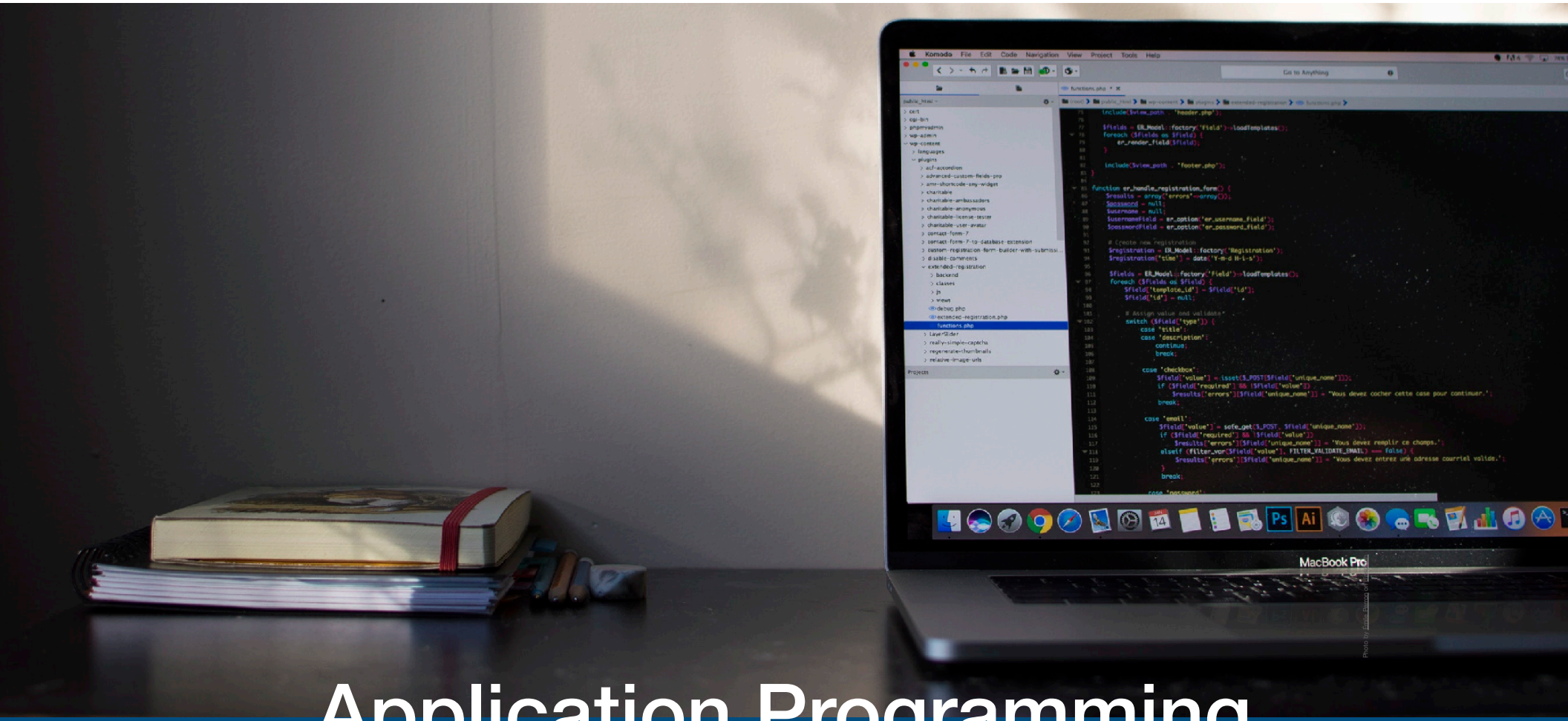


Application Programming

UTSA Computer Science 3443

Fall 2022

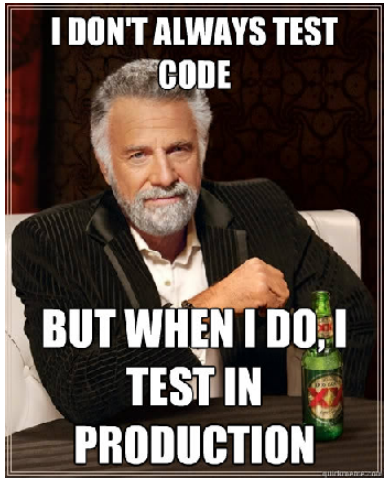
Dr. Amanda Fernandez

Fall 2022*

Week	Dates	Topic	Lab	Quiz	Chapter(s)	Project
1	8/22 - 8/26	Introductions, Syllabus, Java basics, Eclipse			1-6	
2	8/29 - 9/2	Java & OOP Concepts, Javadoc, Strings	1	1	8,14	Teams created
3	9/5* - 9/9	ArrayList, File I/O			7,15	
4	9/12 - 9/16	UML, MVC, Introduction to JavaFX	2	2	12,13	
5	9/19 - 9/23	SOLID Principles & JavaFX				Proposal
6	9/26 - 9/30	JavaFX & Review	3	3		
7	10/3 - 10/7	Introduction to Git. *Review*				
8	10/10 - 10/14	**Midterm Exam**				
9	10/17 - 10/21	JavaFX applications	4	4	13,20	
10	10/24 - 10/28	Exception Handling			11	
