# Java Programming Tutorial

# Java Archive (Jar)

TABLE OF CONTENTS (HIDE)



## 1. Java Archive (JAR) Files

Java Archive (JAR) is a platform-independent file format that allow you to compress and bundle multiple files associated with a Java application, applet, or WebStart application into a single file. JAR is based on the ZIP algorithm, and minic the Unix's tar (or tape archive) file format (e.g., `jar` and `tar` tools have the same command-line options).

JAR files provide the following functions:

1. Data compression (using the ZIP algorithm).
2. Ease in distribution: All files in a Java package can be placed in a single file to facilitate distribution. Furthermore, transfer one big file over the network instead of many small files is faster and more efficient because it involves less overhead.
3. Authentication: JAR file can be digitally signed by its author. You can authenticate the author of the JAR file by checking the signature and the author's digital certificate.

The Java Runtime (JRE) (or Java applications) can load classes from the JAR file directly, without un-jarring the file. Furthermore, the JAR files use the same file format as ZIP files, and can be openned and manipulated via ZIP programs such as WinZIP or WinRAR.

## 2. Creating JAR File

There are a few ways that you can create a JAR file:

1. Via the comand-line `jar` tool: JDK provides a command-line tool called "`jar.exe`", to create, manage and manipulate JAR files. The `jar` tool reference is available at http://docs.oracle.com/javase/7/docs/technotes/tools/windows/jar.html.
2. Via the "export" option of IDE such as Eclipse or NetBeans, in practice.

Needless to say, using IDE to create a JAR file is much more easier than the command-line `jar` tool.

## 2.1 Create JAR File using Command-Line "jar" Tool

Creating JAR file using command-line jar `tool` is clumpy! It is presented here for your basic understanding and completeness. In practice, you should use your IDE (such as Eclipse/NetBeans) to create JAR file instead (which are presented in the following sections).

### Syntax

To create a JAR file using the `jar` tool, issue the `jar` command (on CMD shell) with `'c'` option:

```
> jar cvf jarFile inputFileDir1 inputFileDir2 ...
```

where:

- `'c'` option indicates that you want to create a new JAR file;
- `'v'` option asks for *verbose* mode, which displays information messages;
- `'f'` specifies that the output go to a file specified in *`jarfile`* (instead of the standard output by default). Option `'f'` and *`jarfile`* are a pair.
- *`inputFileDir`* give the input filenames or directory names for the files to be jarred. Multiple names are separated by space. The name may contain wildcard `'*'`.

### Example

For example, suppose that "`images`" is an sub-directory under the current directory, the following command jar-up all the "`.class`" files of the current directory and the entire "`images`" sub-directory into "`hello.jar`". The `'v'` option instructs the tool to produce the information messages.

```
> jar cvf hello.jar *.class images
added manifest
adding: HelloJar$1.class(in = 893) (out= 520)(deflated 41%)
adding: HelloJar$2.class(in = 393) (out= 284)(deflated 27%)
adding: HelloJar.class(in = 1808) (out= 1014)(deflated 43%)
adding: images/(in = 0) (out= 0)(stored 0%)
adding: images/high.png(in = 978) (out= 983)(deflated 0%)
adding: images/muted.png(in = 839) (out= 844)(deflated 0%)
```

### Jarring-Up Files in Packages

Java classes are often placed in packages. To jar up all the classes of a package, you must provide the proper sub-directory structure (as depicted by the package name). Recall that package name with '.' are mapped to sub-directory, e.g., the class file `com.test.MyClass` is stored as `com\test\MyClass.class`.

For example, the following command jar-up all the classes in `mypackage` and the image sub-directory. Take note that the `jar` command should be issued from the project root directory, i.e., the base directory of the packages.

```
> jar cvf hello.jar mypackage\*.class images
added manifest
adding: mypackage/HelloJarInPackage$1.class(in = 1016) (out= 536)(deflated 47%)
```

```
adding: mypackage/HelloJarInPackage$2.class(in = 440) (out= 303)(deflated 31%)
adding: mypackage/HelloJarInPackage.class(in = 1931) (out= 1034)(deflated 46%)
adding: images/(in = 0) (out= 0)(stored 0%)
adding: images/high.png(in = 978) (out= 983)(deflated 0%)
adding: images/muted.png(in = 839) (out= 844)(deflated 0%)
```

Creating JAR file using `jar` tool with package is clumpy! I shall describe how to create a JAR file via an IDE (such as Eclipse/NetBeans) in the following sections.

