Module 1-14

Unit Testing

Objectives

- The SDLC (Software Development Life Cycle)
- What is testing?
- Exploratory vs. Regression testing
- Manual vs. Automated testing
- Unit, Integration and Acceptance testing
- How to write unit tests

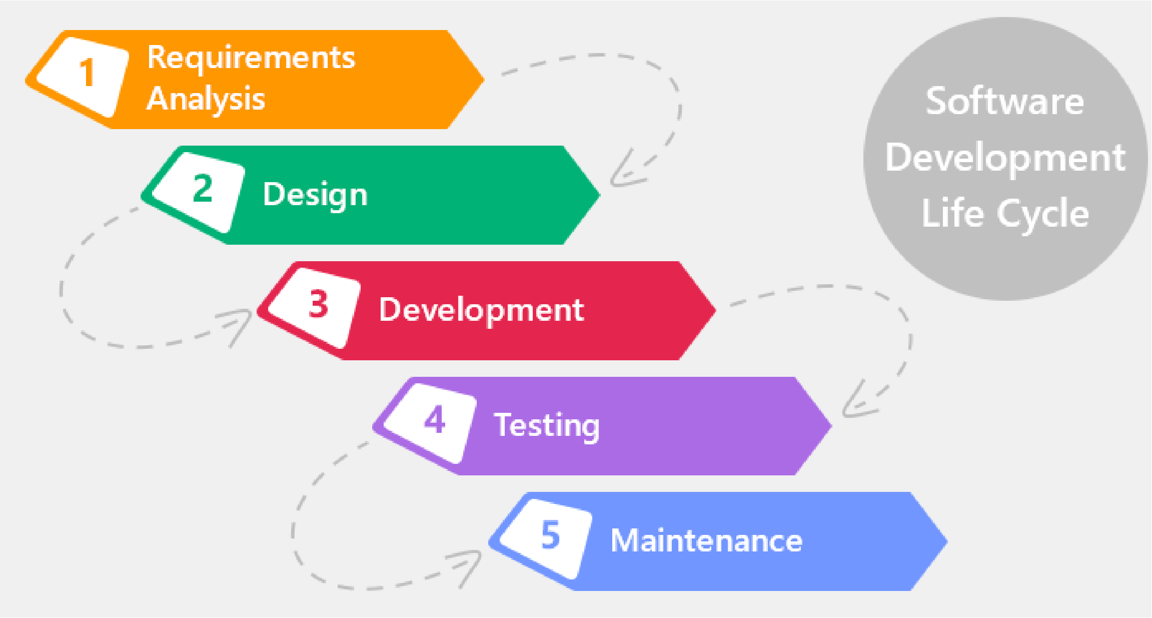
Testing

Goes without saying… we need a way to test the code we’ve written.

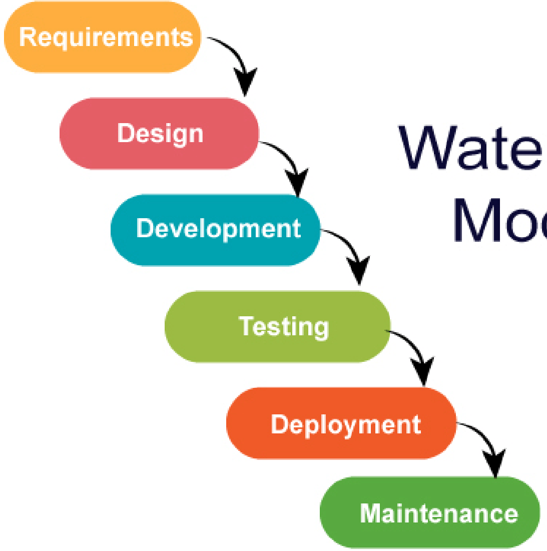
But first, let’s talk about the SDLC (Software Development Life Cycle)

Software Development Life Cycle (SDLC)

The Software Development Life Cycle (SDLC) is a structured process that enables the production of high-quality, low-cost software, in the shortest possible production time.