11	10/31 - 11/4	Collections & Generics		5	16, 19	
12	11/7 - 11/11	JUnit Testing, logging, Application Design	5			
13	11/14 - 11/18	Concurrency & Multithreading, Lambda Expressions		6	17	
14	11/21 - 11/23*	*Review* (+*Thanksgiving*)	6		21	Demo video
15	11/28 - 12/1	**Team project demos**				Code, Presentation, Survey
-	12/5 - 12/9	**Final Exams**				

JUnit Testing

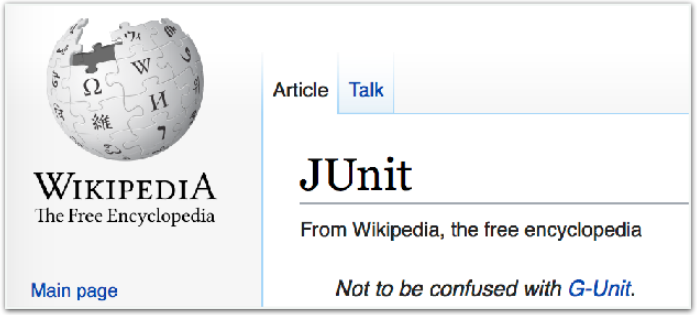
Unit Testing



- Systematic attempt to reveal errors.
  - Pass/fail assigned depending on generation of errors.

- Bugs are **inevitable**!

JUnit



- JUnit is a simple framework to write repeatable tests.
    - It is an instance of the xUnit architecture for unit testing frameworks.

