

Apache Maven and Unit Testing with JUnit

Maven overview

- Apache Maven is a tool for building and managing Java projects
- Maven provides a uniform build system using a project object model (POM) and a set of plug-ins
- Maven has several built-in lifecycles (see next page) which can be modified with plug-ins
- Lifecycles are sequential, e.g., *mvn deploy* will trigger the entire lifecycle starting with *validate*, then *compile*, and so on, ending with *install* and *deploy*

Maven lifecycles

- clean – deletes *target* directory containing previous build artifacts including compiled .class and packaged .jar files
- default
 - validate – validates POM file
 - compile – compiles source code (but not tests)
 - test – compiles tests
 - package – packages compiled Java bytecode (e.g., into .jar file)
 - verify – runs package tests (such as for required code coverage)
 - install – installs package to a local maven repository
 - deploy – uploads package to remote repository
- site (generates documentation to *target/sites*) and site-deploy (deploys to remote site)

Maven in our CSC-360 class

- We generally will not worry about running these lifecycles
 - We will run code using the Play button as we have done previously, which will recognize dependencies in the POM file
- We will use Maven to
 - specify Junit and other dependencies
 - Bind Javadoc generation to the *site* phase
- The POM file is the core configuration file written in XML, which defines the project's structure, identity, dependencies, and build behavior
- XML (extensible markup language) uses tags to store information

Project Identity and properties

- `<project>` - root wrapper for configuration
- `<modelVersion>` - XML structure version (always set to 4.0.0)
- `<groupId>` unique package name (reverse domain name notation)
- `<artifactId>` - project name (used for jar file)
- `<version>` - current version (use SNAPSHOT to indicate in progress)
- `<description>` - project description
- `<properties>` -
 - stores constant values such as `<maven.compiler.source>` and `<maven.compiler.target>`
 - Can be referenced using `${property.name}`

POM file dependencies and plugins

- Dependencies
 - `<dependencies>` - wrapper for all dependencies
 - `<dependency>` - a single external library dependency, includes `<groupId>`, `<artifactId>`, and `<version>`
- Plugins
 - `<plugins>` - wrapper for all plugins
 - `<plugin>` - an individual plugin with a `<groupId>`, `<artifactId>`, and `<version>`
 - Plugins are put in `<build>` to bind them to build lifecycle stages (e.g., *compile*) or `<reporting>` to bind them to reporting (i.e., *site*).

JUnit testing

- JUnit is a framework for writing and running automated tests in Java
- Unit testing allows for sustainable software growth by verifying small, isolated components of code through automated, repeatable tests
- In Java, the standard is to create a Test class for every public class. However, a single Test class can be used for a public class and its helper classes.

The AAA Pattern

- The AAA pattern is a standard testing structure with three steps: Arrange (set up data), Act (call the method), Assert(verify results)

```
@Test
void testSum() {

    // 1. Arrange
    Calculator calculator = new Calculator();
    int firstNumber = 3;
    int secondNumber = 4;

    // 2. Act
    int actualResult = calculator.add(firstNumber, secondNumber);

    // 3. Assert
    int expectedResult = 7;
    assertEquals(expectedResult, actualResult, message: "sum should be 7");
}
```

JUnit annotations

- `@Test`: Marks a method as a test case
- `@BeforeEach`, `@AfterEach`: run before or after every single test, useful for resetting objects or for cleanup tasks
- `@BeforeAll`, `@AfterAll`: runs once before or after all tests

JUnit assertions

- `assertEquals(expected, actual)`: passes if values are equal
- `assertTrue(condition)`: passes if condition is true
- `assertArrayEquals(expected, actual)`: passes if arrays contain the same elements in the same order
- `assertThrows(Exception.class, () -> executable)`: passes if executable code throws the specified exception type
- All assertions can include an optional message as a subsequent argument
- See [documentation](#) for additional information

Unit testing best practices

- Each test case should test one logical concept
- Keep testing code simple (e.g., avoid loops and branches)
- Tests should be isolated (the failure of one test should never impact another)
- Test edge cases, bad inputs, and exceptions; do not just test the “happy path”
- Unit tests are designed to protect against future changes
- Never make the same mistake twice – write a test for it the first time

Example #1

- Comprehensive testing of `Calculator.add()`
 - Cover each category of input that could behave differently
 - Both positive, both negative, one of each, one value is 0, answer is 0
 - Edge cases:
 - What happens when you get close to `Integer.MAX_VALUE` or `Integer.MIN_VALUE`?
 - Exceeding these boundaries should throw an error

Example #2

- Testing of a *firstChar* (*String s*) method to get the first character of a string.
 - Empty string will throw `StringIndexOutOfBoundsException` exception (is this what you want?)
 - Null string will throw a `NullPointerException` (is this what you want?)