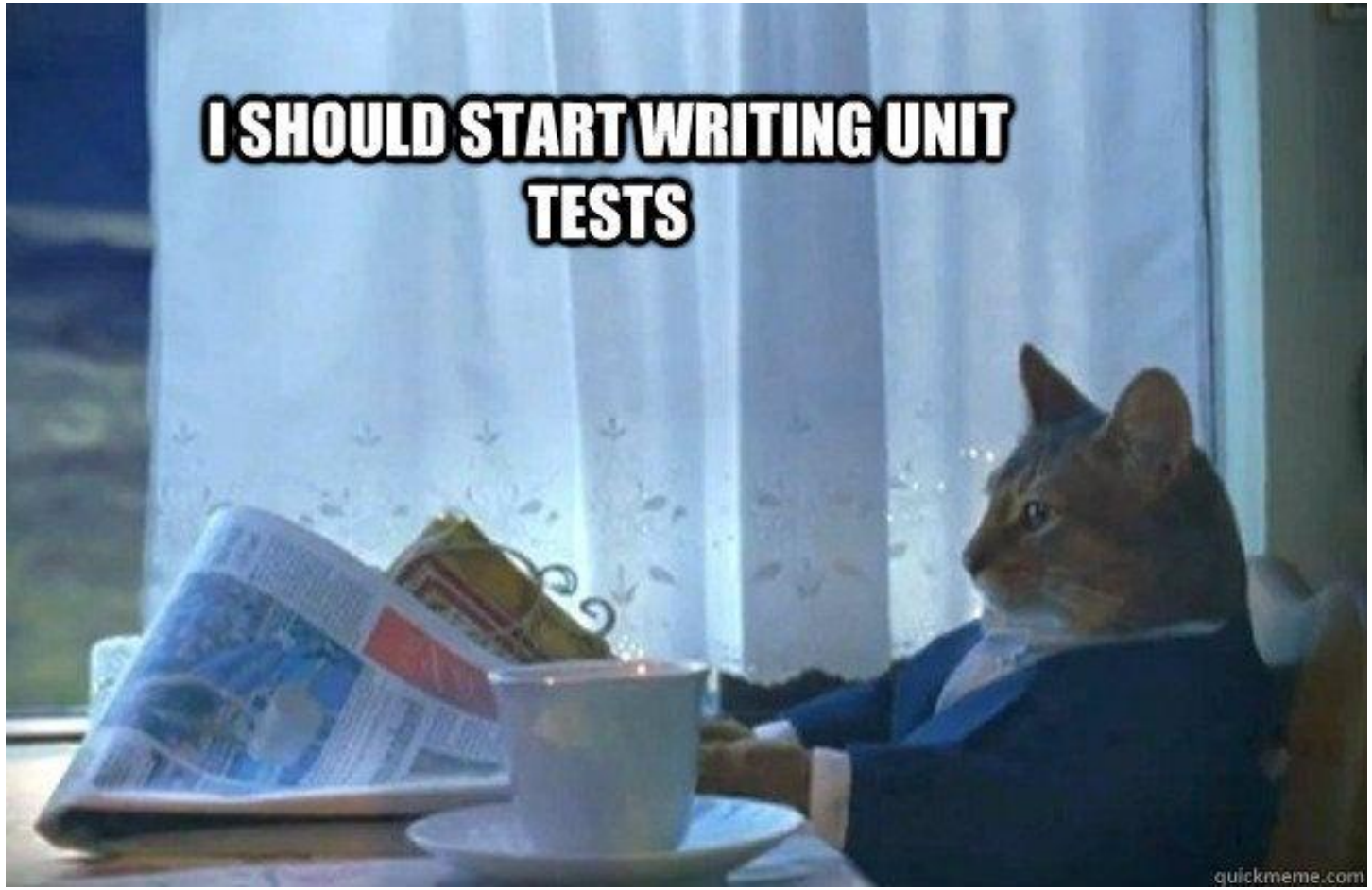
Unit Testing with JUnit



Slides are based on materials from UWashington and NCSU

Review: Test Inputs, Oracles, and Generation

- Formally, a **test case** consists of
  - an **input** (**data**) that is fed to the subject program,
  - an **oracle** (output) that *should be* produced by the program, and
  - a **comparator** that compares actual output against the oracle
- We **automatically generate** high-coverage inputs (and corresponding oracular outputs) with
  - *Path enumeration:* considering various branches of execution through the program
  - *Path predicates*: conditions required to execute a particular branch
  - Mathematical *Constraint solving*: determining required inputs to enforce certain predicates
- Generating test oracles is an **expensive problem**
  - *Invariants*: predicate that is *always* true for all executions of a program (e.g., $x^2 \geq 0 \; \forall x \in \mathbb{R}$)
  - *Mutation*: applying random changes to the program to check invariants
- **Test suite minimization** finds the smallest subset of tests that meet a coverage goal.

