PROMINEO TECH

Week 12 Coding Assignment

Instructions :

1. Follow the **Coding Steps** below to complete this assignment.

    - In Eclipse, or an IDE of your choice, write the code that accomplishes the objectives listed below. Ensure that the code compiles and runs as directed.
    - Create a new repository on GitHub for this week's assignment and push your completed code to this dedicated repo.
    - Create a video showcasing your work:
        - In this video: record and present your project verbally while showing the results of the working project.
        - Easy way to Create a video: Start a meeting in Zoom, share your screen, open Eclipse with the code and your Console window, start recording & record yourself describing and running the program showing the results.
        - Your video should be a maximum of 5-minutes.
        - Upload your video with a public link.
        - Easy way to Create a Public Video Link: Upload your video recording to YouTube with a public link.

2. In addition, please include the following in your Project Submission Quiz

    - The URL for this week's GitHub repository.
    - The URL of the public link of your video.

PROMINEO TECH

Week 12 Coding Assignment

Overview

In this assignment, you will create a **JUnit Test Case**, step by step!

1. Example Unit Test - Code This

Follow Steps 1-1 to 1-4. This Assignment builds a new **Maven Project** (*TestDemo.java*) with a **JUnit Test Case** (*TestDemoJUnitTest.java*).

This part will give you the opportunity to write a **@ParameterizedTest**.

2. Write a New JUnit Test

This section will give you the opportunity to write a **@Test** for **addPositive().** Do not use parameters.

3. Create Your Own Method & JUnit Test

Try this on your own by doing the following:

(a) Creating a new method in *TestDemo.java*

(b) Write a test for your new method in *TestDemoJunitTest.java*

4. Mocking a Class - TestDemo -- *more advanced!*

Follow Steps 4-1 & 4-2. Using the same **Maven Project** (*TestDemo.java)* and **JUnit Test Case** (*TestDemoJUnitTest.java)*, create another method and test, this time using Mockito to mock the *TestDemo* inside of *TestDemoJUnitTest.java*.

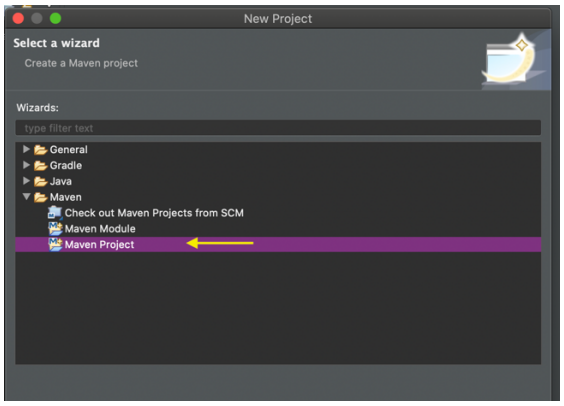
Coding Steps:

1. Example Unit Test - Code This

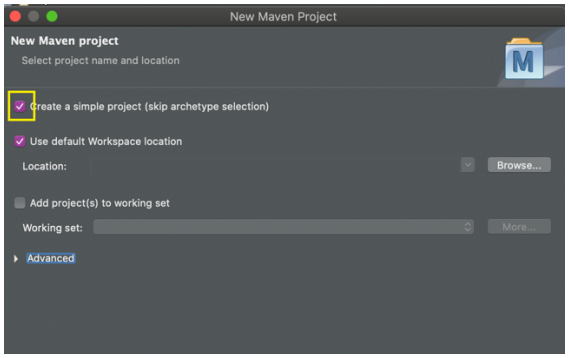
Step 1-1:

Create a new Maven project.

a. In Eclipse, right-click in Project Explorer, select "New / Project". Expand "Maven". Select "Maven Project". Click "Next".

