JUnit Framework & SQL Fundamentals for QA and Automation Testing

A comprehensive instructor-led training programme designed to equip QA engineers and automation testers with essential JUnit and SQL skills for effective test automation and database validation.

SECTION 1: INTRODUCTION TO JUNIT

What is JUnit?

JUnit is an open-source Java testing framework that enables developers and QA engineers to write and execute repeatable automated tests. It provides a structured approach to unit testing with annotations, assertions, and test runners that validate application behaviour systematically.

Originally created by Kent Beck and Erich Gamma, JUnit has become the industry standard for Java-based test automation. It integrates seamlessly with build tools like Maven and Gradle, and supports continuous integration pipelines.

Key Characteristics

- Open-source and widely adopted
- Annotation-driven test structure
- Built-in assertion library
- Integration with CI/CD tools
- Test lifecycle management

SECTION 1: INTRODUCTION TO JUNIT

Why JUnit is Used in Automation Testing

Repeatability

Tests execute consistently across environments, ensuring reliable validation every time code changes. This eliminates human error and provides confidence in regression testing.

Early Defect Detection

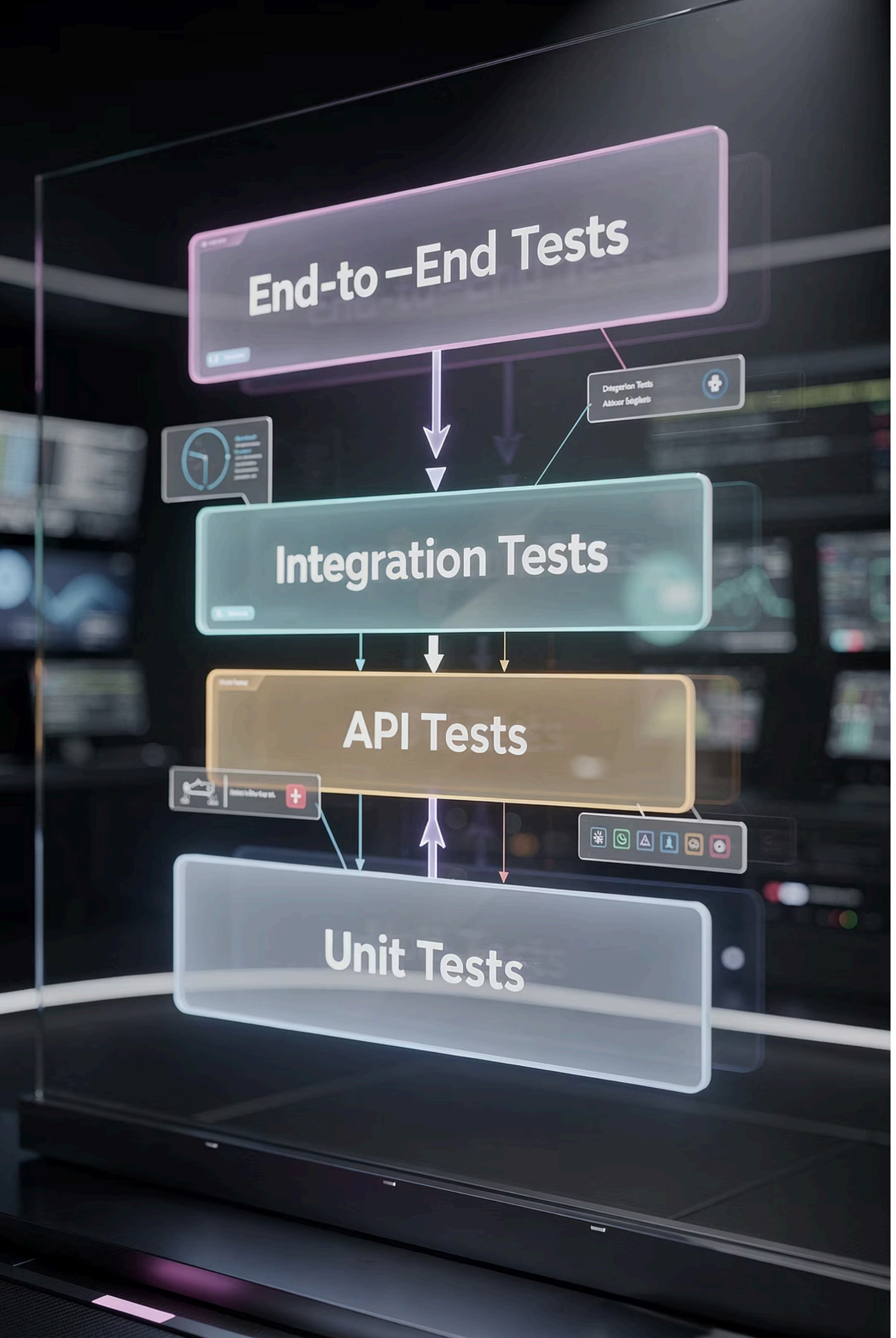
Automated tests run immediately after code commits, identifying issues before they reach later stages. This reduces cost and time spent on debugging production defects.

