# Testing & JUnit

# Reference

```java
import static org.junit.Assert.*;
import org.junit.*;

public class TestingExamples {

    @Test
    public void testingExampleOne() {
        String inputToTest = "example";
        int otherInputToTest = 4;

        double expected = 14.5;
        double actual = functionToTest(inputToTest, otherInputToTest);

        assertEquals(expected, actual);
    }

    @Test
    public void testingExampleTwo() {
        int[] input = { 3, 4, 5 };
        int result = countEvens(input);
        assertTrue(result < 3);

    }
}
```

# "*Units*" of Code

Now that we have functions, we can write programs that consist of individual, atomic *units* of logic.

In Caesar:

- Encrypting requires converting to symbol array, shifting, and converting back to String
- Each step along the way represented by a function with well-defined inputs and output.

# *Testing a Unit of Code*

How do we test our code to determine if it's right?

# *Testing a Unit of Code*

How do we test our code to determine if it's right?

- Identify the **INPUT**, possibly including any state variables
- Generate, manually or through means OUTSIDE of your code an **EXPECTED OUTPUT**
- Execute your code to get an **ACTUAL OUTPUT**
- Compare the expected and actual output

# *Unit Test Case*

Comprised of:

- An **INPUT**
- An **EXPECTED** output (usually manually coded in)
- An **ACTUAL** output (generated by the code we are testing)
- A mechanism of comparing **ACTUAL** and **EXPECTED**

If an expected output doesn't match the actual output, one of the two is wrong

- Usually, *hopefully,* but not necessarily, the actual output is wrong

# A Function to Test

```
public static int findMax(int a, int b, int c) {
    if (a > b) {
        if (a > c) {
            return a;
        } else {
            return c;
        }
    } else {
        if (b > c) {
            return b;
        } else {
            return a;
        }
    }
}
```

# Generating Tests

`int findMax(int a, int b, int c)` is a function that should return the largest of its three inputs.

- input types: `int`, `int`, `int`
- output type: `int`

What is the *expected output* for `findMax` when called on inputs `3, 2, 1`?
What is the *expected output* for `findMax` when called on inputs `1, 2, 3`?

# *Testing a unit of code*

Is this a *passing* or *failing* test case?

Test Case #1: Input = `{3,2,1}`; Expected output = `3`; Actual output = `3`

Is this a *passing* or *failing* test case?

Test Case #2: Input = `{1,2,3}`; Expected output = `3`; Actual output = `1`

# *Testing a unit of code*

Is this a *passing* or *failing* test case? **PASSING!** ✅

Test Case #1: Input = `{3,2,1}`; Expected output = `3`; Actual output = `3`

Is this a *passing* or *failing* test case? **FAILING!** ❗

Test Case #2: Input = `{1,2,3}`; Expected output = `3`; Actual output = `1`

# *Testing is like potato chips*

> "They both contribute to my overall poor health. Also, you can't have just one." - Will McBurney

One test passing may have no bearing on another test passing! One test is not enough to decide if your implementation is bug-free.

# *Why does Test 1 pass and not Test 2?*

Test 1 does not cover/execute the underlying **fault** in the code.

A **fault** is a particular defect in the code, or bug.

Test Case #1: Input = `{3,2,1}`; Expected output = `3`; Actual output = `3`

Test Case #2: Input = `{1,2,3}`; Expected output = `3`; Actual output = `1`

# *JUnit*

An automatic testing tool that allows you to write tests once and continue to use them again and again.

In this way, if you change something later that breaks code that worked previously, you will immediately know because your tests fail

Technically not built into Java, so we have a bit of wrangling to do.

