Pure Java Main Application

jim stafford

Fall 2022 v2022-08-29: Built: 2022-12-07 06:10 EST

Table of Contents

Chapter 1. Introduction

This material provides an introduction to building a bare bones Java application using a single, simple Java class, packaging that in a Java ARchive (JAR), and executing it two ways:

- as a class in the classpath
- as the Main-Class of a JAR

1.1. Goals

The student will learn:

- foundational build concepts for simple, pure-Java solution

1.2. Objectives

At the conclusion of this lecture and related exercises, the student will be able to:

1. create source code for an executable Java class
2. add that Java class to a Maven module
3. build the module using a Maven pom.xml
4. execute the application using a classpath
5. configure the application as an executable JAR
6. execute an application packaged as an executable JAR

Chapter 2. Simple Java Class with a Main

Our simple Java application starts with a public class with a static main() method that optionally accepts command-line arguments from the caller

```
package info.ejava.examples.app.build.javamain;

import java.util.List;

public class SimpleMainApp { ①
    public static final void main(String...args) { ② ③
        System.out.println("Hello " + List.of(args));
    }
}
```

① public class

② implements a static main() method

③ optionally accepts arguments

Chapter 3. Project Source Tree

This class is placed within a module source tree in the `src/main/java` directory below a set of additional directories (`info/ejava/examples/app/build/javamain`) that match the Java package name of the class (`info.ejava.examples.app.build.javamain`)

```
|-- pom.xml ①
`-- src
    |-- main ②
    |   |-- java
    |   |   `-- info
    |   |       `-- ejava
    |   |           `-- examples
    |   |               `-- app
    |   |                   `-- build
    |   |                       `-- javamain
    |   |                           `-- SimpleMainApp.java
    |   `-- resources ③
    `-- test ④
        |-- java
        `-- resources
```

① `pom.xml` will define our project artifact and how to build it

② `src/main` will contain the pre-built, source form of our artifacts that will be part of our primary JAR output for the module

③ `src/main/resources` is commonly used for property files or other resource files read in during the program execution

④ `src/test` is will contain the pre-built, source form of our test artifacts. These will not be part of the primary JAR output for the module

Chapter 4. Building the Java Archive (JAR) with Maven

In setting up the build within Maven, I am going to limit the focus to just compiling our simple Java class and packaging that into a standard Java JAR.

4.1. Add Core pom.xml Document

Add the core document with required `GAV` information (`groupId`, `artifactId`, `version`) to the `pom.xml` file at the root of the module tree. Packaging is also required but will have a default of `jar` if not supplied.

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>info.ejava.examples.app</groupId> ①
    <artifactId>java-app-example</artifactId> ②
    <version>6.0.1-SNAPSHOT</version> ③
    <packaging>jar</packaging> ④
<project>
```

① `groupId`

② `artifactId`

③ `version`

④ `packaging`

Module directory should be the same name/spelling as artifactId to align with default directory naming patterns used by plugins.

Packaging optional in this case. The default is to `jar`

4.2. Add Optional Elements to pom.xml

- name

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>
```

```
    <groupId>info.ejava.examples.app</groupId>
    <artifactId>java-app-example</artifactId>
    <version>6.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>App::Build::Java Main Example</name> ①
<project>
```

① `name` appears in Maven build output but not required

4.3. Define Plugin Versions

Define plugin versions so the module can be deterministically built in multiple environments

- Each version of Maven has a set of default plugins and plugin versions
- Each plugin version may or may not have a set of defaults (e.g., not Java 17) that are compatible with our module

```
<properties>
    <java.target.version>17</java.target.version>
    <maven-compiler-plugin.version>3.10.1</maven-compiler-plugin.version>
    <maven-jar-plugin.version>3.2.2</maven-jar-plugin.version>
</properties>

<pluginManagement>
 <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>${maven-compiler-plugin.version}</version>
      <configuration>
        <release>${java.target.version}</release>
      </configuration>
    </plugin>
  </plugins>
</pluginManagement>
```

The `jar` packaging will automatically activate the `maven-compiler-plugin` and `maven-jar-plugin`. Our definition above identifies the version of the plugin to be used (**if** used) and any desired configuration of the plugin(s).

