12

# Schema Objects and Oracle JVM Utilities

This chapter describes the schema objects that you use in Oracle Database Java environment and Oracle JVM utilities. You run these utilities from a UNIX shell or from the Microsoft Windows DOS prompt.

> **Note:**
>
> All names supplied to these tools are case-sensitive. As a result, the schema, user name, and password should not be changed to uppercase.

This chapter contains the following sections:


- Overview of Schema Objects
- What and When to Load
- Resolution of Schema Objects
- Compilation of Schema Objects
- The ojvmtc Tool
- The loadjava Tool
- The dropjava Tool
- The ojvmjava Tool


## 12.1 Overview of Schema Objects

Unlike conventional Java virtual machine (JVM), which compiles and loads Java files, Oracle JVM compiles and loads schema objects. The following kinds of Java schema objects are loaded:

- Java class schema objects, which correspond to Java class files.
- Java source schema objects, which correspond to Java source files.
- Java resource schema objects, which correspond to Java resource files.

To ensure that a class file can be run by Oracle JVM, you must use the `loadjava` tool to create a Java class schema object from the class file or the source file and load it into a schema. To make a resource file accessible to Oracle JVM, you must use the `loadjava` tool to create and load a Java resource schema object from the resource file.

The `dropjava` tool deletes schema objects that correspond to Java files. You should always use the `dropjava` tool to delete a Java schema object that was created with the `loadjava` tool. Dropping schema objects using SQL data definition language (DDL) commands will not update auxiliary data maintained by the `loadjava` tool and the `dropjava` tool.

# 12.2 What and When to Load

You must load resource files using the `loadjava` tool. If you create `.class` files outside the database with a conventional compiler, then you must load them with the `loadjava` tool. The alternative to loading class files is to load source files and let Oracle Database compile and manage the resulting class schema objects. The most productive approach is to compile and debug most of your code outside the database, and then load the `.class` files. For a particular Java class, you can load either its `.class` file or the corresponding `.java` file, but not both.

The `loadjava` tool accepts Java Archive (JAR) files that contain either source and resource files or class and resource files. When you pass a JAR or ZIP file to the `loadjava` tool, by default, it opens the archive and loads its members individually.

> **Note:**
>
> When you load the contents of a JAR into the database, you have the option of creating a database object representing the JAR itself.

A file, whose content has not changed since the last time it was loaded, is not reloaded. As a result, there is little performance penalty for loading JAR files. Loading JAR files is a simple, fool-proof way to use the `loadjava` tool.

It is illegal for two schema objects in the same schema to define the same class. For example, assume that `a.java` defines class `x` and you want to move the definition of `x` to `b.java`. If `a.java` has already been loaded, then the `loadjava` tool will reject an attempt to load `b.java`. Instead, do either of the following:

- Drop `a.java`, load `b.java`, and then load the new `a.java`, which does not define `x`.
- Load the new `a.java`, which does not define `x`, and then load `b.java`.

**Related Topics**

- Database Resident JARs

# 12.3 Resolution of Schema Objects

All Java classes contain references to other classes. A conventional JVM searches for classes in the directories, ZIP files, and JAR files named in the `CLASSPATH`. In contrast, Oracle JVM searches schemas for class schema objects. Each class in the database has a resolver specification, which is Oracle Database counterpart to `CLASSPATH`. For example, the resolver specification of a class, `alpha`, lists the schemas to search for classes that `alpha` uses. Notice that resolver specifications are per-class, whereas in a classic JVM, `CLASSPATH` is global to all classes.

In addition to a resolver specification, each class schema object has a list of interclass reference bindings. Each reference list item contains a reference to another class and one of the following:

- The name of the class schema object to call when the class uses the reference
- A code indicating whether the reference is unsatisfied, that is, whether the referent schema object is known

Oracle Database facility known as **resolver** maintains reference lists. For each interclass reference in a class, the resolver searches the schemas specified by the resolver specification of the class for a valid class schema object that satisfies the reference. If all references are resolved, then the resolver marks the class valid. A class that has never been resolved, or has been resolved unsuccessfully, is marked invalid. A class that depends on a schema object that becomes invalid is also marked invalid at the time the first class is marked invalid. In other words, invalidation cascades upward from a class to the classes that use it and the classes that use these classes, and so on. When resolving a class that depends on an invalid class, the resolver first tries to resolve the referenced class, because it may be marked invalid only because it has never been resolved. The resolver does not resolve classes that are marked valid.

A developer can direct the `loadjava` tool to resolve classes or can defer resolution until run time. The resolver runs automatically when a class tries to load a class that is marked invalid. It is best to resolve before run time to learn of missing classes early. Unsuccessful resolution at run time produces a `ClassNotFound` exception. Furthermore, run-time resolution can fail for the following reasons:

- Lack of database resources, if the tree of classes is very large
- Deadlocks due to circular dependencies

The `loadjava` tool has two resolution modes:

- Load-and-resolve

  The `-resolve` option loads all classes you specify on the command line, marks them invalid, and then resolves them. Use this mode when initially loading classes that refer to each other, and, in general, when reloading isolated classes as well. By loading all classes and then resolving them, this mode avoids the error message that occurs if a class refers to a class that will be loaded later while the command is being carried out.
- Load-then-resolve

  This mode resolves each class at run time. The `-resolve` option is not specified.

> **Note:**
>
> As with a Java compiler, the `loadjava` tool resolves references to classes but not to resources. Ensure that you correctly load the resource files that your classes need.

If you can, defer resolution until all classes have been loaded. This avoids a situation in which the resolver marks a class invalid because a class it uses has not yet been loaded.

# 12.4 Compilation of Schema Objects

Loading a source file creates or updates a Java source schema object and invalidates the class schema objects previously derived from the source. If the class schema objects do not exist, then the `loadjava` tool creates them. The `loadjava` tool invalidates the old class schema objects because they were not compiled from the newly loaded source. Compilation of a newly loaded source, for example, class `A`, is automatically triggered by any of the following conditions:

- The resolver, while working on class `B`, finds that class `B` refers to class `A`, but class `A` is invalid.

- The compiler, while compiling the source of class `B`, finds that class `B` refers to class `A`, but class `A` is invalid.
- The class loader, while trying to load class `A` for running it, finds that class `A` is invalid.

To force compilation when you load a source file, use the `loadjava -resolve` option.

The compiler writes error messages to the predefined `USER_ERRORS` view. The `loadjava` tool retrieves and displays the messages produced by its compiler invocations.

The compiler recognizes some options. There are two ways to specify options to the compiler. If you run the `loadjava` tool with the `-resolve` option, then you can specify compiler options on the command line. You can additionally specify persistent compiler options in a per-schema database table, `JAVA$OPTIONS`. You can use the `JAVA$OPTIONS` table for default compiler options, which you can override selectively using a `loadjava` tool option.

> **Note:**
>
> A command-line option overrides and clears the matching entry in the `JAVA$OPTIONS` table.

A `JAVA$OPTIONS` row contains the names of source schema objects to which an option setting applies. You can use multiple rows to set the options differently for different source schema objects. The compiler looks up options in `JAVA$OPTIONS` when it has been called by the class loader or when called from the command line without specifying any options. When compiling a source schema object for which there is neither a `JAVA$OPTIONS` entry nor a command-line value for an option, the compiler assumes a default value, as follows:

- `encoding = System.getProperty("file.encoding");`
- `debug = true`

  This option is equivalent to `javac -g`.

