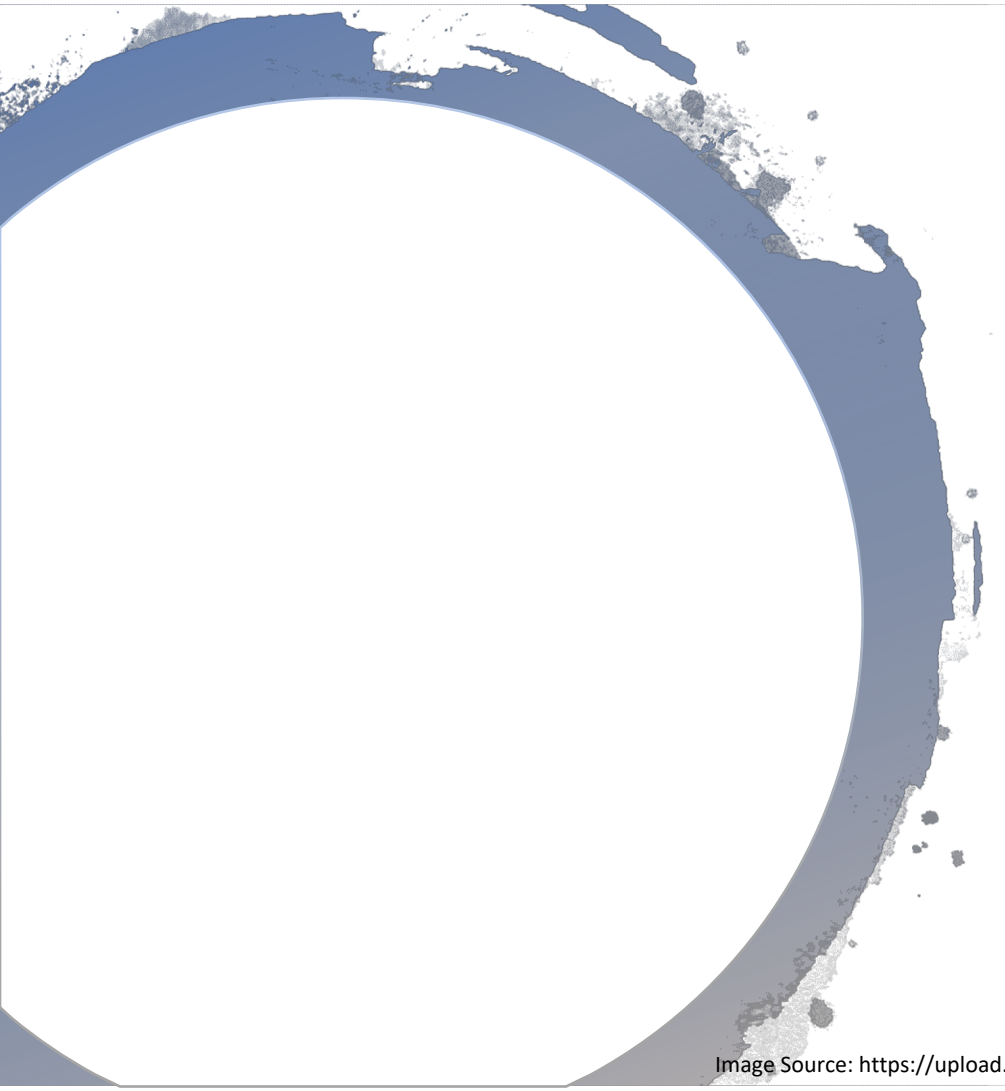


Jin L.C. Guo

M4 – Unit Testing

Image Source: https://upload.wikimedia.org/wikipedia/commons/2/2b/Cepaea_nemoralis_active_pair_on_tree_trunk.jpg

Covered by TAs

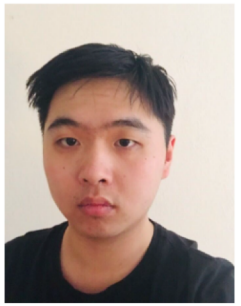
- TA Team

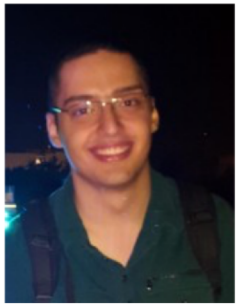

Mathieu Nassif


Deeksha Arya


Alexander Nicholson


Cheryl Wang


Shi Yan Du


Kian Ahrabian

Material from:

Introduction to Software Design with Java, by Martin Robillard, lecture notes for COMP 303. (LN)

(Module 4)

The Pragmatic Programmer by Andrew Hunt and David Thomas, Addison-Wesley, 2000. (PP)

(34, 43)

Clean Code by Robert C. Martin, Prentice Hall, 2008

(Chapter 9)

Objectives

- Be able to explain the foundational concepts of testing using the proper terminology;
- Understand type annotations and program reflection and be able to use them effectively;
- Be able to write unit tests with JUnit;
- Be able to approach more advanced testing problems requiring reflection or mock objects;
- Be able to understand the output of a test coverage tool such as EclEmma;
- Be able to understand basic test suite adequacy criteria and the relations between them;

Objectives – This Class (Feb 5.)

