Course Name: Mastering Unit Testing with JUnit and Mockito

Module 1: Introduction to Unit Testing

- Importance of Testing in Software Development
- Different Levels of Testing: Unit Testing, Integration Testing, System Testing
- Introduction to Unit Testing: What, Why, and When
- Characteristics of Good Unit Tests

Module 2: Introduction to JUnit

- What is JUnit and why use it?
- Setting up JUnit in a Java Project
- Writing Simple Unit Tests using JUnit
- Understanding Assertions and Test Failures
- JUnit Annotations and Test Lifecycle
- Testing Exceptions in JUnit

Module 3: Advanced JUnit Techniques

- Parameterized Tests in JUnit
- Using Assumptions in JUnit
- Grouping Tests with @Nested
- Test Interfaces and Default Methods
- Dynamic Tests in JUnit

Module 4: Introduction to Mockito

- What is Mockito and why use it?
- Setting up Mockito in a Java Project
- Mocking and Stubbing with Mockito
- Verifying Behavior with Mockito
- Argument Matchers and Captors

Module 5: Advanced Mockito Techniques

- Stubbing Void Methods with Mockito
- Using Spies in Mockito
- Mocking and Stubbing Asynchrony
- Exception Handling in Mockito

Module 6: Integration of JUnit and Mockito

- Writing Unit Tests for Service Layer Code
- Writing Unit Tests for Data Access Layer Code

- Best Practices for Unit Testing with JUnit and Mockito

Module 7: Project - Implementing Unit Tests in a Java Application

- implement unit tests in a simple Java application, using JUnit and Mockito. write tests for various layers (e.g., service layer, data access layer), demonstrating effective unit testing strategies and proper use of mocking and stubbing.

Introduction to Unit Testing.

Part 1: Importance of Testing in Software Development

Testing is a crucial component of software development that helps ensure the quality of the software being delivered. Here are some reasons why testing is important:

1. **Identify Defects and Errors**: Testing helps to discover bugs and issues in the software before it is released to the end-users.

2. **Ensure Software Quality**: Testing verifies that the software meets the requirements and expectations of the users. It ensures the software works correctly and as expected.

3. **Reduce Risks and Costs**: By finding and fixing bugs early in the development process, testing can save time and cost in the long run.

4. **Increase Confidence in the Software**: Testing the software gives confidence to the developers and stakeholders that the software functions as intended.

Part 2: Different Levels of Testing: Unit Testing, Integration Testing, System Testing

There are various levels of testing that typically occur during the software development lifecycle:

1. **Unit Testing**: This is the process of testing individual components or units of a software to verify that they function as expected.

2. **Integration Testing**: This testing phase focuses on the interfaces between components, interactions with different parts of a system, such as the file system, databases, or web services.

3. **System Testing**: It involves testing an integrated system to verify that it meets the specified requirements.

In this course, we will focus mainly on unit testing.

Part 3: Introduction to Unit Testing: What, Why, and When

What is Unit Testing?
Unit testing is a type of testing where individual units or components of a software are tested. The purpose is to validate that each unit of the software code performs as expected.

Why Unit Testing?
- It helps in maintaining and changing the code.
- It can catch bugs at early stages.
- It helps in the design of the code.
- It can serve as documentation for the system.

When to use Unit Testing?
Unit testing is typically performed at the development stage, and developers often use it throughout the coding process.

Part 4: Characteristics of Good Unit Tests

1. **Isolated**: Unit tests should focus on testing a single unit of work. They should not depend on any external resources such as databases, file systems, or network services.

2. **Repeatable**: You should be able to run a unit test as many times as you want, in any order, and it should always produce the same result.

3. **Self-Checking**: The test should be able to automatically detect if it passed or failed without any manual intervention.

4.**Timely**: Unit tests should be written just before the production code that makes them pass. This helps in the design of the code and catches any potential issues early.

5. **Fast**: Unit tests should execute quickly. This is important because they are run frequently during development.

Thought exercise: Consider an application you have worked on or used recently. Can you think of examples of how unit testing might be beneficial for this application?

Module 2: Introduction to JUnit.

Part 5: What is JUnit and why use it?