Waterfall Model



Requirements

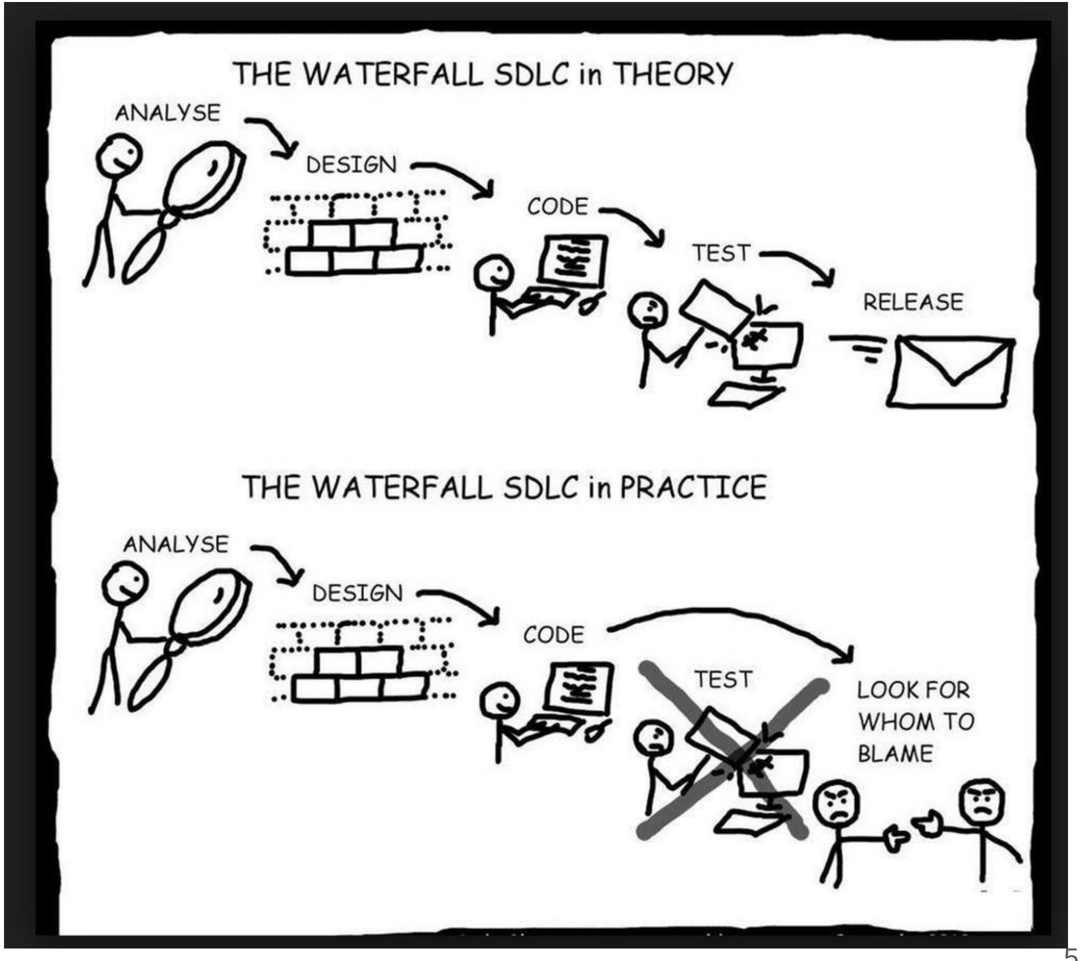
Design

Development

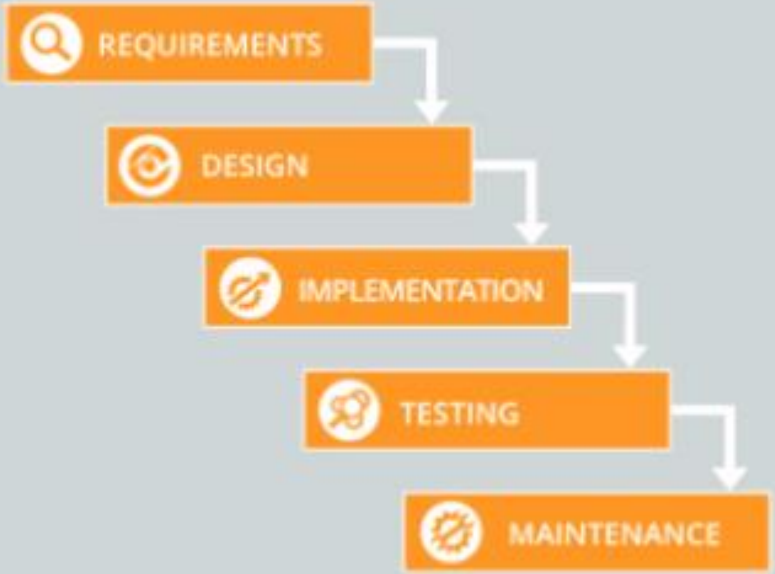
Testing

Deployment

Maintenance



Waterfall vs. Agile



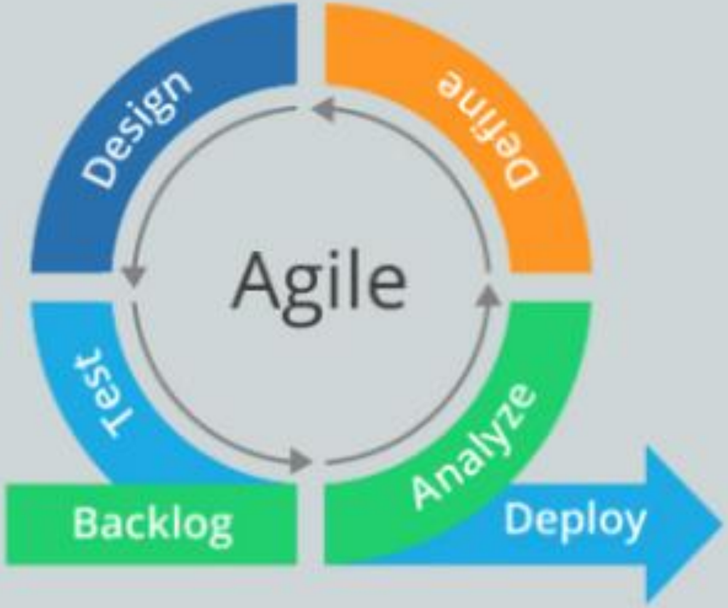
REQUIREMENTS

DESIGN

IMPLEMENTATION

TESTING

MAINTENANCE



Agile

Define

Design

Test

Analyze

Backlog

Deploy

SOFTWARE TESTER
What my friends think I do
What my mom thinks I do
HACKED
SITE DOWN
What society thinks I do
What programmers think I do
Bug Free Software
What I think I do
What I actually do

I've executed all the tests so we have 100% coverage now.
Pinocchio never understood why his nose grew whenever he talked about coverage
AG
Andy Glover cartoontester.blogspot.com Copyright 2011

A software tester walks into a bar and then runs into the bar... strolls into the bar... gallops into the bar... saunters into the bar...
by Jasmine Harpley
www.ministryoftesting.com

