JAR File Specification

Contents

Introduction

JAR file is a file format based on the popular ZIP file format and is used for aggregating many files into one. A JAR file is essentially a zip file that contains an optional META-INF directory. A JAR file can be created by the command-line jar tool, or by using the `java.util.jar` API in the Java platform. There is no restriction on the name of a JAR file, it can be any legal file name on a particular platform.

In many cases, JAR files are not just simple archives of java classes files and/or resources. They are used as building blocks for applications and extensions. The META-INF directory, if it exists, is used to store package and extension configuration data, including security, versioning, extension and services.

The META-INF directory

The following files/directories in the META-INF directory are recognized and interpreted by the Java 2 Platform to configure applications, extensions, class loaders and services:

- `MANIFEST.MF`

The manifest file that is used to define extension and package related data.

- `INDEX.LIST`

This file is generated by the new "`-i`" option of the jar tool, which 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 their class loading process.

- `x.SF`

The signature file for the JAR file. 'x' stands for the base file name.

- `x.DSA`

The signature block file associated with the signature file with the same base file name. This file stores the digital signature of the corresponding signature file.

- `services/`

This directory stores all the service provider configuration files.

Name-Value pairs and Sections

Before we go to the details of the contents of the individual configuration files, some format convention needs to be defined. In most cases, information contained within the manifest file and signature files is represented as so-called "name: value" pairs inspired by the RFC822 standard. We also call these pairs headers or attributes.
Groups of name-value pairs are known as a "section". Sections are separated from other sections by empty lines.

Binary data of any form is represented as base64. Continuations are required for binary data which causes line length to exceed 72 bytes. Examples of binary data are digests and signatures.

Implementations shall support header values of up to 65535 bytes.

All the specifications in this document use the same grammar in which terminal symbols are shown in fixed width font and non-terminal symbols are shown in italic type face.

Specification:

section:	**header +newline*
nonempty-section:	*+header +newline*
newline:	`CR LF` | `LF` | `CR` (*not followed by* `LF`)
header:	*name* `:` *value*
name:	*alphanum *headerchar*
value:	SPACE **otherchar newline *continuation*
continuation:	SPACE **otherchar newline*
alphanum:	{`A-Z`} | {`a-z`} | {`0-9`}
headerchar:	*alphanum* | `-` | `_`
otherchar:	*any UTF-8 character except* `NUL`, `CR` *and* `LF`

; Also: To prevent mangling of files sent via straight e-mail, no
; header will start with the four letters "From".

Non-terminal symbols defined in the above specification will be referenced in the following specifications.

JAR Manifest

Overview

A JAR file manifest consists of a main section followed by a list of sections for individual JAR file entries, each separated by a newline. Both the main section and individual sections follow the section syntax specified above. They each have their own specific restrictions and rules.
The main section contains security and configuration information about the JAR file itself, as well as the application or extension that this JAR file is a part of. It also defines main attributes that apply to every individual manifest entry. No attribute in this section can have its name equal to "`Name`". This section is terminated by an empty line.

The individual sections define various attributes for packages or files contained in this JAR file. Not all files in the JAR file need to be listed in the manifest as entries, but all files which are to be signed must be listed. The manifest file itself must not be listed. Each section must start with an attribute with the name as "`Name`", and the value must be a relative path to the file, or an absolute URL referencing data outside the archive.

If there are multiple individual sections for the same file entry, the attributes in these sections are merged. If a certain attribute have different values in different sections, the last one is recognized.

Attributes which are not understood are ignored. Such attributes may include implementation specific information used by applications.

Manifest Specification:

manifest-file:	*main-section newline *individual-section*
main-section:	*version-info newline *main-attribute*
version-info:	`Manifest-Version :` *version-number*
version-number :	*digit+{.digit+}**
main-attribute:	*(any legitimate main attribute) newline*
individual-section:	`Name :` *value newline *perentry-attribute*
perentry-attribute:	*(any legitimate perentry attribute) newline*
newline :	`CR LF | LF | CR` (*not followed by* `LF`)
digit:	`{0-9}`

In the above specification, attributes that can appear in the main section are referred to as main attributes, whereas attributes that can appear in individual sections are referred to as per-entry attributes. Certain attributes can appear both in the main section and the individual sections, in which case the per-entry attribute value overrides the main attribute value for the specified entry. The two types of attributes are defined as follows.

