- **Explanation:** A test plan is a document outlining the scope, approach, resources, and schedule of testing activities. It serves as a guide for the testing process.

31. How do you perform usability testing?

- **Explanation:** Usability testing evaluates how user-friendly an application is. It involves real users performing tasks while observers note issues and gather feedback.

32. What is static testing?

- **Explanation:** Static testing involves examining code and documentation without executing the program. It includes code reviews, inspections, and static analysis tools.

33. How do you handle flaky tests?

- **Explanation:** Flaky tests are tests that yield inconsistent results. To handle them:
    - Investigate and identify the cause.
    - Isolate the flaky tests and run them separately.
    - Refactor the code or tests to eliminate flakiness.

34. What are code coverage tools, and why are they important?

- **Explanation:** Code coverage tools analyze which parts of the code are executed during testing. They are important for identifying untested paths and ensuring sufficient test coverage.

35. How do you document test results?

- **Explanation:** Test results can be documented using:
    - **Test Management Tools:** Tools like TestRail or Zephyr.
    - **Reports:** Generate reports that summarize test cases, pass/fail status, and issues found.
    - **Dashboards:** Use dashboards to visualize results for stakeholders.

This comprehensive list should help you understand key testing concepts and prepare you for your interviews! If you have further questions or need clarifications on specific topics, feel free to ask!

no man more detail in answer i need

Got it! Here's a more detailed explanation of each testing-related question, including examples, concepts, and additional context to help you understand better.

1. What is TDD (Test-Driven Development)?

- **Explanation:** TDD is a software development process where you write tests for your code before writing the actual code itself. It encourages better design, as you need to think about how your code will be used. The cycle consists of three steps:

1. **Red:** Write a test that defines a function or improvements of a function, which will fail initially because the function isn't implemented yet.
2. **Green:** Write the simplest code necessary to make the test pass.
3. **Refactor:** Clean up the code while ensuring that all tests still pass.

Example: Suppose you want to create a function `add` that adds two numbers:

```java
// Step 1: Write a failing test
@Test
public void testAdd() {
    assertEquals(5, add(2, 3)); // Fails initially
}

// Step 2: Write the minimal code to pass the test
public int add(int a, int b) {
    return a + b;
}

// Step 3: Refactor if needed, but the test should still pass
```

2. What are the different types of testing?

- **Explanation:** Different types of testing focus on various aspects of an application. Here's a breakdown:
    - **Unit Testing:** Tests individual components (functions or classes) in isolation. Each unit is tested independently.
        - **Example:** Testing a method that calculates taxes.
    - **Integration Testing:** Tests how different modules work together. This could be testing how a controller interacts with a service and the database.
        - **Example:** Checking if data flows correctly from the database to the user interface.
    - **Functional Testing:** Tests specific functionalities of the application against requirements.
        - **Example:** Testing whether a user can log in with valid credentials.
    - **Performance Testing:** Evaluates the application's performance under load. This can include stress testing and load testing.
        - **Example:** Measuring how many users can concurrently access the application.
    - **Load Testing:** A type of performance testing where you assess how the application behaves under normal and peak loads.
        - **Example:** Simulating 1,000 users logging in simultaneously.
    - **Stress Testing:** Determines the application's robustness by testing it under extreme conditions.
        - **Example:** Checking how many transactions can be processed before the system crashes.
    - **Regression Testing:** Ensures that new code changes do not adversely affect existing features. It's essential after changes are made to the application.
        - **Example:** Running previously successful test cases after a new feature is added.
    - **Exploratory Testing:** Manual testing where testers explore the application without predefined tests, relying on their intuition and experience to find defects.

3. How do you implement unit testing in Spring Boot?

- **Explanation:** In Spring Boot, unit testing is typically done using JUnit and Mockito. Here's a step-by-step guide:
    1. **Add Dependencies:** Ensure you have dependencies for JUnit and Mockito in your `` `pom.xml` `` or `` `build.gradle` ``.
    2. **Create Test Class:** Create a test class annotated with `` `@SpringBootTest` `` for integration testing or just `` `@ExtendWith(MockitoExtension.class)` `` for unit testing.
    3. **Mock Dependencies:** Use `` `@MockBean` `` to mock external dependencies, allowing you to test your service or component in isolation.
    4. **Write Test Cases:** Use the `` `@Test` `` annotation for your test methods and assert the expected outcomes.

