Unit Testing

Contents

[1 Introduction – Testing 2](#_Toc135732609)

[2 Introduction – Unit Testing 2](#_Toc135732610)

[3 Introduction – JUnit Testing 2](#_Toc135732611)

[4 JUnit Example 3](#_Toc135732612)

[5 JUnit in Eclipse 3](#_Toc135732613)

[6 Other Useful JUnit Annotations 6](#_Toc135732614)

[7 Guidelines for Writing Unit Tests 7](#_Toc135732615)

[*8* Unit Testing Associated Classes 8](#_Toc135732616)

[*9* Mocking and Test Class Independence 9](#_Toc135732617)

[10 Integration Testing 11](#_Toc135732618)

[11 Code Coverage 11](#_Toc135732619)

[12 Continuous Integration and Delivery 11](#_Toc135732620)

[13 Example – Omit Fall 2022 12](#_Toc135732621)

[14 Interviews with Developers about Testing 12](#_Toc135732622)

[Appendix 1 JUnit API – assertEquals(exp,act) & assertSame(Object exp, Object act) 14](#_Toc135732623)

[Appendix 2 JUnit API – assertTrue(booleanExpression) & assertFalse(booleanExpression) 15](#_Toc135732624)

[Appendix 3 JUnit API – assertArrayEquals(Object[] expected, Object[] actual) 16](#_Toc135732625)

[Appendix 4 JUnit API – assertThrows(Class<T> expectedType, Executable executable) 17](#_Toc135732626)

[Appendix 5 JUnit API – Comparing ArrayList’s with assertEquals(Object expected, Object actual) 17](#_Toc135732627)

[Appendix 6 JUnit API – assertAll(Executable… executables) 17](#_Toc135732628)

[Appendix 7 JUnit API – assertTimeout(Duration timeout, Executable executable) 18](#_Toc135732629)

# Introduction – Testing

[Unit Testing](https://en.wikipedia.org/wiki/Unit_testing) refers to testing software at the lowest level. For object-oriented programming, usually that is a class. Unit tests are usually written by the developer. Usually, unit testing means testing each class in isolation. In other words, there is no dependency on another class, a database, file system, or communication across a network.

[Integration Testing](https://en.wikipedia.org/wiki/Integration_testing) involves combining unit tested modules into larger structures and testing. There are various approaches: big-bang, bottom-up, top-down, sandwich, and others.

[System Testing](https://www.guru99.com/system-testing.html) is conducted after integration testing and involves the complete software system and is conducted using the UI.

A very thorough overview of software testing: <https://en.wikipedia.org/wiki/Software_testing>

# Introduction – Unit Testing

A unit test for a class is composed individual test cases where each test case is implemented as a separate method inside a test class. This general steps in each test method are:

1. Create expected output
2. Create object to be tested
3. Call method being tested and extract actual output
4. Compare expected and actual output

In order to automate unit testing, the code being tested is usually executed within a framework. A common framework for Java is [JUnit](https://en.wikipedia.org/wiki/JUnit), which is integrated into Eclipse, IntelliJ, and most other IDEs. There are [frameworks](https://en.wikipedia.org/wiki/Unit_testing#Unit_testing_frameworks) for unit testing for many languages. The remainder of these notes assumes you are using JUnit.

# Introduction – JUnit Testing

When using JUnit, the way that the expected and actual output are “compared”, is to *assert* something about the output that must be true for the test to be considered successful (pass), otherwise it fails. Often, we assert that the expected and actual output are equal. For example:

**int** expectedNumAccounts = 3;

**int** actualNumAccounts = person.getNumAccounts();

*assertEquals*(expectedNumAccounts, actualNumAccounts);

*assertEquals* is a method in the JUnit framework that compares the two arguments. If they have the same value then the framework records this test as passing.

Another type of assert is to assert that some condition is true:

*assertTrue*(course.isClassFull());

*assertTrue* is a method in the JUnit framework that accepts a boolean expression. If the expression is *true*, then the framework records this test as passing.

