Unit Testing in Java

Brandon Krakowsky

Penn Engineering

About Unit Testing & Test-Driven Development (TDD)

Unit Testing

- Bottom line, you *must* test your code to get it working

Unit Testing

- Bottom line, you *must* test your code to get it working
- You *can do* ad hoc testing by testing whatever occurs to you at the moment
  - For example:
    - Calling random methods with different inputs from your *main* method and printing/comparing the results
    - Or running your program and trying different inputs for a *Scanner*

Unit Testing

- Bottom line, you *must* test your code to get it working
- You *can do* ad hoc testing by testing whatever occurs to you at the moment
  - For example:
    - Calling random methods with different inputs from your *main* method and printing/comparing the results
    - Or running your program and trying different inputs for a *Scanner*
- Or you *can write* a set of unit tests that can be run at any time
  - This will test your code for you, and always in the same way(s)
  - For reference, this is just like the testing class we write to test a program in Python

What is Unit Testing?

- *Unit testing a*llows you to test individual units of a program
  - The simplest way to think about it is testing individual methods

What is Unit Testing?

- *Unit testing a*llows you to test individual units of a program
  - The simplest way to think about it is testing individual methods
- *Unit testing* does not guarantee that the different methods work together
  - That's the job of *integration testing*, which tests the interaction between the units (methods)

What is Unit Testing?

- *Unit testing a*llows you to test individual units of a program
  - The simplest way to think about it is testing individual methods
- *Unit testing* does not guarantee that the different methods work together
  - That's the job of *integration testing*, which tests the interaction between the units (methods)
- *Unit testing* is part of the *test-driven development (TDD)* approach to software development, where program requirements and features are turned into very specific test cases
  - Code is written (software is developed) to pass those tests

Why Unit Test?

- The *disadvantages* of writing unit tests:
  - It *can* require (a lot of) extra programming
    - But use of a good testing framework can help with the process
  - You don't have time to do all that extra work
    - But testing reduces debugging time more than the amount of time spent building the actual tests

Why Unit Test?

- The *advantages* of writing unit tests:
  - It helps you track down bugs, which is often the most time-consuming part of software development
    - Guaranteed, your program will have fewer bugs
    - And it will save you a lot of time in the long run!
  - When something breaks, it helps you identify exactly which unit of your program (method) is broken
    - If you implement incorrect code, at least one of your unit tests SHOULD fail
  - Overall, it will be a lot easier to maintain and modify your program
    - This is a huge win for programs that get actual use in production!

Why Unit Test?

- A program is written as a collection of classes and methods
  - Designing the individual methods for testability enhances the overall design of the program

Why Unit Test?

- A program is written as a collection of classes and methods
  - Designing the individual methods for testability enhances the overall design of the program
- A method should either do computation or input/output, but not both
  - Following this rule makes testing computational methods easier
  - If a method being tested requests input from the user, the method can't be tested easily (with a "single click")
  - If a method being tested produces output, the output must be examined for correctness

JUnit

- JUnit is a (Java) framework for writing unit tests
  - JUnit uses Java's *reflection* capabilities, which allows Java programs to examine their own code
  - JUnit helps the programmer:
    - Define and execute tests
    - Formalize requirements and clarify program architecture
    - Write and debug code
    - Integrate code and always be ready to release a working version

Terminology

- A unit test tests the units (methods) in a *single* class

Terminology

- A unit test tests the units (methods) in a *single* class
- A test case tests the response of a *single* unit (method) to a particular set of inputs
  - You can (and should) have multiple test cases for a single unit test method

Terminology

- A unit test tests the units (methods) in a *single* class
- A test case tests the response of a *single* unit (method) to a particular set of inputs
  - You can (and should) have multiple test cases for a single unit test method
- An integration test is a test of how well classes and methods work together
  - Integration testing (testing that it all works together) is not well supported by Junit and we won't cover this

How To Do Unit Testing

- Create a testing class for writing and running unit tests
  - Import the JUnit testing framework
  - Write test methods to test the individual methods in the program class

How To Do Unit Testing

- Create a testing class for writing and running unit tests
  - Import the JUnit testing framework
  - Write test methods to test the individual methods in the program class
- The unit testing process for an individual method:
  - Call a method being tested in your program and get the actual result
  - “Assert” what the correct result should be with one of the assert methods
  - Repeat steps as many times as necessary

How To Do Unit Testing

- Create a testing class for writing and running unit tests
  - Import the JUnit testing framework
  - Write test methods to test the individual methods in the program class
- The unit testing process for an individual method:
  - Call a method being tested in your program and get the actual result
  - "Assert" what the correct result should be with one of the assert methods
  - Repeat steps as many times as necessary
- An assert method is a JUnit method that performs a test, and throws an AssertionError if the test fails
  - JUnit catches these Errors and shows you the result

Assert Methods

- Some assert methods:

```
void assertTrue(boolean test)
void assertTrue(boolean test, String message)
```

  - Throws an *AssertionError* if the test fails
  - The optional *message* is included in the Error

```
void assertFalse(boolean test)
void assertFalse(boolean test, String message)
```

  - Throws an *AssertionError* if the test fails
  - The optional *message* is included in the Error

Unit Testing Example – SimpleMath Class

- As an example, let's look at a trivial *SimpleMath* class
  - The *isPositive* method will return a boolean value indicating if the given number is positive (or not)

Unit Testing Example – SimpleMath Class

- As an example, let's look at a trivial *SimpleMath* class
  - The *isPositive* method will return a boolean value indicating if the given number is positive (or not)
- A good approach is to write the program method stubs first, and let Eclipse generate the test method stubs

