
# TestNG

# What is TestNG?

- TestNG is a testing framework designed to simplify a broad range of testing needs, from *unit testing* to *integration testing*.
- TestNG is a testing framework inspired from *JUnit* and *NUnit*.
- It has more functionalities compared to JUnit and NUnit and it makes more *powerful* and *easier to use*.
- In TestNG, NG stands for "*Next Generation*".

# Course Content

- Introduction to TestNG
- TestNG Installation and Setup
- TestNG Annotations
- Understanding TestNG configuration (testng.xml)
- TestNG Assertions
- Prioritizing and Grouping Tests
- Parameterizing Tests
- TestNG Listeners
- Using TestNG in Test Automation
- TestNG Best Practices & Tips

# Features of TestNG:

- XML Based Test Configuration
- Before and After Annotations
- Dependant Methods & Groups
- Data Driven Testing
- Better Reporting
- Parameterization of Test Methods
- Multi-threaded/Parallel Execution

# Prerequisites:

1. Java should be installed in the system

```
java -version
openjdk version "11.0.19" 2023-04-18
OpenJDK Runtime Environment Temurin-11.0.19+7 (build 11.0.19+7)
OpenJDK 64-Bit Server VM Temurin-11.0.19+7 (build 11.0.19+7, mixed mode)
```

2. Maven should be installed in the system

```
mvn -version
Apache Maven 3.8.4 (9b656c72d54e5bacbed989b64718c159fe39b537)
Maven home: ~/softwares/apache-maven-3.8.4
```

# Installing TestNG:

1. Add below maven dependency to pom.xml file

```
<!-- https://mvnrepository.com/artifact/org.testng/testng -->
<dependency>
    <groupId>org.testng</groupId>
    <artifactId>testng</artifactId>
    <version>7.10.2</version>
    <scope>test</scope>
</dependency>
```

2. (Optional) Get the TestNG IDE plugin, if applicable - for ex: Eclipse IDE

   *IntelliJ Plugin to generate TestNG XML automatically*
   *https://plugins.jetbrains.com/plugin/9556-create-testng-xml*

# JAVA Annotations:

Java Annotation is a tag that represents the metadata i.e, attached within class, interface, methods or fields to indicate some additional information which can be used by Java compiler and JVM.

They are used to provide additional information, so it is an alternative option for XML and Java marker interfaces.

*TestNG uses this feature provided by Java to define its own annotations and develop an execution framework by using it.*


# Built-In Annotations:

**Used in Java Code:**

- @override
- @deprecated
- @suppresswarnings

**Used in other annotations:**

- @target
- @retention
- @documented

# Important Terminology:

**Suite**

A ***suite*** is made of one or more ***tests***

**Test**

A ***test*** is made of one or more ***classes***

**Class**

A ***class*** is made of one or more ***methods***

# TestNG Annotations:

- Annotation in TestNG is the set of code that controls how method below them has to be executed. i.e. the order of the execution of method below them will be decided by an annotation that you give.

- TestNG supports a powerful set of different annotations that run your tests in an organized and predictable manner.

- An annotation is always preceded by a @ symbol which allows performing some java logic before and after a certain point.