The JUnit framework runs all test methods and provides a summary at completion as we illustrate [below](#_JUnit_in_Eclipse).

# JUnit Example

Example – Consider the *Employee* class shown on the right where we want to test the *getPay* method shown below:

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

 **if**(hours>0.0)

 **return** hours\*payRate;

 **return** 0.0;

}

 A test case in a JUnit test might look like this:

**void** testGetPay\_positive\_hours() {

 **double** expectedPay = 800.00;

 Employee e = **new** Employee("dg", 20.0);

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

 *assertEquals*(expectedPay, actualPay);

}

Resources:

* API for the *Assertions* class that contains all the static *assert…* methods:

<https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/Assertions.html>

* Some common assert methods are considered in the [Appendix](#JUnit_API): assertAll, assertEquals, assertTrue, assertFalse, assertArrayEquals, assertIterableEquals, assertNotEquals, assertNotSame, assertNull, assertSame, assertThrows, assertTimeout
* JUnit 5 User Guide: <https://junit.org/junit5/docs/current/user-guide/>
* JUnit 4 FAQ: <https://junit.org/junit4/faq.html>

# JUnit in Eclipse

JUnit 5 (also called JUnit Jupiter) is the most current framework for doing unit testing in Java. JUnit 5 support is built-in to Eclipse since 2017.

To create a JUnit test class in Eclipse, choose:

1. File, New, JUnit Test Case
2. Make sure *New JUnit Jupiter test* is selected
3. Supply a class name. The convention for naming test classes is: *ClassNameTest*. For example shown on the right, the class under test is *Employee* and the test class name is: *EmployeeTest*
4. Specify the class under test
5. Next.
6. On the next dialog, you can choose the methods you want to test and it will create test method stubs for these. I sometimes do this as a reminder of all the methods. But, two points: (1) we don’t have to test all methods, we only test methods that have some logic (we say more about this later), (2) there will be several or more tests for each method that we do test.
7. The resulting test class will look like this:

**class** EmployeeTest {

 @Test

 **void** testEmployee() {

 *fail*("Not yet implemented");

 }

 @Test

 **void** testGetName() {

 *fail*("Not yet implemented");

 }

 ...

}

Example – A (abbreviated) set of JUnit tests for the *Employee* class above, might look like as shown below.

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

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

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

**class** EmployeeTest {

 @Test

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

 **void** testGetPay\_positive\_hours() {

 **double** expectedPay = 800.00;

 Employee e = **new** Employee("dg", 20.0);

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

 *assertEquals*(expectedPay, actualPay);

 }

 @Test

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

 **void** testGetPay\_negative\_hours() {

 Employee e = **new** Employee("dg", 20.0);

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

 **double** expectedPay = 0.0;

 *assertEquals*(expectedPay, actualPay);

 }

}

Note:

* A number of imports are required. Most of them must be added by letting Eclipse resolve the compilation error (or typing them yourself)
* The “@Test” annotation is required for a method to be considered a test method which is run automatically.
* By convention, test methods are named with this syntax:

testMethodNameContext

for example:

testGetPayPositiveHours

I don’t think this is common, but I generally put “\_” between words in the context portion, for readability:

testGetPay\_positive\_hours

* The “@DisplayName” annotation is optional; however, it is useful because it appears in JUnit results tab (considered shortly). Thus, it should be descriptive.

To run the test class, simply make sure it is the active window and choose the green run button (just as you would run any Java program). The package explorer will display a new tab as shown below. Sometimes it will only run one method. When this happens, I click somewhere at the top of the file (outside a method) and then it usually runs all of them.



