Objectives

- Testing
- Collaboration

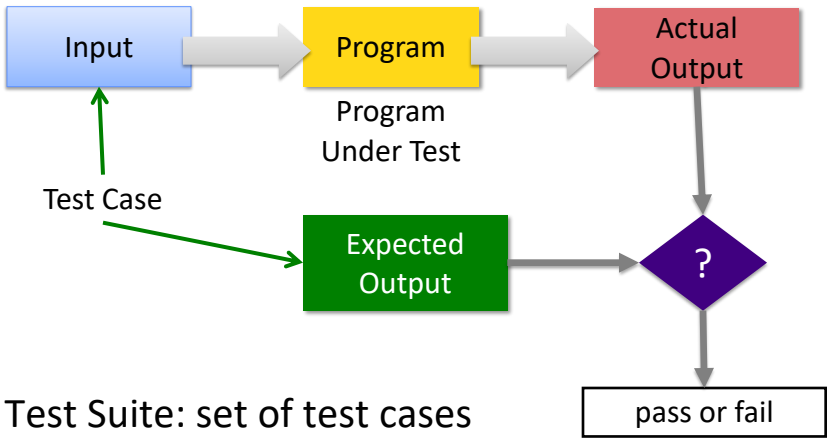
1

Review

1. What tells the compiler/JVM where to find classes?
2. How can we package up Java classes for distribution?
3. Describe the general testing process
4. What is a set of test cases called?
5. What is *unit testing*?
6. What are the benefits of unit testing?
7. What are the characteristics of good unit tests?
8. What are the steps in a JUnit Test Case?
   - How do we implement those steps?
9. 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
- As application evolves, new code is more likely to break existing code
  - Suite of (small) test cases to run after code changes
  - Also called **regression** testing

Review: Characteristics of Good Unit Testing

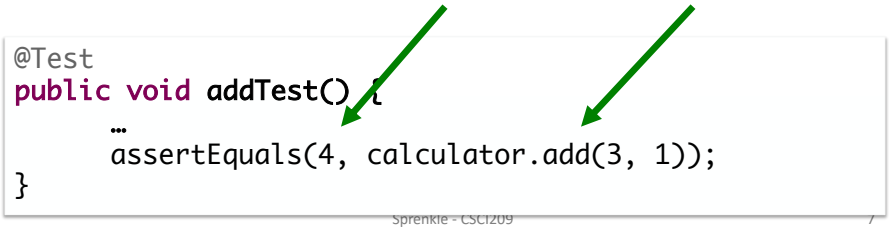
- **Automatic**
  - Since unit testing is done frequently, don't want humans slowing the process down
  - Automate executing test cases and evaluating results
  - Input: in test itself or from a file
- **Thorough**
  - Covers all code/functionality/cases
- **Repeatable**
  - Reproduce results (correct, 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 @Test method
3. Verify the correctness of the results
   - Within @Test method – 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 `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

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); }
    );
}
```

Example of a *Lambda expression*

How to read assertThrows:
Execute the executable (after the first ,)
and check if it throws an exception of that type (before the ,)

Expecting an Exception: Breaking It Down (2)

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 now

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

Tracking Down Problems with Lab

- JUnit 5 is backwards-compatible
  - But, can't use parts of JUnit 4 and JUnit 5
  - All one or the other (ish …)
- Why did my code work but yours didn't?
  - 2 repositories of code
    1. My "source" repo
    2. Template repo for your version
- Spent 1+ hour tracking down that I needed to change

```
import org.junit.Test; (JUnit 4) to
import org.junit.jupiter.api.Test;
(JUnit 5)
```


Lab: Catching the Mutants

(~10 more minutes)

- Objective: Practice writing JUnit test cases
- Goal: reveal all the bugs/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!

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?

15

Characteristics of Good Unit Testing

- **Automatic**
- **Thorough**
- **Repeatable**
- **Independent**

Why are these characteristics of good (unit) testing?

16

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: Test-Driven Development

- A development style, evolved from Extreme Programming
- Idea: write tests first *without code bias*
- The Process:

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

  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

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
- Due before class Monday, Oct 12
  - First step: create teams (and *team names!*) today

How to Implement an Effective Solution

- Understand the problem
- Understand external constraints
- Design an effective solution to the problem
- While designing the solution, design some ***tests*** to verify that the problem is solved (and remains solved)
- Code the effective solution to the problem
- Teach other team members about your solution to the problem

How to Implement an Effective Solution

- Understand the problem (interact with ***people***)
- Understand external constraints (interact with ***people***)
- Design an effective solution to the problem
- While designing the solution, design some ***tests*** to verify that the problem is solved (and remains solved)
- Code the effective solution to the problem
- Teach other team members about your solution to the problem (interact with ***people***)

Probably interaction with people while designing, testing, and coding too

21

21

Collaboration: Team Project

- Need to talk about the solution
- Discuss your plan, e.g.,
  - Your system for testing to make sure that you test everything
  - Your assumptions about the Car class
  - Organization of test cases
  - Naming
  - Division of labor
- Maintain planning documents too
  - in GitHub or elsewhere

22

Collaboration: Team Project

- Version Control does not eliminate need for communication
  - Process becomes much more difficult if developers do not communicate
- Keep the version to be graded in `master` branch
- Before picking up again, **pull** the repository
  - Get others' changes

Collaboration: Workflow

1. Create a branch for your work
   - Commit periodically
   - Write descriptive comments so your team members know what you did and why
2. Switch back to master
3. Pull master branch
4. Merge your branch into the master branch
   - Handle merge conflicts
   - Commit
5. Push master branch

Looking Ahead

- Testing Project due next Monday
  1. THINK
  2. DISCUSS as a team
  3. Then write the tests
- Teams finalized today