Unit Testing Example – SimpleMath Class

- As an example, let's look at a trivial *SimpleMath* class
  - The *isPositive* method will return a boolean value indicating if the given number is positive (or not)
- A good approach is to write the program method stubs first, and let Eclipse generate the test method stubs
- Don't be alarmed if, in this simple example, the JUnit tests are more code than the class itself

Unit Testing Example – SimpleMath Class

- As an example, let's look at a trivial *SimpleMath* class
  - The *isPositive* method will return a boolean value indicating if the given number is positive (or not)

SimpleMath.java

```
1  /**
2   * Contains a very simple math method.
3   * @author lbrandon
4   *
5   */
6  public class SimpleMath {
7
8      /**
9       * Returns a boolean value indicating
10      * if the given number x is positive (or not).
11      * @param x to analyze
12      * @return true if x is positive, otherwise false
13      */
14     boolean isPositive(double x) {
15         return x > 0;
16     }
17 }
18
```

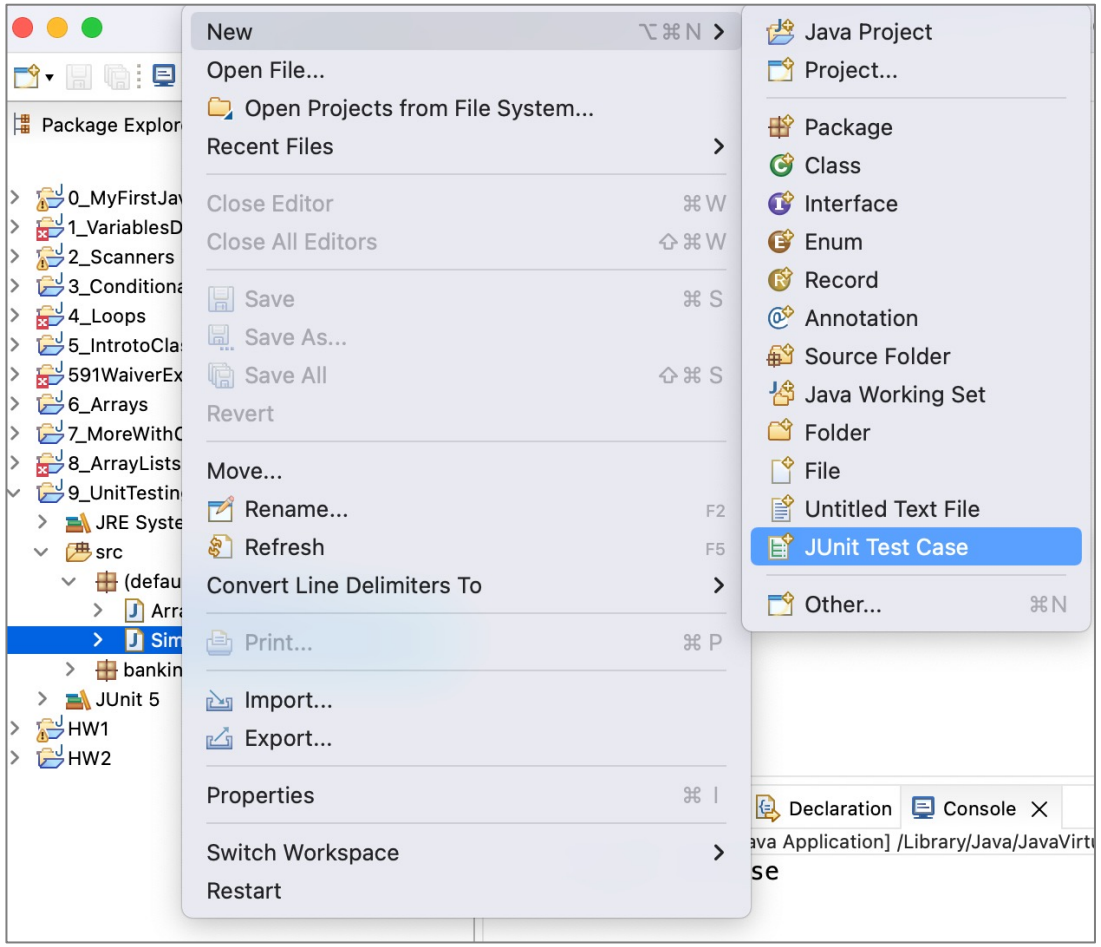
Unit Testing Example – SimpleMath Class

- As an example, let's look at a trivial *SimpleMath* class
  - If you like, add a *main* method and call the *isPositive* method

```
17
18      public static void main(String[] args) {
19
20          //create instance of SimpleMath and call isPositive method
21          SimpleMath sm = new SimpleMath();
22          boolean positive = sm.isPositive(0);
23
24          System.out.println("0 is positive: " + positive);
25      }
26  }
27
```

