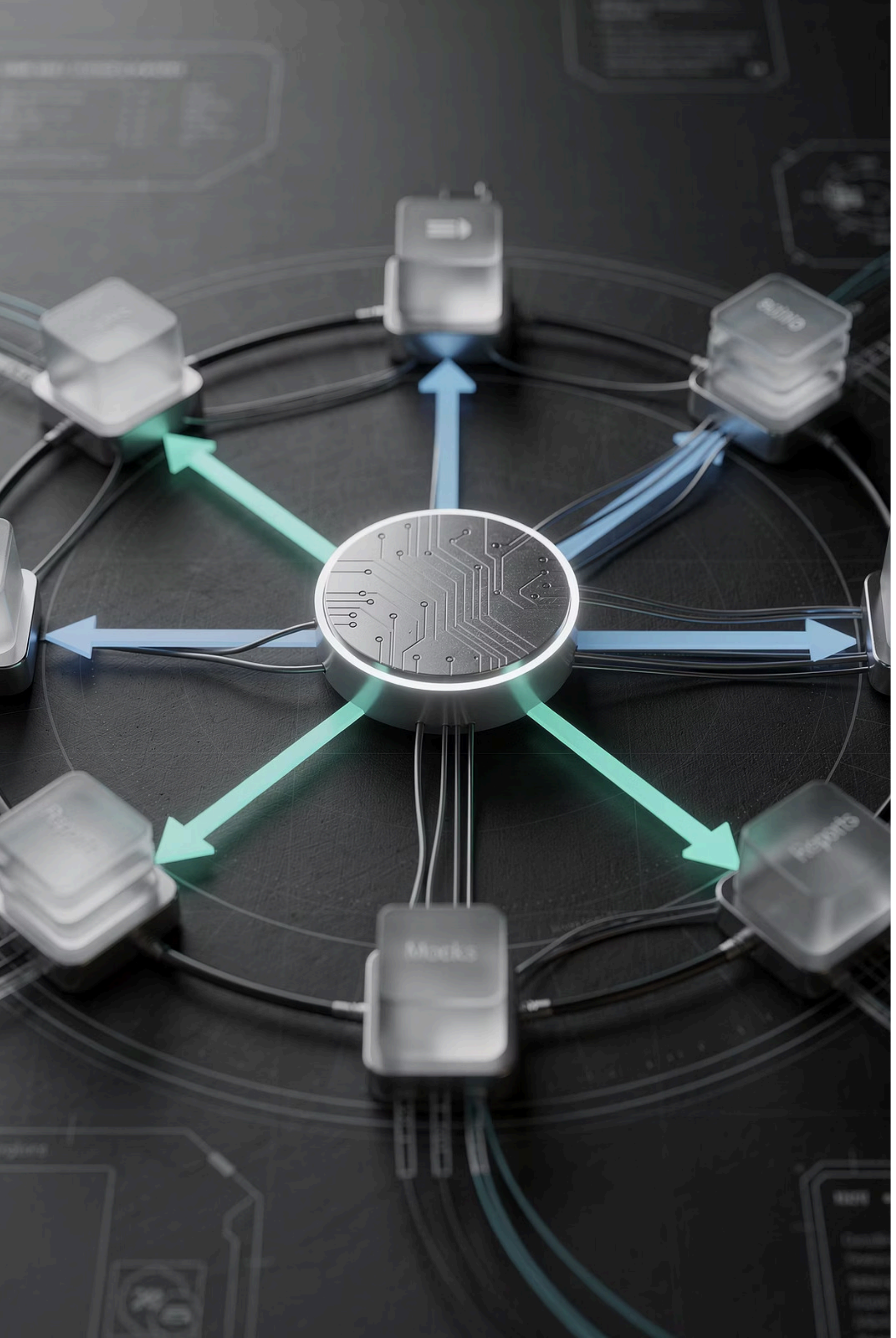
TestNG Framework with Maven – Data-Driven Automation, Framework Design & Reporting

A comprehensive guide to building scalable, maintainable automation frameworks using TestNG and Maven for professional QA engineers and automation testers.



CHAPTER 1

Introduction to TestNG

TestNG (Test Next Generation) is a powerful testing framework designed to cover a broader range of test categories including unit, functional, end-to-end, and integration testing. It provides advanced capabilities that make test automation more efficient and maintainable in enterprise environments.

What is TestNG?

TestNG is an open-source automated testing framework inspired by JUnit but introduces more powerful and flexible features. It provides annotations, grouping, parameterisation, and parallel execution capabilities that are essential for modern automation frameworks.