# Writing a JUnit Test

```
@Test  // This must be before every test function
public void testFindMax0() {      // Notice – no static keyword
    // inputs
    int a = 3;
    int b = 2;
    int c = 1;
    // expected – generated manually
    int expected = 3;
    // actual – Execute the code with the above input
    int actual = findMax(a, b, c);
    // Assertion – if the two things below aren't equal, the test fails.
    // Always put expected argument first.
    assertEquals(expected, actual);
}
```

# writing a JUnit Test

This is the same as before, but we print an error message now that explains what happens when `assertEquals` does not receive matching inputs.

```
@Test  // This must be before every test function
public void testFindMax0() {      // Notice – no static keyword
    // inputs
    int a = 3;
    int b = 2;
    int c = 1;
    String message = "findMax(3, 2, 1) returns the wrong value";
    // expected – generated manually
    int expected = 3;
    // actual – Execute the code with the above input
    int actual = findMax(a, b, c);
    // Assertion – if the two things below aren't equal, the test fails.
    // Always put expected argument first.
    assertEquals(message, expected, actual);
}
```

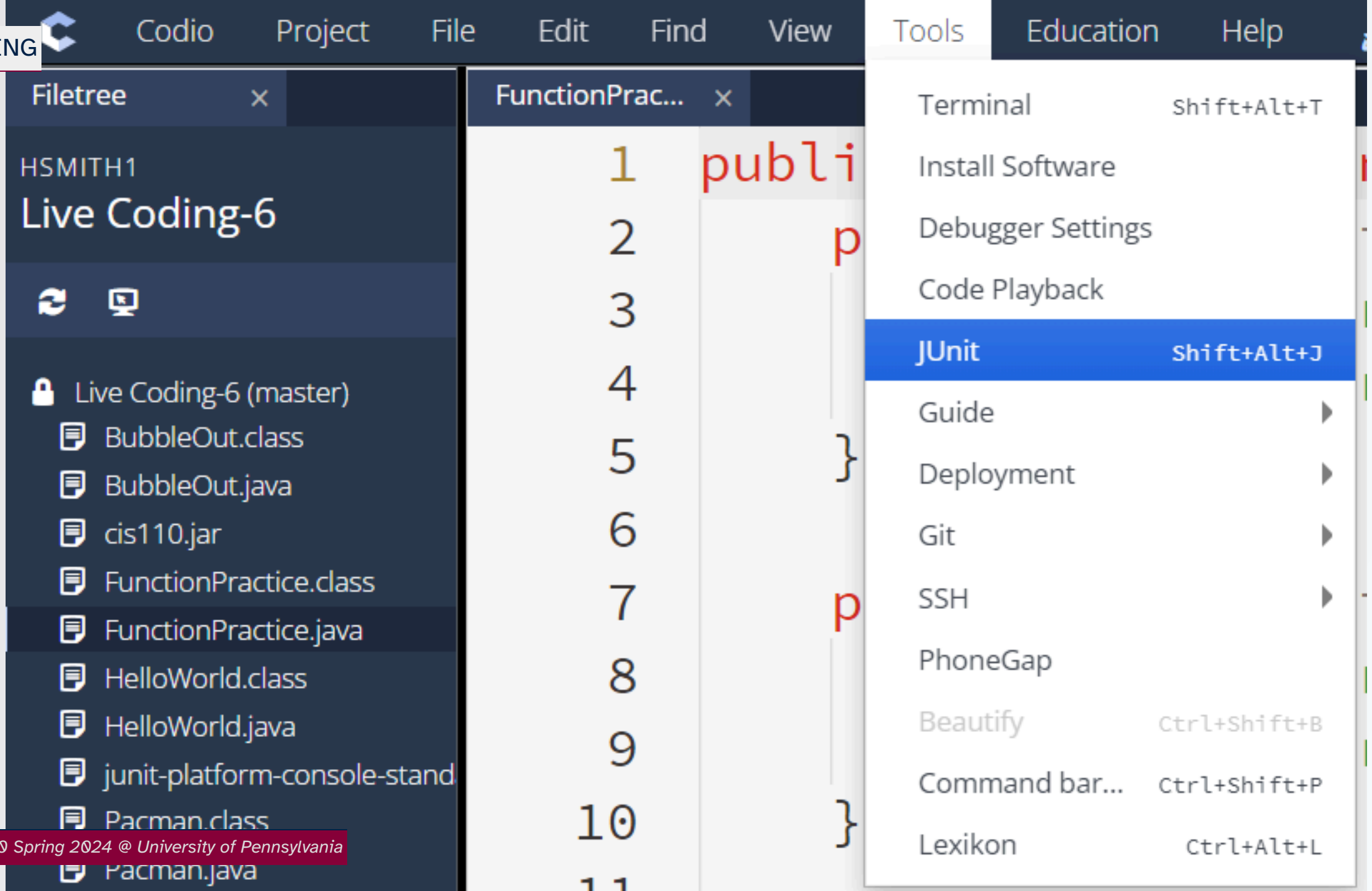
# *Import Statements*

Start all Test files with the two important statements below:

```
import static org.junit.Assert.*;
import org.junit.*;
```

```
1  import static org.junit.Assert.*;
2  import org.junit.*;
3
4  public class FunctionTest {
5
6      @Test
7      public void testMean() {
8
9          double b = 1;
10         double expected = 19;
11         double actual = FunctionPractice.mean(a, b);
12
13         assertEquals(expected, actual, 0.001);
14     }
```

***Writing JUnit (Demo)***


Codio
Project
File
Edit
Find
View
Tools
Education
Help
Filetree
HSMITH1
Live Coding-6
Live Coding-6 (master)
BubbleOut.class
BubbleOut.java
cis110.jar
FunctionPractice.class
FunctionPractice.java
HelloWorld.class
HelloWorld.java
junit-platform-console-stand
Pacman.class
Pacman.java
FunctionPrac...
1 publi
2 p
3
4
5 }
6
7 p
8
9
10 }
Terminal Shift+Alt+T
Install Software
Debugger Settings
Code Playback
JUnit Shift+Alt+J
Guide
Deployment
Git
SSH
PhoneGap
Beautify Ctrl+Shift+B
Command bar... Ctrl+Shift+P
Lexikon Ctrl+Alt+L

HSMITH1

# Live Coding-9



- Live Coding-9 (master)
  - .codioJUnit
  - .settings
  - ArrayManipulations.class
  - ArrayManipulations.java
  - BouncingBall.class
  - BouncingBall.java
  - BouncingBallN.class
  - BouncingBallN.java
  - BubbleOut.class
  - BubbleOut.java
  - BuggyFunctions.java
  - BuggyFunctionsTest.java
  - cis110.jar
  - ClosestTwo.class