Exploratory Testing vs Regression Testing

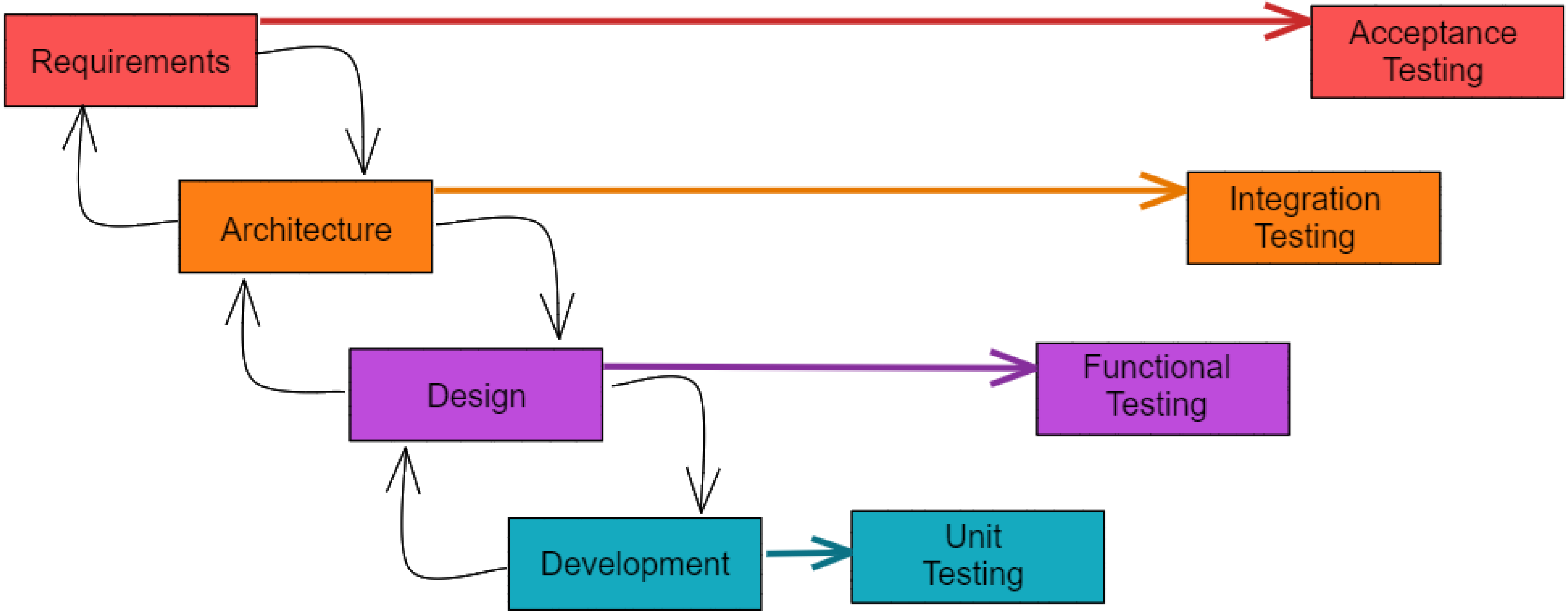
- **Exploratory Testing** explores the functionality of the system looking for defects, missing features, or other opportunities for improvement. Almost always manual.

- **Regression Testing** validates that existing functionality continues to operate as expected.

