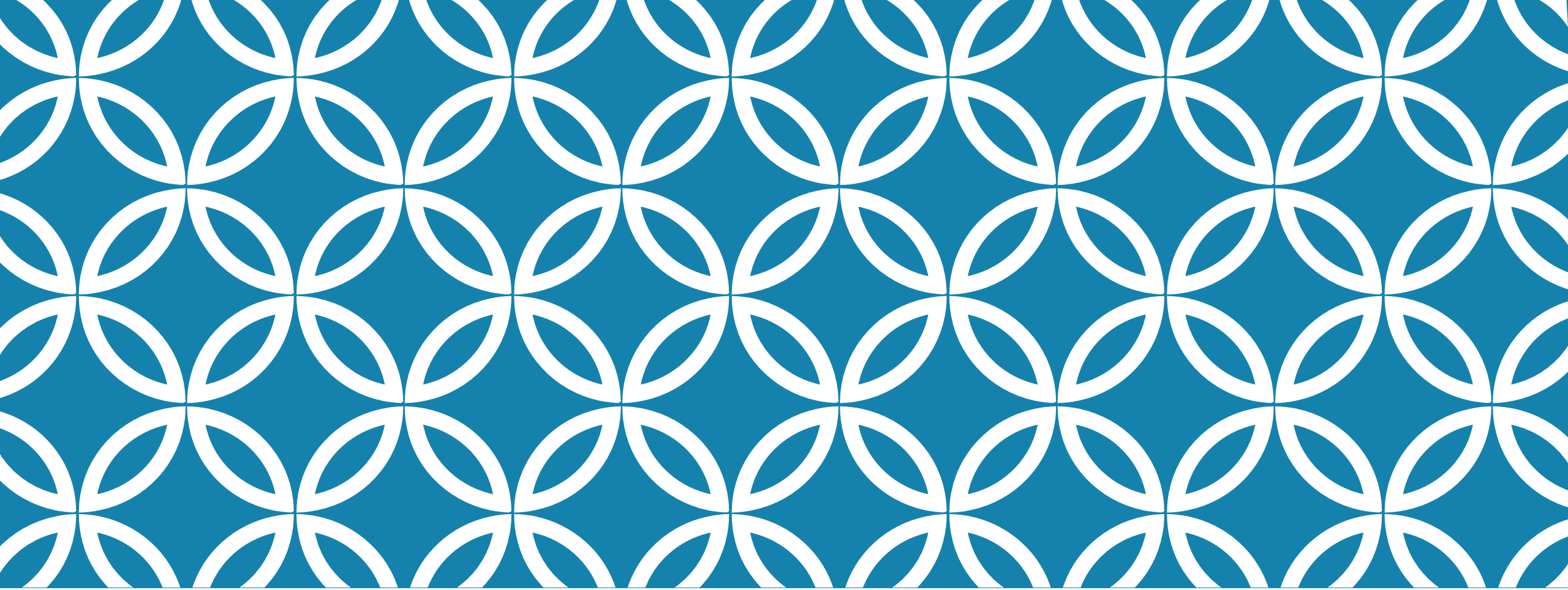


# JUNIT TESTING FRAMEWORK

CS-HU 374 Lecture 2

# OBJECTIVES

## Learn JUnit framework

- Understand the structure of a JUnit test file
- Write passing and failing test cases for BoundedStack.java
- Learn JUnit 4 annotations (@Test, @Before and etc)

## Automated test case generation

- Program that creates a JUnit test file from a text file of inputs
- Test OptimizedMultiplier.java implementation
- Show me your program that ← P1

# JUNIT

JUnit is a Java library that helps with testing

Most useful for testing small pieces of code, i.e., unit testing

- a method
- a class
- not for GUI testing or component testing

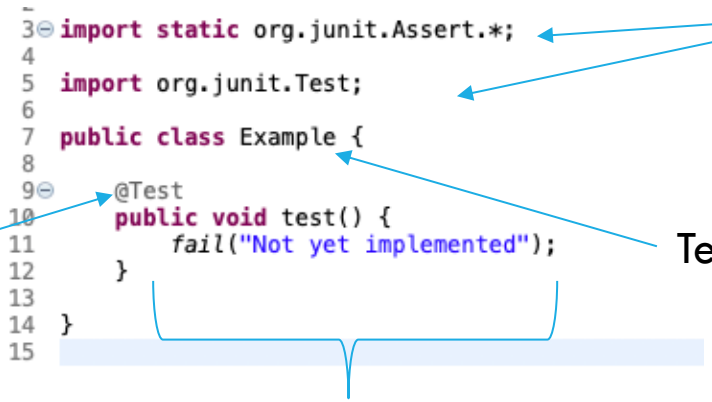
Unit testing focuses on a specific code functionality

Test case description table is given

- How to create test case description tables is the topic of the next three weeks

# JUNIT TEST CASE

JUnit 4 uses @Test annotation to identify a test case method. When a test class runs, only @Test annotated method are executed!