JUnit is a popular testing framework in Java for performing unit testing. It provides annotations to identify test methods and contains assertions for testing expected results. It's easy to use, open-source, and has strong community support.

JUnit enables the writing of repeatable tests, which helps in regression testing, i.e., when developers modify the source code, they can run the tests again to ensure that the existing functionality is not broken.

Part 6: Setting up JUnit in a Java Project

To use JUnit, you need to include its dependency in your build file. If you're using Maven, you can add the following to your `pom.xml`:

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.13.2</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Part 7: Writing Simple Unit Tests using JUnit

Here's how to write a simple test using JUnit:

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class CalculatorTest {
  @Test
  public void testAddition() {
    Calculator calculator = new Calculator();
    int result = calculator.add(2, 3);
    assertEquals(5, result);
  }
}
```

In this example, `**@Test**` is a JUnit annotation that marks the `testAddition` method as a test method. `**assertEquals**` is a JUnit assertion that checks if two values are equal.

Part 8: Understanding Assertions and Test Failures

Assertions are used for checking the correctness of a test. If the assertion condition is true, the test passes; if it's false, the test fails. JUnit provides a variety of assertion methods such as `**assertEquals**`, `**assertTrue**`, `**assertFalse**`, `**assertNotNull**`, and more.

When a test fails, JUnit provides a clear message about what was expected and what was the actual result. This helps to quickly identify and fix the problem.

Part 9: JUnit Annotations and Test Lifecycle

JUnit provides several important annotations that control the execution of your tests:

- `**@Test**`: Marks a method as a test method.
- `**@BeforeEach**`: Used for setup. This method is run before each test.
- `**@AfterEach**`: Used for teardown. This method is run after each test.
- `**@BeforeAll**`: This static method is run once before any tests.
- `**@AfterAll**`: This static method is run once after all tests.

The setup (`**@Before**`) and teardown (`**@After**`) methods are commonly used for initializing and cleaning up test resources.

Part 10: Testing Exceptions in JUnit

JUnit provides a way to test for exceptions by using the `**@Test**` annotation along with the `**expected**` attribute. For example:

```
@Test(expected = IllegalArgumentException.class)
public void testException() {
  Calculator calculator = new Calculator();
  calculator.divide(10, 0); // This should throw IllegalArgumentException
}
```

In this example, the test will pass if `calculator.divide(10, 0)` throws an `**IllegalArgumentException**`.

Exercise:Set up a simple Java project, add the JUnit dependency, and try writing some unit tests. Experiment with different assertion methods, the `@Before` and `@After` methods, and testing for exceptions.

Module 3: Advanced JUnit Techniques.

Part 11: Parameterized Tests in JUnit

JUnit provides a way to run the same test multiple times with different inputs, called Parameterized Tests. This is especially useful when you want to test a method with various input values.

Here's an example of how to write a parameterized test:

```
import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;

@RunWith(Parameterized.class)
public class CalculatorParameterizedTest {

  private int input1;
  private int input2;
  private int expectedResult;

  public CalculatorParameterizedTest(int input1, int input2, int expectedResult) {
    this.input1 = input1;
    this.input2 = input2;
    this.expectedResult = expectedResult;
  }

  @Parameterized.Parameters
  public static Collection<Object[]> data() {
    return Arrays.asList(new Object[][] {
      { 1, 1, 2 },
      { 2, 3, 5 },
      { 3, 3, 6 },
      { 4, 5, 9 }
    });
  }

  @Test
  public void testAddition() {
    Calculator calculator = new Calculator();
    assertEquals(expectedResult, calculator.add(input1, input2));
  }
}
```

In this example, the test `**testAddition**` will run four times with different inputs and expected results.

Part 12: Using Assumptions in JUnit

Assumptions in JUnit can be used to only run a test when certain conditions are met. If the assumption fails, the test is skipped. This can be useful when you only want to run a test under specific circumstances.

Here's an example of using assumptions:

```
import org.junit.Assume;
import org.junit.Test;

public class AssumptionsTest {

  @Test
  public void testOnlyOnLinux() {
    Assume.assumeTrue(System.getProperty("os.name").contains("Linux"));
    // Rest of the test, which will only run on Linux
  }
}
```

Part 13: Grouping Tests with @Nested

JUnit provides a way to logically group related tests using `@Nested`. This can be useful for better organization of tests and can also be used to share setup and teardown code between related tests.

Here's an example of how to use `**@Nested**`:

```
java
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Nested;
import org.junit.jupiter.api.Test;

public class NestedTest {

  @BeforeEach
  public void setUp() {
    // Set up code here
  }

  @Test
  public void someTest() {
    // Test code here
  }

  @Nested
  class WhenCondition {

    @BeforeEach
```

```
    public void setUp() {
      // Additional setup for this group of tests
    }

    @Test
    public void someOtherTest() {
      // Test code here
    }
  }
}
```

Part 14: Test Interfaces and Default Methods

JUnit allows test interfaces and default methods. This can be useful for sharing common test code across multiple test classes.

```
public interface CalculatorTest {

  Calculator createInstance();

  @Test
  default void additionShouldWorkCorrectly() {
    Calculator calculator = createInstance();
    assertEquals(4, calculator.add(2, 2));
  }
}


public class BasicCalculatorTest implements CalculatorTest {
  @Override
  public Calculator createInstance() {
    return new BasicCalculator();
  }
}
```

In this example, `*BasicCalculatorTest*` inherits the `*additionShouldWorkCorrectly*` test method from `*CalculatorTest*`.

Part 15: Dynamic Tests in JUnit

JUnit allows the creation of tests at runtime with its Dynamic Tests feature. This can be useful when you don't know all the tests at compile time, such as when the tests are data-driven.

Here's an example of how to create dynamic tests:

```
import org.junit.jupiter.api.DynamicTest;
import org.junit.jupiter.api.TestFactory;

public class DynamicTestsDemo {

  @TestFactory
  public Stream<DynamicTest> dynamicTests() {
    List<Integer> inputList = Arrays.asList(1, 2, 3, 4, 5);
    List<Integer> outputList = Arrays.asList(2, 4, 6, 8, 10);

    return IntStream.range(0, inputList.size())
        .mapToObj(i -> DynamicTest.dynamicTest("Test " + i, () -> {
          int input = inputList.get(i);
          int expectedOutput = outputList.get(i);
          assertEquals(expectedOutput, input * 2);
        }));
  }
}
```

In this example, we create a dynamic test for each pair of inputs and expected outputs.

Exercise: Implement some parameterized tests in your project. Try using assumptions to control when tests are run. Experiment with `@Nested` tests, test interfaces, and dynamic tests. Consider how these techniques can make your tests more flexible and maintainable.

Module 4: Introduction to Mockito.

Part 16: What is Mockito and why use it?

Mockito is a mocking framework for unit tests in Java. Mockito allows you to create and configure mock objects. Using Mockito, you can verify that certain methods were called on the mock object with specific parameters, stub method calls and return specific values, and more.

Mockito can be useful in writing unit tests for classes that have external dependencies. By using mocks for these dependencies, you can isolate the class you are testing and ensure that your unit tests are fast and reliable.

Part 17: Setting up Mockito in a Java Project

To use Mockito, you need to include its dependency in your build file. If you're using Maven, you can add the following to your `*pom.xml*`:

```
<dependencies>
  <dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>3.11.2</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Part 18: Mocking and Stubbing with Mockito

With Mockito, you can create mocks using the `mock()` method:

List mockedList = mock(List.class);

You can then stub the behavior of the mock object using `*when()*` and `*thenReturn()*`:

```
when(mockedList.size()).thenReturn(10);
int size = mockedList.size(); // size will be 10
```

Part 19: Verifying Behavior with Mockito

Mockito allows you to verify that certain methods were called on the mock object during the test. This is done using the `*verify()*` method:

```
mockedList.add("one");
verify(mockedList).add("one");
```

In this example, we verify that the `*add("one")*` method was called on the mockedList.

