Chapter 8: The Java Command - 'java' and 'javaw'

Section 8.1: Entry point classes

A Java entry-point class has a `main` method with the following signature and modifiers:

```
public static void main(String[] args)
```

> Sidenote: because of how arrays work, it can also be `(String args[])`

When the `java` command starts the virtual machine, it loads the specified entry-point classes and tries to find `main`. If successful, the arguments from command line are converted to Java `String` objects and assembled into an array. If `main` is invoked like this, the array will *not* be **`null`** and won't contain any **`null`** entries.

A valid entry-point class method must do the following:

- Be named `main` (case-sensitive)
- Be **`public`** and **`static`**
- Have a **`void`** return type
- Have a single argument with an array `String[]`. The argument must be present and no more than one argument is allowed.
- Be generic: type parameters are not allowed.
- Have a non-generic, top-level (not nested or inner) enclosing class

It is conventional to declare the class as **`public`** but this not strictly necessary. From Java 5 onward, the `main` method's argument type may be a `String` varargs instead of a string array. `main` can optionally throw exceptions, and its parameter can be named anything, but conventionally it is `args`.

JavaFX entry-points

From Java 8 onwards the `java` command can also directly launch a JavaFX application. JavaFX is documented in the JavaFX tag, but a JavaFX entry-point must do the following:

- Extend `javafx.application.Application`
- Be **`public`** and not **`abstract`**
- Not be generic or nested
- Have an explicit or implicit **`public`** no-args constructor

Section 8.2: Troubleshooting the 'java' command

This example covers common errors with using the 'java' command.

"Command not found"

If you get an error message like:

```
java: command not found
```

when trying to run the `java` command, this means that there is no `java` command on your shell's command search path. The cause could be:

- you don't have a Java JRE or JDK installed at all,
- you have not updated the `PATH` environment variable (correctly) in your shell initialization file, or
- you have not "sourced" the relevant initialization file in the current shell.

Refer to "Installing Java" for the steps that you need to take.

"Could not find or load main class"

This error message is output by the `java` command if it has been unable to find / load the entry-point class that you have specified. In general terms, there are three broad reasons that this can happen:

- You have specified an entry point class that does not exist.
- The class exists, but you have specified it incorrectly.
- The class exists and you have specified it correctly, but Java cannot it find it because the classpath is incorrect.

Here is a procedure to diagnose and solve the problem:

1. Find out the full name of the entry-point class.
    - If you have source code for a class, then the full name consists of the package name and the simple class name. The instance the "Main" class is declared in the package "com.example.myapp" then its full name is "com.example.myapp.Main".
    - If you have a compiled class file, you can find the class name by running `javap` on it.
    - If the class file is in a directory, you can infer the full class name from the directory names.
    - If the class file is in a JAR or ZIP file, you can infer the full class name from the file path in the JAR or ZIP file.
2. Look at the error message from the `java` command. The message should end with the full class name that `java` is trying to use.
    - Check that it exactly matches the full classname for the entry-point class.
    - It should not end with ".java" or ".class".
    - It should not contain slashes or any other character that is not legal in a Java identifier1.
    - The casing of the name should exactly match the full class name.
3. If you are using the correct classname, make sure that the class is actually on the classpath:
    - Work out the pathname that the classname maps to; see Mapping classnames to pathnames
    - Work out what the classpath is; see this example: Different ways to specify the classpath
    - Look at each of the JAR and ZIP files on the classpath to see if they contain a class with the required pathname.
    - Look at each directory to see if the pathname resolves to a file within the directory.

If checking the classpath by hand did not find the issue, you could add the `-Xdiag` and `-XshowSettings` options. The former lists all classes that are loaded, and the latter prints out settings that include the effective classpath for the JVM.

Finally, there are some *obscure* causes for this problem:

- An executable JAR file with a `Main-Class` attribute that specifies a class that does not exist.
- An executable JAR file with an incorrect `Class-Path` attribute.
- If you mess up2 the options before the classname, the `java` command may attempt to interpret one of them

as the classname.
- If someone has ignored Java style rules and used package or class identifiers that differ only in letter case, and you are running on a platform that treats letter case in filenames as non-significant.
- Problems with homoglyphs in class names in the code or on the command line.

"Main method not found in class <name>"

This problem happens when the `java` command is able to find and load the class that you nominated, but is then unable to find an entry-point method.

There are three possible explanations:

- If you are trying to run an executable JAR file, then the JAR's manifest has an incorrect "Main-Class" attribute that specifies a class that is not a valid entry point class.
- You have told the `java` command a class that is not an entry point class.
- The entry point class is incorrect; see Entry point classes for more information.

Other Resources

- What does "Could not find or load main class" mean?
- http://docs.oracle.com/javase/tutorial/getStarted/problems/index.html

1 - From Java 8 and later, the `java` command will helpfully map a filename separator ("/" or "") to a period (".") However, this behavior is not documented in the manual pages.

2 - A really obscure case is if you copy-and-paste a command from a formatted document where the text editor has used a "long hyphen" instead of a regular hyphen.