TestNG integrates seamlessly with build tools like Maven and Ant, making it the preferred choice for continuous integration pipelines. It generates detailed HTML reports automatically, providing clear visibility into test execution results.

Key Characteristics

- Annotation-driven test configuration
- Flexible test execution control
- Built-in reporting mechanisms
- Data-driven testing support
- Parallel execution capabilities

Why TestNG is Used in Automation Frameworks

Annotation Control

Provides fine-grained control over test execution order and dependencies using annotations like @BeforeMethod, @AfterMethod, @BeforeClass, and more.

Data-Driven Testing

Supports @DataProvider for supplying multiple data sets to test methods, enabling comprehensive test coverage with minimal code duplication.

Parallel Execution

Executes tests in parallel across multiple threads, classes, or test suites, significantly reducing overall execution time for large test suites.

Flexible Configuration

XML-based configuration files (testng.xml) allow grouping tests, setting parameters, and defining suite-level settings without modifying test code.

Where TestNG Fits in Selenium-Based Automation

In Selenium-based automation, TestNG serves as the test orchestration layer. Selenium WebDriver handles browser interactions, whilst TestNG manages test execution flow, assertions, and reporting. Maven provides dependency management and build automation, creating a complete automation ecosystem.

This combination—Selenium for browser automation, TestNG for test management, and Maven for project structure—forms the industry-standard approach for web application testing frameworks used by most organisations worldwide.

Maven
TestNG
Selenium WebDriver

TestNG vs JUnit: A Comparison

Feature	TestNG	JUnit
Annotations	More comprehensive (@BeforeSuite, @BeforeTest, @BeforeClass, @BeforeMethod)	Limited options (@BeforeClass, @Before)
Data-Driven Testing	Native @DataProvider support	Requires external libraries
Parallel Execution	Built-in parallel execution at suite, test, class, and method levels	Limited parallel support
Test Dependencies	Tests can depend on other tests using dependsOnMethods	No native support
Grouping	Tests can be grouped and executed selectively	No native grouping
Reports	Generates detailed HTML reports automatically	Basic console output

Real-World Automation Use Cases

E-Commerce Testing

Automating end-to-end user journeys including product search, cart operations, checkout processes, and payment validation across multiple browsers and environments.

Banking Applications

Testing critical transaction flows, account operations, and security features with data-driven approaches to cover multiple user scenarios and edge cases.

Healthcare Systems

Validating patient data management, appointment scheduling, prescription handling, and compliance requirements across integrated healthcare platforms.

Knowledge Check: Introduction to TestNG

1

Which feature distinguishes TestNG from JUnit?

A) Support for Selenium WebDriver

B) Native @DataProvider annotation

C) Ability to write test methods

D) Integration with Eclipse IDE

Correct Answer: B) Native @DataProvider annotation enables data-driven testing without external libraries.

2

What is the primary purpose of TestNG in automation?

A) Browser automation

B) Test execution management and reporting

C) Database connectivity

D) API request handling

Correct Answer: B) TestNG manages test execution flow, provides annotations for setup/teardown, and generates execution reports.

Interview Keywords: Framework Essentials

Framework

A structured approach to automation with reusable components, design patterns, and standardised practices.

Annotations

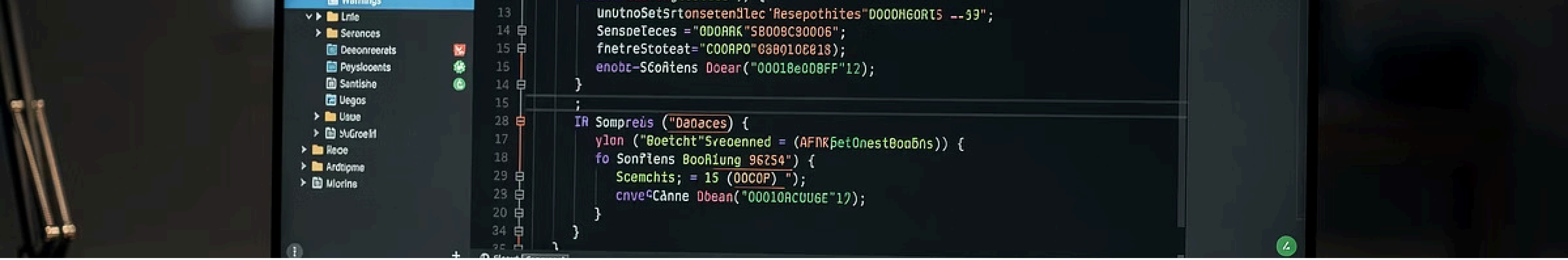
Metadata markers that control test execution flow and configuration without modifying core test logic.