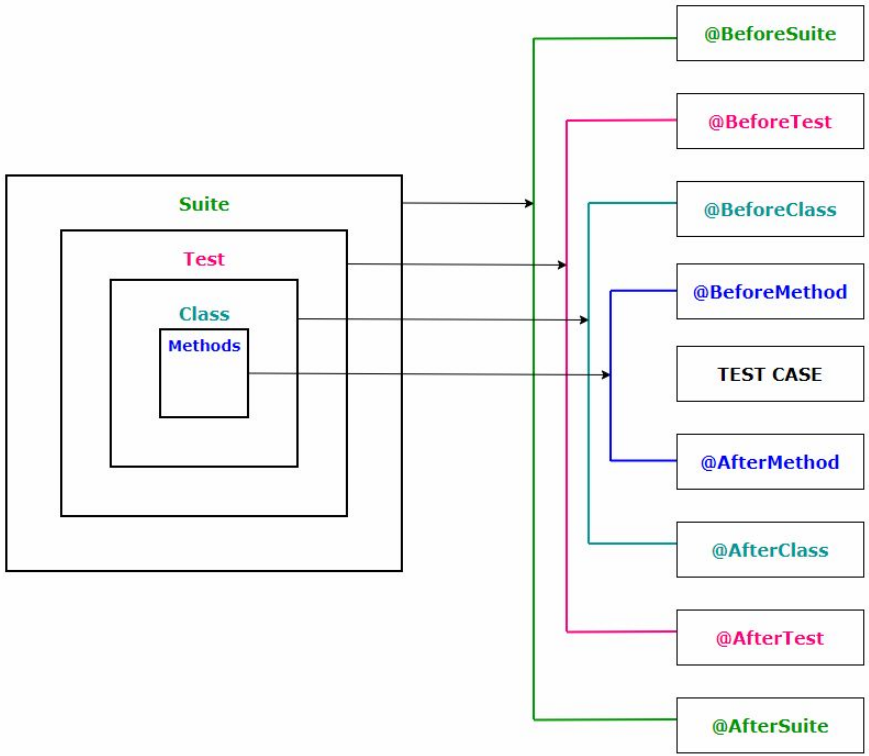
| | |
|---|---|
| **@Test** | Methods annotated with @Test are called test methods that serve as a unit test. In @Test methods, we will write the logic of the application which we want to automate. |
| **@BeforeSuite** | The annotated method will be run before all tests in this suite have run |
| **@AfterSuite** | The annotated method will be run after all tests in this suite have run |
| **@BeforeTest** | The annotated method will be run before any test method belonging to the classes inside the <test> tag is run |
| **@AfterTest** | The annotated method will be run after all the test methods belonging to the classes inside the <test> tag have run |
| **@BeforeGroups** | The list of groups that this configuration method will run before. This method is guaranteed to run shortly before the first test method that belongs to any of these groups is invoked |
| **@AfterGroups** | The list of groups that this configuration method will run after. This method is guaranteed to run shortly after the last test method that belongs to any of these groups is invoked |
| **@BeforeClass** | The annotated method will be run before the first test method in the current class is invoked |
| **@AfterClass** | The annotated method will be run after all the test methods in the current class have been run |
| **@BeforeMethod** | The annotated method will be run before each test method |
| **@AfterMethod** | The annotated method will be run after each test method |

# TestNG Annotation Contd.

| | |
|---|---|
| **@Parameters** | Used to pass parameters to a test method. The values of parameters are provided using the testng.xml file at runtime.<br><br>This can be used with any of the Before/After, Factory, Test annotated methods. |
| **@DataProvider** | It marks a method as a data providing method for a test method. It returns an object double array(Object [ ][ ]) as data to the test method. |
| **@Groups** | Groups test methods for selective execution. |
| **@Listeners** | This annotation is defined as an array of test listeners classes that extends org.testng.ITestNGListener. It is used to track the execution status and to customize TestNG reports or logs. |

| **@Test** | **Marks a class or a method as part of the test.** |
|---|---|
| `alwaysRun` | If set to true, this test method will always be run even if it depends on a method that failed. |
| `dataProvider` | The name of the data provider for this test method. |
| `dataProviderClass` | The class where to look for the data provider. If not specified, the data provider will be looked on the class of the current test method or one of its base classes. If this attribute is specified, the data provider method needs to be static on the specified class. |
| `dependsOnGroups` | The list of groups this method depends on. |
| `dependsOnMethods` | The list of methods this method depends on. |
| `description` | The description for this method. |
| `enabled` | Whether methods on this class/method are enabled. |
| `expectedExceptions` | The list of exceptions that a test method is expected to throw. If no exception or a different than one on this list is thrown, this test will be marked a failure. |
| `groups` | The list of groups this class/method belongs to. |
| `invocationCount` | The number of times this method should be invoked. |
| `invocationTimeOut` | The maximum number of milliseconds this test should take for the cumulated time of all the invocationcounts. This attribute will be ignored if invocationCount is not specified. |
| `priority` | The priority for this test method. Lower priorities will be scheduled first. |
| `successPercentage` | The percentage of success expected from this method |
| `singleThreaded` | If set to true, all the methods on this test class are guaranteed to run in the same thread, even if the tests are currently being run with parallel="methods". This attribute can only be used at the class level and it will be ignored if used at the method level. Note: this attribute used to be called `sequential` (now deprecated). |
| `timeOut` | The maximum number of milliseconds this test should take. |
| `threadPoolSize` | The size of the thread pool for this method. The method will be invoked from multiple threads as specified by invocationCount.<br>Note: this attribute is ignored if invocationCount is not specified |

***TestNG life cycle*** defines the way and order in which the annotated method is executed in order.



# Hierarchy of TestNG Annotations Order

@BeforeSuite

@BeforeTest

@BeforeClass

@BeforeMethod

@Test

@AfterMethod

@AfterClass

@AfterTest

@AfterSuite

# TestNG XML file (testng.xml)

- Testng.xml file is a configuration file (XML file) for TestNG in which we can create test suites, test groups, mark tests for parallel execution, add listeners, and pass parameters to test script.