Integration Support

JUnit integrates with Jenkins, GitLab CI, Maven, and Gradle, enabling automated test execution in continuous integration pipelines without manual intervention.

Framework Foundation

JUnit serves as the foundation for frameworks like TestNG and works alongside Selenium, RestAssured, and Cucumber to support functional and API testing efforts.



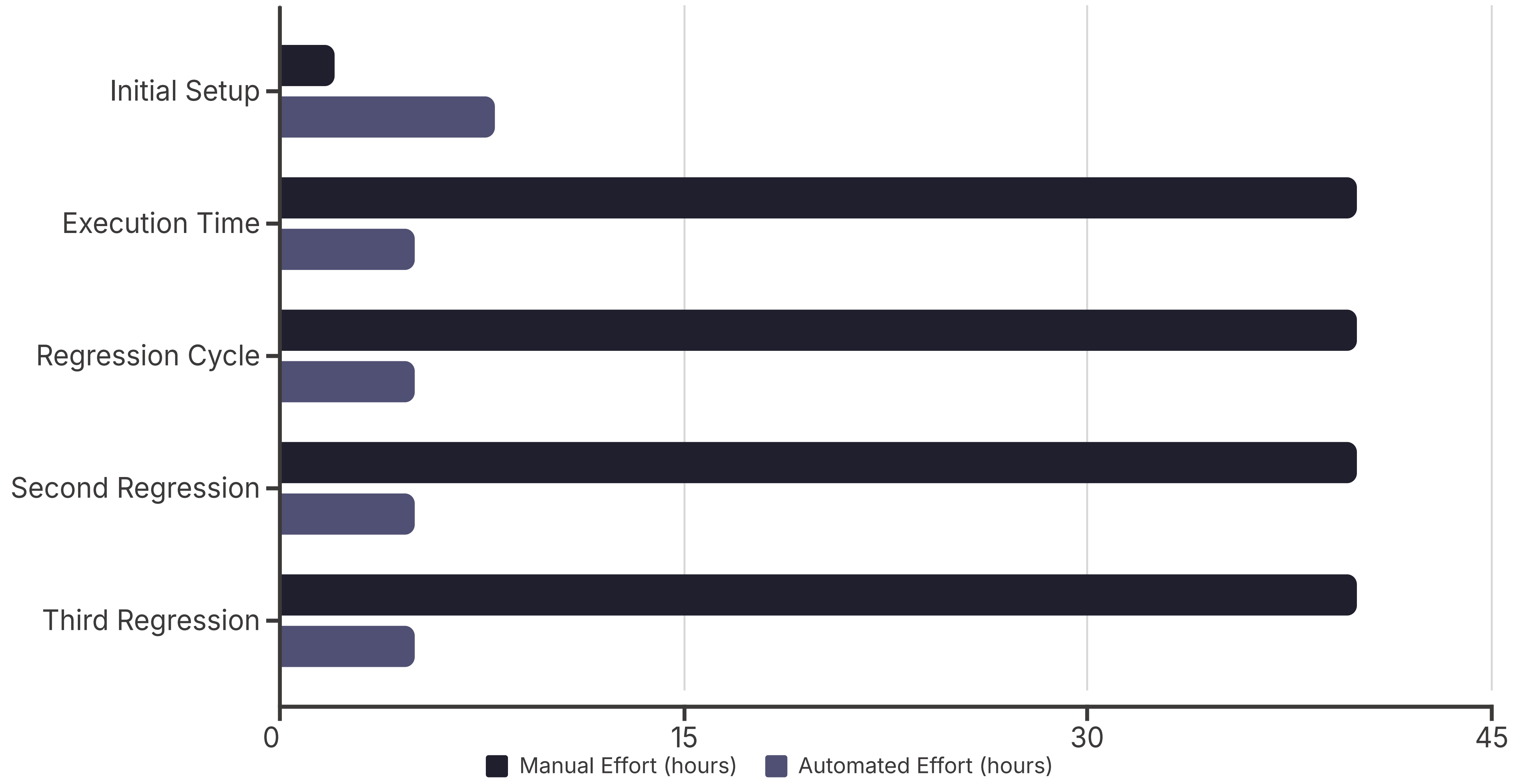
SECTION 1: INTRODUCTION TO JUNIT

Where JUnit Fits in Test Automation Frameworks

JUnit operates at the unit and integration testing layers of the test automation pyramid. It validates individual methods, classes, and modules before full system integration. In a typical automation framework, JUnit manages test execution, whilst Selenium handles browser interactions, RestAssured manages API calls, and reporting tools like Allure present results.