Lifecycle

The sequence of setup, execution, and teardown phases that TestNG manages through annotations.

Data-Driven

Testing approach where test logic remains constant whilst data inputs vary, maximising test coverage.



```
        UnUtnoSetSrtonsetenIlec'Resepothites"DOODMGORIS --39";
        SensoeTeces ="ODOARK"SB009C90006";
        fnetreStoteat="COOAPO"0880108818);
        enobt-SCoñtens Doear("0001BeOD8FF"12);
    }
    ;
    IR Sompreus ("Daoaces) {
        ylon ("Boetcht"Sreoenned = (AFNKpetOnestBoaбns)) {
        fo Sonflens BooAiung 96254") {
            Scemchts; = 15 (OOCOP) ");
            cnveCânne Dbean("OOO10ACUUGE"17);
        }
    }
```

CHAPTER 2

TestNG Annotations in Detail

TestNG annotations are special markers that provide metadata about test methods and control the test execution lifecycle. Understanding annotation hierarchy and execution order is fundamental to designing robust automation frameworks.

What are Annotations?

Annotations in TestNG are Java metadata elements prefixed with the @ symbol that provide instructions to the TestNG framework about how and when to execute specific methods. They eliminate the need for complex configuration code and make test structure more readable.

Annotations define the test execution lifecycle, from initialisation through to cleanup. They establish a clear contract between the test framework and your test code, ensuring consistent behaviour across test suites regardless of execution order or environment.

Common Annotation Uses

- Initialise test data and resources
- Configure browser instances
- Mark methods as test cases
- Clean up resources after execution
- Control execution dependencies

Why Annotations are Important in Controlling Test Execution

Execution Order Control

Annotations define a hierarchical execution sequence from suite-level setup through to method-level operations, ensuring proper initialisation and cleanup at each level.

Resource Management

Setup and teardown annotations guarantee that resources like database connections, browser instances, and test data are properly initialised before tests and cleaned up afterwards.

