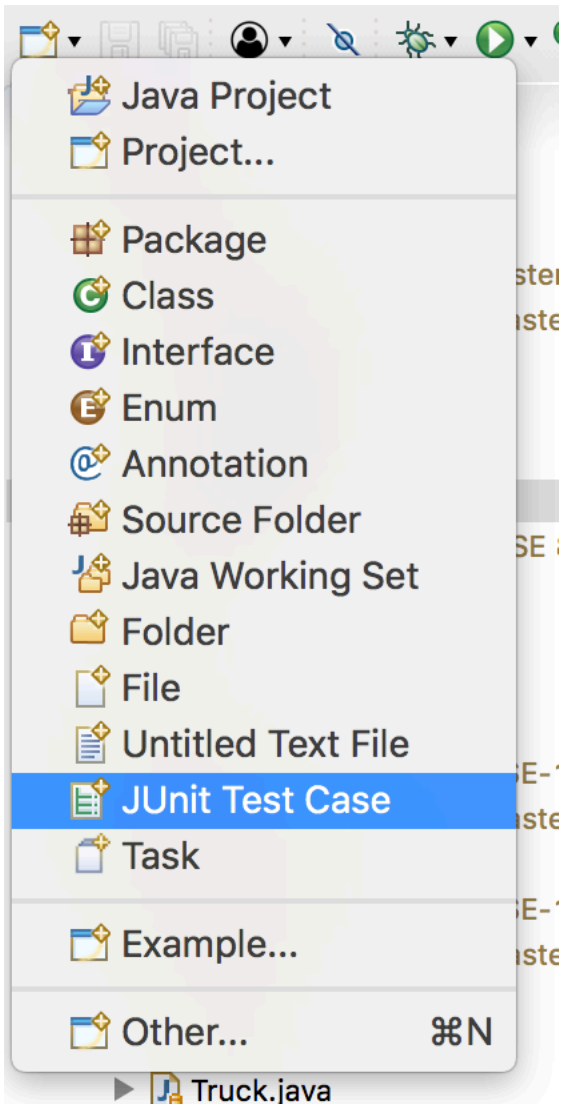
CMSC 206: Data Structures
Lab 3: Unit testing

This lab will cover the concept of *unit testing*, allowing you to test individual parts (say, a class or a method) of a Java project without relying on user interaction or the `main` method. We will be using the JUnit 5 unit testing library, which ships with Eclipse.