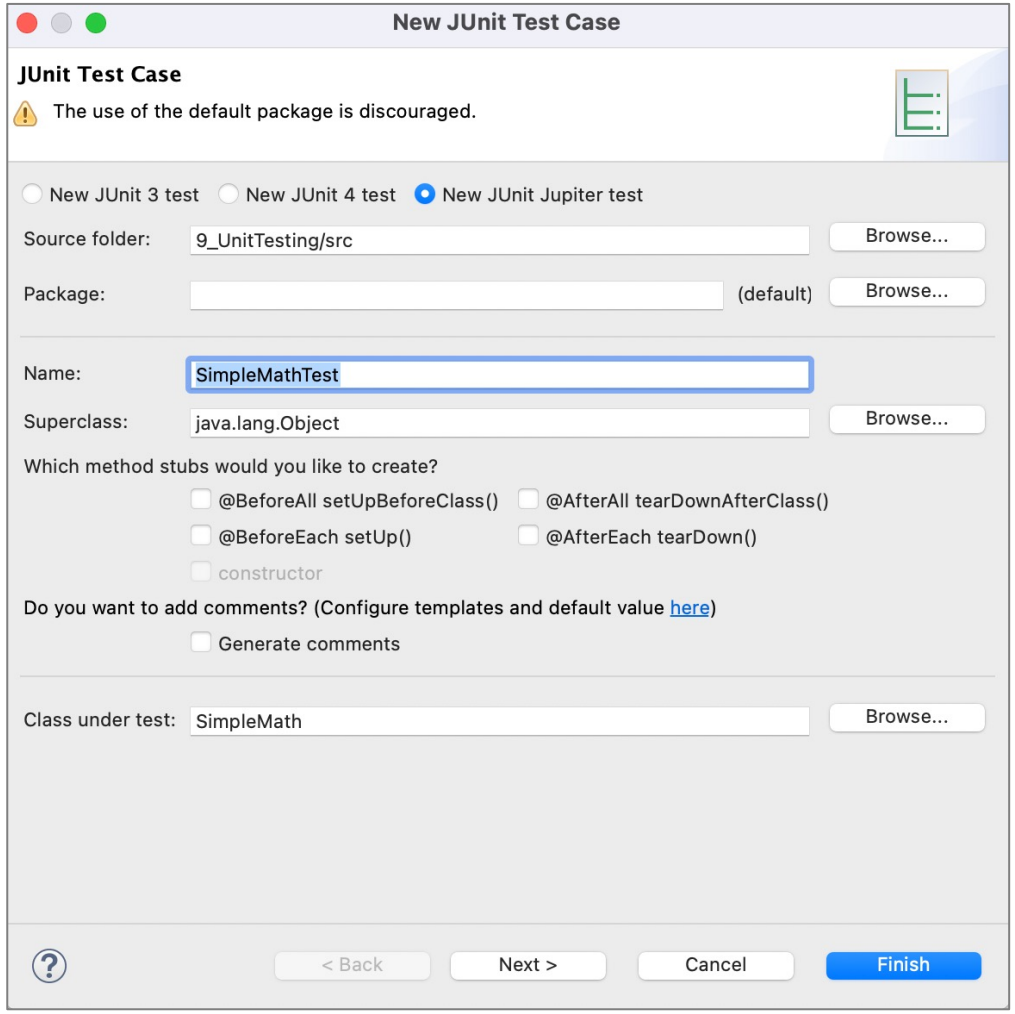
Unit Testing Example - Create JUnit Tests

- To create JUnit Tests, select the class file in the Package Explorer, and go to “New” → “JUnit Test Case”



Unit Testing Example - Create JUnit Tests

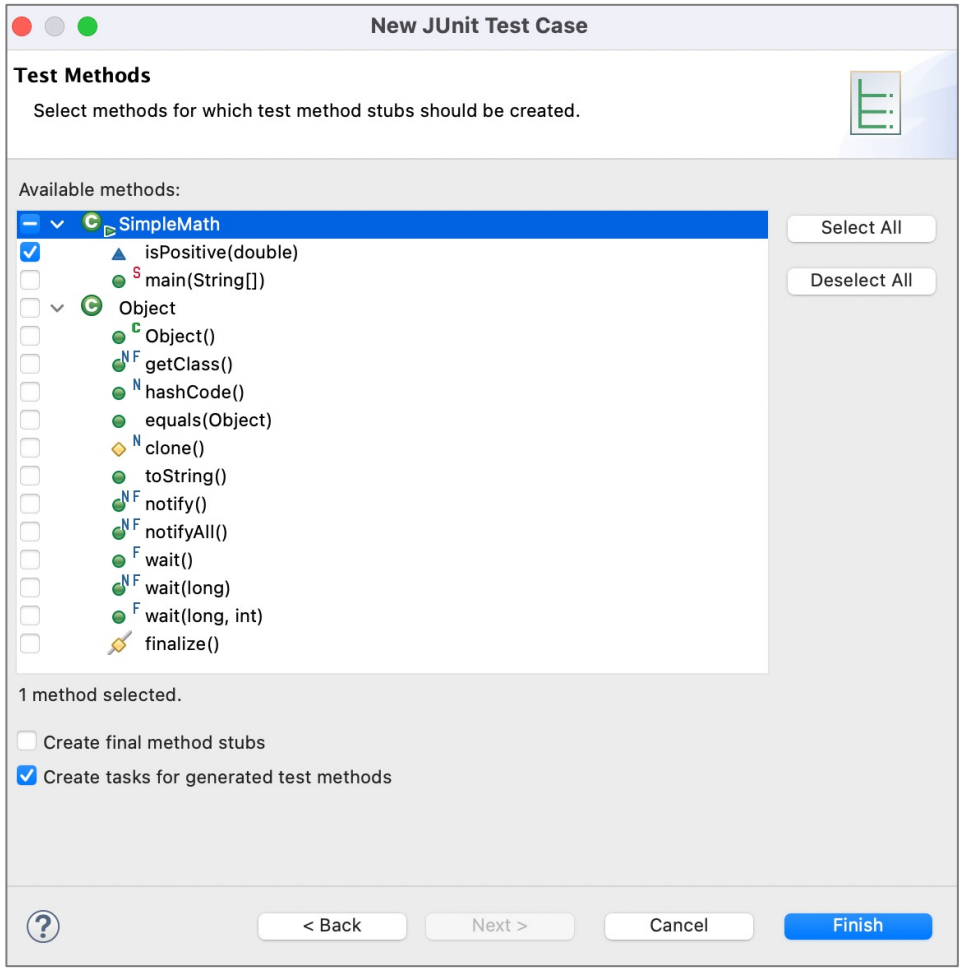
- Use the default name provided for your JUnit Test Case class



