Tutorial: Unit Testing With JUnit

Creating And Running Unit Tests

Introduction

An important aspect of every software engineer's life is testing the code he/she created. There are different types of tests but one of the most important, and the type we typically write more than any other is **Unit Tests**. Unit tests test each individual unit of a program independently of all of the other units. In object-oriented programming, we typically define a unit as a class. Unit testing is at the center of test-driven development and other kinds of programming paradigms.

There are numerous ways to implement unit testing in your projects, depending on the language and the IDE you use, as well as personal preference. In this tutorial, you will learn how to perform unit testing on a simple program using the JUnit testing framework. We will be running the unit tests in the Intellij IDEA development environment.

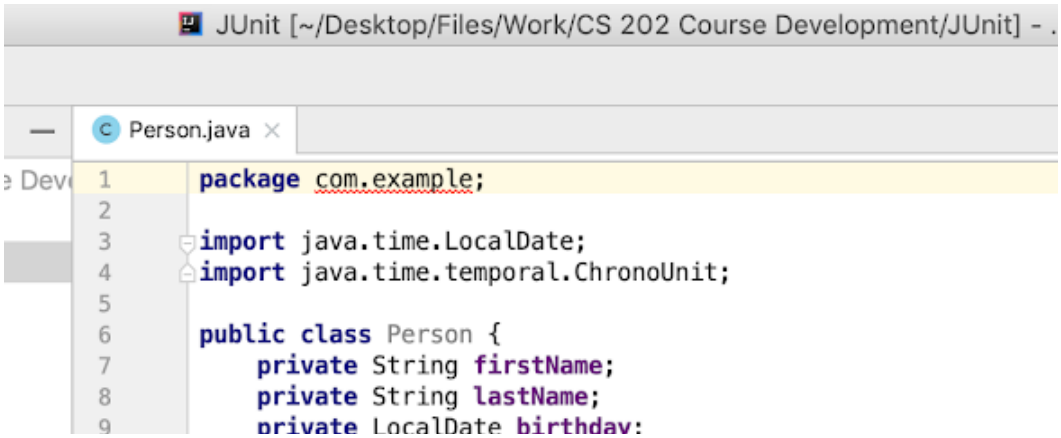
Note: On this and the next several labs and tutorials, you may notice setup issues involving Java versions that seem out of sync, or errors that prevent your tests from executing. If you see any of these types of problems, you may find solutions here.

Tutorial

Intellij IDEA comes with built-in support for JUnit. However, you still need to include JUnit as a dependency in your project to use it. We will show you how to do that later.

1. We are going to demonstrate the basics of JUnit using an example source file. Start by creating a new Maven or Gradle project in IntelliJ. If you need a refresher on how to do that, refer to the "Dependency Management Tools" tutorial you completed earlier.

   Normally we would import our Java file straight into the project but for learning purposes we will do things a bit differently. Create a new Java file called Person.java in your src/main/java folder and copy the code from the provided Person.java into this new Person.java class. The first thing you will notice about this file is the red underline indicating an error with the package declaration:



If you don't have this error in your project, then you have your directory structure set up properly. This error indicates that our file is in the wrong package directory. Package declarations in Java are required to mirror the project/directory structure. Source files must be located in folder structure corresponding to the declared package name. You could set up the folders manually, or you could let IDEA do the work for you. If you place your cursor on the `com.example` line, a little red light bulb will appear just below the package declaration. Hover over it, click on the dropdown arrow, and click "Move to package 'com.example'."

Now your file has been moved to the proper package directory, as in the following image:

Intellij IDEA is a very powerful tool that can aid you in your programming endeavors. It has many checks in place to make your life easier. Whenever you have an error or a warning that can be easily fixed, a small lightbulb icon will appear in either red or yellow (depending on whether it is an error or a warning) with a suggestion for a possible fix. Use these suggestions whenever you can. They can save you a lot of time and keep you from needing to spend time on mundane, scriptable tasks or refactors. This allows you to focus your efforts on the more important programming problems at hand.