Modern test frameworks leverage JUnit's annotations and lifecycle hooks to structure tests, manage test data, and control execution flow. Page Object Model (POM) frameworks use JUnit to organise test classes, whilst data-driven frameworks utilise JUnit's parameterised tests for executing multiple test scenarios efficiently.

Manual Testing vs JUnit-Based Automation



Manual testing requires consistent effort for each execution cycle, whilst JUnit-based automation demands higher initial investment but dramatically reduces recurring effort. After three regression cycles, automation achieves cost parity and subsequently delivers substantial time savings.

Knowledge Check: Introduction to JUnit

1

Question 1

Which of the following best describes JUnit's primary purpose?

- A) Managing database connections
- B) Writing and executing automated unit tests
- C) Designing user interfaces
- D) Deploying applications to production

Correct Answer: B) Writing and executing automated unit tests

2

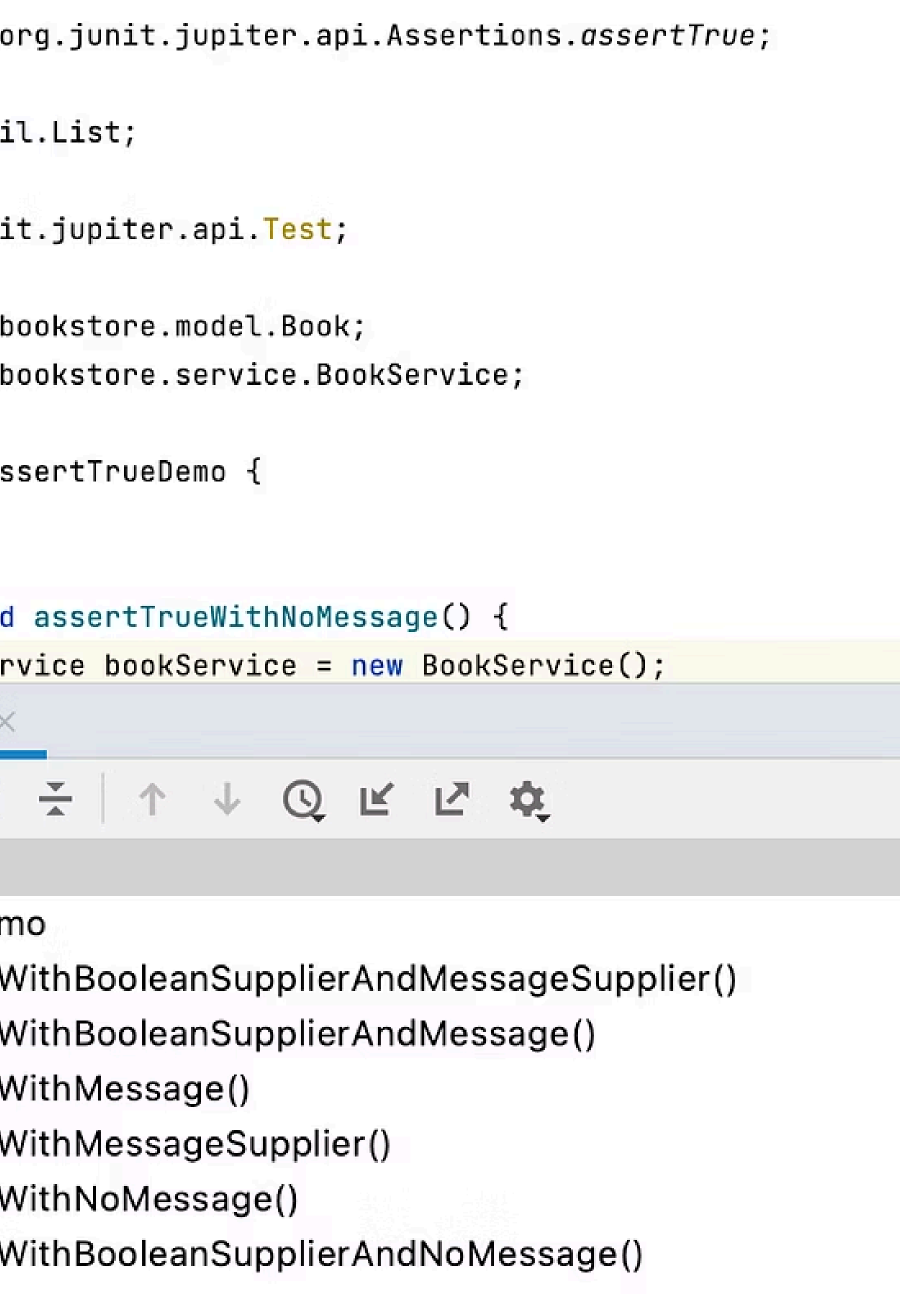
Question 2

In the test automation pyramid, where does JUnit primarily operate?

- A) UI testing layer only
- B) Unit and integration testing layers
- C) Performance testing layer
- D) Security testing layer

Correct Answer: B) Unit and integration testing layers

Interview Keywords: Unit testing, test framework, automation, test lifecycle, continuous integration, regression testing, test-driven development


```
org.junit.jupiter.api.Assertions.assertTrue;

il.List;

it.jupiter.api.Test;

bookstore.model.Book;
bookstore.service.BookService;

ssertTrueDemo {


d assertTrueWithNoMessage() {
rvice bookService = new BookService();
```

```
mo
WithBooleanSupplierAndMessageSupplier()
WithBooleanSupplierAndMessage()
WithMessage()
WithMessageSupplier()
WithNoMessage()
WithBooleanSupplierAndNoMessage()
```

SECTION 2: JUNIT ANNOTATIONS

Understanding JUnit Annotations

Annotations in JUnit are special markers prefixed with @ that provide metadata to the JUnit framework about how and when to execute test methods. They eliminate the need for manual test configuration and enable declarative test structure, making test code more readable and maintainable.

Rather than implementing complex interfaces or extending base classes, developers simply add annotations to methods, and JUnit handles the execution logic automatically. This annotation-driven approach is central to modern Java testing practices and reduces boilerplate code significantly.

SECTION 2: JUNIT ANNOTATIONS

Core JUnit Annotations Overview

@Test

Marks a method as a test case that JUnit should execute. The method must be public and void. Each @Test method runs independently.

@BeforeEach

Executes before every @Test method. Used for test setup like initialising objects or opening connections. Runs multiple times per test class.

@AfterEach

Executes after every @Test method. Used for cleanup activities like closing connections or releasing resources. Ensures no test pollution.

@BeforeAll

Executes once before all test methods. Must be static. Used for expensive setup like database connections or loading configuration files.

