Basics and Advanced Questions on Unit Testing and Scenario based

🔹 Basic Interview Questions on JUnit and Unit Testing

1. What is unit testing?

Answer:
Unit testing is a software testing method where individual units or components of a software are tested in isolation to ensure they work as expected. In Java, JUnit is one of the most popular frameworks for unit testing.

2. What is JUnit?

Answer:
JUnit is an open-source unit testing framework for Java. It provides annotations and assertions to write and run repeatable tests.

---

3. What are the advantages of using JUnit?

Answer:

- Early bug detection
- Simplifies regression testing
- Supports automation
- Integrates with build tools like Maven and Gradle
- Works well with CI/CD pipelines

🔹 JUnit Annotations with Examples

1. @Test

Purpose: Marks a method as a test method.

Example:

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

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

public class CalculatorTest {

  @Test

  void testAddition() {

    Calculator calc = new Calculator();   -- that calculator class is defined already then we are using it

    assertEquals(5, calc.add(2, 3));
```

```
    }
}
```

---

2. @BeforeEach

Purpose: Runs before each test method.

Example:

```
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
public class CalculatorTest {
    Calculator calc;
    @BeforeEach
    void setUp() {
        calc = new Calculator();
    }
    @Test
    void testSubtraction() {
        assertEquals(1, calc.subtract(3, 2));
    }
}
```

3. @AfterEach

Purpose: Runs after each test method.

Example:

```
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.Test;
public class CalculatorTest {
    Calculator calc;
    @Test
    void testMultiplication() {
        calc = new Calculator();
```

```
    assertEquals(6, calc.multiply(2, 3));
  }
  @AfterEach
  void tearDown() {
    calc = null;
  }
}
```

---

4. @BeforeAll

Purpose: Runs once before all test methods in the class. Must be static.

Example:

```
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.Test;
public class CalculatorTest {
  static Calculator calc;
  @BeforeAll
  static void init() {
    calc = new Calculator();
  }
  @Test
  void testDivision() {
    assertEquals(2, calc.divide(4, 2));
  }
}
```

---

5. @AfterAll

Purpose: Runs once after all test methods in the class. Must be static.

Example:

```
import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.Test;
public class CalculatorTest {
```

```
    static Calculator calc;
    @Test
    void testPower() {
        assertEquals(8, calc.power(2, 3));
    }
    @AfterAll
    static void cleanup() {
        calc = null;
    }
}
```

---

6. @Disabled

Purpose: Disables a test method or class.

Example:

```
import org.junit.jupiter.api.Disabled;
import org.junit.jupiter.api.Test;
public class CalculatorTest {
    @Disabled("Not implemented yet")
    @Test
    void testModulo() {
        // test code here
    }
}
```

---

7. @DisplayName

Purpose: Provides a custom name for the test method.

Example:

```
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;
public class CalculatorTest {
    @Test
```

```
    @DisplayName("Addition of two positive numbers")
    void testAddPositiveNumbers() {
        Calculator calc = new Calculator();
        assertEquals(7, calc.add(3, 4));
    }
}
```

Explain the lifecycle annotations in JUnit 5.

Annotation	Purpose	Example
@BeforeAll	Runs once before all tests	static void init()
@BeforeEach	Runs before each test	void setUp()
@Test	Marks a method as a test	void testMethod()
@AfterEach	Runs after each test	void tearDown()
@AfterAll	Runs once after all tests	static void cleanup()

Advanced JUnit & Mockito Interview Questions and Answers

✅ 1. How do you use @Mock, @InjectMocks, and @Spy in Mockito?

Answer:

- @Mock: Creates a mock instance of a class.
- **Purpose**: Creates a mock instance of a class or interface.
- **Usage**: Used to mock dependencies of the class under test.
- @InjectMocks: Injects mock dependencies into the class under test.
- @Spy: Wraps a real object but allows selective mocking.

Example:

```
@ExtendWith(MockitoExtension.class)
public class OrderServiceTest {
    @Mock
    private PaymentService paymentService;
    @InjectMocks
```

```
    private OrderService orderService;
    @Test
    void testPlaceOrder() {
        when(paymentService.processPayment(anyDouble())).thenReturn(true);
        boolean result = orderService.placeOrder(100.0);
        assertTrue(result);
    }
}
```

✅ 2. What is the difference between @Mock and @Spy?

Feature	@Mock	@Spy
Behavior	Fake object	Real object with partial mocking
Use case	When you want full control	When you want to override specific methods

✅ 3. How do you verify method calls in Mockito?

Answer: Use verify() to check if a method was called.

```
verify(paymentService).processPayment(100.0);
verify(paymentService, times(1)).processPayment(anyDouble());
```

✅ 4. How do you write parameterized tests in JUnit 5?

Parameterized tests are used in unit testing to run the same test logic multiple times with different inputs, making your tests more efficient, readable, and maintainable.

Answer: Use @ParameterizedTest with sources like @ValueSource, @CsvSource, or @MethodSource, **@EnumSource**, **@CsvFileSource**, **@ArgumentsSource**

Example:

```
@ParameterizedTest
@CsvSource({"2,3,5", "1,1,2"})
void testAddition(int a, int b, int expected) {
    assertEquals(expected, calculator.add(a, b));
}
```

✅ 5. How do you test exception scenarios?

Answer: Use assertThrows() to verify exceptions.

```
@Test
void testDivideByZero() {
    assertThrows(ArithmeticException.class, () -> calculator.divide(10, 0));
}
```

✅ 6. How do you mock static methods using Mockito?

Answer: Use Mockito.mockStatic() (available in Mockito 3.4+).

```
try (MockedStatic mocked = Mockito.mockStatic(Utils.class)) {
    mocked.when(() -> Utils.getCurrentDate()).thenReturn("2025-01-01");
    assertEquals("2025-01-01", Utils.getCurrentDate());
}
```

✅ 7. How do you test asynchronous code or methods returning CompletableFuture?

Answer: Use CompletableFuture.get() or join() and assertions.

```
@Test
void testAsyncMethod() {
    CompletableFuture future = service.getDataAsync();
    assertEquals("data", future.join());
}
```

✅ 8. How do you use @Nested tests in JUnit 5?

Answer: @Nested allows grouping related tests for better readability.

```
@Nested
class AdditionTests {
    @Test
    void testPositiveNumbers() {
        assertEquals(5, calculator.add(2, 3));
```

```
    }
}
```

✅ 9. How do you use @TestInstance(Lifecycle.PER_CLASS) and why?

Answer: It allows using non-static @BeforeAll and @AfterAll.

```
@TestInstance(TestInstance.Lifecycle.PER_CLASS)
public class CalculatorTest {
    @BeforeAll
    void setup() {
        // non-static setup
    }
}
```

✅ 10. How do you ensure high-quality unit tests in a real-world project?

Answer:

- Follow **AAA pattern** (Arrange, Act, Assert)
- Use **mocking** for external dependencies
- Ensure **code coverage** with tools like **JaCoCo**
- Avoid testing implementation details
- Use **parameterized tests** for input variations
- Integrate with **CI/CD pipelines**

🔹 Bonus: Real-World Scenario-Based Questions

🔸 Q: How would you test a service that calls an external REST API?

Answer:

- Mock the HTTP client (e.g., RestTemplate)
- Use @MockBean in Spring Boot tests
- Verify response handling and error scenarios

🔸 **Q: How do you test a method that writes to a database?**

Answer:

- Use **mock repositories** or **in-memory DB** (e.g., H2)
- Use @DataJpaTest for repository layer
- Use @Transactional to rollback after tests

Here's a **real-world advanced unit testing scenario** for a **Spring Boot Java developer with 5+ years of experience**, explained in a clear and human-friendly way with code examples.

✅ **Scenario: Testing a Spring Boot Service Layer with External API and Database**

🔹 **Context:**

You're working on an e-commerce application. The OrderService places an order by:

1. Validating the product from the database.
2. Calling an external payment API.
3. Saving the order to the database.

You need to **unit test** this service without actually calling the database or external API.

✅ **Classes Involved**

🔸 **OrderService.java**

```
@Service
public class OrderService {
   private final ProductRepository productRepository;
   private final PaymentClient paymentClient;
   private final OrderRepository orderRepository;
   public OrderService(ProductRepository productRepository, PaymentClient paymentClient, Or
derRepository orderRepository) {
      this.productRepository = productRepository;
      this.paymentClient = paymentClient;
      this.orderRepository = orderRepository;
   }
   public boolean placeOrder(Long productId, double amount) {
      Product product = productRepository.findById(productId)
```

```
            .orElseThrow(() -> new IllegalArgumentException("Product not found"));
    boolean paymentSuccess = paymentClient.processPayment(amount);
    if (paymentSuccess) {
      Order order = new Order(product.getName(), amount);
      orderRepository.save(order);
      return true;
    }
    return false;
  }
}
```

✅ **Unit Test: OrderServiceTest.java**

🔹 **Goals:**

- Mock the database and external API.
- Verify behavior and interactions.
- Test both success and failure paths.

🔸 **Test Code:**

```
@ExtendWith(MockitoExtension.class)
class OrderServiceTest {
  @Mock
  private ProductRepository productRepository;
  @Mock
  private PaymentClient paymentClient;
  @Mock
  private OrderRepository orderRepository;
  @InjectMocks
  private OrderService orderService;
  @Test
  void testPlaceOrderSuccess() {
    Product product = new Product(1L, "Laptop");
    when(productRepository.findById(1L)).thenReturn(Optional.of(product));
    when(paymentClient.processPayment(1000.0)).thenReturn(true);
```

```
    boolean result = orderService.placeOrder(1L, 1000.0);
    assertTrue(result);
    verify(orderRepository).save(any(Order.class));
  }
  @Test
  void testPlaceOrderProductNotFound() {
    when(productRepository.findById(2L)).thenReturn(Optional.empty());
    assertThrows(IllegalArgumentException.class, () -> orderService.placeOrder(2L, 500.0));
    verify(paymentClient, never()).processPayment(anyDouble());
    verify(orderRepository, never()).save(any(Order.class));
  }
  @Test
  void testPlaceOrderPaymentFailed() {
    Product product = new Product(3L, "Phone");
    when(productRepository.findById(3L)).thenReturn(Optional.of(product));
    when(paymentClient.processPayment(800.0)).thenReturn(false);
    boolean result = orderService.placeOrder(3L, 800.0);
    assertFalse(result);
    verify(orderRepository, never()).save(any(Order.class));
  }
}
```

✅ **Key Concepts Demonstrated**

Concept	Explanation
@Mock	Simulates dependencies like DB and API
@InjectMocks	Injects mocks into the service
when(...).thenReturn(...)	Defines mock behavior
verify(...)	Ensures expected interactions

Concept	Explanation
assertThrows(...)	Validates exception handling
assertTrue/assertFalse	Validates logic outcomes

✅ **Industry Best Practices**

1. **Isolate the unit**: Don’t hit real DB or APIs.
2. **Use meaningful test names**: Describe the scenario.
3. **Test edge cases**: Missing data, failed payments.
4. **Verify interactions**: Ensure correct methods are called.
5. **Keep tests fast and independent**.