TestNG Framework — Complete Course

Contents

TestNG Framework — Complete Course

A step-by-step guide to every feature, with interview cross-questions

---

Table of Contents

1. Introduction to TestNG

---

1. Introduction to TestNG

TestNG ("Test Next Generation") is a Java testing framework inspired by JUnit and NUnit, built by Cédric Beust to fix gaps in JUnit 3/4: no built-in support for parallel execution, weak dependency management, no native data-driven testing, and rigid grouping.

Why teams use it:

- Annotation-based, readable test structure
- Native data-driven testing via `@DataProvider`
- Test grouping and dependency control
- Parallel execution at method/class/suite level
- Flexible configuration via XML (`testng.xml`)
- Built-in HTML/XML reporting
- Widely used with Selenium WebDriver and REST Assured for automation frameworks

TestNG vs JUnit (core difference reasoning):

Feature	TestNG	JUnit 5
Data-driven tests	`@DataProvider` (native)	`@ParameterizedTest` (needs extra annotations)

Feature	TestNG	JUnit 5
Test dependency	`dependsOnMethods`, `dependsOnGroups`	Not natively supported
Grouping	`groups` attribute	Tags (`@Tag`)
Parallel execution	Built into testng.xml	Needs config + JUnit Platform properties
XML suite config	Yes (`testng.xml`)	No equivalent

This matters because interviewers frequently ask "why TestNG over JUnit" — the honest answer is: dependency management and suite-level XML configuration, not raw speed or simplicity.

---

2. Setting Up TestNG

Maven (pom.xml):

```xml
<dependency>
    <groupId>org.testng</groupId>
    <artifactId>testng</artifactId>
    <version>7.10.2</version>
    <scope>test</scope>
</dependency>
```

Gradle (build.gradle):

```
testImplementation 'org.testng:testng:7.10.2'
test {
    useTestNG()
}
```

IDE: IntelliJ IDEA and Eclipse both bundle TestNG plugin support (IntelliJ: built-in; Eclipse: install "TestNG for Eclipse" from Marketplace). This gives you right-click → Run as TestNG Test.

---

3. Your First TestNG Test

```java
import org.testng.annotations.Test;

public class FirstTest {

    @Test
    public void verifyAddition() {
        int sum = 2 + 3;
        assert sum == 5;
```

```
    }
}
```

Run it: right-click → Run, or via Maven: `mvn test`. TestNG auto-discovers any method annotated `@Test` — no need to extend a base class or follow naming conventions like `testXxx()` (a JUnit 3 legacy requirement TestNG deliberately drops).

---

4. Annotations — The Building Blocks

Annotation	Runs
`@BeforeSuite`	Once, before all tests in the suite
`@BeforeTest`	Once per `<test>` tag in testng.xml
`@BeforeClass`	Once per class, before its first `@Test` method
`@BeforeMethod`	Before every `@Test` method
`@Test`	The actual test
`@AfterMethod`	After every `@Test` method
`@AfterClass`	Once per class, after its last `@Test` method
`@AfterTest`	Once per `<test>` tag
`@AfterSuite`	Once, after all tests in the suite

```java
import org.testng.annotations.*;

public class AnnotationDemo {

    @BeforeSuite
    public void beforeSuite() { System.out.println("Suite starting"); }

    @BeforeClass
    public void beforeClass() { System.out.println("Class setup"); }

    @BeforeMethod
    public void beforeMethod() { System.out.println("Before each test"); }

    @Test
    public void testOne() { System.out.println("Test One"); }

    @Test
    public void testTwo() { System.out.println("Test Two"); }

    @AfterMethod
    public void afterMethod() { System.out.println("After each test"); }

    @AfterClass
    public void afterClass() { System.out.println("Class teardown"); }
```

```
    @AfterSuite
    public void afterSuite() { System.out.println("Suite finished"); }
}
```

Mental model: think of it as nested brackets — Suite wraps Test wraps Class wraps Method. Anything "Before" fires outside-in, anything "After" fires inside-out.

---

5. Annotation Execution Order

For the class above, actual console output:

```
Suite starting
Class setup
Before each test
Test One
After each test
Before each test
Test Two
After each test
Class teardown
Suite finished
```

Key rule interviewers probe: `@BeforeClass`/`@AfterClass` run once regardless of how many `@Test` methods exist. `@BeforeMethod`/`@AfterMethod` run once per test method. If you have 2 test methods, `@BeforeMethod` fires twice, `@BeforeClass` fires once.

Order of multiple `@Test` methods with no priority/dependency set: TestNG does NOT guarantee alphabetical or declaration order by default — it depends on the JVM's method reflection order, which is why relying on implicit order is a bug source. Use `priority` or `dependsOnMethods` to force order.

---

6. testng.xml — The Control Center

`testng.xml` lets you run tests without touching code — include/exclude classes, methods, groups, set parallelism, inject parameters.

```
<!DOCTYPE suite SYSTEM "https://testng.org/testng-1.0.dtd">
<suite name="RegressionSuite" verbose="1">
    <test name="SmokeTests">
        <classes>
            <class name="com.demo.LoginTest"/>
            <class name="com.demo.CheckoutTest"/>
        </classes>
    </test>
</suite>
```

Run via Maven Surefire by pointing to it:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <configuration>
        <suiteXmlFiles>
            <suiteXmlFile>testng.xml</suiteXmlFile>
        </suiteXmlFiles>
    </configuration>
</plugin>
```

---

7. Suites, Tests, Classes, Packages

Hierarchy: **Suite → Test → Class → Method**. One `testng.xml` = one suite. A suite can have multiple `<test>` blocks, each with its own set of classes — useful for splitting "Smoke" vs "Regression" runs in a single file.

```
<suite name="MasterSuite">
    <test name="Smoke">
        <classes>
            <class name="com.demo.LoginTest"/>
        </classes>
    </test>
    <test name="Regression">
        <packages>
            <package name="com.demo.regression.*"/>
        </packages>
    </test>
</suite>
```

`<packages>` lets you include every test class in a package without listing them individually.

---

8. Parameters (testng.xml driven)

Pass values from XML into test code — useful for environment URLs, browser names, credentials.

```
<suite name="ParamSuite">
    <test name="ParamTest">
        <parameter name="browser" value="chrome"/>
        <classes>
            <class name="com.demo.BrowserTest"/>
        </classes>
```

```
    </test>
</suite>
```

```
import org.testng.annotations.Parameters;
import org.testng.annotations.Test;

public class BrowserTest {
    @Test
    @Parameters("browser")
    public void launchBrowser(String browser) {
        System.out.println("Launching: " + browser);
    }
}
```

Limitation to know for interviews: XML parameters are static strings only — for dynamic or multi-row data, you need `@DataProvider`.

---

9. DataProviders (data-driven testing)

The core mechanism for running the same test with multiple data sets.

```
import org.testng.annotations.DataProvider;
import org.testng.annotations.Test;

public class LoginDataTest {

    @DataProvider(name = "loginData")
    public Object[][] getData() {
        return new Object[][] {
            { "user1", "pass1" },
            { "user2", "pass2" },
            { "admin", "adminPass" }
        };
    }

    @Test(dataProvider = "loginData")
    public void login(String username, String password) {
        System.out.println("Logging in with " + username + "/" + password);
    }
}
```

This runs `login()` three times, once per row. `@DataProvider` can also live in a separate class:

```
@Test(dataProvider = "loginData", dataProviderClass = ExternalDataProviders.class)
```

For real-world frameworks, DataProviders are typically wired to Excel/CSV/JSON readers so test data lives outside code.

Parallel data providers: `@DataProvider(name = "data", parallel = true)` lets each data row run on a separate thread.

---

10. Groups

Groups let you tag tests logically (`smoke`, `regression`, `sanity`) and run subsets without code changes.

```
public class GroupTest {

    @Test(groups = "smoke")
    public void loginTest() { System.out.println("Login smoke test"); }

    @Test(groups = "regression")
    public void fullCheckoutTest() { System.out.println("Full checkout"); }

    @Test(groups = { "smoke", "regression" })
    public void criticalPathTest() { System.out.println("Critical path"); }
}
```

```
<suite name="GroupSuite">
    <test name="SmokeOnly">
        <groups>
            <run>
                <include name="smoke"/>
                <exclude name="regression"/>
            </run>
        </groups>
        <classes>
            <class name="com.demo.GroupTest"/>
        </classes>
    </test>
</suite>
```

MetaGroups: you can nest groups inside groups using `<define>` in XML, grouping "smoke" and "sanity" under a "checkin" meta-group.

---

11. Dependencies Between Tests

Force execution order and skip dependents when a prerequisite fails.

```
public class DependencyTest {

    @Test
    public void login() {
        System.out.println("Logging in");
```

```
    }

    @Test(dependsOnMethods = "login")
    public void addToCart() {
        System.out.println("Adding to cart");
    }

    @Test(dependsOnMethods = "addToCart")
    public void checkout() {
        System.out.println("Checking out");
    }
}
```

Critical behavior: if `login()` fails, `addToCart()` and `checkout()` are **skipped** (not failed) — reported as `SKIP` in results. This is a favorite interview trap: "does TestNG mark dependent tests as failed or skipped?" — answer: skipped.

`dependsOnGroups` works the same way but at group granularity:

```
@Test(dependsOnGroups = "login.*")
public void postLoginTest() { }
```

`alwaysRun = true` forces a dependent test to run even if its dependency failed:

```
@Test(dependsOnMethods = "login", alwaysRun = true)
public void logCleanup() { }
```

---

12. Priority

Controls execution order among independent `@Test` methods within a class. Lower number runs first; default priority is 0.

```
public class PriorityTest {

    @Test(priority = 2)
    public void thirdStep() { System.out.println("Runs third"); }

    @Test(priority = 0)
    public void firstStep() { System.out.println("Runs first"); }

    @Test(priority = 1)
    public void secondStep() { System.out.println("Runs second"); }
}
```

Priority vs dependsOnMethods: priority only controls order — it does NOT skip a test if an earlier-priority test fails. `dependsOnMethods` controls both order and skip-on-failure. Interviewers ask this distinction often.

---

13. Parallel Execution

Set in `testng.xml` via the `parallel` attribute: `methods`, `tests`, `classes`, or `instances`.

```
<suite name="ParallelSuite" parallel="methods" thread-count="4">
    <test name="ParallelTest">
        <classes>
            <class name="com.demo.ParallelDemo"/>
        </classes>
    </test>
</suite>
```

- `parallel="methods"` — every `@Test` method runs on its own thread
- `parallel="classes"` — each class runs on its own thread, methods within a class run sequentially
- `parallel="tests"` — each `<test>` tag in XML runs on its own thread
- `thread-count` — max threads in the pool

Thread safety warning (common real-world gotcha): if your test class shares a single WebDriver instance as an instance field and you run `parallel="methods"`, tests will collide on the same browser session. Fix: use `ThreadLocal<WebDriver>` so each thread gets its own driver instance.

---

14. Assertions (Hard vs Soft)

Hard assertions (`org.testng.Assert`) stop test execution immediately on first failure:

```
import org.testng.Assert;

@Test
public void hardAssertExample() {
    Assert.assertEquals(2 + 2, 4);
    Assert.assertTrue(5 > 3);
    Assert.assertEquals(2 + 2, 5); // test stops here if this fails
    Assert.assertTrue(10 > 20);    // never reached
}
```

Soft assertions (`org.testng.asserts.SoftAssert`) collect all failures and report them together — you must call `assertAll()` at the end:

```
import org.testng.asserts.SoftAssert;

@Test
public void softAssertExample() {
    SoftAssert softAssert = new SoftAssert();
    softAssert.assertEquals(2 + 2, 5);   // recorded, execution continues
    softAssert.assertTrue(10 > 20);      // recorded, execution continues
```

```
    softAssert.assertAll();              // now the test fails, reporting both
}
```

When to use which: hard assertions for critical checks where continuing is meaningless (e.g., "did the page load at all"). Soft assertions for form validation scenarios where you want to see every failing field in one run instead of fixing one, rerunning, finding the next.

---

15. Ignoring / Disabling Tests

```
@Test(enabled = false)
public void temporarilyDisabledTest() {
    // skipped by TestNG, shown as SKIPPED in report
}
```

Useful for flaky or work-in-progress tests without deleting code. Prefer this over commenting out code — it keeps the test visible in reports as intentionally skipped.

---

16. Timeouts

```
@Test(timeoutInMilliseconds = 1000)
public void mustFinishQuickly() throws InterruptedException {
    Thread.sleep(2000); // exceeds timeout → test fails with ThreadTimeoutException
}
```

Guards against hanging tests (e.g., a stuck network call). If the method doesn't complete within the given milliseconds, TestNG fails it and interrupts the thread.

---

17. Exception Testing

```
@Test(expectedExceptions = ArithmeticException.class)
public void divideByZero() {
    int result = 10 / 0;
}
```

The test **passes** only if the specified exception is thrown; if no exception or a different exception is thrown, the test fails. You can pass multiple exception types: `expectedExceptions = { ArithmeticException.class, NullPointerException.class }`.

For message-level checks, combine with `expectedExceptionsMessageRegExp`:

```
@Test(expectedExceptions = IllegalArgumentException.class,
      expectedExceptionsMessageRegExp = "Invalid input.*")
```

```
public void validateInput() {
    throw new IllegalArgumentException("Invalid input: negative value");
}
```

---

18. Listeners

Listeners hook into the test lifecycle to add cross-cutting behavior (logging, screenshots on failure, custom reporting).

ITestListener — most commonly used:

```
import org.testng.ITestListener;
import org.testng.ITestResult;

public class TestListenerImpl implements ITestListener {

    @Override
    public void onTestFailure(ITestResult result) {
        System.out.println("FAILED: " + result.getName());
        // e.g., capture screenshot here
    }

    @Override
    public void onTestSuccess(ITestResult result) {
        System.out.println("PASSED: " + result.getName());
    }

    @Override
    public void onTestSkipped(ITestResult result) {
        System.out.println("SKIPPED: " + result.getName());
    }
}
```

Register it either in code:

```
@Listeners(TestListenerImpl.class)
public class MyTest { ... }
```

or in `testng.xml`:

```
<suite name="Suite">
    <listeners>
        <listener class-name="com.demo.TestListenerImpl"/>
    </listeners>
    ...
</suite>
```

Other common listener interfaces: `ISuiteListener` (suite start/finish), `IAnnotation-`

`Transformer` (modify annotations at runtime, e.g., inject retry analyzer globally), `IReporter` (build custom reports).

---

19. Retry Failed Tests

TestNG has no automatic retry by default — you implement `IRetryAnalyzer`:

```
import org.testng.IRetryAnalyzer;
import org.testng.ITestResult;

public class RetryAnalyzer implements IRetryAnalyzer {
    private int retryCount = 0;
    private static final int MAX_RETRY = 2;

    @Override
    public boolean retry(ITestResult result) {
        if (retryCount < MAX_RETRY) {
            retryCount++;
            return true;
        }
        return false;
    }
}
```

Attach per test:

```
@Test(retryAnalyzer = RetryAnalyzer.class)
public void flakyTest() { ... }
```

To apply globally without annotating every method, use `IAnnotationTransformer` to inject the retry analyzer into every `@Test` during suite setup — a common ask in senior-level interviews ("how do you add retry logic to hundreds of existing tests without editing each one").

---

20. Reporting

TestNG generates default reports automatically under `test-output/`:

- `index.html` — full suite summary
- `emailable-report.html` — condensed, good for CI email notifications
- `testng-results.xml` — machine-readable results, consumed by CI dashboards (Jenkins TestNG plugin reads this)

For richer reports, teams commonly integrate **ExtentReports** or **Allure**, hooking into `ITestListener` to log steps, screenshots, and pass/fail status with charts. TestNG itself doesn't produce screenshots — that's your `onTestFailure()` listener logic using Selenium's `TakesScreenshot`.

---

21. TestNG with Selenium (integration pattern)

Typical structure in an automation framework:

```java
public class BaseTest {
    protected WebDriver driver;

    @BeforeMethod
    public void setUp() {
        driver = new ChromeDriver();
        driver.manage().window().maximize();
    }

    @AfterMethod
    public void tearDown() {
        if (driver != null) driver.quit();
    }
}

public class LoginTest extends BaseTest {

    @Test
    public void validLogin() {
        driver.get("https://example.com/login");
        // ... Selenium actions
        Assert.assertTrue(driver.getTitle().contains("Dashboard"));
    }
}
```

`@BeforeMethod`/`@AfterMethod` guarantee a fresh browser session per test — critical for test isolation. Combine with `ThreadLocal<WebDriver>` when running `parallel="methods"`.

---

22. Best Practices

- Never rely on default method execution order — use `priority` or `dependsOnMethods` explicitly.
- Keep test data out of code — feed `@DataProvider` from Excel/CSV/JSON/DB.
- Use `groups` to separate smoke/regression/sanity so CI pipelines can run subsets fast.
- Use `ThreadLocal` for any shared resource (WebDriver, DB connection) when running in parallel.
- Prefer soft assertions for multi-field validation, hard assertions for blocking preconditions.

- Externalize environment config (URLs, browser, credentials) via `testng.xml` parameters or a properties file, not hardcoded strings.
- Keep `@BeforeSuite`/`@BeforeClass` lightweight — heavy setup there slows every run even when only one test is needed.

---

23. Interview Cross-Questions (50 Q&A)

Q1. Why TestNG over JUnit? Native support for dependency management, grouping, parallel execution, and data providers without extra libraries; JUnit needs additional annotations/extensions to match this.

Q2. What happens if a method a test `dependsOnMethods` fails? The dependent test is marked **SKIPPED**, not failed, and does not execute.

Q3. Does `priority` guarantee that a lower-priority test runs even if an earlier one fails? Yes — priority only affects order, not skip behavior. Only `dependsOnMethods`/`dependsOnGroups` cause skipping.

Q4. What's the default priority value if none is specified? 0.

Q5. Can two tests have the same priority? Yes; TestNG then falls back to non-deterministic (reflection-based) order between them, so don't rely on it for ordering ties.

Q6. How many times does `@BeforeMethod` run for a class with 5 `@Test` methods? 5 times — once before each test method.

Q7. How many times does `@BeforeClass` run for the same class? Once, regardless of the number of test methods.

Q8. What's the difference between `@BeforeTest` and `@BeforeClass`? `@BeforeTest` runs once per `<test>` tag in testng.xml (can span multiple classes); `@BeforeClass` runs once per Java class.

Q9. How do you pass a value from testng.xml into a test method? `<parameter>` tag in XML + `@Parameters("name")` annotation on the method parameter.

Q10. Can `@Parameters` values change per test row like a DataProvider? No — XML parameters are static per suite/test run. Use `@DataProvider` for multiple/varying rows.

Q11. What return type must a `@DataProvider` method have? `Object[][]` (or `Iterator<Object[]>` for large/streamed datasets).

Q12. How do you make a DataProvider run its rows in parallel? `@DataProvider(name = "x", parallel = true)`.

Q13. Can a DataProvider live in a different class than the test? Yes, via `dataProviderClass` attribute on `@Test`.

Q14. What's the difference between a hard assertion and a soft assertion? Hard assertion (`Assert`) halts the test on first failure. Soft assertion (`SoftAssert`) records failures and continues; you must call `assertAll()` to surface them.

Q15. What happens if you forget to call `assertAll()` after using SoftAssert? The test reports as passed even if soft assertions failed internally — a classic bug.

Q16. How do you mark a test to be skipped without deleting it? `@Test(enabled = false)`.

Q17. How do you fail a test if it doesn't finish within a time limit? `@Test(timeoutInMilliseconds = X)`.

Q18. How do you test that a method throws a specific exception? `@Test(expectedExceptions = SomeException.class)`.

Q19. What happens if the method throws no exception when `expectedExceptions` is set? The test fails, because the expected exception never occurred.

Q20. What are the three main `parallel` modes in testng.xml? `methods`, `classes`, `tests` (also `instances`).

Q21. If `parallel="methods"` is set with `thread-count="1"`, does it still run in parallel? No effectively — one thread means sequential execution despite the parallel flag.

Q22. Why is ThreadLocal important with parallel Selenium tests? Without it, all threads share one WebDriver instance, causing tests to interfere with each other's browser state.

Q23. What interface do you implement to add custom retry logic for failed tests? `IRetryAnalyzer`.

Q24. How can you apply a retry analyzer to every test without annotating each method? Implement `IAnnotationTransformer` and inject the retry analyzer during suite setup, registered as a listener.

Q25. Name three types of TestNG listeners. `ITestListener`, `ISuiteListener`, `IAnnotationTransformer` (also `IReporter`, `IInvokedMethodListener`).

Q26. Where do you register a listener without modifying test classes? In `testng.xml` under `<listeners>`.

Q27. What report files does TestNG generate by default? `index.html`, `emailable-report.html`, `testng-results.xml` under `test-output/`.

Q28. Which of the default TestNG output files is typically consumed by CI tools like Jenkins? `testng-results.xml`.

Q29. What's the difference between `groups` and `dependsOnGroups`? `groups` tags a test for inclusion/exclusion in XML; `dependsOnGroups` makes a test wait on (and be skipped if) an entire group's execution/failure.

Q30. Can a single `@Test` method belong to multiple groups? Yes: `@Test(groups = {"smoke", "regression"})`.

Q31. How do you exclude a group from a run in testng.xml? `<exclude name="groupName"/>` inside `<groups><run>`.

Q32. What does `alwaysRun = true` do on a dependent test? Forces the test to execute even if its declared dependency failed.

Q33. Is method execution order guaranteed within a class if no priority or dependency is set? No — it depends on JVM reflection order, not source-code or alphabetical order.

Q34. What's the hierarchy of elements in testng.xml? Suite → Test → Classes/Packages → Methods.

Q35. Can one testng.xml file define multiple independent test runs? Yes, via multiple `<test>` tags, each potentially targeting different classes/groups/parameters.

Q36. How do you include an entire package instead of listing classes individually? `<packages><package name="com.demo.*"/></packages>`.

Q37. What's the purpose of `@Factory` in TestNG? Creates multiple instances of a test class dynamically (e.g., with different constructor parameters) at runtime — useful when you need to run the same test class against different configurations.

Q38. How is `@Factory` different from `@DataProvider`? `@DataProvider` supplies data rows to a single method; `@Factory` creates multiple test *class instances*, each potentially running its own full suite of `@Test` methods with different constructor state.

Q39. What happens to `@AfterMethod` if the `@Test` method throws an exception? It still runs — `@AfterMethod`/`@AfterClass`/`@AfterSuite` execute regardless of test outcome, ensuring cleanup (e.g., `driver.quit()`).

Q40. Can you assign a description to a test method for reporting purposes? Yes: `@Test(description = "Verifies login with valid credentials")`.

Q41. How do you skip a group of tests conditionally at runtime, not via XML? Throw `org.testng.SkipException` inside the test or a `@BeforeMethod`.

Q42. What does `invocationCount` do on `@Test`? Repeats the same test method N times: `@Test(invocationCount = 5)` — useful for stress-testing flaky or timing-sensitive logic.

Q43. How do `invocationCount` and `@DataProvider` differ? `invocationCount` reruns the same test with the same (or no) input N times; `@DataProvider` runs it once per distinct data row.

Q44. Can you combine `groups` and `dependsOnMethods` on the same test? Yes, both attributes can coexist on one `@Test` annotation.

Q45. What's `verbose` in testng.xml used for? Controls console log detail level (0 = silent, higher = more diagnostic output) — useful for debugging suite execution issues.

Q46. If a `@BeforeSuite` method throws an exception, what happens to the rest of the suite? All tests in the suite are skipped, since the suite-level setup never completed successfully.

Q47. How do you run only specific methods of a class from testng.xml? `<class name="com.demo.Test1"><methods><include name="methodName"/></methods></class>`.

Q48. What's the benefit of `@Listeners` over registering listeners in XML? It's code-based and travels with the class (self-contained), useful when you don't control the XML or want default behavior baked in; XML registration is preferred when you want it configurable per environment.

Q49. How does TestNG differ from JUnit in reporting test dependencies visually? TestNG's default HTML report explicitly shows SKIP status for dependency-chain failures; JUnit has no native equivalent to `dependsOnMethods`, so this distinction doesn't exist there.

Q50. What's a common real-world bug caused by misunderstanding `@BeforeClass` vs `@BeforeMethod`? Putting stateful setup (like initializing a counter or a shared object) in `@BeforeClass` when it needs to be reset per test — causing state to leak between test methods since `@BeforeClass` runs only once.

End of course.