Assignment 3 is posted. Due the Friday after Reading Week. Work on exercise 7 and review lecture examples – Halloween and the OOP version of the Zilch game.

JUnit Testing, Cont.
Testing in General. TDD.
Coverage Testing.
Debugging in Eclipse. *(if we have time).*

JUnit 5 Assertions
Use *(expected, actual)* or *(expected, actual, String)*. The optional argument is a string message that would describe what is being tested.
- `assertArrayEquals()`
- `assertEquals()`
- `assertFalse()`
- `assertNotEquals()`
- `assertNotNull()`
- `assertNull()`
- `assertTrue()`

JUnit 5 Assertions, Cont.
- Use `assertThrows()` to make sure an exception is thrown when it should be.
- This assertion uses two arguments that are built using syntax we have not yet discussed.
- The two are arguments are a Class object and an Executable object. The Class object:
  - Any object has a `.class` constant attribute that supplies the Class object. So we will use: `IllegalHalloweenException.class` to supply the Class object for the first argument, the expected exception type.

JUnit 5 Assertions, Cont.
The second argument is the Executable object.
This will contain the code that is supposed to throw the identified exception.
It is easiest to construct this object using a `Lambda Function`.
*Which we will discuss, just not yet!*

(`) -> new Halloween5(badYear, numKids, temps, condition)
assertThat()

- Takes a Matcher<T> object for its second (or third) parameter.
- Possible object types:
  AllOf, AnyOf, BaseMatcher, CombinableMatcher, CustomMatcher,
  CustomTypeSafeMatcher, DescribedAs, DiagnosingMatcher, Every,
  FeatureMatcher, is, IsAnything, IsCollectionContaining, IsEqual,
  IsInstanceOf, IsNot, IsNull, IsSame, StringContains, StringStartsWith,
  StringEndsWith, SubstringMatcher, TypeSafeDiagnosingMatcher,
  TypeSafeMatcher

assertThat(), Cont.

- For example (from JUnit.org):
  ```
  assertThat("albumen",
              both(containsString("a")).and(containsString("b"))
  ```
- And, there is a:
  ```
  import static org.hamcrest.CoreMatchers.*;
  ```
- See also: https://stackoverflow.com/questions/43280250/how-do-i-use
  hamcrest-with-junit-5-when-junit-5-doesnt-have-an-assertthat-fun

assertThat, Cont.

- Note that the order is different - the actual comes first followed by
  the expected.
- You can still have a String message as the first parameter.
- You can build more sophisticated assertions with this method.

Some JUnit Annotations

- Annotations are needed to help the testing framework understand
  the purpose of the methods in the testing class and how (and how
  often) they should be run:
- @Test is used most often and it identifies a method that is a JUnit
  test. Methods without this annotation will not be considered tests,
  although they can be invoked from methods that are tests.

More JUnit Annotations - Setup and Teardown

- Use @BeforeEach annotation with methods that will run before
every test.
- Similarly @AfterEach for methods to run after every test.
- @BeforeAll runs once before all tests.
- @AfterAll runs once after all tests.

