CODE DSM: JAVA WEEK 11 CREATING JAR FILES

DMACC Fall 2019

Instructor: Greg Hazen

AGENDA REVIEW

WEEK 10 ASSIGNMENT

AGENDA

BUILDING JAR FILES

MAVEN

LIBRARIES

The compiled code for a text .java source file is placed in a binary .class file

The compiled Java code is called ‘bytecode’

Sometimes, we want to

- Use more than one .class file at a time
- Have a way to share multiple .class files with other developers

LIBRARIES

A Java library is a JVM bytecode archive stored in a format similar to a ZIP file called a ‘jar’

Jar – Java Archive

To use a jar in your application, you must specify it in something called ‘the classpath’

LIBRARIES

A classpath in Java denotes

- The provided jar files that an application can use at compile time
- The necessary jar files to run an application

To use a classpath

- Pass it as a parameter as part of compilation or JVM startup
- Set it as an environment variable of the OS you are using

USING A JAR IN COMPILATION

In this scenario, a jar named ‘AccountingFunctions.jar’ is necessary to compile an application

- The application uses an object that is encoded in the bytecode of AccountingFunctions
- The AccountingFunctions.jar is on a Windows OS and is located in the ‘c:\development\libraries\’ directory
- Command-line:

```
javac –cp “c:\development\libraries\AccountingFunctions.jar”
MyApp.java
```

USING A JAR AT RUNTIME

In this scenario, a jar named ‘AccountingFunctions.jar’ is necessary to run an application

- The application uses an object that is encoded in the bytecode of AccountingFunctions
- The main method for the application is in MyAppRunner
- The AccountingFunctions.jar is on a Windows OS and is located in the ‘c:\development\libraries\’ directory
- Command-line:

```
java –cp “c:\development\libraries\AccountingFunctions.jar” MyAppRunner
```

CREATING OUR OWN JARS

To make our own jar

- Compile all of our .java source files to .class bytecode files
- Use the ‘jar’ command with the path to the directory with the .class files in it

Example

- javac ./edu/dmacc/codedsm/app/*.java
- jar cf MyJar.jar ./edu/dmacc/codedsm/app/*.class

EXERCISE: OUR FIRST JAR FILE

Open git bash

- javac

You get an error because Windows doesn’t know about “javac” command

EXERCISE: ADD JAVA TO ENVIRONMENT VARS

In the Start menu type “environment variables”

Open the “Edit the system environment variables”

Click Environment Variables…

Under System Variables find path and Edit

Create a New variable for path

- C:\Program Files\Java\jdk1.8.0_221\bin

EXERCISE: OUR FIRST JAR FILE

Create a new Command Line Project using IntelliJ
- Name it ExampleNonMavenProject
- Base Package edu.dmacc.coma502
- Create a Main.java with main that prints “Hello, World!”

Open git bash
- cd /c/COMA502/code/ExampleNonMavenProject/src

Compile the code
- javac ./edu/dmacc/coma502/*.java

Use the jar command to create a jar file
- jar cf NonMavenProject.jar ./edu/dmacc/coma502/*.class

EXERCISE: OUR FIRST JAR FILE

We created

- .class in the ExampleNonMavenProject/src/edu/dmacc/coma502 folder
- .jar file in ExampleNonMavenProject/src folder

In a Windows explorer

- Rename the .jar file to .zip
- Open the .zip file to explore it

EXECUTABLE JARS

A jar can also be used as a way to create an executable program

Just like any executable program we've seen so far, you need to have a class with a 'main' method in it

Change the 'jar' command to the following, making sure to include the full package and name of the class with the 'main' method in it

```
jar cfe MyJar.jar edu.dmacc.codedsm.app.Runner
./edu/dmacc/codedsm/app/*.class
```

To execute a jar, use the command 'java –jar JarName.jar'

EXERCISE: OUR FIRST EXECUTABLE JAR FILE

Using our Command Line Project from earlier

Open git bash

- cd /c/COMA502/code/ExampleNonMavenProject/src

We already compiled a *.class file using javac

Use the jar command to create a jar file

- jar **cfe** NonMavenProject.jar **edu.dmacc.coma502.Main** ./edu/dmacc/coma502/*.class

EXERCISE: OUR FIRST EXECUTABLE JAR FILE

We created

- .jar file in ExampleNonMavenProject/src folder

Execute the JAR file

- java –jar NonMavenProject.jar

In a Windows explorer

- Rename the .jar file to .zip
- Open the .zip file to explore it

MAVEN

Working with multiple jar files in a project is common

Can get very hard to keep track of all jar files

- Even harder to make sure you have the right version of the jar file all of the time

Maven is an XML (extensible markup language) based tool that allows us to keep track of jar dependencies and help us build our complicated applications

MAVEN

Based on a concept called the ‘project object model’ or POM

Free and open source (https://maven.apache.org)