If a test fails, the JUnit tab looks like this:



Below this, you will see a Failure Trace pane as shown on the left below. Clicking on the right-most icon in the upper-right displays the Result Comparison dialog. Note that this dialog is not always available (the icon will be grayed out). I’m not sure why. I haven’t found this useful, but maybe I am missing something.

|  |  |
| --- | --- |
|  |  |

# Other Useful JUnit Annotations

1. Other useful method annotations:
2. *@BeforeAll* – code in this method is run before any tests have been run
3. *@AfterAll* – code in this method is run after all tests have been run
4. *@BeforeEach* – code in this method is run before each test is run
5. *@AfterEach* – code in this method is run after each test is run
6. *@Disabled* – test method is not run
7. Example

**class** EmployeeTest {

 @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. Tests can also be nested by writing inner classes:

@DisplayName("Tests for HW 5")

**class** HW5\_Tester {

 @Nested

 @DisplayName("Tests for MartianManager class")

 **class** MartianManagerTest {

 @Test

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

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

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

 }

 ...

 @Nested

 @DisplayName("Tests for Martian class")

 **class** MartianTest {

 @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. A reference is [here](https://www.vogella.com/tutorials/JUnitEclipse/article.html) (directions should say: File, New, Other, Java, JUnit, JUnit Test Suite. However, no test classes show up to select)

# Guidelines for Writing Unit Tests

1. **Each test method should do one thing**. For example, if testing a *Stack* class, you wouldn’t have a test that tested *push*, *pop*, and *peek* in the same test. However, to *pop*, you must first have pushed, so, *push* should be tested first before testing *pop*.
2. **Test all relevant situations for each method**. Almost all methods will have at least two tests: one that is a success scenario (*e.g.* there are 3 items in the stack when *pop* is called) and one that is a failure scenario (*e.g.* there are no items in the stack when *pop* is called).
3. **Test all side-effects of a method**. Some methods only have one side-effect. For example, a *deposit(amt)* method might only modify the balance. Other methods may have several side-effects. For example, the *pop* method for the *Stack* class has (at least) two side-effects:
* The top of the stack is returned
* The number of items in the stack is reduced by 1.

Thus, when testing a method, you need to carefully think of all the side-effects that can result. I’m not sure if best practice is to test each side-effect in a separate method, or all in one method. I recommend testing them all in one test method.

1. **Test methods should be independent**. In other words, each test method should start from scratch. A simple example of non-independent tests is if you have some setup code that sets the state of some class, then run a test method that alters that state, then run a test method that continues to alter that previously-altered state.

Many test methods will have exactly the same setup code, followed different code for the actual test. In this case, it is best to write a helper method to do the setup. For example, suppose you have a *Bank* class which has a number of *Account* objects. A *buildBank(numAccounts:int):Bank* method would be useful as some of the test methods could call this to create a *Bank* object with a number of *Account* objects attached. Of course, you could only use such a method once the *Bank’s addAccount* method had been tested.

1. **Only test methods with logic**. For example, getters and setters aren’t tested unless they have some logic (computation, loop, branch). The same for constructors or any methods with no logic. In practice, you find that many methods don’t need to be tested.
2. **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>

 “…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 … 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>

“Rule of thumb here on a failed test report: if you have to read the test's code first then your tests 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.

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

1. A set of JUnit Best Practices that I found useful.

<https://howtodoinjava.com/best-practices/unit-testing-best-practices-junit-reference-guide/>

<http://www.kyleblaney.com/junit-best-practices/>

# Unit Testing Associated Classes

Consider the case where you have an association between two classes, *e.g.* a *Company* has many *Employee*s.



One approach to unit testing these classes is to use *sociable* tests. In the case above, we would unit test *Employee*, and when all the tests passed, then we would unit test *Company.* This is, of course not possible if there is two-way navigability.

Another approach is to use *solitary tests* where we would use a *mock* of the *Employee* class when testing the *Company* class. We consider *mocking* next.