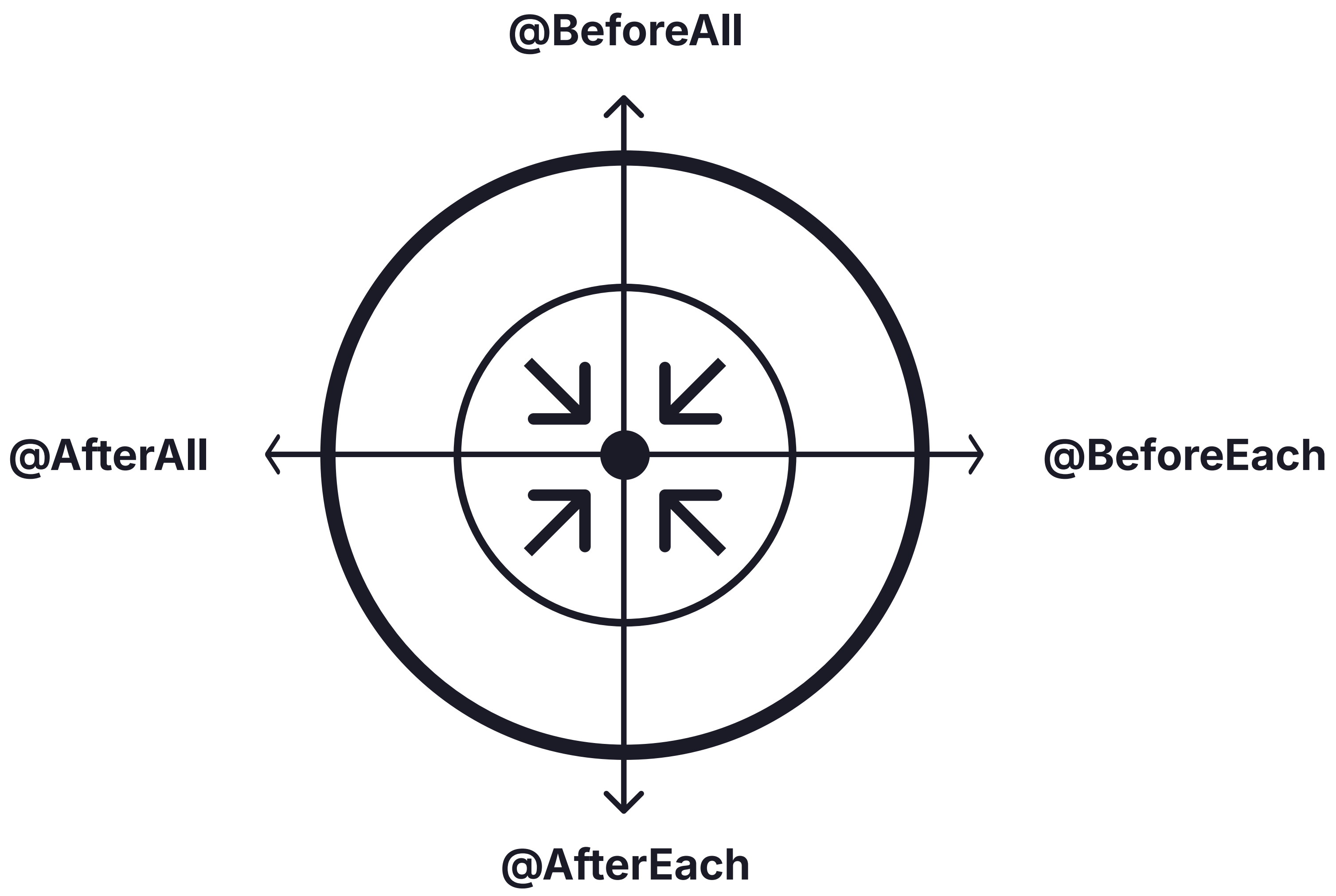
@AfterAll

Executes once after all test methods. Must be static. Used for final cleanup like closing database connections or deleting temporary files.

@Disabled

Temporarily disables a test method or entire test class. Used when tests are failing due to known bugs or incomplete features requiring investigation.

JUnit Annotation Execution Flow



Understanding execution order is critical for writing reliable tests. @BeforeAll and @AfterAll execute only once per test class, whilst @BeforeEach and @AfterEach execute before and after every single test method. This ensures proper test isolation and prevents one test from affecting another.

Practical Code Example: Annotations in Action

```java
import org.junit.jupiter.api.*;

public class LoginTest {

    @BeforeAll
    static void setupClass() {
        System.out.println("Setting up test suite - runs once");
        // Load configuration, establish DB connection
    }

    @BeforeEach
    void setUp() {
        System.out.println("Setting up test - runs before each test");
        // Initialize browser, navigate to login page
    }

    @Test
    void testValidLogin() {
        System.out.println("Executing valid login test");
        // Test logic here
    }

    @Test
    void testInvalidLogin() {
        System.out.println("Executing invalid login test");
        // Test logic here
    }

    @AfterEach
    void tearDown() {
        System.out.println("Cleaning up - runs after each test");
        // Close browser, clear session
    }

    @AfterAll
    static void tearDownClass() {
        System.out.println("Final cleanup - runs once");
        // Close DB connection, delete test data
    }
}
```

This example demonstrates proper annotation usage. The console output reveals execution order, helping testers understand the test lifecycle.

Real-World Use Cases for Annotations

@BeforeAll: Database Connection

Establishing database connections is expensive. Use @BeforeAll to create one connection shared across all tests, improving execution speed by eliminating redundant connection overhead.

@BeforeEach: Browser Initialisation

Each UI test needs a fresh browser instance to prevent test interference. @BeforeEach ensures every test starts with clean browser state, avoiding false failures from cached data.

@AfterEach: Data Cleanup

Tests that create records must clean up afterwards. @AfterEach deletes test data, ensuring the database remains clean and subsequent tests don't encounter unexpected records.

@Disabled: Known Defects

When a feature has a confirmed bug, use @Disabled with a comment referencing the bug ticket. This prevents false negatives in test reports whilst preserving the test for future use.

Knowledge Check: JUnit Annotations

1

Question 1

Which annotation executes only once before all test methods in a class?

- A) @BeforeEach
- B) @BeforeAll
- C) @Test
- D) @Setup

Correct Answer: B) @BeforeAll

2

Question 2

What is the primary purpose of @AfterEach annotation?

