Meme Credit: Reddit user u/sachintripathi007

CS 2112 Lab: JUnit & Debugging

September 15 / 17, 2019

Why Write Tests?

- Ensure code performs as expected
- Prevent introduction and reintroduction of bugs
- Make sure tests are actually used
  - The problem with manual testing is that it takes too long, so is frequently skipped or put off until the end
  - Automated test cases can be run as frequently as needed, ensuring test cases actually provide valuable development feedback

Putting the 'Unit' in Unit Tests

When writing test cases, try to make them as small as possible. If you have e.g. one test that checks three things, consider breaking it into three tests that each check one thing.

This is especially helpful with JUnit because JUnit stops test execution at the first failure, so having three checks in one test makes it difficult to figure out if more than one of the checks is failing. Furthermore, test fixtures make it easy to break tests up into smaller pieces.

Black vs. White Box Testing

Black box tests are written based on the specification alone. They can help ensure that the code works correctly from the client perspective.

White (or glass) box tests are written by looking at the implementation and writing tests to target the specific code. They can help find additional edge cases.

New Bugs

When a new bug is discovered, it's very helpful to write a test case covering that bug. Not only does this provide an easy way to check that the bug has been fixed, it also ensures that if the bug is ever reintroduced into the program, you'll know exactly where and when.

Keep Track of Control Flow

With that said, it's also important to not write superfluous test cases. If a method has a check that a variable `x` is not `null`, it's pointless to add in test cases for `x` being `null` between that check and the next time `x` gets assigned to.

Keeping track of control flow can give you an idea of how many test cases a method should have. In general, one test case per possible control flow is sufficient, although it can be tricky to count all of the possible paths of execution.

Corner Cases

Make sure to cover all corner cases with your tests. When writing tests for data structures, always test with structures that have one or zero elements in them. When writing numerical tests, always test 0 and 1, and so on.

With Java, it's also a good idea to add tests for `null` objects. Without proper testing, a `NullPointerException` might not be noticed until enough code has been written to make it difficult to track down where in the program the `nulls` are coming from.

Eclipse: Setting Up JUnit

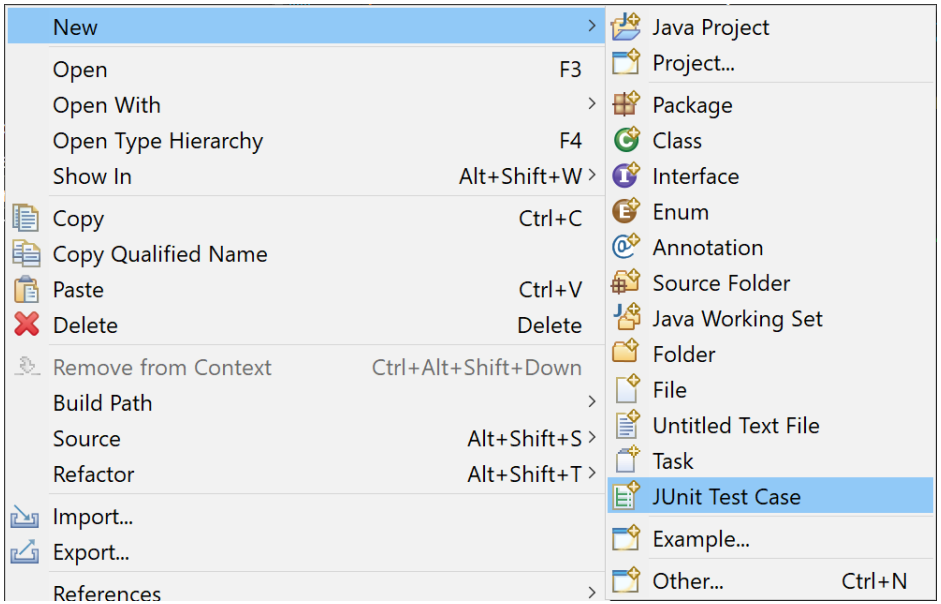
Setting up JUnit with Eclipse is fairly simple. Eclipse should have come with JUnit support, so all that you need to do is include the appropriate library.