Bugs and testing

- **software reliability**: Probability that a software system will not cause failure under specified conditions.
- **Bugs** are inevitable in any complex software system.
- **testing**: A systematic attempt to reveal errors.
  - Failed test: an error was demonstrated.
  - Passed test: no error was found (for this particular situation).

Difficulties of testing

- Limitations of testing:
  - It is impossible to completely test a system.
  - It is ok to give up on some paths
  - It is hard to get testing oracles
  - Testing does not always directly reveal the actual bugs in the code.
  - Testing does not prove the absence of errors in software.
- But:
  - Testing increases confidence that your program works correctly
  - Can be automated (partially)

Unit Testing

- The most basic level of software testing
- Many programming languages provide unit testing framework (xUnit)
- Looking for errors in a subsystem in isolation.
  - Generally a "subsystem" means a particular class or object; Testing the functionality of individual methods
    - Independent paths within the source code
    - Logical decisions as both true and false
    - Loops at their boundaries
    - Internal data structures
    - …

Unit Testing

- The basic idea:
  - For a given class `Foo`, create another class `FooTest` to test it, containing various "test case" methods to run.
  - Each method looks for particular results and passes / fails.
- Testing Strategies
  - Test Requirements
  - Test Boundary Values
  - Test All Paths
  - Test Exceptions
  - …
- JUnit provides "**assert**" commands to help us write tests.
  - The idea: Put assertion calls in your test methods to check things you expect to be true. If they aren't, the test will fail.

Tips for testing

- You cannot test every possible input, parameter value, etc.
  - So you must think of a limited set of tests likely to expose bugs.

- Think about boundary cases
  - positive; zero; negative numbers
  - right at the edge of an array or collection's size

- Think about empty cases and error cases
  - 0, -1, null; an empty list or array

- test behavior in combination
  - maybe `add` usually works, but fails after you call `remove`
  - make multiple calls; maybe `size` fails the second time only

Trustworthy tests

- Test one thing at a time per test method.
  - 10 small tests are much better than 1 test 10x as large.

- Each test method should have few (likely 1) assert statements.
  - If you assert many things, the first that fails stops the test.
  - You won't know whether a later assertion would have failed.

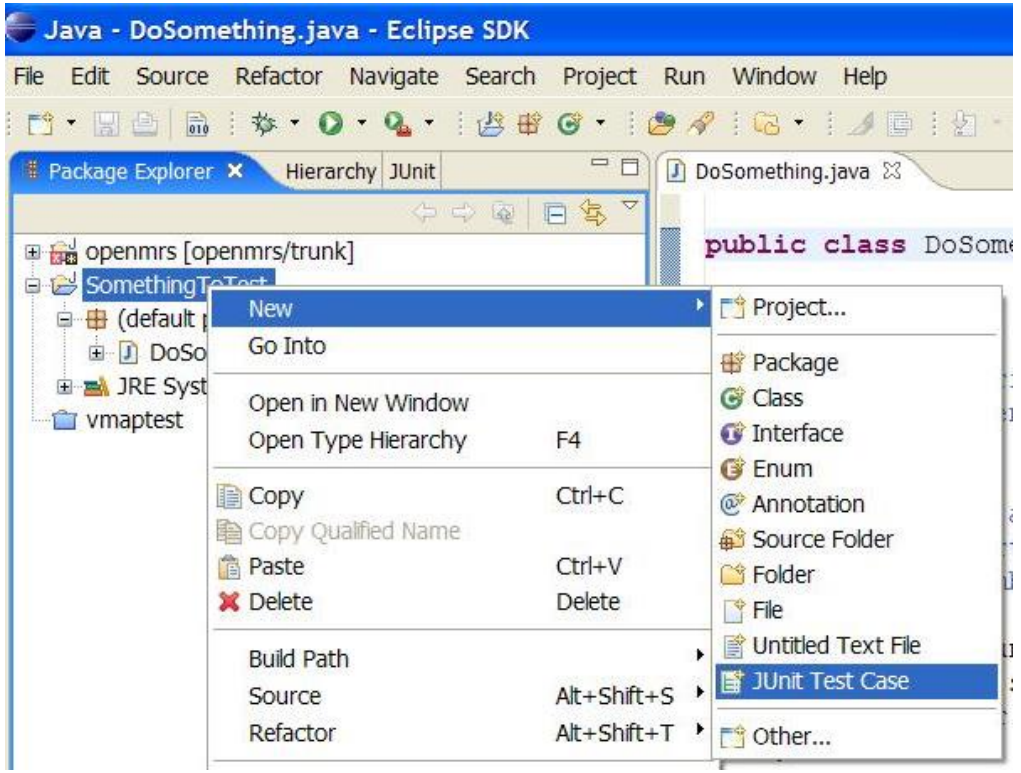
- Tests should avoid logic.
  - minimize `if/else`, `loops`, `switch`, etc.
  - avoid `try/catch`
    - If it's supposed to throw, use `expected=` ... if not, let JUnit catch it.

