Objectives

- Unit Testing

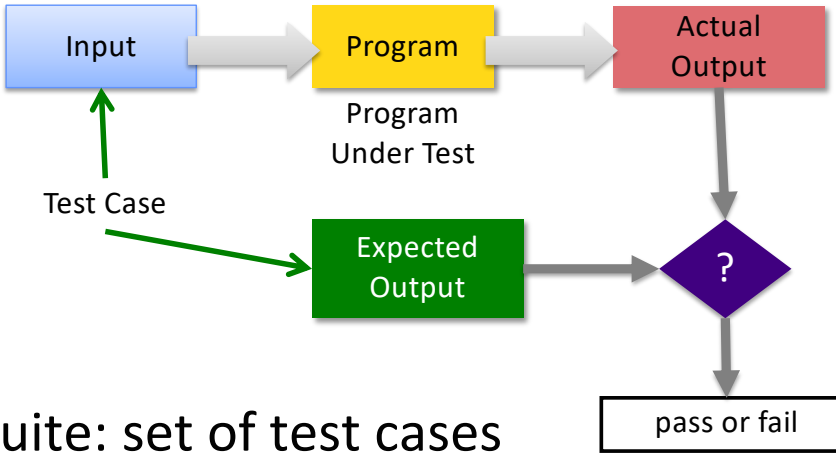
1

Review

1. Describe the general testing process
2. What is a set of test cases called?
3. What is *unit testing*?
4. What are the benefits of unit testing?
5. What are the characteristics of good unit tests?
6. What are the steps in a JUnit Test Case?
   - How do we implement those steps?
7. What is test-driven development?

2

Review: Software Testing Process



- Test Suite: set of test cases

Review: Why Unit Test?

- Verify code works as intended in isolation
- Find defects ***early*** in development
  - Easier to test small pieces
  - Less cost than at later stages (e.g., when integrating)
- Suite of (small) test cases to run after code changes
  - As application evolves, new code is more likely to break existing code
  - Also called **regression** testing

Review: Characteristics of Good Unit Testing

- **Automatic**
  - Since unit testing is done frequently, don't want humans slowing the process down
  - Automate executing test cases and evaluating results
  - Input: in test itself or from a file
- **Thorough**
  - Covers all code/functionality/cases
- **Repeatable**
  - Reproduce results (correct, failures)
- **Independent**
  - Test cases are independent from each other
  - Easier to trace fault to code

Review: Structure of a JUnit Test

1. Set up the test case (optional)
   - Example: Creating objects
   - `@BeforeAll` (once per class), `@BeforeEach` (before each test)
2. Exercise the code under test
   - Within method annotated with `@Test`
3. Verify the correctness of the results
   - Within method annotated with `@Test` – use assert methods
4. Teardown (optional)
   - Example: reclaim created objects
   - `@AfterEach` (after each test), `@AfterAll` (once per class)

Review: Assert Methods

- Defined in `org.junit.jupiter.api.Assertions`
  - Variety of assert methods available
- If fail, throw an error
- Otherwise, test keeps executing
- All are `static void`
- Example: `assertEquals(Object expected, Object actual)`

```
@Test
public void addTest() {
        …
        assertEquals(4, calculator.add(3, 1));
}
```


Review: Example Testing the CD class

```
private CD testCD;

@BeforeEach
public void setUp() {
        testCD = new CD("CD title", "CD Artist", 100, 1997, 11, false);
}

@Test
public void testInCollection() {
    assertFalse( testCD.isInCollection() );
    testCD.setInCollection();
    assertTrue( testCD.isInCollection() );
}
```

Exercising the code and verifying its correctness

Review: Expecting an Exception

- Sometimes an exception *is* the expected result

```
@Test
public void testIndexOutOfBoundsException() {
    List emptyList = new ArrayList();

    assertThrows(IndexOutOfBoundsException.class,
        () -> { Object o = emptyList.get(0); }
    );
}
```

Test case passes only if exception is thrown

Expecting an Exception: Breaking It Down

`assertThrows(Class<T> expectedType, Executable executable)`

```
@Test
public void testIndexOutOfBoundsException() {
    List emptyList = new ArrayList();

    assertThrows(IndexOutOfBoundsException.class,
        () -> { Object o = emptyList.get(0); }
    );
}
```

