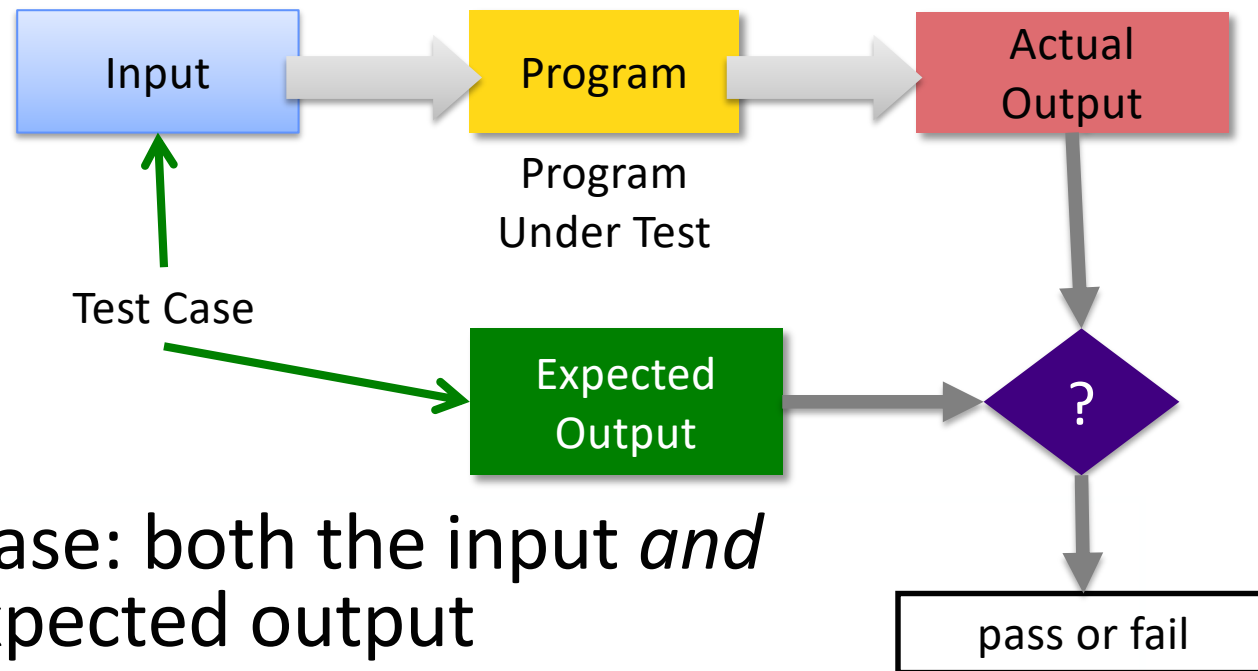


# Objectives

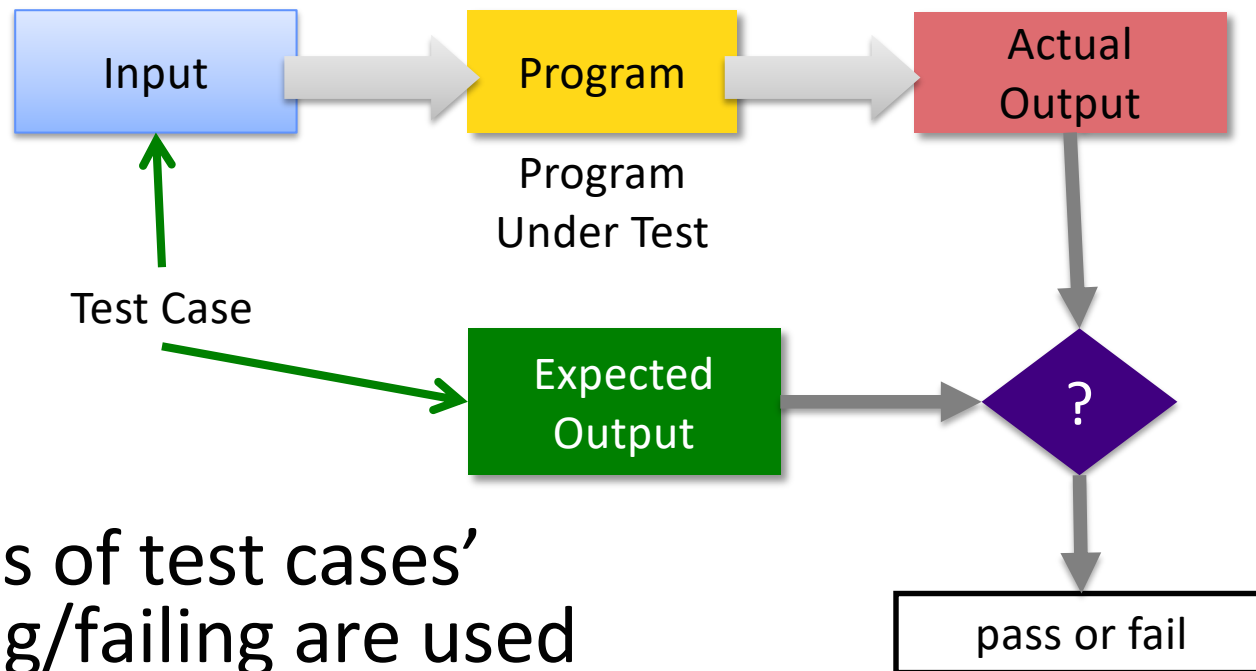
- Testing Overview
- Unit Testing
- JUnit

# Review: Software Testing Process



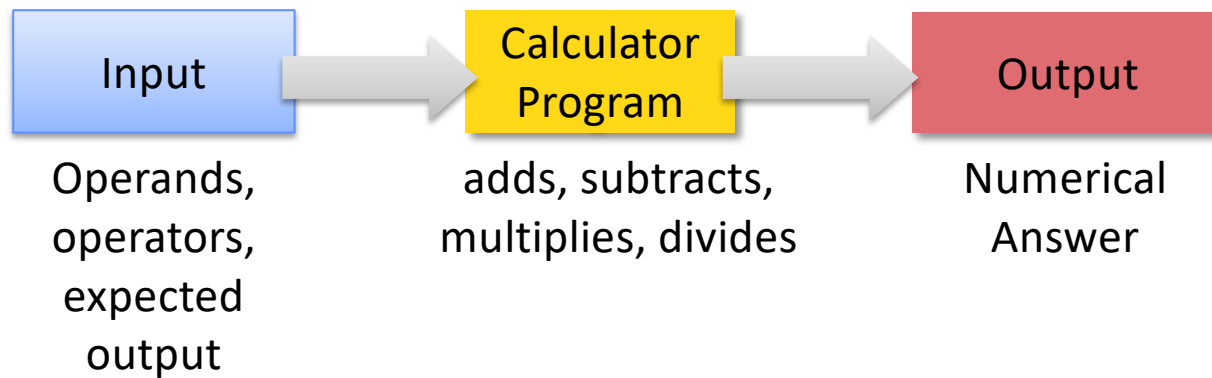