You may be wondering at this point in the tutorial why we are covering Java packages in a tutorial about JUnit. Depending on what courses you have taken thus far, packages may be a review or they may be new to you. Either way, it is important to understand how packages work in Java. Unit tests are intended to test small units of code, not entire projects. The principle of Cohesion states that unrelated things (bits of code, methods, source files) should not be located together, but should be separated by how closely they are related to each other.

In the same way, Unit Tests should be closer to the units that they are related to than to units they are unrelated to. In Java, Unit Tests are expected to reside in the same package as the units they are testing. You will run into problems if you do not set up your packages correctly, so always be sure to take some time to make sure that you are following the Java conventions related to package structure.

2. Although both files should be located in the same package, they should not be placed in the same folder, since it is generally a bad idea to ship test files along with source code. To properly separate the source code from the files, we will use a separate testing directory. When you created your project with Maven or Gradle, a ‘src’ directory was

created for you. Within your ‘src’ directory you have a ‘main/java’ directory and a ‘test/java’ directory. Your main project code goes in the ‘main/java’ directory, and your test code goes in your ‘test/java’ directory.

3. Import the latest version of Junit 5 into your pom or build.gradle file from https://mvnrepository.com/artifact/org.junit.jupiter/junit-jupiter-api

   Maven:

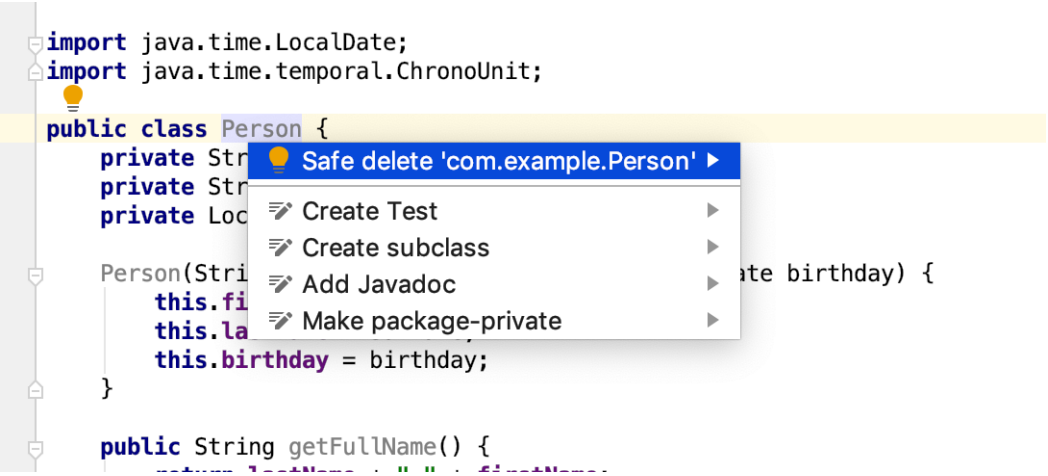
```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>TutorialProject</groupId>
    <artifactId>TutorialProject</artifactId>
    <version>1.0-SNAPSHOT</version>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <source>8</source>
                    <target>8</target>
                </configuration>
            </plugin>
        </plugins>
    </build>
    <dependencies>
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-api</artifactId>
            <version>5.0.3</version>
            <scope>test</scope>
        </dependency>
    </dependencies>


</project>
```

   Gradle:

```
plugins {
    id 'java'
}

group 'TutorialProjectGradle'
version '1.0-SNAPSHOT'

sourceCompatibility = 1.8

repositories {
    mavenCentral()
}

dependencies {
    testCompile group: 'org.junit.jupiter', name: 'junit-jupiter-api', version: '5.0.3'
```

4. Import/Sync your pom or build.gradle file.

5. To create a JUnit test file for the Person.java file, place your cursor on the class name on the class definition and press ALT+ENTER (OPTION+ENTER on Mac) to bring up a list of available options:



4. Select ‘Create Test’ and then select ‘JUnit5’ from the Testing Library dropdown.

5. You should now have an empty PersonTest class in a test directory structure that properly mirrors your src directory structure, including package structure. It is important to maintain this parallel structure to prevent problems in your testing.