# 12.5 The ojvmtc Tool

This section describes the following topics:

- About the ojvmtc Tool
- Arguments of ojvmtc Command

## 12.5.1 About the ojvmtc Tool

The `ojvmtc` tool enables you to resolve all external references, prior to running the `loadjava` tool. The `ojvmtc` tool allows the specification of a classpath that specifies the JARs, classes, or directories to be used to resolve class references. When an external reference cannot be resolved, this tool either produces a list of unresolved references or generated stub classes to allow resolution of the references, depending on the options specified. Generated stub classes throw a `java.lang.ClassNotfoundException` if it is referenced at runtime.

The syntax is:

```
ojvmtc [-help ] [-bootclasspath] [-server connect_string] [-jar jar_name] [-list] -
classpath jar1:path2:jar2
jars,...,classes
```

For example:

```
ojvmtc -bootclasspath $JAVA_HOME/jre/lib/rt.jar -classpath classdir/lib1.jar:classdir/
lib2.jar -jar set.jar app.jar
```

The preceding example uses `rt.jar`, `classdir/lib1.jar`, and `classdir/lib2.jar` to resolve references in `app.jar`. All the classes examined are added to `set.jar`, except for those found in `rt.jar`.

Another example is:

```
ojvmtc -server thin:HR/@localhost:5521:orcl -classpath jar1:jar2 -list app2.jar
Password:password
```

The preceding example uses classes found in the server specified by the connection string as well as `jar1` and `jar2` to resolve `app2.jar`. Any missing references are displayed to `stdout`.

## 12.5.2 Arguments of ojvmtc Command

Table 12-1 summarizes the arguments of this command.

**Table 12-1 ojvmtc Argument Summary**

| Argument | Description |
|---|---|
| `-classpath` | Uses the specified JARs and classes for the closure set. |
| `-bootclasspath` | Uses the specified classes for closure, but does not include them in the closure set. |
| `-server connect_string` | Connects to the server using visible classes in the same manner as `-bootclasspath`. |
| `connect_string thin\|OCI` | Connects to the server using thin or Oracle Call Interface (OCI) specific driver.<br>If you use thin driver, the syntax is as follows:<br>`thin:user/passwd@host:port:sid`<br>If you use OCI driver, the syntax is as follows:<br>`oci:user/passwd@host:port:sid`<br>`oci:user/passwd@tnsname`<br>`oci:user/passwd@(connect descriptor)` |
| `-jar jar_name` | Writes each class of the closure set to a JAR and generates stubs for missing classes |
| `-list` | Lists the missing classes. |

# 12.6 The loadjava Tool

The `loadjava` tool creates schema objects from files and loads them into a schema. Schema objects can be created from Java source, class, and data files.

You must have the following SQL database privileges to load classes:

- `CREATE PROCEDURE` and `CREATE TABLE` privileges to load into your schema.
- `CREATE ANY PROCEDURE` and `CREATE ANY TABLE` privileges to load into another schema.

- `oracle.aurora.security.JServerPermission.loadLibraryInClass.`*`classname`*.

You can run the `loadjava` tool either from the command line or by using the `loadjava` method contained in the `DBMS_JAVA` class. To run the tool from within your Java application, do the following:

```
call dbms_java.loadjava('... options...');
```

The options are the same as those that can be specified on the command line with the `loadjava` tool. Separate each option with a space. Do not separate the options with a comma. The only exception for this is the `-resolver` option, which contains spaces. For `-resolver`, specify all other options in the first input parameter and the `-resolver` options in the second parameter, as follows:

```
call dbms_java.loadjava('..options...', 'resolver_options');
```

Do not specify the `-thin`, `-oci`, `-user`, and `-password` options, because they relate to the database connection for the `loadjava` command-line tool. The output is directed to `stderr`. Set `serveroutput` on, and call `dbms_java.set_output`, as appropriate.

> **Note:**
>
> The `loadjava` tool is located in the `bin` directory under `$ORACLE_HOME`.

Just before the `loadjava` tool exits, it checks whether the processing was successful. All failures are summarized preceded by the following header:

```
The following operations failed
```

Some conditions, such as losing the connection to the database, cause the `loadjava` tool to terminate prematurely. These errors are displayed with the following syntax:

```
exiting: error_reason
```

This section covers the following:

- loadjava Tool Syntax
- loadjava Tool Argument Summary
- loadjava Tool Argument Details

## 12.6.1 loadjava Tool Syntax

The syntax of the `loadjava` tool command is as follows:

> **Note:**
>
> - The `(* -)` option is the preferred option over the `-genmissing` and `-genmissingjar` options for resolving class references.
> - The the `-genmissing` and `-genmissingjar` options cannot be used in an option file or an option table. These options are applicable to all the classes to be loaded and cannot be used only for specific classes.

```
loadjava {-user | -u} user [@database] [options]
file.java | file.class | file.jar | file.zip | resourcefile | URL...
  [-casesensitivepub]
  [-cleargrants]
  [-debug]
  [-d | -definer]
  [-dirprefix prefix]
  [-add-modules module-list]
  [-module module-name]
  [-automatic]
  [-hotload]
  [-e | -encoding encoding_scheme]
  [-fileout file]
  [-f | -force]
  [-genmissing]
  [-genmissingjar jar_file]
  [-g | -grant user [, user]...]
  [-help]
  [-jarasresource]
  [-noaction]
  [-nosynonym]
  [-nousage]
  [-noverify]
  [-o | -oci | oci8]
  [-optionfile file]
  [-optiontable table_name]
  [-publish package]
  [-pubmain number]
  [-recursivejars]
  [-r | -resolve]
  [-R | -resolver "resolver_spec"]
  [-append-resolver "resolver_spec"]
  [-prepend-resolver "resolver_spec"]
  [-resolveonly]
  [-S | -schema schema]
  [-stdout]
  [-stoponerror]
  [-s | -synonym]
  [-tableschema schema]
  [-t | -thin]

  [-unresolvedok]
  [-v | -verbose]
  [-jarsasdbobjects]
  [-prependjarnames]
  [-nativecompile]
```

## 12.6.2 loadjava Tool Argument Summary

This section summarizes the `loadjava` tool command arguments. If you run the `loadjava` tool multiple times, specifying the same files and different options, then the options specified in the most recent invocation hold.

However, there are the following two exceptions to this rule:

- If the `loadjava` tool does not load a file because it matches a digest table entry, then most options on the command line have no effect on the schema object. The exceptions are `-grant` and `-resolve`, which always take effect. You must use the `-force` option to direct the `loadjava` tool to skip the digest table look up.

- The `-grant` option is cumulative. Every user specified in every invocation of the `loadjava` tool for a given class in a given schema has the `EXECUTE` privilege.