Main Attributes

Main attributes are the attributes that are present in the main section of the manifest. They fall into the following different groups:

- general main attributes
  - Manifest-Version: Defines the manifest file version. The value is a legitimate version number, as described in the above spec.
  - Created-By: Defines the version and the vendor of the java implementation on top of which this manifest file is generated. This attribute is generated by the `jar` tool.
  - Signature-Version: Defines the signature version of the jar file. The value should be a valid *version-number* string.
  - Class-Path: The value of this attribute specifies the relative URLs of the extensions or libraries that this application or extension needs. URLs are separated by one or more spaces. The application or extension class loader uses the value of this attribute to construct its internal search path. See the Class-Path Attribute section for details.
- attribute defined for stand-alone applications: This attribute is used by stand-alone applications that are bundled into executable jar files which can be invoked by the java runtime directly by running "`java -jar x.jar`".
  - Main-Class: The value of this attribute is the class name of the main application class which the launcher will load at startup time. The value must *not* have the `.class` extension appended to the class name.
- attributes defined for applets: **Deprecated: These attributes and the mechanism which implements it may be removed in a future release.** These attributes are used by an applet which is bundled into JAR files to define requirements, version and location information for the extensions which this applet depends on. (see Extension Versioning).

  - Extension-List: This attribute indicates the extensions that are needed by the applet. Each extension listed in this attribute will have a set of additional attributes that the applet uses to specify which version and vendor of the extension it requires.
  - <extension>-Extension-Name: This attribute is the unique name of the extension. The Java Plug-in will compare the value of this attribute with the Extension-Name attribute in the manifests of installed extensions to determine if the extension is installed.
  - <extension>-Specification-Version: This attribute specifies the minimum extension specification version that is required by the applet. The Java Plug-in will compare the value of this attribute with the Specification-Version attribute of the installed extension to determine if the extension is up to date.
  - <extension>-Implementation-Version: This attritute specifies the minimum extension implementation version number that is required by the applet. The Java Plug-in will compare the value of this attribute with the Implementation-Version attribute of the installed extension to see if a more recent implementation needs to be downloaded.
  - <extension>-Implementation-Vendor-Id: This attribute can be used to identify the vendor of an extension implementation if the applet requires an implementation from a specific vendor. The Java Plug-in will compare the value of this attribute with the Implementation-Vendor-Id attribute of the installed extension.
  - <extension>-Implementation-URL: This attribute specifies a URL that can be used to obtain the most recent version of the extension if the required version is not already installed.
- attribute defined for extension identification: This attribute is used by extensions to define their unique identity.
  - Extension-Name: This attribute specifies a name for the extension contained in the Jar file. The name should be a unique identifier such as the name of the main package comprising the extension.
- attributes defined for extension and package versioning and sealing information: These attributes define features of the extension which the JAR file is a part of. The value of these attributes apply to all the packages in the JAR file, but can be overridden by per-entry attributes.
  - Implementation-Title: The value is a string that defines the title of the extension implementation.
  - Implementation-Version: The value is a string that defines the version of the extension implementation.
  - Implementation-Vendor: The value is a string that defines the organization that maintains the extension implementation.
  - Implementation-Vendor-Id: **Deprecated: This attribute may be ignored in a future release.** The value is a string id that uniquely defines the organization that maintains the extension implementation.
  - Implementation-URL: **Deprecated: This attribute may be ignored in a future release.** This attribute defines the URL from which the extension implementation can be downloaded from.
  - Specification-Title: The value is a string that defines the title of the extension specification.
  - Specification-Version: The value is a string that defines the version of the extension specification.
  - Specification-Vendor: The value is a string that defines the organization that maintains the extension specification.
  - Sealed: This attribute defines whether this JAR file is sealed or not. The value can be either "true" or "false", case is ignored. If it is set to "true", then all the packages in the JAR file are defaulted to be sealed, unless they are defined otherwise individually.