Manual Testing vs Automated Testing

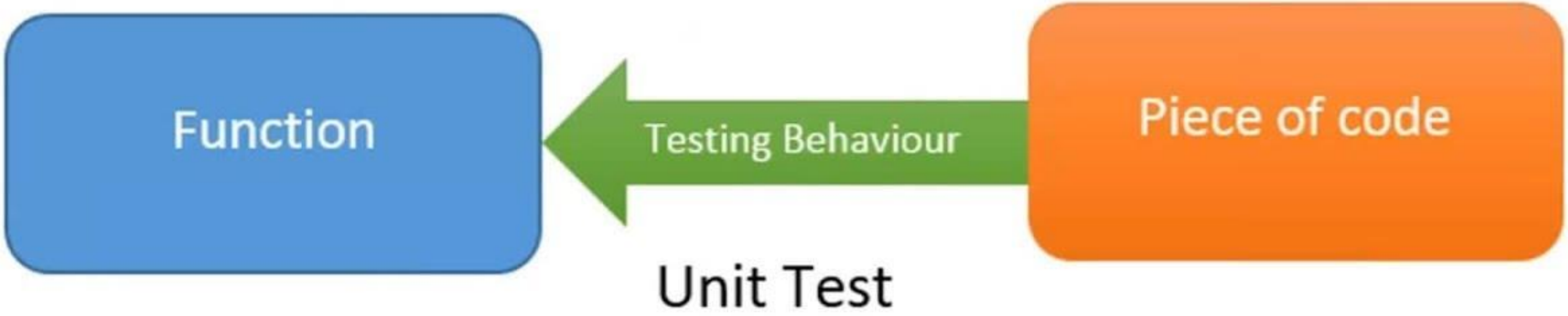
- Historically, tests were written on a third party tool (i.e. Excel) with a script a tester should follow. The results are recorded.
    - This is a very error prone manual process.
    - Exploratory and Usability testing are best done manually.

- Over time, testing frameworks were introduced so that we could write code that tests code in your system.
    - This made testing more automated.
    - However, the quality of the tests now partially depends on the developer’s knowledge of the testing framework.



- **Acceptance Testing** is performed from the perspective of a user of a system in order to verify the requirements have been satisfied.
- **Integration Testing** is a broad category of tests that validate integration between units of code or code and outside dependencies such as a database, network resource, or file.
- **Functional Testing** validates that the functional design has been satisfied
- **Unit Testing** is low level testing performed by the programmer to validate that individual units of code function as expected by the programmer.

What is Unit Test



Function

Testing Behaviour

Piece of code

Unit Test

Unit Test is a piece of code which tests behaviour of a function or class.

Unit Tests are written by the developers.

Unit Tests

Should be:

1. Fast
2. Repeatable
3. Independent
4. Obvious

Steps:

1. Arrange (Setup)
2. Act (Test)
3. Assert (Verify)

Rules:

1. No external dependencies
2. One logical assertion per test
3. Test code should be the same quality as product code
4. Test early, test often

JUnit

JUnit is a Java *Framework* for writing and running Unit Tests.

Package: `org.junit`

Unit Testing in Java: Introduction

The most commonly used testing framework in Java is **JUNIT**.

- JUNIT is written in Java and will leverage all the concepts you’ve learned so far: declaring variables, calling methods, instantiating objects.
- All related tests can be written in a single test class containing several methods, each method could be a test.
- Each method should contain an assertion, which compares the result of your code against an expected value.

@annotation

Annotations are metadata added to Java code to communicate with a the JVM or a Framework, like JUnit, the purpose of the code or how/when to use it. Usually added above a method, member variable, or class.

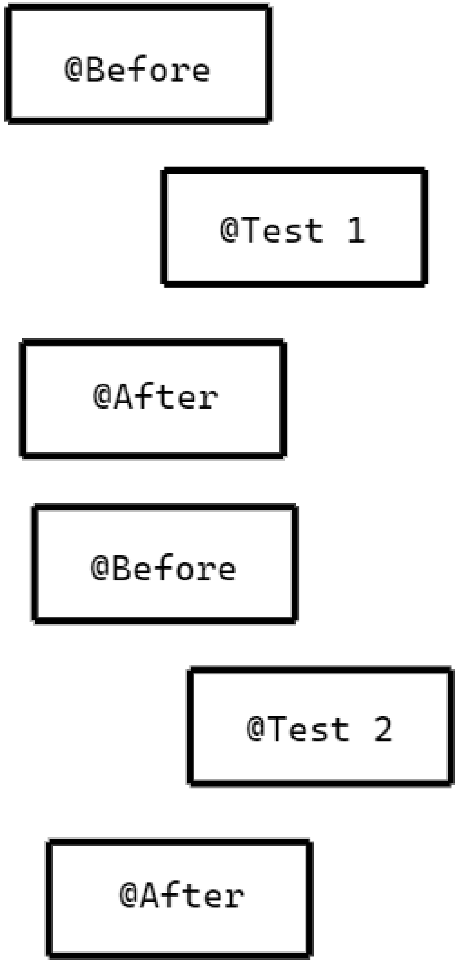
@Test - runs as the test

JUnit Test Lifecycle

The *life cycle* of JUnit tests are controlled by *annotations* on public methods in the class.