Make sure @New Junit Jupiter test is selected

Unit Testing Example - Create JUnit Tests

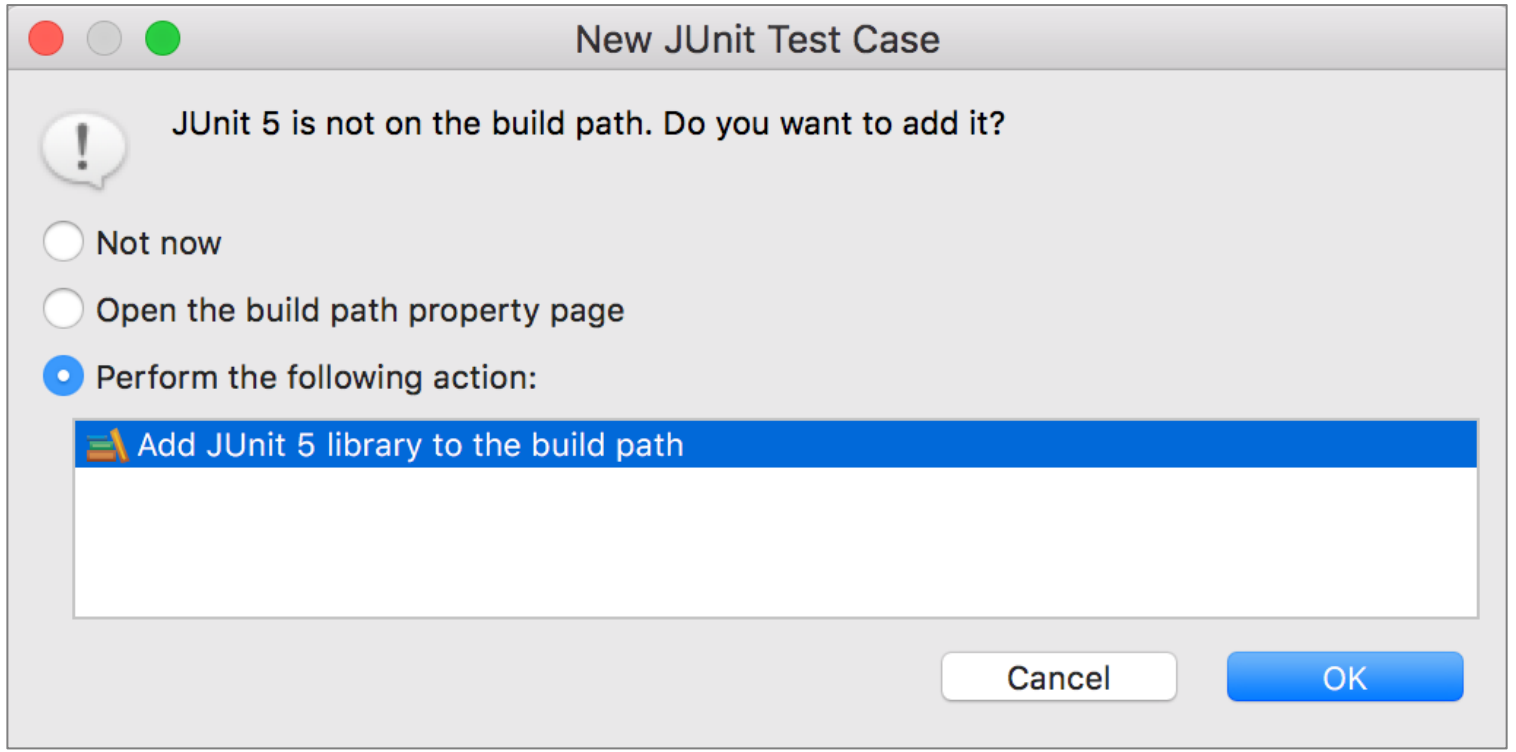
- To have Eclipse generate test method stubs for you, use the checkboxes to decide which methods you want test cases for. Don't select Object or anything under it.



Check Create tasks for generated test methods

Unit Testing Example - Create JUnit Tests

- Add the JUnit 5 library to the project build path
  - This includes the necessary JUnit framework in your project



Unit Testing Example - Create JUnit Tests

- Eclipse will add a new JUnit Test class in the same package (or default)
  - You'll see test method stubs to be implemented
  - The code in each test method is calling *fail* (with a message), to force the test methods to initially fail

SimpleMath.java | SimpleMathTest.java

```
1  import static org.junit.jupiter.api.Assertions.*;
4
5  class SimpleMathTest {
6
7      @Test
8      void testIsPositive() {
9          fail("Not yet implemented"); // TODO
10     }
11
12 }
13
```

Unit Testing Example - Create JUnit Tests

- Eclipse will add a new JUnit Test class in the same package (or default)
  - You'll see test method stubs to be implemented
  - The code in each test method is calling *fail* (with a message), to force the test methods to initially fail

SimpleMath.java | SimpleMathTest.java

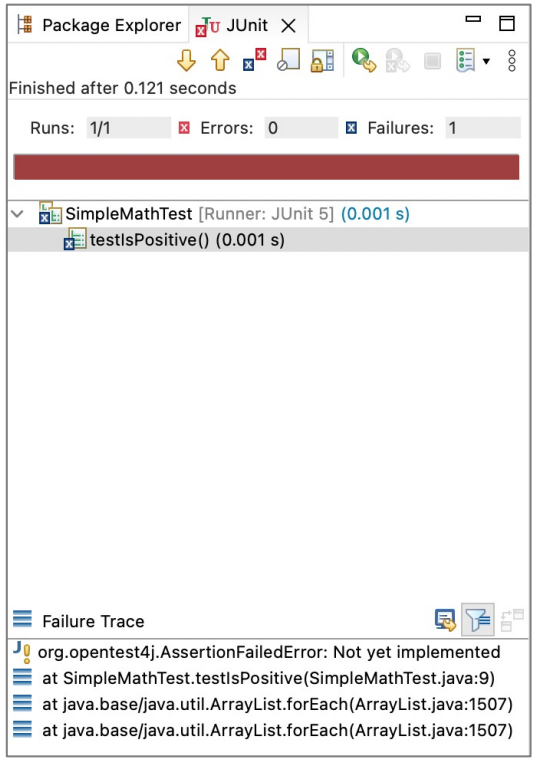
```
1  import static org.junit.jupiter.api.Assertions.*;
4
5  class SimpleMathTest {
6
7      @Test
8      void testIsPositive() {
9          fail("Not yet implemented"); // TODO
10     }
11
12 }
13
```