To do this, right-click on your project and go to **Build Path** → **Add Libraries...**. Select JUnit 5. Once you have done this, Eclipse should be able to run any Java class that is properly set up as a JUnit test class. You may need to right-click on the file and select **Run As** → **JUnit Test**.

The JUnit website is: `http://junit.org/`

Creating a Test Class

You can create a JUnit test for a specific class by right-clicking a class in the package explorer, and then choosing **New** → **JUnit Test Case**.



Imported Packages

Your JUnit test classes should have the following import declarations:

```
1 import static org.junit.jupiter.api.Assertions.*;
2 import org.junit.jupiter.api.Test;
```

The keyword `static` makes it so that instead of having to write `org.junit.jupiter.Assertions.assertTrue(...)`, you can just write `assertTrue(...)`.

If you are creating a test suite, the imports that are needed are different (see slides for test suites).

Basic Test Case

```
1 @Test
2 public void basicTest() {
3     assertTrue(true, "true is true");
4 }
```

Any method that is preceded by `@Test`, is `public`, returns `void` and has no arguments will be run as a test case. The actual testing in the test case is done using assertion statements such as `assertTrue`.

Useful Assertion Statements

- `assertFalse(boolean cond)`
- `assertNotNull(Object o)`
- `assertNull(Object o)`
- `assertTrue(boolean cond)`
- `fail(String msg)`

`fail(String msg)` is useful when you have code that is supposed to be unreachable, e.g. if you expect an exception to be thrown.

Each of these methods can also take a description as the second argument: `assertTrue(boolean cond, String msg)`

For a complete list, go to:

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

Testing Exceptions

```
1  @Test
2  public void exceptionTest() {
3      try {
4          Integer.parseInt("error");
5      } catch (NumberFormatException e) {
6          // Desired exception, so test passes
7      }
8  }
```

If you want to test if a particular method throws an exception, you can use try/catch blocks with the appropriate assert statements to make the test fail if the appropriate exception isn't thrown.

Testing Exceptions

```
1 @Test
2 void exceptionTest() {
3     assertThrows(NumberFormatException.class, () -> {
4         Integer.parseInt("error")
5     });
6 }
```

A cleaner way to test if an exception should be thrown is to use the new assertThrows method in JUnit 5. Now the test case will be considered to have passed if that exception was thrown, and failed if a different exception or no exception was thrown.

Example: BasicTestClass

```
1  import static org.junit.jupiter.api.Assertions.*;
2  import org.junit.jupiter.api.Test;
3
4  public class BasicTestClass {
5      @Test
6      public void basicTest() {
7        assertTrue(true, "true is true");
8      }
9
10     @Test
11     public void exceptionTest() {
12       assertThrows(NumberFormatException.class,
13       () -> Integer.parseInt("error"));
14     }
15 }
```

Ignoring Tests

If you want to temporarily disable a test case (this might come up if you have test cases for parts of your program that aren't fully implemented yet, for instance), you can do so by putting `@Disabled` above `@Test`. When running tests, JUnit will distinguish between tests that pass, tests that fail due to an assertion, tests that fail due to an unexpected and uncaught exception, and tests that were ignored.

Example: IgnoredTestClass

```
1  import static org.junit.jupiter.api.Assertions.*;
2
3  import org.junit.jupiter.api.Disabled;
4  import org.junit.jupiter.api.Test;
5
6  public class IgnoredTestClass {
7      @Test
8      public void basicTest() {
9          assertFalse(false, "false is false");
10     }
11
12     @Disabled
13     @Test
14     public void ignoredTest() {
15         fail("ignore me");
16     }
17 }
```

Test Fixtures

Sometimes you have a specific setup that you want to run several tests on. Rather than copy and paste the code to recreate that setup, you can create a test fixture.

A test fixture is just an ordinary test class, but with some special functions that are called at certain times: before and after each test, and also before and after the entire set of tests. Instead of being labeled with `@Test`, these special functions are labeled with `@BeforeEach`, `@AfterEach`, `@BeforeAll`, and `@AfterAll`, respectively. Class members are used to store environment variables and make them available to the tests.

Test Fixture Special Methods

Just like a normal test method, the `@BeforeEach` and `@AfterEach` methods are expected to be `public`, `void`, and have no arguments. The `@BeforeAll` and `@AfterAll` methods are expected to be `public static void` and have no arguments.