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

# Mocking and Test Class Independence

A class under test should be independent of other classes it depends on (has associations with) or any external resources such as databases, files, *etc.* There are a number of techniques[[1]](#footnote-1) [[2]](#footnote-2) [[3]](#footnote-3) for doing this including test doubles, mocks, fakes, stubs, and others. We will only consider the general notion of a *mock*. There are two general approaches to mocking:

1. Class-based – If *A* depends on *B* and you want to test A, you create a mock of *B, MockB* that extends *B* and overrides the methods to simply return hard-coded data. For example, suppose the *setB* method in *A* calls *foo* on its *B* instance and does some other processing. To unit test *setB*, we would mock the call to *foo.* In *MockB,* the overridden *foo* simply returns an integer that was supplied in the mock’s constructor.



1. Interface-based – Create an interface that both *B* (the real class) and *MockB* implement, similar to the *class-based* approach.



Example – Consider an example where a *Company* has a number of *Employee* objects and that we want to test the *getPayrollTotal* method in the *Company* class:

**public** **double** getPayrollTotal() {

 **double** sum=0.0;

 **for**(**int** i=0; i<numEmps; i++) {

 sum += emps[i].getPay(40);

 }

 **return** sum;

}

Notice that the method above depends on the *getPay* method in the *Employee* class. Thus, we could create a mock of *B* with:

**public** **class** MockEmployee **extends** Employee {

 **private** **double** pay;

 **public** MockEmployee(String name, **double** payRate, **double** pay) {

 **super**(name, payRate);

 **this**.pay = pay;

 }

 @Override

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

 **return** pay;

 }

}

Notice that we are passing in the *pay* to the constructor and that *getPay* simply returns that value. Finally, the unit test would be:

**void** test\_getPayRollTotal() {

 Employee[] empsExpected = **new** Employee[10];

 Company company = **new** Company();

 Employee e1 = **new** MockEmployee("Todd", 50.0, 2000.0);

 Employee e2 = **new** MockEmployee("Rita", 40.0, 1600.0);

 Employee e3 = **new** MockEmployee("Suze", 60.0, 2400.0);

 company.addEmployee(e1);

 company.addEmployee(e2);

 company.addEmployee(e3);

 **double** expOutput = 6000.0;

 **double** actOutput = company.getPayrollTotal();

 *assertEquals*(expOutput,actOutput);

}

Notice that when mocking above, we are only testing *getPayrollTotal* without any possible interference from *getPay*.

Manual mocking is occasionally done; however, mostly developers use a mocking framework. Three common frameworks are: [Mockito](https://site.mockito.org/), [JMockit](https://jmockit.github.io/), [EasyMock](http://easymock.org/).

This article explains at a high-level, the two major approaches that mocking frameworks take to implement mocking.

<https://dzone.com/articles/the-concept-mocking>

Finally, consider unit testing code that depends on a database:

Because some classes may have references to other classes, testing a class can frequently spill over into testing another class. A common example of this is classes that depend on a database: in order to test the class, the tester often writes code that interacts with the database. This is a mistake, because a unit test should usually not go outside of its own class boundary, and especially should not cross such process/network boundaries because this can introduce unacceptable performance problems to the unit test-suite. Crossing such unit boundaries turns unit tests into integration tests, and when such test cases fail, it may be unclear which component is causing the failure. Instead, the software developer should create an abstract interface around the database queries, and then implement that interface with their own mock object. By abstracting this necessary attachment from the code (temporarily reducing the net effective coupling), the independent unit can be more thoroughly tested than may have been previously achieved. This results in a higher-quality unit that is also more maintainable.

<https://en.wikipedia.org/wiki/Unit_testing>

# Integration Testing

Integration testing takes as its input modules (classes) that have been unit tested, groups them in larger aggregates, applies tests. Some approaches:

* Big-bang – Most of the developed modules are coupled together to form a complete software system or major part of the system and then used for integration testing.
* Bottom-up – The lowest level components are tested first, then used to facilitate the testing of higher level components. The process is repeated until the component at the top of the hierarchy is tested.
* Top-down – Top integrated modules are tested and the branch of the module is tested step by step until the end of the related module.
* Sandwich testing is an approach to combine top down testing with bottom up testing.