How to read assertThrows:
Execute the highlighted code (in {})
and check if it throws that exception type

A lot more can be said about lambda expressions… but not in CSCI209

Expecting an Exception

- Can also check characteristics of the thrown exception

```
@Test
public void testIndexOutOfBoundsException() {
  List myList = new ArrayList();
  IndexOutOfBoundsException ioobExc =
        assertThrows(IndexOutOfBoundsException.class, () -> {
                myList.get(0);
        });
  System.out.println(ioobExc.getMessage());
  assertEquals("Index 0 out of bounds for length 0",
        ioobExc.getMessage());
}
```

Test case passes only if exception is thrown *and* message matches

Review: Some Approaches to Testing Methods

- Typical case
  - Test typical values of input/parameters
- Boundary conditions
  - Test at boundaries of input/parameters
  - Many faults live "in corners"
- Parameter validation
  - Verify that parameter and object bounds are documented and checked
  - Example: pre-condition that parameter isn't null

➥ All black-box testing approaches

EVALUATING TEST SUITES

Evaluating Test Suites

- Software testing research question:
  Is my approach to generating a test suite better than the state-of-the-art test suite generation?
- One approach to answer question:
  Fault-based Evaluation
  - Given known faults (a.k.a. mutants)
  - How many faults/mutants does my test suite kill/reveals?
    - *Kill* a fault by creating at least one test case that fails when exercising that fault

Lab: Catching the Mutants

- Objective: Practice writing JUnit test cases
- In Mutant.java, you have the specification for how the method `thirdShortest` *should* work
- Write test cases that test that the method works as expected
- Goal: reveal all the bugs/mutants using test cases!

Lab: Catching the Mutants

- Why designed this way:
  - You get feedback on if you've tested "enough"
  - Practice testing – knowing how much more you need to do
    - Not typically known in the real world!

Lab: Catching the Mutants

- Set Up
  - Jar file (contains mutant class files)
  - Classpath – tell compiler/JVM to use JUnit and mutants.jar

Catching the Mutants: Post-Mortem

- What are the benefits of unit testing/using JUnit?
  - Consider if you were developing/maintaining the method
  - How would your testing/development process change?
- Why did the output come out in strange orders sometimes?
- Is it okay that some mutants passed some of the test cases?
- Recall the characteristics of good unit tests
  - How did you achieve them in your testing?

Are These Effective Tests?

```
@Test
public void testThirdShortest() {
    String[] words = { "a", "ab", "abc" };
    String actual = mutant.thirdShortest(words);
    assertEquals(3, actual.length());
}
```

```
@Test
public void testExceptionThrown() {
    String[] words = { "a" };
    assertThrows(Exception.class, () -> {
        mutant.thirdShortest(words);
    });
}
```


Test Discussion

```
@Test
public void testThirdShortest() {
    String[] words = { "a", "ab", "abc" };
    String actual =
    mutant.thirdShortest(words);
    assertEquals("abc", actual);
}
```

Check the actual result

- They are correct tests
  - They will reveal bugs
- However, they are *weak* tests
  - Cover necessary invariants, but they are not sufficient to expose failures

```
@Test
public void testExceptionThrown() {
    String[] words = { "a" };
    assertThrows(IllegalArgumentException.class,
    () -> {      Expect the exact exception
        mutant.thirdShortest(words);
    });
}
```

Testing More Than One Possible Answer

- `thirdShortest` only returns one answer (a String) but there could be multiple different correct answers
  - We can discuss if this is the best design but …
- Example test

```
@Test
public void testMoreInArray2() {
    String[] words = { "a", "b", "bc", "ab", "bye", "and" };
    String result = mutant.thirdShortest(words);
    assertTrue(result.equals("bye") || result.equals("and"));
}
```