4.4. pluginManagement vs. plugins

- Use `pluginManagement` to define a plugin **if** it activated in the module build
  - useful to promote consistency in multi-module builds
  - commonly seen in parent modules

- Use `plugins` to declare that a plugin be active in the module build
  - ideally only used by child modules
  - our child module indirectly activated several plugins by using the `jar` packaging type

Chapter 5. Build the Module

Maven modules are commonly built with the following commands/ phases

- `clean` removes previously built artifacts
- `package` creates primary artifact(s) (e.g., JAR)
  - processes main and test resources
  - compiles main and test classes
  - runs unit tests
  - builds the archive

```
$mvn clean package
[INFO] Scanning for projects...
[INFO]
[INFO] --------------< info.ejava.examples.app:java-app-example >--------------
[INFO] Building App::Build::Java App Example 6.0.1-SNAPSHOT
[INFO] --------------------------------[ jar ]---------------------------------
[INFO]
[INFO] --- maven-clean-plugin:3.2.0:clean (default-clean) @ java-app-example ---
[INFO] Deleting .../java-app-example/target
[INFO]
...
[INFO] --- maven-resources-plugin:3.2.0:resources (default-resources) @ java-app-
example ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Using 'UTF-8' encoding to copy filtered properties files.
[INFO] Copying 0 resource
[INFO]
...
[INFO] --- maven-compiler-plugin:3.10.1:compile (default-compile) @ java-app-example
---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 1 source file to .../java-app-example/target/classes
[INFO]
[INFO] --- maven-resources-plugin:3.2.0:testResources (default-testResources) @ java-
app-example ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Using 'UTF-8' encoding to copy filtered properties files.
[INFO] Copying 0 resource
[INFO]
[INFO] --- maven-compiler-plugin:3.10.1:testCompile (default-testCompile) @ java-app-
example ---
[INFO] Changes detected - recompiling the module!
[INFO]
[INFO] --- maven-surefire-plugin:3.0.0-M7:test (default-test) @ java-app-example ---
[INFO]
[INFO] --- maven-jar-plugin:3.2.2:jar (default-jar) @ java-app-example ---
```

```
[INFO] Building jar: .../java-app-example/target/java-app-example-6.0.1-SNAPSHOT.jar
[INFO]
...
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  3.428 s
```

Chapter 6. Project Build Tree

The produced build tree from `mvn clean package` contains the following key artifacts (and more)

```
|-- pom.xml
|-- src
`-- target
    |-- classes ①
    |   `-- info
    |       `-- ejava
    |           `-- examples
    |               `-- app
    |                   `-- build
    |                       `-- javamain
    |                           `-- SimpleMainApp.class
...
    |-- java-app-example-6.0.1-SNAPSHOT.jar ②
...
    `-- test-classes ③
```

① `target/classes` for built artifacts from `src/main`

② primary artifact(s) (e.g., Java Archive (JAR))

③ `target/test-classes` for built artifacts from `src/test`

Chapter 7. Resulting Java Archive (JAR)

Maven adds a few extra files to the META-INF directory that we can ignore. The key files we want to focus on are:

- `SimpleMainApp.class` is the compiled version of our application
- [META-INF/MANIFEST.MF](https://docs.oracle.com/javase/tutorial/deployment/jar/manifestindex.html) contains properties relevant to the archive

```
$ jar tf target/java-app-example-*-SNAPSHOT.jar | egrep -v "/$" | sort
META-INF/MANIFEST.MF
META-INF/maven/info.ejava.examples.app/java-app-example/pom.properties
META-INF/maven/info.ejava.examples.app/java-app-example/pom.xml
info/ejava/examples/app/build/javamain/SimpleMainApp.class
```

- `jar tf` lists the contents of the JAR
- `egrep` is being used to exclude non-files (i.e., directores) that end with "/"
- `sort` performs an ordering of the output
- `|` pipe character sends the stdout of previous command to the stdin of the next command

Chapter 8. Execute the Application

The application is executed by

- invoking the `java` command
- adding the JAR file (and any other dependencies) to the classpath
- specifying the fully qualified class name of the class that contains our main() method

Example with no arguments

```
$ java -cp target/java-app-example-*-SNAPSHOT.jar
info.ejava.examples.app.build.javamain.SimpleMainApp

Output:
Hello []
```

Example with arguments

```
$ java -cp target/java-app-example-*-SNAPSHOT.jar
info.ejava.examples.app.build.javamain.SimpleMainApp arg1 arg2 "arg3 and 4"

Output:
Hello [arg1, arg2, arg3 and 4]
```

- example passed three (3) arguments separated by spaces
  - third argument (`arg3 and arg4`) used quotes around the entire string to escape spaces and have them included in the single parameter

Chapter 9. Configure Application as an Executable JAR

To execute a specific Java class within a classpath is conceptually simple. However, there is a lot more to know than we need to when there may be only a single entry point. In the following sections we will assign a default Main-Class by using the MANIFEST.MF properties

