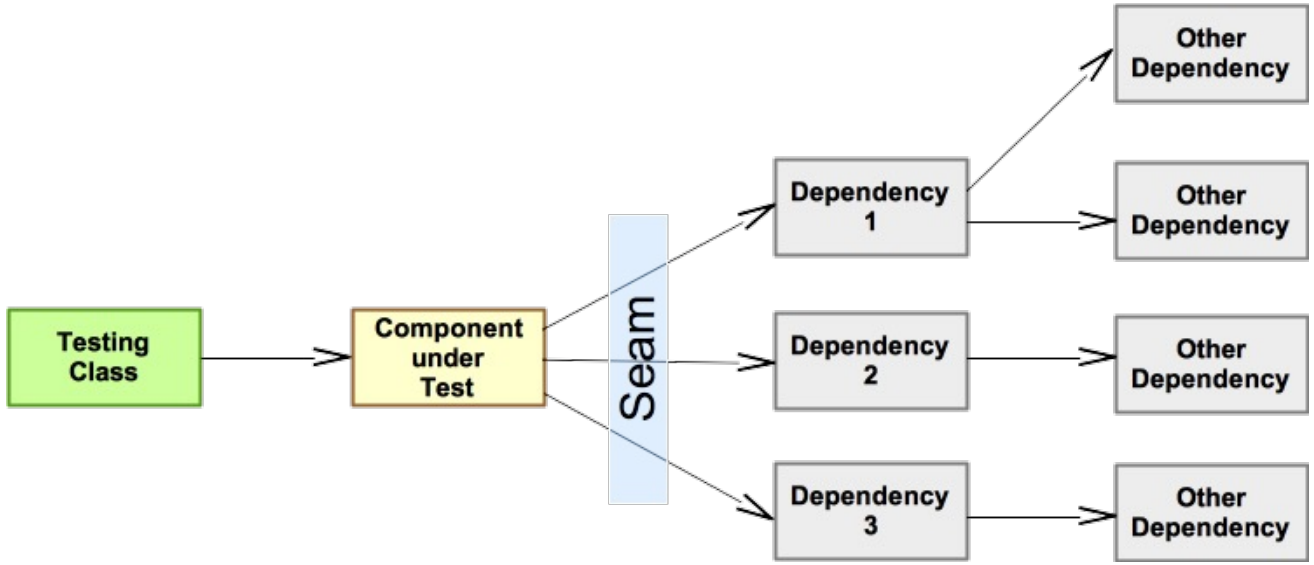
Unit Testing Activity



Your activity for the Unit Testing lesson is to build tests for existing Project components.

- These slides direct you through the process of creating unit tests for your project.
  ✓ ***Activity actions are highlighted in green with a checkmark.***
- But first, these slides provide technical details on:
  1. ***How to organize test classes using Maven***
  2. ***How to run tests using Maven***
  3. ***How to structure a typical test class***
  4. ***How to use JUnit assertion functions***
  5. ***How to use package-private constants in test code***
  6. ***How to use Mockito mock objects using Dependency Injection***

Maven provides a convenient structure for organizing unit test code.

- Put your test classes in a separate source path.
  - *The goal is to keep the test source code separate from the production source code.*
  - *Using Maven that is usually `src/test/java`.*

  ✓ *Create this directory if it doesn't already exist.*

  ✓ *(Optional) Link your IDE to this source path.*
- Most IDEs provide wizards for creating unit tests.
  - *Make sure the IDE stores the test classes in the proper source path.*
- The unit test code examples in these slides are from the Heroes API starter code

Maven will run tests during builds and there is also the test target.

```
PS C:\Users\student\heroes-api-starter> mvn clean
test [INFO] Scanning for projects...
                                                    <SKIPPING SOME Maven OUTPUT>
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running com.heroes.api.heroesapi.controller.HeroControllerTest
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.947 s -
in com.heroes.api.heroesapi.controller.HeroControllerTest
[INFO] Running com.heroes.api.heroesapi.persistence.HeroFileDAOTest
[INFO] Tests run: 10, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.656 s - in
com.heroes.api.heroesapi.persistence.HeroFileDAOTest [INFO] Running com.heroes.api.heroesapi.HeroesApiApplicationTests
                                                    <SKIPPING SOME Maven OUTPUT>
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 3.185 s - in
com.heroes.api.heroesapi.HeroesApiApplicationTests [INFO] Running com.heroes.api.heroesapi.model.HeroTest
[INFO] Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.003 s - in com.heroes.api.heroesapi.model.HeroTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 18, Failures: 0, Errors: 0, Skipped:
0 [INFO]
[INFO]
[INFO] --- jacoco-maven-plugin:0.8.7:report (report) @ heroes-api ---
[INFO] Loading execution data file C:\Users\student\heroes-api-
starter\target\jacoco.exec [INFO] Analyzed bundle 'heroes-api' with 5 classes
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  10.815 s
[INFO] Finished at: 2021-11-25T09:57:12-05:00
[INFO] ------------------------------------------------------------------------
```

