Unit Testing and Automatic Build Tools

This recitation introduces you to several tools to help you test your applications as you develop and maintain them. You will first use Gradle and Travis CI to build and test your code as you develop it, and then you will write some unit tests.

Build and test automation

In addition to building your project, Gradle can automate the build and execution of your project's unit tests. Specifically, after building your project Gradle runs all test cases in `src/test/main` and reports any test failures.

Follow these steps to use Gradle and Travis CI to build and test your code:

1. Inspect the `build.gradle` file in your `recitation/02` directory.
2. Run `gradle test` from the command line inside your `recitation/02` directory.
3. Edit and commit a minor change to the recitation code and push your repository to GitHub.
4. Log into Travis CI (http://travis-ci.com) and investigate the result of the last build. Travis CI should report that your project passed all test cases.
5. (Optional) Edit a test case so that the test fails and push to GitHub. Again inspect the results of Travis CI.

JUnit overview

JUnit is a simple unit testing framework for Java. In a JUnit test class, the methods are typically annotated to specify their purpose. Frequently used annotations are:

1. `@Test`: marks a public method without parameters as a test case
2. `@Before`: marks a method as a setup method, to be called before every test case.
3. `@After`: marks method as a teardown method, to be called after every test case.
4. `@Ignore`: marks a test case as ignored; such test cases are reported but not executed.
5. `@Test(expected = NullPointerException)`: marks a test case that expects the exception specified after *expected* =. If the expected type of exception arises, the test case passes.

Inside test cases, you can specify expected output using various assertions, such as:

1. `assertEquals(`*`expected, actual`*`)`: requires the value *expected* to be equal to *actual*
2. `assertArrayEquals(`*`expected, actual`*`)`: requires the array *expected* to be equal to *actual*.
3. `assertTrue(`*`cond`*`)`: requires *cond* to have value 'true'
4. `assertNotNull(`*`obj`*`)`: requires *obj* to be non-null

Unit testing and code coverage

In this part of recitation you will write JUnit tests for code we've provided and use a code coverage tool to measure the coverage of your unit tests. Coverage is just the percentage of

code which has been tested; there are several coverage metrics, but one common metric is simply the percentage of lines (or statements) that have been executed by your unit tests.

Two distinct types of unit testing are *black-box testing* and *white-box testing*. Black-box testing is when you write high-level test cases against the specification of a program, without using (or testing) the specific underlying implementation of the program. In white-box testing, however, your tests are tailored to the specific underlying implementation of the code you are testing.

In the exercise below you will write black-box tests for a `LinkedIntQueue` class and white-box tests for an `ArrayIntQueue` class. The `LinkedIntQueue` test is an example of black-box testing because the source code for `LinkedIntQueue` is not provided, and your tests are based on the specified functionality of the `IntQueue` interface rather than the features of the underlying `LinkedIntQueue` implementation. Your `ArrayIntQueue` tests should be white-box tests; you should write test cases that specifically test underlying features of the `ArrayIntQueue` class, not just test against the `IntQueue` specification.

After writing unit tests, use a code coverage tool to measure the coverage of your tests. Jacoco is a coverage plugin you can run with Gradle, and EclEmma is an Eclipse plugin; IntelliJ has a built-in code coverage tool.

For unit testing and measurement of code coverage, you should:

1. Read the Javadoc comments for the `IntQueue` interface, and familiarize yourself with the preconditions and postconditions for each method.
2. Write unit tests for the `LinkedIntQueue` class in the `IntQueueTest` class. Each `IntQueueTest` method should test a specific functionality of the `IntQueue`.
3. Modify `IntQueueTest` to run on the `ArrayIntQueue` class instead of the `LinkedIntQueue` class. There are a few bugs in `ArrayIntQueue`, so at least one test should fail.
4. Fix each bug that was found.
5. Run a coverage tool.
    - Run Jacoco by executing `gradle jacocoTestReport` in terminal and investigating the report in `build/reports/coverage/index.html`. To view the page, right click on `index.html`, go to `Open in Browser` and choose which browser you would like to view the page in.
    - For IntelliJ, right click on test class and select `Run 'classname' with Coverage`; there should be a panel that pops up on the right side of the screen showing class coverage.
    - For Eclipse, go to `Help>Eclipse Marketplace`, search for EclEmma and install it. Restart Eclipse and right click on the test class and go to `Coverage as> JUnit Test`. A panel should pop up and show coverage.
6. Continue writing test cases until your unit test completely covers all statements of the main source code.