Wesleyan University COMP 342: Programming Methods
Connecticut College COM 306: Software Engineering
Trinity College CPSC 415: Software Development
Classpaths, packages, and JAR files

Non-trivial Java applications (whether standalone, applets, or webapps) often package up all their classes in so-called *JAR* (Java ARchive) files. Likewise, class libraries that you might use in development will also often be packaged as JAR files. For that reason, we need to learn how to use and create them. And to do this, it helps to understand the Java classpath and packages.

To start, let's step back for a minute and try to understand what happens when you compile one class that depends on another that is defined in a different source file. For example, suppose we want to compile the application class `CounterTest.java`, which in turn depends on the `ConcreteCounter` class. Let's further assume that `CounterTest.java` looks something like this:

```
import comp342.examples.* ;
import com.cooljavastuff.randomstuff.* ;

class CounterTest {
    public static void main(String[] args) {
        ConcreteCounter c = new ConcreteCounter() ;
        ...
    }
}
```

When you try to compile `CounterTest.java` with a command like

```
$ javac CounterTest.java
```

the `javac` compiler sees that your class uses a `ConcreteCounter`. Thus the compiler needs to find the file `ConcreteCounter.class`, because that is where the definition of `ConcreteCounter` is. So where does the compiler look?

When looking for class files, the compiler has a list of directories that it looks in; this list is called the *classpath*. In fact, the classpath is broken into two pieces: the *system* classpath and the *user* classpath. The system classpath is set when Java is installed on the computer, and consists of a directory somewhere in the Java SDK installation directory. We don't have to worry about that too much. Unless you specify otherwise (as we'll see how later), the user classpath is set to the current directory (i.e., the directory from which you run `javac`). From now on, when I say "classpath," I really mean user classpath.

Now let's suppose that we have changed the user classpath to contain the following directories: `/home/joe/comp342work`, `/usr/share/java`, and the current directory.[1] The compiler does *not* look for `ConcreteCounter.class` in only these directories. Instead, it notices that we have imported the `comp342.examples` and the `com.cooljavastuff.randomstuff` packages. The compiler translates both package names into directories by replacing the `.` with `/`. It then adds these directories to the end of each of the directories in the classpath and looks for class files in the resulting directories. Thus, the compiler will look for `ConcreteCounter.class` in the following directories:

[1] I'm assuming Unix conventions here; Windows would use \ instead of `/` and might include drive specifications.

```
    /home/joe/comp342work
    /home/joe/comp342work/comp342/examples
    /home/joe/comp342work/com/cooljavastuff/randomstuff
    /usr/share/java
    /usr/share/java/comp342/examples
    /usr/share/java/com/cooljavastuff/randomstuff
    .
    ./comp342/examples
    ./com/cooljavastuff/randomstuff
```

Here, "`.`" means the current directory. Thus `ConcreteCounter.class` must live in one of these directories.

Now let's talk a little about packages. The point behind Java packages is to manage namespaces. Suppose you are writing a specialized application or library and you want a class that represents chronological dates. A good name for such a class would be `Date`. Unfortunately, Java already defines a class `Date`! Packages allow both us and Java to have our cake. In Java, all classes are in some package; the *fully qualified* name of a class consists of the package name followed by the class name. For example, the Java `Date` class is defined in the `java.util` package, so its fully qualified name is `java.util.Date`. And there is no confict for classes with *different* fully qualified names. Thus as long as we put our `Date` class in our own package (maybe called `myapp`), there will be no conflict. In source files, we can either import `java.util.Date` or `myapp.Date` as appropriate. The only time things get a bit sticky is if we want both classes; in that case, we cannot import them, and have to refer to them by fully qualified name whenever we need them (e.g., `new java.util.Date()`). To put a class in a package, the first line of the file must be a `package` statement. For example, to make `ConcreteCounter` part of the `comp342.examples` package, the source code would looks something like:

```
package comp342.examples ;
public class ConcreteCounter {
    ...
}
```

But now `javac` requires that `ConcreteCounter.class` appear in the `comp342/examples` subdirectory of one of the directories in the classpath! In other words, `javac` must find `ConcreteCounter.class` in either `/home/joe/comp342work/comp342/examples`, `/usr/share/java/comp342/examples`, or `./comp342/examples`. If it finds the file anywhere else, it will report an error. So the important point is: *class files must be placed in a directory that corresponds to the package name.*

One way to arrange that your class files end up in the right directories is to just move your compiled files into an appropriate directory hierarchy. But that is a real pain. Instead, you can use the `-d` command-line argument to `javac` to instruct it to place compiled files in a specified directory; `javac` will create subdirectories as appropriate for classes that are in packages. So if we compile `ConcreteCounter.java` with the command

```
$ javac -d classes ConcreteCounter.java
```

then `ConcreteCounter.class` will be put in `classes/comp342/examples`. You could now put `classes` in your classpath.

So, how do we set the classpath? Actually, we can only set the user classpath. One way is to use a command-line argument to the `javac` program. The argument is `-classpath` and it must be followed by a list of directories separated by colons. Thus, we might type:

```
$ javac -classpath /home/joe/comp342work:/usr/share/java:. CounterTest.java
```

Notice that this is the same kind of specification as used for the shell path. Also notice that we have put `.` (the current directory) in the classpath. This is required: if you specify a classpath, you must specify all the directories in the classpath. Another way, which is more convenient if you plan to use the same classpath many times, is to set the `CLASSPATH` environment variable. The rule that `javac` follows is that if the `-classpath` argument is specified, it uses that for the classpath. If not, but the `CLASSPATH` variable is set, then it uses the value of the variable. If neither case holds, then the user classpath is just the current directory.

Now for the last point about classpaths. Everything we learned about classpaths and the Java compiler applies to the Java Virtual Machine (`java`). In other words, when you run an application that uses classes that are not in the current directory, you must specify the classpath with either the `-classpath` command-line argument to `java` or the `CLASSPATH` environment variable. This sets the user classpath, and those paths will be added onto with package names just as `javac` does. So, if `CLASSPATH` is not set, then to run our `CounterTest` program, we would type

```
$ java -classpath /home/joe/comp342work:/usr/share/java:. CounterTest
```

So what about JAR (Java ARchive) files? When somebody wants to distribute a large number of classes, they usually package them up into a single JAR file. A JAR file is really just a ZIP file with some extra metadata. The key point for us is that *a JAR file can be a classpath entry just like an ordinary directory.* Thus, if our `CounterTest` class used some classes that were distributed as the JAR file `misclib.jar` which we saved in the directory `/home/joe/comp342work` we could add the JAR file to the classpath as follows:

```
$ javac -classpath /home/joe/comp342work:/home/joe/comp342work/misclib.jar:
        /usr/share/java:.  CounterTest.java
```

(all one line). We could also set the `CLASSPATH` environment variable correspondingly. We create JAR files with the `jar` command. Suppose we have compiled a number of source files, and all the class files are in subdirectories of the `classes` directory corresponding to package names. We can create a JAR file of all the classes with the following command executed in the `classes` directory:

```
$ jar cvf classes.jar .
```

This says to create a JAR file (the `c`) named `classes.jar` (the `f` followed by `classes.jar`) that consists of everything in this directory. The `v` says to tell us what is happening as the file is created and is not really necessary.