- Be able to explain the foundational concepts of testing using the proper terminology;
- Understand type annotations and program reflection and be able to use them effectively;
- Be able to write unit tests with JUnit;
- Be able to approach more advanced testing problems requiring reflection or mock objects;
- Be able to understand the output of a test coverage tool such as EclEmma;
- Be able to understand basic test suite adequacy criteria and the relations between them;

Objectives – Next Class (Feb 7.)

- Be able to explain the foundational concepts of testing using the proper terminology;
- Understand type annotations and program reflection and be able to use them effectively;
- Be able to write unit tests with JUnit;
- Be able to approach more advanced testing problems requiring reflection or mock objects;
- Be able to understand the output of a test coverage tool such as EclEmma;
- Be able to understand basic test suite adequacy criteria and the relations between them;

Objective 1: Foundational Concepts

Me: I don't need to test this functionality...

Functionality: *breaks immediately*

Me:

What is Testing? 🤔

Based on the idea that: software issues arise when code does not do what we expect.

- A fairly intuitive concept.
- You might have done some testing before!

“All software you write *will* be tested—if not by you and your team, then by the eventual users—so you might as well plan on testing it thoroughly … ”

The Pragmatic Programmer by Andrew Hunt and David Thomas, Addison-Wesley, 2000. (PP)

What is Testing? 🤔

A proposal: a process of *automatically* running code to verify expectations about it.

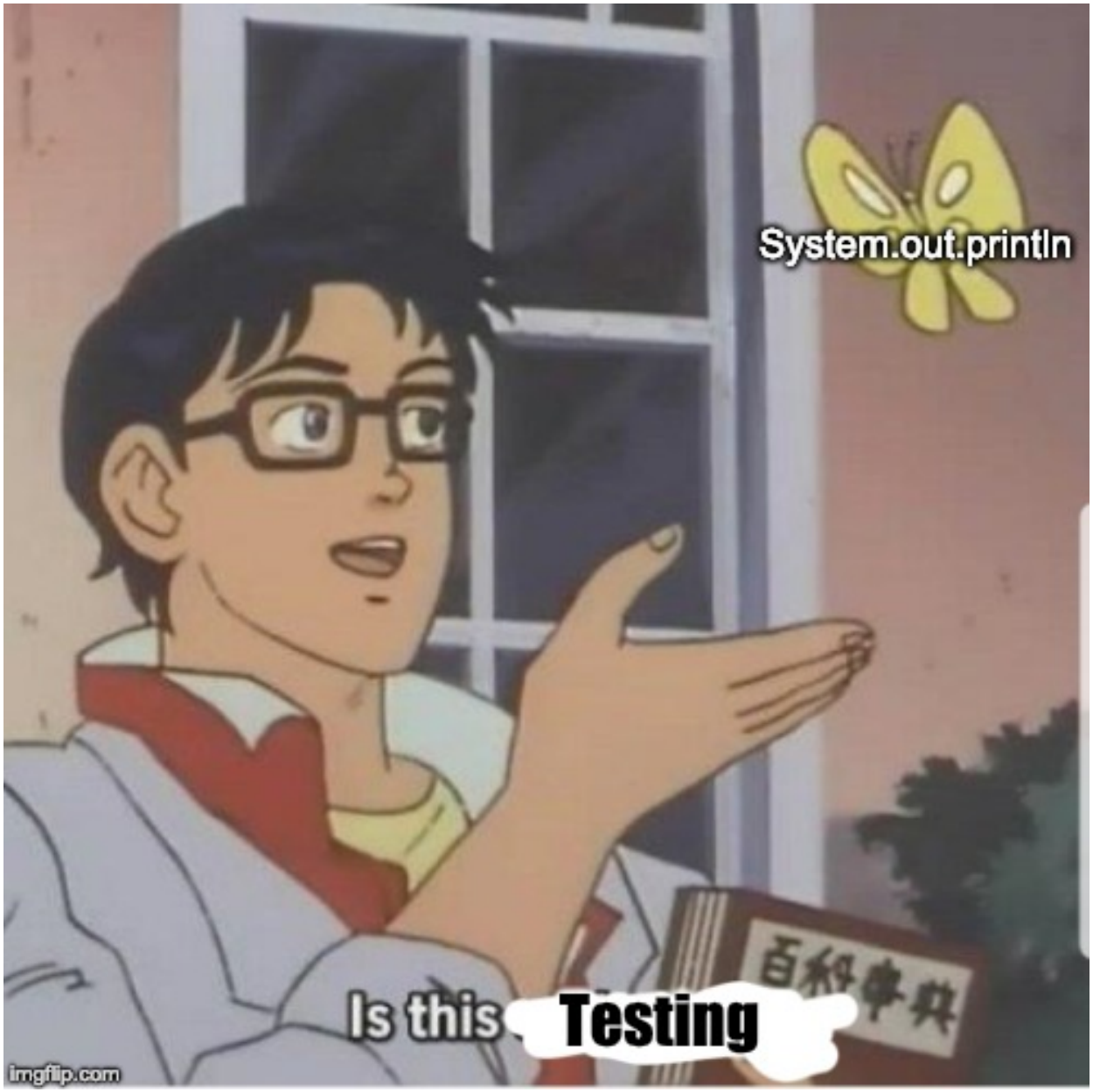
Activity: Why automatically?