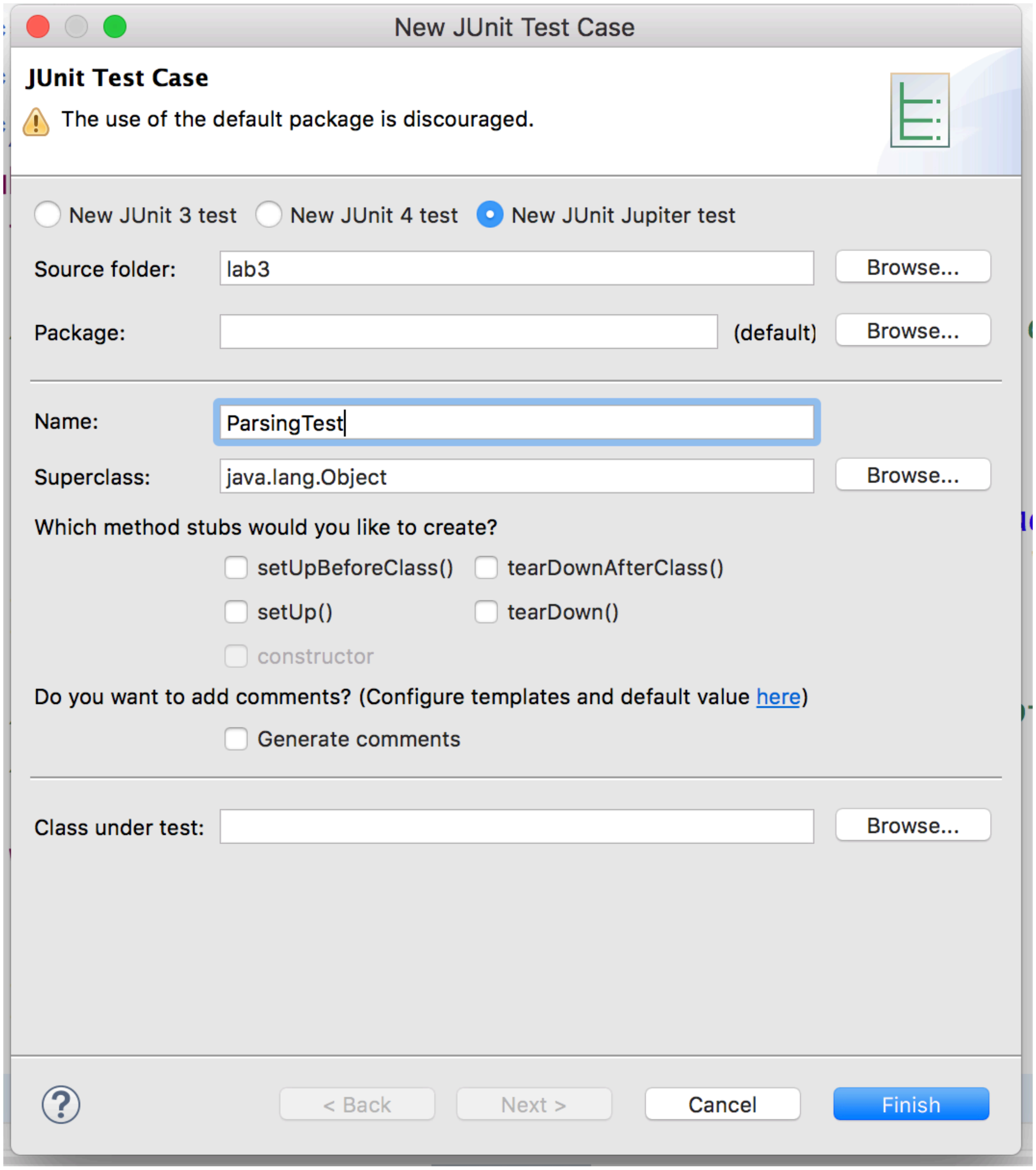
Part I: Testing `parseNumber`

1. Load up your work from the parsing exercise from yesterday's class. Your code should contain a static `parseNumber` method. Make sure this file compiles, but you can continue with this lab even if it doesn't work.

2. In Eclipse, click to make a new JUnit test case:



3. Give a name to your test class in the window that appears, choosing to make a *JUnit Jupiter* test, and click *Finish*. (This "Jupiter" is unrelated to the "Jupyter" used in CS 110.)



New JUnit Test Case

JUnit Test Case

The use of the default package is discouraged.

( ) New JUnit 3 test ( ) New JUnit 4 test [✓ New JUnit Jupiter test]

Source folder: lab3

Package: (default)

Name: ParsingTest

Superclass: java.lang.Object

Which method stubs would you like to create?

- [ ] setUpBeforeClass()
- [ ] tearDownAfterClass()
- [ ] setUp()
- [ ] tearDown()
- [ ] constructor

Do you want to add comments? (Configure templates and default value here)

- [ ] Generate comments

Class under test:

4. Eclipse will ask you about adding JUnit to the build path. Yes, you want to add JUnit to the build path.

5. You will get code that looks like this:

```
import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.*;

class ParsingTest
{

	@Test
	void test()
	{
		fail("Not yet implemented");
	}

}
```