- It defines the runtime definition of a test suite. The **testng.xml** file provides us different options to include packages, classes, and independent test methods in our test suites.

- Using the testng.xml file, we can also configure multiple tests in a single test suite and run them in a multithreaded environment.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">
<suite name="All Test Suite">
    <test verbose="2" preserve-order="true" name="SampleTests">
        <classes>
            <class name="io.learn.BeforeAfterClassTest"/>
            <class name="io.learn.BeforeAfterMethodTest"/>
        </classes>
    </test>
</suite>
```

# <suite>

**<suite>** is the root tag in the testng.xml file. It is represented by one XML file. It contains one or more **<test>** tags. It represents one **testng.xml** file, which is made of several <test> tags.

## Attributes:

- ***name:*** It is name of the suite. It is a mandatory attribute.
- ***verbose:*** It is an attribute that is mostly used when reporting a bug or when trying to diagnose the execution of test run.
- ***parallel:*** It is used to run multiple tests in parallel. The value that it takes will either be methods or classes.
- ***thread-count:*** The thread-count attribute is used to run the maximum number of threads for each suite, if the parallel mode is enabled (otherwise ignore it).
- ***annotations:*** It is the type of annotations that you are using in your tests.
- ***time-out:*** It is used to declare a time-out period that will be applied to all test methods in the suite.

# <test>

**<test>** is a child tag of <suite> tag. It contains one or more TestNG classes that have to run. <test> tag can also be parent of tags that can be declared on <suite>.

For example, <group> tag, <parameter> tag, and <package>. It takes only mandatory attribute name. <test> takes also other attributes such as verbose, parallel, and time-out.

# <classes> and <class>

**<classes>** is a child tag of <test>. It enables you to define Java classes that have to be included in the test run. A class can contain at least one TestNG annotation and one or more test methods. It is represented by the **<class>** tag. It takes only an attribute name.

```
<!DOCTYPE suite SYSTEM "https://testng.org/testng-1.0.dtd" >
<suite name="Suite1" verbose="1" >
<test name="First Test" >
<classes>
      <class name = "class1" />
</classes>
</test>
<test name = "Second Test">
<classes>
     <class name = "testngtests.Class2"/>
     <class name = "testngtests.Class3"/>
</classes>
</test>
</suite>
```

# <packages> and <package>

<packages> tag is also a child tag of <test> tag which is used to define a set of Java packages to include or exclude from the suite. You can specify package names instead of class names just like given below:

```
<!DOCTYPE suite SYSTEM "https://testng.org/testng-1.0.dtd" >
<suite name="Package suite">
<test name="First Test" >
<packages>
        <package name = "test.sample" />
 </packages>
</test>
</suite>
```

# <groups>

**<groups>** tag is a child of <test> tag inside test suite. It is used to run tests in groups. It contains a **<run>** tag which represents the group that needs to be run.

- The <include> tag specifies the name of a group that has to be executed.
- The <exclude> tag represents the name of group that has not to be executed.

```
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd" >
<suite name="Test Suite">
<test name="Group Test">
<groups>
    <run>
        <exclude name = "GroupTest1" />
        <include name = "GroupTest2" />
    </run>
 </groups>
<classes>
     <class name = "testngtests.TestGroupss"/>
</classes>
</test>
</suite>
```

# <methods>

**<methods>** tag is an optional child of <classes>. This tag is used to execute on the basis of include or exclude test methods of a given class. We can use any number of <include> or <exclude> tag within <methods> tag.

```
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd" >
<suite name="Test Suite">
<test name="Methods Test">
<classes>
    <class name = "test.IndividualMethodsTest">
  <methods>
      <include name = "testMethod1" />
      <exclude name = "testMethod"2 />
  </methods>
 </class>
</classes>
</test>
</suite>
```

# Grouping Tests in TestNG:

## What is Grouping?

- **Definition:** Grouping allows you to categorize test methods into named groups, making it easier to run specific subsets of tests.
- **Purpose:** Helps in organizing tests, managing test execution, and running only a relevant subset of tests based on criteria.

## Benefits of Grouping

- **Organization:** Group related tests together for better management.
- **Selective Execution:** Run specific groups of tests as needed (e.g., smoke tests, regression tests).
- **Flexibility:** Easily include or exclude groups from test suites.

# Defining and Using Groups:

### 1. Defining Groups in Test Methods

- **Syntax:** Use the groups attribute in the @Test annotation to assign a test method to one or more groups.

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

public class GroupExample {

    @Test(groups = {"smoke"})
    public void smokeTest1() {
        System.out.println("Smoke Test 1");
    }

    @Test(groups = {"regression"})
    public void regressionTest1() {
        System.out.println("Regression Test 1");
    }

    @Test(groups = {"smoke", "regression"})
    public void smokeAndRegressionTest() {
        System.out.println("Smoke and Regression Test");
    }
}
```