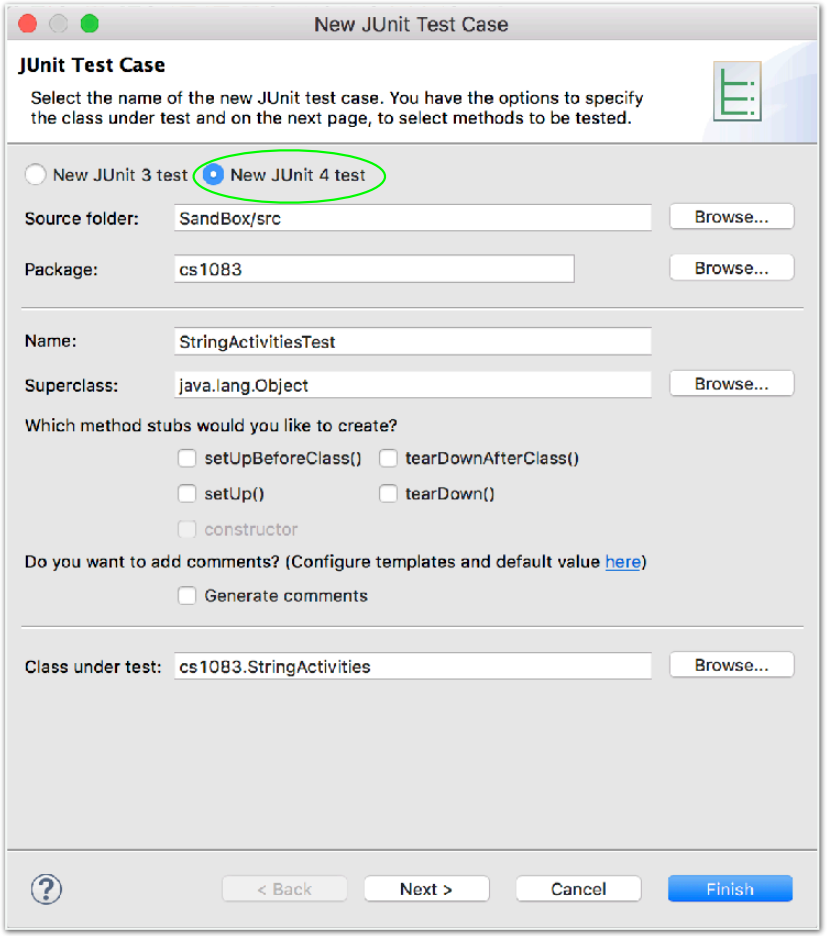
JUnit

- In Eclipse > New JUnit Test Case
  - Previous versions inherited from **junit.framework.TestCase** (JUnit 4+ does *not*)

- JUnit is linked as a JAR at compile-time
  - The framework resides under package `org.junit`

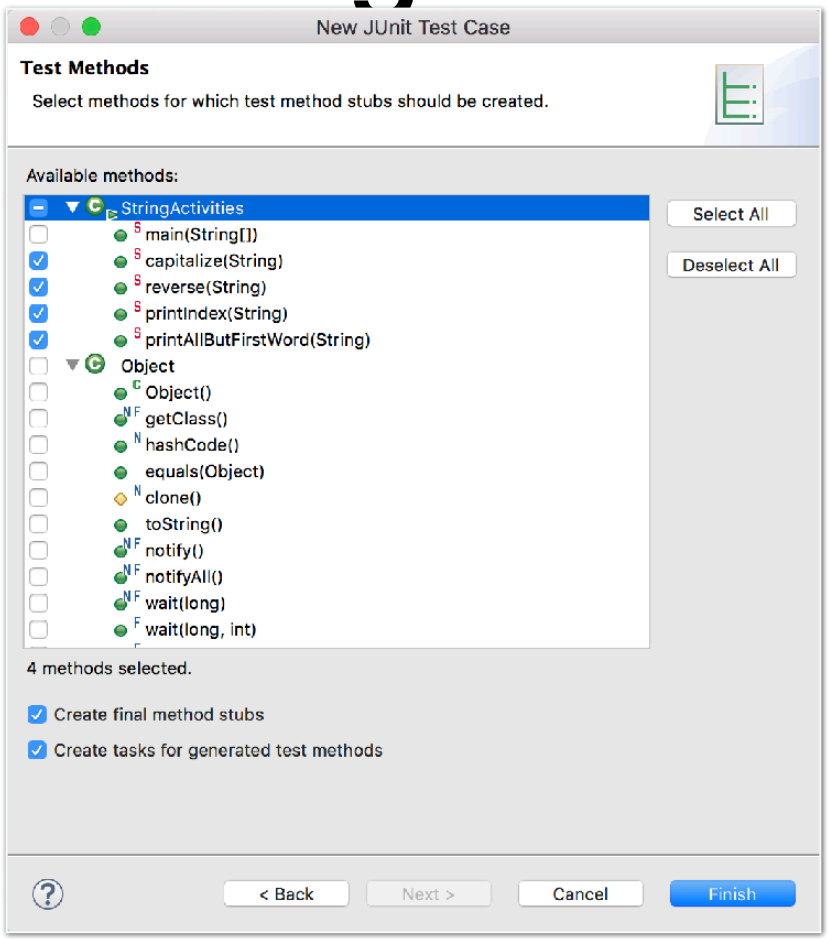
Unit Testing

- Select JUnit **4**
- Declare the default methods needed for your test class.