Section 8.3: Running a Java application with library dependencies

This is a continuation of the "main class" and "executable JAR" examples.

Typical Java applications consist of an application-specific code, and various reusable library code that you have implemented or that has been implemented by third parties. The latter are commonly referred to as library dependencies, and are typically packaged as JAR files.

Java is a dynamically bound language. When you run a Java application with library dependencies, the JVM needs to know where the dependencies are so that it can load classes as required. Broadly speaking, there are two ways to deal with this:

- The application and its dependencies can be repackaged into a single JAR file that contains all of the required classes and resources.
- The JVM can be told where to find the dependent JAR files via the runtime classpath.

For an executable JAR file, the runtime classpath is specified by the "Class-Path" manifest attribute. *(Editorial Note: This should be described in a separate Topic on the `jar` command.)* Otherwise, the runtime classpath needs to be supplied using the `-cp` option or using the `CLASSPATH` environment variable.

For example, suppose that we have a Java application in the "myApp.jar" file whose entry point class is `com.example.MyApp`. Suppose also that the application depends on library JAR files "lib/library1.jar" and "lib/library2.jar". We could launch the application using the `java` command as follows in a command line:

```
$ # Alternative 1 (preferred)
```

```
$ java -cp myApp.jar:lib/library1.jar:lib/library2.jar com.example.MyApp

$ # Alternative 2
$ export CLASSPATH=myApp.jar:lib/library1.jar:lib/library2.jar
$ java com.example.MyApp
```

(On Windows, you would use `;` instead of `:` as the classpath separator, and you would set the (local) `CLASSPATH` variable using `set` rather than `export`.)

While a Java developer would be comfortable with that, it is not "user friendly". So it is common practice to write a simple shell script (or Windows batch file) to hide the details that the user doesn't need to know about. For example, if you put the following shell script into a file called "myApp", made it executable, and put it into a directory on the command search path:

```
#!/bin/bash
# The 'myApp' wrapper script

export DIR=/usr/libexec/myApp
export CLASSPATH=$DIR/myApp.jar:$DIR/lib/library1.jar:$DIR/lib/library2.jar
java com.example.MyApp
```

then you could run it as follows:

```
$ myApp arg1 arg2 ...
```

Any arguments on the command line will be passed to the Java application via the `"$@"` expansion. (You can do something similar with a Windows batch file, though the syntax is different.)

Section 8.4: Java Options

The `java` command supports a wide range of options:

- All options start with a single hyphen or minus-sign (-): the GNU/Linux convention of using `--` for "long" options is not supported.
- Options must appear before the **`<classname>`** or the `-jar <jarfile>` argument to be recognized. Any arguments after them will be treated as arguments to be passed to Java app that is being run.
- Options that do not start with `-X` or `-XX` are standard options. You can rely on all Java implementations1 to support any standard option.
- Options that start with `-X` are non-standard options, and may be withdrawn from one Java version to the next.
- Options that start with `-XX` are advanced options, and may also be withdrawn.

Setting system properties with `-D`

The `-D<property>=<value>` option is used to set a property in the system `Properties` object. This parameter can be repeated to set different properties.

Memory, Stack and Garbage Collector options

The main options for controlling the heap and stack sizes are documented in Setting the Heap, PermGen and Stack sizes. *(Editorial note: Garbage Collector options should be described in the same topic.)*

Enabling and disabling assertions

The `-ea` and `-da` options respectively enable and disable Java **`assert`** checking:

- All assertion checking is disabled by default.
- The `-ea` option enables checking of all assertions
- The `-ea:<packagename>...` enables checking of assertions in a package *and all subpackages*.
- The `-ea:<classname>...` enables checking of assertions in a class.
- The `-da` option disables checking of all assertions
- The `-da:<packagename>...` disables checking of assertions in a package *and all subpackages*.
- The `-da:<classname>...` disables checking of assertions in a class.
- The `-esa` option enables checking for all system classes.
- The `-dsa` option disables checking for all system classes.

The options can be combined. For example.

```
$ # Enable all assertion checking in non-system classes
$ java -ea -dsa MyApp

$ # Enable assertions for all classes in a package except for one.
$ java -ea:com.wombat.fruitbat... -da:com.wombat.fruitbat.Brickbat MyApp
```

Note that enabling to assertion checking is liable to alter the behavior of a Java programming.

- It is liable make the application slower in general.
- It can cause specific methods to take longer to run, which could change timing of threads in a multi-threaded application.
- It can introduce serendipitous *happens-before* relations which can cause memory anomalies to disappear.
- An incorrectly implemented **`assert`** statement could have unwanted side-effects.

Selecting the VM type

The `-client` and `-server` options allow you to select between two different forms of the HotSpot VM:

- The "client" form is tuned for user applications and offers faster startup.
- The "server" form is tuned for long running applications. It takes longer capturing statistic during JVM "warm up" which allows the JIT compiler to do a better of job of optimizing the native code.