### 2. Running Tests by Group

- **Configuration:** Use TestNG XML configuration to specify which groups to include or exclude in the test suite.

```xml
<!-- testng.xml -->
<suite name="Test Suite">
    <test name="Smoke Tests">
        <groups>
            <run>
                <include name="smoke"/>
            </run>
        </groups>
        <classes>
            <class name="com.example.GroupExample"/>
        </classes>
    </test>
    <test name="Regression Tests">
        <groups>
            <run>
                <include name="regression"/>
            </run>
        </groups>
        <classes>
            <class name="com.example.GroupExample"/>
        </classes>
    </test>
</suite>
```

# Priority in TestNG:

## What is Priority?

- **Definition:** Priority is an attribute in the @Test annotation that specifies the order in which test methods are executed within a class.
- **Purpose:** Ensures that tests are run in a specific sequence, which is useful for scenarios where the order of execution matters.

## Benefits of Using Priority

- **Controlled Execution:** Define the order of tests to handle dependencies or set up conditions.
- **Flexibility:** Easily adjust the sequence of test methods without changing the test code.

# Defining Priority in Test Methods

### 1. Basic Priority Usage

- **Syntax:** Use the `priority` attribute in the `@Test` annotation to assign a priority value to test methods. Lower values indicate higher priority.

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

public class PriorityExample {

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

    @Test(priority = 2)
    public void test2() {
        System.out.println("Test 2 with priority 2");
    }

    @Test(priority = 0)
    public void test0() {
        System.out.println("Test 0 with priority 0");
    }
}
```

### 2. Default Priority

- **Behavior:** If no priority is specified, TestNG assigns a default priority of 0 to methods. Without explicit priority values, TestNG executes methods in the ascending order.
- **Order:** Methods with the same priority are executed in an undefined order relative to each other.

# Combining Priority with Grouping

## 1. Prioritizing Grouped Tests

- **Example:** You can use priority in combination with grouping to control execution order within specific groups.

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

public class GroupAndPriorityExample {

    @Test(groups = {"smoke"}, priority = 1)
    public void smokeTest1() {
        System.out.println("Smoke Test 1 with priority 1");
    }

    @Test(groups = {"smoke"}, priority = 2)
    public void smokeTest2() {
        System.out.println("Smoke Test 2 with priority 2");
    }

    @Test(groups = {"regression"}, priority = 0)
    public void regressionTest() {
        System.out.println("Regression Test with priority 0");
    }
}
```

## 2. Execution Scenario

- **Test Suite Execution:** You can configure TestNG to run specific groups with priorities, controlling both group inclusion and method execution order.

```xml
<!-- testng.xml -->
<suite name="Test Suite">
    <test name="Smoke and Regression Tests">
        <groups>
            <run>
                <include name="smoke"/>
                <include name="regression"/>
            </run>
        </groups>
        <classes>
            <class name="com.example.GroupAndPriorityExample"/>
        </classes>
    </test>
</suite>
```

# Best Practices and Key Points:

## 1. Best Practices

- **Logical Priorities:** Use priority values logically to reflect test dependencies or setup/teardown sequences.
- **Maintainability:** Avoid hard-coding priority values too rigidly; aim for a balance between order and maintainability.

## 2. Key Points to Remember

- **Priority Values:** Lower values are executed first.
- **Default Behavior:** Methods without priority are executed in the ascending order of method name.
- **Combination with Groups:** Use priority and groups together for complex test scenarios.

# Parameterizing Tests in TestNG:

## What is Parameterization?

- **Definition:** Parameterization allows you to run a test multiple times with different sets of data.
- **Purpose:** Helps in testing the same functionality with various inputs to ensure robustness and correctness.

## Why Use Parameterization?

- **Reusability:** Write a single test method and run it with different data sets.
- **Efficiency:** Reduces code duplication and enhances test coverage.
- **Maintainability:** Easier to manage test data and scenarios in one place.