Automatic: Easy to (re-)run

- Code is evolutionary: when we change code, are we (re-)introducing bugs?

- Code should run in multiple environments, with multiple configurations.

- <<Your ideas>>


System.out.println
Is this
Testing
imgflip.com

Automatic

(Automatic) Testing is important!

How do we get started?

What should we be testing?

What should we test?

Recall: Design by Contract

Contracts, a Perspective:

Contracts:

- Help us clarify (and verify) expectations
- Help us write code that is easy to test

What should we test?

Recall: Design by Contract

Problem: no built-in support…
We will see how to address this.

Automatic Contracts

What specifically can we test?

Recall: Separation of concerns.

We have a design that minimizes (eliminates) changes in one place affecting other places.

Let's test small, self-contained units of functionality.

Unit Testing 💡

An idea: We can test each unit of functionality in isolation – gives us confidence that when we put them together they won't break.

*Note: This is just one approach to testing.

Breaking things down into units (of functionality)

Activity:

What are some units of functionality in:

- A Car (Driving)
- A Laptop (Playing Music)
- …

Key Point 🔑

Should you test everything?

In a car/laptop what if you buy some components?

When designing should you be testing everything?

Would you test ArrayList for example?

Automatic
Contracts
Units

Objective 3: Unit Testing in Java

Unit Testing in Java

Using: JUnit[1]

- A Unit Testing framework
- Part of the xUnit family of frameworks (PHPUnit, PyUnit)

It's a framework in that it provides structure for writing your tests.

(1) - https://junit.org/junit4/

Let’s start with an example:

What does a unit test in JUnit look like? 🤔

A minimal test:

```
1  package testpackage;
2
3  import static org.junit.Assert.*;
6
7  public class TestCase {
8
9      @Test
10     public void test() {
11         assertTrue(true);
12     }
13
14 }
```

Note this is a fully functional unit test.

Things to Notice:

Still a regular class:

(line 9) This is new ->

No main method?

```
1  package testpackage;
2
3  import static org.junit.Assert.*;
6
7  public class TestCase {
8
9      @Test
10     public void test() {
11         assertTrue(true);
12     }
13
14 }
```

@Test – An Annotation

Annotations:

- Are metadata, i.e. data about the program.

Recall: @Override - Interfaces, Subclasses

Annotations – Cont’d

The @Test annotation tells JUnit that the following method can be run as a test case.

Annotations can take (optional) arguments:

@Test(expected=…)

@Test(timeout=…)

Annotations – Cont’d

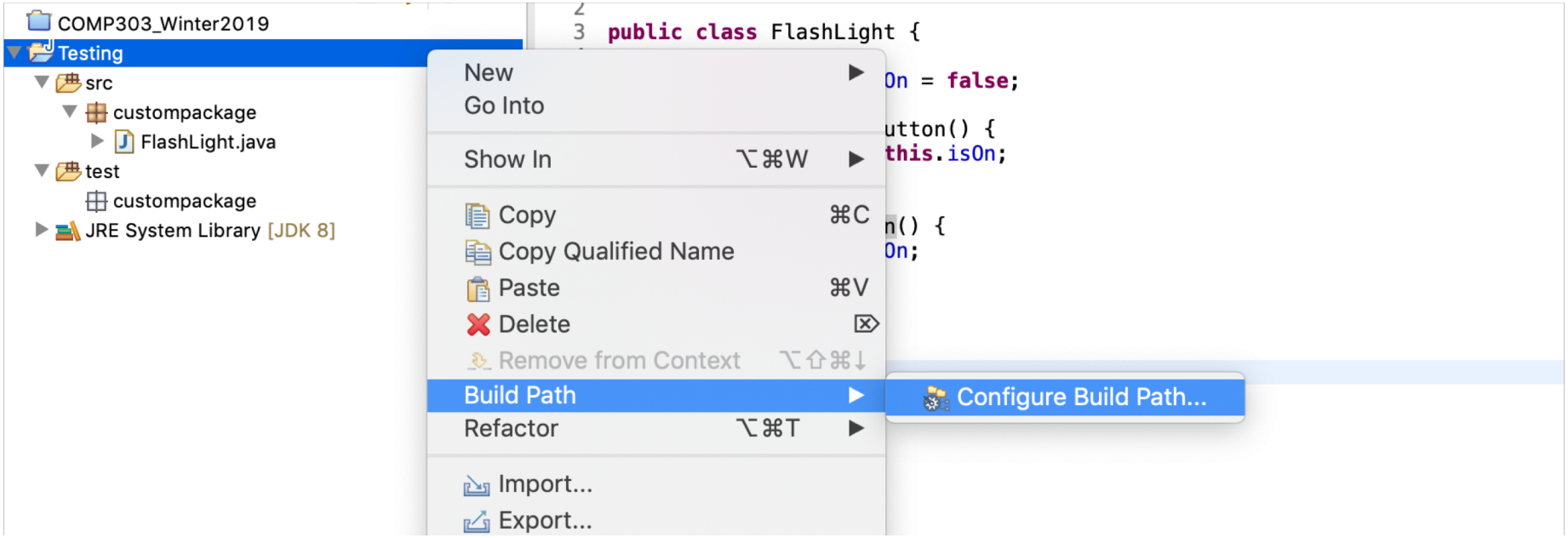
Tutorial on the Oracle documentation website:

https://docs.oracle.com/javase/tutorial/java/annotations/index.html

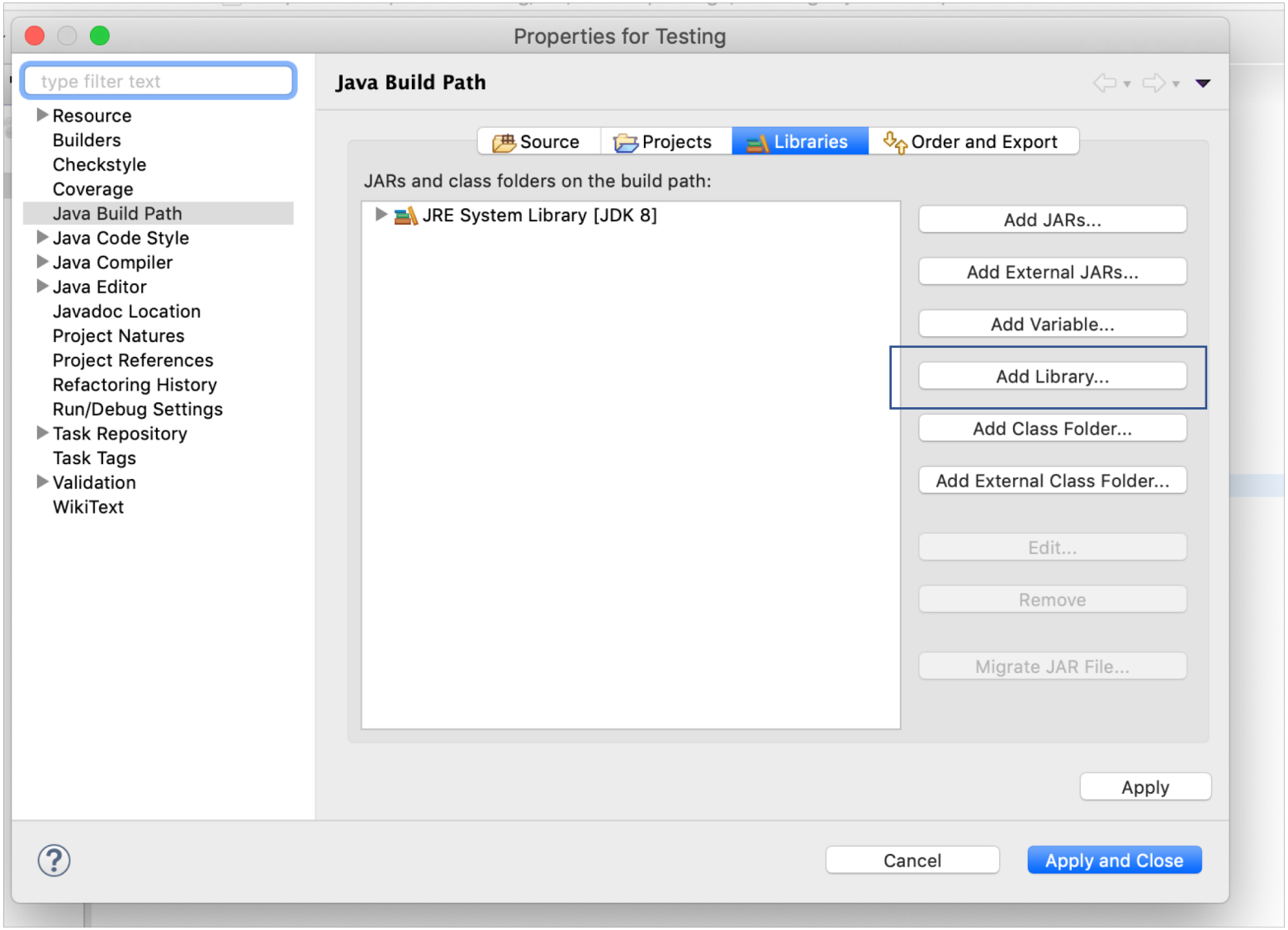
We'll come back to this question:

How does JUnit run tests without a main method?