Uses JUnit libraries to help with running test cases

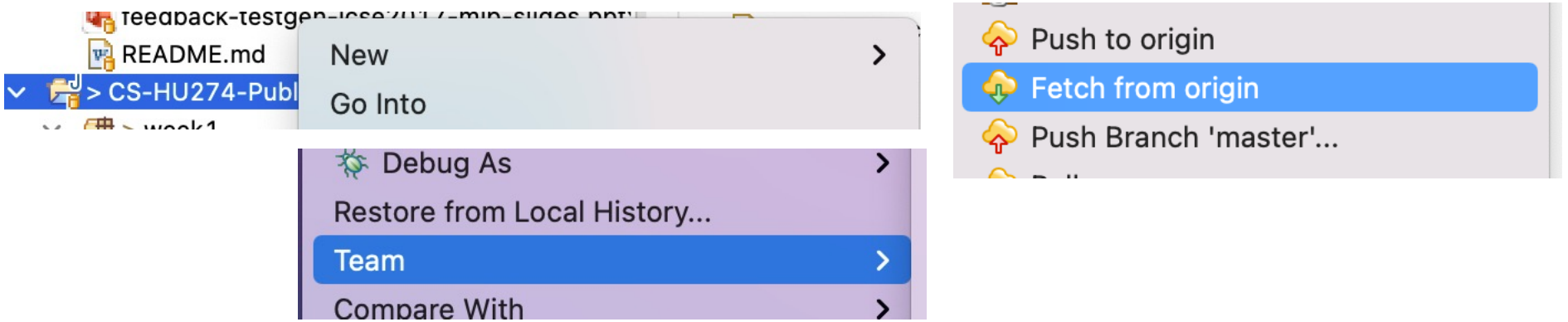
Test class can contain several test cases

The code inside a test case:

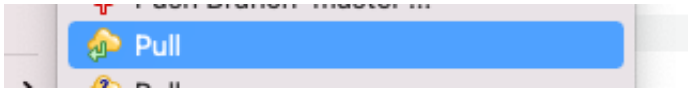
- sets up an initial state
- executes code under test on an input
- checks results

# UPDATE CS-HU 374 REPOSITORY

Right click on the responsory project → Team → Fetch from origin




Right click on the responsory project → Team → Pull



## Code Under Test

```
3  public class BoundedStack {
4
5      private Integer[] elms;
6      private int size = 0;
7      public BoundedStack(int n){
8        elms = new Integer[n];
9      }
10     public void push(int x){
11       elms[size] = x;
12       size++;
13     }
14     public void pop(){
15       size--;
16       elms[size] = null;
17     }
18     public Integer top(){
19       return elms[size -1];
20     }
21
22     public int getSize() {
23         return size;
24     }
25 }
```

w1_code package

## Test Case Description

| Test Case | Initial | Input | Expected |
|---|---|---|---|
| tc1 | an empty stack | push an element x | top element is x |
| tc2 | one element stack | remove an element | size is 0 |

## JUnit test class `BoundedStackTest.java` with the test cases

```
7
8
9      @Test
10     public void tc1() {
11         BoundedStack bStack = new BoundedStack(5);
12
13         Integer el = 5;
14         bStack.push(el);
15
16         assertEquals(el, bStack.top());
17     }
18
19     @Test
20     public void tc2() {
21         BoundedStack bStack = new BoundedStack(5);
22         bStack.push(-3);
23
24         bStack.pop();
25
26         assertEquals(0,bStack.getSize());
27     }
28
29 }
```