# Parameterizing Tests in TestNG:

## Using @Parameters Annotation

- **Purpose:** Inject parameters from XML test suite configuration.

```xml
<!-- testng.xml -->
<suite name="Test Suite">
    <test name="Test">
        <parameters>
            <parameter name="username" value="testUser"/>
            <parameter name="password" value="testPass"/>
        </parameters>
        <classes>
            <class name="com.example.TestClass"/>
        </classes>
    </test>
</suite>
```

```java
// TestClass.java
import org.testng.annotations.Parameters;
import org.testng.annotations.Test;

public class TestClass {
    @Test
    @Parameters({"username", "password"})
    public void testLogin(String username, String password) {
        System.out.println("Username: " + username);
        System.out.println("Password: " + password);
        // Add assertions and test logic here
    }
}
```

# Data Provider in TestNG:

## What is @DataProvider?

- **Definition:** A TestNG annotation that provides data to test methods in the form of an array of objects.
- **Purpose:** Allows running a test method multiple times with different sets of data, which can be dynamic and more complex.

## Benefits of Using @DataProvider

- **Flexibility:** Supports complex data structures and dynamic data generation.
- **Reusability:** Define the data in one place and reuse it across multiple test methods.
- **Scalability:** Easily manage and scale data inputs for comprehensive testing.

# Data Provider in TestNG:

### 1. Defining a @DataProvider Method

- **Syntax:** Use the @DataProvider annotation on a method that returns Object[][] or Stream<Object[]>

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

public class DataProviderExample {

    @DataProvider(name = "loginData")
    public Object[][] createData() {
        return new Object[][] {
            {"user1", "pass1"},
            {"user2", "pass2"},
            {"user3", "pass3"}
        };
    }
}
```

### 2. Using the @DataProvider in Test Methods

- **Syntax:** Reference the @DataProvider by name in your test method using the dataProvider attribute.

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

public class LoginTest {

    @Test(dataProvider = "loginData")
    public void testLogin(String username, String password) {
        System.out.println("Username: " + username);
        System.out.println("Password: " + password);
        // Add assertions and test logic here
    }
}
```

# TestNG Assertions:

## What are Assertions?

- **Definition:** Assertions are used to verify that the actual results of your test match the expected results.
- **Purpose:** To ensure that your code behaves as expected and to identify failures in your tests.

## Benefits of Using Assertions

- **Validation:** Confirm that your code produces the correct outputs.
- **Error Detection:** Quickly identify where and why a test fails.
- **Automation:** Automate the verification process to ensure consistent results.

# Common TestNG Assertions:

### 1. assertEquals

- **Purpose:** Checks if two values are equal.
- **Syntax:** `Assert.assertEquals(actual, expected);`

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

public class AssertEqualsExample {

    @Test
    public void testEquals() {
        int actual = 5;
        int expected = 5;
        Assert.assertEquals(actual, expected,
                            "Values are not equal");
    }
}
```

### 2. assertTrue

- **Purpose:** Checks if a condition is true.
- **Syntax:** Assert.assertTrue(condition);

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

public class AssertTrueExample {

    @Test
    public void testTrue() {
        boolean condition = (5 > 3);
        Assert.assertTrue(condition,
                          "Condition is not true");
    }
}
```

# Common TestNG Assertions:

| Assertion | Description | Syntax | Example |
|---|---|---|---|
| `assertEquals` | Checks if two values are equal. | `Assert.assertEquals(actual, expected, [message]);` | `Assert.assertEquals(5, 5, "Values are not equal");` |
| `assertNotEquals` | Checks if two values are not equal. | `Assert.assertNotEquals(actual, expected, [message]);` | `Assert.assertNotEquals(5, 6, "Values should not be equal");` |
| `assertTrue` | Checks if a condition is true. | `Assert.assertTrue(condition, [message]);` | `Assert.assertTrue(5 > 3, "Condition is not true");` |
| `assertFalse` | Checks if a condition is false. | `Assert.assertFalse(condition, [message]);` | `Assert.assertFalse(5 < 3, "Condition is not false");` |
| `assertNull` | Checks if an object is null. | `Assert.assertNull(object, [message]);` | `Assert.assertNull(null, "Object is not null");` |
| `assertNotNull` | Checks if an object is not null. | `Assert.assertNotNull(object, [message]);` | `Assert.assertNotNull(new Object(), "Object is null");` |
| `assertSame` | Checks if two objects refer to the same object. | `Assert.assertSame(actual, expected, [message]);` | `Assert.assertSame(str1, str2, "Objects are not the same instance");` |
| `assertNotSame` | Checks if two objects do not refer to the same object. | `Assert.assertNotSame(actual, expected, [message]);` | `Assert.assertNotSame(str1, new String("Test"), "Objects are the same instance");` |
| `assertArrayEquals` | Checks if two arrays are equal. | `Assert.assertArrayEquals(actualArray, expectedArray, [message]);` | `Assert.assertArrayEquals(new int[]{1, 2, 3}, new int[]{1, 2, 3}, "Arrays are not equal");` |

# Best Practices for using Assertions:

### 1. Use Assertions to Validate Important Conditions

- **Focus:** Apply assertions to validate critical functionality and business logic.