Each test method is annotated by @Test

Each test method typically starts with the letters 'test'

Unit Testing Example - Create JUnit Tests

- If you run the tests, they should ALL fail
  - Eclipse will open the JUnit panel (on the left)
    - The top bar will show red
    - The number next to “Failures” will show 1 (in this case)
    - The message at the bottom will explain why the tests failed



SimpleMath.java | SimpleMathTest.java

```
1  import static org.junit.jupiter.api.Assertions.*;
4
5  class SimpleMathTest {
6
7      @Test
8      void testIsPositive() {
9          fail("Not yet implemented"); // TODO
10     }
11
12 }
13
```

Unit Testing Example - Create JUnit Tests

- Add test cases with assert methods in the test method

```
@Test
void testIsPositive() {

    //create instance of SimpleMath for testing
    SimpleMath sm = new SimpleMath();

    //tests that the method returns true value
    assertTrue(sm.isPositive(2));

    //tests that the method returns false value
    assertFalse(sm.isPositive(0));

}
```

Unit Testing Example - Create JUnit Tests

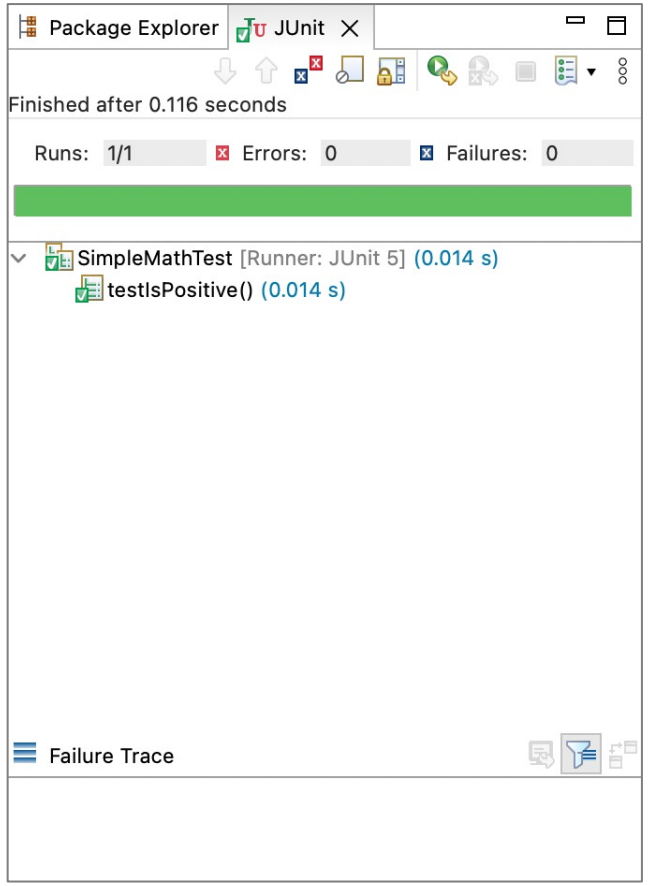
- Other assert methods you could use to test *isPositive*

```
    //tests that 2 values are equal
    assertEquals(false, sm.isPositive(-1));

    //tests that 2 values are equal, with custom error message
    assertEquals(false, sm.isPositive(0.0), "0.0 is actually not positive!");
}
```

Unit Testing Example - Create JUnit Tests

- If you run the tests, they should ALL pass
  - In the JUnit panel (on the left) – the top bar will show green
  - The number next to “Failures” will show 0




```java
@Test
void testIsPositive() {

    //create instance of SimpleMath for testing
    SimpleMath sm = new SimpleMath();

    //tests that the method returns true value
    assertTrue(sm.isPositive(2));

    //tests that the method returns false value
    assertFalse(sm.isPositive(0));

    //tests that 2 values are equal
    assertEquals(false, sm.isPositive(-1));

    //tests that 2 values are equal, with custom error message
    assertEquals(false, sm.isPositive(0.0), "0.0 is actually not positive!");
}
```

More Assert Methods

```
void assertEquals(expected, actual)
void assertEquals(expected, actual, String message)
```

- *expected* and *actual* must both be Objects *or* the same primitive type
- For primitives, this method compares using ==
- For Objects, this method compares using the *equals* method
  - For your own objects, you'll need to define the *equals* method properly (as described in *"About_Equality"* lecture)

Assert Methods with Arrays

```
void assertArrayEquals(int[] expected, int[] actual)
void assertArrayEquals(int[] expected, int[] actual, String message)
```

- Asserts that two int arrays are equal

Assert Methods with Floating Points Types

- Note: When you want to compare floating point types (e.g. double or float) with a high amount of precision
  - You should use *assertEquals* with the additional parameter *delta* to avoid problems with round-off errors while doing floating point comparisons

- The assert method syntax to use is:

```
void assertEquals(double expected, double actual, double delta)
```

  - This asserts that the *expected* and *actual* are equal, within the given *delta*
  - *delta* is typically a very small double (e.g. 0.000001) used for comparison