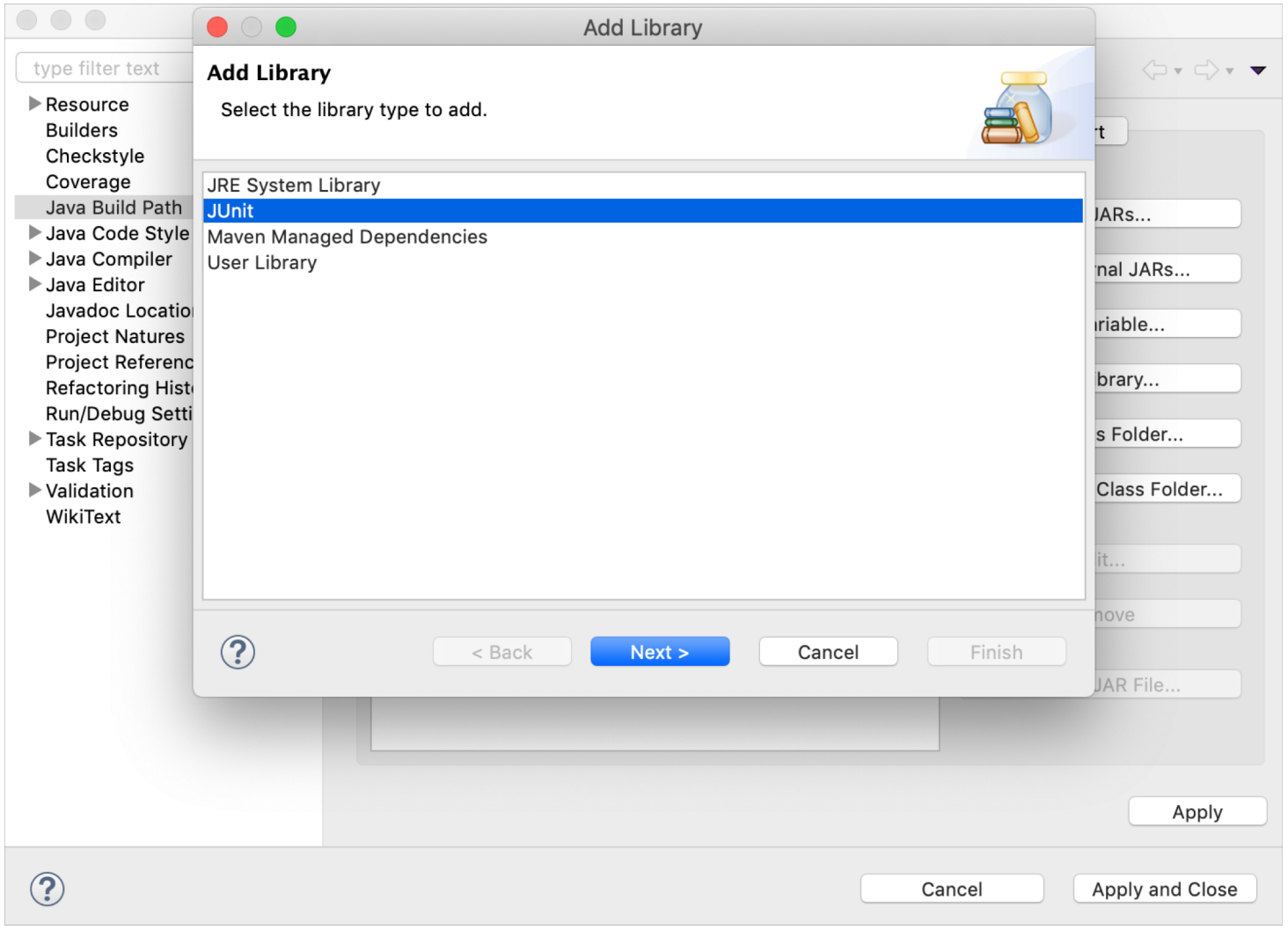
Including JUnit in Eclipse



Including JUnit in Eclipse



Including JUnit in Eclipse



Anatomy of a Unit Test 🔬

Idea: We compare the result of executing a unit under test with an oracle.

Result: Obtained after running some specific code

Unit Under Test: an isolated, usually small part of code

Oracle: The expected result

Flash Light Example

Let's model a (simple) Flash Light 🔦

It has one button that toggles it on and off:

```
1  package custompackage;
2
3  public class FlashLight {
4
5      private boolean isOn = false;
6
7      public void pressButton() {
8          this.isOn = ! this.isOn;
9      }
10
11     public boolean isOn() {
12         return this.isOn;
13     }
14 }
```

Activity

1. Identify UUTs
2. Compare the result of execution with some oracle
3. Write the tests

```
1  package custompackage;
2
3  public class FlashLight {
4
5      private boolean isOn = false;
6
7      public void pressButton() {
8          this.isOn = ! this.isOn;
9      }
10
11     public boolean isOn() {
12         return this.isOn;
13     }
14 }
```

Step 1: Identify UUTs

- pressButton
- The constructor (Technically?) – We'd just like to test the default state of the Flash Light.