### 2. Avoid Overuse of Assertions in a Single Test

- **Balance:** Keep tests focused and avoid cluttering with too many assertions; consider breaking them into smaller tests if necessary.

### 3. Provide Meaningful Failure Messages

- **Clarity:** Ensure failure messages are descriptive to facilitate debugging and understanding of the issue.

### 4. Combine Assertions with Test Data

- **Flexibility:** Use data-driven approaches with assertions to test a variety of scenarios efficiently.

# Introduction to Soft Assertions:

## What are Soft Assertions?

- **Definition:** Soft assertions allow test execution to continue even if one or more assertions fail.
- **Purpose:** To gather all assertion failures in a single test execution and report them at the end, rather than stopping at the first failure.

## Benefits of Using Soft Assertions

- **Comprehensive Testing:** Identify multiple issues in a single test run.
- **Improved Test Coverage:** Ensures that additional failures are detected and reported without halting execution.

# Using Soft Assertions in TestNG:

**1. Importing SoftAssert Class**

- **Syntax:** Use SoftAssert class from org.testng.asserts package.

**2. Creating a SoftAssert Instance**

- **Syntax:** Instantiate SoftAssert in your test class.

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

public class SoftAssertExample {

    @Test
    public void testSoftAssertions() {
        SoftAssert softAssert = new SoftAssert();

        // First assertion
        softAssert.assertEquals(5, 6, "First assertion failed");

        // Second assertion
        softAssert.assertTrue("Hello".startsWith("H"), "Second assertion failed");

        // Third assertion
        softAssert.assertNull(null, "Third assertion failed");

        // Call assertAll() at the end to collect and report all assertion failures
        softAssert.assertAll();
    }
}
```

# Best Practices for using Soft Assertions:

### 1. Use Soft Assertions for Comprehensive Testing

- **Scenario:** Ideal for scenarios where you want to verify multiple conditions within a single test case, such as in end-to-end tests.

### 2. Avoid Overusing Soft Assertions

- **Focus:** Use them judiciously to avoid masking critical issues; excessive use may make it harder to pinpoint root causes.

### 3. Ensure assertAll() is Called

- **Reminder:** Always call assertAll() at the end of the test to ensure all collected failures are reported.

### 4. Combine with Hard Assertions for Critical Checks

- **Strategy:** Use hard assertions for critical checks that should immediately fail the test, and soft assertions for less critical, multiple checks.

# Introduction to TestNG Reporting:

## What is TestNG Reporting?

- **Definition:** Reporting in TestNG involves generating detailed summaries of test results, including passed, failed, and skipped tests.
- **Purpose:** To provide insights into the test execution process, including successes, failures, and overall test coverage.

## Benefits of TestNG Reporting

- **Visibility:** Clear visibility into test execution status and issues.
- **Debugging:** Helps in identifying and diagnosing problems in test cases.
- **Documentation:** Provides a record of test results for further analysis or auditing.

# Built-in TestNG Reports:

### 1. HTML Report

- **Description:** TestNG generates an HTML report that includes a summary of test execution, detailed results for each test, and logs.
- **Location:** By default, the report is generated in the test-output directory. Is using maven-surefire plugin the reports will be generated in target

### 2. XML Report

- **Description:** TestNG also generates an XML report (testng-results.xml) that provides detailed execution data in XML format.
- **Location:** Typically found in the test-output directory.

### Example of Accessing HTML Report:

- **File Path:** test-output/index.html; target/surefire-reports/index.html; target/surefire-reports/emailable-report.html

### Example of Accessing XML Report:

- **File Path:** test-output/testng-results.xml; target/surefire-reports/testng-results.xml

### Sample HTML Report Features:

- **Summary Tab:** Overview of passed, failed, and skipped tests.
- **Detailed Results:** Clickable links to view individual test case results and logs.

# Best Practices for Reporting:

### 1. Choose the Right Reporting Tool

- **Decision:** Select a reporting tool that fits your project needs, whether it’s the default TestNG reports, ReportNG, Extent Reports, or custom implementations.

### 2. Integrate Reporting into CI/CD Pipelines

- **Automation:** Ensure reports are generated and archived as part of your continuous integration/continuous deployment (CI/CD) process.

### 3. Customize Reports for Clarity

- **Detailing:** Customize reports to include relevant information and make them easy to understand for stakeholders.

### 4. Use Listeners for Enhanced Reporting

- **Flexibility:** Implement custom listeners to track specific events or customize how results are reported.

# Parallel Execution in TestNG:

## What is Parallel Execution?

- **Definition:** Parallel execution allows running multiple test methods or classes simultaneously.
- **Purpose:** To speed up the test execution process by leveraging multi-core processors and improving test suite efficiency.

## Benefits of Parallel Execution

- **Reduced Test Execution Time:** Parallelism minimizes the overall time required for running large test suites.
- **Enhanced Utilization:** Makes better use of available CPU resources by running tests concurrently.

# Configuring Parallel Execution:

## 1. Parallel Execution at the Method Level

- **Description:** Tests are executed in parallel within the same class.
- **Configuration:** Set parallel="methods" in the <suite> tag in the TestNG XML file.

*Example TestNG XML for Method-Level Parallel Execution:*

```xml
<suite name="Suite" parallel="methods" thread-count="5">
    <test name="Test">
        <classes>
            <class name="com.example.MyTests"/>
        </classes>
    </test>
</suite>
```

## 2. Parallel Execution at the Class Level

- **Description:** Tests are executed in parallel across different classes.
- **Configuration:** Set parallel="classes" in the <suite> tag in the TestNG XML file.

*Example TestNG XML for Class-Level Parallel Execution:*

```xml
<suite name="Suite" parallel="classes" thread-count="3">
    <test name="Test">
        <classes>
            <class name="com.example.FirstTest"/>
            <class name="com.example.SecondTest"/>
        </classes>
    </test>
</suite>
```

# Configuring Parallel Execution:

### 3. Parallel Execution at the Test Level

- **Description:** Tests are executed in parallel across different test nodes.
- **Configuration:** Set parallel="tests" in the <suite> tag in the TestNG XML file.

*Example TestNG XML for Test-Level Parallel Execution:*

```xml
<suite name="Suite" parallel="tests" thread-count="4">
    <test name="Test1">
        <classes>
            <class name="com.example.Test1"/>
        </classes>
    </test>
    <test name="Test2">
        <classes>
            <class name="com.example.Test2"/>
        </classes>
    </test>
</suite>
```

# Introduction to TestNG Listeners:

## What are TestNG Listeners?

- **Definition:** Listeners are interfaces provided by TestNG that allow you to listen to and react to events during test execution.
- **Purpose:** To perform custom actions or modifications during test execution, such as logging, reporting, or handling test results.

## Benefits of Using Listeners

- **Customization:** Tailor the behavior of your test suite according to your needs.
- **Flexibility:** Handle different events and take actions like logging, reporting, or cleanup.
- **Extensibility:** Extend TestNG's functionality without modifying the test code itself.

# Commonly Used TestNG Listener Interfaces:

### 1. ITestListener

- **Description:** Provides methods to listen to test start, success, failure, and skip events.
- **Methods:**
  - `onTestStart(ITestResult result)`
  - `onTestSuccess(ITestResult result)`
  - `onTestFailure(ITestResult result)`
  - `onTestSkipped(ITestResult result)`
  - `onStart(ITestContext context)`
  - `onFinish(ITestContext context)`

### 2. IReporter

- **Description:** Allows you to generate custom reports at the end of the test suite execution.
- **Methods:**
  - `generateReport(List<XmlSuite> xmlSuites, List<ISuite> suites, String outputDirectory)`

### 3. ITestResult

- **Description:** Provides detailed information about the result of a test method.
- **Attributes:**
  - `getMethod()`
  - `getThrowable()`
  - `getStatus()`

# Configuring Listeners in TestNG:

## 1. Adding Listeners via XML Configuration

- **Description:** Specify listeners in the TestNG XML file to apply them to your test suite.
- **Example Configuration:**

```xml
<suite name="Suite">
    <listeners>
        <listener class-name="com.example.CustomTestListener"/>
        <listener class-name="com.example.CustomReporter"/>
    </listeners>
    <test name="Test">
        <classes>
            <class name="com.example.MyTests"/>
        </classes>
    </test>
</suite>
```

## 2. Adding Listeners via Annotation

- **Description:** Attach listeners programmatically within test classes using @Listeners.
- **Example Usage:**

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

@Listeners(com.example.CustomTestListener.class)
public class MyTests {

    @Test
    public void testMethod() {
        System.out.println("Executing test method");
    }
}
```