Part 20: Argument Matchers and Captors

Mockito provides Argument Matchers that allow flexible verification or stubbing. Matchers can be used in place of actual method arguments when stubbing or verifying:

```
when(mockedList.get(anyInt())).thenReturn("element");
String result = mockedList.get(999); // result will be "element"
```

Argument Captors are a more advanced feature that allows capturing arguments for further assertions or for use in stubbing:

```
ArgumentCaptor<String> argumentCaptor = ArgumentCaptor.forClass(String.class);
verify(mockedList).add(argumentCaptor.capture());
assertEquals("one", argumentCaptor.getValue());
```

Exercise: Set up a simple Java project, add the Mockito dependency, and try writing some unit tests using Mockito. Create some mocks, stub their behavior, and verify that certain methods were called. Experiment with argument matchers and captors. Consider how Mockito can help you write more effective unit tests.

Module 5: Advanced Mockito Techniques.

Part 21: Stubbing Void Methods with Mockito

Stubbing void methods can be a little tricky with Mockito, as you can't use the `*when().thenReturn()*` pattern that we use for methods that return a value. However, Mockito provides a solution for this:

```
doThrow(new RuntimeException()).when(mockedList).clear();

mockedList.clear(); // this will throw RuntimeException
```

In this example, we've stubbed the `*clear()*` method (which is a void method) to throw a `*RuntimeException*` when it's called.

Part 22: Using Spies in Mockito

While mocks are completely dummy implementations that simply record how they are called, spies are stubbed copies of real objects. When you create a spy, by default, all methods have their typical behavior. However, you can still stub specific methods as you would do with a mock.

Here's an example of how to use a spy:

```
List<String> list = new ArrayList<String>();
List<String> spyList = spy(list);

spyList.add("one");
spyList.add("two");

verify(spyList).add("one");
verify(spyList).add("two");

assertEquals(2, spyList.size());
```

In this example, `*spyList*` is a spy of `*list*`. By default, it behaves like the real `*ArrayList*` that it's spying on. However, we could still stub its methods if we wanted to.

Part 23: Mocking and Stubbing Asynchrony

In scenarios where we deal with asynchronous code, Mockito can be used to effectively test such cases. The key idea is to "stub" the asynchronous behavior and make it behave synchronously in the test.

Here's an example of how you might test an asynchronous method:

```
Future<String> future = mock(Future.class);
when(future.get()).thenReturn("Mockito");

SomeService someService = new SomeService();
someService.setFuture(future);

assertEquals("Mockito", someService.getValueFromFuture());
```

In this example, we've stubbed the `*Future*`'s `*get()*` method, which is typically asynchronous, to return a value immediately.

Part 24: Exception Handling in Mockito

With Mockito, you can also stub a method to throw an exception. This can be useful when you want to test how your code handles an error.

```
when(mockedList.get(0)).thenThrow(new IndexOutOfBoundsException());

try {
```

```
  mockedList.get(0);
} catch (IndexOutOfBoundsException e) {
  // Handle exception here, you could also verify it in your test
}
```

In this example, we've stubbed `*get(0)*` to throw an `*IndexOutOfBoundsException*`. This could be useful for testing how your code handles this exception.

Exercise: Try these advanced Mockito techniques in your project. Stub a void method to change its behavior, create a spy of a real object, test asynchronous code, and stub a method to throw an exception. Consider how these techniques can help you create more effective and comprehensive tests.

Module 6: Integration of JUnit and Mockito.

Part 25: Writing Unit Tests for Service Layer Code

Let's look at how to write unit tests for service layer code with JUnit and Mockito.

Suppose you have a UserService with a dependency on UserRepository:

```
public class UserService {

  private UserRepository userRepository;

  // Dependency injection
  public UserService(UserRepository userRepository) {
    this.userRepository = userRepository;
  }

  public User getUser(String id) {
    return userRepository.findById(id);
  }
}
```

You can test UserService by creating a mock of UserRepository and using it to define the expected behavior:

```
public class UserServiceTest {

  @Test
  public void testGetUser() {
```

```
    // Create a mock UserRepository
    UserRepository mockRepository = mock(UserRepository.class);

    // Create a user
    User user = new User();
    user.setId("123");
    user.setName("John Doe");

    // Define the behavior of the mock: when findById is called with "123",
    // it should return the user
    when(mockRepository.findById("123")).thenReturn(user);

    // Create a UserService with the mock UserRepository
    UserService userService = new UserService(mockRepository);

    // Call getUser and verify the result
    User result = userService.getUser("123");
    assertEquals("John Doe", result.getName());
  }
}
```

Part 26: Writing Unit Tests for Data Access Layer Code

Writing unit tests for data access code can be challenging because they interact with a database. It's not unit testing if you're actually hitting the database. The test wouldn't be isolated and could be slow and flaky.