Terminal | BuggyFunction... | BuggyFunction... | JUnit

**JUnit Settings** | JUnit Executions

JUnit version: JUnit4

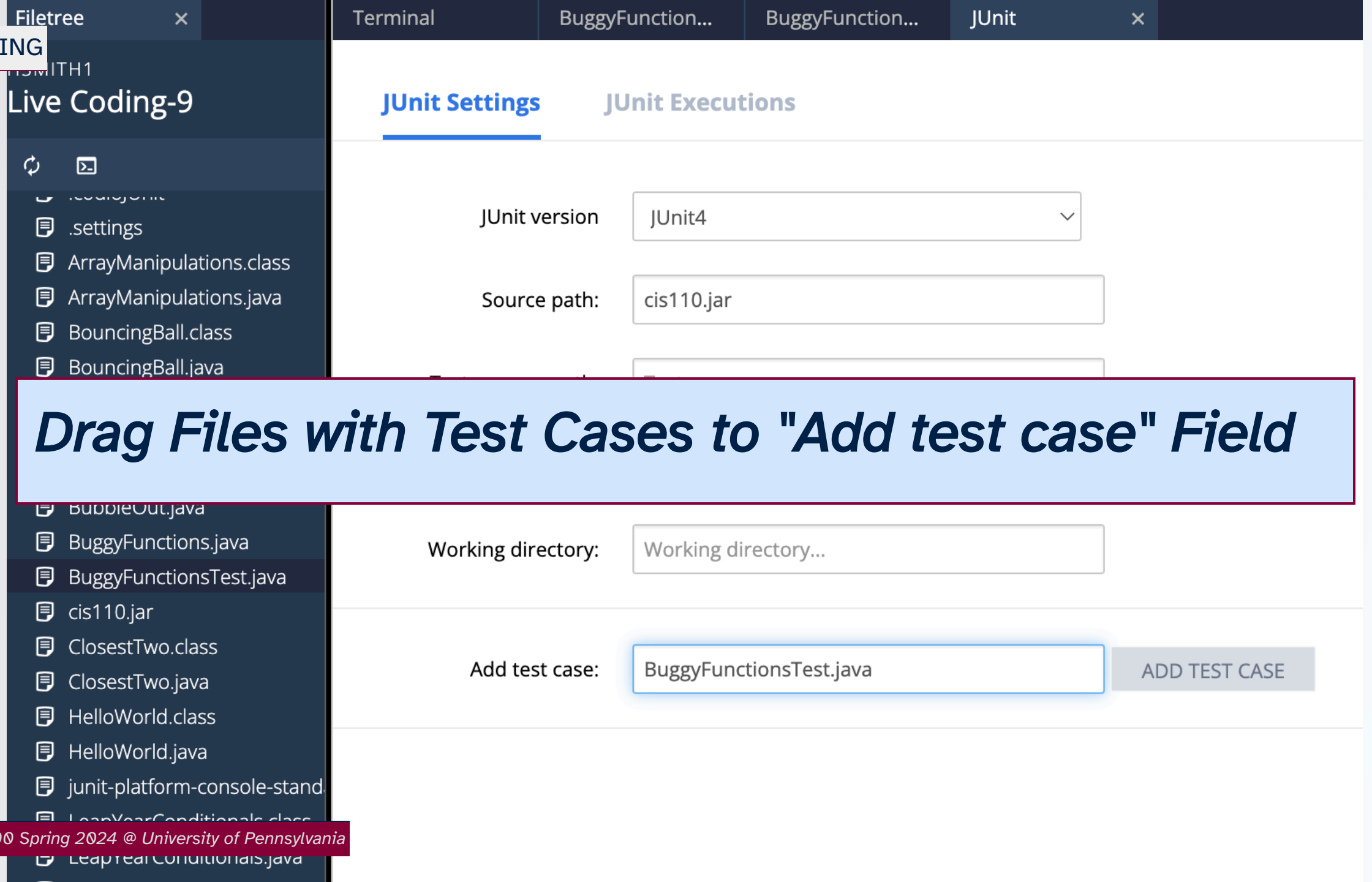
Source path: cis110.jar

Tests source path: Test sources...