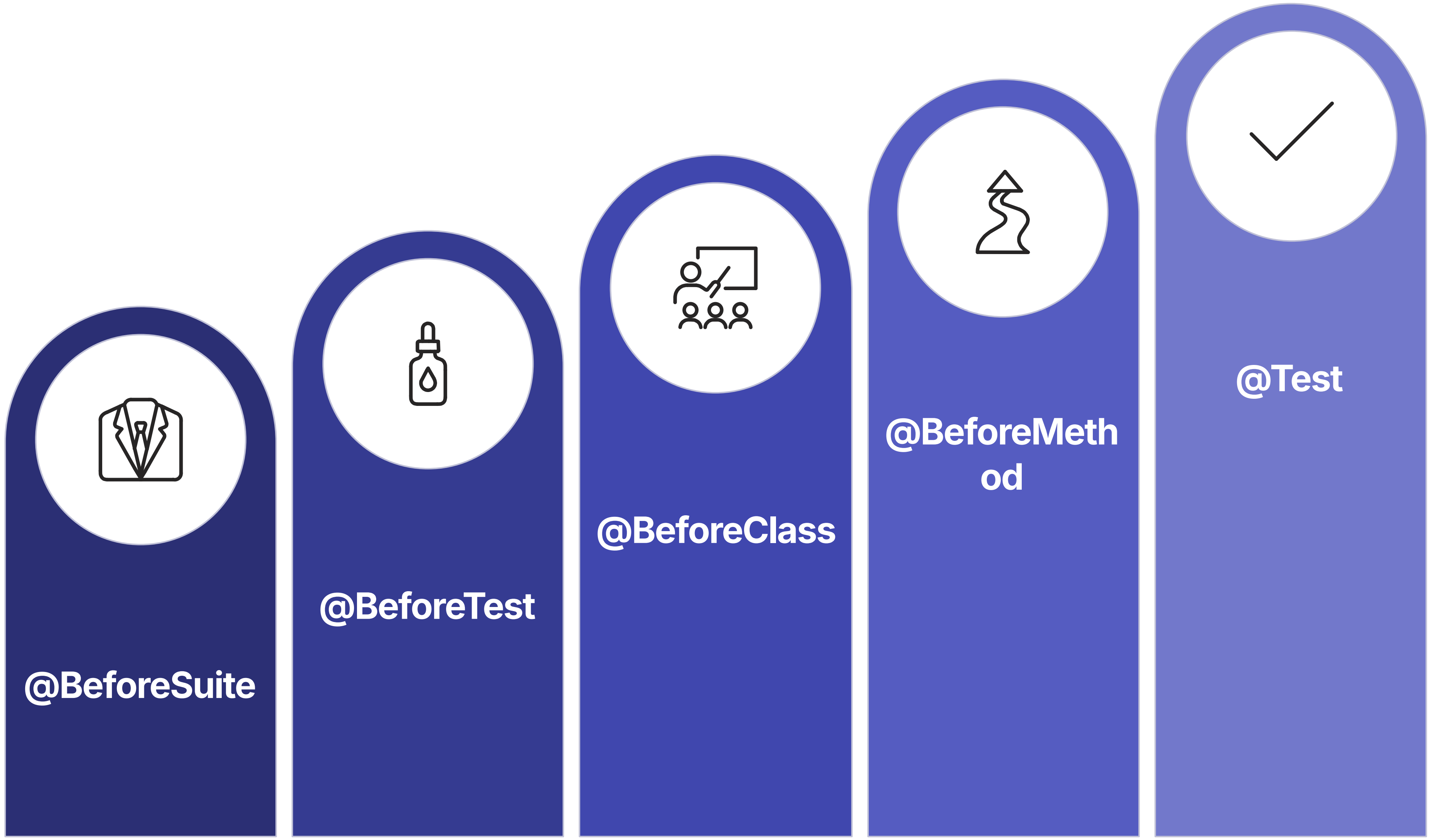
Code Reusability

Common setup and teardown logic can be centralised in annotated methods, eliminating duplication across test classes and ensuring consistent test environment configuration.

Maintenance Efficiency

Clear annotation structure makes tests easier to understand and maintain, as setup, execution, and cleanup phases are explicitly defined and separated.

TestNG Annotation Execution Order



TestNG executes annotations in a specific hierarchy from suite level down to method level for setup, then reverses the order for teardown. Understanding this sequence is critical for proper resource management and avoiding test interference.

Core TestNG Annotations

Annotation	Execution Timing	Common Usage
@BeforeSuite	Once before all tests in suite	Database connection, global configuration
@AfterSuite	Once after all tests in suite	Close database, cleanup global resources
@BeforeTest	Before each <test> tag in testng.xml	Test-specific configuration, logging setup
@AfterTest	After each <test> tag completes	Test-level cleanup, report generation
@BeforeClass	Once before any method in class	Browser initialisation, page object creation
@AfterClass	Once after all methods in class	Close browser, save screenshots
@BeforeMethod	Before each @Test method	Navigate to URL, clear cache, login
@AfterMethod	After each @Test method	Logout, capture failure screenshots
@Test	Marks method as test case	Actual test logic and assertions

@Test Annotation

The @Test annotation identifies a method as a test case. TestNG will execute all @Test annotated methods during test execution. This annotation supports various attributes for controlling test behaviour.

```
@Test(priority = 1)
public void verifyLoginPageTitle() {
    String actualTitle = driver.getTitle();
    String expectedTitle = "Login Page";
    Assert.assertEquals(actualTitle, expectedTitle);
}

@Test(priority = 2, dependsOnMethods = "verifyLoginPageTitle")
public void verifyLoginFunctionality() {
    loginPage.enterUsername("testuser");
    loginPage.enterPassword("password123");
    loginPage.clickLoginButton();
    Assert.assertTrue(homePage.isDisplayed());
}
```

Key Attributes

- **priority:** Execution order (lower runs first)
- **enabled:** Enable/disable test (true/false)
- **dependsOnMethods:** Execute only if dependencies pass
- **description:** Test case description for reports
- **groups:** Categorise tests for selective execution

@BeforeMethod and @AfterMethod

These annotations execute before and after each @Test method, making them ideal for operations that need to run for every test case, such as navigating to a starting URL or capturing screenshots after test completion.

```
public class LoginTests {
 WebDriver driver;

 @BeforeMethod
 public void setupTest() {
 // Runs before each test method
 driver.get("https://example.com/login");
 driver.manage().timeouts().implicitlyWait(Duration.ofSeconds(10));
 System.out.println("Navigated to login page");
 }

 @Test
 public void testValidLogin() {
 // Test logic here
 }

 @Test
 public void testInvalidLogin() {
 // Test logic here
 }

 @AfterMethod
 public void teardownTest(ITestResult result) {
 // Runs after each test method
 if (result.getStatus() == ITestResult.FAILURE) {
 // Capture screenshot for failed tests
 captureScreenshot(result.getName());
 }
 driver.manage().deleteAllCookies();
 }
}
```

@BeforeClass and @AfterClass

These annotations execute once per class, before any test methods run and after all test methods complete. They're suitable for expensive operations that can be shared across multiple tests in the same class, such as browser initialisation.

```
public class CheckoutTests {
 WebDriver driver;
 HomePage homePage;

 @BeforeClass
 public void setupClass() {
 // Runs once before any test in this class
 System.setProperty("webdriver.chrome.driver", "path/to/chromedriver");
 driver = new ChromeDriver();
 driver.manage().window().maximize();
 homePage = new HomePage(driver);
 System.out.println("Browser initialised for CheckoutTests");
 }

 @Test
 public void testAddToCart() {
 // Test logic using the same browser instance
 }

 @Test
 public void testCheckoutProcess() {
 // Test logic using the same browser instance
 }

 @AfterClass
 public void teardownClass() {
 // Runs once after all tests in this class
 if (driver != null) {
 driver.quit();
 System.out.println("Browser closed for CheckoutTests");
 }
 }
}
```

@BeforeTest / @AfterTest and @BeforeSuite / @AfterSuite

@BeforeTest / @AfterTest

Execute before and after each <test> tag in testng.xml. A testng.xml file can contain multiple <test> tags, each representing a logical grouping of test classes.

```
@BeforeTest
public void setupTestLevel() {
    // Setup for this test tag
    configureLogger();
    loadTestData();
}

@AfterTest
public void teardownTestLevel() {
    // Cleanup for this test tag
    generateTestReport();
    clearTestData();
}
```

@BeforeSuite / @AfterSuite

Execute once before and after the entire test suite. Ideal for one-time setup operations like database connections or global configuration that applies to all tests.

```
@BeforeSuite
public void setupSuite() {
 // Runs once for entire suite
 connectToDatabase();
 loadGlobalConfig();
}

@AfterSuite
public void teardownSuite() {
 // Runs once after suite
 disconnectFromDatabase();
 emailTestReport();
}
```

Real Project Usage Scenarios

@BeforeMethod for Authentication

Use @BeforeMethod to log in before each test, ensuring tests start from an authenticated state. This approach isolates tests and prevents failures from affecting subsequent tests.

@BeforeClass for Browser Setup

Initialise the browser once per test class to reduce execution time whilst maintaining isolation between different test classes and their purposes.

@AfterMethod for Screenshot Capture

Capture screenshots after each test method, particularly for failures, providing visual evidence for debugging and test result analysis.

@BeforeSuite for Database Connection

Establish database connections once at the suite level when multiple test classes require database access, improving efficiency and resource management.

Knowledge Check: TestNG Annotations

1

Which annotation executes once before any method in a test class?

A) @BeforeMethod
B) @BeforeTest
C) @BeforeClass
D) @BeforeSuite

Correct Answer: C) @BeforeClass runs once per class, before all test methods execute.

2

When should you use @BeforeMethod instead of @BeforeClass?

A) When you need to initialise the browser once
B) When each test needs a fresh state
C) When connecting to a database
D) When generating test reports

Correct Answer: B) @BeforeMethod runs before every @Test method, ensuring each test starts with a clean state.

3

What is the execution order from highest to lowest level?

A) Method → Class → Test → Suite
B) Suite → Test → Class → Method
C) Class → Suite → Method → Test
D) Test → Suite → Class → Method

Correct Answer: B) TestNG executes from Suite level down to Method level, then reverses for teardown annotations.

Interview Trap: @BeforeMethod vs @BeforeTest

Common Interview Question

"What's the difference between @BeforeMethod and @BeforeTest?"

Many candidates confuse these annotations because their names sound similar. However, they operate at completely different levels of the test execution hierarchy.

The Key Distinction

- **@BeforeMethod:** Runs before each @Test method within classes
- **@BeforeTest:** Runs before all classes within a <test> tag in testng.xml
- **Scope:** Method is narrower, Test is broader
- **Use @BeforeMethod for:** Per-test setup like navigation
- **Use @BeforeTest for:** Test-level configuration like logging

Remember: @BeforeTest is linked to the testng.xml structure, whilst @BeforeMethod is linked to individual test methods regardless of XML configuration.

CHAPTER 3

Data-Driven Testing in TestNG

Data-driven testing is a methodology where test logic remains constant whilst test data varies, enabling comprehensive test coverage with minimal code. This approach is essential for validating applications against multiple input scenarios efficiently.

What is Data-Driven Testing?

Data-driven testing separates test logic from test data, allowing the same test script to execute multiple times with different data sets. Instead of creating separate test methods for each data combination, you write one test method that accepts parameters.

This approach dramatically improves test coverage whilst reducing code duplication and maintenance effort. When data requirements change, you modify the data source rather than the test code itself.

Benefits

- Execute one test with multiple data sets
- Reduce code duplication significantly
- Easier to add new test scenarios
- Separate test logic from test data
- Non-technical users can manage data
- Improved test coverage and efficiency

Why Data-Driven Testing is Critical in Automation

Maximise Coverage

Test applications against edge cases, boundary values, and various data combinations without writing repetitive test code.

Reduce Maintenance

When application behaviour changes, update one test method instead of multiple similar tests, significantly reducing maintenance time.

Enable Collaboration

Business analysts and QA engineers can contribute test data in Excel or CSV format without modifying test code.

Improve Quality

Systematically validate application behaviour across positive scenarios, negative scenarios, and edge cases with organised data sets.

@DataProvider Annotation

The @DataProvider annotation marks a method that returns test data in a two-dimensional Object array. TestNG automatically feeds this data to test methods, executing the test once for each data set provided.

```
@DataProvider(name = "loginData")
public Object[][] getLoginData() {
 return new Object[][] {
 { "user1@example.com", "password123", true },
 { "user2@example.com", "password456", true },
 { "invalid@example.com", "wrongpass", false },
 { "", "password123", false },
 { "user3@example.com", "", false }
 };
}

@Test(dataProvider = "loginData")
public void testLogin(String username, String password, boolean shouldSucceed) {
 loginPage.enterUsername(username);
 loginPage.enterPassword(password);
 loginPage.clickLoginButton();

 if (shouldSucceed) {
 Assert.assertTrue(homePage.isDisplayed(),
 "Login should succeed for: " + username);
 } else {
 Assert.assertTrue(loginPage.isErrorDisplayed(),
 "Login should fail for: " + username);
 }
}
```

Parameterisation Using testng.xml

TestNG supports passing parameters from testng.xml to test methods using the @Parameters annotation. This approach is suitable for environment-specific values like URLs, browser types, or configuration settings that remain constant during a test run.

```xml
<suite name="Test Suite">
  <test name="Chrome Tests">
    <parameter name="browser"
           value="chrome"/>
    <parameter name="url"
           value="https://example.com"/>
    <classes>
      <class name="LoginTests"/>
    </classes>
  </test>
</suite>
```

```java
// Test class
@Parameters({"browser", "url"})
@BeforeClass
public void setup(String browser,
 String url) {
 if (browser.equals("chrome")) {
 driver = new ChromeDriver();
 } else if (browser.equals("firefox")) {
 driver = new FirefoxDriver();
 }
 driver.get(url);
}

@Test
public void testHomePage() {
 // Test logic
}
```

External Data Sources: Excel and CSV

For large data sets or when non-technical team members need to maintain test data, external files like Excel or CSV provide a practical solution. Libraries like Apache POI enable reading Excel files, whilst Java's built-in utilities handle CSV files.

Excel Data Approach

- Use Apache POI library to read .xlsx files
- Store test data in structured sheets
- Suitable for complex data structures
- Business analysts can manage data
- Supports formulas and formatting

Use Case: Customer profile data with multiple fields, calculation requirements, or when data needs to be reviewed in a familiar format.

CSV Data Approach

- Lightweight comma-separated values
- Easy to generate from databases
- Simple parsing with Java utilities
- Version control friendly (text format)
- Fast read performance

Use Case: Simple tabular data, database exports, or when data needs to be version-controlled alongside test code.

Real-World Example: Login with Multiple Credentials

This example demonstrates testing login functionality with various credential combinations, including valid users, invalid users, empty fields, and special characters—all using a single test method.

```
@DataProvider(name = "loginCredentials")
public Object[][] getLoginCredentials() {
 return new Object[][] {
 // username, password, expectedResult, description
 { "admin@company.com", "Admin@123", "success", "Valid admin login" },
 { "user@company.com", "User@456", "success", "Valid user login" },
 { "test@company.com", "wrongpassword", "failure", "Invalid password" },
 { "notregistered@company.com", "Pass@123", "failure", "Unregistered email" },
 { "", "Pass@123", "failure", "Empty username" },
 { "user@company.com", "", "failure", "Empty password" },
 { "user@company.com", "pass", "failure", "Weak password" },
 { "invalid-email", "Pass@123", "failure", "Invalid email format" },
 { "user@company.com' OR '1'='1", "Pass@123", "failure", "SQL injection attempt" }
 };
}

@Test(dataProvider = "loginCredentials")
public void testLoginScenarios(String username, String password,
 String expectedResult, String description) {
 System.out.println("Testing: " + description);

 loginPage.enterUsername(username);
 loginPage.enterPassword(password);
 loginPage.clickLoginButton();

 if (expectedResult.equals("success")) {
 Assert.assertTrue(homePage.isDisplayed(),
 "Login should succeed: " + description);
 } else {
 Assert.assertTrue(loginPage.getErrorMessage().contains("Invalid"),
 "Login should fail: " + description);
 }
}
```

Benefits and Limitations of Data-Driven Testing

Benefits

- Significantly reduced code duplication
- Improved test coverage with minimal effort
- Easy to add new test scenarios
- Clear separation of concerns
- Business users can contribute test data
- Faster test creation for similar scenarios
- Easier regression testing

Limitations

- Initial setup requires more planning
- Debugging failures can be more complex
- Large data sets increase execution time
- External files need version control
- Not suitable for all test scenarios
- Requires proper data management strategy
- Failed data sets may block subsequent tests

Knowledge Check: Data-Driven Testing

1

What does a @DataProvider method return?

A) A List of test data

B) A two-dimensional Object array

C) A HashMap of parameters

D) A JSON string

Correct Answer: B) @DataProvider returns Object[][] where each row represents one set of test data.

2

When should you use testng.xml parameters instead of @DataProvider?