@Before - runs before each test, to do setup
@Test - runs as the test
@After - runs after each test to cleanup

```
@Before
public void setup() { }

@Test
public void test_something() { }

@After
public void cleanup() { }
```



org.junit.Assert

The Assert class allows verification of results. Some examples:

```
        Assert.assertTrue( optionalMessage, booleanCondition )

        Assert.assertFalse( optionalMessage, booleanCondition )

        Assert.assertEquals( optionalMessage, expectedValue, actualValue )

    Assert.assertEquals( optionalMessage, expectedDouble, actualDouble, precision )

        Assert.fail()
```

Asserts can be in the test method or any private methods that it calls!

The AAA pattern of unit testing

- Arrange – arrange the conditions of the test
- Act – perform action of interest
- Assert – validate that the expected outcome occurred by means on an assertion

```
public class Calculator {
   public double sum(double first, double second) {
     return first + second;
   }
}
```

(1) class container
(2) xUnit’s attribute indicating a test (Java uses Junit and @Test)
(3) Name of unit Test
(4) Arrange
(5) Act
(6) Assert

```
public class CalculatorTest {  (1)

    @Test (2)
    public void sum_of_two_numbers() { (3)
       // Arrange
       double first = 10;                              (4)
       double second = 20;                             (4)
       Calculator calculator = new Calculator();  (4)

       // Act
       double actual = calculator.sum(first, second); (5)

       // Assert
        assertEquals(30, actual);  (6)
```

Unit Testing in Java: Assertions

An assertion is the result of a a comparison between an actual value of an expected value. Supposed we have a Java method that returned the following:

```
public static boolean divBy2(int i) {
        return i%2 == 0;
}
```

Assertion 1: If I run divBy2(4) the result of invoking the method should be true.

If divBy2(4) returns false, then the assertion has failed.

Assertion 2: If I run divBy2(5) the result of invoking the method should be false.

If the method is invoked and the result is true, then the assertion has failed.

Unit Testing in Java: Production Code vs Test Code

- Production code refers to the actual code for your project.
- Test code is the code that is designed to test Production Code

- Production code often tends to be DRY (Don’t Repeat Yourself), test code tends to be more WET (Write Everything Twice)

- Production code is for fulfilling the contract. Test code is for verify this contract.

Unit Testing in Java: Example

Production Code

```java
package te.examples.testingexamples;

public class MyApp {

    public boolean divBy2(int number) {
        return number%2==0;
    }

    public String concatenator(String [] wordArray) {
        String output = "";

        for (String word : wordArray) {
            output += word;
        }

        return output;
    }
}
```