Source: <https://en.wikipedia.org/wiki/Integration_testing>

The term integration testing is used in various ways:

* Narrow integration tests – Uses test doubles and are often no larger in scope than a unit test
* Broad Integration tests – Uses live versions of all services. This is sometimes referred to a *system test* or an *end-to-end* test.
* Some refer to sociable unit tests as integration tests. This is the approach we take.

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

# Code Coverage

Code coverage measures (in percentage) how much of the code is executed when the unit tests are run. Normally, code with high coverage has a decreased chance of containing undetected bugs, as more of its source code has been executed in the course of testing. One such plugin for Eclipse is: <https://www.eclemma.org/>

# Continuous Integration and Delivery

*Continuous Integration* is the process of merging all working code in branches to the master branch some number of times per day. Its intention is to prevent integration problems.

The longer a branch of code remains checked out, the greater the risk of multiple integration conflicts and failures when the developer branch is reintegrated into the main line. When developers submit code to the repository they must first update their code to reflect the changes in the repository since they took their copy. The more changes the repository contains, the more work developers must do before submitting their own changes.

Eventually, the repository may become so different from the developers' baselines that they enter what is sometimes referred to as "merge hell", or "integration hell", where the time it takes to integrate exceeds the time it took to make their original changes.

Continuous integration involves integrating early and often, so as to avoid the pitfalls of "integration hell". The practice aims to reduce rework and thus reduce cost and time.

A complementary practice to CI is that before submitting work, each programmer must do a complete build and run (and pass) all unit tests. Integration tests are usually run automatically on a CI server when it detects a new commit.

 Source: <https://en.wikipedia.org/wiki/Continuous_integration>

Additional information:

For practical purposes, nearly every CI setup uses automated test suites. When code is ready to merge, an automated process kicks off a new codebase build. The automation also runs a suite of quality assurance tests to check for bugs and determines if the update introduces integration problems. Continuous delivery is the next step after CI, in which you also automate release cycles.

Source: <https://www.hpe.com/us/en/insights/articles/continuous-integration-and-delivery-tool-basics-1807.html>

[Gradle](https://gradle.org/) is a framework and API that is used for CI and possibly CD. It can be found in Eclipse by choosing: File, New, Project and choosing Gradle.

[Maven](https://maven.apache.org/) is similar and also supports reporting and documentation. It is also available in Eclipse.

# Example – Omit Fall 2022

Here, we consider unit and integration testing for the Student-Course example. The code is on the Schedule, beside the “many-to-many” handout considered earlier. It is found in the *ver4* package. Code will be explained in class or their will be a video about it.

# Interviews with Developers about Testing

|  |  |
| --- | --- |
| Interviewee:  | VSU CS Graduate |
| Company: | Rally Software – Develops tools/platform that supports agile software development. |
| Time Frame:  | 2018 |

* Codebase of 200K lines of code (LOC) was refactored, removing about 50K LOC, leaving the code base about 150K LOC
* 3500 unit tests totaling about 60K LOC, about 4.5 minutes to run. Employee mocks for database and other external services.
* 7000 integration tests, about 12 minutes to run.
* End-to-end testing, tests the entire stack. 8.5 hours to run. Uses browser automation. After a project to speed this up, reduced to 20 minutes.

|  |  |
| --- | --- |
| Interviewee:  | VSU CIS Graduate |
| Company: | IBM – WebSphere Application Server, framework that hosts Java based web applications |
| Time Frame:  | 2005 (updated in 2012) |