21

21

Is This An Effective Test?

```
@Test
public void testAll() {
     String[][] tests = { { "a", "ab", "abc" },
          { "1", "12", "12345", "12345345", "234oi34iuwer" },
          { "cba", "abc", "bca", "a", "a", "a", "ab", "ab", "ab" } };
     assertEquals(mutant.thirdShortest(tests[0]), "abc");
     assertEquals(mutant.thirdShortest(tests[1]), "12345");
     assertTrue(mutant.thirdShortest(tests[2]).equals("cba") ||
          mutant.thirdShortest(tests[2]).equals("abc") ||
          mutant.thirdShortest(tests[2]).equals("bca"));
     assertThrows(IllegalArgumentException.class, () -> {
          mutant.thirdShortest(null) });
     assertThrows(IllegalArgumentException.class, () -> {
          mutant.thirdShortest(new String[]{}); });
     assertThrows(IllegalArgumentException.class, () -> {
          mutant.thirdShortest(new String[]{ "hey" }); });
     assertThrows(IllegalArgumentException.class, () -> {
          mutant.thirdShortest(new String[]{ "hey", "there" }); });
    String[] words = { "abcds", "b", "bc", "ab", "bye", "and" };
    String[] original = { "abcds", "b", "bc", "ab", "bye", "and" };
    result = mutant.thirdShortest(words);
    assertTrue(result.equals("bye") || result.equals("and"));
    assertEquals(Arrays.asList(words), Arrays.asList(original));
          …
```

22

Is This An Effective Test?

May be effective but hard to use
Tests are not independent
Will be hard to pinpoint bugs

```
@Test
public void testAll() {
     String[][] tests = { { "a", "ab", "abc" },
          { "1", "12", "12345", "12345345", "234oi34iuwer" },
          { "cba", "abc", "bca", "a", "a", "a", "ab", "ab", "ab" } };
     assertEquals(mutant.thirdShortest(tests[0]), "abc");
     assertEquals(mutant.thirdShortest(tests[1]), "12345");
     assertTrue(mutant.thirdShortest(tests[2]).equals("cba") ||
          mutant.thirdShortest(tests[2]).equals("abc") ||
          mutant.thirdShortest(tests[2]).equals("bca"));
     assertThrows(IllegalArgumentException.class, () -> {
          mutant.thirdShortest(null) });
     assertThrows(IllegalArgumentException.class, () -> {
          mutant.thirdShortest(new String[]{}); });
     assertThrows(IllegalArgumentException.class, () -> {
          mutant.thirdShortest(new String[]{ "hey" }); });
     assertThrows(IllegalArgumentException.class, () -> {
          mutant.thirdShortest(new String[]{ "hey", "there" }); });
    String[] words = { "abcds", "b", "bc", "ab", "bye", "and" };
    String[] original = { "abcds", "b", "bc", "ab", "bye", "and" };
    result = mutant.thirdShortest(words);
    assertTrue(result.equals("bye") || result.equals("and"));
    assertEquals(Arrays.asList(words), Arrays.asList(original));
          …
```

23

Guidance for Writing Tests

- Group tests in methods, classes
  - Class could be by behavior, by error conditions, …
- Test methods should focus on one behavior
  - If test case fails, should be helpful in narrowing down where the problem is
- See examples on course schedule

24

Review: Test-Driven Development

- A development style, evolved from Extreme Programming
- Idea: write tests first *without code bias*
- The Process:
  1. Write tests that code/new functionality should pass
     - Like a specification for the code (pre/post conditions)
     - All tests will initially *fail*
  2. Write the code and verify that it passes test cases
     - Know you're done coding when you pass **all** tests

How do you know you're "done" in traditional development?

What assumption does this make?

Project: Test-Driven Development

- Given: a `Car` class that only has enough code to compile
- Your job: Create a ***good*** set of test cases that ***thoroughly***/***effectively*** test `Car` class
  - Find faults in my faulty version of `Car` class
  - Start: look at code, think about how to test, set up JUnit tests
  - Written analysis of process
- First team project: teams of **3**
  - Practice collaboration
  - Every student must commit code to the repository
- First step: create teams (and *team names!*) today
  - Due before 10 a.m. tomorrow

Looking Ahead

- Testing Project due next Wednesday before class
  1. THINK
  2. DISCUSS as a team
  3. Then write the tests
- Teams finalized tomorrow
- Lab was an in-class exercise
  - Practice JUnit testing before project