**Exercise 1: Introduction to JUnit**

**Objective:**

Get started with JUnit by setting up a simple JUnit test.

**Instructions:**

1. Create a new Java project in your IDE.
2. Add JUnit 5 dependencies to your project’s build file (e.g., **pom.xml** for Maven).
3. Create a simple Java class (e.g., **Calculator**) with a method to be tested (e.g., **add(int a, int b)**).
4. Create a test class for Calculator.
5. Write a test method that tests the add method using JUnit.
   * Use the **@Test** annotation.
   * Use assertions like **assertEquals** to verify the method's output.

**Exercise 2: Understanding Test-Driven Development (TDD)**

**Objective:**

Implement a simple feature using TDD.

**Instructions:**

1. Write a test for a new feature (e.g., a **subtract** method in the **Calculator** class).
   * The test should initially fail.
2. Implement the subtract method to pass the test.
3. Refactor the code if necessary.
4. Ensure the test still passes after refactoring.

**Exercise 3: Setting Up JUnit**

**Objective:**

Configure JUnit in your project and run a basic test case.

**Instructions:**

1. Ensure JUnit is properly set up in your build configuration.
2. Create a Java class with multiple methods to be tested.
3. Write test methods for each of these methods using JUnit.
4. Run the tests and verify they pass.

**Exercise 4: Basic JUnit Annotations**

**Objective:**

Use basic JUnit annotations to set up and tear down test environments.

**Instructions:**

1. Write test methods using the **@Test** annotation.
2. Use **@BeforeEach** and **@AfterEach** to set up and tear down resources for each test.
3. Use **@BeforeAll** and **@AfterAll** for setup and teardown that runs once for all tests.
4. Use **@Disabled** to disable a test temporarily.

**Exercise 5: Structure of a Test Case**

**Objective:**

Understand the structure of a JUnit test case and proper naming conventions.

**Instructions:**

1. Write a test case with the following structure:
   * Arrange: Set up the initial conditions.
   * Act: Execute the method under test.
   * Assert: Verify the results.
2. Follow proper naming conventions for your test methods to reflect the behavior being tested.

**Exercise 6: Assertions**

**Objective:**

Use various JUnit assertions to verify test outcomes.

**Instructions:**

1. Write tests using assertions like **assertEquals**, **assertTrue**, **assertFalse**, etc.
2. Test various scenarios and edge cases for your methods.

**Exercise 7: @Suite and @SelectClasses**

**Objective:**

Group multiple test classes into a test suite.

**Instructions:**

1. Create multiple test classes for different components of your application.
2. Create a test suite class using **@Suite** and **@SelectClasses**.
3. Add your test classes to the test suite.
4. Run the test suite to execute all included tests.

**Exercise 8: Parameterized Tests**

**Objective:**

Write parameterized tests to run the same test with different inputs.

**Instructions:**

1. Create a test class that uses the **@ParameterizedTest** annotation.
2. Use **@ValueSource** or other sources to provide different input values.
3. Write a test method that uses these input values to test your method.

**Exercise 9: Test Suites and Categories**

**Objective:**

Organize tests into suites and categories for better management.

**Instructions:**

1. Create categories for different types of tests (e.g., unit tests, integration tests).
2. Annotate your test classes with these categories.
3. Create test suites for each category and include relevant test classes.

**Exercise 10: Test Execution Order**

**Objective:**

Control the order of test execution.

**Instructions:**

1. Use **@TestMethodOrder** to define the execution order of your tests.
2. Create test methods that depend on the execution order.
3. Verify the order in which the tests run.

**Exercise 11: Exception Testing**

**Objective:**

Write tests that verify exceptions are thrown under specific conditions.

**Instructions:**

1. Write a test method that expects an exception using **assertThrows**.
2. Create scenarios in your application where exceptions should be thrown.
3. Verify that the exceptions are correctly thrown and handled.

**Exercise 12: Timeout and Performance Testing**

**Objective:**

Write tests that verify methods complete within a specified time.

**Instructions:**

1. Use the **@Timeout** annotation to set a time limit for your test methods.
2. Write a test method that verifies a long-running process completes within the time limit.
3. Optimize the method to ensure it meets the performance requirements.