* Product: IBM's WebSphere Application Server. 600 programmers, 300 testers.
* The software lifecycle conforms to ISO standards as some customers require this. These standards basically spell out the software development process: practices, documents, players, signoffs,...
* Test group has areas: configuration testing, language verification testing, functional testing, system testing (longer form tests), and performance testing (stress loads).
* 23 Operating systems, 13 languages. Very few bugs are operating system specific. So, for a given product version test, only 2-3 operating systems may be used.
* All testing is currently Black Box testing. The philosophy is, "use it like the customer would." In the past (as recent as 5 years ago?), extensive white-box testing was done.
* Test plans define the test and what areas will be tested, what areas have changed, and map to target release dates.
* Tests are written as *Scenarios* with a very detailed description of the assumptions and setup.
* Regression testing is essential, but balanced with economics.
* Automated testing has its economic tradeoffs. You can put a lot of time into developing an automated test that can be rerun at any time. However, most testing is not automated in this way due to the economics. Automated testing is also problematic in software that is being constantly updated because constant changes often confuse the automation. Automation is best used for regression testing to ensure that new changes have not broken established functionality.
* On a recent project, he managed a group of 80 testers; 60 on-site and 20 in China. After the scenarios are made, he spends lots of his time making graphs of defect rates and other statistics. He recently wrote a program in VB to automate the graph making process.
* Defects (which occur by the hundreds) are tracked in a database with detailed descriptions of the errors and relevant output. They are also ranked with a severity code. Engineers try to fix them and return the code for retesting.
* Outsourcing has its benefits and drawbacks. Average cost of labor is $100,000 in US and $15,000 in China, about 7 times less than the US. Testers corresponding with software developers in China can take time. Ask a question, get response 12 hrs. later, 12-hour cycle. Are they as good? Probably not, in his opinion, but are they at least 1/7 as good? That is the economic question.
* All products go out the door with defects. It is economics again.
* At IBM and many other companies testing is a viable career path. Compensation is similar to the pay for developers.

Appendix

1. JUnit API – assertEquals(exp,act) & assertSame(Object exp, Object act)
2. A few basic overloaded *assertEquals* are show below and should be self-explanatory: the test passes if both arguments have the same value.

static void assertEquals​(char expected, char actual)

static void assertEquals​(double expected, double actual)

static void assertEquals​(int expected, int actual)

1. This assert below asserts that expected and actual are equal within the given delta, $\left|exp-act\right|\leq delta$

static void assertEquals​(double expected, double actual, double delta)

1. To compare two objects you may use: assertEquals(Object exp, Object act). This method uses *equals* to decide if two objects are equal, and hence, return true. If *equals* is not overridden then *Object*’s implementation is used which compares memory locations.
2. Example - Consider the class diagram on the right: a *Company* has many *Employees.*
3. Note the following about the *Employee* class:
* It does not override *equals*.

And the *Company* class:

* It has an array of *Employee* objects which are stored sequentially. For example, if there are three employees, they will occupy positions: 0, 1, 2.
* *addEmployee* adds the argument to the next sequential element in the array.
* *removeLastEmployee* removes and returns the last employee in the array
1. A correct test of the *removeLastEmployee* method is shown below which passes. We add two employees and then call *removeLastEmployee* verifying that it is the same as the second employee added.

Company company = **new** Company();

company.addEmployee(**new** Employee(4334, "Todd", 53.23));

Employee e\_Expected = **new** Employee(1393, "Rita", 61.86);

company.addEmployee(e\_Expected);

Employee e\_Actual = company.removeLastEmployee();

*assertEquals*(e\_Expected,e\_Actual);

1. An incorrect test of the *removeLastEmployee* method is shown below which fails. It is essentially the same test as the last one; however, *e\_Expected* and *e\_Actual* are not the same instance as careful examination of the code shows. Thus, you need to be careful about testing object equality.