**Table 12-2 loadjava Argument Summary**

| Argument | Description |
|---|---|
| *filenames* | Specifies any number and combination of `.java`, `.class`, `.jar`, `.zip`, and resource file name arguments. |
| `-proxy host:port` | Specifies the host name and the port of the proxy server.<br>If you do not have physical access to the server host or the `loadjava` client for loading classes, resources, and Java source, then you can use an HTTP URL with the `loadjava` tool to specify the JAR, class file, or resource and load the class from a remote server. `host` is the host name or address and `port` is the port the proxy server is using. The URL implementation must be such that the `loadjava` tool can determine the type of file to load, that is, JAR, class, resource, or Java source. For example:<br>`loadjava -u HR -r -v -proxy proxy_server:1020 http://my.server.com/this/is /the/path/my.jar`<br>`Password: password`<br>When the URL support is used inside the server, you should have proper Java permissions to access to the remote source. The URL support also includes `ftp:` and `file:` URLs. |
| `-casesensitivepub` | Instructs to create case-sensitive names, when the package is published. Unless the names are already all upper case, it usually requires quoting the names in PL/SQL. |
| `-cleargrants` | Specifies that the `loadjava` tool should revoke any existing grants of execute privilege before it grants execute privilege to the users and roles specified by the `-grant` operand. For example, if the intent is to have execute privilege granted to only `HR`, then the proper options are:<br>`-grant HR -cleargrants` |
| `-debug` | Turns on SQL logging. |
| `-definer` | Confers definer privileges upon classes. By default, class schema objects run with the privileges of their invoker. This option is conceptually similar to the UNIX `setuid` facility. |
| `-dirprefix prefix` | Specifies that the *prefix* should be deleted from the name of any files or JAR entries that start with *prefix*, before the name of the schema object is determined. For classes and sources, the name of the schema object is determined by their contents. Therefore, this option will only have an effect for resources. |
| `-add-modules <module-list>` | Specifies a comma-separated list of modules to automatically add, when a class being loaded is invoked as the main class in a Java session. Any modules explicitly required by these modules are also added to the session. |

**Table 12-2 (Cont.) loadjava Argument Summary**

| Argument | Description |
|---|---|
| `-resolver <"resolver-spec">` | Uses the `resolver-spec` option as the resolver specification for the loaded classes. A `-resolver` specification completely overrides the default generated resolver. As resolvers contain special characters, they should be enclosed with quotation marks on the command line. When loading into the *unnamed* module, the default resolver is `((* <schema>) (* PUBLIC))`. If the JAR files are loaded with the `-prependjarnames` option, then the default resolver is `((///<jar_name>///* <schema>) (///<jar_name>///* PUBLIC) (* <schema>) (* PUBLIC))`.<br>When loading into a named module, the default resolver is `((<module>///* <schema>) (<module>///* PUBLIC) (* <schema>) (* PUBLIC))`. When the `-add-modules` option is specified, the additional module resolver terms are included for each added module. If the module JAR files are loaded with the `-prependjarnames` option, then the default resolver is `((<module>///<jar_name>///* <schema>) (<module>///<jar_name>///* PUBLIC) (* <schema>) (* PUBLIC))`. |
| `-prepend-resolver <"resolver-spec">` | Prepends the specified list of resolver terms to the default resolver. |
| `-append-resolver <"resolver-spec">` | Appends the specified list of resolver terms to the default resolver. For example, you can specify the `--append-resolve ((* -))` option to add this term to the end of the default resolver. |
| `-automatic` | Creates an automatic module from any JAR file that is loaded without any module-info. |
| `-hotload` | Captures the current list of packages defined by the specified modules, in the `module-data` object during hotloading. If packages are added later, or removed from these modules, the module must be reloaded or re-hotloaded. |
| `-module <name>` | Loads classes into the specified module. This option is not required when you load JAR files that either have a `module-info` JAR entry or if the `-automatic loadjava` option is specified. It is also not required if a `module-info` file is also loaded from the standard corresponding directory location by the same `loadjava` command. It throws an error if the module name specified does not match the name specified in the `module-info` class or JAR file loaded. If a module name is not specified or inferred, then objects are loaded into the `unnamed` module. |
| `-encoding` | Identifies the source file encoding for the compiler, overriding the matching value, if any, in the `JAVA$OPTIONS` table. Values are the same as for the `javac -encoding` option. If you do not specify an encoding on the command line or in the `JAVA$OPTIONS` table, then the encoding is assumed to be the value returned by:<br>`System.getProperty("file.encoding");`<br>This option is relevant only when loading a source file. |
| `-fileout` *file* | Displays all message to the designated file. |
| `-force` | Forces files to be loaded, even if they match digest table entries. |

**Table 12-2 (Cont.) loadjava Argument Summary**

| Argument | Description |
|---|---|
| `-genmissing` | Determines what classes and methods are referred to by the classes that the `loadjava` tool is asked to process. Any classes not found in the database or file arguments are called missing classes. This option generates dummy definitions for missing classes containing all the referred methods. It then loads the generated classes into the database. This processing happens before the class resolution.<br>Because detecting references from source is more difficult than detecting references from class files, and because source is not generally used for distributing libraries, the `loadjava` tool will not attempt to do this processing for source files.<br>The schema in which the missing classes are loaded will be the one specified by the `-user` option, even when referring classes are created in some other schema. The created classes will be flagged so that tools can recognize them. In particular, this is needed, so that the verifier can recognize the generated classes. |
| `-genmissingjar` *jar_file* | Performs the same actions as `-genmissing`. In addition, it creates a JAR file, *jar_file*, that contains the definitions of any generated classes. |
| `-grant` | Grants the `EXECUTE` privilege on loaded classes to the listed users. Any number and combination of user names can be specified, separated by commas, but not spaces.<br>Granting the `EXECUTE` privilege on an object in another schema requires that the original `CREATE PROCEDURE` privilege was granted with the `WITH GRANT` options.<br>**Note:**<br>• `-grant` is a cumulative option. Users are added to the list of those with the `EXECUTE` privilege.<br>• The schema name should be used in uppercase.<br>• The `-grant` option causes the `loadjava` tool to grant `EXECUTE` privileges to classes, sources, and resources. However, it does not cause it to revoke any privileges. To remove privileges, use the `-cleargrants` option. |
| `-help` | Displays usage message on how to use the `loadjava` tool and its options. |
| `-jarasresource` | Loads the whole JAR file into the schema as a resource, instead of unpacking the JAR file and loading each class within it.[1] |
| `-noaction` | Takes no action on the files. Actions include creating the schema objects, granting execute permissions, and so on. The typical use is within an option file to suppress creation of specific classes in a JAR. When used on the command line, unless overridden in the option file, it will cause the `loadjava` tool to ignore all files. Except that JAR files will still be examined to determine if they contain a `META-INF/loadjava-options` entry. If so, then the option file is processed. The `-action` option in the option file will override the `-noaction` option specified on the command line. |
| `-recursivejars` | Specifies that the `loadjava` tool should process the contained JAR files as if they were top-level JAR files. That is, it should read their entries and load classes, sources, and resources. Usually, if the `loadjava` tool encounters an entry in a JAR with a `.jar` extension, it loads the entry as a resource. |
| `-norecursivejars` | Treats the JAR files contained in other JAR files as resources. This is the default behavior. This option is used to override the `-recursivejars` option. |

