CMPE212 – Reminders

• Assignment 3 is posted. Due the Friday after Reading Week. Work on exercise 7 and review lecture examples.

Today

• Continue a look at an OOP version of the Zilch Game.

• JUnit Testing.

An OOP Version of the Zilch Game

• In the procedural solution everything was jammed together in the same class.

• To break up the program, create objects that represent aspects of the game:
  – The AI player.
  – The human player.
  – A scoring class.
  – The game itself.

An OOP Version of the Zilch Game, Cont.

• The objects can contain just the behaviour relevant to what the objects are supposed to do:

• A Player:
  – Knows its name.
  – Takes a turn, which involves:
    • Rolling the dice until > 300.
    • Deciding on whether to bank or roll again.
    • Keeping track of turn score, game score, and zilch count.

An OOP Version of the Zilch Game, Cont.

• The Scoring class is responsible for:
  – Scoring a roll.
  – Providing the roll in a String format.
  – Providing the scoring combination in a String format.
  – Providing the number of dice left to roll.
  – Deciding on whether the player gets a free roll or not.

An OOP Version of the Zilch Game, Cont.

• The Game class:
  – Accepts the Player objects and the game limit value.
  – Displays the game introduction/instructions.
  – Decides which Player goes first (at random).
  – Has the players taking alternate turns until a player goes over the game limit.
  – Provides player game totals each turn.
  – Shows who won the game.
And we should have another small class that just has a main method that will:
- Obtain player names and the game limit from the user.
- Instantiates the players.
- Starts the game.

See the result, which consists of the classes:
- Scoring
- AIPlayer
- HumanPlayer
- ZilchGame
- ZilchException
- Main

Main just has a main method and uses the IOHelper to obtain inputs. If any input is not legal the exception is thrown.

Public methods:
- Scoring:
  - static scoreThrow, accepts an int[], returns int.
  - static Accessors for the number left to roll (an int), the scoring description (a String), and whether a free roll was awarded (a boolean).
- AIPlayer and Player:
  - Constructor (takes a String for the player’s name).
  - takeTurn() (void return).
  - Accessors for the player’s name, the player’s score and roll count.

Public methods, cont.:
- ZilchGame:
  - Constructor accepts two player instances and the game limit (an int).
  - playGame() (void return).
- Main:
  - main()

Why is Scoring all static?
- This class only scores one roll at a time.
- We do not need multiple instances of Scoring – only one.
- However, the class has private static attributes that must be reset when a new dice roll is provided to scoreThrow. This is OK, since this method is the only one that can be invoked to supply a score.

Problems with this structure:
- A lot of duplicate code in HumanPlayer and AIPlayer. Why not have these two classes inherit this common code from a single parent class called Player? We need to know how to build this inheritance structure!
- The console interface is gross. We need to know how to build a GUI version!
Testing

- Until now, “we” have used another class with a main method to run tests on the class we have “harnessed” for testing. See TestHalloween4.java and TestHalloween5.java for example.
- JUnit is a framework designed for this kind of work and it is very easy to use in Eclipse.

JUnit Testing

- Best reference is https://junit.org/junit5/
- For use in Eclipse see: “Java development user guide” > “Getting Started” > “Basic tutorial” > “Writing and running JUnit tests”.

JUnit Testing in Eclipse

- In the project containing the class(es) you wish to test, add a Testing class by choosing “New JUnit Test Case”.
- Name the class and choose the classes to be tested.
- Choose extra stubs, if needed.

JUnit Testing in Eclipse, Cont.

- Next, let the wizard create test method stubs in your testing class for all methods you wish to test:

JUnit Testing in Eclipse, Cont.

- Allow the wizard to add the JUnit library to the Build Path, if prompted.

JUnit Testing in Eclipse, Cont.

- Fill in the method stubs with tests.
  - You can have multiple tests in a method or multiple methods or both, as required. *Diagnosis might be easier if you have one test per method.*
  - You could end up with hundreds of tests, just for one object!
  - See Halloween5Test.java.
  - It contains one test that will not pass – just to show what a failed test looks like.
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)

JUnit 5 Assertions, Cont.

- Uses setup and teardown methods.
- Also uses `assertThat (for a silly test…).`

assertThat()

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

assertThat(), Cont.

- For example (from JUnit.org):
  ```java
  assertThat("albumen",
              both(containsString("a")).and(containsString("b"))
  ```
  - And, there is a:
    ```java
    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.

• beforeEach runs once before all tests.

• afterEach 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!

Test Suites

• You can combine separate JUnit testing classes into a single suite and run them all at once.

• In Eclipse, go “New”, “Other”, “JUnit”, “JUnit Test Suite”.

• This does not work properly in Eclipse and the JUnit5 instructions are not consistent! But you will not need to build a suite.

• Specify the src folder location for the existing testing *.java files and choose the ones you want to include in the suite.

Advantages

• Tests are all separate from the code being harnessed.

• Very easy to add tests.

• Running tests is very simple and fast.

• Results are nicely summarized and you can zoom in on failed tests easily and get some decent diagnostics.
Designing and Running Unit Tests

- Start by testing the most concrete independent methods first.
- Then test the methods that depend on these ones.
- Then test classes.
- Then test systems.

Designing and Running Unit Tests, Cont.

- You will need to write stub and driver code as required in the testing classes.
- Stub code substitutes something “fake” to a method that depends upon it.
- Driver code simulates code that uses the method being tested.
- Keep the stub and driver code simple so that errors can only come from the harnessed code, not the testing code.

Designing and Running Unit Tests, Cont.

- Start with tests that easily fall into the normal, expected range of inputs. Few or many? Use test generator tools?
- Create more tests for inputs that you suspect are related to each other.
- Choose tests for inputs that are just barely legal.
- Choose tests for inputs that are just barely illegal.
- Choose tests for inputs that are way out there!
- Make sure you exercise all the code being harnessed.
- Hundreds or thousands of tests!

Designing and Running Unit Tests, Cont.

- 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.
- Fixes applied to some methods can invoke a ripple effect, causing previously passed tests to fail.
- In a team effort always run (and pass) all of your unit tests before committing your code to the repository.

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!