By default, the JVM will run in 64bit mode if possible, depending on the capabilities of the platform. The `-d32` and `-d64` options allow you to select the mode explicitly.

1 - Check the official manual for the `java` command. Sometimes a *standard* option is described as "subject to change".

Section 8.5: Spaces and other special characters in arguments

First of all, the problem of handling spaces in arguments is NOT actually a Java problem. Rather it is a problem that needs to be handled by the command shell that you are using when you run a Java program.

As an example, let us suppose that we have the following simple program that prints the size of a file:

```
import java.io.File;

public class PrintFileSizes {
```

```
    public static void main(String[] args) {
        for (String name: args) {
            File file = new File(name);
            System.out.println("Size of '" + file + "' is " + file.size());
        }
    }
}
```

Now suppose that we want print the size of a file whose pathname has spaces in it; e.g. `/home/steve/Test File.txt`. If we run the command like this:

```
$ java PrintFileSizes /home/steve/Test File.txt
```

the shell won't know that `/home/steve/Test File.txt` is actually one pathname. Instead, it will pass 2 distinct arguments to the Java application, which will attempt to find their respective file sizes, and fail because files with those paths (probably) do not exist.

Solutions using a POSIX shell

POSIX shells include `sh` as well derivatives such as `bash` and `ksh`. If you are using one of these shells, then you can solve the problem by *quoting* the argument.

```
$ java PrintFileSizes "/home/steve/Test File.txt"
```

The double-quotes around the pathname tell the shell that it should be passed as a single argument. The quotes will be removed when this happens. There are a couple of other ways to do this:

```
$ java PrintFileSizes '/home/steve/Test File.txt'
```

Single (straight) quotes are treated like double-quotes except that they also suppress various expansions within the argument.

```
$ java PrintFileSizes /home/steve/Test\ File.txt
```

A backslash escapes the following space, and causes it not to be interpreted as an argument separator.

For more comprehensive documentation, including descriptions of how to deal with other special characters in arguments, please refer to the quoting topic in the Bash documentation.

Solution for Windows

The fundamental problem for Windows is that at the OS level, the arguments are passed to a child process as a single string (source). This means that the ultimate responsibility of parsing (or re-parsing) the command line falls on either program or its runtime libraries. There is lots of inconsistency.

In the Java case, to cut a long story short:

- You can put double-quotes around an argument in a `java` command, and that will allow you to pass arguments with spaces in them.
- Apparently, the `java` command itself is parsing the command string, and it gets it more or less right
- However, when you try to combine this with the use of `SET` and variable substitution in a batch file, it gets really complicated as to whether double-quotes get removed.

- The `cmd.exe` shell apparently has other escaping mechanisms; e.g. doubling double-quotes, and using `^` escapes.

For more detail, please refer to the Batch-File documentation.

Section 8.6: Running an executable JAR file

Executable JAR files are the simplest way to assemble Java code into a single file that can be executed. *(Editorial Note: Creation of JAR files should be covered by a separate Topic.) *

Assuming that you have an executable JAR file with pathname **`<jar-path>`**, you should be able to run it as follows:

```
java -jar <jar-path>
```

If the command requires command-line arguments, add them after the **`<jar-path>`**. For example:

```
java -jar <jar-path> arg1 arg2 arg3
```

If you need to provide additional JVM options on the `java` command line, they need to go *before* the `-jar` option. Note that a `-cp` / `-classpath` option will be ignored if you use `-jar`. The application's classpath is determined by the JAR file manifest.

Section 8.7: Running a Java applications via a "main" class

When an application has not been packaged as an executable JAR, you need to provide the name of an entry-point class on the `java` command line.

Running the HelloWorld class

The "HelloWorld" example is described in Creating a new Java program . It consists of a single class called `HelloWorld` which satisfies the requirements for an entry-point.

Assuming that the (compiled) "HelloWorld.class" file is in the current directory, it can be launched as follows:

```
java HelloWorld
```

Some important things to note are:

- We must provide the name of the class: not the pathname for the ".class" file or the ".java" file.
- If the class is declared in a package (as most Java classes are), then the class name we supply to the `java` command must be the full classname. For instance if `SomeClass` is declared in the `com.example` package, then the full classname will be `com.example.SomeClass`.

Specifying a classpath

Unless we are using in the `java -jar` command syntax, the `java` command looks for the class to be loaded by searching the classpath; see The Classpath. The above command is relying on the default classpath being (or including) the current directory. We can be more explicit about this by specifying the classpath to be used using the `-cp` option.

```
java -cp . HelloWorld
```

This says to make the current directory (which is what "." refers to) the sole entry on the classpath.

The `-cp` is an option that is processed by the `java` command. All options that are intended for the `java` command should be before the classname. Anything after the class will be treated as an command line argument for the Java application, and will be passed to application in the `String[]` that is passed to the `main` method.

(If no `-cp` option is provided, the `java` will use the classpath that is given by the `CLASSPATH` environment variable. If that variable is unset or empty, `java` uses "." as the default classpath.)