**Table 12-2 (Cont.) loadjava Argument Summary**

| Argument | Description |
|---|---|
| `-nosynonym` | Specifies that a public synonym for the classes should not be created. This is the default behavior. This overrides the `-synonym` option. |
| `-nousage` | Suppresses the specified usage message, when either no option is specified or the `-help` option is specified. |
| `-noverify` | Causes the classes to be loaded without byte code verification. `oracle.aurora.security.JServerPermission(Verifier) must be granted` to use this option. To be effective, this option must be used in conjunction with `-resolve`. |
| `-oci \| -oci8` | Directs the `loadjava` tool to communicate with the database using the JDBC Oracle Call Interface (OCI) driver. `-oci` and `-thin` are mutually exclusive. If neither is specified, then `-oci` is used by default. Choosing `-oci` implies the syntax of the `-user` value. You do not need to provide the URL. |
| `-optionfile` *file* | Specifies the file can be provided with `loadjava` options. |
| `-optiontable` *tablename* | Works like the `-optionfile` option, except that the source for the patterns and options is a SQL table rather than a file. |
| `-publish` *package* | Specifies the *package* to be created or replaced by the `loadjava` tool. Wrappers for the eligible methods are defined in this package. Through the use of option files, a single invocation of the `loadjava` tool can be instructed to create more than one package. Each package will undergo the same name transformations as the methods. |
| `-pubmain` *number* | Specifies a special case applied to methods with a single argument, which is of type `java.lang.String[]`. Multiple variants of the SQL procedure or function will be created, each of which takes a different number of arguments of type `VARCHAR`. In particular, variants are created taking all arguments up to and including *number*. The default value is `3`. This option applies to `main`, as well as any method that has exactly one argument of type `java.lang.String[]`. |
| `-resolve` | Compiles, if necessary, and resolves external references in classes after all classes on the command line have been loaded. If you do not specify the `-resolve` option, the `loadjava` tool loads files, but does not compile or resolve them. |
| `-resolveonly` | Causes the `loadjava` tool to skip the initial creation step. It will still perform grants, resolves, create synonyms, and so on. |

**Table 12-2 (Cont.) loadjava Argument Summary**

| Argument | Description |
|---|---|
| `-schema` *schema_name* | Designates the schema where schema objects are created. If not specified, then the `-user` schema is used. To create a schema object in a schema that is not your own, you must have the following privileges:<br>• `CREATE TABLE` or `CREATE ANY TABLE`<br>• `CREATE INDEX` or `CREATE ANY INDEX`<br>• `SELECT ANY TABLE`<br>• `UPDATE ANY TABLE`<br>• `INSERT ANY TABLE`<br>• `DELETE ANY TABLE`<br>• `CREATE PROCEDURE or CREATE ANY PROCEDURE`<br>• `ALTER ANY PROCEDURE`<br>Finally, you must have the `JServerPermission loadLibraryInClass` for the class.<br>**Note:** The above-mentioned privileges allow the grantee to create and manipulate tables in any schema except the SYS schema. For security reasons, Oracle recommends that you use these settings only with great caution. |
| `-stdout` | Causes the output to be directed to `stdout`, rather than to `stderr`. |
| `-stoponerror` | Stops processing when an error occurs. Usually, if an error occurs while the `loadjava` tool is processing files, it issues a message and continue to process other classes. In addition, it reports all errors that apply to Java objects and are contained in the `USER_ERROR` table of the schema in which classes are being loaded.<br>This option does not report `ORA-29524` errors. These are errors that are generated when a class cannot be resolved, because a referred class could not be resolved. Therefore, these errors are a secondary effect of whatever caused a referred class to be unresolved. |
| `-synonym` | Creates a `PUBLIC` synonym for loaded classes making them accessible outside the schema into which they are loaded. To specify this option, you must have the `CREATE PUBLIC SYNONYM` privilege. If `-synonym` is specified for source files, then the classes compiled from the source files are treated as if they had been loaded with `-synonym`. |
| `-tableschema` *schema* | Creates the `loadjava` tool internal tables within the specified schema, rather than in the Java file destination schema. |
| `-thin` | Directs the `loadjava` tool to communicate with the database using the JDBC Thin driver. Choosing `-thin` implies the syntax of the `-user` value. You do need to specify the appropriate URL through the `-user` option. |
| `-unresolvedok` | Ignores unresolved errors,when combined with the `-resolve` option. |
| `-user` | Specifies a user name, password, and database connection string. The files will be loaded into this database instance. |
| `-verbose` | Directs the `loadjava` tool to display detailed status messages while running. Use the `-verbose` option to learn when the `loadjava` tool does not load a file, because it matches a digest table entry. |
| `-jarsasdbobjects` | Creates a separate Database object containing the entire content of the JAR file, in addition to the database objects that are created for the classes and the resources in the JAR file. Database JAR objects are always created for module JARs and multi-release JARs. |

**Table 12-2 (Cont.) loadjava Argument Summary**

| Argument | Description |
|---|---|
| `-prependjarnames` | Prepends the name of the class or resource, when loading classes and resources from a JAR file. For module JARs, use the JAR file name rather than the module name as the database object prefix. |

[1] If you load a JAR file in this manner, then you *cannot* use it for resolution or execution.

## 12.6.3 loadjava Tool Argument Details

This section describes the details of some of the `loadjava` tool arguments whose behavior is more complex than the summary descriptions contained in the *loadjava Argument Summary* table.

### File Names

You can specify as many `.class`, `.java`, `.jar`, `.zip`, and resource files as you want and in any order. If you specify a JAR or ZIP file, then the `loadjava` tool processes the files in the JAR or ZIP. There is no JAR or ZIP schema object. If a JAR or ZIP contains another JAR or ZIP, the `loadjava` tool does not process them.

The best way to load files is to put them in a JAR or ZIP and then load the archive. Loading archives avoids the resource schema object naming complications. If you have a JAR or ZIP that works with the Java Development Kit (JDK), then you can be sure that loading it with the `loadjava` tool will also work, without having to learn anything about resource schema object naming.

Schema object names are different from file names, and the `loadjava` tool names different types of schema objects differently. Because class files are self-identifying, the mapping of class file names to schema object names done by the `loadjava` tool is invisible to developers. Source file name mapping is also invisible to developers. The `loadjava` tool gives the schema object the fully qualified name of the first class defined in the file. JAR and ZIP files also contain the names of their files.

However, resource files are not self identifying. The `loadjava` tool generates Java resource schema object names from the literal names you supply as arguments. Because classes use resource schema objects and the correct specification of resources is not always intuitive, it is important that you specify resource file names correctly on the command line.

The perfect way to load individual resource files correctly is to run the `loadjava` tool from the top of the package tree and specify resource file names relative to that directory.

> **Note:**
>
> The top of the package tree is the directory you would name in a `CLASSPATH`.

If you do not want to follow this rule, then observe the details of resource file naming that follow. When you load a resource file, the `loadjava` tool generates the resource schema object name from the resource file name, as literally specified on the command line. For example, if you type:

```
% cd /home/HR/javastuff
% loadjava options alpha/beta/x.properties
% loadjava options /home/HR/javastuff/alpha/beta/x.properties
```

Although you have specified the same file with a relative and an absolute path name, the `loadjava` tool creates two schema objects, `alpha/beta/x.properties` and `ROOT/home/HR/javastuff/alpha/beta/x.properties`. The name of the resource schema object is generated from the file name as entered.

Classes can refer to resource files relatively or absolutely. To ensure that the `loadjava` tool and the class loader use the same name for a schema object, enter the name on the command line, which the class passes to `getResource()` or `getResourceAsString()`.