You can use the `jar` tool to inspect and manipulate JAR file. However, it is much easier to use a graphical ZIP program (such as WinZIP or WinRAR) to inspect and manipulate JAR file. Try openning a JAR file using WinZIP or WinRAR or any ZIP tool.

## 2.2 Manifest

Many of the JAR functions, such as main-class specification, digital signing, version control, package sealing, are supported though a file called *manifest*. The manifest is a special file, called "`MANIFEST.MF`" under the "`META-INF`" sub-directory, that contains information about the files contained in a JAR file.

When you create a JAR file without an input manifest, it automatically receives a default manifest file (called "`META-INF\MANIFEST.MF`") which contains the following. Try openning a JAR file created in the previous section via WinZIP or WinRAR to view the manifest.

```
Manifest-Version: 1.0
Created-By: 1.7.0_03 (Oracle Corporation)
```

The entries in manifest take the form of "`name: values`" pair. The name and value are separated by a colon '`:`'. The name is also called the attribute.

Refer to JDK API Specification on the JAR for detailed syntax on the manifest [@ http://docs.oracle.com/javase/7/docs/technotes/guides/jar/jar.html#JAR_Manifest].

## 2.3 Creating an Executable JAR File with Manifest using "`jar`" tool

As mentioned, JRE can load classes from a JAR file directly without un-jarring the file. In fact, you can run a Java application directly from a JAR file. The JAR file, however, must contain a manifest with a header called `Main-Class` which specifies the main-class that contains the entry `main()` method. (Otherwsie, which class to lanuch the application?)

### Example

Let's create an exeutable JAR file.

First, prepare the following text file (to be used as input manifest) called "`hello.mf`" (using a text editor). This file specifies the main-class that contains the entry `main()` method for launching the application. Take note that the file MUST BE terminated with a blank line as shown.

```
Manifest-Version: 1.0
Main-Class: mypackage.HelloJarInPackage

```

Next, create the JAR file with the input manifest using the command-line `jar` tool:

```
> jar cvfm hello.jar hello.mf mypackage\*.class images
added manifest
adding: mypackage/HelloJarInPackage$1.class(in = 1016) (out= 536)(deflated 47%)
adding: mypackage/HelloJarInPackage$2.class(in = 440) (out= 303)(deflated 31%)
adding: mypackage/HelloJarInPackage.class(in = 1931) (out= 1034)(deflated 46%)
adding: images/(in = 0) (out= 0)(stored 0%)
adding: images/high.png(in = 978) (out= 983)(deflated 0%)
adding: images/muted.png(in = 839) (out= 844)(deflated 0%)
```

where option `'m'` specifies the inclusion of an input manifest. Take note that the `'m'` option comes after the `'f'` option, hence, the manifest file shall be placed after the output JAR file.

Use WinZIP or WinRAR to inspect the manifest `"META-INF\MANIFEST.MF"` created:

```
Manifest-Version: 1.0
Created-By: 1.7.0_03 (Oracle Corporation)
Main-Class: mypackage.HelloJarInPackage
```

To run the application directly from JAR file, invoke the JRE with option "-jar":

```
> java -jar Hello.jar
```

In windows, you can also double-click the JAR file to launch the application, proivded that the file type `".jar"` is associated with the JRE `"java.exe"`.

## 2.4 Creating JAR File in Eclipse 3.7

You can create a JAR file easily in Eclipse via the "Export" option.

1. Right-click on the project ⇒ "Export..." ⇒ Select "Java", "JAR File" ⇒ Next.
2. In the "JAR File Specification" dialog, "Select the resources to export" ⇒ in "Select the export destination", set the output jar-file's path and name ⇒ Next.
3. In the "JAR Packaging Options" dialog ⇒ Next.
4. In the "JAR Manifest Specification" dialog ⇒ If you wish to create an executable JAR, you can specify the main-class (which contains the entry `main()` method) by setting the "Main Class" field ⇒ Finish.

Try openning the JAR file created using ZIP tool such as WinZIP or WinRAR. Inspect the manifest at `"META-INF\MANIFEST.MF"`.

## 2.5 Creating JAR File in NetBeans 7

Simply "build" your project (Right-click on the project ⇒ "Build"), a JAR file for the proejct is created under the `"dist"` directory. Try openning the JAR file with a ZIP tool (such as WinZIP or WinRAR to inspect its content),

To set the main-class for an executable JAR file: Right-click on the project ⇒ "Properties" ⇒ Select "Run" ⇒ Set the "Main Class" field.

To include additional JAR files and Libraries: See article below.

