JavaOne BOF:
The Legacy Developer's Guide to Java 9

Wayne Citrin

CTO, JNBridge

October 2017

Outline

- The agony of the legacy Java developer
- Java 9 and the legacy developer
  - Multi-release JAR files
  - Modules
  - Jlink

The agony of the legacy Java developer

- Legacy Java developers are like the kids with their noses pressed up against the window of the candy store
- Attractive new APIs
  - But the code needs to run on older JREs
- New non-API language constructs and JDK features
  - Can't be used when targeting older JREs
- Users might be using the new JRE, behavior could be different despite promised backward compatibility

@ Iam Muttoo

Java 9 features that help the legacy developer

- Multi-release JAR files
- Modules
- Jlink

Multi-release JAR files

- Include code that will run against new JRE
  - Use new features, APIs
- Also include code that will run against legacy JREs
- Appropriate code is always chosen

What we do now

```
import java.lang.management.ManagementFactory;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;

public class ProcessDetector {
    private static Class processHandleClass;
    private static Method currentMethod;
    private static Method getPidMethod;

    static {
        try {
            processHandleClass = Class.forName("java.lang.ProcessHandle");
            currentMethod = processHandleClass.getMethod("current", null);
            getPidMethod = processHandleClass.getMethod("getPid", null);
        } catch (ReflectiveOperationException e) {
            processHandleClass = null;
            currentMethod = null;
            getPidMethod = null;
        }
    }
    public static long getPid() {
        if (processHandleClass == null) {   // it's pre-Java 9
            String vmName = ManagementFactory.getRuntimeMXBean().getName();
            return Long.parseLong(vmName.split("@")[0]);
        } else {
            Object currentProcessHandle;
            try {
                currentProcessHandle = currentMethod.invoke(null, null);
                Long pid = (Long) getPidMethod.invoke(currentProcessHandle, null);
                return pid.longValue();
            } catch (IllegalAccessException | IllegalArgumentException | InvocationTargetException e) {
                return -1;
            }
        }
    }
}
```

- Use late binding – probe for desired APIs, use if available
- Tedious and error-prone

How multi-release JAR files work

- Create special nooks that can only be seen by specific versions of the JRE and that override previous versions
- Manifest file contains

```
Multi-Release:  true
```

```
JAR content root
  A.class
  B.class
  C.class
  D.class
  META-INF
     MANIFEST.MF
     versions
        9
           A.class
           B.class
        10
           A.class
```

How to create a multi-release JAR file

- Place your legacy code (Java 8 and earlier) in one folder hierarchy
- Place new Java 9-targeted code in another folder hierarchy
- Use new Java 9 jar.exe tool:

```
jar cfe pid.jar -C legacyFolder . --release 9 -C java9Folder .
```

Module basics

- *Modularized jar files*: Contain module-info.class file
  - Indicates what they expose to the outside world, who can read them, and what they require from elsewhere
- *Ordinary jar files*: Plain old jar files – no module-info.class file
- In addition to the class path, there is now a module path
- Can you use your old ordinary jar files in Java 9 alongside modularized jar files?
- Can your legacy ordinary jar files participate in the module system unchanged?
- Can a multi-release jar file be modularized when used in Java 9?

Using the classpath

- The classpath works the way it always has
- Modularized jar files in the classpath are treated like any other jar file
  - If you've modularized one of your jar files, but aren't ready to actually use the module system in your application, you can do it without consequence
- All the jar files in the classpath are considered part of a single *unnamed module*
  - Exports everything to all other modules
  - Can access all other modules
- If you have a modularized application, but have some old unmodularized modules, just put them in the classpath. It'll just work!

Putting jar files in the module path

- Modularized jar files can be placed in the module path (modpath)
  - Each is treated as a module, and the runtime enforces module rules and visibility
- What happens if you put an ordinary jar file in the modpath?
  - Becomes an *automatic module*
  - It's name is the name of the jar file
  - It exposes everything and accesses everything
- If you have a legacy jar file, can almost always add it directly to the modpath and it'll just work
  - Caveat: A package can only be part of one named module
  - If a package appears in more than one jar file, only one of them can be made an automatic module. The rest must be left in the classpath and remain part of the unnamed module.

Can you use modularized jar files in Java 8 and earlier?

- It's complicated
- To have a module-info.class, you need to be targeting Java 9
  - Means that you can't use those JAR files on Java 8
- Can we put module-info.class in the versioned section of a multi-release JAR?
  - Yes!
  - Modular multi-release jar files can have module-info.class in the versioned section, if:
    - There's no module-info.class in the root section (i.e., if you're targeting Java 8), or
    - There is a module-info.class in the root section, in which case they need to conform to certain similarity rules, but that's irrelevant if you're targeting pre-Java 9

Jlink: Automatically constructing private JREs

- Problem:
  - Can we guarantee that the end user has the correct JRE to run your legacy application?
- What we do now:
  - Create a *private* JRE and ship it with the application
- Instructions on creating a private JRE come in every JRE's README file
  - What files and folders must be included
  - Which are optional
  - What folder structure needs to be enforced
- Process is error-prone. If you make a mistake, it won't work

How Jlink works

- Java 9's Jlink tool automates the process
- The Java 9 runtime is itself modularized
  - Given the modules in your application, Jlink figures out which runtime modules to include, too. Leaves the rest out.
  - When invoking Jlink, supply the module path and the modules of your application, and it'll include everything else you'll need, and will package it in a folder hierarchy (the run-time image)
- Caveats:
  - If your jar files are not modularized, Jlink will be forced to include everything. Still, it's convenient to let Jlink worry about assembling the run-time image.
  - Jlink is a Java 9 tool and only works with the modularized Java 9 JRE, so this really only supports legacy development and maintenance post-Java 9

Summary

- Java 9's designers did think about how legacy Java developers might use Java 9 features and APIs
- These features let you ease into Java 9's offerings, without breaking your legacy code, or having to rewrite things you've already written.
- Using these Java 9 features, your code should continue to run on older JREs, too!
- **Multi-release jar files**: You can include code for Java 9, and equivalent code for pre-Java 9, and your application will continue to work on all platforms.
- **Module system**: Modularize your jar files, or not. Put your jar files in the classpath, or put them in the module path. Things will just work, and they'll continue to work in Java 8 and earlier. Of course, the more modularization you do, the more benefits you'll get.
- **Jlink**: Starting with Java 9, you'll easily be able to generate tailored run-time images so that your users will have exactly the JRE they need.

Discussion

Resources

- Overview: https://techbeacon.com/legacy-developers-guide-java-9
- Multi-release JAR files:
  - Spec: http://openjdk.java.net/jeps/238
  - Tutorial: https://www.voxxed.com/2016/11/java-9-series-multi-release-jar-files/
- Modules:
  http://openjdk.java.net/projects/jigsaw/
  https://jaxenter.com/java-9-modules-jpms-basics-135885.html
- Jlink:
  - http://openjdk.java.net/jeps/220 and http://openjdk.java.net/jeps/282

SPANNING JAVA & .NET

jnbridge.com

@JNBridge