- Test case: both the input *and* the expected output
- Test Suite: set of test cases

# Review: Software Testing Process



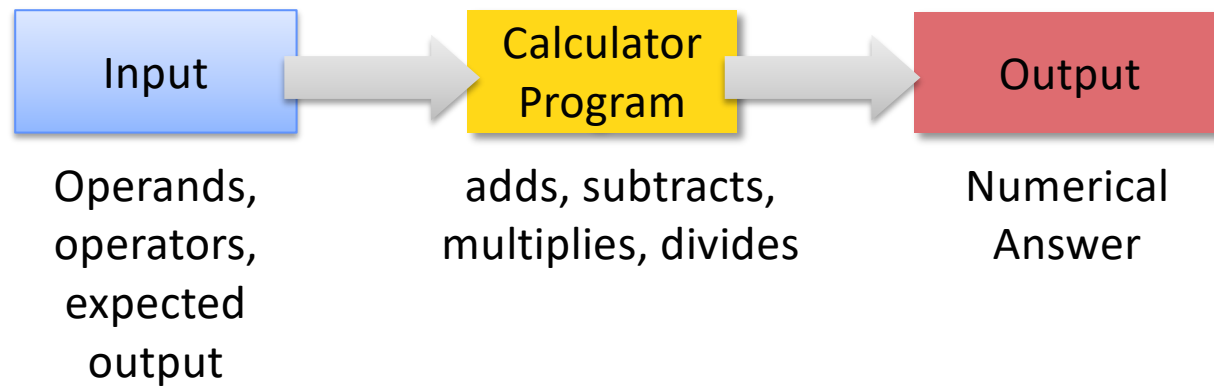
- Results of test cases' passing/failing are used in the subsequent step: ***debugging***

