5 things you didn't know about ...: JARs

A Java Archive is more than just a bundle of classes

Ted Neward
Alex Theedom

May 17, 2017
(First published June 15, 2010)

Many Java™ developers never think beyond the basics of JARs — only using them to bundle classes before shipping them off to the production servers. But a JAR is more than just a renamed ZIP file. Learn how to use Java Archive files at their fullest capacity, including tips for jarring Spring dependencies and configuration files.

For most Java developers, JAR files and their specialized cousins, WARs and EARs, are simply the end result of a long Ant or Maven process. It's standard procedure to copy the JAR to the right place on the server (or, more rarely, the user's machine) and forget about it.

Actually, JARs can do more than store source code, but you have to know what else is possible, and how to ask for it. The tips in this installment of the *5 things* series will show you how to make the most of Java Archive files (and in some cases WARs and EARs, too), especially at deployment time.

Because so many Java developers use Spring (and because the Spring framework presents some particular challenges to our traditional use of JARs), several of the tips specifically address JARs in Spring applications.

About this series

So you think you know about Java programming? The fact is, most developers scratch the surface of the Java platform, learning just enough to get the job done. In this series, Ted Neward digs beneath the core functionality of the Java platform to uncover little-known facts that could help you solve even the stickiest programming challenges.

I'll start out with a quick example of a standard Java Archive file procedure, which will serve as a foundation for the tips that follow.

Put it in a JAR

Normally, you build a JAR file after your code source has been compiled, collecting the Java code (which has been segregated by package) into a single collection via either the `jar` command-line

utility, or more commonly, the Ant `jar` task. The process is straightforward enough that I won't demonstrate it here, though later in the article we'll return to the topic of how JARs are constructed. For now, we just need to archive `Hello`, a stand-alone console utility that does the incredibly useful task of printing a message to the console, shown in Listing 1:

Listing 1. Archiving the console utility

```
package com.tedneward.jars;

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

The `Hello` utility isn't much, but it's a useful scaffold for exploring JAR files, starting with executing the code.

1. JARs are executable

Languages like .NET and C++ have historically had the advantage of being OS-friendly, in that simply referencing their name at the command-line (`helloWorld.exe`) or double-clicking their icon in the GUI shell would launch the application. In Java programming, however, a launcher application —`java`— bootstraps the JVM into the process, and we have to pass a command-line argument (`com.tedneward.Hello`) indicating the class whose `main()` method we want to launch.

These additional steps make it harder to create user-friendly applications in Java. Not only does the end user have to type all of these elements at the command-line, which many end users would rather avoid, but chances are good that he or she will somehow fat-finger it and get an obscure error back.

The solution is to make the JAR file "executable" so that the Java launcher will automatically know which class to launch when executing the JAR file. All we have to do is introduce an entry into the JAR file's manifest (`MANIFEST.MF` in the JAR's `META-INF` subdirectory), like so:

Listing 2. Show me the entrypoint!

```
Main-Class: com.tedneward.jars.Hello
```

The manifest is just a set of name-value pairs. Because the manifest can sometimes be touchy about carriage returns and whitespace, it's easiest to use Ant to generate it when building the JAR. In Listing 3, I've used the `manifest` element of the Ant `jar` task to specify the manifest:

Listing 3. Build me the entrypoint!

```
    <target name="jar" depends="build">
        <jar destfile="outapp.jar" basedir="classes">
            <manifest>
                <attribute name="Main-Class" value="com.tedneward.jars.Hello" />
            </manifest>
        </jar>
    </target>
```

All a user has to do to execute the JAR file now is specify its filename on the command-line, via `java -jar outapp.jar`. In the case of some GUI shells, double-clicking the JAR file works just as well.

2. JARs can include dependency information

It seems that word of the `Hello` utility has spread, and so the need to vary the implementation has emerged. Dependency injection (DI) containers like Spring or Guice handle many of the details for us, but there's still a hitch: modifying code to include the use of a DI container can lead to results like what you see in Listing 4:

Listing 4. Hello, Spring world!

```
package com.tedneward.jars;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class Hello {

    public static void main(String... args)  {

        AnnotationConfigApplicationContext context
                          = new AnnotationConfigApplicationContext(AppConfig.class);

        Speaker speaker = context.getBean("speaker", Speaker.class);
        System.out.println(speaker.sayHello());

    }
}
```

More about Spring

This tip assumes you are familiar with dependency injection and the Spring framework.

Because the `-jar` option to the launcher overwrites whatever happens to be in the `-classpath` command-line option, Spring needs to be on the `CLASSPATH` and in the environment variable when you run this code. Fortunately, JARs permit a declaration of other JAR dependencies to appear in the manifest, which implicitly creates the CLASSPATH without you having to declare it, shown in Listing 5:

Listing 5. Hello, Spring CLASSPATH!