The purpose of the `@BeforeEach` method is to set up the environment for each test case. The purpose of the `@AfterEach` method is to clean up any external resources used by the test, such as deleting temporary files. `@BeforeAll` and `@AfterAll` are used for one-time setup and cleanup for the entire test fixture. If your test fixture does not require some of these methods, you can just leave them out.

Example: BasicTestFixture

```
1  import static org.junit.jupiter.api.Assertions.*;
2  import org.junit.jupiter.api.BeforeEach;
3  import org.junit.jupiter.api.Test;
4
5  public class BasicTestFixture {
6      private int[] x;
7
8      @BeforeEach
9      public void setup () {
10         x = new int[1];
11     }
12
13     @Test
14     public void sumTest () {
15         x[0] = 4;
16         assertEquals(4, x[0]);
17     }
18 }
```

Test Suites

Suppose you have a lot of separate test classes for each piece of your program. How do you run all of them at the same time?

A test suite is just a list of test classes which all get run together. Test suites require different imports from test classes and test fixtures. To write a test suite, start with an empty Java class, and put the following above the class definition:

```
1 @RunWith(Suite.class)
2 @Suite.SuiteClasses({
3     TestClass1.class,
4     TestClass2.class,
5     ...
6 })
```

Example: BasicTestSuite

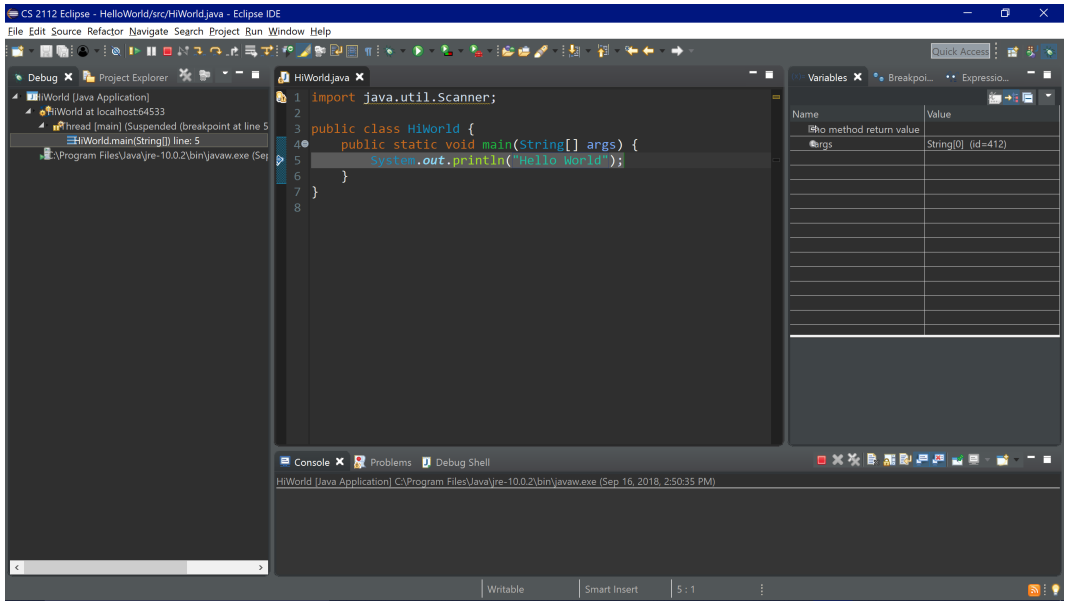
```
1  import org.junit.runner.RunWith;
2  import org.junit.runners.Suite;
3
4  @RunWith(Suite.class)
5  @Suite.SuiteClasses({
6      BasicTestClass.class,
7      IgnoredTestClass.class,
8      BasicTestFixture.class
9  })
10
11 public class BasicTestSuite {}
```

In addition to running test classes and test fixtures, test suites can also run other test suites.

Debug View

You can enter debug view by clicking the bug icon next to Run. Debug view allows you to add breakpoints, step through individual lines of code, inspect variable contents, and test expressions to see their results.

To exit debug view, choose Window → Perspective → Open Perspective → Java



Debug Exercise

Write JUnit tests to ensure your solution maintains the class invariants in Lab03.java. Then use the debugger to fix any bugs you discover.