Now that your project is set up, it’s time to start writing tests. Ideally you would want to write tests as you write code, but the focus of this tutorial is gaining familiarity with JUnit, so we will write the tests independently of the code.

6. We’ll start by writing a test for the `getFullName()` method. All tests in a JUnit file are placed inside of the class declaration brackets. Tests are written as methods belonging to the class. You may also write “helper” methods that are used by the tests but not run

as tests themselves. To distinguish between test methods and other methods, we place a `@Test` annotation before each method that is to be run as a test. If you have multiple tests in a single file, they will be executed in the order they are defined in the file. Write an empty test for this method now, using the following as a guide:

```
2
3   import org.junit.jupiter.api.Test;
4
5   import static org.junit.jupiter.api.Assertions.*;
6
7   class PersonTest {
8
9       @Test
10      public void testGetFullName() {
11
12      }
```

Because this test is empty, it is immediately considered "passing." If we were to run this test right now, we would get a message stating that all tests had passed, though it wouldn't tell us anything interesting about the program since it isn't actually testing anything. In JUnit, a test is considered passing as long as all `assert` statements (if there are any) pass, and there are no compiler errors. Make sure your tests assert enough to determine if the method is doing its job properly.

7. In order to ensure your test is meaningful, it is important to consider the expected behavior of the method. This will take into account things such as what the method is doing, the expected output, and how the method handles invalid inputs, if it has any parameters (which ours does not). The `getFullName()` method simply returns the full name of the Person object. So, for our test of `getFullName()` to pass, it will need to correctly return the full name for any valid Person object. So, what is a correct full name? In this case, it is entirely up to the discretion of the programmer. Correct full name could be a number of things depending on the use. For this tutorial, we are going to assume that a correct full name follows the form "firstname lastname".

   Write the body of the test method to generate a person object, generate a string containing the expected (correct) full name, and then compare the result of the `getFullName()` method with this expected name string using `assertEquals()`. If the method is working properly, the test will pass, otherwise it will fail. Your test method should look something like this (you can use whatever person name you like):

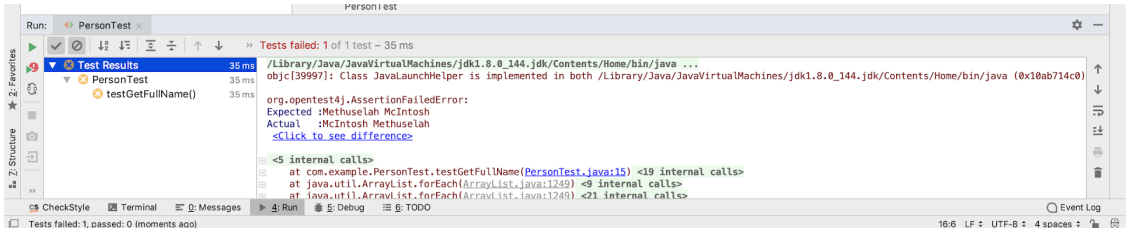
```
@Test
public void testGetFullName() {
    Person testPerson = new Person( givenName: "Methuselah",  surname: "McIntosh", LocalDate.parse("1996-01-01"));
    String expectedFullName = "Methuselah McIntosh";
    assertEquals(expectedFullName, testPerson.getFullName());
}
```

Be sure to put the expected parameter first, followed by the actual parameter, or the error logging will be reversed.

8. Now that we have written our test, it's time to see if it passes. Run the test by right-clicking on the test method and clicking on "Run 'testGetFullName()'", and then wait for Intellij IDEA to build and run the test.