Step 2: Compare the result of execution with some oracle.

```
3  public class FlashLight {
4
5      private boolean isOn = false;
6
```

Expect it to be off by default

```
7      public void pressButton() {
8          this.isOn = ! this.isOn;
9      }
```

Expect: it to be on after a press

it to be off after two presses

It to be on after three presses

…

Step 3: Let's write this in JUnit

```
1  package custompackage;
2
3  import static org.junit.Assert.*;
4  import org.junit.Test;
5  import custompackage.FlashLight;
6
7  public class FlashLightTest {
8      @Test
9      public void testOffByDefault() {
10         FlashLight kitchenFlashLight = new FlashLight();
11         assertFalse(kitchenFlashLight.isOn());
12     }
13
14     @Test
15     public void testTurnsOnAfterOnePress() {
16         FlashLight kitchenFlashLight = new FlashLight();
17         kitchenFlashLight.pressButton();
18         assertTrue(kitchenFlashLight.isOn());
19     }
20
21     @Test
22     public void testTurnsBackOffAfterTwoPresses() {
23         FlashLight kitchenFlashLight = new FlashLight();
24         kitchenFlashLight.pressButton();
25         kitchenFlashLight.pressButton();
26         assertEquals(false, kitchenFlashLight.isOn());
27     }
28 }
29
```

What did we observe?

- Independency of Tests
  - JUnit provides no ordering guarantee of any kind for the execution of unit tests
  - Tests need to be independent from one another
  - Example: tests on OnePress and TwoPress need to be two independent and complete tests
- Repeat Initialization
  - FlashLight kitchenFlashLight = new FlashLight()
  - We can improve this...

FlashLightTest Version 2.0

```java
package custompackage;

import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.Test;
import custompackage.FlashLight;

public class FlashLightTest {
    private FlashLight kitchenFlashLight;

    @Before
    public void initialize() {
        kitchenFlashLight = new FlashLight();
        System.out.println("new FlashLight");
    }

    @Test
    public void testTurnsOnAfterOnePress() {
        kitchenFlashLight.pressButton();
        assertEquals(false, kitchenFlashLight.isOn());
    }

    @Test
    public void testTurnsBackOffAfterTwoPresses() {
        kitchenFlashLight.pressButton();
        kitchenFlashLight.pressButton();
        assertEquals(false, kitchenFlashLight.isOn());
    }
}
```

Keypoints in JUnit

- Independency of Tests
  - JUnit provides no ordering guarantee of any kind for the execution of unit tests
  - Tests need to be independent from one another
  - Example: tests on onepress and twopress need to be two independent and complete tests
- @Before runs before every @Test
  - Other useful annotations includes @BeforeClass, @After, @AfterClass

Tests and Exceptional Conditions

Next, we modify the code for FlashLight Class a little...

To add brightness level (0-5), where 0 means off and 5 means brightest

```
1  package custompackage;
2
3  public class FlashLight {
4      private int brightnessLevel = 0;
5
6      public void setBrightnessLevel(int pBrightnessLevel) {
7          if(pBrightnessLevel > 5 || pBrightnessLevel < 0 ) {
8             throw new IllegalArgumentException("BrightnessLevel should in the range of [0-5].");
9          }
10         this.brightnessLevel = pBrightnessLevel;
11     }
12
13     public int getBrightnessLevel() {
14         return brightnessLevel;
15     }
16 }
```

Tests and Exceptional Conditions

```
23      // Method 1:
24      @Test(expected = IllegalArgumentException.class)
25      public void testSetBrightnessLevelException_1() {
26          kitchenFlashLight.setBrightnessLevel(100);
27          assertEquals(2, kitchenFlashLight.getBrightnessLevel());
28      }
29
30      // Method 2:
31      @Test
32      public void testSetBrightnessLevelException_2() {
33          try {
34              kitchenFlashLight.setBrightnessLevel(100);
35              fail();
36          }
37          catch (IllegalArgumentException e) {}
38      }
```

Keypoints in JUnit

- Independency of Tests
  - JUnit provides no ordering guarantee of any kind for the execution of unit tests
  - Tests need to be independent from one another
  - Example: tests on onepress and twopress need to be two independent and complete tests
- @Before runs before every @Test
  - Other useful annotations includes @BeforeClass, @After, @AfterClass
- 2 ways to test exceptions
  - Test annotation parametre
  - Try and catch + fail()

Activity

1. Create a ***PasswordValidator*** Class that checks the password format
   The password has to satisfy the following conditions...
   - Be between 8 and 40 characters long
   - Contain at least one digit.
   - Contain at least one lower case character.
   - Contain at least one upper case character.
   - Contain at least one special character from [@ # $ % ! .].

