**JUnit**

**Introduction**

1. [*Unit Testing*](https://en.wikipedia.org/wiki/Unit_testing) is a method for testing an individual class and unit tests are usually written by the developer.
2. When a pull request is initiated unit tests must be present and passed before being merged.
3. How do you test a class with an association to another classes? There are two approaches: sociable tests and solitary tests. For example, suppose class *A* has-a *B*. A sociable unit testing approach has unit tests for class *B* and when they pass, unit tests for class *A* are written using the actual *B* class. Solitary tests may have unit tests for *B*, but the unit tests for *A* use a [*test double*](https://martinfowler.com/bliki/TestDouble.html)([2](https://en.wikipedia.org/wiki/Test_double))for *B.* A test double is an object(s) that looks and behaves like their release-intended counterparts, but are actually simplified versions that reduce the complexity and/or resources needed for testing. For now, we will only consider sociable unit testing.

Source: <https://martinfowler.com/bliki/UnitTest.html>

1. JUnit 5 (also called JUnit Jupiter) is a framework for doing unit testing in Java. JUnit 5 support is available in Eclipse Oxygen 4.7.1a or higher. A JUnit test is a class with methods that test another class(es). If the class you are testing is *MyClass* then typically, the name of the test class is *MyClassTester*.

**Example 1**

1. Suppose we have an *Employee* class with a method:

**public** **double** getPay(**double** hours) {

**if**(hours>0.0)

**return** hours\*payRate;

**return** 0.0;

}

1. To create a JUnit test class in Eclipse, choose: File, New, JUnit Test Case (make sure New JUnit Jupiter test is selected), give it a name, and choose: Finish.
2. For example, the test class might look like this:

**import** **static** org.junit.jupiter.api.Assertions.\*;

**import** org.junit.jupiter.api.DisplayName;

**import** org.junit.jupiter.api.Test;

**class** EmployeeTester {

@Test

@DisplayName("getPay with hours greater than zero")

**void** test\_getPay\_positive\_hours() {

Employee e = **new** Employee(20.0);

**double** actualPay = e.getPay(40.0);

**double** expectedPay = 800.0;

*assertEquals*(expectedPay, actualPay);

}

@Test

@DisplayName("getPay with hours less than zero")

**void** test\_getPay\_negative\_hours() {

Employee e = **new** Employee(20.0);

**double** actualPay = e.getPay(-40.0);

**double** expectedPay = 0.0;

*assertEquals*(expectedPay, actualPay);

}

}

1. To run the test class, simply make sure it is the active window and choose the green run button. The package explorer will display a new tab as shown below:



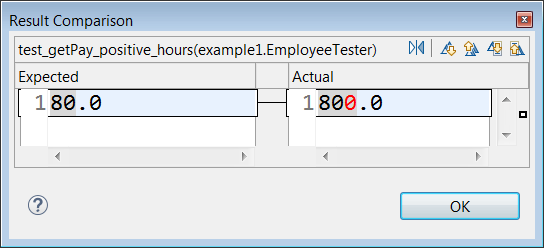
1. If a test fails, the jUnit tab looks like this:



Below this, you will see a Failure Trace pane:



Clicking on the right-most icon in the upper-right displays the Result Comparison dialog:



**Writing Tests**

1. The link below shows the API for JUnit. To see the API for the the *Assertions* class, choose “Assertions” from the *All Classes* menu on the left.

<http://junit.org/junit5/docs/current/api/>

Mostly you would use these methods: *assertEquals, assertTrue, assertFalse, fail* Next most frequent might be: *assertArrayEquals, assertIterableEquals, assertNotEquals, assertNotSame, assertNull, assertSame, assertThrows*

1. Each test method should do one thing.

Source: <https://stackoverflow.com/questions/235025/why-should-unit-tests-test-only-one-thing>

“Rule of thumb here on a failed test report: if you have to read the test's code first then your test are not structured well enough and need more splitting into smaller tests.” In other words, the *DisplayName* should explain adequately what the test does without having to read the code.

1. Tests should be readable:

“The intent of a unit test should be clear. A good unit test tells a story about some behavioral aspect of our application, so it should be easy to understand which scenario is being tested and — if the test fails — easy to detect how to address the problem. With a good unit test, we can fix a bug without actually debugging the code!”

Source: <https://www.toptal.com/qa/how-to-write-testable-code-and-why-it-matters>

“Professional: In the long run you'll have as much test code as production (if not more), therefore follow the same standard of good-design for your test code. Well factored methods-classes with intention-revealing names, No duplication, tests with good names, etc.”

Source: <https://stackoverflow.com/questions/61400/what-makes-a-good-unit-test>

“Readable – This can be considered part of Professional - however it can't be stressed enough. An acid test would be to find someone who isn't part of your team and asking him/her to figure out the behavior under test within a couple of minutes. Tests need to be maintained just like production code - so make it easy to read even if it takes more effort. Tests should be symmetric (follow a pattern) and concise (test one behavior at a time). Use a consistent naming convention (e.g. the TestDox style). Avoid cluttering the test with "incidental details”, become a minimalist.

Source: <https://stackoverflow.com/questions/61400/what-makes-a-good-unit-test>

1. Other useful annotations

**class** EmployeeTester2 {

@BeforeAll

**static** **void** setUpBeforeClass() **throws** Exception {

}

@AfterAll

**static** **void** tearDownAfterClass() **throws** Exception {

}

@BeforeEach

**void** setUp() **throws** Exception {

}

@AfterEach

**void** tearDown() **throws** Exception {

}

@Test

@DisplayName("Brief description of test")

**void** test() {

*fail*("Not yet implemented");

}

@Disabled("Failing for unknown reason")

@Test

@DisplayName("Brief description of test")

**void** test02() {

*fail*("Not yet implemented");

}

}

1. Nested tests:

@DisplayName("Tests for HW 5")

**class** HW5\_Tester {

@Nested

@DisplayName("Tests for MartianManager class")

**class** MartianManagerTester {

@Test

@DisplayName("addMartian return correct when adding RedMartian")

**void** testMartianManager\_addMartian\_red\_successful() {

*assertTrue*(mm.addMartian(r1));

}

...

@Nested

@DisplayName("Tests for Martian class")

**class** MartianTester {

@Test

@DisplayName("RedMartian toString contains id and volume")

**void** testRedMartian\_toString() {

*assertTrue*(r1.toString().contains(String.*valueOf*(r1.getId())));

*assertTrue*(r1.toString().contains("1"));

}

...

}

}

1. You can also define Test Suites which specify which test classes and/or methods to run. There is a bug in Eclipse and they cannot be run from within Eclipse, but can be run from the command line.