Test files are kept in separate directories from the application code but in the same package.

- Name the test class after the *component under test* (CuT) in the same package.
  - ***So if CuT is com.heroes.api.heroesapi.model.Hero***
  - ***Then test class is: com.heroes.api.heroesapi.model.HeroTest***
  - ***Doing so gives the test code package-private access to CuT class.***

JUnit understands several annotations that you must use in your test files.

- Annotate each class with `@Tag` to indicate which architectural tier the class is in.
  - ***Use these tags:*** `Persistence-tier, Model-tier, Controller-tier`
  - ***You will learn more about the role of these tags in the Code Coverage lesson.***
- Annotate each test method with `@Test`.
- Use `@BeforeEach` or `@BeforeAll` annotated methods for setup before each test or setup done once before all tests
- Method annotations `@AfterEach` and `@AfterAll` serve similar clean up tasks after running tests.

Recall the checklist of types of unit tests.

- Business logic
  - ***Tests for each path through a method***
  - ***Happy path as well as failure paths***
- Constructors and accessors
- Defensive programming checks
  - ***Validation of method arguments***
    - `NullPointerException`
    - `IllegalArgumentException`
  - ***Validation of component state***
    - `IllegalStateException`
- Special methods, e.g. `equals, hashcode, toString`, as needed
- Exception handling

Here's an example unit test suite for the `Hero` class.

```
package com.heroes.api.heroesapi.model;

import static org.junit.jupiter.api.Assertions.assertEquals;
import org.junit.jupiter.api.Tag;
import org.junit.jupiter.api.Test;

@Tag("Model-tier")
public class HeroTest {

    @Test
    public void testCtor()
    {…}

    @Test
    public void testName()
    {…}

    @Test
    public void testToString() {…}
}
```

Import JUnit assertion functions and Tags

Indicate Model tier

Test constructors

Test accessors and mutators

Special methods

Here's an example test methods for the Hero class.

```
@Test
public void testToString() {
    // Setup
    int id = 99;
    String name = "Wi-Fire";
    String expected_string = String.format(Hero.STRING_FORMAT,id,name);
    Hero hero = new Hero(id,name);

    // Invoke
    String actual_string = hero.toString();

    // Analyze
    assertEquals(expected_string,actual_string);
}
```

Here's an example of how to test an expected exception.

- Example from HeroFileDAOTest.java
  - ***Use the assertThrows assertion:***

```
@Test
public void testConstructorException() throws IOException {
    // Setup
    ObjectMapper mockObjectMapper = mock(ObjectMapper.class);
    doThrow(new IOException()).when(mockObjectMapper).readValue(new File("doesnt_matter.txt"),Hero[].class);

    // Invoke & Analyze
    assertThrows(IOException.class,() -> new HeroFileDAO("doesnt_matter.txt",mockObjectMapper),"IOException not
    thrown");
}
```