2. Create unit test for your code

```java
import java.util.regex.*;

public class PasswordValidator {

    private Pattern pattern;
    private Matcher matcher;

    private static final String PASSWORD_PATTERN = "((?=.*[a-z])(?=.*\\d)(?=.*[A-Z])(?=.*[@#$%!]).{8,40})";

    public PasswordValidator() {
        pattern = Pattern.compile(PASSWORD_PATTERN);
    }

    public boolean validate(final String password) {

        matcher = pattern.matcher(password);
        return matcher.matches();

    }
}
```

When testing...

- Test your program and reassure that your regular expression meets the rules on your policy about the form of the passwords.

  For example, you might have a black list of passwords that you don't want to have to your system, You can test your validator against these values to see how it responds.

  Invalid Password

n!k@s	less that 8 characters long
gregorymarjames-law	It doesn't contain an digits or upper case characters
n!koabcD#AX	there should be a digit
ABCASWF2!	there should be a lower case character
n!k@sn1Kosn!k@sn1Kosn!k@sn1Kosn!k@sn1Kos1	need to be less than 40 chars

  Valid Password

  n!k@sn1Kos
  J@vaC0deG##ks
  n!k1abcD#!

```java
@RunWith(Parameterized.class)
public class PasswordValidatorTest {
    private String arg;
    private Boolean expectedValidation;
    private static PasswordValidator passwordValidator;
```

```
18      public PasswordValidatorTest(String str, Boolean expectedValidation) {
19          this.arg = str;
20          this.expectedValidation = expectedValidation;
21      }
```

```
28      @Parameters
29      public static Collection<Object[]> data() {
30          Object[][] data = new Object[][] {
31                  {"n!k@s",false },                           // it's less that 8 characters long
32                  { "gregorymarjames-law", false },           // it doesn't contain an digits or upper case characters
33                  { "n!koabcD#AX", false },                   // there should be a digit
34                  { "ABCASWF2!", false   },                   // there should be a lower case character
35                  { "n!k@sn1Kosn!k@sn1Kosn!k@sn1Kosn!k@sn1Kos1", false   },  // need to be less than 40 chars
36
37                  // valid passwords
38
39                  {"n!k@sn1Kos",true },
40                  { "J@vaC0deG##ks", true },
41                  { "n!k1abcD#!", true } };
42
43          return Arrays.asList(data);
44      }
45
```

```java
@Test
public void test() {
    Boolean res = passwordValidator.validate(this.arg);
    String validv = (res) ? "valid" : "invalid";
    System.out.println("Password "+arg+ " is " + validv);
    assertEquals("Result", this.expectedValidation, res);

}
```

```java
@RunWith(Parameterized.class)
public class PasswordValidatorTest {

    private static PasswordValidator passwordValidator;
    private String arg;
    private Boolean expectedValidation;

    public PasswordValidatorTest(String str, Boolean expectedValidation) {
        this.arg = str;
        this.expectedValidation = expectedValidation;
    }

    @BeforeClass
    public static void initialize() {
        passwordValidator = new PasswordValidator();
    }

    @Parameters
    public static Collection<Object[]> data() {
        Object[][] data = new Object[][] {
                {"n!k@s",false },                          // it's less that 8 characters long
                { "gregorymarjames-law", false },           // it doesn't contain an digits or upper case characters
                { " abcdFg45*", false },                   // characters ~ in not allowed
                { "n!koabcD#AX", false },                  // there should be a digit
                { "ABCASWF2!", false   },                  // there should be a lower case character
                { "n!k@sn1Kosn!k@sn1Kosn!k@sn1Kosn!k@sn1Kos1", false   },  // need to be less than 40 chars

                // valid passwords

                {"n!k@sn1Kos",true },
                { "J@vaC0deG##ks", true },
                { "n!k1abcD#!", true } };

        return Arrays.asList(data);
    }

    @Test
    public void test() {
        Boolean res = passwordValidator.validate(this.arg);
        String validv = (res) ? "valid" : "invalid";
        System.out.println("Password "+arg+ " is " + validv);
        assertEquals("Result", this.expectedValidation, res);

    }
}
```

Reference: https://examples.javacodegeeks.com/core-java/util/regex/matcher/validate-password-with-java-regular-expression-example/

Keypoints in JUnit

- Independency of Tests
  - JUnit provides no ordering guarantee of any kind for the execution of unit tests
  - Tests need to be independent from one another
  - Example: tests on onepress and twopress need to be two independent and complete tests
- @Before runs before every @Test
  - Other useful annotations includes @BeforeClass, @After, @AfterClass
- 2 ways to test exceptions
  - Test annotation parametrs
  - Try and catch + fail()
- Parameterized tests
  - Annotation
  - Constructor
  - List of parameter and expected value