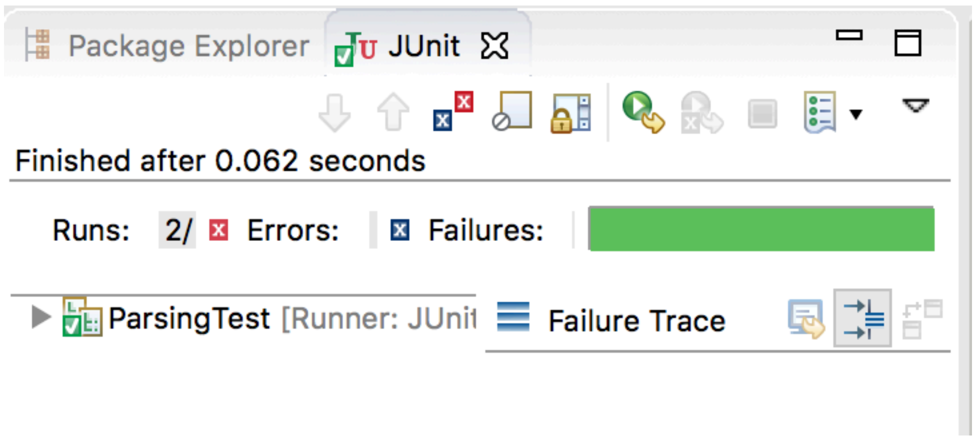
6. A JUnit test class contains many test methods. Each defines a separate test you might want to run against your code. Edit the file to look like this:

```
import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.*;

class ParsingTest
{
	@Test
	void testSmallNumbers()
	{
		assertEquals(123, ParsingExercise.parseNumber("123"));
		assertEquals(45, ParsingExercise.parseNumber("45"));
		assertEquals(2, ParsingExercise.parseNumber("2"));
	}

	@Test
	void testZero()
	{
		assertEquals(0, ParsingExercise.parseNumber("0"));
	}
}
```

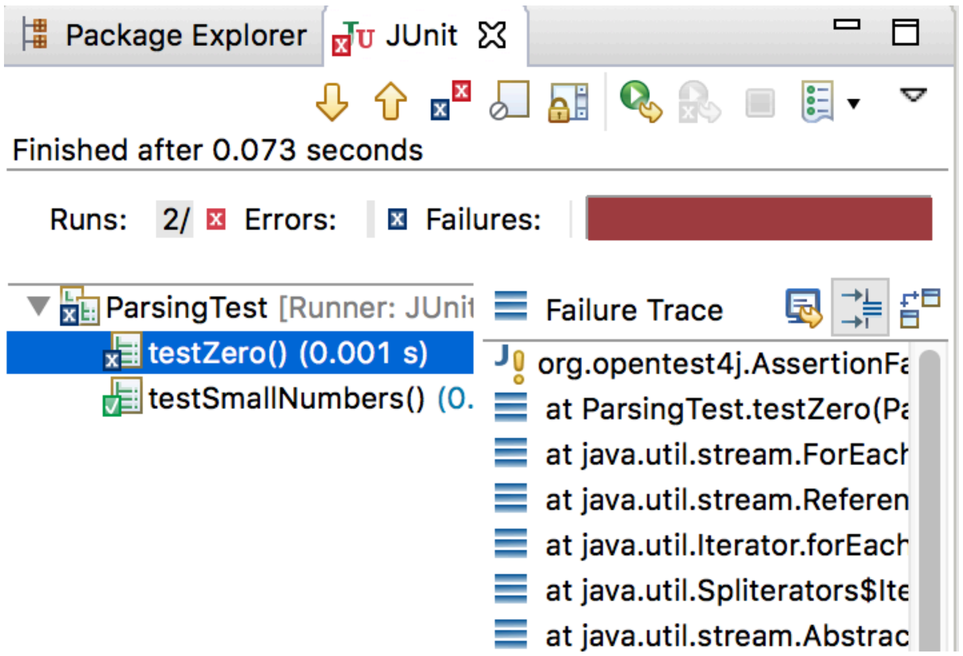
   This file includes tests for small numbers (no comma) and 0. It uses the `assertEquals` method. That method checks whether its two arguments are the same. If not, the method records the test has having failed.

There is a new form of `import` declaration at the top of the file: `import static`. This kind of `import` allows you to access all the `static` methods (and other declarations) in a class unqualified. The `assertEquals` method is actually in the `Assertions` class, and so normally would have to be called by saying `Assertions.assertEquals`. However, because of this `import static`, we can call the `assertEquals` method directly, without writing the `Assertions.` prefix.

7. Run your tests by clicking the usual run button in Eclipse. You will see a *JUnit* view appear, possibly as a tab next to your *Package Explorer* view, to the left of the Eclipse window. This view tells you whether or not your tests pass.