Per-Entry Attributes

Per-entry attributes apply only to the individual JAR file entry to which the manifest entry is associated with. If the same attribute also appeared in the main section, then the value of the per-entry attribute overwrites the main attribute's value. For example, if JAR file a.jar has the following manifest content:

```
Manifest-Version: 1.0
Created-By: 1.2 (Sun Microsystems Inc.)
Sealed: true
Name: foo/bar/
Sealed: false
```

It means that all the packages archived in a.jar are sealed, except that package foo.bar is not.

The per-entry attributes fall into the following groups:

- attributes defined for file contents:

    - Content-Type: This attribute can be used to specify the MIME type and subtype of data for a specific file entry in the JAR file. The value should be a string in the form of *type/subtype.* For example "image/bmp" is an image type with a subtype of bmp (representing bitmap). This would indicate the file entry as an image with the data stored as a bitmap. RFC 1521 and 1522 discuss and define the MIME types definition.
- attributes defined for package versioning and sealing information: These are the same set of attributes defined above as main attributes that defines the extension package versioning and sealing information. When used as per-entry attributes, these attributes overwrites the main attributes but only apply to the individual file specified by the manifest entry.
- attribute defined for beans objects:
    - Java-Bean: Defines whether the specific jar file entry is a Java Beans object or not. The value should be either "true" or "false", case is ignored.
- attributes defined for signing: These attributes are used for signing and verifying purposes. More details here.
    - x-Digest-y: The name of this attribute specifies the name of the digest algorithm used to compute the digest value for the corresponding jar file entry. The value of this attribute stores the actual digest value. The prefix 'x' specifies the algorithm name and the optional suffix 'y' indicates to which language the digest value should be verified against.
    - Magic: This is an optional attribute that can be used by applications to indicate how verifier should compute the digest value contained in the manifest entry. The value of this attribute is a set of comma separated context specific strings. Detailed description is here.

Signed JAR File

Overview

A JAR file can be signed by using the command line jarsigner tool or directly through the `java.security` API. Every file entry, including non-signature related files in the `META-INF` directory, will be signed if the JAR file is signed by the jarsigner tool. The signature related files are:

- `META-INF/MANIFEST.MF`
- `META-INF/*.SF`
- `META-INF/*.DSA`
- `META-INF/*.RSA`
- `META-INF/SIG-*`

Note that if such files are located in `META-INF` subdirectories, they are not considered signature-related. Case-insensitive versions of these filenames are reserved and will also not be signed.

Subsets of a JAR file can be signed by using 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. When jarsigner is not used, the signing program has to construct both the signature file and the signature block file.

For every file entry signed in the signed JAR file, an individual manifest entry is created for it as long as it does not already exist in the manifest. Each manifest entry lists one or more digest attributes and an optional Magic attribute.

Signature File

Each signer is represented by a signature file with extension `.SF`. The major part of the file is similar to the manifest file. It consists of a main section which includes information supplied by the signer but not specific to any particular jar file entry. In addition to the `Signature-Version` and `Created-By` attributes (see Main Attributes), the main section can also include the following security attributes:

- x-Digest-Manifest-Main-Attributes (where x is the standard name of a `java.security.MessageDigest` algorithm): The value of this attribute is the digest value of the main attributes of the manifest.
- x-Digest-Manifest (where x is the standard name of a `java.security.MessageDigest` algorithm): The value of this attribute is the digest value of the entire manifest.

The main section is followed by a list of individual entries whose names must also be present in the manifest file. Each individual entry must contain at least the digest of its corresponding entry in the manifest file.

Paths or URLs appearing in the manifest file but not in the signature file are not used in the calculation.

