

URL to GitHub Repository: <https://github.com/tmccarthy91/Week-12-Coding-Assignment/>

URL to Public Link of your Video: <https://youtu.be/6ScwKZ9JdCY>

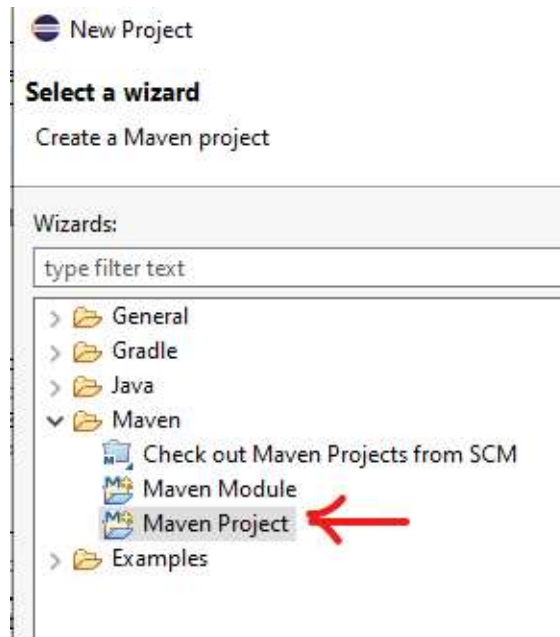
Instructions :

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 Coding Assignment Document:
 - The URL for this week's GitHub repository.
 - The URL of the public link of your video.
 3. Save the Coding Assignment Document as a .pdf and do the following:
 - Push the .pdf to the GitHub repo for this week.
 - Upload the .pdf to the LMS in your Coding Assignment Submission.
-

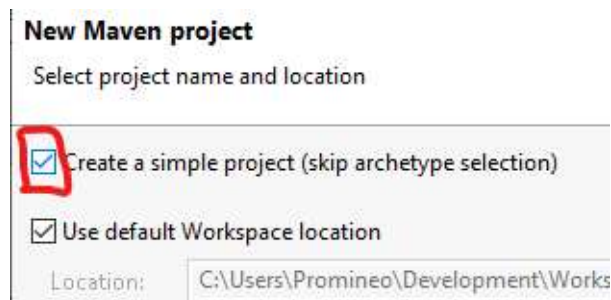
Coding Steps:

Create a new Maven project. In Eclipse...

Right-click in Project Explorer, select "New / Project". Expand "Maven". Select "Maven Project". Click "Next".



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



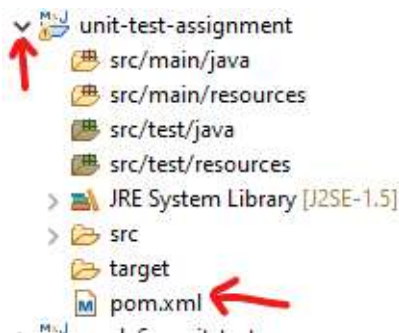
Enter the Group Id: "my.unit.test". Enter the Artifact Id: "unit-test-assignment". Click "Finish".

New Maven project

Configure project

Artifact	
Group Id:	my.unit.test
Artifact Id:	unit-test-assignment
Version:	0.0.1-SNAPSHOT

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



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

```
1 <project xmlns="http://maven.apache.org/POM/4.0.0" >
2   <modelVersion>4.0.0</modelVersion>
3   <groupId>my.unit.test</groupId>
4   <artifactId>unit-test-assignment</artifactId>
5   <version>0.0.1-SNAPSHOT</version>
6
7   |
8
9 </project>
```

Copy and paste the following code into the blank area you just created.

```
<properties>
  <java.version>11</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>
```

```

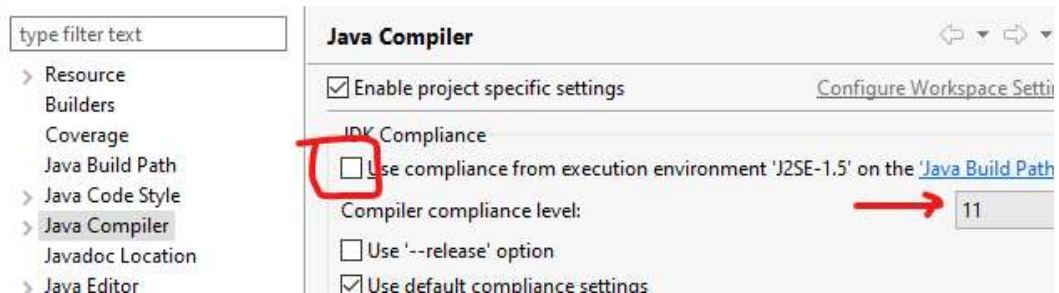
</dependency>
</dependencies>

<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>

```

Save the file.

