IDEs and Unit Testing

This recitation encourages you to get more familiar with your IDE and introduces how to test your Java code with JUnit unit tests.

IDE Tips and Tricks

Modern IDEs are much more powerful than traditional text editors and can help you to significantly reduce the amount of work it takes to write code. Explore the features of your IDE and try to use at least the following features during this recitation:

1. `Quick Fix and Quick Assist`: For most problems underlined with a problem highlight line, the IDE can offer corrections. For example, IDEs can automatically add imports or create variable declarations.
2. `Code Refactoring`: IDEs can rename all instances of a given class or attribute in a project and can automate even more complicated changes.
3. `Navigating Through Code`: IDEs can help you navigate through large projects.

JUnit overview

JUnit is a simple unit testing framework for Java. JUnit will consider every public method annotated with `@Test` as a test case. Further commonly used annotations are:

1. `@Ignore`: marks a test case as ignored; such test cases are reported but not executed.
2. `@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.
3. `@Before`: method will be executed before every test, for example to share common setup code

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

1. `assertEquals(`*`expected, actual`*`)`: requires that the value *expected* is equal to *actual*
2. `assertArrayEquals(`*`expected, actual`*`)`: requires that the array *expected* is equal to the array *actual.*
3. `assertTrue(`*`cond`*`)`: requires that *cond* has value true
4. `assertNotNull(`*`obj`*`)`: requires that *obj* is not a null reference

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 `gradle.build` file in your `recitation/01` directory.
2. Run `gradle test` from the command line inside your `recitation/01` 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.

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.

In the exercise below you will write tests for the `LinkedIntQueue` and `ArrayIntQueue` classes. Note that 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. The `ArrayIntQueue` class is provided and thus you are expected to test based on the source code for the class.

There are many ways to illustrate coverage for your test suite. Calling `gradle cobertura` will produce HTML files with a coverage report in your `build/reports/cobertura` directory. Many IDEs can show coverage directly in the code editor (e.g., Cobertura or EclEmma plugins for Eclipse; builtin code coverage runner in IntelliJ IDEA). You can find information about how to use such tools on Piazza.

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. Fix these bugs!
4. Measure code coverage by executing '`gradle cobertura`' and investigating the report in `build/reports/cobertura/index.html`; or use your IDE's coverage tool.
5. Continue writing test cases until your unit test completely covers all statements of the main source code.