Uses a central repository of all commonly used jar dependencies (https://search.maven.org)

MAVEN

To use Maven, we make a file called ‘pom.xml’ at the root of our new project

A pom.xml file contains

- The name of the application or jar file it defines
- Metadata about the application, like author and version number
- A list of all jar dependencies needed to compile and run
- A list of ‘goals’ that help put together our application

Using a Maven pom file is called ‘building’ an application

MAVEN

Maven projects are organized a specific way

All code goes under a directory called ‘src’

- Application code then goes under ‘main/java’
- Test code then goes under ‘test/java’

Compiled code and jars goes under a directory called ‘target’

MAVEN BUILDS: LIFECYCLE

1. validate
2. compile
3. test
4. package
5. integration-test
6. verify
7. install
8. deploy

MAVEN BUILDS: LIFECYCLE

Executing a phase can be done from the command-line or your IDE

- Command line: mvn <phase>

Additionally, ‘clean’ can be added to the command to force a full rebuild of the entire project

For this class, we should always do ‘mvn clean install’ to perform a build

MAVEN BUILDS

Each phase of a build has ‘goals’ that can be associated to it

- Most goals are added to the build by using ‘plugins’
- Since a plugin can be used multiple times in the same build, the use of a goal might need to be included in an ‘execution’ block

EXERCISE: OUR FIRST MAVEN JAR FILE

Create a new Maven project in IntelliJ

- groupId: edu.dmacc.coma502
- artifactId: example-maven-project
- version: 1.0-SNAPSHOT

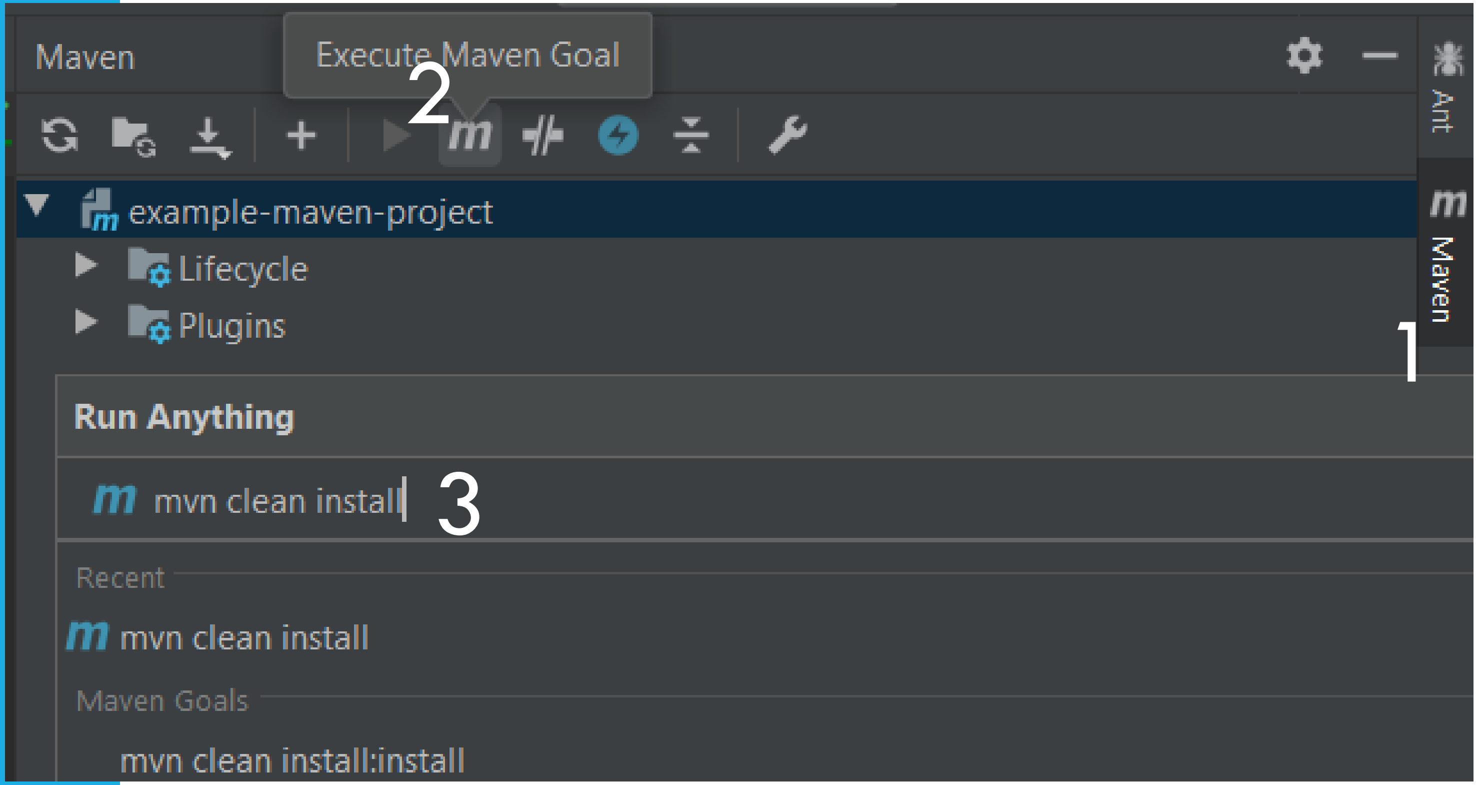
Create a new package called ‘edu.dmacc.coma502’ (if you do not have it already)