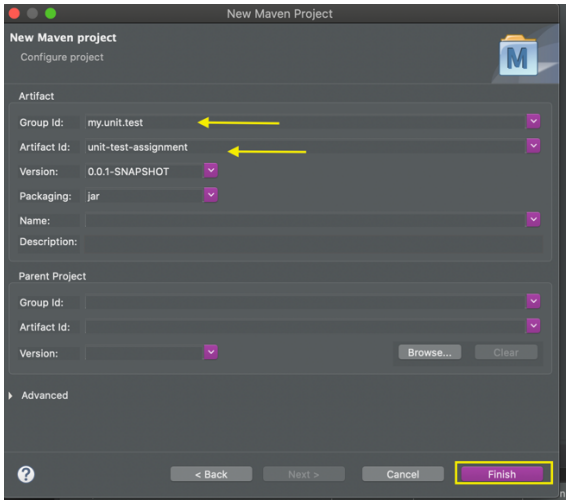


b. Check "Create a simple project (skip archetype selection)". Click "Next".

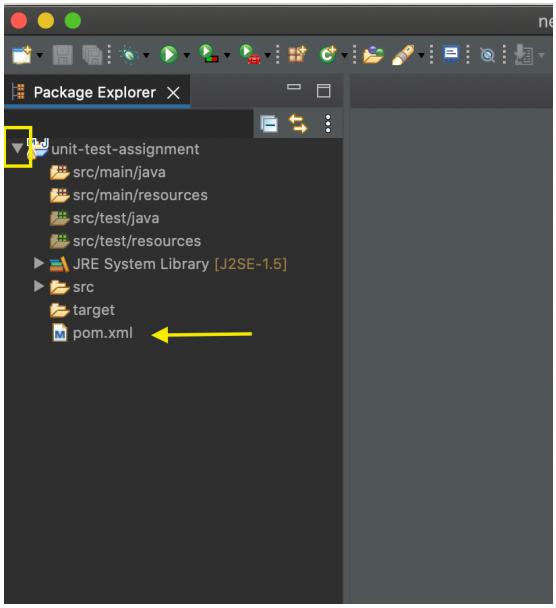


c. Enter the Group Id: "my.unit.test". Enter the Artifact Id: "unit-test-assignment". Click "Finish".

Week 12 Coding Assignment

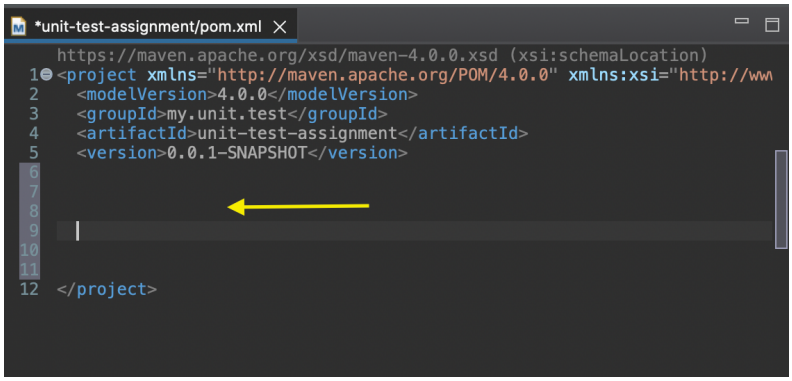


d. The project "unit-test-assignment" should appear in the Package Explorer. Click the right arrow next to "unit-test-assignment" to expand it. Double-click on "pom.xml" to open it in the editor.



e. Put a couple of blank lines between <version>0.0.1-SNAPSHOT</version> and </project>.

Week 12 Coding Assignment



f. Copy and paste the following code (not the dotted lines) into the blank area you just created.