No try/catch

- Don’t bother with try/catch blocks.
- If you are testing code that has a throws decoration, then just have
  your unit test method throw Exception to satisfy the compiler.
- If an exception is thrown unexpectedly (a run-time error!) you will
  get a different kind of failure notice in the report. Note that the
  testing does not stop.
<table>
<thead>
<tr>
<th>Test Suites</th>
<th>Advantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>• You can combine separate JUnit testing classes into a single suite and run them all at once.</td>
<td>• Tests are all separate from the code being harnessed.</td>
</tr>
<tr>
<td>• In Eclipse, go “New”, “Other”, “JUnit”, “JUnit Test Suite”.</td>
<td>• Very easy to add tests.</td>
</tr>
<tr>
<td>• This does not work properly in Eclipse and the JUnit5 instructions are not consistent! But you will not need to build a suite.</td>
<td>• Running tests is very simple and fast.</td>
</tr>
<tr>
<td>• Specify the src folder location for the existing testing *.java files and choose the ones you want to include in the suite.</td>
<td>• Results are nicely summarized and you can zoom in on failed tests easily and get some decent diagnostics.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Designing and Running Unit Tests</th>
<th>Designing and Running Unit Tests, Cont.</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Start by testing the most concrete independent methods first.</td>
<td>• You will need to write stub and driver code as required in the testing classes.</td>
</tr>
<tr>
<td>• Then test the methods that depend on these ones.</td>
<td>• Stub code substitutes something “fake” to a method that depends upon it.</td>
</tr>
<tr>
<td>• Then test classes.</td>
<td>• Driver code simulates code that uses the method being tested.</td>
</tr>
<tr>
<td>• Then test systems.</td>
<td>• Keep the stub and driver code simple so that errors can only come from the harnessed code, not the testing code.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Designing and Running Unit Tests, Cont.</th>
<th>Designing and Running Unit Tests, Cont.</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Start with tests that easily fall into the normal, expected range of inputs. Few or many? Use test generator tools?</td>
<td>• Keep all tests unless you have deleted some of the methods/classes that you were testing. You can even leave these in with a “fail” call.</td>
</tr>
<tr>
<td>• Create more tests for inputs that you suspect are related to each other.</td>
<td>• Fixes applied to some methods can invoke a ripple effect, causing previously passed tests to fail.</td>
</tr>
<tr>
<td>• Choose tests for inputs that are just barely legal.</td>
<td>• In a team effort always run (and pass) all of your unit tests before committing your code to the repository.</td>
</tr>
<tr>
<td>• Choose tests for inputs that are just barely illegal.</td>
<td></td>
</tr>
<tr>
<td>• Choose tests for inputs that are way out there!</td>
<td></td>
</tr>
<tr>
<td>• Make sure you exercise all the code being harnessed.</td>
<td></td>
</tr>
<tr>
<td>• Hundreds or thousands of tests!</td>
<td></td>
</tr>
</tbody>
</table>
Unit Testing

- You can never test all possible input conditions.
- Unit testing doesn’t prove that you code is without errors, but you can end up feeling pretty good about your code if it passes all tests.

Test Driven Development or “TDD”

- How much code can you “implement” without testing it?
- I get nervous writing 20 or 30 lines without testing.
- And whenever I complete a non-trivial method I test it.
- You should not separate the process of implementation and testing.
- Not surprising – this is a basic tenant of Agile Development.

TDD, Cont.

- Is all about writing tests while you are writing, or decomposing your code.
- In fact the testing drives and leads the coding.
- We need a systematic technique to create tests that ensures coverage:

TDD, Cont.

- The three Laws of TDD:
  1. You may not write production code until you have written a failing unit test.
  2. You may not write more of a unit test than what is sufficient to cause a failure, and not compiling is a failure.
  3. You may not write more production code than what is sufficient to pass the current failing test.

TDD, Cont.

- So, tests should be written at the same time as the production code. You are always one test ahead of what the production code can pass.
- As the amount of production code grows so do the number of tests.
- Testing code should be in separate files from production code.
- It really helps to have some kind of framework to help organize and run these tests!

Coverage Testing in Eclipse

- Provides an easy way of seeing which lines of code have been covered by your testing code.
Coverage Testing in Eclipse, Cont.

- Let's run Coverage on Halloween5Test.java to see if we missed anything.
  - Green – covered.
  - Yellow – partially covered. (Conditional branch(s) missed, for example).
  - Red – not covered.
  - We did miss some tests!

Coverage Testing in Eclipse, Cont.

- Getting rid of highlighting:
  - Or just edit the source code file.
  - Uses a module called EclEmma. See:
    http://www.eclemma.org/

Debugging

- Sometimes a mental "walk-through" of your code is not enough to figure out an error.
  - With the debugger you can run to a breakpoint, stop your program and then execute one line at a time while watching the call stack, variables and custom expressions.
  - Let's fix UseDebugger.java

Debugging, Summary

- See "Java development users guide" > “Concepts” > “Debugger” and "Java development user guide" > “Getting Started” > "Basic tutorial" > "Debugging your programs" in the Eclipse help system.
  1. Place one or more breakpoints in your edit window.
  2. Run in debug mode.
  3. Use the "step" choices.
  4. View the method call stack, variables, and use expressions if needed.

Aside – Skipping API Code in Debugger

- Goto Window > Preferences > Java > Debug > Step Filtering.
  - "Select All":
  - Uncheck "Step through filters"
  - Then "Apply and Close".

Skipping JRE Code in Debugger, Cont.

- When in Debug mode:
  - Turn on "Use Step Filters":

Now, you won't see as much API code when you are stepping through your code.
### Upcoming Topics & Terms

- Packages, Modules
- Enumerated Types
- Hierarchies, *inheritance*, Polymorphism
- Interfaces, Anonymous Classes, Abstract Classes, Lambda Functions, Inner Classes
- Generic Classes