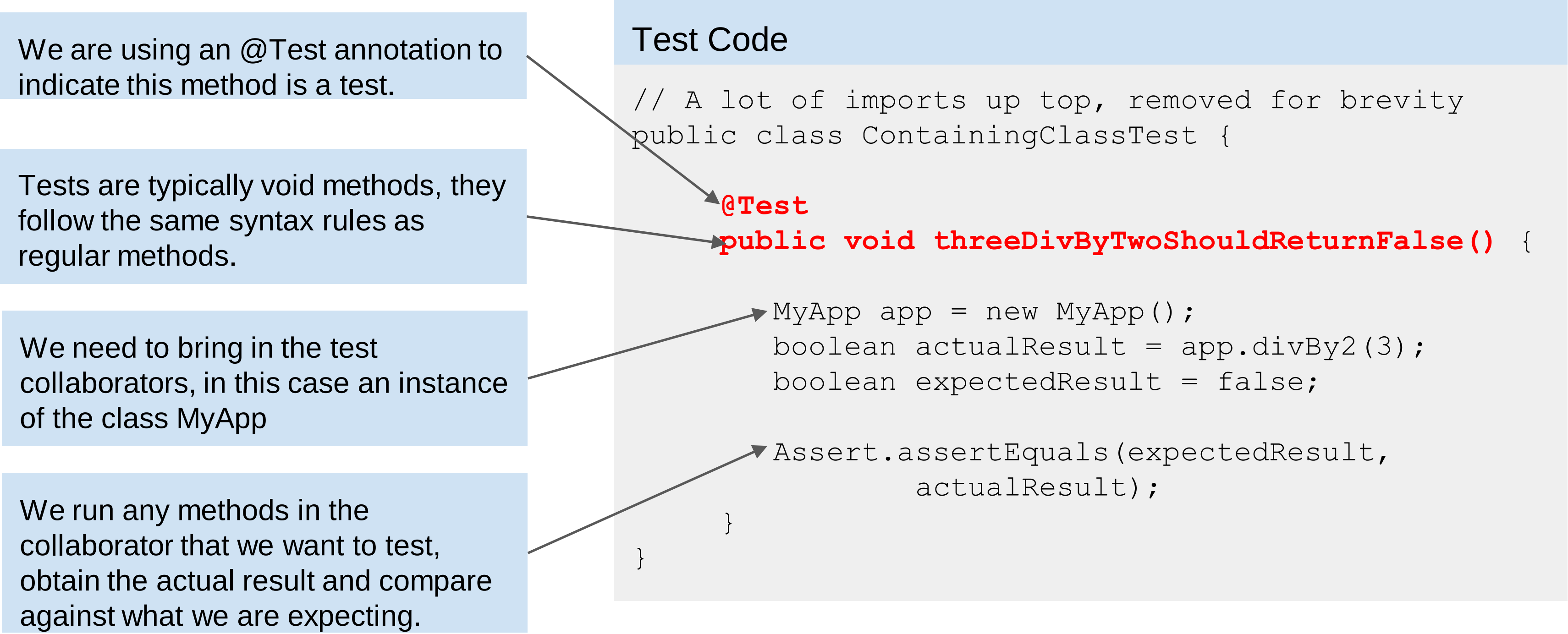
These two are tests designed to check if divBy2 is working properly.

Test Code

```java
// A lot of imports up top, removed for brevity
public class TestContainingClass {

    @Test
    public void threeDivByTwoShouldReturnFalse() {

        MyApp app = new MyApp();
        boolean actualResult = app.divBy2(3);
        boolean expectedResult = false;

        Assert.assertEquals(expectedResult, actualResult);
     //  Assert.assertFalse(actualResult);
    }

    @Test
    public void fourDivByTwoShouldReturnTrue() {

        MyApp app = new MyApp();
        boolean actualResult = app.divBy2(4);
        boolean expectedResult = true;

        Assert.assertEquals(expectedResult, actualResult);
        //  Assert.assertTrue(actualResult);
    }
}
```

Unit Testing in Java: Anatomy of Test Method

Let’s take a closer look at a test method and what happens inside it:



We are using an @Test annotation to indicate this method is a test.

Tests are typically void methods, they follow the same syntax rules as regular methods.