# How Would You Test a Calculator Program?



**What test cases?**  
**Provide both input and expected output**

# Example Calculator Test Cases



Operation	Input	Expected Output
Add	1, 1	2
Add	1, -1	0
Add	1.5, 0	1.5
...		

# Software Testing Questions

- How should you test? How often?

- Code may change frequently
- Code may depend on others' code
- A lot of code to validate

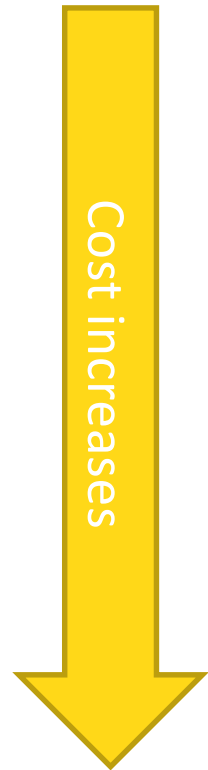
- How do you know that an output is correct?

- Complex output
- Human judgment?

➔ Need a *systematic, automated, repeatable* approach

- What caused a code failure?

# Levels of Testing



- Unit
  - Tests minimal software component, in isolation
  - For us, Class-level testing
  - Web: Web pages (Http Request)
- Integration
  - Tests interfaces & interaction of classes
- System
  - Tests that completely integrated system meets requirements
- System Integration
  - Test system works with other systems, e.g., third-party systems



# UNIT TESTING

# Unit Testing

- Tests minimal software component, in isolation
- For us, Class-level testing
- Web: Web pages (Http Request)

# 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

# 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

# Approaches to Testing

## Traditional Approach

1. Write code
2. Write tests of code
  - May need to update code to make sure they all pass

## Test-Driven Development

1. Write tests that correctly functioning code must pass
2. Write code

Consider tradeoffs of approaches

- When do you know you're done?
- What assumptions are you making?

## Another Use of Unit Testing:

### Test-Driven Development (TDD)

- 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

## Another Use of Unit Testing:

### Test-Driven Development (TDD)

- 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?

# Characteristics of Good Unit Testing

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

STOP: Why are these characteristics of good (unit) testing?

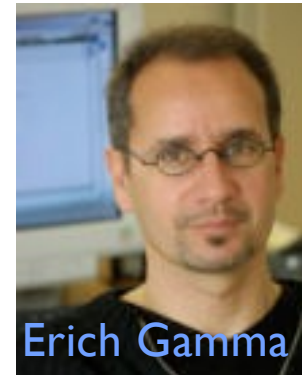
# Characteristics of Good Unit Testing

- **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 (Comprehensive)**
  - Covers all code/functionality/cases
  - Verification of results
- **Repeatable**
  - Reproduce results (correct, failures)
- **Independent**
  - Test cases are independent from each other and from other code
  - Easier to trace failure to code

**JUNIT**

# JUnit Framework

- A framework for unit testing Java programs
  - Supported by Eclipse and other IDEs
  - Originally developed by Erich Gamma and Kent Beck
- Functionality
  - Write tests
    - Validate output, automatically
  - Automate execution of test suites
  - Display pass/fail results of test execution
    - Stack trace where fails
  - Organize tests, separate from code
- But, you still need to come up with the tests!