  - ***Roughly the same as:***

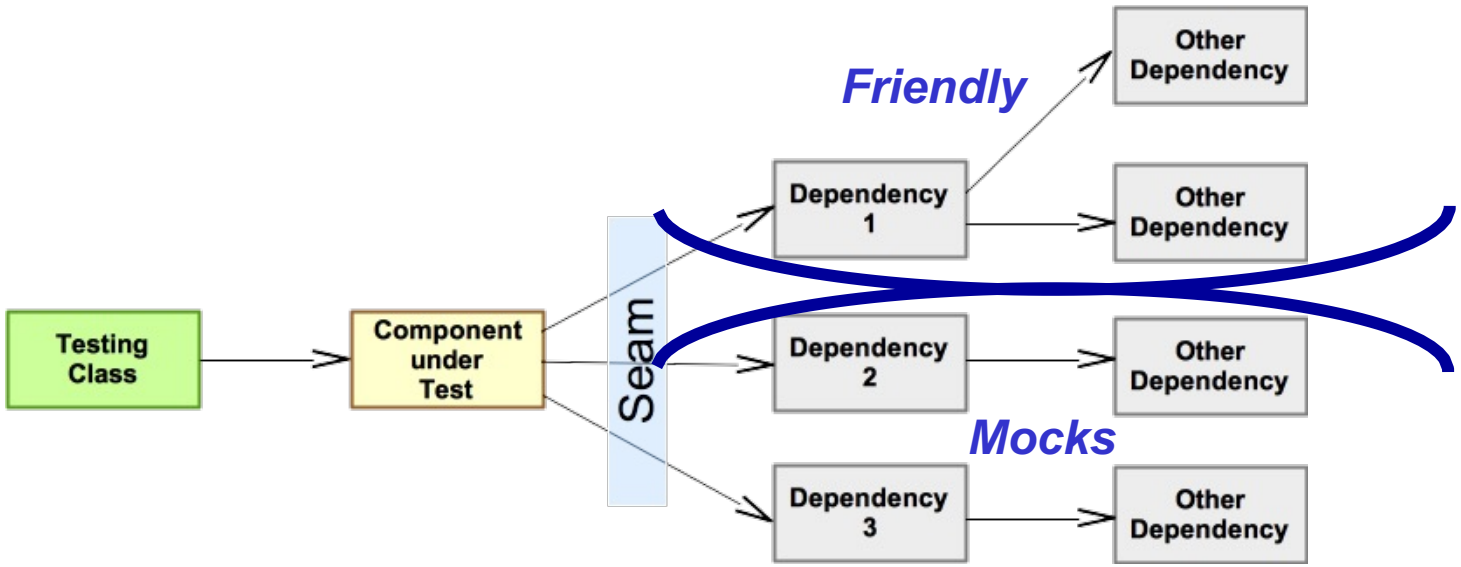
```
@Test
public void testConstructorException() throws IOException {
    // Setup
    ObjectMapper mockObjectMapper = mock(ObjectMapper.class);
    doThrow(new IOException()).when(mockObjectMapper).readValue(new File("doesnt_matter.txt"),Hero[].class);

    // Invoke
    try {
        new HeroFileDAO("doesnt_matter.txt",mockObjectMapper);
    // Analyze
        fail("IOException not thrown");
    }
    catch (IOException ioe) {}
}
```

JUnit has many built-in assertions you can use.

- Test truth-hood
  - *assertTrue(condition[, message])*
  - *assertFalse(condition[, message])*
- Test values or objects for equality
  - *assertEquals(expected, actual[, message])*
  - *assertNotEquals(expected, actual[, message])*
- Test objects for identity (obj1 == obj2)
  - *assertSame(expected, actual[, message])*
  - *assertNotSame(expected, actual[, message])*
- Test null-hood
  - *assertNull(object[, message])*
  - *assertNotNull(object[, message])*
- Test exceptions
  - *assertThrows(exception class, executable [, message])*
  - *assertDoesNotThrow(executable [, message]])*
- Automatic failure
  - *fail(message)*

When components have dependencies you have to consider how to isolate the dependencies.

- Dependencies are isolated along the testing seam for a component.
- There are three elements to consider
  - *Component under Test (CuT)*
  - *Friendly dependencies that can be trusted to work*
  - *Other dependencies that must have mocks because they are not trusted or we need special control during the test*



Here's an example unit test suite for the HeroFileDAO class.

- Import the necessary Junit, Mockito, and Friendly dependencies

```
package com.heroes.api.heroesapi.persistence;

import java.io.File;
import java.io.IOException;
import static org.junit.jupiter.api.Assertions.assertDoesNotThrow;
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertNotNull;
import static org.junit.jupiter.api.Assertions.assertNull;
import static org.junit.jupiter.api.Assertions.assertThrows;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Tag;
import org.junit.jupiter.api.Test;

import static org.mockito.ArgumentMatchers.any;
import static org.mockito.Mockito.doThrow;
import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.when;

import com.fasterxml.jackson.databind.ObjectMapper;

import com.heroes.api.heroesapi.model.Hero;
```

Import JUnit assertion functions and tags

Import Mockito functions and classes

Used to create a mock object

Friendly

Here's an example unit test suite for the HeroFileDAO class.