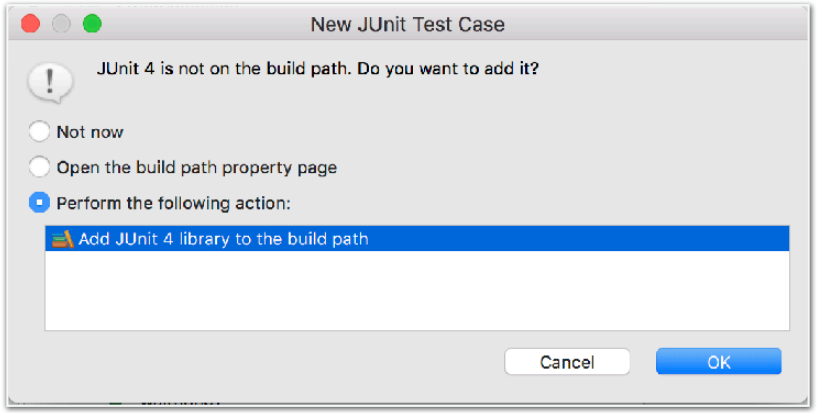
Unit Testing

- Select which methods to be **tested** in the generated class.



Unit Testing

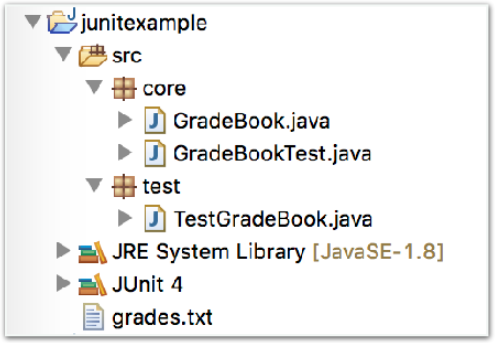
- Add the JUnit library to the build path.



JUnit Annotations

Annotation	Description
`@Test` `public void method()`	The `Test` annotation indicates that the public void method to which it is attached can be run as a test case.
`@Before` `public void method()`	The `Before` annotation indicates that this method must be executed before each test in the class, so as to execute some preconditions necessary for the test.
`@BeforeClass` `public static void method()`	The `BeforeClass` annotation indicates that the static method to which is attached must be executed once and before all tests in the class. That happens when the test methods share computationally expensive setup (e.g. connect to database).
`@After` `public void method()`	The `After` annotation indicates that this method gets executed after execution of each test (e.g. reset some variables after execution of every test, delete temporary variables etc)
`@AfterClass` `public static void method()`	The `AfterClass` annotation can be used when a method needs to be executed after executing all the tests in a JUnit Test Case class so as to clean-up the expensive set-up (e.g disconnect from a database). Attention: The method attached with this annotation (similar to `BeforeClass`) must be defined as static.
`@Ignore` `public static void method()`	The `Ignore` annotation can be used when you want temporarily disable the execution of a specific test. Every method that is annotated with `@Ignore` won't be executed.

Unit Testing




```java
import org.junit.*;

public class FoobarTest {
    @BeforeClass
    public static void setUpClass() throws Exception {
        // Code executed before the first test method
    }

    @Before
    public void setUp() throws Exception {
        // Code executed before each test
    }

    @Test
    public void testOneThing() {
        // Code that tests one thing
    }

    @Test
    public void testSomethingElse() {
        // Code that tests something else
    }

    @After
    public void tearDown() throws Exception {
        // Code executed after each test
    }

    @AfterClass
    public static void tearDownClass() throws Exception {
        // Code executed after the last test method
    }
}
```

Source

```
3  import java.util.Scanner;
4
5  public class StringActivities {
6
7      public static void main( String[] args ) {
19
20     public static String capitalize( String input ) {
21         return input.toUpperCase();
22     }
23
24     public static String reverse( String input ) {
25         String text = "";
26         for( int i=input.length()-1; i>=0; i-- )
27             text += input.charAt(i);
28
29         return text;
30     }
31
32     public static void printIndex( String input ) {
33         for( int i=0; i<input.length(); i++ )
34             System.out.println( input.charAt(i) + ":" + i );
35     }
36
37     public static void printAllButFirstWord( String input ) {
38         String[] tokens = input.split( " " );
39
40         for( int i=1; i<tokens.length; i++ )
41             System.out.print( tokens[i] + " ");
42     }
43
44 }
45
```

JUnit Test

```
3  import static org.junit.Assert.*;
6
7  public class StringActivitiesTest {
8
9      @Test
10     public final void testCapitalize() {
11         fail("Not yet implemented"); // TODO
12     }
13
14     @Test
15     public final void testReverse() {
16         fail("Not yet implemented"); // TODO
17     }
18
19     @Test
20     public final void testPrintIndex() {
21         fail("Not yet implemented"); // TODO
22     }
23
24     @Test
25     public final void testPrintAllButFirstWord() {
26         fail("Not yet implemented"); // TODO
27     }
28
29 }
30
```

