Chapter 5: Java Compiler - 'javac'

Section 5.1: The 'javac' command - getting started

Simple example

Assuming that the "HelloWorld.java" contains the following Java source:

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello world!");
    }
}
```

(For an explanation of the above code, please refer to Getting started with Java Language .)

We can compile the above file using this command:

```
$ javac HelloWorld.java
```

This produces a file called "HelloWorld.class", which we can then run as follows:

```
$ java HelloWorld
Hello world!
```

The key points to note from this example are:

1. The source filename "HelloWorld.java" must match the class name in the source file ... which is `HelloWorld`. If they don't match, you will get a compilation error.
2. The bytecode filename "HelloWorld.class" corresponds to the classname. If you were to rename the "HelloWorld.class", you would get an error when your tried to run it.
3. When running a Java application using `java`, you supply the classname NOT the bytecode filename.

Example with packages

Most practical Java code uses packages to organize the namespace for classes and reduce the risk of accidental class name collision.

If we wanted to declare the `HelloWorld` class in a package call `com.example`, the "HelloWorld.java" would contain the following Java source:

```
package com.example;

public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello world!");
    }
}
```

This source code file needs to stored in a directory tree whose structure corresponds to the package naming.

```
.    # the current directory (for this example)
|
 ----com
     |
```

```
      ----example
          |
           ----HelloWorld.java
```

We can compile the above file using this command:

```
$ javac com/example/HelloWorld.java
```

This produces a file called "com/example/HelloWorld.class"; i.e. after compilation, the file structure should look like this:

```
.    # the current directory (for this example)
|
 ----com
     |
      ----example
          |
           ----HelloWorld.java
           ----HelloWorld.class
```

We can then run the application as follows:

```
$ java com.example.HelloWorld
Hello world!
```

Additional points to note from this example are:

1. The directory structure must match the package name structure.
2. When you run the class, the full class name must be supplied; i.e. "com.example.HelloWorld" not "HelloWorld".
3. You don't have to compile and run Java code out of the current directory. We are just doing it here for illustration.

Compiling multiple files at once with 'javac'.

If your application consists of multiple source code files (and most do!) you can compile them one at a time. Alternatively, you can compile multiple files at the same time by listing the pathnames:

```
$ javac Foo.java Bar.java
```

or using your command shell's filename wildcard functionality ....

```
$ javac *.java
$ javac com/example/*.java
$ javac */**/*.java #Only works on Zsh or with globstar enabled on your shell
```

This will compile all Java source files in the current directory, in the "com/example" directory, and recursively in child directories respectively. A third alternative is to supply a list of source filenames (and compiler options) as a file. For example:

```
$ javac @sourcefiles
```

where the `sourcefiles` file contains:

```
Foo.java
Bar.java
```

```
com/example/HelloWorld.java
```

Note: compiling code like this is appropriate for small one-person projects, and for once-off programs. Beyond that, it is advisable to select and use a Java build tool. Alternatively, most programmers use a Java IDE (e.g. NetBeans, eclipse, IntelliJ IDEA) which offers an embedded compiler and incremental building of "projects".

Commonly used 'javac' options

Here are a few options for the `javac` command that are likely to be useful to you

- The `-d` option sets a destination directory for writing the ".class" files.
- The `-sourcepath` option sets a source code search path.
- The `-cp` or `-classpath` option sets the search path for finding external and previously compiled classes. For more information on the classpath and how to specify it, refer to the The Classpath Topic.
- The `-version` option prints the compiler's version information.

A more complete list of compiler options will be described in a separate example.

References

The definitive reference for the `javac` command is the Oracle manual page for `javac`.

Section 5.2: Compiling for a different version of Java

The Java programming language (and its runtime) has undergone numerous changes since its release since its initial public release. These changes include:

- Changes in the Java programming language syntax and semantics
- Changes in the APIs provided by the Java standard class libraries.
- Changes in the Java (bytecode) instruction set and classfile format.

With very few exceptions (for example the **`enum`** keyword, changes to some "internal" classes, etc), these changes are backwards compatible.

- A Java program that was compiled using an older version of the Java toolchain will run on a newer version Java platform without recompilation.
- A Java program that was written in an older version of Java will compile successfully with a new Java compiler.

Compiling old Java with a newer compiler

If you need to (re-)compile older Java code on a newer Java platform to run on the newer platform, you generally don't need to give any special compilation flags. In a few cases (e.g. if you had used **`enum`** as an identifier) you could use the `-source` option to disable the new syntax. For example, given the following class:

```
public class OldSyntax {
    private static int enum;  // invalid in Java 5 or later
}
```

the following is required to compile the class using a Java 5 compiler (or later):

```
$ javac -source 1.4 OldSyntax.java
```

Compiling for an older execution platform

If you need to compile Java to run on an older Java platforms, the simplest approach is to install a JDK for the oldest

version you need to support, and use that JDK's compiler in your builds.

You can also compile with a newer Java compiler, but there are complicated. First of all, there some important preconditions that must be satisfied:

- The code you are compiling must not use Java language constructs that were not available in the version of Java that you are targeting.
- The code must not depend on standard Java classes, fields, methods and so on that were not available in the older platforms.
- Third party libraries that the code depends must also be built for the older platform and available at compile-time and run-time.

Given the preconditions are met, you can recompile code for an older platform using the `-target` option. For example,

```
$ javac -target 1.4 SomeClass.java
```

will compile the above class to produce bytecodes that are compatible with Java 1.4 or later JVM. (In fact, the `-source` option implies a compatible `-target`, so `javac -source 1.4 ...` would have the same effect. The relationship between `-source` and `-target` is described in the Oracle documentation.)

Having said that, if you simply use `-target` or `-source`, you will still be compiling against the standard class libraries provided by the compiler's JDK. If you are not careful, you can end up with classes with the correct bytecode version, but with dependencies on APIs that are not available. The solution is to use the `-bootclasspath` option. For example:

```
$ javac -target 1.4 --bootclasspath path/to/java1.4/rt.jar SomeClass.java
```

will compile against an alternative set of runtime libraries. If the class being compiled has (accidental) dependencies on newer libraries, this will give you compilation errors.