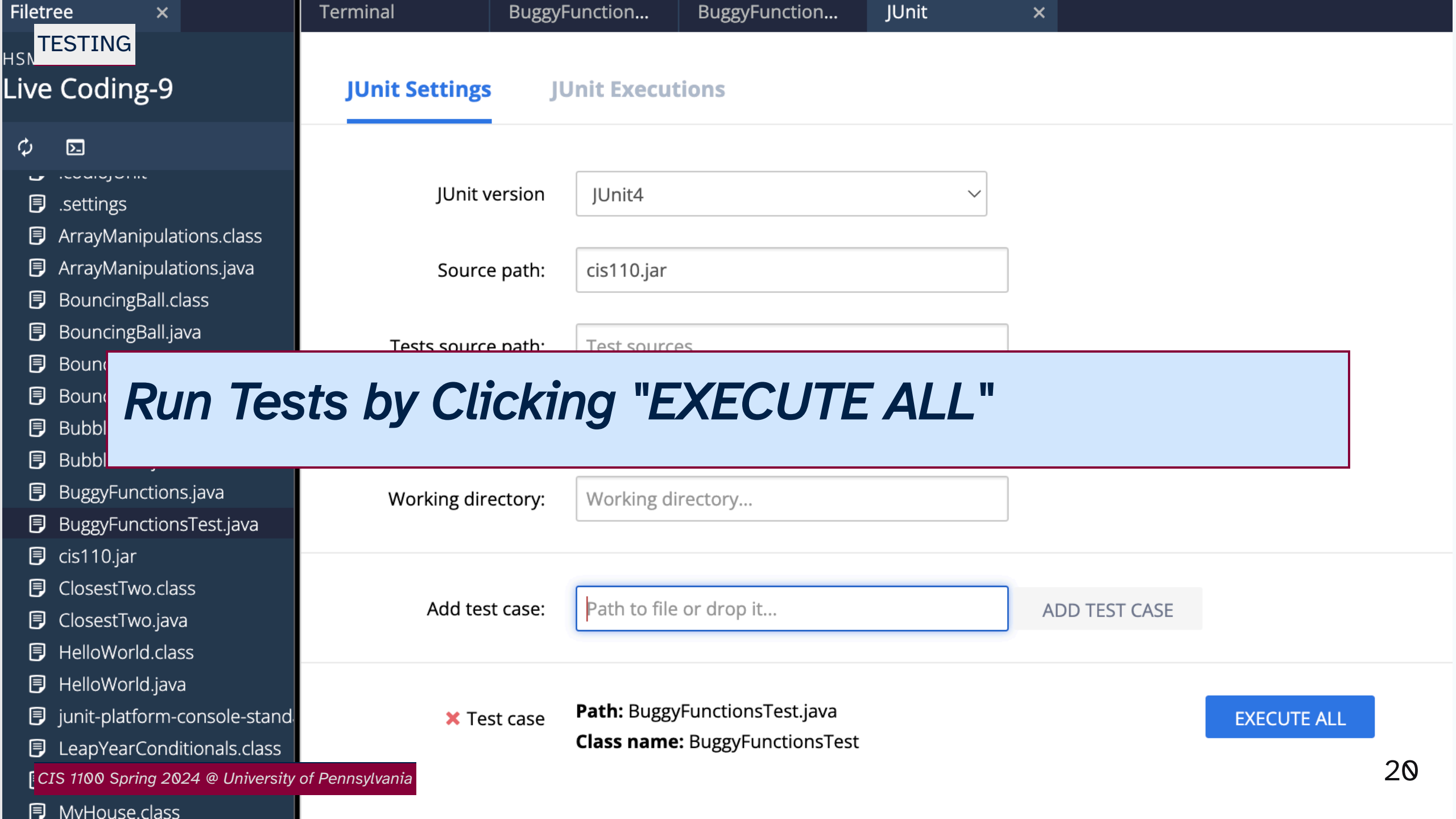
Library path: Libraries...

Working directory: Working directory...