Signature Validation

A successful JAR file verification occurs if the signature(s) are valid, and none of the files that were in the JAR file when the signatures were generated have been changed since then. JAR file verification involves the following steps:

1. Verify the signature over the signature file when the manifest is first parsed. For efficiency, this verification can be remembered. Note that this verification only validates the signature directions themselves, not the actual archive files.

2. If an `x-Digest-Manifest` attribute exists in the signature file, verify the value against a digest calculated over the entire manifest. If more than one `x-Digest-Manifest` attribute exists in the signature file, verify that at least one of them matches the calculated digest value.

3. If an `x-Digest-Manifest` attribute does not exist in the signature file or none of the digest values calculated in the previous step match, then a less optimized verification is performed:

    1. If an `x-Digest-Manifest-Main-Attributes` entry exists in the signature file, verify the value against a digest calculated over the main attributes in the manifest file. If this calculation fails, then JAR file verification fails. This decision can be remembered for efficiency. If an `x-Digest-Manifest-Main-Attributes` entry does not exist in the signature file, its nonexistence does not affect JAR file verification and the manifest main attributes are not verified.

    2. Verify the digest value in each source file information section in the signature file against a digest value calculated against the corresponding entry in the manifest file. If any of the digest values don't match, then JAR file verification fails.

    One reason the digest value of the manifest file that is stored in the `x-Digest-Manifest` attribute may not equal the digest value of the current manifest file is that one or more files were added to the JAR file (using the jar tool) after the signature (and thus the signature file) was generated. When the jar tool is used to add files, the manifest file is changed (sections are added to it for the new files), but the signature file is not. A verification is still considered successful if none of the files that were in the JAR file when the signature was generated have been changed since then, which is the case if the digest values in the non-header sections of the signature file equal the digest values of the corresponding sections in the manifest file.

4. For each entry in the manifest, verify the digest value in the manifest file against a digest calculated over the actual data referenced in the "Name:" attribute, which specifies either a relative file path or URL. If any of the digest values don't match, then JAR file verification fails.

Example manifest file:

```
Manifest-Version: 1.0
Created-By: 1.7.0 (Sun Microsystems Inc.)

Name: common/class1.class
SHA-256-Digest: (base64 representation of SHA-256 digest)

Name: common/class2.class
SHA1-Digest: (base64 representation of SHA1 digest)
SHA-256-Digest: (base64 representation of SHA-256 digest)
```

The corresponding signature file would be:

```
Signature-Version: 1.0
SHA-256-Digest-Manifest: (base64 representation of SHA-256 digest)
SHA-256-Digest-Manifest-Main-Attributes: (base64 representation of SHA-256 digest)

Name: common/class1.class
SHA-256-Digest: (base64 representation of SHA-256 digest)

Name: common/class2.class
SHA-256-Digest: (base64 representation of SHA-256 digest)
```

The Magic Attribute

Another requirement to validate the signature on a given manifest entry is that the verifier understand the value or values of the Magic key-pair value in that entry's manifest entry.

The Magic attribute is optional but it is required that a parser understand the value of an entry's Magic key if it is verifying that entry's signature.

The value or values of the Magic attribute are a set of comma-separated context-specific strings. The spaces before and after the commas are ignored. Case is ignored. The exact meaning of the magic attributes is application specific. These values indicate how to compute the hash value contained in the manifest entry, and are therefore crucial to the proper verification of the signature. The keywords may be used for dynamic or embedded content, multiple hashes for multilingual documents, etc.

Here are two examples of the potential use of Magic attribute in the manifest file:

```
Name: http://www.example-scripts.com/index#script1
SHA-256-Digest: (base64 representation of SHA-256 hash)
Magic: JavaScript, Dynamic

Name: http://www.example-tourist.com/guide.html
SHA-256-Digest: (base64 representation of SHA-256 hash)
SHA-256-Digest-French: (base64 representation of SHA-256 hash)
SHA-256-Digest-German: (base64 representation of SHA-256 hash)
Magic: Multilingual
```

In the first example, these Magic values may indicate that the result of an http query is the script embedded in the document, as opposed to the document itself, and also that the script is generated dynamically. These two pieces of information indicate how to compute the hash value against which to compare the manifest's digest value, thus comparing a valid signature.