Company company = **new** Company();

company.addEmployee(**new** Employee(4334, "Todd", 53.23));

company.addEmployee(**new** Employee(1393, "Rita", 61.86));

Employee e\_Expected = **new** Employee(1393, "Rita", 61.86);

Employee e\_Actual = company.removeLastEmployee();

*assertEquals*(e\_Expected,e\_Actual);

1. *assertSame(Object expected, Object actual)* – This method returns true if the arguments are the same object. Thus, it is the same as *assertEquals* if the class of the argument does not override *equals*.
2. There is not an *assertEquals(String exp, String act)* overload. However, strings are objects so *assertEquals(Object exp, Object act)* is used to compare strings and *String* overrides *equals* so that two string are equal if they have the same content.
3. JUnit API – assertTrue(booleanExpression) & assertFalse(booleanExpression)
4. *assertTrue* and *assertFalse* are illustrated in the examples below:
5. The *addMartian* method adds a *Martian* to the *MartianManager* as long as it doesn’t already exist, and returns *true*. If the martian being added already exists, the it is not added, and returns *false. Martians* are consider equal if they have the same *id* (the value is passed into the constructor for a *Martian*).

@Test

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

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

 Martian r1 = **new** RedMartian(8);

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

}

@Test

@DisplayName("addMartian return correct when adding a duplicate")

**void** testMartianManager\_addMartian\_duplicate() {

 Martian r3 = **new** RedMartian(2);

 Martian g1 = **new** GreenMartian(11);

 Martian g4 = **new** GreenMartian(2);

 mm.addMartian(r3); mm.addMartian(g1);

 *assertFalse*(mm.addMartian(g4));

}

1. Test that *toString* contains the *id* of a *Martian*:

@Test

@DisplayName("RedMartian toString contains id")

**void** testRedMartian\_toString() {

 Martian r1 = **new** RedMartian(8);

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

}

1. JUnit API – assertArrayEquals(Object[] expected, Object[] actual)
2. A few basic *assertArrayEquals* are show below and should be self-explanatory: the test passes if there is a one-to-one match in values between elements in the two arrays.

static void assertArrayEquals​(boolean[] expected, boolean[] actual)

static void assertArrayEquals​(char[] expected, char[] actual, String message)

static void assertArrayEquals​(double[] expected, double[] actual)

static void assertArrayEquals​(double[] expected, double[] actual, double delta)

static void assertArrayEquals​(int[] expected, int[] actual)

static void assertArrayEquals​(Object[] expected, Object[] actual)

1. *assertArrayEquals(Object[] expected, Object[] actual) –* Asserts that expected and actual object arrays are deeply equal. Thus, if the class of the array elements override *equals*, then that is used to compare elements. Otherwise, the default *equals* is used which returns true if two elements are the same object. Some examples:
2. If the *Employee* class overrides *equals* such that equality is defined as two instances having the same *name* property, then this test passes:

Employee e1 = **new** Employee("dg", 20.0);

Employee e2 = **new** Employee("ab", 40.0);

Employee[] emps1 = {e1,e2};

Employee e3 = **new** Employee("ab", 90.0);

Employee[] emps2 = {e1,e3};

*assertArrayEquals*(emps1,emps2);

1. Consider the *Company* and *Employee* classes from above. Test to see that *addEmployee* is working correctly. This test passes. Note that since the *Company* class’s *emps* array is protected, we can directly access it as the “actual” argument.

Employee[] empsExpected = **new** Employee[10];

Company company = **new** Company();

empsExpected[0] = **new** Employee("Todd", 53.23);

empsExpected[1] = **new** Employee("Rita", 61.86);

empsExpected[2] = **new** Employee("Suze", 33.93);

**for**(**int** i=0; i<3; i++) company.addEmployee(empsExpected[i]);

*assertArrayEquals*(empsExpected,company.emps);

1. Test to see if *removeLastEmployee* is working. Note that this method does not nullify the element that is removed, it simply decrements the number of employees (see code above). Thus, this test fails because the 4th elements are different. In the “expected” argument it is null; while in the “actual” argument it is: Employee(5911, "Dave", 46.36).