9.1. Add Main-Class property to MANIFEST.MF

```
$ unzip -qc target/java-app-example-*-SNAPSHOT.jar META-INF/MANIFEST.MF

Manifest-Version: 1.0
Created-By: Maven JAR Plugin 3.2.2
Build-Jdk-Spec: 17
Main-Class: info.ejava.examples.app.build.javamain.SimpleMainApp
```

9.2. Automate Additions to MANIFEST.MF using Maven

One way to surgically add that property is thru the maven-jar-plugin

```
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-jar-plugin</artifactId>
      <version>${maven-jar-plugin.version}</version>
      <configuration>
        <archive>
          <manifest>
            <mainClass>
info.ejava.examples.app.build.javamain.SimpleMainApp</mainClass>
          </manifest>
        </archive>
      </configuration>
    </plugin>
```

This is a very specific plugin configuration that would only apply to a specific child module. Therefore, we would place this in a `plugins` declaration versus a `pluginsManagement` definition.

Chapter 10. Execute the JAR versus just adding to classpath

The executable JAR is executed by

- invoking the `java` command
- adding the -jar option
- adding the JAR file (and any other dependencies) to the classpath

Example with no arguments

```
$ java -jar target/java-app-example-*-SNAPSHOT.jar

Output:
Hello []
```

Example with arguments

```
$ java -jar target/java-app-example-*-SNAPSHOT.jar one two "three and four"

Output:
Hello [one, two, three and four]
```

- example passed three (3) arguments separated by spaces
    - third argument (`three and four`) used quotes around the entire string to escape spaces and have them included in the single parameter

Chapter 11. Configure pom.xml to Test

At this point we are ready to create an automated execution of our JAR as a part of the build. We have to do that after the `packaging` phase and will leverage the `integration-test` Maven phase

```
<build>
   ...
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-antrun-plugin</artifactId> ①
      <executions>
          <execution>
              <id>execute-jar</id>
              <phase>integration-test</phase> ④
              <goals>
                  <goal>run</goal>
              </goals>
              <configuration>
                  <tasks>
                      <java fork="true" classname=
"info.ejava.examples.app.build.javamain.SimpleMainApp">  ②
                          <classpath>
                              <pathelement path=
"${project.build.directory}/${project.build.finalName}.jar"/>
                          </classpath>
                          <arg value="Ant-supplied java -cp"/>
                          <arg value="Command Line"/>
                          <arg value="args"/>
                      </java>

                      <java fork="true"
                            jar=
"${project.build.directory}/${project.build.finalName}.jar">  ③
                          <arg value="Ant-supplied java -jar"/>
                          <arg value="Command Line"/>
                          <arg value="args"/>
                      </java>
                  </tasks>
              </configuration>
          </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

① Using the `maven-ant-run` plugin to execute Ant task

② Using the `java` Ant task to execute shell `java -cp` command line

③ Using the `java` Ant task to execute shell `java -jar` command line

④ Running the plugin during the `integration-phase`

- Order
  1. `package`
  2. `pre-integration`
  3. `integration-test`
  4. `post-integration`
  5. `verify`

11.1. Execute JAR as part of the build

```
$ mvn clean verify
[INFO] Scanning for projects...
[INFO]
[INFO] -------------< info.ejava.examples.app:java-app-example >--------------
...
[INFO] --- maven-jar-plugin:3.2.2:jar (default-jar) @ java-app-example -①
[INFO] Building jar: .../java-app-example/target/java-app-example-6.0.1-SNAPSHOT.jar
[INFO]
...
[INFO] --- maven-antrun-plugin:3.1.0:run (execute-jar) @ java-app-example ---
[INFO] Executing tasks ②
[INFO]      [java] Hello [Ant-supplied java -cp, Command Line, args]
[INFO]      [java] Hello [Ant-supplied java -jar, Command Line, args]
[INFO] Executed tasks
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
```

① Our plugin is executing

② Our application was executed and the results displayed

Chapter 12. Summary

1. The JVM will execute the static `main()` method of the class specified in the java command
2. The class must be in the JVM classpath
3. Maven can be used to build a JAR with classes
4. A JAR can be the subject of a java execution
5. The Java `META-INF/MANIFEST.MF` `Main-Class` property within the target JAR can express the class with the `main()` method to execute
6. The maven-jar-plugin can be used to add properties to the `META-INF/MANIFEST.MF` file
7. A Maven build can be configured to execute a JAR