In the second example, the Magic value indicates that the document retrieved may have been content-negotiated for a specific language, and that the digest to verify against is dependent on which language the document retrieved is written in.

Digital Signatures

A digital signature is a signed version of the `.SF` signature file. These are binary files not intended to be interpreted by humans.
Digital signature files have the same filenames as the .SF files but different extensions. The extension varies depending on the type of digital signature.

- `.RSA` (PKCS7 signature, SHA-256 + RSA)
- `.DSA` (PKCS7 signature, DSA)

Digital signature files for signature algorithms not listed above must reside in the `META-INF` directory and have the prefix "`SIG-`". The corresonding signature file (`.SF` file) must also have the same prefix.
For those formats that do not support external signed data, the file shall consist of a signed copy of the `.SF` file. Thus some data may be duplicated and a verifier should compare the two files.

Formats that support external data either reference the `.SF` file, or perform calculations on it with implicit reference.

Each `.SF` file may have multiple digital signatures, but those signatures should be generated by the same legal entity.

File name extensions may be 1 to 3 *alphanum* characters. Unrecognized extensions are ignored.

Notes on Manifest and Signature Files

Following is a list of additional restrictions and rules that apply to manifest and signature files.

- Before parsing:
  - If the last character of the file is an EOF character (code 26), the EOF is treated as whitespace. Two newlines are appended (one for editors that don't put a newline at the end of the last line, and one so that the grammar doesn't have to special-case the last entry, which may not have a blank line after it).
- Attributes:
  - In all cases for all sections, attributes which are not understood are ignored.

    - Attribute names are case insensitive. Programs which generate manifest and signature files should use the cases shown in this specification however.
    - Attribute names cannot be repeated within a section.
- Versions:
    - Manifest-Version and Signature-Version must be first, and in exactly that case (so that they can be recognized easily as magic strings). Other than that, the order of attributes within a main section is not significant.
- Ordering:
    - The order of individual manifest entries is not significant.
    - The order of individual signature entries is not significant, except that the digests that get signed are in that order.
- Line length:
    - No line may be longer than 72 bytes (not characters), in its UTF8-encoded form. If a value would make the initial line longer than this, it should be continued on extra lines (each starting with a single SPACE).
- Errors:
    - If a file cannot be parsed according to this spec, a warning should be output, and none of the signatures should be trusted.
- Limitations:
    - Because header names cannot be continued, the maximum length of a header name is 70 bytes (there must be a colon and a SPACE after the name).
    - NUL, CR, and LF can't be embedded in header values, and NUL, CR, LF and ":" can't be embedded in header names.
    - Implementations should support 65535-byte (not character) header values, and 65535 headers per file. They might run out of memory, but there should not be hard-coded limits below these values.
- Signers:
    - It is technically possible that different entities may use different signing algorithms to share a single signature file. This violates the standard, and the extra signature may be ignored.
- Algorithms:
    - No digest algorithm or signature algorithm is mandated by this standard. However, at least one of MD5 and SHA1 digest algorithm must be supported.

JAR Index

Overview

Since 1.3, JarIndex is introduced to optimize the class searching process of class loaders for network applications, especially applets. Originally, an applet class loader uses a simple linear search algorithm to search each element on its internal search path, which is constructed from the "ARCHIVE" tag or the "Class-Path" main attribute. The class loader downloads and opens each element in its search path, until the class or resource is found. If the class loader tries to find a nonexistent resource, then 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 startup, sluggish response and wasted network bandwidth. The JarIndex mechanism collects the contents of all the jar files defined in an applet and stores the information in an index file in the first jar file on the applet's class path. After the first jar file is downloaded, the applet class loader will use the collected content information for efficient downloading of jar files.
The existing `jar` tool is enhanced to be able to examine a list of jar files and generate directory information as to which classes and resources reside in which jar file. This directory information is stored in a simple text file named `INDEX.LIST` in the `META-INF` directory of the root jar file. When the classloader loads the root jar file, it reads the `INDEX.LIST` file and uses it to construct a hash table of mappings from file and package names to lists of jar file names. In order to find a class or a resource, the class loader queries the hashtable to find the proper jar file and then downloads it if necessary.