We need to bring in the test collaborators, in this case an instance of the class MyApp

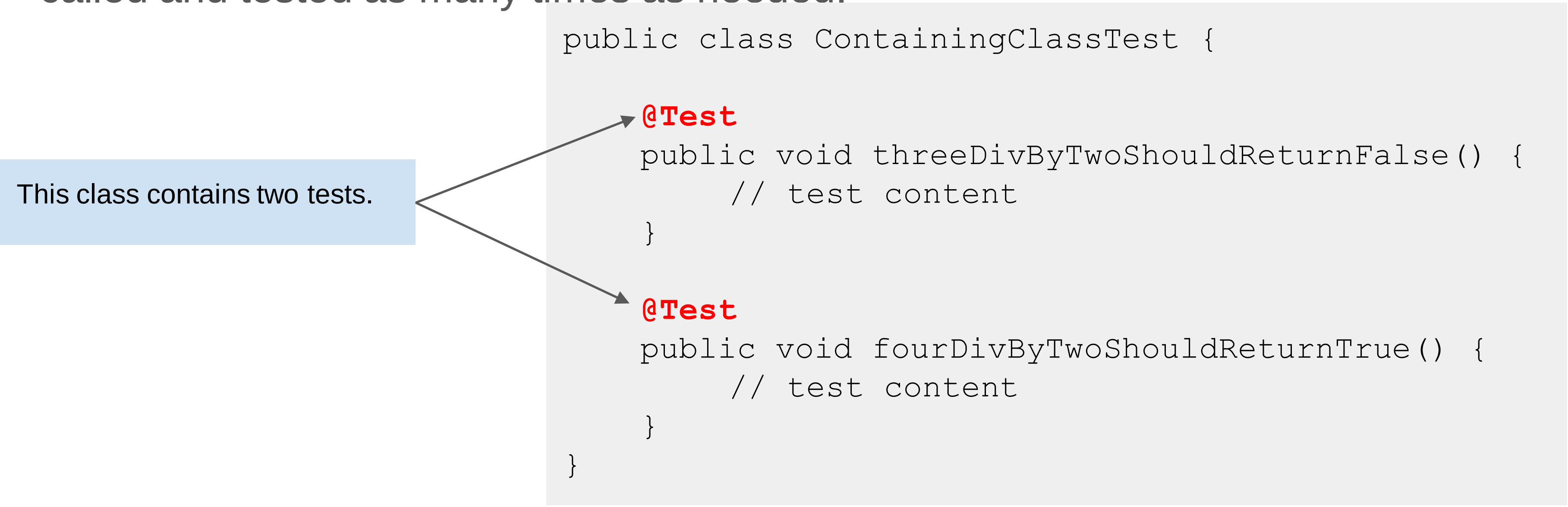
We run any methods in the collaborator that we want to test, obtain the actual result and compare against what we are expecting.

Test Code

```
// A lot of imports up top, removed for brevity
public class ContainingClassTest {

     @Test
     public void threeDivByTwoShouldReturnFalse() {

        MyApp app = new MyApp();
        boolean actualResult = app.divBy2(3);
        boolean expectedResult = false;

        Assert.assertEquals(expectedResult,
                    actualResult);
     }
}
```

Unit Testing in Java: Multiple Tests

A testing class can contain multiple tests. The same production method can be called and tested as many times as needed.



This class contains two tests.

```
public class ContainingClassTest {

    @Test
    public void threeDivByTwoShouldReturnFalse() {
         // test content
    }

    @Test
    public void fourDivByTwoShouldReturnTrue() {
         // test content
    }
}
```

Unit Testing in Java: Before & After

You can specify that certain pieces of code be run before and after every single test.

```
public class ContainingClassTest {

    @Before
    public void setUp() throws Exception {

       System.out.println("Test starting.");
    }

    @After
    public void tearDown() throws Exception {

       System.out.println("Test complete.");
    }

    @Test
    public void threeDivByTwoShouldReturnFalse() {
        // Test content.
    }

    @Test
    public void fourDivByTwoShouldReturnTrue() {
        // Test content.
    }
}
```

Anything in the @Before block will run right before a test.

Anything in the @After block will run right after the test.

So the order of operations is:

1. run setup()
2. Run threeDivByTwo… test
3. run tearDown()
4.run setup()
5. Run fourDivByTwo… test
6. Run tearDown();