  Example:

  ```java
  @SpringBootTest
  public class MyServiceTest {
      @MockBean
      private MyRepository myRepository; // Mock the repository

      @Autowired
      private MyService myService; // Autowire the service to be tested

      @Test
      public void testFindById() {
          // Arrange
          MyEntity entity = new MyEntity(1, "Test");
          when(myRepository.findById(1)).thenReturn(Optional.of(entity)); // Mock
  behavior

          // Act
          MyEntity result = myService.findById(1); // Call the method

          // Assert
          assertNotNull(result); // Check that the result is not null
          assertEquals("Test", result.getName()); // Verify the entity returned is
  correct
      }
  }
  ```

4. What is integration testing?

- **Explanation:** Integration testing evaluates how different parts of a system work together. It focuses on the interaction between modules and can help catch issues that are not visible in unit tests, such as data format mismatches or interaction failures.

  Example: You have a REST API that retrieves user data from a database. An integration test would verify:

    - The API endpoint correctly interacts with the database.
    - The correct data is returned in the expected format.

  Steps to Implement:

    1. Use `` `@SpringBootTest` `` to start the application context.
    2. Use `` `TestRestTemplate` `` or `` `MockMvc` `` to call the API endpoints.
    3. Assert that the response is as expected.

Example Code:

```java
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
public class UserControllerIntegrationTest {
    @Autowired
    private TestRestTemplate restTemplate;

    @Test
    public void testGetUser() {
        ResponseEntity<User> response = restTemplate.getForEntity("/users/1",
User.class);
        assertEquals(HttpStatus.OK, response.getStatusCode());
        assertEquals("John Doe", response.getBody().getName());
    }
}
```

5. How do you test a Spring Boot application?

- **Explanation:** Testing a Spring Boot application can be done at multiple levels, including unit testing, integration testing, and functional testing. Here’s how you can approach each type:
    - **Unit Testing:** Write tests for individual methods using JUnit and Mockito. Focus on business logic without involving Spring context.
    - **Integration Testing:** Use `` `@SpringBootTest` `` to load the complete application context and test interactions between components.
    - **Functional Testing:** Test the application's endpoints using `` `MockMvc` `` or `` `TestRestTemplate` ``, ensuring that the application meets user requirements.

6. What is the testing pyramid?

- **Explanation:** The testing pyramid is a model that emphasizes the balance of different types of tests. The three layers are:
    - **Unit Tests (Base):** These should be the most numerous, testing individual components or methods. They are fast to run and should cover all business logic.
    - **Integration Tests (Middle):** Fewer in number than unit tests, they test how components work together. They are slower than unit tests but still necessary for quality assurance.
    - **End-to-End Tests (Top):** The least numerous, these tests simulate real user scenarios to ensure the entire system functions as expected. They are the most time-consuming and fragile.

Visual Representation:

```lua
          E2E Tests
      ----------------
      Integration Tests
  ---------------------------
  Unit Tests (Most numerous)
```

7. What is the purpose of mocking in tests?

- **Explanation:** Mocking is a technique used in unit testing to simulate the behavior of complex objects, allowing you to isolate the unit under test. It helps in:
  - Testing how the unit interacts with dependencies.
  - Avoiding calls to external systems (like databases or APIs) during tests.
  - Ensuring that tests run quickly and consistently.

  Example: If a service fetches data from an external API, you can mock the API call to return a predefined response without making a real network call.

  Mocking with Mockito:

```java
@Mock
private ExternalService externalService;

@Test
public void testServiceMethod() {
    when(externalService.fetchData()).thenReturn("Mock Data");
    String result = myService.serviceMethod();
    assertEquals("Expected Result", result);
}
```

8. What is performance testing?

- **Explanation:** Performance testing evaluates how a system performs under load and is essential for understanding system behavior in real-world scenarios. It helps identify bottlenecks and ensures the application meets performance standards. Key aspects include:
  - **Response Time:** How long it takes to process a request.
  - **Throughput:** Number of requests processed in a given time frame.
  - **Resource Utilization:** CPU, memory, and network usage during testing.

  Types of Performance Testing:

  - **Load Testing:** Tests the system under expected user loads.
  - **Stress Testing:** Tests beyond normal operational capacity to identify breaking points.
  - **Endurance Testing:** Tests the system under sustained load over an extended period.