**Reference**: Packaging and Deploying Desktop Java Applications @ http://netbeans.org/kb/articles/javase-deploy.html[TODO]

## 2.6 Creating JAR File with 3rd-Party Packages

A JAR file may contain other JAR files or Native Libraries, typically contain third-party API which you used in your program (e.g., JOGL, LWJGL etc), in a sub-directory "`lib`". This task can be easily managed from IDE such as Eclipse/NetBeans.

You need to include attribute "`Class-Path`" in the manifest to provide proper classpath to these JAR files.

[TODO]

# 3. Using JAR File for Deployment

## 3.1 Deploy as an Application

To deploy an application in JAR file, you need to create the JAR file with a manifest specifying its main-class, which contains the entry main() method to launch the application. Refer to "Creating JAR File" on how to include a `Main-Class` attribute into the JAR file.

## 3.2 Deploy as an Applet

To deploy an applet in JAR file, simply jar-up all the classes and the relevant resources, and provide an HTML file with an <applet> tag. For example,

```
<applet code="YourAppletMainClass"
        width="640"
        height="480"
        archive="YourJarFile">
</applet>
```

where attribute "`code`" specifies the applet class; and "`archive`" specifies the JAR file.

No "`Main-Class`" attribute is needed in the JAR file, as the main-class is identified in the `<applet>` tag's "`code`" attribute.

Read "Java Applet and WebStart App" for more details.

## 3.3 Deploy as a WebStart App (JDK 1.4)

To deploy your application as a Java WebStart application, jar-up all the classes and relevant resource and proivde an JNLP lanuching file. For example, suppose your JNLP file is called "`hello.jnlp`":

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <!-- JNLP File for WebStart Application -->
3  <jnlp spec="1.0+" codebase="." href="hello.jnlp">
4    <information>
5      <title>Your title</title>
6      <vendor>Your company</vendor>
7      <description>Your description</description>
8    </information>
9
10   <resources>
```

```
    <j2se version="1.4+" />
    <jar href="YourJarFile.jar" />
  </resources>

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

Again, no "`Main-Class`" attribute is needed in the JAR file, as the main-class is identified in the `<application-desc>` tag's "`main-class`" attribute.

Read "Java Applet and WebStart App" for more details.

## 3.4 Deploy as a JNLP Applet (JDK 1.6u14)

[TODO]

# 4. Signing and Verifying a JAR file

A JAR file author can digitally sign the JAR file for ownership authentication.

Public Key Infrastructure (PKI) is used for digital signature. PKI uses a pair of keys - a private key, which should be kept privately and securely by the owner, and a public key, which can be made available publicly. The private key is used to encrypt the file, which can be decrypted using the corresponding public key. Suppose that you manage to decrypt using Alice's public key, the information must have been encrypted using Alice's private key. Since only Alice (but no one else) has the private key, the signer must be Alice, the file must have come from Alice, and Alice cannot deny the ownership.

There is a problem though: How to be certain that the public key belongs to Alice? A Certification Authority (CA) issues a digital certificate to Alice, which contains the public key of Alice. This digital certificate is in turns signed using the CA's private key, and requires CA's public key to verify and authenticate. CA's public key is usually pre-configured into the operating system.

When you sign a JAR file, your certificate (contains the public key) is placed inside the JAR file, so that anyone can verify your signature.

The digest (or hash) is computed for every file in the JAR and included inside the manifest, e.g.,

```
Name: Hello.class
SHA1-Digest: (a 160-bit hash value for the file)
```

When the JAR file is signed, a signature file with extension "`.SF`" is created in the directory `META-INF`. The digest value of each file is signed (or encrypted) using the signer's private key. e.g.,

```
Name: Hello.class
SHA1-Digest: (digest value encrypted using signer's private key)
```

In addition to the signature file, a signature block file (with extension "`.DSA`" for the default Digital Signature Algorithm) is created in directory `META-INF`. This file includes the digital signature for the JAR file, and the digital certificate and the public key of the signer.