# Testing with JUnit

- Typical organization:

- Set of testing classes

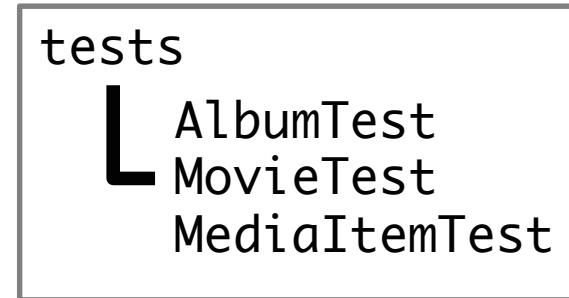
- Testing classes packaged together in a **tests** package

- Separate package from code testing

- A test class typically

- Focuses on a specific class

- Contains methods, each of which represents another test of the class



# Structure of a JUnit Test

1. Set up the test case (optional)

➤ Example: Creating objects

2. Exercise the code under test

3. Verify the correctness of the results

4. Teardown (optional)

➤ Example: reclaim created objects

# Annotations

- Testing in JUnit 5: uses *annotations*
- Provide information about a program that is not part of program itself
- Have no direct effect on operation of the code
  - But compiler or tools may use them
- Example uses of annotations:
  - `@Override`: method declaration is intended to override a method declaration in parent class
    - If method does not override parent class method, compiler generates error message
  - Information for the compiler to suppress warnings (`@SuppressWarnings`)

# Creating Tests

- Tests are contained in *classes*
- The class is named for the functionality you're testing
- Typically located in a separate package named **tests**

```
package edu.wlu.cs.calculator.tests;  
  
public class CalculatorTest {  
    This class contains tests for the calculator  
}
```

# Methods are Test Cases

- Mark your testing method with `@Test`
  - From `org.junit.jupiter.api.Test`

```
public class CalculatorTest {  
    @Test  
    public void testAdd() {  
        ...  
    }  
}
```

Class for testing the  
Calculator class

A method to test the  
“add” functionality


- Convention: Method name describes what you’re testing

# Assert Methods

Defined in  
org.junit.jupiter.api.Assertions

- Used to verify that execution results are what you expect
- 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 testAdd() {
    ...
    assertEquals(4, calculator.add(3, 1));
}
```



# Assert Methods

- To use asserts, need *static* import:

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

➤ **static** allows us to not have to use classname when calling method

- More examples

➤ `assertTrue(boolean condition)`

➤ `assertSame(Object expected, Object actual)`

- Refer to same object

➤ `assertEquals(double expected, double actual, double delta)`

- Doubles are equal within a delta

# Example Uses of Assert Methods

```
@Test
public void testEmptyCollection() {
    Collection collection = new ArrayList();
    assertTrue(collection.isEmpty());
}
```

```
assertEquals(double expected, double actual, double delta)
```

```
@Test
public void testPI() {
    final double ERROR_TOLERANCE = .01;
    assertEquals(Math.PI, 3.14, ERROR_TOLERANCE);
}
```


Test will fail if ERROR\_TOLERANCE = .001

# Set Up/Tear Down

- May want methods to set up objects for every test in the class
  - Called **fixtures**
  - If have multiple, no guarantees for order executed

```
@BeforeEach  
public void prepareTestData() { ... }  
  
@BeforeEach  
public void setupMocks() { ... }  
  
@AfterEach  
public void cleanupTestData() { ... }
```

Executed before  
**each** test method



# Example Set Up Method

```
private Album testAlbum;
```

← Declare the instance variable

```
@BeforeEach
```

```
public void setUp() {
```

```
    testAlbum = new Album("Album title", "Artist",  
                           100, 1997, 11);
```

```
}
```

@BeforeEach Executed before **each** test method

- Can use `testAlbum` object in test methods
- Helps make test methods *independent*
  - Changes to instance variable in one test method do not affect the other test methods

# Example: Testing the Album class

```
private Album testAlbum;    1. Declare the instance variable

@BeforeEach
public void setUp() {
    testAlbum = new Album("Album title", "Artist",
                          100, 1997, 11);
}    2. Instantiate the instance variable before every test

