CS514 -

Object Oriented Programming

Getting to know Junit

Fall 2025

Nancy Stevens

UNIVERSITY OF SAN FRANCISCO

Agenda

- Finalize Lab 4 - submit files by Tuesday evening
- Reflection due tonight
- 5 minute break at 10:50
- Upcoming HW 4 assignment

Test-driven programming

- Remember our test methods for the Dog/Cat classes, Movie Review Classifier?
  - Test out each method as you build it
  - Helps think about what a successful method should do
  - Incremental design

- Building test classes is a bit of effort
- We need a better way to test our code - Unit tests
- Today, we will explore an industry standard - JUnit

- jUnit is a *framework* that simplifies the development of unit tests in Java.
  - Note: a *unit test* is a test that just tests one piece of code – a method or class.
  - We can also think about other kinds of tests, such as *integration tests,* which make sure components are working together properly.

- jUnit lets us automate the generation, sequencing and running of tests.

What are Unit Tests?

A software development practice where small, isolated sections of code are tested

- Test for functionality (does my code do what I want it to do?)
- Test for accuracy (is my logic correct?)

What do we test?

- a single method
  - what should a method contain?

A unit test: automated process of verifying a "unit" of code

- Provide specific inputs: check if output matches the expected results
- Basically: write code that tests your other code to catch bugs early

What is being tested?

We can test the following:

- Functional correction and accuracy
- Error handling: do we have a way to prevent a run time error?
- Checking input parameters for values
- Checking for correction of return values
- Optimize an algorithm or performance (we can time the runtime of our code)

Calculator Test

Calculator code:

```
public class Calculator {
    public int add(int a, int b) {
        return a + b;
    }

    public int subtract(int a, int b){
        return a - b;
    }
}
```

Test code for `add`

```
@Test
void testAdd() {
    Calculator calc = new Calculator();
    int expected = 5;
    int actual = calc.add(2, 3);
    assertEquals(expected, actual, "2+3 should be 5!");
}
```

`assertEquals` in JUnit is an assertion method used to verify that two values are equal

Calculator Test

Calculator code:

```
public class Calculator {

    public int add(int a, int b) {...}


    public int subtract(int a, int b){

        return a - b;

    }
}
```

Test code for `subtract`

```
@Test
void subtract() {
 Calculator calc = new Calculator();
    int expected = -5;
    int actual = calc.subtract(5, 10);
    assertEquals(expected, actual,
"5-10 should be -5!");
}
```

`assertEquals` in JUnit is an assertion method used to verify that two values are equal

Calculator Test - AssertionFailed

```
class CalculatorTest {
    @Test
    void testAdd() {
        Calculator calculator = new Calculator();
        int expected = -5;
        int actual = calculator.add(2, 3);
        assertEquals(expected, actual, "2+3 should be 5!");
    }
```

```
org.opentest4j.AssertionFailedError: 2+3 should be 5! ==>
Expected :-5
Actual   :5
```

Calculator Test - Correct approach

```
@Test
void testAddPositiveNegRes() {
    Calculator calculator = new Calculator();
    int expected = -5;
    int actual = calculator.add(2, 3);
    assertNotEquals(expected, actual, "2+3 should be 5, not 5");
}
```