### Signing JAR files with a Test Certificate

JDK provides utilities "`keytool`" for managing public/private keys and digital certificates and "`jarsigner`" for signing the JAR files. The step for signing a JAR file with a test certificate is as follows:

1. Use `keytool` to generate a pair of public and private keys (`-genkey` option) for a certain person called *`provideName`* (`-alias` option) and keep in a new "`keystore`" called *`providerKeyStore`* (`-keystore` option) as follows:

```
> keytool -genkey -keystore providerKeyStore -alias providerName
```

2. Agian, use `keytool` to create a self-signed certificate (`-selfcert` option) for the person *`providerName`* (`-alias` option), whose public and private keys are kept in the *`providerKeyStore`* (`-keystore` option). A digital certificate contains a person's public key, which is signed by a Certificate Authority (CA)'s private key. A self-signed certificate contains a person's public key, which is signed by his own private key.

```
> keytool -selfcert -alias providerName -keystore providerKeyStore
```

   To list the contents of keystore, you can issue:

```
> keytool -list -keystore providerKeyStore
```

3. Finally, use the "`jarsigner`" utility to sign a JAR file "*`jarfile`*`.jar`" with the test certificate of *`providerName`*, which is kept in *`providerKeyStore`* (`-keystore`) as follows:

```
> jarsigner -keystore providerKeyStore jarfile.jar providerName
```

4. To verify a JAR file, you can use `-verify` option of `jarsigner`:

```
> jarsiger -verify jar-file
```

## 5. More on `jar` Tool

As mentioned, you can easily inspect JAR files using ZIP programs such as WinZIP or WinRAR. Nonetheless, you can also use the command-line jar tool, which is pretty clumpy, but described here for completeness!

### List Table of Content of JAR File (`t`)

To *list* the table of contents of a jar file, use option `'t'`, as shown:

```
> jar tvf jarFile
```

The `'v'` (verbose) option displays the information messages; the `'f'` (file) option specifies the name of the jar-file.

### Extract JAR file (`x`)

To *extract* the contents an entire JAR file, use option `'x'`:

```
> jar xvf jarFile
```

To *extract specific file(s)* from a JAR file, use option `'x'` and specify the file(s) to be extracted:

```
> jar xvf jarFile filesToExtract
```

### Update JAR File (`u`)

To update JAR file, use `'u'` option::

```
> jar uvf jarFile inputFiles
```

### Add Index to JAR File (`i`)

Use option `'i'` to generate index information in a file called `INDEX.LIST` inside the specified *jarfile*, which contains location information for each package in JAR file and all the JAR files specified in the `Class-Path` attribute.

```
> jar i jarFile
```

## 6. Processing JAR Files and JAR API

The API reference for "Java Archive (JAR) Files" is available @ http://docs.oracle.com/javase/7/docs/technotes/guides/jar/index.html.

### 6.1 Reading Resources from JAR File - `ClassLoader`'s `getResource()` and `getResourceAsStream()`

Your programs often require reading resources (such as images, data files, native libraries, `ResourceBundle` and properties-files) bundled in a JAR file. You can use `ClassLoader`'s method `getResource()` or `getResourceAsStream()` to access the JAR (as well as the regular file sytem). `getResource()` returns a `java.net.URL`; while `getResourceAsStream()` returns an `java.io.InputStream`.

For example,

```
String filename = "images/duke.gif";
java.net.URL imgURL = this.getClass().getClassLoader().getResource(filename);  // 1
// or
java.net.URL imgURL = this.getClass().getResource(filename);  // 2
```

The difference is [1] and [2] is that the *filename* in [1] is relative to the project root; while [2] is relative to the current class-file. Suppose that this class is `mypackage.myClass` and stored as `somepath\mypackage\myClass.class`. [1] asks for `somepath\images\duke.gif`, while [2] asks for `somepath\mypackage\image\duke.gif`.

### 6.2 Package `java.util.jar`

- Class `JarFile`: used to read the contents of a jar file from any file that can be opened with `java.io.RandomAccessFile`.
- Class `JarEntry`: represent an entry in a JAR file.
- Class `Manifest`: used to maintain manifest entries of "`name: value`" pairs.
- Class `Attributes`: maps manifest attribute names to their values.

- Class `Attributes.Name`: inner class of Attributes, representing the attribute names.

Example:

```
JarFile jarfile = new JarFile(jarFileName);
Manifest mf = jarfile.getManifestI();
Attributes attrs = mf.getMainAttributes();
String mainClassName = attrs.getValue(Attributes.Name.MAIN_CLASS);
```

## REFERENCES & RESOURCES


1. `jar` - The Java Archive Tool @ http://docs.oracle.com/javase/7/docs/technotes/tools/windows/jar.html.
2. Java Archive (JAR) Files @ http://docs.oracle.com/javase/7/docs/technotes/guides/jar/index.html.
3. JAR Manifest Format @ http://docs.oracle.com/javase/7/docs/technotes/guides/jar/jar.html#JAR_Manifest.


Latest version tested: JDK 1.7.3
Last modified: May, 2012


Feedback, comments, corrections, and errata can be sent to Chua Hock-Chuan (ehchua@ntu.edu.sg) | HOME