- Torture tests are okay, but only *in addition to* simple tests.

JUnit and Eclipse

- To add JUnit to an Eclipse project, click:
  - **Project → Properties → Build Path → Libraries → Add Library... → JUnit → JUnit 5 → Finish**

- To create a test case:
  - right-click a file and choose **New → Test Case**
  - or click **File → New → JUnit Test Case**
  - Eclipse can create stubs of method tests for you.



A JUnit test class – Junit 4

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

public class name {
    ...

    @Test
    public void name() {   // a test case
 method
        ...
    }
}
```

- A method with `@Test` is flagged as a JUnit test case.
  - All `@Test` methods run when JUnit runs your test class.

A JUnit test class – Junit 5

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

public class PalindromeTest {

    @Test
    public void testIsPalindrome_valid() {
        assertTrue(Palindrome.isPalindrome(7));
        assertTrue(Palindrome.isPalindrome(11));
        assertTrue(Palindrome.isPalindrome(999));
    }

    @Test
    public void testIsPalindrome_invalid() {
        assertFalse(Palindrome.isPalindrome(10));
    }
}
```

Import JUnit 5 libraries

<SourceCodeClassName>Test.java

test<MethodName>_optionalCondition()

Assert Methods

- Assert methods provide information about the **expected** and **actual values** of a test case
  - assertEquals(expected, actual);
    - For doubles, you will have a third argument, delta
    - Better practice: include an error message in the assertion assertEquals(expected, actual, message);
  - assertTrue(actual);
  - assertFalse(actual);
  - assertNull(actual);
  - assertNotNull(actual);
- Each method can be passed a string to display if it fails:
  - e.g. `assertEquals("message", expected, actual)` - Junit 4
  - e.g. `assertEquals(expected, actual, "message")` - Junit 5

Testing for exceptions – Junit 5

```
@Test
public void whenExceptionThrown_thenAssertionSucceeds() {
    Exception exception = assertThrows(NumberFormatException.class, () -> {
        Integer.parseInt("1a");
    });

    String expectedMessage = "For input string";
    String actualMessage = exception.getMessage();

    assertTrue(actualMessage.contains(expectedMessage));
}
```

Junit 5 Annotations

- @Test
- @ParameterizedTest
  - Run a rest multiple times with difference arguments
  - Must declare at lease one source that will provide the arguments for each invocation and then consume the arguments in the test method

```java
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.ValueSource;

import static org.junit.jupiter.api.Assertions.assertTrue;

class JUnit5Test {

    @ParameterizedTest
    @ValueSource(strings = { "cali", "bali", "dani" })
    void endsWithI(String str) {
        assertTrue(str.endsWith("i"));
    }
}
```

One of the limitations of value sources is that they only support these types:

- *short* (with the *shorts* attribute)
- *byte* (*bytes* attribute)
- *int* (*ints* attribute)
- *long* (*longs* attribute)
- *float* (*floats* attribute)
- *double* (*doubles* attribute)
- *char* (*chars* attribute)
- *java.lang.String* (*strings* attribute)
- *java.lang.Class* (*classes* attribute)