Here, my tests are passing. If I change my function to not handle 0 correctly, this would be my display:



```
org.opentest4j.AssertionFa
at ParsingTest.testZero(Pa
at java.util.stream.ForEach
at java.util.stream.Referen
at java.util.Iterator.forEach
at java.util.Spliterators$Ite
at java.util.stream.Abstrac
```

Note several details:

- In the *JUnit* tab, we see a red "X". Previously we had a green check-mark there.
- The big bar on the right is red instead of green.
- The list of tests is expanded, and we see a small blue "X" next to `testZero`.
- Selecting `testZero` displays the location of the error on the right. You'll see many `java.util.stream` methods -- these are all part of JUnit's framework. Ignore these lines. However, second from the top is `ParsingTest.testZero`, which is the test you wrote. If you click on that line, Eclipse will take you to the line where the assertion failed.

At this point, you should see the result of running these tests against your implementation. If there are any errors, change your code (not the tests!) so that the tests pass.

8. You will see that the tests do not include any tests for input strings with commas. Add a new `@Test` method, following the pattern of the existing ones (the names of the methods do not matter) that tests input strings containing commas. Run the tests and see if your code works correctly. Fix any bugs you find.

9. You have now experienced the typical development cycle of writing unit tests. However, you have seen only `assertEquals`. There are *a lot* of different assertions available to you. Please look at https://junit.org/junit5/docs/current/api/org/junit/jupiter/api/Assertions.html to see the full set of assertions available. In general, you can get the full documentation on JUnit at junit.org.

Part II: A `Fraction` class

For this part of the lab, you will be writing a `Fraction` class that stores a numerical fraction, with a numerator and a denominator. You will also write unit tests against your implementation.

Write a `Fraction` class, with the following constructors and methods. Before diving into writing code, think about what fields your class will need to store to represent a fraction. You will need to have those (private) fields as well as these methods. Additionally, do *not* write the entire `Fraction` class before writing tests. Instead, interleave writing this class with writing tests (in a separate `FractionTest` JUnit test case file). That way, it will be easier to produce correct code.

```
/** Create a fraction representing an integer
 *  @param n The integer to represent
 */
public Fraction(int n)

/** Create a fraction representing the quotient of two
 *  integers
 *  @param n the numerator
 *  @param d the denominator
 */
public Fraction(int n, int d)

/** @return the numerator of this fraction, reduced
 *          so that the numerator and denominator have
 *          no common divisor
 */
public int getNumerator()

/** @return the denominator of this fraction, reduced
 *          so that the numerator and denominator have
 *          no common divisor
 */
public int getDenominator()

/** Adds this fraction with another. This method returns
 *  the sum; it does not modify the current Fraction
 *  object.
 *  @param other The other fraction
 *  @return The sum fraction
 */
public Fraction add(Fraction other)

/** Multiplies this fraction against another. This method
 *  returns the product; it does not modify the current
 *  Fraction object.
 *  @param other The other fraction
 *  @return The product fraction
 */
public Fraction multiply(Fraction other)

/** @return a new fraction that is the negation of the
 *          current one
 */
public Fraction negate()

/** @return a new fraction that is the reciprocal of the
 *          current one
 */
public Fraction reciprocal()

/** @return a string representation of this Fraction
 */
@Override
public String toString()

/** Normalizes the current Fraction, by reducing the
```

```
 *  numerator and denominator until they share no common
 *  factor.
 *  NB: This is a private method, used internally only.
 */
private void normalize()
```

As you are writing your `Fraction` class you may find the following method helpful. It's an implementation of Euclid's algorithm for finding the greatest common divisor of two numbers:

```
public static int gcd(int a, int b)
{
  if(a == 0)
  {
    return b;
  }

  while(b != 0)
  {
    if(a > b)
    {
      a = a - b;
    }
    else
    {
      b = b - a;
    }
  }

  return a;
}
```

When you're done (or even if you're not done by the end of the lab time), please make sure all your lab partners have a copy of your work. We will talk about `Fraction` in class tomorrow.