**Exercise 13: Introduction to Mockito**

**Objective:**

Get started with Mockito for mocking and stubbing dependencies.

**Instructions:**

1. Add Mockito dependencies to your project’s build file.
2. Create a service class with a dependency (e.g., **UserService** with a **UserRepository** dependency).
3. Use Mockito to mock the **UserRepository** in a test for UserService.
4. Stub methods on the mock object and verify interactions.

**Exercise 14: Mocking and Stubbing**

**Objective:**

Mock and stub methods using Mockito.

**Instructions:**

1. Create a test class for **UserService**.
2. Use **@Mock** and **@InjectMocks** annotations to inject mocks into your service.
3. Stub methods on the mock UserRepository to return specific values.
4. Verify that the service methods behave as expected.

**Exercise 15: Verifying Interactions**

**Objective:**

Verify that interactions with mock objects occur as expected.

**Instructions:**

1. Use verify to check that methods on the mock object are called with specific arguments.
2. Write test methods that perform actions on the service and verify interactions with the mock repository.

**Exercise 16: Argument Matching**

**Objective:**

Use argument matchers to verify method calls with specific arguments.

**Instructions:**

1. Use argument matchers like any(), eq(), etc., in your verify statements.
2. Write test methods that call service methods with different arguments and verify interactions.

**Exercise 17: Handling Void Methods**

**Objective:**

Mock and verify interactions with void methods.

**Instructions:**

1. Create a service class with a void method that interacts with a dependency.
2. Use **Mockito** to mock the dependency.
3. Verify that the void method is called with the correct arguments.

**Exercise 18: Overview of Spring Testing**

**Objective:**

Understand how to test Spring components (controllers, services, repositories).

**Instructions:**

1. Create a simple Spring Boot application with a controller, service, and repository.
2. Write unit tests for the service and repository layers.
3. Use **@MockBean** to mock dependencies in your tests.

**Exercise 19: Testing Controllers, Services, and Repositories**

**Objective:**

Write tests for Spring controllers, services, and repositories.

**Instructions:**

1. Create a REST controller in your Spring Boot application.
2. Write unit tests for the controller using **@WebMvcTest**.
3. Write unit tests for the service and repository layers using **@SpringBootTest**.

**Exercise 20: Integration Testing with Spring Boot**

**Objective:**

Perform integration testing for a Spring Boot application.

**Instructions:**

1. Create integration tests for your Spring Boot application using **@SpringBootTest**.
2. Use a test database for your integration tests.
3. Verify that the entire application works as expected.

**Exercise 21: Mocking Dependencies in Spring Tests**

**Objective:**

Mock dependencies in Spring tests using @MockBean.

**Instructions:**

1. Use **@MockBean** to mock a dependency in a Spring Boot test.
2. Write test methods that verify interactions with the mocked dependency.

**Exercise 22: Mocking Databases and Repositories**

**Objective:**

Mock database interactions and repositories in your tests.

**Instructions:**

1. Use **@MockBean** to mock the repository in your service tests.
2. Stub repository methods to return specific values.
3. Verify that the service methods interact with the repository as expected.

**Exercise 23: Mocking External Services**

**Objective:**

Mock external services (e.g., RESTful APIs) in your tests.

**Instructions:**

1. Use **@MockBean** to mock a REST client in your service tests.
2. Stub methods on the REST client to return specific responses.
3. Verify that the service methods interact with the REST client as expected.

**Exercise 24: Mocking File I/O and Network Interactions**

**Objective:**

Mock file I/O and network interactions in your tests.

**Instructions:**

1. Use Mockito to mock file I/O operations in your service tests.
2. Stub methods to simulate file reading and writing.
3. Verify that the service methods perform file operations correctly.

**Exercise 25: Strategies for Testing Code with External Dependencies**

**Objective:**

Develop strategies for testing code with external dependencies.

**Instructions:**

1. Identify external dependencies in your application (e.g., databases, REST APIs, file systems).
2. Develop a plan for mocking these dependencies in your tests.
3. Implement test methods that use mocks to verify the behavior of your code.