   Note: You can run all tests in a test class by right-clicking on the class name.

9. Once the tests have run, you can check the results:



   As you can see from the error messages above, the test failed. Looking more closely at the output, we can see that the actual output was "McIntosh Methuselah", but we were expecting "Methuselah McIntosh". This is a fairly simple error to make (and fix): we just concatenated the string values in the wrong order in our `getFullName()` method. Correct this error by opening the Person.java file and changing `return lastName + " " + firstName;` to `return firstName + " " + lastName;` in the `getFullName()` method in the `Person` class.

10. Now that you've fixed your code, run your test again. This time your test should pass. If your test still isn't passing make sure you have written the test exactly as shown. Because you have compiled and run your test class already, you now have little red rerun icons to the left of the class and method declarations. You can run your tests again from here or from the menu.

11. Now that you have a test passing for one of your methods, you should write a test case for the other method. This method is a little trickier because it is doing an actual computation. The Person's age is determined by taking the date of the Person's birth (which is stored in the object) and subtracting it from the current date (which is calculated whenever the method is called. This means the correct result will change as time passes, meaning a hard coded value won't always be correct.

This is a situation where we would like to use Dependency Injection to put in a “fake” current date so we will always know in the test what the correct answer should be. We have not learned how to do this yet, but later in the semester we will have a lab on Mockito, which is a tool that allows you to do just that. For now, we will assume that this unit test will only be used today so a hard coded expected value will suffice.

12. Create the unit test for `getAge()`. To be consistent with the previous test case, let’s copy over the Person instance to keep the object the same. The birthdate of this person is January 1, 1996. As of the writing of this tutorial, the year is 2019, so this person should be 23 years old (adjust this value depending on the year that you are taking this course). Your unit test should look like this:

```
8
9   class PersonTest {
10
11      @Test
12      public void testGetFullName() {
13          Person testPerson = new Person( givenName: "Methuselah",  surname: "McIntosh", LocalDate.parse("1996-01-01"));
14          String expectedFullName = "Methuselah McIntosh";
15          assertEquals(expectedFullName, testPerson.getFullName());
16      }
17
18      @Test
19      public void testGetAge() {
20          Person testPerson = new Person( givenName: "Methuselah",  surname: "McIntosh", LocalDate.parse("1996-01-01"));
21          double age = 23;
22          assertEquals(age, testPerson.getAge());
23      }
24  }
```

13. Run both tests again. Both tests should pass.

14. Notice that the first line of both tests is the same. Both tests create a test person with exactly the same values. This is duplicate code. Duplicate code is bad.

    JUnit has a `@BeforeEach` annotation that can be attached to a method. This annotation will cause the corresponding method to be executed before each test method, which allows you to perform setup logic before each test. You can clean up the test file we created by creating a new method with the @BeforeEach annotation attached, and by moving the creation of the test person into the new method. Create a private instance variable to hold the test person and remove the line of code that creates the person from both tests. Your test code should now look like this:

```java
class PersonTest {

    private Person testPerson;

    @BeforeEach
    public void setup() {
        testPerson = new Person( givenName: "Methuselah",  surname: "McIntosh", LocalDate.parse("1996-01-01"));
    }

    @Test
    public void testGetFullName() {
        String expectedFullName = "Methuselah McIntosh";
        assertEquals(expectedFullName, testPerson.getFullName());
    }

    @Test
    public void testGetAge() {
        double age = 23;
        assertEquals(age, testPerson.getAge());
    }
}
```

Notice that the test methods no longer contain duplicate code.

JUnit also has annotations `@AfterEach`, `@BeforeAll`, and `@AfterAll`. These annotations allow you to create methods that run after each test, once before all the tests, and once after all the tests respectively.

Submission

Electronically submit copies of your `PersonTest.java` file and your modified `Person.java` file to receive credit for today's lab.