Instead of remembering whether classes use relative or absolute resource names and changing directories so that you can enter the correct name on the command line, you can load resource files in a JAR, as follows:

```
% cd /home/HR/javastuff
% jar -cf alpharesources.jar alpha/*.properties
% loadjava options alpharesources.jar
```

To simplify the process further, place both the class and resource files in a JAR, which makes the following invocations equivalent:

```
% loadjava options alpha.jar
% loadjava options /home/HR/javastuff/alpha.jar
```

The preceding `loadjava` tool commands imply that you can use any path name to load the contents of a JAR file. Even if you run the redundant commands, the `loadjava` tool would realize from the digest table that it need not load the files twice. This implies that reloading JAR files is not as time-consuming as it might seem, even when few files have changed between the different invocations of the `loadjava` tool.

#### definer

```
{-definer | -d}
```

This option is identical to the definer rights in stored procedures and is conceptually similar to the UNIX `setuid` facility. However, you can apply the `-definer` option to individual classes, in contrast to `setuid`, which applies to a complete program. Moreover, different definers may have different privileges. Because an application can consist of many classes, you must apply `-definer` with care to achieve the desired results. That is, classes run with the privileges they need, but no more.

#### noverify

```
[-noverify]
```

This option causes the classes to be loaded without bytecode verification. `oracle.aurora.security.JServerPermission(Verifier)` must be granted to run this option. Also, this option must be used in conjunction with `-resolve`.

The verifier ensures that incorrectly formed Java binaries cannot be loaded for running on the server. If you know that the JAR or classes you are loading are valid, then the use of this option will speed up the process associated with the `loadjava` tool. Some Oracle Database-specific optimizations for interpreted performance are put in place during the verification process. Therefore, the interpreted performance of your application may be adversely affected by using this option.

### optionfile

```
[-optionfile <file>]
```

This option enables you to specify a file with different options that you can specify with the `loadjava` tool. This file is read and processed by the `loadjava` tool before any other `loadjava` tool options are processed. The file can contain one or more lines, each of which contains a pattern and a sequence of options. Each line must be terminated by a newline character (`\n`).

For each file or JAR entry that is processed by the `loadjava` tool, the long name of the schema object that is going to be created is checked against the patterns. Patterns can end in a wildcard (`*`) to indicate an arbitrary sequence of characters, or they must match the name exactly.

Options to be applied to matching Java schema objects are supplied on the rest of the line. Options are appended to the command-line options, they do not replace them. In case more than one line matches a name, the matching rows are sorted by length of pattern, with the shortest first, and the options from each row are appended. In general, the `loadjava` tool options are not cumulative. Rather, later options override earlier ones. This means that an option specified on a line with a longer pattern will override a line with a shorter pattern.

This file is parsed by a `java.io.StreamTokenizer`.

You can use Java comments in this file. A line comment begins with a `#`. Empty lines are ignored. The quote character is a double quote ("). That is, options containing spaces should be surrounded by double quotes. Certain options, such as `-user` and `-verbose`, affect the overall processing of the `loadjava` tool and not the actions performed for individual Java schema objects. Such options are ignored if they appear in an option file.

To help package applications, the `loadjava` tool looks for the `META-INF/loadjava-options` entry in each JAR it processes. If it finds such an entry, then it treats it as an options file that is applied for all other entries in the JAR file. However, the `loadjava` tool does some processing on entries in the order in which they occur in the JAR.

If the `loadjava` tool has partially processed entities before it processes `META-INF/loadjava-options`, then it attempts to patch up the schema object to conform to the applicable options. For example, the `loadjava` tool alters classes that were created with invoker rights when they should have been created with definer rights. The fix for `-noaction` is to drop the created schema object. This yields the correct effect, except that if a schema object existed before the `loadjava` tool started, then it would have been dropped.

### optiontable

```
[-optiontable table_name]
```

This option enables you to specify the properties of classes persistently. No mechanism is provided for loading the table. The table name must contain three character columns, PATTERN, OPTION, and VALUE. The value of PATTERN is interpreted in the same way as a pattern in an option file. The other two columns are the same as the corresponding command-line options and take an operand. Suppose, you create a table FOO with the following command:

```
create table foo (pattern varchar2(2000), option_name varchar2(2000), value
varchar2(2000));
```

Then, you can use the `optiontable` option in the following way:

```
loadjava -optiontable foo myjar.jar
```

For options that do not take an operand, the VALUE column should be NULL. The rows are processed in the same way as the lines of an option file are processed. To determine the options for a given schema object, the rows are examined and for any match the option is appended to the list of options. If two rows have the same pattern and contradictory options, such as -synonym and -nosynonym, then it is unspecified which will prevail. If two rows have the same pattern and option columns, then it is unspecified which VALUE will prevail.

### publish

```
[-publish <package>]
[-pubmain <number>]
```

The publishing options cause the `loadjava` tool to create PL/SQL wrappers for methods contained in the processed classes. Typically, a user wants to publish wrappers for only a few classes in a JAR. These options are most useful when specified in an option file.

To be eligible for publication, the method must satisfy the following:

- It must be a member of a `public` class.
- It must be declared `public` and `static`.
- The method signature should satisfy the following rules so that it can be mapped:
  - Java arithmetic types for arguments and return values are mapped to `NUMBER`.
  - `char` as an argument and return type is mapped to `VARCHAR`.
  - `java.lang.String` as an argument and return type is mapped to `VARCHAR`.
  - If the only argument of the method has type `java.lang.String`, special rules apply, as listed in the `-pubmain` option description.
  - If the return type is `void`, then a procedure is created.
  - If the return type is an arithmetic, `char`, or `java.lang.String` type, then a function is created.

Methods that take arguments or return types that are not covered by the preceding rules are not eligible. No provision is made for `OUT` and `IN OUT` SQL arguments, `OBJECT` types, and many other SQL features.

### resolve

```
{-resolve | -r}
```

Use `-resolve` to force the `loadjava` tool to compile and resolve a class that has previously been loaded. It is not necessary to specify `-force`, because resolution is performed after, and independent of, loading.

### resolver

```
{-resolver | -R} resolver_specification
```

This option associates an explicit resolver specification with the class schema objects that the `loadjava` tool creates or replaces.

A resolver specification consists of one or more items, each of which consists of a name specification and a schema specification expressed in the following syntax:

```
"((name_spec schema_spec) [(name_spec schema_spec)] ...)"
```

A name specification is similar to a name in an `import` statement. It can be a fully qualified Java class name or a package name whose final element is the wildcard character asterisk (`*`) or simply an asterisk (`*`). However, the elements of a name specification must be separated by slashes (`/`), not periods (`.`). For example, the name specification `a/b/*` matches all classes whose names begin with `a.b`. The special name `*` matches all class names.

A schema specification can be a schema name or the wildcard character dash (`-`). The wildcard does not identify a schema, but directs the resolve operation not to mark a class invalid, because a reference to a matching name cannot be resolved. Use dash (`-`) when you must test a class that refers to a class you cannot or do not want to load. For example, GUI classes that a class refers to but does not call, because when run in the server there is no GUI.

When looking for a schema object whose name matches the name specification, the resolution operation looks in the schema named by the partner schema specification.

The resolution operation searches schemas in the order in which the resolver specification lists them. For example,

```
-resolver '((* HR) (* PUBLIC))'
```

This implies that search for any reference first in `HR` and then in `PUBLIC`. If a reference is not resolved, then mark the referring class invalid and display an error message.

