Compiling and Running Simple Java Code

Here is the obligatory "hello world" application:

```
public class T {
    public static void main(String[] args) {
        System.out.println("Salut, Le Monde");
    }
}
```

which I have placed in ~/USF/CS601/code/tools/T.java.

Jumping into ~/USF/CS601/code/tools, I can compile T.java with the javac tool:

```
$ cd ~/USF/CS601/code/tools
$ javac T.java
```

which leaves T.class in the current directory.

To run the T.main() method, use the java interpreter/run-time-compiler tool:

```
$ java T
Salut, Le Monde
$
```

where java takes the name of a Java class, T, not the name of the file it is in, T.java. The java interpreter looks for T.class and then begins execution at the main() method within. Assume for now that java looks for T.class and any other class files in the current directory.

Now, lets use two Java files; one, U.java:

```
public class U {
    public static void main(String[] args) {
        Hello.speak();
    }
}
```

that refers to a method in the new file, Hello.java:

```
public class Hello {
    public static void speak() {
        System.out.println("hello");
    }
}
```

To compile these two files, use

```
$ javac U.java Hello.java
```

or, more generally:

```
$ javac *.java
$ ls | more
Hello.class
Hello.java
T.class
T.java
U.class
U.java
```

Now you can run U's main() method via:

```
$ java U
hello
$
```

CLASSPATH Environment Variable

Jump into /tmp and now try to run U:

```
$ cd /tmp
$ java U
Exception in thread "main" java.lang.NoClassDefFoundError: U
```

The problem is that the Java interpreter does not know where to find U.class. You can specify where to look directly:

```
$ java -classpath ~/USF/CS601/code/tools T
Salut, Le Monde
$ java -classpath ~/USF/CS601/code/tools U
hello
```

The notion of a class path is similar to the UNIX and DOS PATH environment variable that tells the command line where to find applications and tools.

In general, however, you will not want to repeat the path each time. Java looks for a CLASSPATH environment variable and uses it as a colon-separated (on UNIX; PC's use semicolon I think) list of directories. These directories are understood to potentially contain your code. For example, I could set my CLASSPATH to include directory ~/USF/CS601/code/tools:

```
$ echo $CLASSPATH   # what is it set to currently?
.:/home/parrt/lib/antlr-2.7.2.jar:/home/parrt/lib/jguru.util.jar:...
$ export CLASSPATH="$CLASSPATH:/home/parrt/tmp"  # append
$ echo $CLASSPATH   # what is it now?
.:/home/parrt/lib/antlr-2.7.2.jar:/home/parrt/lib/jguru.util.jar:...:~/USF/CS601/code/tools
```

Note that you should generally include dot ('.'), the current directory, in your CLASSPATH.

Anyway, now from /tmp, you can run your programs without the explicit java option:

```
$ cd /tmp
$ java T
Salut, Le Monde
$ java U
hello
$
```

Packages and Directory Structure

Most of the time, Java code is organized into packages just like you organize your general files in your home directory. In fact, there is a prefix convention that is the reversal of your domain as the root package. For example, my ANTLR research code uses package org.antlr and jGuru uses com.jguru. For our purposes here, we will use a single package and move our test files into it.

Modify the above code to live in package foo:

```
$ cd ~/USF/CS601/code/tools
$ mkdir foo
$ cp *.java foo
$ cd foo
$ vi *.java
```

Where, with your editor, you have added

package foo;
as the first line of every file. Also add "foo: " to the strings so that we can tell which version of our code is executing. So Hello.java looks like:

```
package foo;

public class Hello {
    public static void speak() {
        System.out.println("foo: hello");
    }
```

}
Note the obvious correlation between package and directory: if your class T is in package a.b.c then it must be in file a/b/c/T.java.

Compile as before (except you are in the foo) subdirectory:

```
$ javac *.java
```

Now, try to run the simplest class T:

```
$ java T
Exception in thread "main" java.lang.NoClassDefFoundError: T (wrong name: foo/T)
        at java.lang.ClassLoader.defineClass0(Native Method)
        at java.lang.ClassLoader.defineClass(ClassLoader.java:502)
...
```

Java started looking for class T, which it knows will be in T.class. It found T.class in the current directory (remember '.' is in your CLASSPATH first), but the compiled code showed that it is actually class foo.T not T.

Besides, you want to run the new class which is foo.T not T. Try that:

```
$ cd ~/USF/CS601/code/tools/foo
$ java foo.T
Exception in thread "main" java.lang.NoClassDefFoundError: foo/T
```

Now, java is looking for foo.T in a file called foo/T.class. You are in foo, which has no foo subdirectory. If you move up a directory, then file foo/T.class will be found when you ask for class foo.T:

```
$ cd ..
$ pwd   # print working directory
/Users/parrt/tmp
$ java foo.T
foo: Salut, Le Monde
$ java foo.U
foo: hello
```

To execute foo.T, java uses the following algorithm:

1. Convert class to filename; foo.T -> foo/T.class
1. Look for a directory in the class path that contains directory foo
1. Find file T.class inside directory foo.
1. Execute method main() within

Jar'ing Java Code

Tool jar (java archive) is analogous the UNIX tar utility and packs a bunch of

Java source, Java .class files, resources etc... into a single file for easy deployment.