@Test
public void testDefaultConstructor() {
    // can use testAlbum in here
    assertEquals(11, testAlbum.getNumTracks());
    assertEquals(1997, testAlbum.getCopyrightYear());
    assertTrue(testAlbum.isInCollection());
    ...    3. Use the instance variable in your test methods
}
```

# Example: Testing the Album class

```
private Album testAlbum;

@BeforeEach
public void setUp() {
    testAlbum = new Album("Album title", "Artist",
        100, 1997, 11);
}

@Test
public void testInCollection() {
    assertTrue(testAlbum.isInCollection() );
    testAlbum.checkOutOfCollection();
    assertFalse(testAlbum.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 in CSCI209

## Expecting an Exception – More!

- Can also check characteristics of the thrown exception

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

Above test case passes only if exception is thrown  
*and* message matches

# Expecting an Exception: Birthday

```
class BirthdayTest {  
  
    private Birthday bday;  
  
    @BeforeEach  
    void setUp() throws Exception {  
        bday = new Birthday();  
    }  
  
    @Test  
    void testSetBirthday() {  
        IllegalArgumentException iaEx =  
            assertThrows(IllegalArgumentException.class, () -> {  
                bday.setBirthday(0, 1);  
            });  
        assertEquals("Month must be between 1 and 12, inclusive",  
            iaEx.getMessage());  
    }  
}
```

## Set Up/Tear Down For Test Class

- May want methods to set up objects for set of tests
  - Executed *once* before any test in class executes

```
@BeforeAll
public static void
setupDatabaseConnection() { ... }

@AfterAll
public static void
teardownDatabaseConnection() { ... }
```

# JUnit Examples

- Check out the examples of testing the Chicken and Birthday classes

[https://cs.wlu.edu/~sprenkle/cs209/examples/junit\\_testing/code.html](https://cs.wlu.edu/~sprenkle/cs209/examples/junit_testing/code.html)

# Writing Good Test Cases

- A test method should focus on one behavior
  - If test case fails, the test case should be helpful in narrowing down where the problem is
- Use assert statements well to verify the results are what you expect
  - May use multiple asserts to verify one result
- Testing isn't typically "creative" and doesn't need to be generalizable
  - Code should be straightforward
- See examples linked from course schedule page

# Unit Testing & JUnit Summary

- Unit Testing: testing smallest component of your code
  - For us: class and its methods
- JUnit provides framework to write test cases and run test cases automatically
  - Easy to run again after code changes

# JUNIT IN ECLIPSE

# Using JUnit in Eclipse

- Eclipse can help make our job easier
  - Automatically create test templates
  - Automatically execute tests (i.e., methods)
- We can focus on coming up with good tests

# Using JUnit in Eclipse: Creating a New Test Class

- In Eclipse, go to your Assignment4 project
- Create a new JUnit Test Case (under Java)
  - Select **JUnit Jupiter test**
    - When prompted, add JUnit to build path
  - Put in package `edu.wlu.cs.username.tests`
  - Name: `MovieTest`
  - Choose to test `Movie` class
    - Select `setUp` and `tearDown`
    - Select methods to test
- Run the class as a JUnit Test Case

# Using JUnit in Eclipse: Creating a New Test Class

- Alternatively...
- Right-click on the class you want to test (e.g., Movie)
- Select New → JUnit Test Case
  - Select JUnit Jupiter test
    - When prompted, add JUnit to build path
  - Put in package `edu.wlu.cs.username.tests`
  - Name: `MovieTest`
  - Movie should already be selected as “Class under test”
    - Select `setUp`
    - Select methods to test
- Run the test class as a JUnit Test Case

# Example

- Create a test method that tests the method that gets the length of the Movie
  - Revise: Add code to `setUp` method that creates a Movie object and use that object in your test
- Notes
  - Replaying all the test cases: right click on tests package
  - FastView vs Detached
  - Hint: CTL-Spacebar to get auto-complete options

## Got It? Good!

- Take the reading quiz on Canvas!
  - You can/should refer back to the slides