Consider the following example:

```
-resolver "((* HR) (* PUBLIC) (my/gui/* -))"
```

This implies that search for any reference first in `HR` and then in `PUBLIC`. If the reference is to a class in the package `my.gui` and is not found, then mark the referring class valid and do not display an error. If the reference is not to a class in `my.gui` and is not found, then mark the referring class invalid and produce an error message.

**user**

```
{-user | -u} user/password[@database_url]
```

By default, the `loadjava` tool loads into the logged in schema specified by the `-user` option. You use the `-schema` option to specify a different schema to load into. This does not require you to log in to that schema, but does require that you have sufficient permissions to alter the schema.

The permissible forms of `@database_url` depend on whether you specify `-oci` or `-thin`, as described:

- `-oci:@database_url` is optional. If you do not specify, then the `loadjava` tool uses the user's default database. If specified, `database_url` can be a TNS name or an Oracle Net Services name-value list.
- `-thin:@database_url` is required. The format is `host:lport:SID`.

  where:

  - `host` is the name of the computer running the database.
  - `lport` is the listener port that has been configured to listen for Oracle Net Services connections. In a default installation, it is `5521`.
  - `SID` is the database instance identifier. In a default installation, it is `ORCL`.

The following are examples of the `loadjava` tool commands:

- Connect to the default database with the default OCI driver, load the files in a JAR into the `TEST` schema, and then resolve them:

```
loadjava -u joe -resolve -schema TEST ServerObjects.jar
Password: password
```

- Connect with the JDBC Thin driver, load a class and a resource file, and resolve each class:

```
loadjava -thin -u HR@dbhost:5521:orcl \
  -resolve alpha.class beta.props
Password: password
```

- Add Betty and Bob to the users who can run `alpha.class`:

```
loadjava -thin -schema test -u HR@localhost:5521:orcl \
  -grant BETTY,BOB alpha.class
Password: password
```

**jarsasdbobjects**

This option indicates that JARs processed by the current `loadjava` tool are to be stored in the database along with the classes they contain, and knowledge of the association between the classes and the JAR is to be retained in the database. In other words, this argument indicates that the JARs processed by the current `loadjava` tool are to be stored in the database as database resident JARs.

**prependjarnames**

This option is used with the `-jarsasdbobjects` option. This option enables classes with the same names coming from different JARs to coexist in the same schema.

**Related Topics**

- Overview of Controlling the Current User

# 12.7 The dropjava Tool

The `dropjava` tool is the converse of the `loadjava` tool. It transforms command-line file names and JAR or ZIP file contents to schema object names, drops the schema objects, and deletes their corresponding digest table rows. You can enter `.java`, `.class`, `.zip`, `.jar`, and resource file names on the command line and in any order.

Alternatively, you can specify a schema object name directly to the `dropjava` tool. A command-line argument that does not end in `.jar`, `.zip`, `.class`, `.java` is presumed to be a schema object name. If you specify a schema object name that applies to multiple schema objects, then all will be removed.

Dropping a class invalidates classes that depend on it, recursively cascading upwards. Dropping a source drops classes derived from it.

> **Note:**
>
> If you load a JAR using the `-jarsasdbobjects (-prependjarnames)` option, then you must specify the `-prependjarnames` argument to successfully remove the JAR.

You can run the `dropjava` tool either from the command line or by using the `dropjava` method in the `DBMS_JAVA` class. To run the `dropjava` tool from within your Java application, use the following command:

```
call dbms_java.dropjava('... options...');
```

The options are the same as specified on the command line. Separate each option with a space. Do not separate the options using commas. The only exception to this is the `-resolver` option. The connection is always made to the current session. Therefore, you cannot specify another user name through the `-user` option.

For `-resolver`, you should specify all other options first, a comma (`,`), then the `-resolver` option with its definition. Do not specify the `-thin`, `-oci`, `-user`, and `-password` options, because they relate to the database connection for the `loadjava` tool. The output is directed to `stderr`. Set `serveroutput` on and call `dbms_java.set_output`, as appropriate.

This section covers the following topics:

- dropjava Tool Syntax
- dropjava Tool Argument Summary
- dropjava Tool Argument Details
- List Based Deletion
- About Dropping Resources Using dropjava Tool

## 12.7.1 dropjava Tool Syntax

The syntax of the `dropjava` tool command is:

```
dropjava [options] {file.java | file.class |
file.jar | file.zip | resourcefile} ...
  -u | -user user/[password][@database]
  [-genmissingjar JARfile]
  [-jarasresource]
  [-module module-name]
  [-automatic]

  [-o | -oci | -oci8]
  [-optionfile file]
  [-optiontable table_name]
  [-S | -schema schema]
  [-stdout]
  [-s | -synonym]
  [-t | -thin]

  [-v | -verbose]
  [-jarsasdbobjects]
  [-prependjarnames]
  [-list]
[-listfile]
```

## 12.7.2 dropjava Tool Argument Summary

Table 12-3 summarizes the `dropjava` tool arguments.

**Table 12-3 dropjava Argument Summary**

| Argument | Description |
|---|---|
| `-user` | Specifies a user name, password, and optional database connection string. The files will be dropped from this database instance. |
| `automatic` | Drops a JAR file that was loaded with the `-automatic` option. |
| `-module <name>` | Drops the classes from a named module. This option is generally not required, except in the circumstances where the `-module` option was required by `loadjava`. |
| *`filenames`* | Specifies any number and combination of `.java`, `.class`, `.jar`, `.zip`, and resource file names. |
| `-genmissingjar` *`JARfile`* | Treats the operand of this option as a file to be processed. |
| `-jarasresource` | Drops the whole JAR file, which was previously loaded as a resource. |
| `-jarsasdbobjects` | Drop the JAR files that were loaded with the `-jarsasdbobjects` option. |
| `-oci \| -oci8` | Directs the `dropjava` tool to connect with the database using the OCI JDBC driver. The `-oci` and the `-thin` options are mutually exclusive. If neither is specified, then the `-oci` option is used by default. Choosing the `-oci` option implies the form of the `-user` value. |
| `-optionfile` *`file`* | Has the same usage as for the `loadjava` tool. |
| `-optiontable` *`table_name`* | Has the same usage as for `loadjava`. |
| `-prependjarnames` | Drops the JAR files that were loaded with the `-prependjarnames` option. |
| `-schema` *`schema`* | Designates the schema from which schema objects are dropped. If not specified, then the logon schema is used. To drop a schema object from a schema that is not your own, you need the `DROP ANY PROCEDURE` and `UPDATE ANY TABLE` privileges. |
| `-stdout` | Causes the output to be directed to `stdout`, rather than to `stderr`. |
| `-synonym` | Drops a `PUBLIC` synonym that was created with the `loadjava` tool. |
| `-thin` | Directs the `dropjava` tool to communicate with the database using the JDBC Thin driver. Choosing the `-thin` option implies the form of the `-user` value. |
| `-verbose` | Directs the `dropjava` tool to emit detailed status messages while running. |
| `-list` | Drops the classes, Java source, or resources listed on the command line without them being present on the client machine or server machine. |
| `-listfile` | Reads a file and drops the classes, Java source, or resources listed in the file without them being present on the client machine or the server machine. The file contains the internal representation of the complete class, Java source, or resource name one per line. |

## 12.7.3 dropjava Tool Argument Details

This section describes a few of the `dropjava` tool arguments, which are complex.