9. How do you improve TPS (Transactions Per Second)?

- **Explanation:** Improving Transactions Per Second (TPS) involves optimizing the application and its environment. Here are several strategies:
  1. **Optimize Database Queries:** Reduce query complexity, add indexes, and use efficient data retrieval techniques.
     - **Example:** Instead of a `SELECT *`, only fetch necessary columns.
  2. **Caching:** Store frequently accessed data in memory to reduce database load.
     - **Example:** Use an in-memory cache like Redis to cache user sessions.
  3. **Load Balancing:** Distribute incoming traffic across multiple servers to enhance performance.
     - **Example:** Use tools like Nginx or AWS Elastic Load Balancing.
  4. **Asynchronous Processing:** Offload long-running tasks to background jobs using message queues (e.g., RabbitMQ, Kafka).

    - **Example:** Instead of processing an image upload synchronously, enqueue the task for background processing.
5. **Profiling and Monitoring:** Use profiling tools to identify performance bottlenecks and optimize them.
    - **Example:** Use Java profilers like VisualVM or YourKit.

10. What tools do you use for performance testing?

- **Explanation:** Common tools for performance testing include:
    - **Apache JMeter:** A popular open-source tool for load testing web applications.
        - **Example:** Simulating multiple users accessing your application simultaneously.
    - **Gatling:** A powerful load testing tool that provides a Scala-based DSL for creating tests.
    - **LoadRunner:** A comprehensive tool for performance testing applications across various protocols.
    - **NeoLoad:** An enterprise-level tool for continuous performance testing and monitoring.
    - **k6:** A modern load testing tool for testing APIs and microservices.

11. What is JMeter, and how do you use it?

- **Explanation:** JMeter is an open-source tool designed for load testing and performance measurement. It can simulate multiple users interacting with web applications to assess their performance under stress.

  Basic Steps to Use JMeter:

  1. **Download and Install:** Download JMeter from the official Apache website and extract it.
  2. **Create a Test Plan:** Open JMeter, create a test plan, and add a Thread Group to define the number of users.
  3. **Add Samplers:** Add HTTP Request samplers to define the requests to be sent to your application.
  4. **Add Listeners:** Add listeners to capture and visualize the results (e.g., View Results Tree, Summary Report).
  5. **Run the Test:** Execute the test and analyze the results for response times, error rates, and throughput.

  Example: To test a login endpoint, set up a Thread Group with 100 users, add an HTTP Request with the login URL, and configure the parameters accordingly.

12. What are the benefits of continuous testing?

- **Explanation:** Continuous testing is the practice of executing automated tests throughout the software development lifecycle to provide immediate feedback. Benefits include:
    - **Early Detection of Defects:** Finding bugs early reduces the cost of fixing them later.
    - **Faster Release Cycles:** Automated tests speed up the testing process, allowing for more frequent releases.
    - **Increased Confidence:** Continuous testing provides a safety net, allowing teams to make changes with confidence.
    - **Improved Collaboration:** Development and testing teams work closely, leading to better communication and fewer misunderstandings.

13. How do you handle test data?

- **Explanation:** Managing test data is crucial for effective testing. Here are some approaches:
  - **Use Fixtures:** Create predefined data sets for testing. This can be done using SQL scripts or setup methods.
  - **Database Seeding:** Populate the database with test data before tests run.
  - **Mocking:** Use mocking frameworks to simulate external systems and data responses.
  - **Data Management Tools:** Use tools like Factory Boy or Faker to generate random test data programmatically.

  Example of Seeding Data in JUnit:

  ```java
  @BeforeEach
  public void setUp() {
      // Code to seed the database with test data
      userRepository.save(new User("test@example.com", "password"));
  }
  ```

14. What is regression testing, and why is it important?

- **Explanation:** Regression testing is the practice of running existing test cases after changes are made to ensure that new code hasn't introduced bugs in existing functionality. It is crucial because:
  - **Confidence in Changes:** Ensures that new features or fixes do not break existing functionality.
  - **Quality Assurance:** Maintains the overall quality of the software.
  - **Reduced Risk:** Identifies issues early, reducing the risk of failures in production.