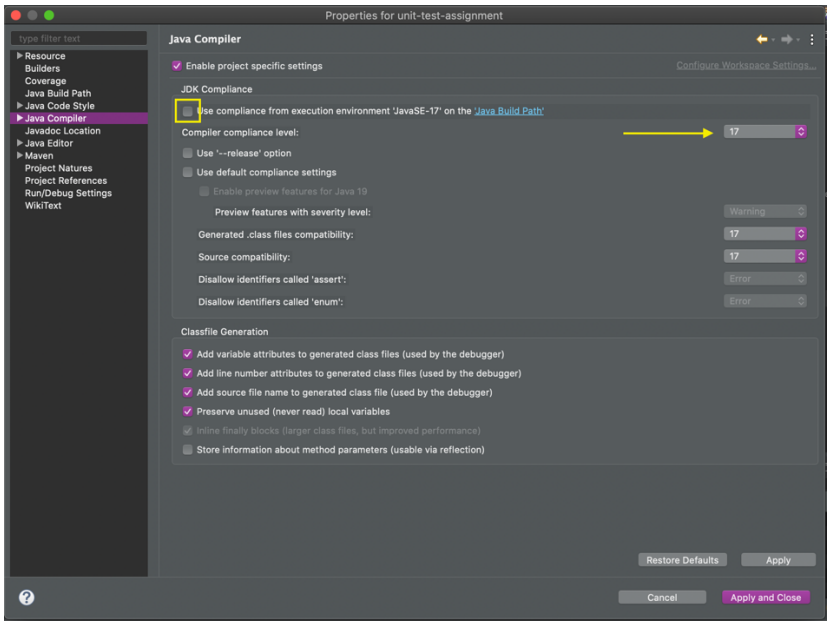
```
-----------------------------------------------
  <properties>
    <java.version>17</java.version>
    <project.build.sourceEncoding>utf-8</project.build.sourceEncoding>
  </properties>

  <dependencies>
    <dependency>
      <groupId>com.google.guava</groupId>
      <artifactId>guava</artifactId>
      <version>30.1.1-jre</version>
    </dependency>

    <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter</artifactId>
      <version>5.7.2</version>
      <scope>test</scope>
    </dependency>
```

```
    <dependency>
      <groupId>org.assertj</groupId>
      <artifactId>assertj-core</artifactId>
      <version>3.20.2</version>
      <scope>test</scope>
    </dependency>

   <dependency>
      <groupId>org.mockito</groupId>
      <artifactId>mockito-junit-jupiter</artifactId>
      <version>3.11.2</version>
      <scope>test</scope>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.8.1</version>
        <configuration>
          <source>${java.version}</source>
          <target>${java.version}</target>
        </configuration>
      </plugin>
    </plugins>
  </build>
-------------------------------------------------
```


g. Save the modified ***pom.xml*** file.

Suggested**: Execute a **Maven Update Project

h. **For Eclipse only**: right-click on "unit-test-assignment" in the Project Explorer. Click on "Properties". Click "Java Compiler". Make sure "Enable project specific settings" is checked. Uncheck "Use compliance from execution environment 'J2SE-1.5' on the 'Java Build Path'. Set "Compiler compliance level" to 17. Click "Apply and Close".

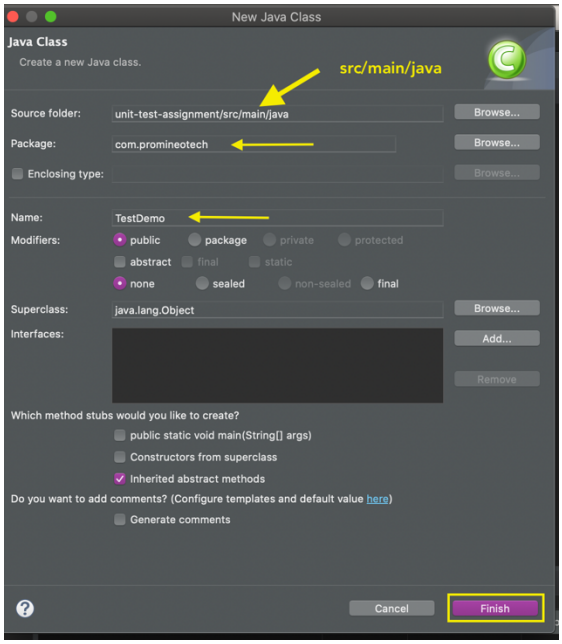


i. If asked to rebuild the project, click "**Yes**".

Step 1-2:

Create a class named "TestDemo" under src/main/java in the default package. (In Package Explorer, expand "unit-test-assignment". Right-click on "src/main/java" and select "New / Class". Enter "TestDemo" in the "Name" field and click "Finish".)