Installing jUnit

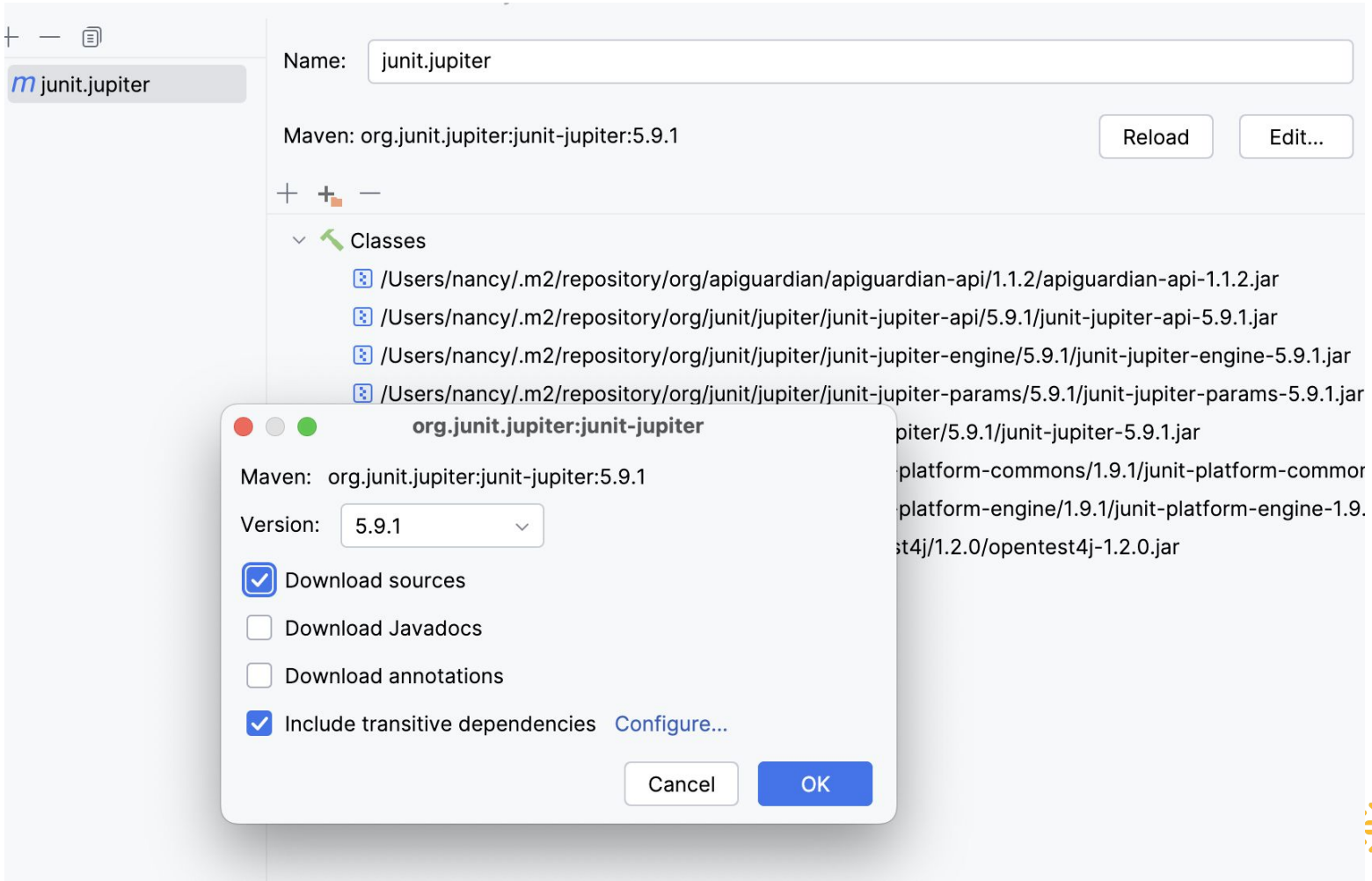
- We will be using the methods defined in Lab 4 to

- If you've never used jUnit before, you'll need to install it
- Instructions are in Canvas:
  - Module 4-> Links -> IntelliJ set-up

- Go to File->Project Structure->Libraries, click '+', choose 'from Maven', and search for **org.junit.jupiter:junit-jupiter:5.9.1** and choose OK.

