Maven

Maven Intro

What is Maven

Maven is a build automation tool. Think of it as a Makefile but instead of you writing the Makefile yourself, it writes that Makefile for you automatically to run and produce the final executable.

Maven also handle external third party dependency for you, meaning if you have another library that you would like to use and is available in the Maven repository, you can include it in the pom.xml file, and when you are writing your Maven project, you can just install those dependency and on your way to write your application with those dependency.

Maven + Vscode

Keep in mind that Maven doesn't actually do auto compiling for you, you would have to run `mvn compile` everytime when you make changes to your code. Or any other way of compiling depending on other plugins you use.

Maven compiles the `.class` files under the `target` directory!

For Spring boot and maven-compiler-plugin you would have to compile it yourself everytime you make code changes.

However, if you use Vscode along with the Java language support then it will compile those classes for you. For projects without build tool the `.class` files are under `~/.vscode-server`, for those projects with build tool then it is under `bin` folder by default in your project root. You can change that by editing your pom.xml file.

Parent and Child pom.xml

https://howtodoinjava.com/maven/maven-parent-child-pom-example/

package, install, release, deploy phases

- `mvn package` will construct your artifactory, i.e. a jar file and then place it in the current working directory
- `mvn install` will put your packaged maven project i.e. a jar file into the **local** repository, so that your other local application can use it as a project dependency. Just like dms-encryption-util, we have to install it into `~/.m2/repository` directory so that it can be found.
  - When your project does not find the dependency locally, it will fetch it from the remote repository and put it into `~/.m2/repository`
- `mvn release` will tag your current code in the specified SCM (version control system) and then change your version in your project. Basically it help you do automatic version tagging on GitHub
  - This is not the default life cycle phase, it is
- `mvn deploy` will put your packaged maven project into the remote central repository, maven or internal repository that you have specified for sharing with other developers

Lifecycle, Phases, goals, and Plugins?!

Maven Lifecycle

Maven is a build automation tool at heart. It has three built-in lifecycle that clearly define how the project will be build. There are three built-in build lifecycles:

1. default: handles the actual project compilation and deployment
2. clean: handles project cleaning, removing the target folder
3. site: handles the creation of project site documentation

You "can" add more lifecycle but then that go against the point of creating those three built-in build lifecycles. Those three built-in ones should be the standard of all the projects, you can create your own plugins and then hook it into the phases within the lifecycle.

Adding more lifecycle would take away the ease of reproducing the build and maintaining it.

Each of the lifecycle consists of phases. So you have life cycles like **default**, it consists of **phases**, and each **phases** consists of **goals**.

Phases in default

Not going to list them all but here are some of the phases in order:

- validate
- compile
- test
- package
- verify
- install
- deploy

When you execute a phase all the goals tied to the goal will be executed. In addition, when you execute a phase all the previous steps will be executed. For example, executing `mvn install` will execute, validate, compile, test, package, verify, and install in order.

> There is no way to execute the "lifecycle", to execute the lifecycle you would just specify the last phase to run through the entire phases. In this case running `mvn deploy` will in a sense run the entire lifecycle

Maven phases

Let's clear something up, all the available goals are provided by plugins. The compile phase's goals are provided by the "maven-compile**r**-plugin", the package phase's goal are provided by the "maven-jar-plugin", you get the idea.

> Official maven plugin like compile, package have standard name of "maven-<name>-plugin", whereas non-official maven plugin have "<name>-maven-plugin" naming convention.
>
> The implication is that when you want to invoke the goal directly without invoking the phase if you are using official maven plugin is just "<name>:<goal>" for example, `mvn compiler:compile`. You don't need to do `mvn maven-compiler-plugin:compile`. However, unofficial maven plugins then you would need to provide the entire plugin name to invoke the goal.

A build phase doesn't necessarily need to have any goal bound to it. If it has no goal bound to it, it will simply not execute any goal.

However, the same goal can be bounded to multiple different build phases, so that goal will just be executed multiple times when the phase is executed.

Furthermore, not all the goals from a plugin are bind a phase. For example, the `maven-compiler-plguin` have two goals, `compile / testCompile`. However, only the `compile` goal is bound to the compile phase. You can add `testCompile` to the phase by adding an `<executions>` section in your plugin tag, will go more over that in the next section.

Maven goals (Mojos)

Goals are the actual task that are executed, they help building and manage the project. Phases and lifecycle are essentially just abstraction of goals.

Goals may or may not not bound to a phase, those that are tied to a phase will be executed when you run the phase. Those are not bound to a phase, you can invoke them directly. By using the syntax discussed above.