Assert Methods with Floating Points Types

- Note: When you want to compare floating point types (e.g. double or float) with a high amount of precision
  - You should use *assertEquals* with the additional parameter *delta* to avoid problems with round-off errors while doing floating point comparisons

- The assert method syntax to use is:

```
void assertEquals(double expected, double actual, double delta)
```

  - This asserts that the *expected* and *actual* are equal, within the given *delta*
  - *delta* is typically a very small double (e.g. 0.000001) used for comparison

- For example:

```
void assertEquals(aDoubleValue, anotherDoubleValue, 0.000001)
```

  - This evaluates to: Math.abs(aDoubleValue – anotherDoubleValue) <= delta

More Assert Methods

```
void assertSame(Object expected, Object actual)
void assertSame(Object expected, Object actual, String message)
```

- Asserts that two arguments refer to the *same* object

```
void assertNotSame(Object expected, Object actual)
void assertNotSame(Object expected, Object actual, String message)
```

- Asserts that two objects do not refer to the same object

More Assert Methods

```
void assertNull(Object object)
void assertNull(Object object, String message)
```

- Asserts that the object is null (undefined)

```
void assertNotNull(Object object)
void assertNotNull(Object object, String message)
```

- Asserts that the object is not null

```
fail()
fail(String message)
```

- Causes the test to fail and throw an *AssertionFailedError*

A Good Test-Driven Development (TDD) Strategy

- Pick a method that doesn't depend on other, untested methods

A Good Test-Driven Development (TDD) Strategy

- Pick a method that doesn't depend on other, untested methods
- While the method isn't complete:
  - Write a new test for a feature (e.g. new/improved functionality)
  - Run all tests and make sure the new one fails
  - While any test fails:
    - Add/fix just enough code in the method to make it pass the test
  - *Refactor* the code to make it cleaner

A Good Test-Driven Development (TDD) Strategy

- Pick a method that doesn't depend on other, untested methods
- While the method isn't complete:
  - Write a new test for a feature (e.g. new/improved functionality)
  - Run all tests and make sure the new one fails
  - While any test fails:
    - Add/fix just enough code in the method to make it pass the test
  - *Refactor* the code to make it cleaner
- *Refactoring* code is the process of restructuring the code (changing the factoring) without changing the behavior

A Good Test-Driven Development (TDD) Strategy

- Test typical examples
  - Use expected inputs and arguments to test your methods
  - You should expect these tests to pass easily

A Good Test-Driven Development (TDD) Strategy

- Test typical examples
  - Use expected inputs and arguments to test your methods
  - You should expect these tests to pass easily
- Test edge-cases
  - Use more unexpected inputs and method arguments
  - These are typically examples at the extreme ends of the input spectrum
    - Largest/smallest possible inputs
    - Negative numbers
    - Zero
    - Incorrect data types
    - Empty arrays
    - etc.

Testing Examples

Array Testing

- Create a new *ArrayExamples* class. Write a *sum13* method that returns the sum of a given array of numbers (ints)
  - The number 13 is very unlucky, so it doesn't count
  - Also, the numbers that come immediately after the number 13 don't count
  - Return 0 for an empty array

Array Testing

- Create a new *ArrayExamples* class. Write a *sum13* method that returns the sum of a given array of numbers (ints)
  - The number 13 is very unlucky, so it doesn't count
  - Also, the numbers that come immediately after the number 13 don't count
  - Return 0 for an empty array
- For this example, you're going to write the tests first, then implement the method to pass the tests
  - To do this, first create the *sum13* method "stub" and return 0
  - The idea here is to start with an empty method so our tests fail

Array Testing

- Create a new *ArrayExamples* class. Write a *sum13* method that returns the sum of a given array of numbers (ints)

ArrayExamples.java

```java
/**
 * Examples of methods that deal with arrays.
 * @author lbrandon
 */
public class ArrayExamples {

    /**
     * Returns the sum of a given array of numbers.
     * The number 13 is very unlucky, so it doesn't count.
     * Also, the numbers that come immediately after the number 13 don't count.
     * @param arr of numbers
     * @return sum of numbers
     */
    int sum13(int[] arr) {
        return 0;
    }

}

```

Array Testing

- Create a new *ArrayExamplesTest* class and a test method

ArrayExamples.java | ArrayExamplesTest.java

```
1  import static org.junit.jupiter.api.Assertions.*;
5
6  class ArrayExamplesTest {
7
8      @Test
9      void testSum13() {
10         fail("Not yet implemented");
11     }
12
13 }
14
```

Array Testing

- Create your tests in the test method
  - All tests should fail

```
 8    @Test
 9    void testSum13() {
10
11        ArrayExamples ae = new ArrayExamples();
12
13        int[] arr1 = {1, 2, 1};
14        assertEquals(4, ae.sum13(arr1));
15
16        int[] arr2 = {1, 2, 13};
17        assertEquals(3, ae.sum13(arr2));
18
19        int[] arr3 = {1, 2, 13, 1};
20        assertEquals(3, ae.sum13(arr3));
21
22        int[] arr4 = {1, -2, 13, 1};
23        assertEquals(-1, ae.sum13(arr4));
24
25        int[] arr5 = {};
26        assertEquals(0, ae.sum13(arr5));
27
28        int[] arr6 = {13};
29        assertEquals(0, ae.sum13(arr6));
30    }
```

Array Testing

- Implement the *sum13* method to pass the tests

```
14      int sum13(int[] arr) {
15          if (arr.length == 0) {
16              return 0;
17          }
18
19          int arrSum = 0;
20
21          //iterate over each value
22          for (int i : arr) {
23              //if it's 13, we're done, break out of loop
24              if (i == 13) {
25                  break;
26              }
27
28              arrSum += i;
29          }
30
31          return arrSum;
32      }
```