15. What are the challenges of automated testing?

- **Explanation:** Automated testing has several challenges:
  - **Initial Investment:** Setting up automation requires time and resources for creating tests and infrastructure.
  - **Maintenance Overhead:** Tests can become outdated with code changes, requiring regular updates.
  - **False Positives/Negatives:** Automated tests may fail due to environmental issues, leading to confusion.
  - **Limited Coverage:** Not all scenarios can be easily automated, so manual testing is still necessary.

16. How do you document test cases?

- **Explanation:** Documenting test cases is essential for clarity and future reference. This can be done through:
  - **Test Management Tools:** Use tools like TestRail or Zephyr to create and manage test cases.
  - **Excel/Spreadsheets:** A simple approach is to maintain test cases in a spreadsheet format.
  - **Confluence/Wiki:** Document test cases in a shared space for team access.
  - **Version Control:** Keep test cases in version control along with the codebase.

  Typical Documentation Structure:

  - **Test Case ID:** Unique identifier for the test case.
  - **Description:** Brief overview of what the test case verifies.
  - **Preconditions:** Any setup needed before running the test.
  - **Steps to Execute:** Detailed steps to follow during testing.
  - **Expected Result:** What the outcome should be.
  - **Actual Result:** The result obtained when the test is executed.

17. What is a test strategy?

- **Explanation:** A test strategy outlines the testing approach and objectives for a project. It defines:
  - **Testing Scope:** What features and functionalities will be tested.
  - **Testing Types:** Types of testing to be performed (e.g., unit, integration, system).
  - **Resources:** The tools, environments, and personnel required for testing.
  - **Risk Assessment:** Identification of potential risks and how to mitigate them.

18. How do you ensure the reliability of your tests?

- **Explanation:** Ensuring the reliability of tests involves:
  - **Writing Clear Tests:** Ensure tests are well-defined and focused on a single functionality.
  - **Avoiding Flaky Tests:** Isolate tests from external dependencies and avoid statefulness.
  - **Regular Maintenance:** Update tests as the code changes to ensure they remain relevant.
  - **Using Assertions Wisely:** Use assertions effectively to verify expected outcomes.

19. What are smoke tests?

- **Explanation:** Smoke tests are a subset of test cases that check the basic functionality of an application to ensure that it is stable enough for further testing. They are often run after a new build or deployment.

  Example: In a web application, smoke tests might check if the homepage loads, the login works, and key features are accessible.

20. What is the role of a QA (Quality Assurance) engineer?

- **Explanation:** A QA engineer is responsible for ensuring that the software meets quality standards before release. Their responsibilities include:
  - **Designing Test Cases:** Creating test cases based on requirements.
  - **Executing Tests:** Running manual and automated tests to identify defects.
  - **Reporting Bugs:** Documenting and tracking defects found during testing.
  - **Collaboration:** Working with developers to understand features and ensure quality.
  - **Test Planning:** Contributing to the overall test strategy and planning activities.

21. How do you conduct usability testing?

- **Explanation:** Usability testing evaluates how user-friendly an application is by observing real users as they interact with it. Steps include:
  1. **Define Objectives:** Determine what aspects of usability you want to evaluate.
  2. **Select Participants:** Choose a representative group of users to participate in the testing.
  3. **Create Scenarios:** Develop scenarios for users to complete tasks within the application.
  4. **Observe and Record:** Observe users as they complete tasks, noting any issues or challenges they face.

5. **Gather Feedback:** Collect feedback through surveys or interviews to understand user satisfaction.

22. What is static testing?

- **Explanation:** Static testing involves examining code and documentation without executing the program. It includes techniques like:
    - **Code Reviews:** Peers review code for errors and adherence to coding standards.
    - **Static Code Analysis:** Automated tools analyze code for potential issues (e.g., SonarQube).
    - **Inspection:** Formal evaluations of documents or code to identify defects.

  Example: Using a tool like Checkstyle to ensure code adheres to defined coding standards.

23. How do you handle flaky tests?

- **Explanation:** Flaky tests are tests that yield inconsistent results. To handle them:
    - **Identify Causes:** Investigate and find the root cause of the flakiness (e.g., timing issues, dependency failures).
    - **Isolate Flaky Tests:** Run flaky tests separately to avoid interference with other tests.
    - **Refactor Code/Tests:** Modify tests or code to remove flakiness, such as increasing timeouts or adding proper setup and teardown processes.