A) For multiple data sets per test

B) For environment-specific configuration

C) For user credential combinations

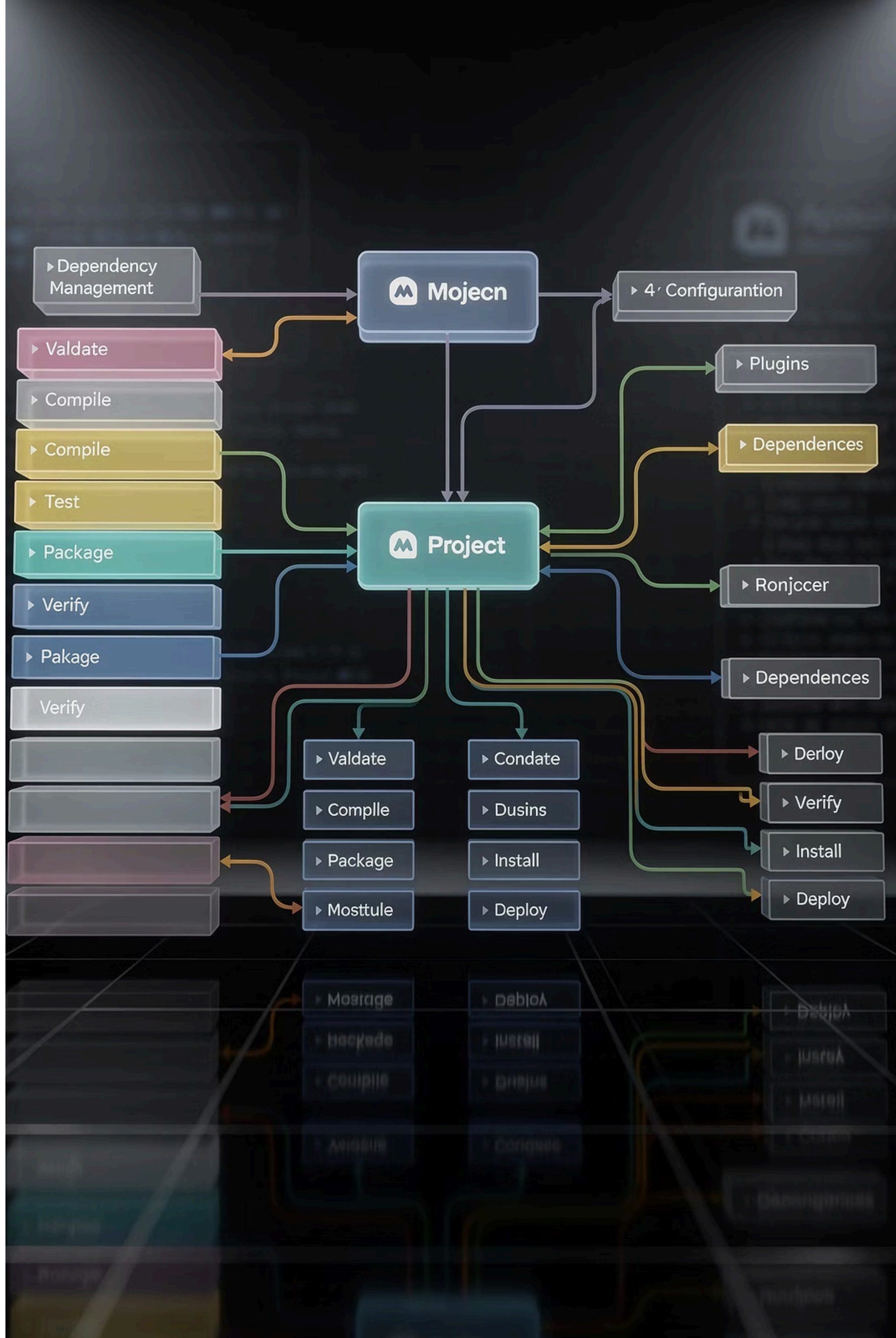
D) For boundary value testing

Correct Answer: B) XML parameters suit configuration values like browser type or environment URLs, not varying test data.

CHAPTER 4

Maven Project Basics

Apache Maven is a build automation and project management tool that standardises project structure, manages dependencies, and automates the build lifecycle. It's the industry standard for Java-based automation frameworks.



What is Maven?

Maven is a powerful project management tool that provides a uniform build system, dependency management, and project structure standardisation. It uses an XML file (pom.xml) to describe the project, its dependencies, build configuration, and plugins.

Maven follows a "convention over configuration" approach, meaning it has default behaviours that work for most projects without extensive configuration. This standardisation makes it easier for developers to understand and navigate any Maven project.

Core Maven Concepts

- POM (Project Object Model) file
- Dependency management
- Standard directory structure
- Build lifecycle phases
- Plugin architecture
- Repository management

Why Maven is Used in Automation Projects

Dependency Management

Automatically downloads and manages Selenium, TestNG, and other library versions, including transitive dependencies, from Maven Central Repository.

Standardised Structure

Enforces consistent project organisation that any developer can understand immediately, improving team collaboration and code maintainability.

Build Automation

Compiles code, runs tests, and generates reports with simple commands, integrating seamlessly with CI/CD pipelines like Jenkins and GitLab.

Plugin Ecosystem

Extends functionality through plugins for Surefire (test execution), compiler settings, reporting, code coverage, and integration with other tools.

Understanding pom.xml

The Project Object Model (pom.xml) is the fundamental unit of work in Maven. It contains project information, configuration details, dependencies, and build specifications. Every Maven project must have a pom.xml file in its root directory.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0">
 <modelVersion>4.0.0</modelVersion>

 <!-- Project Identification -->
 <groupId>com.company.automation</groupId>
 <artifactId>selenium-testng-framework</artifactId>
 <version>1.0.0</version>
 <packaging>jar</packaging>

 <name>Selenium TestNG Automation Framework</name>

 <properties>
 <maven.compiler.source>11</maven.compiler.source>
 <maven.compiler.target>11</maven.compiler.target>
 <selenium.version>4.15.0</selenium.version>
 <testng.version>7.8.0</testng.version>
 </properties>
</project>
```

Maven Dependencies

Dependencies are external libraries your project needs. Maven downloads them automatically from Maven Central Repository along with their transitive dependencies (libraries that your dependencies need).

```xml
<dependencies>
    <!-- Selenium WebDriver -->
    <dependency>
        <groupId>org.seleniumhq.selenium</groupId>
        <artifactId>selenium-java</artifactId>
        <version>${selenium.version}</version>
    </dependency>

    <!-- TestNG Framework -->
    <dependency>
        <groupId>org.testng</groupId>
        <artifactId>testng</artifactId>
        <version>${testng.version}</version>
        <scope>test</scope>
    </dependency>

    <!-- WebDriverManager (manages browser drivers) -->
    <dependency>
        <groupId>io.github.bonigarcia</groupId>
        <artifactId>webdrivermanager</artifactId>
        <version>5.6.2</version>
    </dependency>
</dependencies>
```

Scope attribute: Defines when the dependency is available. Common scopes include `compile` (default, available everywhere), `test` (only during testing), and `provided` (supplied by runtime environment).