### File Names

The `dropjava` tool interprets most file names as the `loadjava` tool does:

- `.class` files

Finds the class name in the file and drops the corresponding schema object.

- `.java` files

  Finds the first class name in the file and drops the corresponding schema object.

- `.jar` and `.zip` files

  Processes the archived file names as if they had been entered on the command line.

If a file name has another extension or no extension, then the `dropjava` tool interprets the file name as a schema object name and drops all source, class, and resource objects that match the name.

If the `dropjava` tool encounters a file name that does not match a schema object, then it displays a message and processes the remaining file names.

**user**

`{-user | -u} user/password[@database]`

The permissible forms of `@database` depend on whether you specify `-oci` or `-thin`:

- `-oci:@database` is optional. If you do not specify, then the `dropjava` tool uses the user's default database. If specified, then `database` can be a TNS name or an Oracle Net Services name-value list.
- `-thin:@database` is required. The format is `host:lport:SID`.

  where:

  - `host` is the name of the computer running the database.
  - `lport` is the listener port that has been configured to listen for Oracle Net Services connections. In a default installation, it is `5521`.
  - `SID` is the database instance identifier. In a default installation, it is `ORCL`.

The following are examples of the `dropjava` tool command:

- Drop all schema objects in the `TEST` schema in the default database that were loaded from `ServerObjects.jar`:

  ```
  dropjava -u HR -schema TEST ServerObjects.jar
  Password: password
  ```

- Connect with the JDBC Thin driver, then drop a class and a resource file from the user's schema:

  ```
  dropjava -thin -u HR@dbhost:5521:orcl alpha.class beta.props
  Password: password
  ```

**List Based Deletion**

Earlier versions of the `dropjava` tool required that the classes, JARs, source, and resources be present on the machine, where the client or server side utility is running. The current version of `dropjava` has an option that enables you to drop classes, resources, or sources based on a list of classes, which may not exist on the client machine or the server machine. This list can be either on the command line or in a text file. For example:

```
dropjava -list -u HR -v this.is.my.class this.is.your.class
Password: password
```

The preceding command drops the classes `this.is.my.class` and `this.is.your.class` listed on the command line without them being present on the client machine or server machine.

```
dropjava -listfile my.list -u HR -s -v
Password: password
```

The preceding command drops classes, resources, or sources and their synonyms based on a list of classes listed in `my.list` and displays verbosely.

> **Note:**
>
> The `'-install'` flag ignores the loading and dropping of system owned schema objects that cannot be modified.
>
> These schema objects are the runtime classes, and resources provided by the `CREATE JAVA COMMAND`.

## 12.7.4 About Dropping Resources Using dropjava Tool

You must be careful if you are removing a resource that was loaded directly into the server. The fully qualified schema object name of a resource that was generated on the client and loaded directly into the server, depends on path information in the `.jar` file or that specified on the command line at the time you loaded it. If you use a `.jar` file to load resources and use the same `.jar` file to remove resources, then there are no problems. However, if you use the command line to load resources, then you must be careful to specify the same path information when you run the `dropjava` tool to remove the resources.

# 12.8 The ojvmjava Tool

The `ojvmjava` tool is an interactive interface to the session namespace of a database instance. You specify database connection arguments when you start the `ojvmjava` tool. It then presents you with a prompt to indicate that it is ready for commands.

The shell can launch an executable, that is, a class with a `static main()` method. This is done either by using the command-line interface or by calling a database resident class. If you call a database resident class, the executable must be loaded with the `loadjava` tool.

This section covers the following topics:

- ojvmjava Tool Syntax
- ojvmjava Tool Argument Summary
- ojvmjava Tool Example
- ojvmjava Tool Functionality

## 12.8.1 ojvmjava Tool Syntax

The syntax of the `ojvmjava` tool command is:

```
ojvmjava {-user user[/password@database ] [options]
  [@filename]
  [-batch]
  [-c | -command command args]
  [-debug]
  [-d | -database conn_string]
  [-fileout filename]
```

```
  [-o | -oci | -oci8]
  [-oschema schema]
  [-t | -thin]
  [-version | -v]
 -runjava [server_file_system]
 -jdwp port [host]
 -verbose
```

## 12.8.2 ojvmjava Tool Argument Summary

Table 12-4 summarizes the `ojvmjava` tool arguments.

**Table 12-4 ojvmjava Argument Summary**

| Argument | Description |
|---|---|
| `-user \| -u` | Specifies user name for connecting to the database. This name is not case-sensitive. The name will always be converted to uppercase. If you provide the database information, then the default syntax used is OCI. You can also specify the default database. |
| `-password \| -p` | Specifies the password for connecting to the database. |
| `@filename` | Specifies a script file that contains the `ojvmjava` tool commands to be run. |
| `-batch` | Disables all messages displayed to the screen. No help messages or prompts will be displayed. Only responses to commands entered are displayed. |
| `-command` | Runs the desired command. If you do not want to run the `ojvmjava` tool in interpretive mode, but only want to run a single command, then run it with this option followed by a string that contains the command and the arguments. Once the command runs, the `ojvmjava` tool exits. |
| `-debug` | Displays debugging information. |
| `-d \| -database conn_string` | Provides a database connection string. |
| `-fileout file` | Redirects output to the provided file. |
| `-o \| -oci \| -oci8` | Uses the JDBC OCI driver. The OCI driver is the default. This flag specifies the syntax used in either the `@database` or `-database` option. |
| `-o schema schema` | Uses this schema for class lookup. |
| `-t \| -thin` | Specifies that the database syntax used is for the JDBC Thin driver. The database connection string must be of the form `host:port:SID` or an Oracle Net Services name-value list. |
| `-verbose` | Displays the connection information. |
| `-version` | Shows the version. |
| `-runjava` | Uses `DBMS_JAVA.runjava` when executing Java commands. With no argument, interprets `-classpath` as referring to the client file system. With argument `server_file_system` interprets `-classpath` as referring to the file system on which Oracle server is running, as `DBMS_JAVA.runjava` typically does. |
| `-jdwp` | Makes the connection listen for a debugger connection on the indicated port and host. The default value of host is `localhost`. |

## 12.8.3 ojvmjava Tool Example

Open a shell on the session namespace of the database `orcl` on listener port `2481` on the host `dbserver`, as follows:

```
ojvmjava -thin -user HR@dbserver:2481:orcl
Password: password
```

## 12.8.4 ojvmjava Tool Functionality

The `ojvmjava` tool commands span several different types of functionality, which are grouped as follows:

- ojvmjava Tool Command-Line Options
- ojvmjava Tool Shell Commands

### 12.8.4.1 ojvmjava Tool Command-Line Options

This section describes the following options available with the `ojvmjava` tool command:

- Scripting the ojvmjava Tool Commands in the @filename Option
- -runjava
- -jdwp

**Scripting the ojvmjava Tool Commands in the @filename Option**

This `@filename` option designates a script file that contains one or more `ojvmjava` tool commands. The specified script file is located on the client. The `ojvmjava` tool reads the file and runs all commands on the designated server. In addition, because the script file is run on the server, any interaction with the operating system in the script file, such as redirecting output to a file or running another script, occurs on the server. If you direct the `ojvmjava` tool to run another script file, then this file must exist in `$ORACLE_HOME` on the server.

You must enter the `ojvmjava` tool command followed by any options and any expected input arguments. The script file contains the `ojvmjava` tool command followed by options and input parameters. The input parameters can be passed to the `ojvmjava` tool on the command line. The `ojvmjava` tool processes all known options and passes on any other options and arguments to the script file.