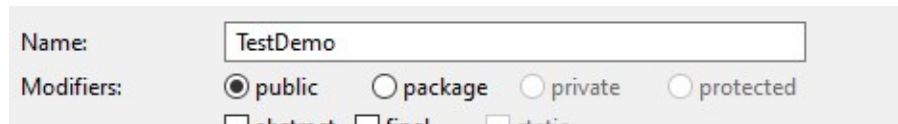
For Eclipse only: right-click on "unit-test-assignment" in the Project Explorer. Click on "Properties". Click "Java Compiler". Make sure "Enable project specific settings" is checked. Uncheck "Use compliance from execution environment 'J2SE-1.5' on the 'Java Build Path'". Set "Compiler compliance level" to 11. Click "Apply and Close".



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

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".)



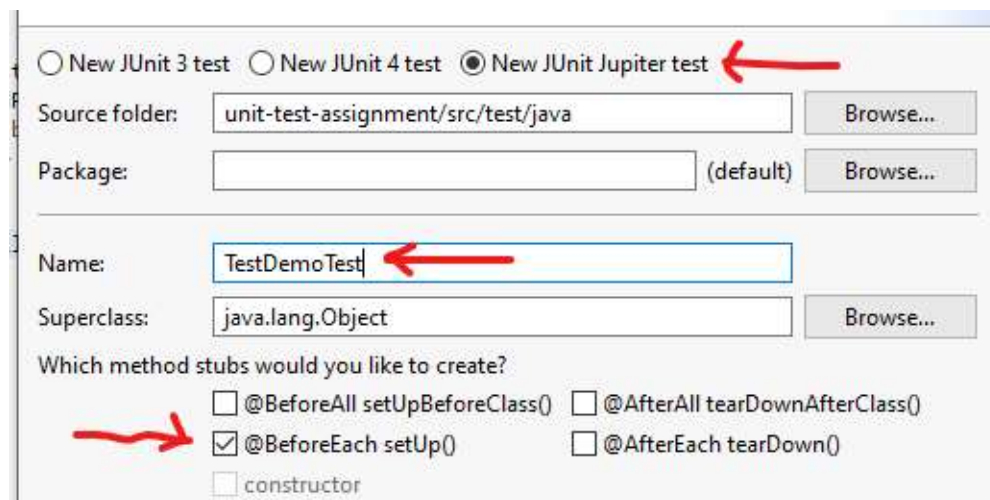
Create an instance method (not static) named `addPositive`. It should take two `int` parameters and return an `int`.

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

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.

Save the file.

In Package Explorer, find "`src/test/java`" and right-click on it. Select "New / JUnit Test Case". In the "Name" field, enter "TestDemoTest". Make sure that "New JUnit Jupiter test" is selected. Make sure that "`@BeforeEach setUp()`" is checked. Click "Finish".



In `TestDemoTest.java`, add a private instance variable of type `TestDemo` named `testDemo`.

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

Change "`@Test`" to "`@ParameterizedTest`". Add the import statement for `org.junit.jupiter.params.ParameterizedTest`.

Change the name of method "`test`" to

`"assertThatTwoPositiveNumbersAreAddedCorrectly".`

Add four parameters to `assertThatTwoPositiveNumbersAreAddedCorrectly` as shown:

Type	Name
int	a
int	b
int	expected
Boolean	expectException

a.

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);  
}
```

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`;

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

Use the assertion `assertInstanceOf(IllegalArgumentException.class)` to ensure that the correct exception is thrown.

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

```
assertThatThrownBy(() ->  
    testDemo.addPositive(a, b))  
    .assertInstanceOf(IllegalArgumentException.class);
```

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 and  
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)
```

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

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("TestDemoTest#argumentsForAddPositive")
```

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.

- a. `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.

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

Write a test for `randomNumberSquared` in `TestDemoTest.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.

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.

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);
```

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();
```

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();
```

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

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

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