Initial

Input

Expected

Initial

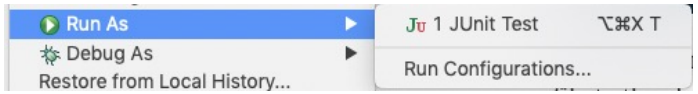
Input

Expected

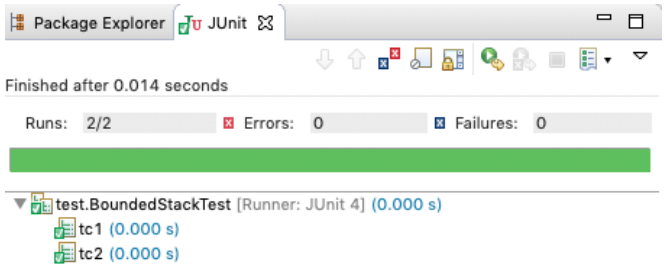
w1_test package

# RUNNING JUNIT TEST CASES

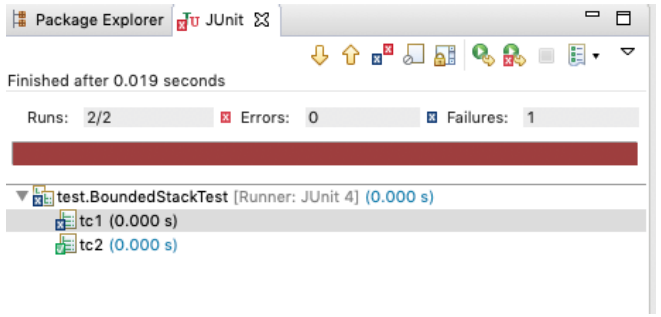
Select BoundedStackTest.java → Run-as → JUnit test case



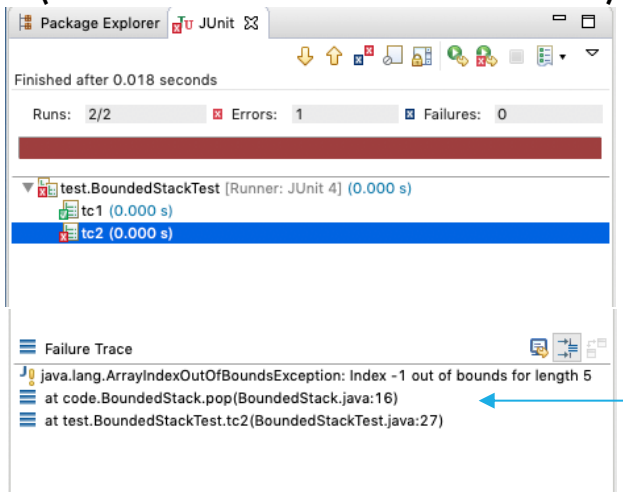

All test cases pass:




tc1 fails (assertion does not hold)




tc1 has a runtime error
(did not make to the assertion)




Error trace

# CREATING A JUNIT TEST CASE

- Extend `BoundedStackTest.java` with the following test cases

| Test Case | Initial | Input | Expected |
|---|---|---|---|
| tc3 | top element is x | push and pop element y | top element is not y |
| tc4 | a nonempty stack | remove an element | old stack size = new stack size +1 |
| tc5 | an empty stack | push an element and pop it | popped = pushed, and stack size = 0 |

- Identify for each test case:
  - Initial
  - Input
  - Expected

- Modify your test class to have
  - Two failing test cases
  - Two error test cases

# SETTING UP A COMMON INITIAL STATE

@Before annotation indicates methods that are executed bore each test case

Used to prepare an environment (e.g., read input data, initialize data structures)

```
import org.junit.Before;
import org.junit.Test;

import code.BoundedStack;

public class Example {
    private BoundedStack bStack;

    @Before
    public void nonEmpty() {
        bStack = new BoundedStack(5);
    }

    @Test
    public void tc1() {
        //BoundedStack bStack = new BoundedStack(5);
        Integer el = 5;
        bStack.push(el);
        assertEquals(el, bStack.top());
    }

    @Test
    public void tc2() {
        //BoundedStack bStack = new BoundedStack(5);
        bStack.push(-3);
        bStack.pop();
        assertEquals(0,bStack.getSize());
    }

}
```

called each time before each test case

- Rewrite your test cases with @Before method

# OTHER JUNIT 4 ANNOTATIONS

| JUnit 4 Annotation | Description |
|---|---|
| `@After` | Executed after each test. It is used to cleanup the test environment (e.g., delete temporary data, restore defaults). It can also save memory by cleaning up expensive memory structures. |
| `@BeforeClass` | Executed once, before the start of all tests. It is used to perform time intensive activities, for example, to connect to a database. Methods marked with this annotation need to be defined as static to work with JUnit |
| `@AfterClass` | Executed once, after all tests have been finished. It is used to perform clean-up activities, for example, to disconnect from a database. Methods annotated with this annotation need to be defined as static to work with JUnit. |
| `@Ignore or @Ignore("why ignored")` | Marks that the test should be disabled. This is useful when the underlying code has been changed and the test case has not yet been adapted. Or if the execution time of this test is too long to be included. It is best practice to provide the optional description, why the test is disabled. |
| `@Test (expected = Exception.class)` | Fails if the method does not throw the named exception. |
| `@Test(timeout=100)` | Fails if the method takes longer than 100 milliseconds to run. |