24. What are code coverage tools, and why are they important?

- **Explanation:** Code coverage tools measure the percentage of code executed during testing. They help identify untested parts of the codebase, ensuring thorough testing.

  Common Tools:

    - **JaCoCo:** A Java code coverage library that integrates well with Maven and Gradle.
    - **Cobertura:** A code coverage tool for Java that provides detailed reports.

  Importance:

    - **Identify Gaps:** Helps developers identify untested code paths.
    - **Improve Quality:** Encourages writing more tests, leading to higher quality code.

25. How do you document test results?

- **Explanation:** Documenting test results is essential for tracking progress and communicating findings. Common methods include:
    - **Test Management Tools:** Use tools like TestRail to track and report results.
    - **Excel/Spreadsheets:** Maintain results in a spreadsheet format for simplicity.
    - **Automated Reports:** Generate reports from test automation frameworks (e.g., JUnit or TestNG).

  Typical Documentation Elements:
    - **Test Case ID:** Identifier for the test case.
    - **Test Execution Date:** When the test was run.
    - **Execution Status:** Pass/Fail result.
    - **Comments:** Additional notes about the execution (e.g., environment issues).

This list covers a wide range of topics, but feel free to ask if you want to dive deeper into any specific area or need examples for a particular question!

now add 15 most asked question on junit and mockito

Here's a list of 15 commonly asked questions related to JUnit and Mockito, along with detailed explanations and examples for each.

JUnit and Mockito Questions

1. What is JUnit, and why is it used?

- **Explanation:** JUnit is a widely used testing framework for Java that provides annotations and assertions to write and execute unit tests. It helps developers ensure that individual components of their code function correctly.
- **Use Cases:**
    - Validating business logic.
    - Ensuring code changes do not introduce new bugs.

2. What are the key annotations in JUnit?

- **Explanation:** Some of the key annotations in JUnit include:
    - `` `@Test` ``: Marks a method as a test method.
    - `` `@BeforeEach` ``: Runs before each test method to set up the test environment.
    - `` `@AfterEach` ``: Runs after each test method to clean up.
    - `` `@BeforeAll` ``: Runs once before all test methods in the class (requires static method).
    - `` `@AfterAll` ``: Runs once after all test methods in the class (requires static method).

  Example:

```java
@BeforeEach
public void setUp() {
    // Setup code here
}

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

@AfterEach
public void tearDown() {
    // Cleanup code here
}
```

3. How do you assert expected outcomes in JUnit?

- **Explanation:** Assertions in JUnit are used to verify the expected results of tests. Common assertions include:
    - `assertEquals(expected, actual)`: Checks if two values are equal.
    - `assertTrue(condition)`: Checks if a condition is true.
    - `assertFalse(condition)`: Checks if a condition is false.
    - `assertNotNull(object)`: Checks if an object is not null.

  Example:

  ```java
  @Test
  public void testAddition() {
      int sum = add(2, 3);
      assertEquals(5, sum);
  }
  ```

4. What is Mockito, and what is its purpose?

- **Explanation:** Mockito is a mocking framework for Java that allows you to create mock objects for unit testing. It helps isolate the unit being tested by simulating dependencies and their behavior.
- **Use Cases:**
    - Mocking external services or databases to avoid integration issues during testing.

5. How do you create a mock object in Mockito?

- **Explanation:** You can create mock objects using the `mock()` method provided by Mockito.

  Example:

  ```java
  List mockedList = Mockito.mock(List.class);
  ```

6. How do you specify behavior for a mock object in Mockito?

- **Explanation:** Use the `when()` method to specify the behavior of a mock when a particular method is called.

  Example:

  ```java
  when(mockedList.get(0)).thenReturn("Mocked Value");
  ```

7. How do you verify interactions with mock objects in Mockito?

- **Explanation:** You can verify that certain methods were called on a mock object using the `` `verify()` `` method.

  Example:

  ```java
  verify(mockedList).add("element");
  ```

8. What is the difference between `` `@Mock` `` and `` `Mockito.mock()` ``?

- **Explanation:**
    - `` `@Mock` ``: Annotations used to create mock objects automatically when using Mockito with a runner or extension.
    - `` `Mockito.mock()` ``: A method that explicitly creates a mock object.