Array Testing

- Add another method *has22* that returns true if a 2 appears next to another 2 in the given array
- For now, just create the method stub and return false

```
34    /**
35     * Returns true if a 2 appears next to another 2 in the given array.
36     *
37     * For example:
38     * has22(new int[] {1, 2, 2}) returns true
39     * has22(new int[] {1, 2, 1, 2}) returns false
40     * has22(new int[] {2, 1, 2}) returns false
41     *
42     * @param arr of numbers
43     * @return true if a 2 appears next to another 2
44     */
45    boolean has22(int[] arr) {
46        return false;
47    }
```

Array Testing

- Create your tests in the test method
  - All tests should fail

```
@Test
void testHas22() {

    ArrayExamples ae = new ArrayExamples();

    assertEquals(true, ae.has22(new int[] {1, 2, 2}));
    assertEquals(false, ae.has22(new int[] {2, 1, 2}));
    assertEquals(false, ae.has22(new int[] {1, 2, 1, 2}));

    assertEquals(false, ae.has22(new int[] {2}));
    assertEquals(false, ae.has22(new int[] {}));
}
```

Array Testing

- Implement the *has22* method to pass the tests

```
45      boolean has22(int[] arr) {
46
47          for (int i = 0; i < arr.length - 1; i++) {
48              //i is the index of each element
49              if (arr[i] == 2 && arr[i + 1] == 2) {
50                  return true;
51              }
52          }
53
54          return false;
55      }
```

Array Testing

- Add another method *swapFirstLast* method that swaps the first and last elements of an array
- For now, just create the method stub

```
57    /**
58     * Swaps the first and last elements in the given array.
59     * @param arr to swap elements
60     */
61    void swapFirstLast(int[] arr) {
62
63    }
64
```

Array Testing

- Create your tests in the test method
  - All tests should fail

```
45      @Test
46      void testSwapFirstLast() {
47
48          ArrayExamples ae = new ArrayExamples();
49
50          //create data and apply method
51          int[] myArr = {4, 5, 7, -45};
52          ae.swapFirstLast(myArr);
53
54          assertTrue(myArr[0] == -45);
55          assertTrue(myArr[3] == 4);
56
57          //confirms length of array
58          assertEquals(4, myArr.length);
59
60          //compares values (and order) of arrays
61          assertArrayEquals(new int[] {-45, 5, 7, 4}, myArr);
62
63      }
64
```

Array Testing

- Implement the *swapFirstLast* method to pass the tests

```
56
57      /**
58       * Swaps the first and last elements in the given array.
59       * @param arr to swap elements
60       */
61      void swapFirstLast(int[] arr) {
62          int firstElement = arr[0];
63
64          arr[0] = arr[arr.length - 1];
65          arr[arr.length - 1] = firstElement;
66      }
67
```

Array Testing

- Add another method *removeMin* that removes the min element of an array
- It does this by returning a new array with the min element removed
- For now, just create the method stub and return an empty array

```
68      /**
69       * Removes the min element of the given array
70       * by returning a new array with the min element removed.
71       * @param arr to remove min from
72       * @return new array with min element removed
73       */
74      double[] removeMin(double[] arr) {
75
76          double[] newArr = new double[0];
77
78          return newArr;
79      }
```

Array Testing

- Create your tests in the test method
  - All tests should fail

```
65      @Test
66      void testRemoveMin() {
67
68          ArrayExamples ae = new ArrayExamples();
69
70          //Create data and apply method
71          double[] myArr = {1.0, 4.0, -1.3, 5.7, 7.7};
72          double[] myNewArr = ae.removeMin(myArr);
73
74          //confirms length of new array
75          assertTrue(myNewArr.length == 4);
76
77          //compares values (and order) of arrays
78          assertArrayEquals(new double[] {1.0, 4.0, 5.7, 7.7}, myNewArr);
79      }
```

Array Testing

- First implement a *findMin* method to find the min element of an array

```
100     /**
101      * Finds and returns the min element of the given array.
102      * @param arr to find min
103      * @return min
104      */
105     double findMin (double[] arr) {
106
107         //set min to first double in array
108         double minNumber = arr[0];
109
110         //iterate over array
111         for (int i = 0; i < arr.length; i++) {
112
113             //if double is less than current double
114             if (arr[i] < minNumber) {
115
116                 //reset min
117                 minNumber = arr[i];
118             }
119         }
120
121         return minNumber;
122     }
123
```

Array Testing

- Then implement the *removeMin* method to pass the tests
  - You can use the *findMin* method to first find the min value

```
68    /**
69     * Removes the min element of the given array
70     * by returning a new array with the min element removed.
71     * @param arr to remove min from
72     * @return new array with min element removed
73     */
74    double[] removeMin(double[] arr) {
75
76        //find min in array by calling findMin method
77        double minNumber = this.findMin(arr);
78
79        //create new array
80        double[] newArr = new double[arr.length - 1];
81
82        //create index
83        int newArrIndex = 0;
84
85        //iterate over given array
86        //add each element (except min) to new array
87        for (int i = 0; i < arr.length; i++) {
88
89            //if number is not the min
90            if (arr[i] != minNumber) {
91                newArr[newArrIndex] = arr[i];
92                newArrIndex++;  //increment index
93            }
94        }
95
96        //return new array
97        return newArr;
98    }
99
```

Sentence Counter Project

Sentence Counter

- Create a new *SentenceCounter* program. Write a simple *countSentences* method that counts the number of sentences in a string of text, based on a period (.)
- For now, just create the method stub and return 0
    - Again, the idea here is to start with an empty method so our tests fail

Sentence Counter

- Create a new *SentenceCounter* program. Write a simple *countSentences* method that counts the number of sentences in a string of text, based on a period (.)
- For now, just create the method stub and return 0
  - Again, the idea here is to start with an empty method so our tests fail