Installing jUnit

At the next prompt, click on the **Edit** button to select the JUnit source files to be downloaded as well

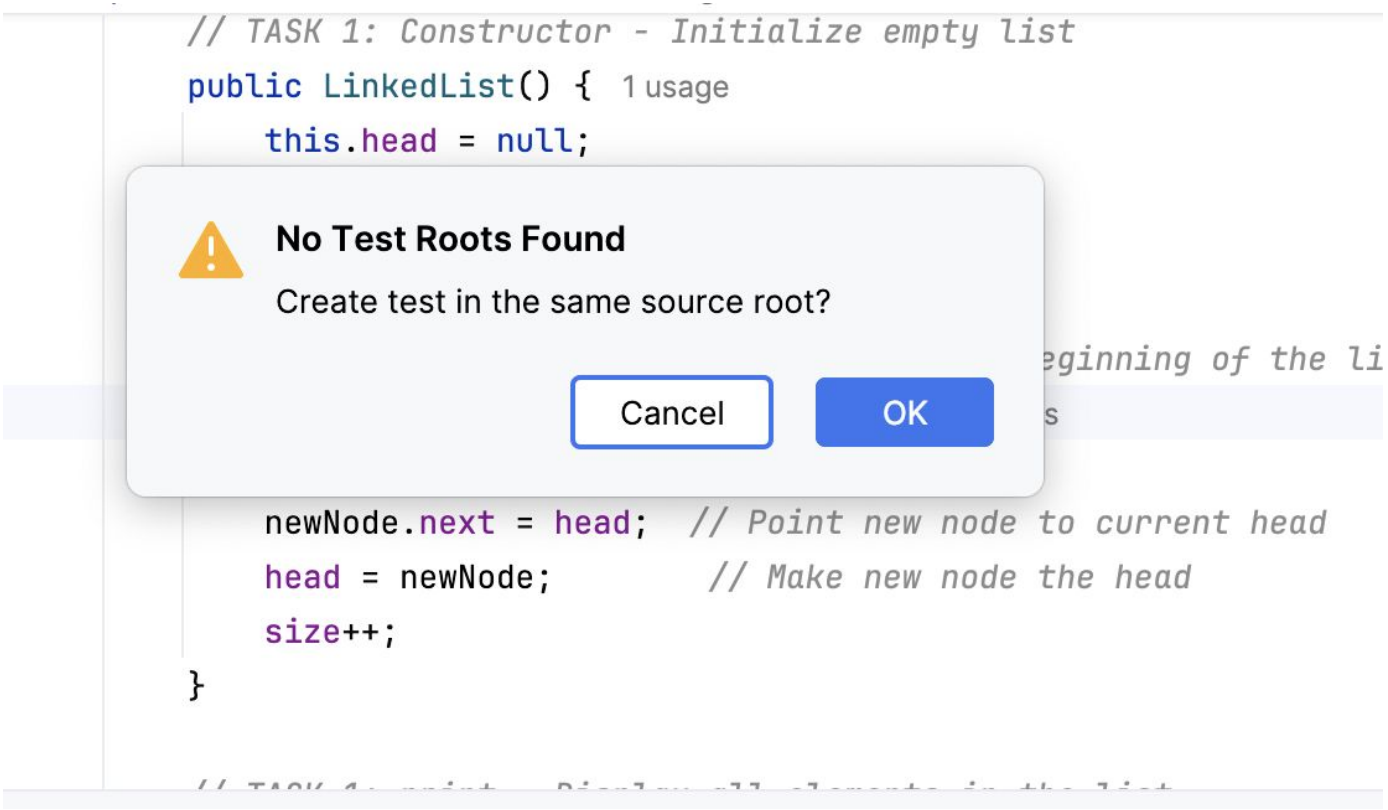


Installing jUnit

JUnit is now installed

Generating a test class

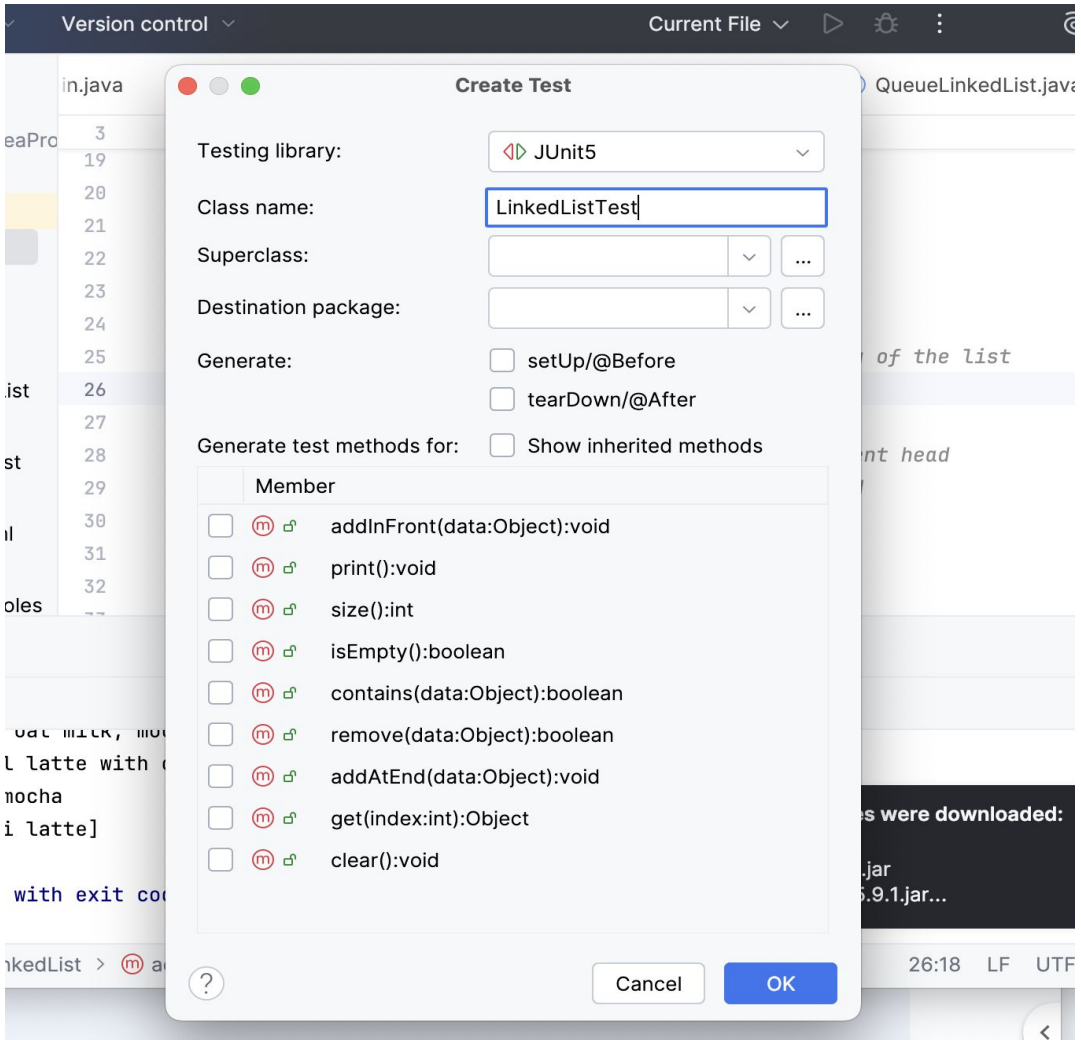
- Go to the LinkedList class, place the cursor in that class someplace, right-click, and choose Generate->Test.



Generating a test class

We can select which methods we want to test.

- Let's choose **`get`**(i), **`size`**()



Generating a test class

jUnit creates a new class for us called **`LinkedListTest`** with test methods included.

- Let's make some changes

```
class LinkedListTest {

    @Test
    void testSizeOfEmptyList() {
    }

    @Test
    void testGetFirstElement() {
    }
}
```

Assertions

- Former way of testing:
  - put print statements throughout the code

- We can use ***assert*** to test directly whether a condition is true.

- Assert says “If this condition holds, the test passes. If not, it fails.”

- No need for us to comb through all the output statements to see if we were correct

Assertions: `assertEquals`

```
class LinkedListTest {
    @Test
    public void testSizeOfEmptyList() {
        LinkedList list = new LinkedList();

        // Empty list should have size 0
        assertEquals(0, list.size());
    }
}
```

We create a LinkedList object (an empty list) and *assert* that the list size is 0 - case equals() returns true.