Once the class loader finds a `INDEX.LIST` file in a particular jar file, it always trusts the information listed in it. If a mapping is found for a particular class, but the class loader fails to find it by following the link, an InvalidJarIndexException is thrown. When this occurs, the application developer should rerun the `jar` tool on the extension to get the right information into the index file.

To prevent adding too much space overhead to the application and to speed up the construction of the in-memory hash table, the INDEX.LIST file is kept as small as possible. For classes with non-null package names, mappings are recorded at the package level. Normally one package name is mapped to one jar file, but if a particular package spans more than one jar file, then the mapped value of this package will be a list of jar files. For resource files with non-empty directory prefixes, mappings are also recorded at the directory level. Only for classes with null package name, and resource files which reside in the root directory, will the mapping be recorded at the individual file level.

Index File Specification

The `INDEX.LIST` file contains one or more sections each separated by a single blank line. Each section defines the content of a particular jar file, with a header defining the jar file path name, followed by a list of package or file names, one per line. All the jar file paths are relative to the code base of the root jar file. These path names are resolved in the same way as the current extension mechanism does for bundled extensions.

The UTF-8 encoding is used to support non ASCII characters in file or package names in the index file.

Specification

index file :	*version-info blankline section**
version-info :	`JarIndex-Version:` *version-number*
version-number :	*digit+{.digit+}**
section :	*body blankline*
body :	*header name**
header :	*char+*`.jar` *newline*
name :	*char+ newline*
char :	*any valid Unicode character except* `NULL, CR` *and*`LF`
blankline:	*newline newline*
newline :	`CR LF | LF | CR` (*not followed by* `LF`)
digit:	{`0-9`}

The `INDEX.LIST` file is generated by running `jar -i.` See the jar man page for more details.

Backward Compatibility

The new class loading scheme is totally backward compatible with applications developed on top of the current extension mechanism. When the class loader loads the first jar file and an `INDEX.LIST` file is found in the `META-INF` directory, it would construct the index hash table and use the new loading scheme for the extension. Otherwise, the class loader will simply use the original linear search algorithm.

Service Provider

Overview

Files in the `META-INF/services` directory are service provider configuration files. A service is a well-known set of interfaces and (usually abstract) classes. A service provider is a specific implementation of a service. The classes in a provider typically implement the interfaces and subclass the classes defined in the service itself. Service providers may be installed in an implementation of the Java platform in the form of extensions, that is, jar files placed into any of the usual extension directories. Providers may also be made available by adding them to the applet or application class path or by some other platform-specific means.

A service is represented by an abstract class. A provider of a given service contains one or more concrete classes that extend this service class with data and code specific to the provider. This provider class will typically not be the entire provider itself but rather a proxy that contains enough information to decide whether the provider is able to satisfy a particular request together with code that can create the actual provider on demand. The details of provider classes tend to be highly service-specific; no single class or interface could possibly unify them, so no such class has been defined. The only requirement enforced here is that provider classes must have a zero-argument constructor so that they may be instantiated during lookup.

Provider-Configuration File

A service provider identifies itself by placing a provider-configuration file in the resource directory `META-INF/services`. The file's name should consist of the fully-qualified name of the abstract service class. The file should contain a newline-separated list of unique concrete provider-class names. Space and tab characters, as well as blank lines, are ignored. The comment character is '#' (0x23); on each line all characters following the first comment character are ignored. The file must be encoded in UTF-8.

Example

Suppose we have a service class named java.io.spi.CharCodec. It has two abstract methods:

- `public abstract CharEncoder getEncoder(String encodingName);`
- `public abstract CharDecoder getDecoder(String encodingName);`

Each method returns an appropriate object or null if it cannot translate the given encoding. Typical CharCodec providers will support more than one encoding.