To bound a plugin's goal to a phase if they are not bound by default, or you would like to bound to a different phase, i.e. adding jar plugin's jar goal to the compile phase. Whenever you run `mvn copmile` it will also package it as a jar, as oppose to whenever you run `mvn package` if you don't want to run test phase before it. What you would do is to add a `<executions>` section in your pom.xml file.

Configuring plugin execution

If you would like to add a goal that isn't by default mapped to one of the default phases for your plugin, or you would like to add it to different phase then you would need to append a `<executions>` section for your plugin.

For example, the `jar:jar` goal which build a JAR from the current project is by default bind to the `package` but if you want to run the goal whenever `compile` plugin is run then you can add the following section to your `pom.xml`

```
        <executions>
          <execution>
            <id>compile-jar</id>
            <phase>compile</phase>
            <goals>
              <goal>jar</goal>
            </goals>
          </execution>
        </executions>
```

What this section does is that it will bind the goal `jar:jar` remember goals can be bind to multiple phases, to the `compile` phase, with the id `compile-jar`. The id is there so that when you invoke the `mvn` command, the execution output will be in the form of

```
<plugin-name>:<plugin-version>:<phase> (<execution-id>)
```

So in this case, as part of the `compile` phase, it will be running `maven-jar-plugin:2.2.2:jar (compile-jar)`, so right after you compile your code, you will pack it into a jar.

Scenarios for executions

```
       <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-docck-plugin</artifactId>
            <version>1.0</version>
            <executions>
                <execution>
                    <id>some-other-other-id</id> <!-- No goal for execution is defined -->
                    <phase>pre-site</phase>
                </execution>
                <execution>
                    <phase>pre-site</phase>   <!-- No id for execution is defined -->
                    <goals>
                        <goal>check</goal>
```

```
                    </goals>
                </execution>
                <execution>
                    <id>some-id</id>          <!-- No phase for execution is defined --
>
                    <goals>
                        <goal>check</goal>
                    </goals>
                </execution>
                <execution>
                    <id>some-other-id</id>     <!-- Both id and phase defined -->
                    <phase>pre-site</phase>
                    <goals>
                        <goal>check</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
```

Results in sample output as following

```
[INFO] --- maven-docck-plugin:1.0:check (default) @ MavenJavaApplication ---
[INFO] Skipping unsupported project: MavenJavaApplication
[INFO] No documentation errors were found.
[INFO]
[INFO] --- maven-docck-plugin:1.0:check (some-other-id) @ MavenJavaApplication ---
[INFO] Skipping unsupported project: MavenJavaApplication
[INFO] No documentation errors were found.
```

Execution 1: No goal for execution is defined

If no goal is specified then that execution is simply ignored. If a phase have no goal then it will just be ignored, because there is nothing to execute.

Execution 2: No id for execution is defined

If no id is defined for a specified execution, but goal and phase is specified then it will simply execute with the id of `default`.

Execution 3: No phase for execution is defined

If no phase is defined, then that execution of plugin goal is simply not run.

Execution 4: id, phase, and goal is defined

Then it will get ran during the specified phase with the specified id

`default-cli` vs `default-<goalName>`

The goals that are invoked directly from the command line for example `mvn jar:jar` (invoking the jar goal from the `maven-jar-plugin`) it will have the execution id of `default-cli` because you ran it in the CLI.

This ID is provided so that you can provide further configuration if needed for the plugin that's ran from CLI, by using the execution id of `default-cli` in your `pom.xml` file.

Plugin goals that are mapped to the default lifecycle like some of the official maven plugin such as `maven-jar-plugin` will have their execution id to be `default-<goalName>` for example:

```
        <artifactId>maven-clean-plugin</artifactId>
        <version>2.5</version>
        <executions>
          <execution>
            <id>default-clean</id>
            <phase>clean</phase>
            <goals>
              <goal>clean</goal>
            </goals>
          </execution>
        </executions>
```

The `clean` plugin's default goal mapping maps the `clean` goal to the `clean` phase with the execution id of `default-clean`.

Again this is there so that you can further configure these official built-in goals, it is a naming convention after all.

You can find out about this by looking at the effective pom via `mvn help:effective-pom.`

Compile time and runtime dependency

Problem

There was bunch of dependency that older version of spring-boot includes as compile time dependency, meaning in your project without you explicitly including those as a dependency in your `pom.xml` you can use them in your code.

However, if you upgrade spring-boot version to more recent versions those dependency became RUNTIME scope, meaning those dependencies will not be included as part of the compile classpath. You must explicitly declare it as a dependency in your project in order to use it.

