for you. **Select information in your developerWorks profile is displayed to the public, but you may edit the information at any time**. Your first name, last name (unless you choose to hide them), and display name will accompany the content that you post.

All information submitted is secure.

Close [x]

Choose your display name

The first time you sign in to developerWorks, a profile is created for you, so you need to choose a display name. Your display name accompanies the content you post on developerWorks.

Please choose a display name between 3-31 characters. Your display name must be unique in the developerWorks community and should not be your email address for privacy reasons.

Display name: (Must be between 3 – 31 characters.)

By clicking **Submit**, you agree to the developerWorks terms of use.

Submit Cancel

All information submitted is secure.

Average rating (173 votes)

1 star 1 star
2 stars 2 stars
3 stars 3 stars
4 stars 4 stars
5 stars 5 stars

Submit

Add comment:

Sign in or register to leave a comment.

Note: HTML elements are not supported within comments.

Notify me when a comment is added1000 characters left

Post

Total comments (2)

hi Shettigar...

I m just putting one general document contents which may be useful to u as it contains only those information which we need frequently and for that I've given the demo example...

just go through this...

Main-Class: com.ngine.main.JPrinterFrame
Class-Path: lib\substance.jar lib\itext-1.3.jar lib\os-javapdf-itextsample.jar lib\PDFRenderer.jar lib\mysql-connector-java-5.0.4-bin.jar lib\calendrica.jar lib\junit.jar lib\scheduling.jar lib\calendrica.jar lib\itext-1.3.jar lib\junit.jar lib\mysql-connector-java-5.0.4-bin.jar lib\os-javapdf-itextsample.jar lib\PDFRenderer.jar lib\scheduling.jar com\ngine\res\access.properties

--> above lines must be put as they are with carriage return in the file named "manifest.txt". (with applicable changes)
--> Here Class-path contains all the jars and property files which are being used in your project.
--> Note that : in the cmd, you've to reach to the directory where your com folder lies that means inside your project folder.(OR from where your directory structure starts)
--> And with reference to the com, I've set above paths.(com,lib and threads are inside project folder inshort where we are standing)
--> To get more clear idea, refer to this : D:\Ravi\Ravi_projects\JavaPrinterApp__
--> Now run the following commands :

D:\Ravi\Ravi_projects\JavaPrinterApp__>jar -cvfm JavaPrinterApplication.jar manifest.txt com threads

D:\Ravi\Ravi_projects\JavaPrinterApp__>jar -tf JavaPrinterApp.jar

java -jar JavaPrinterApp.jar

Posted by **Ravie** on 28 August 2010

Report abuse

Its a nice article.

I have some problem now.. thing is i have many dependent jar files which is will in one directory and i have one runnable jar. in this case how do i reference them to runnable jar through reference?

thanks a lot!!

Posted by **Shettigar** on 23 May 2010

Report abuse

Print this page **Share this page** **Follow developerWorks**

About
Help
Contact us
Submit content
Feeds and apps
Newsletters
Report abuse
Terms of use
IBM privacy
IBM accessibility
Faculty
Students
Business Partners