- A) To execute before each test method
- B) To mark a method as a test
- C) To perform cleanup after each test method
- D) To disable a test temporarily

Correct Answer: C) To perform cleanup after each test method

3

Question 3

Which annotation must be used with static methods?

- A) @Test and @Disabled
- B) @BeforeEach and @AfterEach
- C) @BeforeAll and @AfterAll
- D) All annotations require static methods

Correct Answer: C) @BeforeAll and @AfterAll

Interview Tip: Always explain the difference between setup (@BeforeEach) and teardown (@AfterEach). Employers value candidates who understand test isolation and cleanup strategies.

SECTION 3: JUNIT ASSERTIONS

What Are Assertions?

Assertions are statements that verify expected outcomes against actual results during test execution. They form the validation layer of automated tests, determining whether a test passes or fails. Without assertions, tests would execute actions but never verify correctness, rendering them useless for quality assurance.

When an assertion fails, JUnit immediately stops the test method and marks it as failed, logging detailed information about the expected versus actual values. This rapid feedback mechanism enables developers to identify defects quickly and understand precisely what went wrong during test execution.

Critical Concept: Tests without assertions are merely scripts that exercise code. Assertions transform scripts into meaningful validation that protects application quality.

Core JUnit Assertion Methods

assertEquals

Verifies two values are equal using .equals() method. Most commonly used assertion for comparing strings, numbers, and objects.

```
assertEquals(expected, actual)
```

assertNotEquals

Verifies two values are not equal. Useful for validating that operations produced different results than the original state.

```
assertNotEquals(value1, value2)
```

assertTrue

Verifies a condition evaluates to true. Essential for boolean validations like checking if elements are visible or enabled.

```
assertTrue(condition)
```

assertFalse

Verifies a condition evaluates to false. Used to confirm negative scenarios like verifying elements are hidden or inactive.

```
assertFalse(condition)
```

assertNull

Verifies an object reference is null. Important for validating that optional values are absent or objects have been properly cleared.

```
assertNull(object)
```

assertNotNull

Verifies an object reference is not null. Critical for confirming that required objects were successfully created or retrieved.

```
assertNotNull(object)
```

Practical Assertion Examples

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

public class UserServiceTest {

    @Test
    void testUserCreation() {
        UserService service = new UserService();
        User user = service.createUser("john@example.com", "password123");

        // Verify user object was created
        assertNotNull(user, "User should not be null");

        // Verify correct email was assigned
        assertEquals("john@example.com", user.getEmail(),
                    "Email should match input");

        // Verify user is active by default
        assertTrue(user.isActive(), "New user should be active");

        // Verify user has no admin privileges
        assertFalse(user.isAdmin(), "New user should not be admin");

        // Verify user ID was generated
        assertNotNull(user.getId(), "User ID should be generated");

        // Verify default role is not admin
        assertNotEquals("ADMIN", user.getRole(),
                       "Default role should not be admin");
    }
}
```

This example demonstrates multiple assertions validating different aspects of user creation. Each assertion includes a descriptive message for clarity when failures occur.

SECTION 3: JUNIT ASSERTIONS

Expected vs Actual Values: Order Matters

Correct Usage

```
// Pattern: assertEquals(expected, actual)

String expected = "Welcome";
String actual = getPageTitle();
assertEquals(expected, actual);

// Clear failure message:
// Expected: Welcome
// Actual: Welcom
```

Following the expected-then-actual pattern produces clear error messages that immediately reveal what should have happened versus what actually occurred.

Incorrect Usage

```
// WRONG: assertEquals(actual, expected)

String expected = "Welcome";
String actual = getPageTitle();
assertEquals(actual, expected);

// Confusing failure message:
// Expected: Welcom
// Actual: Welcome
```

Reversing the order creates confusion. The error message suggests the wrong value is correct, making debugging significantly more difficult.

Best Practice: Always place the expected value first, actual value second. This convention is universally understood and produces intuitive failure messages.

Common Assertion Mistakes

Missing Assertions

Tests that perform actions without validating outcomes provide no value. Every test must include at least one assertion verifying expected behaviour. A test without assertions always passes, even when functionality is broken.

Weak Assertions

Using assertTrue(result != null) when assertNotNull(result) is available. Weak assertions reduce code readability and produce less informative failure messages. Always use the most specific assertion available.

Too Many Assertions

Tests with 10+ assertions validate too many concerns in one method, making failures difficult to diagnose. Split complex tests into multiple focused tests, each validating a single aspect of functionality.

Missing Failure Messages

Assertions without descriptive messages force developers to read test code to understand failures. Always include clear failure messages as the third parameter: assertEquals(expected, actual, "Description of what failed").

Case Study: The Danger of Weak Assertions

Scenario: A payment processing test appears to pass consistently, giving false confidence in the payment module. During production deployment, multiple payment failures occur, causing significant revenue loss.

Weak Test (Dangerous)