Why I bring this up

This is from my experience where I upgraded spring-boot to a higher version, in the older version it used a class from a transitive dependency and that compiled fine because that transitive dependency was included as a compile time dependency.

However, after upgrading it, the same transitive dependency became runtime only, thus it isn't available anymore, which lead to compilation error if you use the library without explicitly declaring it as a dependency in your pom.xml

Jacoco Plugin

What is Code Coverage

Code coverage is a quality measurement during development process to see how much of your code has been tested (or executed). Your unit test should cover most of business logic of your code, however, reaching 100% of code coverage does not mean that your code is bug-free. You can reach 100% code coverage while still having bugs.

What is JaCoCo

Jacoco maven plugin is a free code coverage library for Java projects. It is used to generate code coverage reports.

How it works is that JaCoCo plugin attaches a runtime agent to the JVM when it starts. The JaCoCo agent will then instrument the class so that it can see when the class is called what lines of codes are called during the testing process. It then build up the code coverage statistics during the testing phase.

How to setup JaCoCo

To set up Jacoco-maven-plugin you would need to include this following plugin in your pom.xml:

```
<plugin>
  <groupId>org.jacoco</groupId>
  <artifactId>jacoco-maven-plugin</artifactId>
  <version>0.8.6</version>
</plugin>
```

The latest version you would need to determine based on what is released. Then you would also want to add the following executions tag in order to trigger Jacoco maven plugin whenever you run the unit test of your project.

```
<plugin>
  <groupId>org.jacoco</groupId>
  <artifactId>jacoco-maven-plugin</artifactId>
  <version>0.8.6</version>
  <executions>
   <execution>
```

```
      <id>prepare-agent</id>
           <goals>
            <goal>prepare-agent</goal>
           </goals>
      </execution>
      <execution>
           <id>report</id>
           <phase>test</phase>
              <goals>
              <goal>report</goal>
              </goals>
        </execution>
    </executions>
</plugin>
```

- **Prepare-agent goal:** This goal prepares the Jacoco runtime agent to record the execution data, record the number of lines that's executed, backtraced, etc. This must be attached otherwise, Jacoco will not have data to generate the code coverage
- **Report goal**: This goal will finally create the code coverage reports from the execution data recorded by the runtime agent. Since this is attached to the test phase, whenever the test phase finishes it will generate the report automatically without having you to invoke the goal manually.

Add code coverage check

You can also enforce minimum code coverage check when you are executing the test

Toggle me for full code

```
<plugin>
  <groupId>org.jacoco</groupId>
  <artifactId>jacoco-maven-plugin</artifactId>
  <version>0.8.10</version>
  <executions>
    <execution>
      <id>prepare-agent</id>
      <goals>
        <goal>prepare-agent</goal>
      </goals>
    </execution>
    <execution>
```

```
      <id>jacoco-report uwu</id>
      <phase>test</phase>
      <goals>
        <goal>report</goal>
      </goals>
    </execution>
    <execution>
      <phase>test</phase>
      <id>coverage-check</id>
      <goals>
        <goal>check</goal>
      </goals>
      <configuration>
        <rules>
          <rule>
            <element>PACKAGE</element>
            <limits>
              <limit>
                <counter>LINE</counter>
                <value>COVEREDRATIO</value>
                <minimum>0.8</minimum>
              </limit>
            </limits>
          </rule>
        </rules>
      </configuration>
    </execution>
  </executions>
</plugin>
```

The goal will fail if your code coverage doesn't meet the specified percentage.

Ignore class files for coverage

You can add to the configuration the list of class files that you should ignore because it is difficult / unnecessary to test them for coverage.

```
<plugin>
  <groupId>org.jacoco</groupId>
  <artifactId>jacoco-maven-plugin</artifactId>
```

```
    <version>0.8.10</version>
    <configuration>
      <excludes>
        <exclude>**/App.class</exclude>
      </excludes>
    </configuration>
  </plugin>
```

Lombok

Project Lombok

Lombok is a Java library that can help you generate lots of the boiler plate code so that you don't have to end up writing them yourself. For example, getter, setter, toString, and equals method. You just have to slap an annotation onto the class and boom you will have the boiler plate stuff generated all for you.

Jacoco code coverage

If you use Lombok annotation then you will be complained about a lot about code coverage because the method isn't being called in the unit test.

You can ignore Lombok generated method by adding a `lombok.config` file with the following configuration

```
# Adds the Lombok annotation to all of the code generated by
# Lombok so that it will be automatically excluded from coverage by jacoco.
lombok.addLombokGeneratedAnnotation = true
```

Maven Tests

Maven Test

When you execute `mvn test` it will compile and test the source code that's underneath the source code folder!