The following shows the contents of the script file, `execShell`:

```
java myclass a b c
```

To run this file, use the following command:

```
ojvmjava -user HR -thin -database dbserver:2481:orcl @commands
Password: password
```

The `ojvmjava` tool processes all options that it knows about and passes along any other input parameters to be used by the commands that exist within the script file. In this example, the parameters are passed to the `java` command in the script file.

You can add any comments in your script file using the hash sign (`#`). Comments are ignored by the `ojvmjava` tool. For example:

```
#this whole line is ignored by ojvmjava
```

**-runjava**

This option controls whether or not the `ojvmjava` tool shell command Java runs executable classes using the command-line interface or database resident classes. When the `-runjava` option is present the command-line interface is used. Otherwise, the executable must be a database resident class that was previously loaded with the `loadjava` tool. Using the optional argument `server_file_system` means that the `-classpath` terms are on the file system of the machine running Oracle server. Otherwise, they are interpreted as being on the file system of the machine running the `ojvmjava` tool.

**-jdwp**

This option specifies a debugger connection to listen for when the shell command `java` is used to run an executable. This allows for debugging the executable. The arguments specify the port and host. The default value of the host argument is `localhost`. These are used to execute a call to `DBMS_DEBUG_JDWP.CONNECT_TCP` from the RDBMS session, in which the executable is run.

**Related Topics**

- About Using the Command-Line Interface

## 12.8.4.2 ojvmjava Tool Shell Commands

This section describes the following commands available within the `ojvmjava` shell:

- echo
- exit
- help
- java
- version
- whoami
- connect
- runjava
- jdwp

> **Note:**
>
> An error is reported if you enter an unsupported command.

The following table summarizes the commands that share one or more common options:

**Table 12-5 ojvmjava Command Common Options**

| Option | Description |
|---|---|
| `-describe \| -d` | Summarizes the operation of the tool. |
| `-help \| -h` | Summarizes the syntax of the tool. |

**Table 12-5 (Cont.) ojvmjava Command Common Options**

| Option | Description |
|---|---|
| `-version` | Shows the version. |

### echo

This command displays to `stdout` exactly what is indicated. This is used mostly in script files.

The syntax is as follows:

```
echo [echo_string] [args]
```

*`echo_string`* is a string that contains the text you want written to the screen during the shell script invocation and *`args`* are input arguments from the user. For example, the following command displays out a notification:

```
echo "Adding an owner to the schema" &1
```

If the input argument is `HR`, then the output would be:

```
Adding an owner to the schema HR
```

### exit

This command terminates `ojvmjava`. The syntax is as follows:

```
exit
```

For example, to leave a shell, use the following command:

```
$ exit
%
```

### help

This command summarizes the syntax of the shell commands. You can also use the help command to summarize the options for a particular command. The syntax is as follows:

```
help [command]
```

### java

This command is analogous to the JDK `java` command. It calls the static `main()` method of a class. It does this either by using the command-line interface or using a database resident class, depending on the setting of the `runjava` mode. In the latter case, the class must have been previously loaded with the `loadjava` tool. The command provides a convenient way to test Java code that runs in the database. In particular, the command catches exceptions and redirects the standard output and standard error of the class to the shell, which displays them as with any other command output. The destination of standard out and standard error for Java classes that run in the database is one or more database server process trace files, which are inconvenient and may require `DBA` privileges to read.

The syntax of the command with `runjava` mode `off` is:

```
java [-schema schema] class [arg1 ... argn]
```

The syntax of the command with `runjava` mode `on` is:

```
java [command-line options] class [arg1 ... argn]
```

where, command-line options can be any of those mentioned in Table 3-1.

The following table summarizes the arguments of this command.

**Table 12-6 java Argument Summary**

| Argument | Description |
|---|---|
| *class* | Names the Java class schema object that is to be run. |
| -schema | Names the schema containing the class to be run. The default is the invoker's schema. The schema name is case-sensitive. |
| *arg1 ... argn* | Arguments to the `static main()` method of the class. |

Consider the following Java file, `World.java`:

```
package hello;
public class World
{
  public World()
  {
    super();
  }

  public static void main(String[] argv)
  {
    System.out.println("Hello from Oracle Database");
    if (argv.length != 0)
      System.out.println("You supplied " + argv.length + " arguments: ");
    for (int i = 0; i < argv.length; i++)
      System.out.println(" arg[" + i + "] : " + argv[i]);
  }
}
```

You can compile, load, publish, and run the class, as follows:

```
% javac hello/World.java

% loadjava -r -user HR@localhost:2481:orcl hello/World.class
Password: password
% ojvmjava -user HR -database localhost:2481:orcl
Password: password
$ java hello.World alpha beta
Hello from Oracle Database
You supplied 2 arguments:
arg[0] : alpha
arg[1] : beta
```

### version

This command shows the version of the `ojvmjava` tool. You can also show the version of a specified command. The syntax of this command is:

```
version [options] [command]
```

For example, you can display the version of the shell, as follows:

```
$ version
1.0
```

### whoami

This command displays the user name of the user who logged in to the current session. The syntax of the command is:

```
whoami
```

### connect

This command enables the client to drop the current connection and connect to different databases without having to reinvoke the `ojvmjava` tool with a different connection description.

The syntax of this command is:

```
connect [-service service] [-user user][-password password]
```

You can use this command as shown in the following examples:

```
connect -s thin@locahost:5521:orcl -u HR/<password>
connect -s oci@locahost:5521:orcl -u HR -p <password>
```

The following table summarizes the arguments of this command.

**Table 12-7 connect Argument Summary**

| Argument | Description |
|---|---|
| -service \| -s | Any valid JDBC driver URLS, namely, oci @<connection descriptor> and thin@<host:port:db> |
| `-user \| -u` | User to connect as |
| -password \| -p | Password to connect with |

### runjava

This command queries or modifies the `runjava` mode. The `runjava` mode determines whether or not the `java` command uses the command-line interface to run executables. The `java` command:

- Uses the command-like interface when `runjava` mode is `on`
- Uses database resident executables when `runjava` mode is `off`

Using the `runjava` command with no arguments displays the current setting of `runjava` mode.

The following table summarizes the arguments of this command.

**Table 12-8 runjava Argument Summary**

| Argument | Description |
|---|---|
| `off` | Turns `runjava` mode off. |
| `on` | Turns `runjava` mode on. |
| `server_file_system` | Turns `runjava` mode on. Using this option means that `-classpath` terms are on the file system of the machine running Oracle server. Otherwise, they are interpreted as being on the file system of the machine running the `ojvmjava` tool. |

### jdwp

This command queries or modifies whether and how a debugger connection is listened for when an executable is run by the Java command.

> **Note:**
>
> The RDBMS session, prior to starting the executable, executes a `DBMS_DEBUG_JDWP.CONNECT_TCP` call with the specified port and host. This is called **Listening.**

Using this command with no arguments displays the current setting.

The following table summarizes the arguments of this command.

**Table 12-9 jdwp Argument Summary**

| Argument | Description |
|---|---|
| off | Stops listening in future executables. |
| `port` | Enables listening and specifies the port to be used. |
| host | Enables listening and specifies the host to be used. The default value for this argument is `localhost`. |