  Example with `` `@Mock` ``:

  ```java
  @Mock
  private List mockedList;
  ```

9. What is the purpose of `` `@InjectMocks` `` in Mockito?

- **Explanation:** The `` `@InjectMocks` `` annotation is used to create an instance of the class under test and automatically inject mocks into it.

  Example:

  ```java
  @InjectMocks
  private MyService myService; // Service that uses mocked dependencies
  ```

10. How do you handle exceptions in JUnit tests?

- **Explanation:** You can use the `` `assertThrows()` `` method to assert that a specific exception is thrown during a test.

  Example:

  ```java
  @Test
  public void testException() {
      assertThrows(IllegalArgumentException.class, () -> {
          myService.methodThatThrowsException();
  ```

```
    });
}
```

11. How can you run JUnit tests from the command line?

- **Explanation:** You can run JUnit tests from the command line using Maven or Gradle. With Maven, you can use the command:

```bash
mvn test
```

12. What is a test double, and what types are there?

- **Explanation:** A test double is a generic term for any case where you replace a production object with a simplified version for testing. Types include:
    - **Stub:** Provides predefined responses to method calls.
    - **Mock:** Verifies interactions and can provide predefined responses.
    - **Spy:** A partial mock that allows tracking of real method calls.
    - **Fake:** A lightweight implementation of an interface.

13. How do you test private methods in JUnit?

- **Explanation:** It's generally not advisable to test private methods directly. Instead, you should test the public methods that call them. If necessary, you can use reflection, but it's better to design your code to keep methods public if they need to be tested.

14. How do you parameterize tests in JUnit?

- **Explanation:** JUnit provides parameterized tests to run the same test with different inputs. You can use the `` `@ParameterizedTest` `` annotation (JUnit 5) along with `` `@MethodSource` `` or `` `@ValueSource` ``.

  Example:

```java
@ParameterizedTest
@ValueSource(ints = {1, 2, 3})
public void testWithParameters(int number) {
    assertTrue(number > 0);
}
```

15. What are the best practices for writing unit tests?

- **Explanation:** Some best practices include:
    - **Keep Tests Isolated:** Each test should be independent of others.
    - **Use Meaningful Names:** Test method names should describe the behavior being tested.
    - **Test One Thing at a Time:** Each test should focus on one aspect of functionality.

- **Use Assertions Wisely:** Make sure assertions clearly express the intent of the test.
- **Run Tests Regularly:** Execute tests frequently to catch issues early.

Summary

This list covers essential concepts and questions related to JUnit and Mockito that you may encounter in interviews. If you want to delve deeper into any specific topic or need additional examples, feel free to ask!

explain some most commonly used annotation in junit and mockito

Here’s a detailed explanation of the most commonly used annotations in JUnit and Mockito, along with examples for each:

JUnit Annotations

1. `` `@Test` ``
    - **Purpose:** Marks a method as a test method that JUnit should execute.
    - **Usage:** You use this annotation to indicate which methods are test cases.
    - **Example:**

```java
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.assertEquals;

public class CalculatorTest {

    @Test
    public void testAddition() {
        int result = add(2, 3);
        assertEquals(5, result);
    }

    private int add(int a, int b) {
        return a + b;
    }
}
```

2. `` `@BeforeEach` ``
    - **Purpose:** Executes the annotated method before each test method in the current class.
    - **Usage:** Useful for setting up common test data or environment configurations.
    - **Example:**

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

```
public class UserServiceTest {
    private UserService userService;

    @BeforeEach
    public void setUp() {
        userService = new UserService();
    }

    @Test
    public void testUserCreation() {
        User user = userService.createUser("Alice");
        // Assertions go here
    }
}
```

3. `@AfterEach`
   - **Purpose:** Executes the annotated method after each test method in the current class.
   - **Usage:** Useful for cleanup activities after each test.
   - **Example:**

```java
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.Test;

public class DatabaseTest {
    private Database db;

    @BeforeEach
    public void setUp() {
        db = new Database();
        db.connect();
    }

    @Test
    public void testDatabaseQuery() {
        // Perform some query
    }

    @AfterEach
    public void tearDown() {
        db.disconnect();
    }
}
```