https://www.vogella.com/tutorials/JUnit/article.html

# OTHER METHODS AND VARIABLES

Test class can have other methods that can be called by test cases

E.g., pushing some random integer elements onto the stack

Test class can have instance variables that can be used by test cases

- Be careful using them since the order in which test cases are executed is not fixed.

```
public class Example {
    private BoundedStack bStack;

    @Before
    public void nonEmpty() {
        bStack = new BoundedStack(5);
    }

    @Test
    public void tc1() {
        //BoundedStack bStack = new BoundedStack(5);
        pushElem(0);
        Integer el = 5;
        bStack.push(el);
        assertEquals(el, bStack.top());
    }

    @Test
    public void tc2() {
        //BoundedStack bStack = new BoundedStack(5);
        //bStack.push(-3);
        pushElem(1);
        bStack.pop();
        assertEquals(0,bStack.getSize());
    }

    private void pushElem(int number) {
        Random r = new Random();
        while(number > 0) {
            bStack.push(r.nextInt(100)- 50);
            number--;
        }
    }
```

# OTHER JUNIT 4 ASSERT STATEMENTS

| Assert Statement | Description |
|---|---|
| `fail([message])` | Let the method fail. Might be used to check that a certain part of the code is not reached or to have a failing test before the test code is implemented. The message parameter is optional. |
| `assertTrue([message,] boolean condition)` | Checks that the boolean condition is true. |
| `assertFalse([message,] boolean condition)` | Checks that the boolean condition is false. |
| `assertEquals([message,] expected, actual)` | Tests that two values are the same. Note: for arrays the reference is checked not the content of the arrays. |
| `assertEquals([message,] expected, actual, tolerance)` | Test that float or double values match. The tolerance is the number of decimals which must be the same. |
| `assertNull([message,] object)` | Checks that the object is null. |
| `assertNotNull([message,] object)` | Checks that the object is not null. |
| `assertSame([message,] expected, actual)` | Checks that both variables refer to the same object. |
| `assertNotSame([message,] expected, actual)` | Checks that both variables refer to different objects. |

https://www.vogella.com/tutorials/JUnit/article.html

# AUTOMATE JUNIT TEST CASE GENERATION

The test table has 1000 test cases

Writing by hand all 1K JUnit test cases is cruel

Maintaining it is even more challenging

Automate

- Test table in a text file (e.g., CSV file)
- Reade the file and for each line generate a test case
- Read the file line by line using scanner (hasNextLine(), nextLine())
- Extract parameters from each line using scanner (useDelimiter(“,”), next())
- Use a test case template and insert test cases parameters
- Write into a file

# IN-CLASS EXERCISE – P1

Goal: generate JUnit test cases for OptimizedMultiplier.java class

Each test case checks that the values of `stanardMultiply` and `fastMultiply` methods are the same

CSV file contains 1000 inputs

OptimizedMultiplier.java | BigIntTestInputs.txt

```
1 -886421817479149,952669276698546
2 460781641541,893623412240141
3 -522490729530852,-14936649279007
4 -496463304952248,-847959580232673
5 -1011410642986716,705885777532914
6 319120212086176,517518316925150
7 523121253889084,-938005584173269
8 -227259532588339,95965904656791
9 -273865190383556,1052650936049535
10 -750987408359670,-688135609563649
11 873968478305075,-861759624395842
12 1095985105890305,976065283933288
13 12206244503807,938850995098206
```

- Create the first two test cases by hand in w1_test.OptimizedMultipolerTestManual
  - File → New → JUnit Test Case
- Identify a common pattern for the template
- Create MultiplierTestGen.java class
  - It takes a text file (e.g., BigIntTestInputs.txt)
  - It outputs OptimizedMultiplierTest.java file in w1_test package
- Hint: EmptyTestFile.java
- Complete it and demonstrate it to me
  - You will use this approach to generate JUnit test cases through the rest of our class.

Next time: Blackbox testing