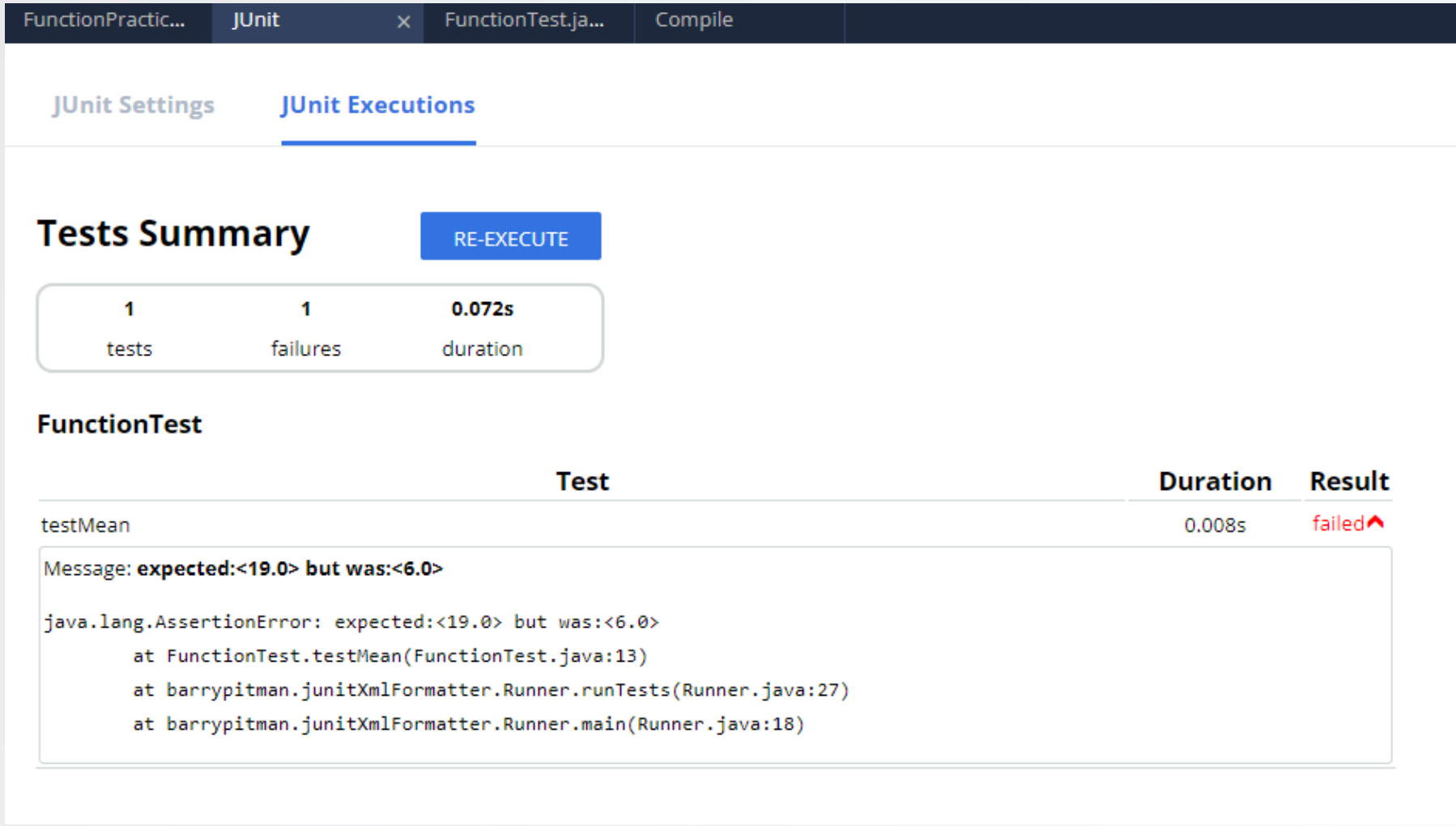
Add test case: Path to file or drop it... ADD TEST CASE