Create a Main.java with main that prints “Hello, World!”

EXERCISE: OUR FIRST MAVEN JAR FILE

Use IntelliJ to run ‘mvn clean install’

See the JAR generated in the /target folder



```
mvn clean install
```

Recent

```
mvn clean install
```

Maven Goals

```
mvn clean install:install
```

MAVEN SCOPES

Sometimes, we only need a dependency for a limited reason

- Testing
- Runtime-only
- Build-only

Scopes allow us to mark dependencies to only be used for a limited purpose

- Helps identify which jars must be shipped with our application and which are just helpers

MAVEN SCOPES

Example: Junit in Test scope

```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>RELEASE</version>
    <scope>test</scope>
</dependency>
```

UNIT TESTING

Unit testing is a way to insure the individual units or components of a piece of software function according to a set of expectations and assertions

The objective is to isolate a section of code and verify its correctness

A ‘unit’ in Java is generally a

- Public method or constructor
- Function
- Any other set of concise instructions that produce a behavior

UNIT TESTING

Unit tests

- Detect bugs early in the development lifecycle
- Act as documentation for application code
- Can be used to prove extensibility and reuse of objects

UNIT TESTING

But do they save time?

- Writing unit tests can add upfront time to development
- Tests allow you to reproduce issues and increases stability, so less time investigating when something goes wrong
- The more complex your system is, the more valuable it is to have tests proving the units of your system
- **Testing is an investment** and in the long run, you will save more time by having a unit test in place

UNIT TESTING

Best Practices

- Each unit should be as independent as possible with few side effects
- Try to test only one unit at a time
- There should be a new unit test for each logical branch in the unit covered
- Any time application code changes, the test code should pass or be amended as needed
- Do not commit to source control if you know that a test is failing

EXERCISE: IDENTIFYING A UNIT

Open your week 8 bakery assignment

Identify some potential units

Count the number of unit tests that should be written to cover all logical paths through each unit

JUNIT

JUnit is a library for writing unit tests in Java

Can be executed from an IDE

Supports

- Test methods with the ability to assert expectations
- Set-up methods before a test is run
- Teardown methods after a test is run
- Custom test runners to support more complex situations (such as Spring)

JUNIT

All test methods in Junit 4 begin with the ‘@Test’ annotation

Junit 4 Assertions

- Utility methods provided by JUnit through the Assert class

Common Assertions

- assertTrue / assertFalse
- assertNull / assertNotNull
- assertEquals
- assertArrayEquals
- fail

JUNIT

Example test method:

```
@Test
public void whenAssertingEquality() {
    String expected = "Nate";
    String actual = "John";

    assertEquals(expected, actual);
}
```

JUNIT

Example setup method:

```
Object classUnderTest;

@Before
public void setup() {
    classUnderTest = new Object();
}
```

EXAMPLE: IMPORT JUNIT

Using your example maven project from earlier:

- Search https://search.maven.org for “junit”
- Copy Apache Maven dependency for 4.13-rc-2
- Paste the junit dependency inside <dependencies></dependencies> tags
- Change it to test scope only

EXAMPLE: FIRST JUNIT TEST

Together, let’s create a method in Main that returns our name in lowercase

In the test folder

- Create the package edu.dmacc.coma502
- Create a test for the name method that asserts the return value is your name but with an uppercase (e.g. return “greg” but assert “Greg”)

EXERCISE: WRITING SOME UNIT TESTS

Create a method in Main that returns a number given a lowercase letter (given “a” returns 1, given “z” return 26)

- Hint: char can be cast to an int

Create a test in the test package that asserts some of the letters

EXERCISE: WRITING SOME UNIT TESTS

Update your method to return the same number for the corresponding uppercase letter (given “A” returns 1, given “Z” return 26)

Create a separate test that asserts some of the uppercase letters

EXERCISE: WRITING SOME UNIT TESTS

Update your method to throw an IllegalArgumentException if any other character is given than an uppercase or lowercase letter

Create a separate test that fails if the exception is not thrown

ASSIGNMENT

No Homework!

SEE YOU JANUARY 3RD!

Don’t forget to ask question early in the week