4. `@BeforeAll`
   - **Purpose:** Executes a method once before any of the test methods in the current class. The method must be static.
   - **Usage:** Useful for expensive setup operations that are shared across tests.
   - **Example:**

```java
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.Test;

public class ApplicationTest {
    @BeforeAll
    public static void setUpAll() {
        // Set up the application context
    }

    @Test
    public void testApplicationFeature() {
        // Test logic here
    }
}
```

5. **`@AfterAll`**
    - **Purpose:** Executes a method once after all the test methods in the current class have been executed. The method must be static.
    - **Usage:** Useful for teardown activities that are shared across tests.
    - **Example:**

```java
import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.Test;

public class ServerTest {
    @BeforeAll
    public static void startServer() {
        // Start server
    }

    @AfterAll
    public static void stopServer() {
        // Stop server
    }

    @Test
    public void testServerResponse() {
        // Test server response
    }
}
```

6. **`@Disabled`**
    - **Purpose:** Disables a test method or class, preventing it from being executed.
    - **Usage:** Useful for temporarily disabling tests under certain conditions.
    - **Example:**

```java
import org.junit.jupiter.api.Disabled;
import org.junit.jupiter.api.Test;

public class ExperimentalTest {
    @Test
    @Disabled("This test is disabled due to pending features")
    public void testPendingFeature() {
        // This test will not run
    }
}
```

Mockito Annotations

1. `@Mock`
   - **Purpose:** Creates a mock object for a specified class or interface.
   - **Usage:** Useful for replacing dependencies with mocks in unit tests.
   - **Example:**

```java
import static org.mockito.Mockito.mock;

public class UserServiceTest {
    @Mock
    private UserRepository userRepository;

    @BeforeEach
    public void setUp() {
        userRepository = mock(UserRepository.class);
    }
}
```

2. `@InjectMocks`
   - **Purpose:** Creates an instance of the class under test and injects mock objects into it.
   - **Usage:** Useful for testing a class with its dependencies automatically.
   - **Example:**

```java
import static org.mockito.Mockito.*;

public class UserServiceTest {
    @Mock
    private UserRepository userRepository;

    @InjectMocks
    private UserService userService;

    @BeforeEach
    public void setUp() {
        MockitoAnnotations.openMocks(this);
    }

    @Test
    public void testUserCreation() {
        User user = new User("Alice");
        when(userRepository.save(any(User.class))).thenReturn(user);

        User createdUser = userService.createUser("Alice");

        assertEquals("Alice", createdUser.getName());
        verify(userRepository).save(any(User.class));
    }
}
```

3. `@Spy`
   - **Purpose:** Creates a partial mock that allows you to call real methods while spying on the interactions.
   - **Usage:** Useful when you want to test a class while still using its real methods.
   - **Example:**

```java
import static org.mockito.Mockito.*;

public class ListTest {
    @Spy
    private ArrayList<String> spyList = new ArrayList<>();

    @Test
    public void testSpy() {
        spyList.add("one");
        spyList.add("two");

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

4. `` `@Captor` ``
   - **Purpose:** Creates a parameter captor that allows you to capture arguments passed to mock methods.
   - **Usage:** Useful for verifying interactions with mock objects.
   - **Example:**

```java
import static org.mockito.ArgumentCaptor.forClass;
import static org.mockito.Mockito.*;

public class UserServiceTest {
    @Mock
    private UserRepository userRepository;

    @Captor
    private ArgumentCaptor<User> userCaptor;

    @InjectMocks
    private UserService userService;

    @Test
    public void testUserCreation() {
        userService.createUser("Alice");
        verify(userRepository).save(userCaptor.capture());
        assertEquals("Alice", userCaptor.getValue().getName());
    }
}
```

5. `` `@ExtendWith` `` (JUnit 5)
   - **Purpose:** Used to register Mockito extension to enable its annotations.
   - **Usage:** Required when using Mockito with JUnit 5 to initialize mocks and inject them.
   - **Example:**

```java
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.junit.jupiter.MockitoExtension;

@ExtendWith(MockitoExtension.class)
public class UserServiceTest {
    // Test class content
}
```

Summary

These annotations are essential for effective testing in Java using JUnit and Mockito. They help organize your tests, manage setup and teardown, create mock objects, and streamline testing processes. Understanding how to use these annotations will significantly enhance your testing capabilities in Java applications. If you need further clarification or examples, feel free to ask!

ChatGPT can make mistakes. Check important info.