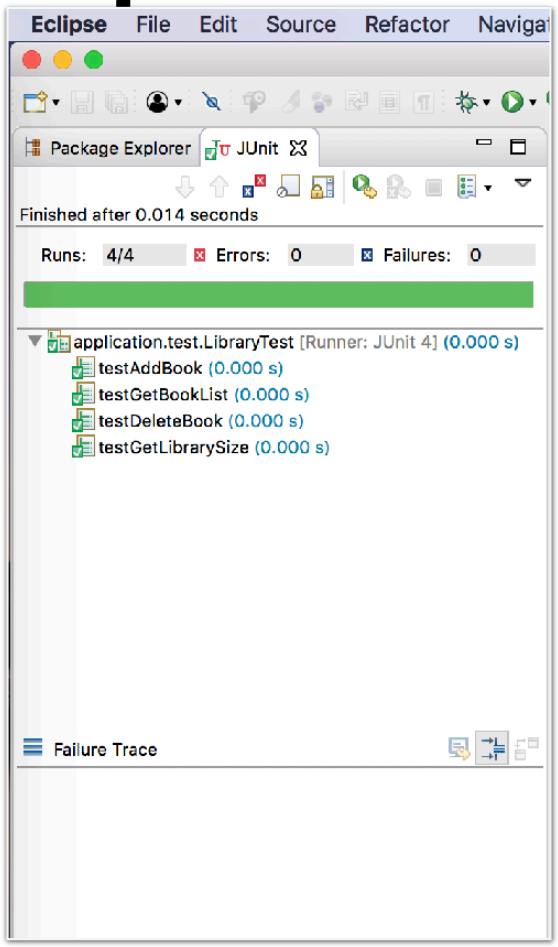
Note that no test method is created for the main method.

JUnit - Example

- In our ebook reader application, let's build an example JUnit test.
    - The Library class is responsible for keeping a list of books to be displayed.

JUnit - Example

- setUp()
- tearDown()



Unit Testing

- For each method implemented, we can examine preconditions and postconditions to help define JUnit tests.
  - **Preconditions**
    - Assumptions/requirements made on the **parameters** or **class variables** to be used in the method.
    - Should be enforced by explicit checks inside methods resulting in particular, specified exceptions
  - **Postconditions**
    - Assumptions/requirements made on the **returned value** (or updated class variables) *at the end of the method.*

Unit Testing

For each method:
1.Preconditions
2. Postconditions
3. Possible exceptions
4. Names and Annotations of JUnit test methods

- **@Before** executes before each test method
- **@BeforeClass** executes before the test *class*
- **@After** executes after each test method
- **@AfterClass** executes after the test *class*

Hello, World - JUnit Edition

```java
public class Hello {


    /**
     * This main method prints out the number 18.
     *
     * @param args - a String[] that contains parameters to my program
     */
    public static void main( String[] args ) {
        int x = 18;

        assert(x==18) : "unexpected value: " + x;

        System.out.println( x );
    }

    public static int updateValue( int x ) {
        if( x!=0 )
            return 4/x;
        return 0;
    }

}
```

```java
public class HelloTest {

    private Hello hello;

    @Before
    public final void setUp() {
        hello = new Hello();
    }

    @Test
    public final void testUpdateValue() {

        // assert( this is true ) : "this is the error message";

        // check good behavior
        int result1 = Hello.updateValue( 1 );
        assert( result1==4 ) : "Error in good check for update value 4!=" + result1;

        // check bad behavior
        int result2 = Hello.updateValue( 0 );
        assert( result2==0 ) : "Error in bad check for update value 0!=" + result2;
    }

}
```

Activity 0

- Suppose you need to create a banking system…
  - Each bank has accounts, belonging to a customer.
  - All accounts have balances, which should be positive.
  - Customers should be able to open an account, deposit money, and withdraw money.
- **How should we program Bank.java?**

Activity 1

- Suppose you need to create a banking system…
  - Each bank has accounts, belonging to a customer.
  - All accounts have balances, which should be positive.
  - Customers should be able to open an account, deposit money, and withdraw money.
- **Discuss what the JUnit test case for Bank.java should include.**