- Setup and Happy Path Tests

```
@Tag("Persistence-tier")
public class HeroFileDAOTest {
    HeroFileDAO heroFileDAO;
    Hero[] testHeroes;
    ObjectMapper mockObjectMapper;

    @BeforeEach
    public void setupHeroFileDAO() throws IOException {…}

    @Test
    public void testGetHeroes() {…}

    @Test
    public void testFindHeroes() {…}

    @Test
    public void testGetHero() {…}


    @Test
    public void testDeleteHero() {…}

    @Test
    public void testCreateHero() {…}

    @Test
    public void testUpdateHero() {…}
```

Indicate controller tier

Setup function to be called before each test

Test business logic / CRUD operations.

Here's an example unit test suite for the HeroFileDAO class.

- Error Handling Tests

```
    @Test
    public void testSaveException() {…}

    @Test
    public void testGetHeroNotFound() {…}

    @Test
    public void testDeleteHeroNotFound() {…}

    @Test
    public void testUpdateHeroNotFound() {…}

    @Test
    public void testConstructorException() throws IOException {…
}                                                                }
```

Test defensive programming checks.

A quick review of dependency injection which is a key design technique to make classes testable.

```
@Component
public class HeroFileDAO implements HeroDAO {
    Map<Integer,Hero> heroes;
    private ObjectMapper objectMapper;
    private static int nextId;
    private String filename;

    public HeroFileDAO(@Value("${heroes.file}") String filename,ObjectMapper objectMapper) throws IOException {…}

    …
}
```

This class is instantiated by the Spring Boot framework by virtue of the @Component annotation.

The dependent ObjectMapper object is injected into the HeroFileDAO constructor

- We could create an ObjectMapper object in the HeroFileDAO constructor, but…
  - ***Recall that the ObjectMapper is responsible for deserialization and serialization of JSON objects to/from Java Objects and to write and read from a file***
  - ***Testing with an actual file would be difficult – the consistent state of the file would need to be ensured and the file may need to be read to validate***
  - ***By injecting the ObjectMapper, we can use a mock object in our tests***

We have to setup the mocks before each test.

```
@BeforeEach
public void setupHeroFileDAO() throws IOException {
    mockObjectMapper = mock(ObjectMapper.class);

    testHeroes = new Hero[3];
    testHeroes[0] = new Hero(99,“Wi-Fire");
    testHeroes[1] = new Hero(100,“Galactic Agent");
    testHeroes[2] = new Hero(101,“Ice Gladiator");

    when(mockObjectMapper
          .readValue(new File("doesnt_matter.txt"),Hero[].class))
            .thenReturn(testHeroes);

    heroFileDAO = new HeroFileDAO("doesnt_matter.txt",mockObjectMapper);
}
```

Use the Mockito `mock` function to create a mock object

Create an array of Hero objects that can be returned

Use the Mockito `when` and `thenReturn` APIs to simulate a reading Hero objects from a file

Inject the mock ObjectMapper into the HeroFileDAO constructor

See HeroControllerTest for more examples of Mock Objects and Dependency Injection

Mockito has a rich API for setting scenarios and for inspecting test activity.

- Arranging scenarios:
  - *when(mock.method(args)).thenReturn(value)*
  - *when(mock.method(args)).thenThrow(new Exception())*
  - *when(mock.method(args)).thenAnswer(lambda)*
- Inspecting activity within the CuT method:
  - *verify(mock).method(args)*
  - *verify(mock, times(1)).method(args)*
  - *Other verification modes:*
    - *times(n), atLeast(n), atMost(n) & never()*
- Specifying arguments:
  - *An actual value or object: eq(value) or eq(object)*
  - *Any value (anyInt(), etc); any() for any Object*
  - *Many more types of **Matchers***

Your exercise is to build unit tests for two classes in your Project.

- ✓Each team member picks two classes in your project and builds unit tests for them.
  - ***Each team member will pick different classes to test.***
  - ***If you need to refactor the code to make the component more testable, then do so.***
- ✓Create the test class in the appropriate package in the test source path.
- ✓Create a reasonable set of test cases.
  - ***At least three test cases for each CuT.***
  - ***Focus on the business logic methods (eg, REST Controllers or Services).***

Your exercise is to build unit tests for two classes in your Project.

- ✓Upload the two unit test source files to the *Unit testing - individual* Dropbox in MyCourses.
- ✓You will now complete the *Definition of Done* unit testing checklist items to consider the story done.