If this is correct, the test passes.

Assertions: `assertEquals`

```
@Test
public void testGetFirstElement() {
    LinkedList list = new LinkedList();

    list.addAtEnd("Hello");
    list.addAtEnd("World");

    //get element at index 0
    assertEquals("Hello", list.get(0));
}
```

We create a LinkedList object, add two elements, and call the get() method. We should get back “Hello” at index 0, so we *assert* that the expected value is “Hello”

If this is correct, the test passes.

Assertions: `assertEquals` - a failed test

```
@Test
public void testSizeofTwoListElements() {
    LinkedList list = new LinkedList();

    list.addAtEnd("Hello");
    list.addAtEnd("World");

    //get element at index 0
    assertEquals("2", list.size());
}
```

Why does this fail?

Assertions: `assertEquals` - a failed test - fixed

```
@Test
public void testSizeofTwoListElements() {
    LinkedList list = new LinkedList();

    list.addAtEnd("Hello");
    list.addAtEnd("World");

    //get element at index 0
    assertEquals(2, list.size());
}
```

Be careful with the expected values

Assertions: `assertTrue`

```
@Test
public void testFindElement() {
    LinkedList list = new LinkedList();

    list.addAtEnd("Hello");
    list.addAtEnd("World");

    //finding an elem should be true
    assertTrue(list.contains("Hello"));
}
```

Assertion would fail under what conditions?

Assertions: `assertFalse`

```
@Test
public void testNegativeEmptyCheck() {
    LinkedList list = new LinkedList();

    list.addAtEnd("Hello");
    list.addAtEnd("World");
   list.remove("Hello");


    //The list is not empty
    assertFalse(list.isEmpty());
}
```

Assertion would fail under what conditions?

We can have many assertions in one test method!

Assertions

- What can we test with an assertion?
  - Whether a condition is true
  - Whether a condition is false
  - Whether two things are equal

- We want to think about how to construct tests that can test for these conditions.
  - This often requires generating some sample objects with a known structure and manipulating them.

Test suites

- As we develop more tests, we construct what is known as a *test suite.*
- IntellIJ and jUnit make it easy to run some or all of the suite, and see which methods have passed and which have not.

- Now we can make testing part of our build cycle:
  1. Code
  2. Compile successfully
  3. Run tests

Setup

- It’s kind of annoying to create all these objects each time.

- We might want a standard set of lists, queues, etc … to test on.
  - Aim for a single point of change

- We can do this with `@BeforeEach` and `@BeforeAll`.

Setup - @BeforeEach/@BeforeAll

```
class LinkedListTest {

    private LinkedList list;

    @BeforeEach
    public void setUp() {
        list = new LinkedList();
    }
```

…

we can use `list` in our source code

Setup - @AfterEach

```
class LinkedListTest {

    private LinkedList list;

    @BeforeEach
    public void setUp() {
        list = new LinkedList();
    }
```

…

we can use `list` in our source code

Setup

- Sometimes we need to do something once at the beginning of our testing. (Create a log file. Or open a database, for example).
  - Perhaps we want to capture the results of our tests someplace.

- We can do this with **`@BeforeAll`**

  - Note that @BeforeAll is *static*:
    - method runs once for the entire test class before any test instances are created
      - or any test methods are executed.
    - Since no instance exists at this point, the method must be static so that JUnit can call it on the class itself

Cleanup

- Sometimes you also need to do something after each test.
  - Maybe make sure that list is empty?

- We can do this with **`@AfterEach`**

- If we have a one-time cleanup at the end (closing a log file or DB), we can do this with **`@AfterAll`**

JUnit Summary

- jUnit makes it really easy to incrementally add tests to your programs.
  - No need to mix test code in with your production code.
  - Can easily test different parts of your program.
  - Encourages integration of testing and development.

  - Can set up test objects at the beginning.

Lab 4

- Let’s discuss how many and what type of tests to create
- Aim to submit tests by Tuesday @ 9pm