Keypoints in JUnit

- Independency of Tests
- @Before runs before every @Test
- 2 ways to test exceptions
- Parameterized tests

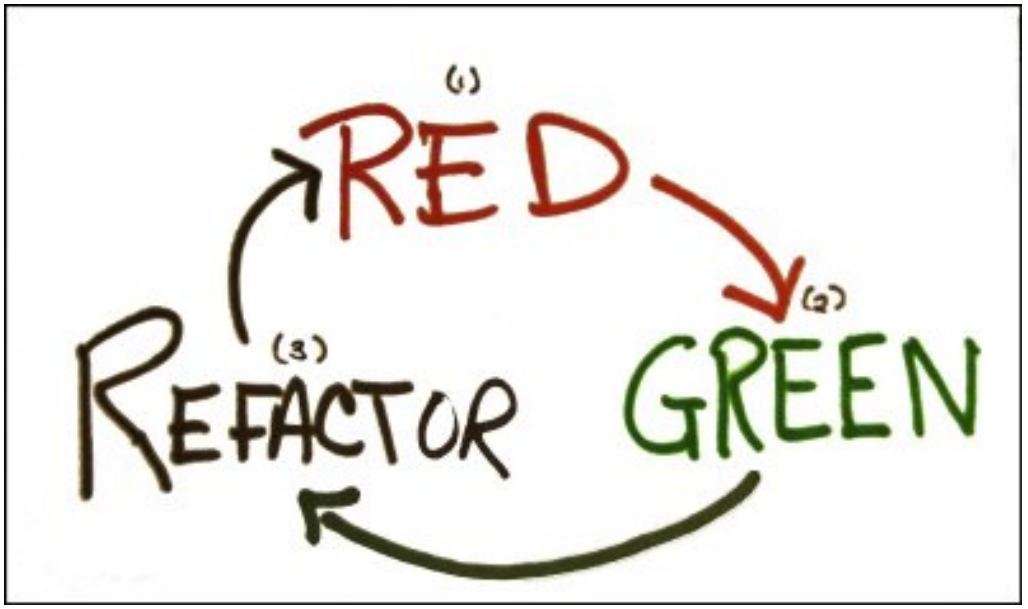
- Much more we will cover in next lecture
- And do not forget to always try things out on your own

Informative tests

```
27  import org.junit.Test;
28
29  public class TestMultiLineString
30  {
31          @Test
32          public void testConstruction()
33          {
34                  MultiLineString string = new MultiLineString();
35                  assertEquals(false, string.isUnderlined());
36                  assertEquals(false, string.isBold());
37                  assertEquals(MultiLineString.CENTER, string.getJustification());
38                  assertEquals("", string.getText());
39
40                  string = new MultiLineString(true);
41                  assertEquals(false, string.isUnderlined());
42                  assertEquals(true, string.isBold());
43                  assertEquals(MultiLineString.CENTER, string.getJustification());
44                  assertEquals("", string.getText());
45          }
46
```

An aside: Test-Driven Development (TDD)

Writing tests before you write the implementation.



https://blog.cleancoder.com/uncle-bob/2014/12/17/TheCyclesOfTDD.html

An aside: Test-Driven Development (TDD)

Writing a failing unit test: RED

Write the minimum code that makes the test pass: GREEN

Rewrite the code so it follows standards, doesn't have code smells, etc: REFACTOR

Reflection

Recall: @RunWith(…)

Reflection lets you examine/modify runtime behavior of your code.

The following class exists: java.lang.Class

Reflection

Very powerful concept.

Through reflection we can invoke methods at runtime *irrespective of the access specifier* used with them (We will see an example next class).

Tutorial:

https://docs.oracle.com/javase/tutorial/reflect/

Recap and Next Class:

Automated Testing

Testing in JUnit

A little on Annotations and Reflection

Recap and Next Class:

- Be able to explain the foundational concepts of testing using the proper terminology;
- Understand type annotations and program reflection and be able to use them effectively;
- Be able to write unit tests with JUnit;
- Be able to approach more advanced testing problems requiring reflection or mock objects;
- Be able to understand the output of a test coverage tool such as EclEmma;
- Be able to understand basic test suite adequacy criteria and the relations between them;

Thanks for attending

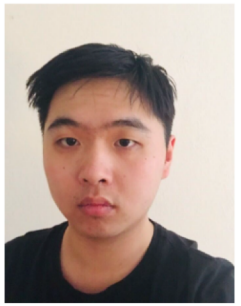
- TA Team

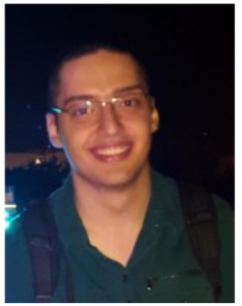

Mathieu Nassif


Deeksha Arya


Alexander Nicholson


Cheryl Wang


Shi Yan Du


Kian Ahrabian