Also, **we can only pass one argument to the test method each time.**

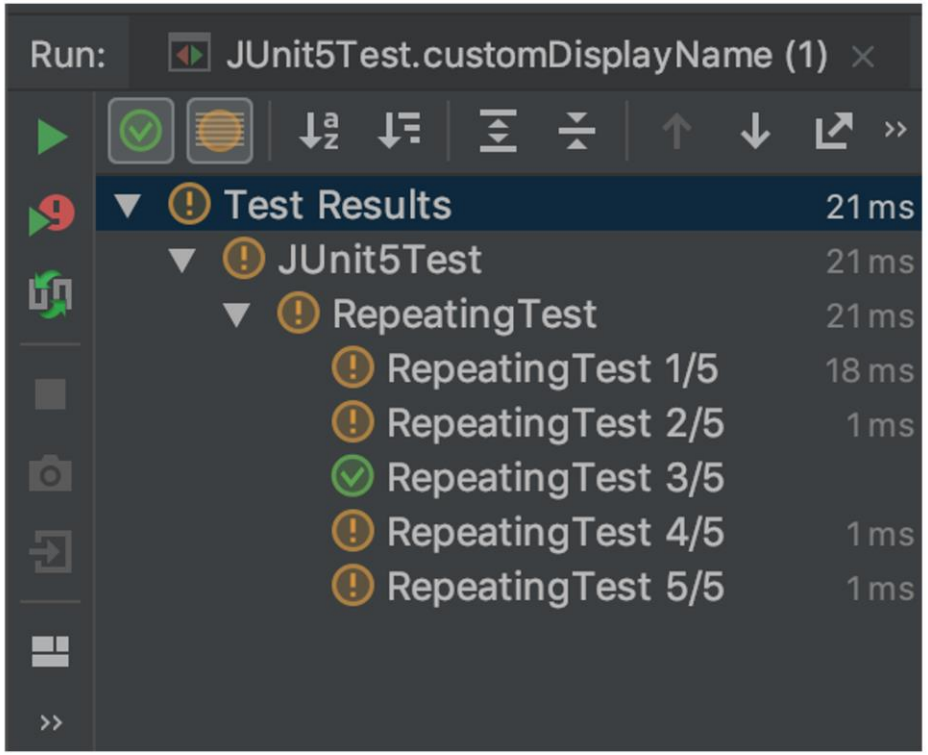
Junit 5 Annotations

- @DisplayName
- @RepeatedTest
  - Repeat a test a specified number of times
  - Each invocation behaves like a regular @Test method

```
class JUnit5Test {

    @RepeatedTest(value = 5, name = "{displayName} {currentRepetition}/{to
    @DisplayName("RepeatingTest")
    void customDisplayName(RepetitionInfo repInfo, TestInfo testInfo) {
        int i = 3;
        System.out.println(testInfo.getDisplayName() +
            "-->" + repInfo.getCurrentRepetition()
        );

        assertEquals(repInfo.getCurrentRepetition(), i);
    }
}
```



Junit 5 Annotations

- @Disabled
  - Skip tests

```java
import org.junit.jupiter.api.Disabled;
import org.junit.jupiter.api.Test;

class DisabledTestsDemo {

    @Disabled
    @Test
    void testWillBeSkipped() {
    }

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

Junit 5 Annotations

- @Timeout -- you should always test with timeout!
  - Default: seconds

```java
@Test
@Timeout(value = 10, unit = TimeUnit.MILLISECONDS)
void usingTimeOutAnnotation(){
    try {
        Thread.sleep(110);
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    }
}

@Test
@Timeout(value = 1) // in seconds
void testDefaultValueForTimeUnit(){
    try {
        Thread.sleep(1001);
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    }
}
```

Thread.sleep() in milliseconds

Junit 5 Annotations

methods to run before/after each test case method is called

methods to run **once** before/after the entire test class runs

JUNIT 4 ANNOTATION	JUNIT 5 ANNOTATION
@Before	@BeforeEach
@After	@AfterEach
@BeforeClass	@BeforeAll
@AfterClass	@AfterAll
@Test	@Test

Demo: Calculator

Demo: Palindrome

- Write a program to test if the input String is a Palindrome in Java. Input can be a Word, Number or even a Phrase.
  - White space - acceptable
  - Punctuation marks - not acceptable
  - Any Case - acceptable

Input : *n = 46355364*

Output: *Reverse of n = 46355364*

Palindrome : *Yes*

Input : *n = 1234561111111111654321*

Output: *Reverse of n = 1234561111111111654321*

Palindrome : *Yes*

JUnit summary

- Tests need *failure atomicity* (ability to know exactly what failed).
  - Each test should have a clear, long, descriptive name.
  - Assertions should always have clear messages to know what failed.
  - Write many small tests, not one big test.
    - Each test should have roughly just 1 assertion at its end.
- Always use a `timeout` parameter to every test.
- Test for expected errors / exceptions.
- Choose a descriptive assert method, not always `assertTrue`.
- Choose representative test cases from equivalent input classes.
- Avoid complex logic in test methods if possible.
- Use helpers, `@BeforeEach @BeforeAll` to reduce redundancy between tests.