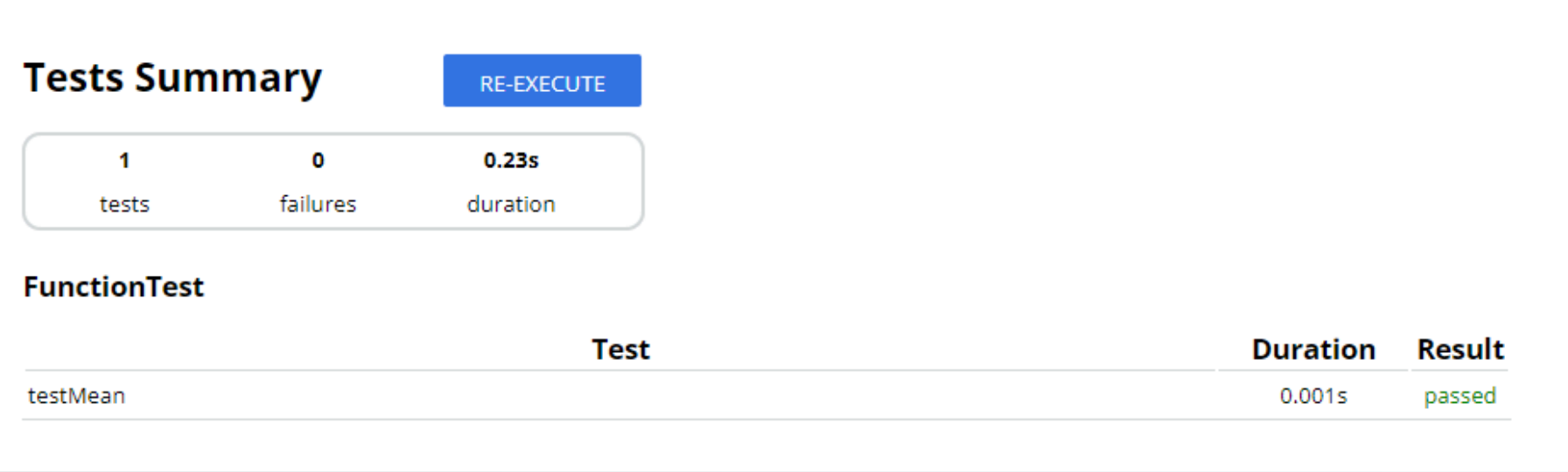

# *Drag Files with Test Cases to "Add test case" Field*


Filetree
TESTING
HSM
Live Coding-9
.settings
ArrayManipulations.class
ArrayManipulations.java
BouncingBall.class
BouncingBall.java
BuggyFunctions.java
BuggyFunctionsTest.java
cis110.jar
ClosestTwo.class
ClosestTwo.java
HelloWorld.class
HelloWorld.java
junit-platform-console-stand
LeapYearConditionals.class
MyHouse.class
Terminal
BuggyFunction...
BuggyFunction...
JUnit
JUnit Settings
JUnit Executions
JUnit version
JUnit4
Source path:
cis110.jar
Tests source path:
Test sources
Run Tests by Clicking "EXECUTE ALL"
Working directory:
Working directory...
Add test case:
Path to file or drop it...
ADD TEST CASE
Test case
Path: BuggyFunctionsTest.java
Class name: BuggyFunctionsTest
EXECUTE ALL

# *A Failing Test*



# *A Passing Test!*



**Tests Summary** RE-EXECUTE

| 1 | 0 | 0.23s |
|---|---|---|
| tests | failures | duration |

**FunctionTest**

| Test | Duration | Result |
|---|---|---|
| testMean | 0.001s | passed |

# *What a test failing means*

A test failing doesn't always mean the code has a bug

- The test could be written wrong (that is, the test writer came up with the wrong expected output)

A test passing doesn't mean there is no bug

- The test code not execute a buggy statement
- The test could execute a buggy statement in a way that a failure doesn't manifest

# *Consider These Test Cases*

Test Case #3: Input = {1,1,1}; Expected output = 1; Actual output = 1;
Test Case #4: Input = {4,5,6}; Expected output=4; Actual output = 4;

#3 is an *edge case*: a set of inputs and output that represent a valid but atypical or unorthodox use case of the function

- at the "edge" of what is a valid way to use the program

#4 is an *incorrectly written* test that passes but does not reveal the presence of the bug. This is considered a *false positive*.

# *False Negatives*

If your test is erroneous, you could also get a false negative.

This test DOESN'T cover the fault, but still fails, due to erroneous testing

Test Case #5: Input = {9,8,7}; Expected output= 7; Actual output = 9;

# *Testing Strategies*

Exhaustive Testing

- Attempt a test with every possible input
- Not even remotely feasible in most cases

Random Testing

- Select random inputs
- Likely to miss narrow inputs that are special cases (example, dividing by zero)

# *Testing Strategies*

Black-box Testing

- Select inputs based on the specification space
- “Assume the code can’t be seen”
- We focus on this one

White-box Testing

- Select inputs based on the code itself
- Have every line of code covered by at least one test

# *The need for automatic testing*

Automatic testing (such as JUnit) allows for testing rapidly after each update

If an update breaks a test, a commit can be rejected

Ensure you don’t break something that already worked

- Not fool proof