Maven Test allows integration with JUnit, TestNG, and POJO. We will go over JUnit because that's what we use most of the time.

JUnit

The popular testing framework for Java. Most of the test are written in JUnit 4 but the latest version is JUnit 5.

Writing them are still the same, however, integrating it with behavioral driven testing framework like Cucumber will be different I will go over the differences for using cucumber with JUnit 4 and JUnit 5 (Mainly in the annotation that is used to specify the glue and feature files)

When you write your unit tests using JUnit you want to structure your file directory like the following:

```
src
├── main
│   ├── java
│   │   └── test
│   │       └── App.java
│   └── resources
│       └── application.properties
└── test
    ├── java
    │   └── test
    │       └── AppTest.java
    └── resources
        └── test.json
```

Your main source code directory will contain two different directories.

- The `main` directory contain the Java source code for your application as well as any resources that your application uses

- The `test` directory contain the Java source code used for testing. This is where you will be writing your unit test for your applications

When you run `mvn test` it will compile the source code from the `src` directory and execute the proper unit test using the JUnit framework.

Dependency needed for JUnit 5

```
<dependencies>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.9.1</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Test Class

A test class is a class that will be used by Maven to carry out the testing.

A test class contains methods that will be tested by Maven. Maven use a pattern to check whether a class is a test class or name, the list of pattern used by Maven to determine if a class is a test class is of the following:

```
**/Test*.java
**/*Test.java
**/*Tests.java
**/*TestCase.java
```

Test Runner Class

> This applies to JUnit 4 test. JUnit 5 has a new API and @RunWith is no longer needed.

A test runner in JUnit is just a simple class that implements the `Describable` interface. The interface just have two abstract methods to implement:

```
public abstract class Runner implements Describable {
  public abstract Description getDescription();
  public abstract void run(RunNotifier notifier);
}
```

1. getDescription: Is used to return a description containing the information about the test, this is used by different tools such as those from IDE
2. run: This is used to actually run the test class or test suite

You can implement your own test runner to do custom logic for running tests, for example you can print out silly messages just before each test is run.

A simple test runner class implementation can be found below: All it does is before it runs the test class is it prints the message "MyRunner <TestClass>"

Eample Test Runner

```
//src/main/java/MinimalRunner
package com.codingninjas.testrunner;



import org.junit.runner.Description;
import org.junit.runner.Runner;
import org.junit.runner.notification.RunNotifier;
import org.junit.Test;
import java.lang.reflect.Method;




public class MinimalRunner extends Runner {



    private Class testClass;
    public MinimalRunner(Class testClass) {
        super();
        this.testClass = testClass;
    }

    //getDescription method inherited from Describable
    @Override
    public Description getDescription() {
        return Description
          .createTestDescription(testClass, "My runner description");
    }
```

```
    //our run implementation
    @Override
    public void run(RunNotifier notifier) {
        System.out.println("running the tests from MyRunner: " + testClass);
        try {
            Object testObject = testClass.newInstance();
            for (Method method : testClass.getMethods()) {
                if (method.isAnnotationPresent(Test.class)) {
                    notifier.fireTestStarted(Description
                      .createTestDescription(testClass, method.getName()));
                    method.invoke(testObject);
                    notifier.fireTestFinished(Description
                      .createTestDescription(testClass, method.getName()));
                }
            }
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

To actually use this example Test Runner you would then just simply annotate your test class with `@RunWith(MinimalRunner.class)` which will delegate the responsibility of running this particular test class to that custom test runner.

```
//src/test/java/CalculatorTest.java
package com.codingninjas.testrunner;


import static org.junit.Assert.assertEquals;
import org.junit.Test;
import org.junit.runner.RunWith;


@RunWith(MinimalRunner.class)
public class CalculatorTest {
    Calculator calculator = new Calculator();
```

```
    @Test
    public void testAddition() {
        System.out.println("in testAddition");
        assertEquals("addition", 8, calculator.add(5, 3));
    }
}
```

> Note that this runner mechanism is no longer present in JUnit 5 and is replaced by Jupiter extension. https://stackoverflow.com/a/59360733/7238625

Default Test Runner class

For Junit 4 at least, if you don't specify a test runner, JUnit will default to the default `BlockJUnit4ClassRunner` which as you know it will just simply executes the test class.

Specify one test class to execute

If you only want to run one of the test class then you can run maven test with the following option

```
 mvn test -Dtest="TestFoo"
```

This will only execute the `TestFoo` class.