If sun.io.StandardCodec is a provider of the CharCodec service then its jar file would contain the file `META-INF/services/java.io.spi.CharCodec`. This file would contain the single line:

```
sun.io.StandardCodec    # Standard codecs for the platform
```

To locate an encoder for a given encoding name, the internal I/O code would do something like this:

```
   CharEncoder getEncoder(String encodingName) {
       Iterator ps = Service.providers(CharCodec.class);
       while (ps.hasNext()) {
           CharCodec cc = (CharCodec)ps.next();
           CharEncoder ce = cc.getEncoder(encodingName);
           if (ce != null)
               return ce;
       }
       return null;
   }
```

The provider-lookup mechanism always executes in the security context of the caller. Trusted system code should typically invoke the methods in this class from within a privileged security context.

Class-Path Attribute

The manifest for an application can specify one or more relative URLs referring to the JAR files and directories for other libraries that it requires. These relative URLs are treated relative to the code base from which the containing application was loaded.

An application (or, more generally, a JAR file) specifies the relative URLs of the libraries that it requires with the manifest attribute `Class-Path`. This attribute lists the URLs to search for implementations of other libraries if they cannot be found on the host Java virtual machine. These relative URLs may include JAR files and directories for any libraries or resources needed by the application. Relative URLs not ending with a slash (`/`) are assumed to refer to JAR files. For example:

```
Class-Path: servlet.jar infobus.jar acme/beans.jar images/
```

At most one `Class-Path` header may be specified in a JAR file's manifest.

Currently, the URLs must be *relative* to the code base of the JAR file for security reasons. Thus, remote optional packages will originate from the same code base as the application.

Each relative URL is resolved against the code base from which the containing application or library was loaded. If the resulting URL is invalid or refers to a resource that cannot be found, then it is ignored.

The resulting URLs are used to extend the class path for the application, applet, or servlet by inserting the URLs in the class path immediately following the URL of the containing JAR file. Any duplicate URLs are omitted. For example, given the following class path:

```
a.jar b.jar
```

Suppose `b.jar` contained the following `Class-Path` manifest attribute:

```
Class-Path: x.jar a.jar
```

As a result, the resulting application class path would be the following:

```
a.jar b.jar x.jar
```

If `x.jar` had dependencies of its own, then these would be added according to the same rules for each subsequent URL. In the actual implementation, JAR file dependencies are processed lazily so that the JAR files are not opened until needed.

Package Sealing

JAR files and packages can be optionally *sealed* so that an package can enforce consistency within a version.

A package sealed within a JAR specifies that all classes defined in that package must originate from the same JAR. Otherwise, a `SecurityException` is thrown.

A sealed JAR specifies that all packages defined by that JAR are sealed unless overridden specifically for a package.

A sealed package is specified through the manifest attribute, `Sealed`, whose value is `true` or `false` (case irrelevant). For example:

```
Name: javax/servlet/internal/
Sealed: true
```

This specifies that the `javax.servlet.internal` package is sealed, and that all classes in that package must be loaded from the same JAR file.

If this attribute is missing, then the package sealing attribute is that of the containing JAR file.

A sealed JAR is specified via the same manifest header, `Sealed`, with the value again of either `true` or `false`. For example:

```
Sealed: true
```

This specifies that all packages in this archive are sealed unless explicitly overridden for a particular package with the `Sealed` attribute in a manifest entry.

If this attribute is missing, the JAR file is assumed to *not* be sealed, for backwards compatibility. The system then defaults to examining package headers for sealing information.

Package sealing is also important for security because it restricts access to package-protected members to only those classes defined in the package that originated from the same JAR file.

An unnamed package is not sealable, so classes that are to be sealed must be placed in their own packages.

API Details

Package java.util.jar

See Also

Package java.security

Package java.util.zip

Copyright © 1993, 2019, Oracle and/or its affiliates. All rights reserved. | Cookie Preferences | Ad Choices. Contact Us