Employee[] empsExpected = **new** Employee[10];

Company company = **new** Company();

empsExpected[0] = **new** Employee("Todd", 53.23);

empsExpected[1] = **new** Employee("Rita", 61.86);

**for**(**int** i=0; i<2; i++) company.addEmployee(empsExpected[i]);

company.addEmployee(**new** Employee("Dave", 46.36));

company.removeLastEmployee();

*assertArrayEquals*(empsExpected,company.emps);

1. A way to fix the previous test is to build an *empsActual* array from the *emps* instance variable:

Employee[] empsActual = **new** Employee[10];

**for**(**int** i=0; i<company.numEmps; i++) {

 empsActual[i] = company.emps[i];

}

*assertArrayEquals*(empsExpected,empsActual);

1. JUnit API – assertThrows(Class<T> expectedType, Executable executable)
2. Asserts that a particular exception is thrown. The *expectedType* can be the actual type of exception that is thrown or any supertype. The *executable* can be specified as a lambda expression.
3. Example:
4. Consider the *getEmployee* method for the *Company* class

**public** Employee getEmployee(**int** i) {

 **if**(i<0 || i>numEmps-1)

 **throw** **new** IllegalArgumentException("Index i=" + i + " invalid");

 **return** emps[i];

}

1. The test below passes. It would also pass when the first argument is: *RuntimeException.class*

Employee[] empsExpected = **new** Employee[10];

Company company = **new** Company();

empsExpected[0] = **new** Employee(4334, "Todd", 53.23);

empsExpected[1] = **new** Employee(1393, "Rita", 61.86);

empsExpected[2] = **new** Employee(8352, "Suze", 33.93);

**for**(**int** i=0; i<3; i++) company.addEmployee(empsExpected[i]);

*assertThrows*(IllegalArgumentException.**class**,

 () -> company.getEmployee(99));

1. Note that:

() -> company.getEmployee(99)

is a [lambda expression](https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html) which implements the *Executable* interface. We will not consider these in this class except to note the syntax above: empty parentheses, followed by the “arrow operator”, followed by a line of code. Should you need more than one line of code, surround the lines of code with {}.

1. JUnit API – Comparing ArrayList’s with assertEquals(Object expected, Object actual)
2. You can compare two *ArrayList’s* using *assertEquals* because the *List* interface defines *equals* so that two lists are equal if they have the same elements in the same order.

*Sets* and *Maps* are similar except that the elements don’t have to be in the same order for *HashSet* and *HashMap*.

1. You could also *assertIterableEquals(Iterable<?> expected, Iterable<?> actual)* as *Lists* and *Sets* are *Iterable.*
2. JUnit API – assertAll(Executable… executables)
3. *assertAll* requires that all asserts inside it (specified as lambda expressions) must be true for the test to pass.
4. Example from: <https://stackoverflow.com/questions/40796756/assertall-vs-multiple-assertions-in-junit5>

Assume you have a simple class like an *Address* with fields *city*, *street*, *number* and would like to assert that those are what you expect them to be. You could write the test like this:

Address address = unitUnderTest.methodUnderTest();

assertEquals("Redwood Shores", address.getCity());

assertEquals("Oracle Parkway", address.getStreet());

assertEquals("500", address.getNumber());

Now, as soon as the first assertion fails, you will never see the results of the second, which can be quite annoying. There are many ways around this and JUnit Jupiter's *assertAll* is one of them:

Address address = unitUnderTest.methodUnderTest();

assertAll("Should return address of Oracle's headquarter",

 () -> assertEquals("Redwood Shores", address.getCity()),

 () -> assertEquals("Oracle Parkway", address.getStreet()),

 () -> assertEquals("500", address.getNumber())

);

1. JUnit API – assertTimeout(Duration timeout, Executable executable)
2. Asserts that *excutable* completes execution before *timeout* is exceeded. Example:

Company company = **new** Company();

*assertTimeout*(Duration.*ofSeconds*(3),

 () -> company.longTask());

1. <https://martinfowler.com/bliki/TestDouble.html> [↑](#footnote-ref-1)
2. <http://engineering.pivotal.io/post/the-test-double-rule-of-thumb/> [↑](#footnote-ref-2)
3. <https://en.wikipedia.org/wiki/Test_double> [↑](#footnote-ref-3)