Specify one method to execute from a test class

If you only want to execute one of the test then you can run maven test with the following option

```
mvn test -Dtest="TestFoo#testFoo"
```

The left of # denotes the test class, and the right of # denotes the method that you want to execute.

Cucumber + JUnit 4

> Note that the following code only works with JUnit 4

Cucumber is a test automation tool that follows BDD. You describe the things you want to test in something called feature files which uses Gherkin language specification. This is so that stakeholders who doesn't understand code can also look at the feature file and understand the behavior of your application without needing to learn the programming language.

Underneath cucumber will then translate the feature files into actual code.

Cucumber framework in Java works by implementing it's own test runner class with JUnit, this is how it makes it work, and why the test class using cucumber test doesn't have anything in them. In the CucumberOptions annotation you will be specifying the feature files you will be looking at, the glue it is using, the specific tags to run, and plugin options.

The list of sample dependency to use Cucumber with JUnit will be the following:

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.13.2</version>
  <scope>test</scope>
</dependency>

<dependency>
  <groupId>io.cucumber</groupId>
  <artifactId>cucumber-java</artifactId>
  <version>7.14.0</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>io.cucumber</groupId>
  <artifactId>cucumber-junit</artifactId>
  <version>7.14.0</version>
  <scope>test</scope>
</dependency>
```

Terminology

1. Glue: A glue is used to associate the Gherkin steps (i.e. what's written in feature files) to a step definition method (i.e. actual code that gets executed). The Gherkin step is "glued" to the step definition. If this is used in the context of the test class, then it is referring to the package that your step definition class lives under.
2. Features: A feature file contains the description of the test in Gherkin. This is what cucumbers look at to execute the actual test. It is meant to be easy to read for stakeholders in plain English so even if they don't understand programming language they can interpret the test cases as well.
3. Plugin: Reporting plugins that can be used to output the result of the cucumber test into different formats. Json and html output, you can also use third party plugins as well!

Writing cucumber test

Your file structure should look like this:

```
src
└── test
    ├── java
    │   └── com
    │       ├── lol
    │       │   └── StepDefinition.java
    │       ├── OneMoreStep.java
    │       └── TestingFoo.java
    └── resources
        └── features
            └── foo.feature
```

Your feature files will be living under the `resources` folder. Your step definition will be living under the Java folder. Honestly, whether or not it is in another package just depend on you, you will be specifying where the step definitions are as part of glue configuration anyway. If you don't specify glue it will look for it under the test class.

Your test class will need to be annotated with @RunWith to tell JUnit that it will be the responsibility of the Cucumber test runner to run the test.

```
package com;


import org.junit.runner.RunWith;


import io.cucumber.junit.Cucumber;
import io.cucumber.junit.CucumberOptions;

@RunWith(Cucumber.class)
@CucumberOptions(
   features = "src/test/resources/features",
   glue = {"com"},
   plugin = {"html:target/out.html"}
)
public class TestingFoo {
}
```

Cucumber + JUnit 5

Now because JUnit 5 has changed a lot of stuff, what worked for JUnit 4 will no longer work with JUnit 5 if you want to integrate cucumber with it. Here are the steps on how to do it.

Include the following dependency:

```
<dependencies>
  <dependency>
    <groupId>org.junit.platform</groupId>
    <artifactId>junit-platform-suite-engine</artifactId>
    <version>1.10.0</version>
  </dependency>
  <dependency>
    <groupId>io.cucumber</groupId>
    <artifactId>cucumber-java</artifactId>
    <version>7.14.0</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>io.cucumber</groupId>
    <artifactId>cucumber-junit-platform-engine</artifactId>
    <version>7.14.0</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

The test class will then have to have these following configurations

```
package com;

import org.junit.platform.suite.api.ConfigurationParameter;
import org.junit.platform.suite.api.IncludeEngines;
import org.junit.platform.suite.api.SelectClasspathResource;
import org.junit.platform.suite.api.Suite;



import static io.cucumber.junit.platform.engine.Constants.GLUE_PROPERTY_NAME;

@Suite
@IncludeEngines("cucumber")
@SelectClasspathResource("features")
@ConfigurationParameter(
   key = GLUE_PROPERTY_NAME,
   value = "com"
```

```
)
public class TestingFoo {


}
```

As you can see although the annotation differs a bit, it is still relatively the same.

Multiple Runners

If for some reason your project is configured to have multiple test classes, remember JUnit will scan all of the test classes underneath the test directory, and you only want to run say one of the test classes. Then you can use the option flag that was reference above for the JUnit section :smile: