TestNG Priority Dependencies and more

In **TestNG**, *prioritization* and *dependency* help control the **execution order of test cases**. Here's a detailed explanation of both:

🔹 1. Prioritization in TestNG

By default, TestNG executes test methods in **lexicographical order** (alphabetically by method name), **not in the order you write them**.

To override this, use the `priority` attribute in the `@Test` annotation.

✅ Syntax:

```
java

@Test(priority = <number>)
```

📘 Example:

```
java

import org.testng.annotations.Test;

public class PriorityTest {

    @Test(priority = 2)
    public void testA() {
        System.out.println("Test A");
    }

    @Test(priority = 1)
    public void testB() {
        System.out.println("Test B");
    }

    @Test(priority = 3)
    public void testC() {
        System.out.println("Test C");
    }
}
```

🔍 Output:

```
css

Test B
Test A
Test C
```

🔑 Notes:

- Lower priority value = higher precedence.

- If two methods have the **same priority**, they run in **lexicographical order**.
- If `priority` is not mentioned, it's treated as `priority = 0`.

🔹 2. Dependency in TestNG

TestNG allows tests to run **only after certain other tests have passed**, using `dependsOnMethods` or `dependsOnGroups`.

✅ Syntax:

```
java

@Test(dependsOnMethods = {"methodName"})
```

📘 Example:

```
java

import org.testng.annotations.Test;

public class DependencyTest {

    @Test
    public void login() {
        System.out.println("Login Successful");
    }

    @Test(dependsOnMethods = {"login"})
    public void accessDashboard() {
        System.out.println("Dashboard Accessed");
    }

    @Test(dependsOnMethods = {"accessDashboard"})
    public void logout() {
        System.out.println("Logout Successful");
    }
}
```

🔍 Output:

```
nginx

Login Successful
Dashboard Accessed
Logout Successful
```

If the `login()` test fails, the other two will be **skipped** (not failed).

🔄 `alwaysRun = true` (Optional)

Used when you want a test to **run even if the dependent test fails**.

```
java

@Test(dependsOnMethods = {"login"}, alwaysRun = true)
public void logout() {
    System.out.println("Logout Attempted");
}
```

✅ Best Practices

- Use `priority` when you want to control **execution order**.
- Use `dependsOnMethods` when a test **requires another to pass** (functional dependency).
- Don't mix both heavily unless necessary—it can make tests harder to manage.

Would you like a real-time **TestNG project example** combining both priority and dependency?

In **TestNG**, the `enabled = false` attribute is used to **disable a test method**, meaning it will be **skipped** during execution.

🔹 Purpose of `enabled = false`

- Temporarily skip a test without deleting or commenting it.
- Useful during development or debugging.
- Helps when a test is known to fail due to an open bug or dependency issue.

✅ Syntax

```
java

@Test(enabled = false)
public void skippedTest() {
    System.out.println("This test will not run.");
}
```

📘 Example

```
java

import org.testng.annotations.Test;

public class EnabledTest {

    @Test
    public void test1() {
        System.out.println("Test 1 Executed");
    }

    @Test(enabled = false)
    public void test2() {
        System.out.println("Test 2 Executed");
    }

    @Test
    public void test3() {
        System.out.println("Test 3 Executed");
    }
}
```

🔍 Output:

```
nginx

Test 1 Executed
Test 3 Executed
```

`test2()` is skipped and will not be shown in the output.

📝 Notes:

- `enabled = true` is the default behavior.
- Skipped tests still appear in the TestNG report under **"Skipped"** section.
- Can be used with `@BeforeMethod`, `@AfterMethod`, etc., too.

Would you like to see how this works in a real Maven/TestNG project setup?

In **TestNG**, the `invocationCount` attribute is used to **execute a test method multiple times** in a single run.

🔹 Purpose of `invocationCount`

- To **repeat** a test method **N times**.
- Useful for:
  - **Load testing**
  - **Stability testing**
  - **Loop-based validations**
  - **Retry logic demonstration**

✅ Syntax:

```
java

@Test(invocationCount = <number>)
```

📘 Example:

```
java

import org.testng.annotations.Test;

public class InvocationCountExample {

    @Test(invocationCount = 3)
    public void repeatTest() {
        System.out.println("Repeated Test Executed");
    }
}
```

🔍 Output:

```
mathematica

Repeated Test Executed
Repeated Test Executed
Repeated Test Executed
```

The `repeatTest()` method is run **3 times**.

🔧 With `threadPoolSize` (for parallel execution)

You can also combine `invocationCount` with `threadPoolSize` for **parallel test execution**:

```java
@Test(invocationCount = 5, threadPoolSize = 2)
public void parallelTest() {
    System.out.println("Thread ID: " + Thread.currentThread().getId());
}
```

This will run the test 5 times using 2 threads in parallel.

🔎 Points to Remember:

Attribute	Description
`invocationCount`	Number of times a test method will be invoked
`threadPoolSize`	Number of threads to use (optional, for parallel runs)
Default Value	**`invocationCount = 1`** (means normal single execution)

⚠️ Use Cases

Scenario	Use `invocationCount`?
Load testing a login API	✅ Yes
Checking retry mechanisms	✅ Yes
Simple test with no loop	❌ Not needed

Would you like to see a **real-world use case** of **`invocationCount`** in API or UI testing using Selenium or RestAssured?

Great! Let me give you **two real-world examples** using `invocationCount` in:

1. ✅ **Selenium WebDriver UI Testing**
2. ✅ **API Testing with RestAssured**

🔹 1. `invocationCount` in Selenium WebDriver UI Testing

📘 Scenario:

You want to test a **login page** multiple times to check for consistency or to stress test it.

✅ Code Example:

```
java

import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import org.openqa.selenium.By;
import org.testng.annotations.*;

public class LoginStressTest {

    WebDriver driver;

    @BeforeMethod
    public void setup() {
        driver = new ChromeDriver();
        driver.get("https://example.com/login");
    }

    @Test(invocationCount = 5)
    public void loginTest() {
        driver.findElement(By.id("username")).sendKeys("admin");
        driver.findElement(By.id("password")).sendKeys("password123");
        driver.findElement(By.id("loginBtn")).click();

        System.out.println("Login Attempted - Thread ID: " + Thread.currentThread().getId());

        // Add assertions here if needed
    }

    @AfterMethod
    public void tearDown() {
        driver.quit();
    }
}
```

🔍 What happens:

- The test method `loginTest()` is executed **5 times**.
- Useful for checking behavior under repeated logins or to simulate light load testing.

🔹 2. `invocationCount` in API Testing with RestAssured

📘 Scenario:

You want to hit a **GET API** endpoint multiple times to test response consistency or performance.

✅ Code Example:

```java
import io.restassured.RestAssured;
import io.restassured.response.Response;
import org.testng.annotations.Test;

import static org.hamcrest.Matchers.*;

public class ApiStressTest {

    @Test(invocationCount = 5)
    public void getUserDataRepeatedly() {
        Response response = RestAssured.get("https://jsonplaceholder.typicode.com/posts/1");

        response.then().statusCode(200)
                .body("userId", equalTo(1));

        System.out.println("API called - Thread ID: " + Thread.currentThread().getId());
    }
}
```

🔍 What happens:

- The GET request to the endpoint runs **5 times**.
- Ensures the API behaves correctly and returns the expected response every time.

🧪 Want to Try It?

You can run both examples using:

- **TestNG XML suite**
- **IntelliJ IDEA / Eclipse**
- Or through **Maven `test` goal**

Would you like me to generate a ready-to-run Maven/TestNG project structure with these examples for hands-on practice?

In **TestNG**, the `timeOut` attribute is used to **specify the maximum time (in milliseconds)** a test method is allowed to run. If it exceeds this time, the test **fails automatically** with a timeout exception.

🔹 Purpose of `timeOut`

- Prevent tests from **hanging indefinitely**
- Validate **performance or response time** requirements
- Detect **slow operations** in UI/API logic

✅ Syntax

```
java

@Test(timeOut = <milliseconds>)
```

📘 Example 1: Simple Timeout Test

```
java

import org.testng.annotations.Test;

public class TimeoutExample {

    @Test(timeOut = 2000)  // 2 seconds
    public void fastTest() throws InterruptedException {
        Thread.sleep(1000); // finishes in 1 second
        System.out.println("fastTest completed within time.");
    }

    @Test(timeOut = 2000)
    public void slowTest() throws InterruptedException {
        Thread.sleep(3000); // sleeps for 3 seconds
        System.out.println("slowTest should fail due to timeout.");
    }
}
```

🔍 Output:

```
pgsql

fastTest completed within time.
slowTest FAILED due to timeout (exceeded 2000 ms)
```

🔹 Real-World Use Cases

✅ 1. Selenium UI Testing

```java
@Test(timeOut = 10000)
public void pageLoadTest() {
    driver.get("https://example.com");
    // Assert title or login element presence
}
```

Fails if the page doesn't load in 10 seconds.

✅ 2. API Testing with RestAssured

```java
@Test(timeOut = 3000)
public void apiResponseTimeTest() {
    RestAssured.given()
        .when()
        .get("https://jsonplaceholder.typicode.com/posts/1")
        .then()
        .statusCode(200);
}
```

Fails if the API takes longer than 3 seconds to respond.

🔎 Difference: `timeOut` vs `invocationTimeout`

Attribute	Scope	Description
`timeOut`	Per invocation	Each test call must complete within time
`invocationTimeout`	Entire test run of repeated calls	Used with **`invocationCount`** to limit total time

Example:

```java
@Test(invocationCount = 5, invocationTimeOut = 10000)
public void multiRunTest() {
    // All 5 invocations must complete within 10 seconds
}
```

Would you like a **Selenium or API testing demo** showing a timeout in real-time?