```
@Test
void testPaymentProcessing() {
    Payment payment =
        processPayment(100.00);

    // Weak assertion
    assertNotNull(payment);

    // Test passes even when:
    // - Amount is wrong
    // - Status is failed
    // - Transaction ID missing
}
```

Strong Test (Reliable)

```
@Test
void testPaymentProcessing() {
    Payment payment =
        processPayment(100.00);

    // Comprehensive validation
    assertNotNull(payment);
    assertEquals(100.00,
                 payment.getAmount());
    assertEquals("SUCCESS",
                 payment.getStatus());
    assertNotNull(
        payment.getTransactionId());
}
```

Lesson: The weak test only verified a payment object existed, not that it was correct. Comprehensive assertions would have caught the defects before production, preventing customer impact and revenue loss.

SECTION 3: JUNIT ASSERTIONS

Knowledge Check: JUnit Assertions

1

Question 1

What is the correct syntax for assertEquals with a custom failure message?

- A) assertEquals(actual, expected, message)
- B) assertEquals(expected, actual, message)
- C) assertEquals(message, expected, actual)
- D) assertEquals(expected, message, actual)

Correct Answer: B) assertEquals(expected, actual, message)

2

Question 2

Which assertion should you use to verify an object reference is not null?

- A) assertTrue(object != null)
- B) assertNotNull(object)
- C) assertFalse(object == null)
- D) All are equally good

Correct Answer: B) assertNotNull(object) - Most specific and produces clearest failure messages

SECTION 4: JUNIT TEST LIFECYCLE

Understanding the JUnit Test Lifecycle

The test lifecycle describes the sequence of events that occur when JUnit executes tests, from initial discovery through final reporting. Understanding this lifecycle is crucial for writing reliable tests, debugging failures, and optimising test execution performance.

Each phase serves a specific purpose in ensuring tests execute correctly and independently. Misunderstanding lifecycle phases leads to common issues like test pollution, where one test affects another, or resource leaks, where connections remain open after test completion.

Five Phases of JUnit Test Lifecycle

Test Discovery

JUnit scans the classpath for classes containing @Test annotations. It identifies all test methods and builds an execution plan based on test ordering rules and configurations.

Setup Phase

@BeforeAll executes once for expensive setup. Then, for each test, @BeforeEach executes to prepare test-specific prerequisites like initialising objects or opening connections.

Test Execution

The @Test method executes, performing actions and assertions. If any assertion fails or exception is thrown, the test immediately fails and execution stops for that test method.

Teardown Phase

@AfterEach executes after each test to perform cleanup like closing connections or deleting test data. Finally, @AfterAll executes once after all tests complete for final cleanup.

Result Reporting

JUnit aggregates results from all tests, reporting passed, failed, and skipped tests. Detailed failure information including stack traces and failure messages is compiled for review.

SECTION 4: JUNIT TEST LIFECYCLE

Mapping Annotations to Lifecycle Phases

Lifecycle Phase	Annotation	Purpose
Setup	@BeforeAll	One-time expensive setup before any tests run (must be static)
Setup	@BeforeEach	Per-test setup before each individual test executes
Execution	@Test	Actual test method containing actions and assertions
Teardown	@AfterEach	Per-test cleanup after each individual test completes
Teardown	@AfterAll	One-time final cleanup after all tests complete (must be static)
Control	@Disabled	Skip test execution without deleting test code

This mapping clarifies when each annotation executes during the test lifecycle, helping testers design proper setup and cleanup strategies for reliable test execution.

Knowledge Check: Test Lifecycle

1

Question 1

In which phase does @BeforeEach execute?

- A) Test Discovery
- B) Setup Phase
- C) Test Execution
- D) Result Reporting

Correct Answer: B) Setup Phase

2

Question 2

What happens immediately when an assertion fails during test execution?

- A) All remaining tests are cancelled
- B) The test method stops and is marked as failed
- C) A warning is logged but test continues
- D) The entire test suite terminates

Correct Answer: B) The test method stops and is marked as failed

Interview Keywords: Test lifecycle, execution order, setup phase, teardown phase, test isolation, lifecycle hooks, test discovery

SECTION 5: INTRODUCTION TO SQL

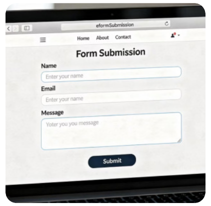
Why Testers Need SQL

SQL (Structured Query Language) enables testers to validate data directly in databases, bypassing the user interface. Whilst UI testing confirms what users see, SQL testing verifies what the system stores, ensuring data integrity, business rule enforcement, and backend processing accuracy.

Many defects exist only at the database level—incorrect calculations, missing records, or data corruption invisible through the UI. SQL empowers testers to perform comprehensive validation, catch subtle defects early, and provide evidence-based defect reports with actual versus expected database states.

SECTION 5: INTRODUCTION TO SQL

The Role of Database Validation in Testing



UI Validation

Confirms what users see on screen. Validates display logic, formatting, and user experience. Cannot verify underlying data storage or business logic implementation.

Database Validation

Confirms what the system stores. Verifies data persistence, referential integrity, business rule enforcement, and calculations. Detects backend defects invisible through UI.

API Validation

Confirms data in transit between systems. Validates API responses, payload structure, and integration contracts. Often paired with database validation for end-to-end verification.

Real-World Database Validation Scenarios

Order Validation

Scenario: User places order for £150.00

UI shows: "Order confirmed"

SQL validates: Order total = £150.00, status = 'CONFIRMED', inventory decremented, payment recorded

User Registration

Scenario: New user registers with email

UI shows: "Account created"

SQL validates: User record exists, email unique, password encrypted, default role assigned, creation timestamp present

Data Deletion

Scenario: User deletes their profile

UI shows: "Account deleted"

SQL validates: User marked inactive (not physically deleted), personal data anonymised, audit trail created, related orphan records handled

Knowledge Check: SQL for Testers

1

Question 1

What is the primary advantage of SQL validation over UI validation?

- A) SQL is faster to write
- B) SQL validates actual stored data, not just display
- C) SQL doesn't require test data
- D) SQL tests never fail

Correct Answer: B) SQL validates actual stored data, not just display

2

Question 2

When should testers use SQL validation?

- A) Only when UI testing fails
- B) Only for performance testing
- C) For backend data verification alongside UI/API testing
- D) Only in production environments

Correct Answer: C) For backend data verification alongside UI/API testing

Interview Keywords: Database validation, backend testing, data integrity, SQL queries, data verification, end-to-end testing

SECTION 6: SQL BASICS

Understanding Databases

A database is an organised collection of structured data stored electronically. Databases use tables (similar to spreadsheets) to store information in rows and columns. Each table represents a specific entity—users, orders, products—and contains records (rows) with attributes (columns).

Tables

Structured containers organising data into rows and columns. Each table stores one type of entity. Example: users table, orders table.

Rows (Records)

Individual entries in a table. Each row represents one instance of the entity. Example: one user, one order.

Columns (Fields)

Attributes or properties of the entity. Each column stores one type of data. Example: user_id, email, first_name.

Primary Key

Unique identifier for each row. Ensures no duplicate records. Example: user_id = 101 identifies one specific user.

Foreign Key

Column referencing primary key in another table. Creates relationships between tables. Example: order.user_id references user.user_id.

Essential SQL Statements for Testers

SELECT

Retrieves data from tables. The most frequently used SQL statement in testing.

```
SELECT * FROM users;
```

WHERE

Filters results based on conditions. Essential for finding specific records.

```
SELECT * FROM users
WHERE email = 'test@example.com';
```

ORDER BY

Sorts results in ascending (ASC) or descending (DESC) order.

```
SELECT * FROM orders
ORDER BY order_date DESC;
```

DISTINCT

Returns unique values only, removing duplicates from results.

```
SELECT DISTINCT country
FROM users;
```

LIMIT

Restricts number of rows returned. Useful for large result sets.

```
SELECT * FROM orders
LIMIT 10;
```

Practical SQL Query Examples

```
-- Find all active users
SELECT user_id, email, first_name, last_name
FROM users
WHERE status = 'active';

-- Find orders above £100 placed in last 30 days
SELECT order_id, user_id, total_amount, order_date
FROM orders
WHERE total_amount > 100.00
 AND order_date >= CURRENT_DATE - INTERVAL '30 days'
ORDER BY order_date DESC;

-- Count unique countries where we have customers
SELECT COUNT(DISTINCT country) as country_count
FROM users
WHERE status = 'active';

-- Find top 5 customers by order value
SELECT user_id, SUM(total_amount) as total_spent
FROM orders
WHERE status = 'completed'
GROUP BY user_id
ORDER BY total_spent DESC
LIMIT 5;

-- Find users who registered today
SELECT user_id, email, created_at
FROM users
WHERE DATE(created_at) = CURRENT_DATE;
```