Week 12 Coding Assignment



a. Create an instance method (not static) named addPositive. It should take two int parameters and return an int.

public int addPositive(int a, int b) {}

b. If both parameters are positive (greater than zero) return the sum of the parameters. If either parameter is zero or negative, throw an IllegalArgumentException with the message "Both parameters must be positive!". IllegalArgumentException is in the java.lang package so you won't need an import statement.

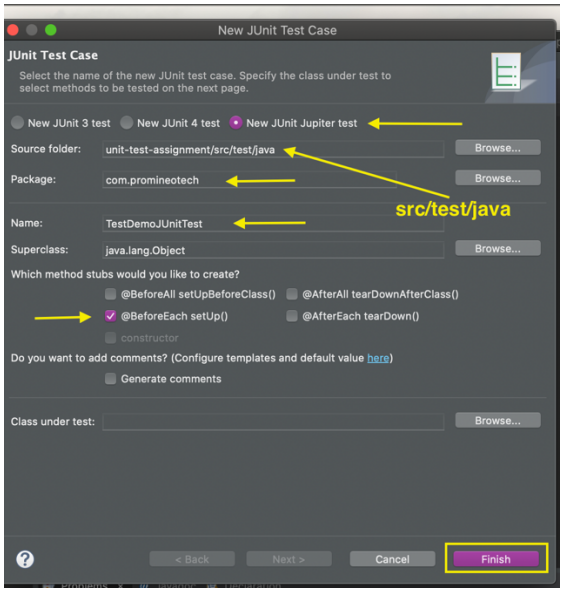
c. Save the file.

Step 1-3:

Create a JUnit Test Case named "TestDemoJUnitTest" under src/**test**/java in **com.promineotech** package.

In Package Explorer, find "src/test/java" and right-click on it. Select "New / JUnit Test Case". In the "Name" field, enter "TestDemoJUnitTest". Make sure that

"New JUnit Jupiter test" is selected. Make sure that "@BeforeEach setUp()" is checked. Click "Finish".



Step 1-4:

In *TestDemoJUnitTest.java*, add a **private instance variable** of type TestDemo named testDemo. Remember that **instance** variables are *non-static variables which are defined **in a class**, but **outside** of any method, constructor or a block.*

a. In the setUp method, create the TestDemo object. This will ensure that a new TestDemo object is created before each test.

b. Change **@Test** to **@ParameterizedTest**. Add the import statement for ***org.junit.jupiter.params.ParameterizedTest***

c. Change the name of method "test" to "assertThatTwoPositiveNumbersAreAddedCorrectly"

d. Add four parameters to assertThatTwoPositiveNumbersAreAddedCorrectly as shown:

Type	Name
int	a
int	b
int	expected
boolean	expectException

e. **Write the test.**

- Remove the "fail" line.
- Test the value of expectException. If it is false, assert that when TestDemo.addPositive is called with values a and b, that the result is the same as the parameter expected.
- The assertion should look like this:

```
If (!expectException) {
    assertThat(testDemo.addPositive(a, b)).isEqualTo(expected);
}
```

f. Add the test for the thrown exception in an else clause. Use assertThatThrownBy for this. Add the static import:

```
org.assertj.core.api.Assertions.assertThatThrownBy;
```

g. As a parameter to assertThatThrownBy, add a Lambda expression with no parameters. The Lambda body should be the method call to testDemo.addPositive.

h. Use the assertion isInstanceOf(IllegalArgumentException.class) to ensure that the correct exception is thrown.

i. If this is too confusing, you can "cheat" and copy this:

```
assertThatThrownBy(() ->
    testDemo.addPositive(a, b))
```


.isInstanceOf(IllegalArgumentException.class);

j. Add the parameter source method.