SentenceCounter.java

```
1  package sentences;
2
3  public class SentenceCounter {
4
5      /**
6       * Counts the number of sentences in the given text.
7       * @param text to extract number of sentences
8       * @return count of sentences
9       */
10     int countSentences(String text) {
11         return 0;
12     }
13 }
14
```

Sentence Counter Testing

- Create a new testing file and create your tests.
  - All tests should fail

```
 9    @Test
10    void testCountSentences() {
11
12        SentenceCounter sc = new SentenceCounter();
13
14        assertEquals(2, sc.countSentences("Today is Monday. This is it."));
15        assertEquals(1, sc.countSentences("."));
16        assertEquals(2, sc.countSentences(".."));
17        assertEquals(2, sc.countSentences("Hi there. My name is Brandon. What's yours?"));
18        assertEquals(0, sc.countSentences(""));
19
20    }
21
```

Sentence Counter Testing

- Implement the *countSentences* method to pass the tests
  - It can count the number of sentences, based on a period (.)

```
 7    /**
 8     * Counts the number of sentences in the given text.
 9     * @param text to extract number of sentences
10     * @return count of sentences
11     */
12    int countSentences(String text) {
13
14        //strip whitespace from beginning and end of string
15        text = text.strip();
16
17        int puncCount = 0;
18        //iterate over string and find instances of period (.)
19        for (int i = 0; i < text.length(); i++) {
20            if (text.charAt(i) == '.') {
21                puncCount++;
22            }
23        }
24
25        int sentenceCount = puncCount;
26        return sentenceCount;
27    }
```

Sentence Counter Testing

- All tests should pass, including the test with a question
  - The last sentence in the text is followed by a question mark (?), so we would only expect to get a sentence count of 2, based on a period (.)

```
 9    @Test
10    void testCountSentences() {
11
12        SentenceCounter sc = new SentenceCounter();
13
14        assertEquals(2, sc.countSentences("Today is Monday. This is it."));
15        assertEquals(1, sc.countSentences("."));
16        assertEquals(2, sc.countSentences(".."));
17        assertEquals(2, sc.countSentences("Hi there. My name is Brandon. What's yours?"));
18        assertEquals(0, sc.countSentences(""));
19
20    }
21
```

Sentence Counter Testing

- But we SHOULD account for other kinds of punctuation, like a question mark (?)
  - Let's fix it. First, update the test with the new expected value (3)
  - If you run this test, it SHOULD fail!

```
 9    @Test
10    void testCountSentences() {
11
12        SentenceCounter sc = new SentenceCounter();
13
14        assertEquals(2, sc.countSentences("Today is Monday. This is it."));
15        assertEquals(1, sc.countSentences("."));
16        assertEquals(2, sc.countSentences(".."));
17        assertEquals(3, sc.countSentences("Hi there. My name is Brandon. What's yours?"));
18        assertEquals(0, sc.countSentences(""));
19
20    }
```

Sentence Counter Testing

- Update the *countSentences* method to account for a question mark (?) and to pass the test

```
5      /**
6       * Counts the number of sentences in the given text.
7       * @param text to extract number of sentences
8       * @return count of sentences
9       */
10     int countSentences(String text) {
11
12         //strip whitespace from beginning and end of string
13         text = text.strip();
14
15         int puncCount = 0;
16         //iterate over string and find instances of period (.)
17         for (int i = 0; i < text.length(); i++) {
18             if (text.charAt(i) == '.' || text.charAt(i) == '?') {
19                 puncCount++;
20             }
21         }
22
23         int sentenceCount = puncCount;
24         return sentenceCount;
25     }
```

More Assert Methods

```
void assertThrows(Exception.class, () -> {
    //code that throws an exception
});
```

- Asserts that the enclosed code throws an Exception of a particular type

```
void assertDoesNotThrow(() -> {
    //code that does not throw an exception
});
```

- Asserts that the enclosed code does not throw an Exception

More Assert Methods

```
void assertThrows(Exception.class, () -> {
    //code that throws an exception
});
```

- Asserts that the enclosed code throws an Exception of a particular type

```
void assertDoesNotThrow(() -> {
    //code that does not throw an exception
});
```

- Asserts that the enclosed code does not throw an Exception

- For example:

```
String test = null;
assertThrows(NullPointerException.class, () -> {
    test.length();
});
```

  - Asserts that *test.length()* throws a NullPointerException
  - Why? *test* is null, so there is no method *length()*

Optional setUp & tearDown Methods

- The following are optional methods (and annotations) when running a testing file
  - These are typically defined at the top of your testing class

```
@BeforeEach
void setUp()
```

  - The setUp method (annotated by @BeforeEach) runs before each unit test method
  - Can be used to initialize everything to a "clean" state

```
@AfterEach
void tearDown()
```

  - The tearDown method (annotated by @AfterEach) runs after each unit test method
  - Can be used to remove artifacts (such as files) that may have been created

Java Midterm

Java Midterm

Will be assigned by Wednesday, October 12th and due Monday, October 17th at midnight

- You will be provided with the skeleton of a program and you will have to implement the methods and pass/write the unit tests. The design of the program has been set up for you.
- Complete all of the required methods
  - Implement all of the methods defined in the provided classes
  - Javadocs have already been provided
  - Add comments to your code
  - You can create any number of helper methods (with Javadocs)
  - The main method has already been implemented for you. DO NOT CHANGE THE CODE IN MAIN.
- Test your code by running (and passing) all of the provided test cases in the given test files. **Write additional test cases as noted** and make sure they pass as expected. Your test cases should be distinct.
- To complete the assignment, submit all the classes in your entire Java project. This will include everything in your "src" folder.