Without packages

To make a jar of the simple test files in ~/USF/CS601/code/tools, jump inside and say

```
$ jar cvf /tmp/tools.jar *.class
added manifest
adding: Hello.class(in = 385) (out= 271)(deflated 29%)
adding: T.class(in = 411) (out= 286)(deflated 30%)
adding: U.class(in = 283) (out= 220)(deflated 22%)
```

This "cvf /tmp/tools.jar" (create, verbose, filename ...) option jars up all the .class files and puts the jar file in /tmp/tools.jar. To see what is inside, use tvf:

```
$ jar tvf /tmp/tools.jar
     0 Tue Sep 02 14:44:44 PDT 2003 META-INF/
    70 Tue Sep 02 14:44:44 PDT 2003 META-INF/MANIFEST.MF
   385 Tue Sep 02 14:05:50 PDT 2003 Hello.class
   411 Tue Sep 02 13:55:06 PDT 2003 T.class
   283 Tue Sep 02 14:05:50 PDT 2003 U.class
```

With packages

When you have packages, you must be careful what directory you are in. Recall that class foo.T must be in file foo/T.class. So to ja up the second round of packaged examples, you would jar everything up for the same directory as before (that is, the directory containing foo):

```
$ jar cvf /tmp/foo.jar foo
added manifest
adding: foo/(in = 0) (out= 0)(stored 0%)
adding: foo/Hello.class(in = 389) (out= 276)(deflated 29%)
adding: foo/Hello.java(in = 106) (out= 94)(deflated 11%)
adding: foo/T.class(in = 415) (out= 290)(deflated 30%)
adding: foo/T.java(in = 124) (out= 113)(deflated 8%)
adding: foo/U.class(in = 291) (out= 227)(deflated 21%)
adding: foo/U.java(in = 100) (out= 90)(deflated 10%)
$ jar tvf /tmp/foo.jar
     0 Tue Sep 02 14:47:44 PDT 2003 META-INF/
    70 Tue Sep 02 14:47:44 PDT 2003 META-INF/MANIFEST.MF
     0 Tue Sep 02 14:34:38 PDT 2003 foo/
   389 Tue Sep 02 14:34:50 PDT 2003 foo/Hello.class
   106 Tue Sep 02 14:34:30 PDT 2003 foo/Hello.java
   415 Tue Sep 02 14:34:50 PDT 2003 foo/T.class
   124 Tue Sep 02 14:34:32 PDT 2003 foo/T.java
   291 Tue Sep 02 14:34:50 PDT 2003 foo/U.class
   100 Tue Sep 02 14:34:36 PDT 2003 foo/U.java
```

~~~
```

## Executing Java Within a Jar

To execute code from within a jar, make sure the jar file is in your class path
    (either explicitly from the command line as shown here or from your CLASSPATH
    environment variable).

```
$ cd some-random-directory
$ java -classpath /tmp/tools.jar T
foo: Salut, Le Monde
$ java -classpath /tmp/tools.jar U
foo: hello
```

To access the code in a package, the same syntax is used:

```
$ java -classpath /tmp/foo.jar foo.T
foo: Salut, Le Monde
~/USF/CS601/code/tools $ java -classpath /tmp/foo.jar foo.U
foo: hello
```

In essence, a jar file is simply mimicking file system directory structure.

In recent versions of Java, you can also use the ``-jar`` option to directly run
    a predefined main program from within a jar.  The ``META-INF/MANIFEST.MF``
    file within the jar must have ``Main-Class:`` property set. For example,
    here's one for my ANTLR jar:

```
$ cat META-INF/MANIFEST.MF
Manifest-Version: 1.0
Implementation-Title: ANTLR 4 Tool
Implementation-Version: 4.4
Built-By: parrt
Build-Jdk: 1.6
Created-By: http://www.bildtool.org
Main-Class: org.antlr.v4.Tool
Implementation-Vendor: ANTLR
```
Here is how to use the option that accesses that file:

```
$ java -jar /usr/local/lib/antlr-4.4-complete.jar
ANTLR Parser Generator  Version 4.4
 -o ___              specify output directory where all output is generated
 -lib ___            specify location of grammars, tokens files
 -atn                generate rule augmented transition network diagrams
 -encoding ___       specify grammar file encoding; e.g., euc-jp
...
```
~~~