- Create a static method named argumentsForAddPositive. It should not have any parameters and it should return a Stream of Arguments. The imports are:

  java.util.stream.Stream
  org.junit.jupiter.params.provider.Arguments

- The method should return a Stream as in Stream.of();
- Each parameter set should be wrapped in an arguments() method call. Add the static import for arguments:

  org.junit.jupiter.params.provider.Arguments.arguments

- So, if you are adding 2 and 4 to get the value of 6 and are not expecting an exception, you need to do:

  arguments(2, 4, 6, false)

- Add as many arguments lines as needed to test the addPositive method thoroughly. Make sure to add some zero or negative arguments.

k. Just below the @ParameterizedTest annotation, add the annotation @MethodSource. Pass a single parameter to @MethodSource. It must be the fully-qualified (includes package) class name of the test followed by a # sign followed by the name of the method that supplies the parameters. Since the test is in the default package, there is no package in the fully-qualified class name. So,

@MethodSource("TestDemoJUnitTest#argumentsForAddPositive")

———————————————————————————————————

---

2. Write a New JUnit Test for addPositive

Review

At this point in the assignment, we have done the following:

- *Created a method in **TestDemo** named **addPositive** that takes two int parameters.*
- *Created a Junit Test in **TestDemoJUnitTest** named **assertThatTwoPositiveNumbersAreAddedCorrectly** that uses **@ParameterizedTest***

***NOW**: Let's write a test that uses **@Test** for the same method!*

Step 2-1

In TestDemoJUnitTest.java, write a test for addPositive.

- Create a method annotated with **@Test** named **assertThatPairsOfPositiveNumbersAreAddedCorrectly**. The method must have package visibility (not public!) or JUnit won't find it. The annotation **@Test** is in the **org.junit.jupiter.api** package.
- Use assertThat to test that the value returned from addPositive is equal to the correct value.

```
assertThat(testDemo.addPositive(4,5)).isEqualTo(9);

assertThat(testDemo.addPositive(40,50)).isEqualTo(90);
```

- Add as many calls like the ones above as you wish.

---

3. Create Your Own Method & JUnit Test:

NOW it's time to write a new method and a test (parameterized or not) that tests your method. This can be a very simple method. Try to replicate the process.

- Create your own method in **TestDemo**
- Create your own Junit Test in **TestDemoJUnitTest** that uses either **@ParameterizedTest** or **@Test**
- Document the code in both places to explain the method, and to explain the test.

---

4. Mocking a Class – TestDemo

Step 4-1:

In TestDemo.java, add another method named randomNumberSquared. This method obtains a random int between 1 and 10 and then returns the square of the number.

- randomNumberSquared should return an int and not take any parameters.
- It should call another method in the same class named getRandomInt. This method takes no parameters and must be package visibility so that the test can see it. getRandomInt should look like this:

```
int getRandomInt() {
        Random random = new Random();
        return random.nextInt(10) + 1;
}
```

The Random class is in the java.util package.

- randomNumberSquared should return the value obtained from getRandomInt multiplied by itself.

Step 4-2:

Write a test for randomNumberSquared in TestDemoJUnitTest.java. Since you don't know what getRandomInt will return (that's the point of random, after all), you will need to mock it out and supply a known value.

a. Create a method annotated with @Test named assertThatNumberSquaredIsCorrect. The method must have package visibility (not public!) or JUnit won't find it. The annotation @Test is in the org.junit.jupiter.api package.

b. To mock the TestDemo class, use Mockito.spy. The spy method can be imported with a static import of org.mockito.Mockito.spy.

```
TestDemo mockDemo = spy(testDemo);
```

c. Program the mocked TestDemo object to return 5 when the getRandomInt method is called. Remember to use the form: doReturn(aValue).when(mockedObject).methodCall(). You can use a static import for doReturn: import static org.mockito.Mockito.doReturn;

```
doReturn(5).when(mockDemo).getRandomInt();
```

d. Call the method randomNumberSquared on the mocked TestDemo object. This will call the stubbed out (mocked) method getRandomInt, which now should return the value 5.

```
int fiveSquared = mockDemo.randomNumberSquared();
```

e. Use assertThat to test that the value returned from randomNumberSquared is equal to 5 squared.

```
assertThat(fiveSquared).isEqualTo(25);
```

f. You don't need to verify the mocked method call – you know it was called since the return value is correct.

———————————————————————————————————