```
<target name="jar" depends="build">
    <jar destfile="outapp.jar" basedir="classes">
        <manifest>
            <attribute name="Main-Class" value="com.tedneward.jars.Hello" />
            <attribute name="Class-Path"
                value="./lib/org.spring-aop-5.0.0.BUILD-SNAPSHOT.jar
                  ./lib/spring-beans-5.0.0.BUILD-SNAPSHOT.jar
                  ./lib/spring-context-5.0.0.BUILD-SNAPSHOT.jar
                  ./lib/spring-core-5.0.0.BUILD-SNAPSHOT.jar
                  ./lib/spring-expression-5.0.0.BUILD-SNAPSHOT.jar
                  ./lib/commons-logging-1.2.jar" />
        </manifest>
    </jar>
</target>
```

Notice that the `Class-Path` attribute contains a relative reference to the JAR files that the application depends on. You could also write this as an absolute reference or without a prefix entirely, in which case it would be assumed that the JAR files were in the same directory as the application JAR.

Unfortunately, the `value` attribute to the Ant `Class-Path` attribute has to appear in one line, because the JAR manifest can't cope with the idea of multiple `Class-Path` attributes. So, all of those dependencies have to appear on one line in the manifest file. Sure, it's ugly, but being able to `java -jar outapp.jar` is worth it!

3. JARs can be implicitly referenced

If you have several different command-line utilities (or other applications) that make use of the Spring framework, it might be easier to put the Spring JAR files into a common location that all of the utilities can reference. Doing so avoids having multiple copies of JARs popping up all over the filesystem. The Java runtime's common location for JARs, known as the "extension directory," is by default located in the `lib/ext` subdirectory, underneath the installed JRE location.

The JRE is a customizable location, but it is so rarely customized within a given Java environment that it's entirely safe to assume that `lib/ext` is a safe place to store JARs, and that they will be implicitly available on the Java environment's `CLASSPATH`.

4. Classpath wildcards allowed

In an effort to avoid huge `CLASSPATH` environment variables (which Java developers should have left behind years ago) and/or command-line `-classpath` parameters, Java 6 introduced the notion of the *classpath wildcard*. Rather than having to launch with each and every JAR file explicitly listed on an argument, the classpath wildcard lets you specify `lib/*`, and all of the JAR files listed in that directory (not recursively), in the classpath.

Unfortunately, the classpath wildcard doesn't hold for the previously discussed `Class-Path` attribute manifest entry. But it does make it easier to launch Java applications (including servers) for developer tasks such as code-gen tools or analysis tools.

5. JARs hold more than code

So many parts of the Java ecosystem, depend on configuration files that describes how the environment should be established and it's entirely common for developers to forget to copy the configuration file alongside the JAR file.

Some configuration files are editable by a sysadmin, but a significant number of them are well outside of the sysadmin's domain, which leads to deployment bugs. A sensible solution would be to package the config file together with the code — and it's doable because a JAR is basically a ZIP in disguise. Just include config files in the Ant task or the `jar` command-line when building a JAR.

JARs can also include other types of files, not just configuration files. For instance, if my `SpeakEnglish` component wanted to access a properties file, I could set that up like Listing 6:

Listing 6. Respond at random

```
package com.tedneward.jars;

import java.util.*;

public class SpeakEnglish implements ISpeak {

    Properties responses = new Properties();
    Random random = new Random();

    public String sayHello() {

        // Pick a response at random
        int which = random.nextInt(5);

        return responses.getProperty("response." + which);
    }
}
```

Putting `responses.properties` into the JAR file means that there's one less file to worry about deploying alongside the JAR file. Doing that just requires including the `responses.properties` file during the JAR step.

Once you've stored your properties in a JAR, though, you might wonder how to get them back. If the data desired is co-located inside the same JAR file, as it is in the previous example, then don't bother trying to figure out the file location of the JAR and crack it open with a `JarFile` object. Instead, let the class's `ClassLoader` find it as a "resource" within the JAR file, using the `ClassLoader getResourceAsStream()` method shown in Listing 7:

Listing 7. ClassLoader locates a Resource

```
package com.tedneward.jars;

import java.util.*;

public class SpeakEnglish implements ISpeak {

    Properties responses = new Properties();
    // ...

    public SpeakEnglish() {

        try {
            ClassLoader myCL = SpeakEnglish.class.getClassLoader();
            responses.load(
                myCL.getResourceAsStream(
                    "com/tedneward/jars/responses.properties"));
        } catch (Exception x)  {
            x.printStackTrace();
        }
    }

    // ...
}
```

You can follow this procedure for any kind of resource: configuration file, audio file, graphics file, you name it. Virtually any file type can be bundled into a JAR, obtained as an `InputStream` (via the `ClassLoader`) and used in whatever fashion suits your fancy.

Conclusion

This article has covered the top five things most Java developers don't know about JARs — at least based on history and anecdotal evidence. Note that all of these JARs-related tips are equally true for WARs. Some tips (the `Class-Path` and `Main-Class` attributes in particular) are less exciting in the case of WARs, because the servlet environment picks up the entire contents of a directory and has a predefined entry point. Still, taken collectively, these tips move us beyond the paradigm of "Okay, start by copying everything in this directory ..." In so doing, they also make deploying Java applications much simpler.

Downloadable resources

Description	Name	Size
Sample code for this article	j-5things6-src.zip	10KB

Related topics

- JAR files revealed
- Packaging programs in JAR files
- Spring
- Dependency injection with Guice
- See JARs run