Let's see how to test a Repository:

```
public class UserRepositoryTest {

  @Test
  public void testFindById() {
    // Create an EntityManager mock
    EntityManager entityManager = mock(EntityManager.class);

    // Create a User
    User user = new User();
    user.setId("123");
    user.setName("John Doe");

    // Define the behavior of the mock
    when(entityManager.find(User.class, "123")).thenReturn(user);
```

```
    // Create a UserRepository with the mock EntityManager
    UserRepository userRepository = new UserRepository(entityManager);

    // Call findById and verify the result
    User result = userRepository.findById("123");
    assertEquals("John Doe", result.getName());
  }
}
```

Part 27: Best Practices for Unit Testing with JUnit and Mockito

Here are some best practices for unit testing with JUnit and Mockito:

1. ***Test One Thing at a Time***: Each test should focus on a single operation or functionality. This makes tests easy to read, understand, and debug when they fail.

2. ***Use Descriptive Test Method Names***: The name of the test method should clearly describe what the test does.

3. ***Keep Tests Fast and Independent***: Tests should run quickly and should not depend on each other. Slow or dependent tests can make the testing process inefficient and complicated.

4. **Don't Ignore Failed Tests**: A failing test indicates that there's a problem that needs to be solved.

5. **Use Mockito's Verification Features**: Mockito provides features to verify that methods were called with specific arguments. Make good use of these features to make your tests more accurate and comprehensive.

6. **Don't Overuse Mocks**: While mocks are very useful, overusing them can lead to tests that are hard to read and maintain. Use mocks sparingly and only when necessary.

Exercise: Write some unit tests for a service layer and a data access layer in your project. Use Mockito to mock the dependencies and define their behavior. Try applying the best practices for unit testing.

Module 7: **Project - Implementing Unit Tests in a Java Application**.

For this project, you'll be applying everything you've learned about JUnit and Mockito to write comprehensive unit tests for a simple Java application. This is a practical, hands-on way to reinforce what you've learned in this course.

The Java application will be a simple book store system with the following classes:

1. **Book**: This class will represent a book in the store. It will have fields like `id`, `title`, `author`, and `price`.

2. **BookRepository**: This will be a data access layer class that will handle fetching and storing `Book` instances.

3. **BookService**: This service layer class will contain business logic for operations like purchasing a book, searching for a book, etc. It will use `BookRepository` to access the data.

The goal of this project is to write unit tests for the `BookService` and `BookRepository` classes, using Mockito to mock any dependencies.

Here's a rough outline of what you'll need to do:

1. **Setup**: Create a new Java project, if you haven't already, and include the necessary dependencies for JUnit and Mockito.

2. **Implement the classes**: Start by implementing the `Book`, `BookRepository`, and `BookService` classes. You can keep the methods simple for now - the main goal here is to create something that you can write tests for.

3. **Write tests for BookRepository**: Create a new test class for `BookRepository`. Use Mockito to mock any dependencies and write tests for all the methods in `BookRepository`.

4. **Write tests for BookService**: Create a new test class for `BookService`. Again, use Mockito to mock any dependencies (including `BookRepository`) and write tests for all the methods in `BookService`.

5. **Run the tests**: Run all your tests and make sure they all pass. If any tests fail, debug them and try to figure out why.

6. **Review**: Take some time to review your tests. Are they clear and easy to understand? Do they cover all the important aspects of your classes? Are there any edge cases you didn't consider?

Remember to apply the best practices for unit testing you've learned in this course, such as writing tests that are independent, fast, and testing one thing at a time. Good luck with your project!

Exercise: Implement the above project. Write comprehensive unit tests for a simple Java application. Use Mockito to mock dependencies in your tests. Apply the best practices for unit testing.