# List of Listeners in TestNG:

| Listener | Description | Key Methods |
|---|---|---|
| `ITestListener` | Listens to test execution events such as test start, success, failure, and skip. | - `onTestStart(ITestResult result)`<br>- `onTestSuccess(ITestResult result)`<br>- `onTestFailure(ITestResult result)`<br>- `onTestSkipped(ITestResult result)`<br>- `onStart(ITestContext context)`<br>- `onFinish(ITestContext context)` |
| `IReporter` | Allows for the generation of custom reports after test execution. | - `generateReport(List<XmlSuite> xmlSuites, List<ISuite> suites, String outputDirectory)` |
| `ITestResult` | Provides detailed information about the result of a test method. | - `getMethod()`<br>- `getThrowable()`<br>- `getStatus()` |
| `IAnnotationTransformer` | Allows for modifying or transforming annotations on test methods before execution. | - `transform(ITestAnnotation annotation, Class testClass, Constructor testConstructor, Method testMethod)` |
| `ITestListener2` | Extended version of `ITestListener` for additional reporting and logging. | - `onTestSuccess(ITestResult result)`<br>- `onTestFailure(ITestResult result)`<br>- `onTestSkipped(ITestResult result)`<br>- `onStart(ITestContext context)`<br>- `onFinish(ITestContext context)`<br>- `onTestFailedButWithinSuccessPercentage(ITestResult result)` |
| `IExecutionListener` | Provides callbacks before and after the execution of the test suite. | - `onExecutionStart()`<br>- `onExecutionFinish()` |
| `ITestContextListener` | Allows for listening to test context events. | - `onTestContextStart(ITestContext context)`<br>- `onTestContextFinish(ITestContext context)` |
| `IClassListener` | Provides callbacks for class level events. | - `onBeforeClass(Class<?> clazz)`<br>- `onAfterClass(Class<?> clazz)` |
| `IConfigurationListener` | Handles configuration methods, useful for setup and teardown operations. | - `beforeConfiguration(ITestResult result)`<br>- `afterConfiguration(ITestResult result)` |

# Use Cases for TestNG Listeners:

### 1. Logging and Reporting

- **Scenario:** Implement custom logging or reporting mechanisms to capture test execution details.
- **Example:** Create a listener that logs test execution details to a file.

### 2. Screenshots on Failure

- **Scenario:** Capture screenshots when tests fail to provide visual evidence of issues.
- **Example:** Use a listener to take screenshots and attach them to reports.

### 3. Test Cleanup

- **Scenario:** Perform cleanup tasks after test execution, such as closing resources or resetting states.
- **Example:** Implement a listener to clean up resources or reset the environment after tests.

### 4. Performance Monitoring

- **Scenario:** Monitor performance metrics during test execution.
- **Example:** Create a listener to record execution times and generate performance reports.

# Best Practices for Using Listeners:

### 1. Keep Listeners Lightweight

- **Focus:** Ensure listeners perform minimal actions to avoid slowing down test execution.

### 2. Use Listeners for Cross-Cutting Concerns

- **Scope:** Utilize listeners for tasks like logging, reporting, and clean-up that span multiple tests or test suites.

### 3. Test Listener Code Thoroughly

- **Validation:** Test and validate listener code to ensure it functions correctly across different scenarios and doesn’t introduce issues.

### 4. Document Listener Behavior

- **Clarity:** Document what each listener does and how it affects test execution to facilitate maintenance and understanding.

# TestNG and Selenium WebDriver Integration:

## What is Selenium?

- **Definition:** A suite of tools for automating web browsers, including Selenium WebDriver, Selenium Grid, and Selenium IDE.
- **Purpose:** To enable automated interaction with web applications and perform functional testing.

## Benefits of Integration

- **Enhanced Testing Capabilities:** Combine TestNG's testing features with Selenium's browser automation.
- **Efficient Test Management:** Utilize TestNG's features like data-driven tests, parallel execution, and custom listeners in conjunction with Selenium.

# Best Practices for TestNG and Selenium Integration:

## 1. Manage WebDriver Instances Properly

- **Strategy:** Use @BeforeClass and @AfterClass to manage WebDriver instances to avoid conflicts and ensure proper resource cleanup.

## 2. Organize Tests Effectively

- **Organization:** Structure tests using TestNG annotations and groups to manage complex test scenarios and ensure maintainability.

## 3. Utilize Page Object Model (POM)

- **Design:** Implement the Page Object Model to create a clear separation between test logic and UI interactions, making tests more readable and maintainable.

## 4. Handle Synchronization Issues

- **Synchronization:** Use explicit waits to handle dynamic web elements and avoid flaky tests due to timing issues.

## 5. Configure TestNG Reports and Listeners

- **Reporting:** Leverage TestNG's reporting capabilities and listeners to capture detailed test execution results and integrate with Selenium for custom logging and reporting.

# TestNG Best Practices:

1. **Organize Tests Logically**

   - **Structure:** Group tests by functionality or feature using TestNG's grouping feature.
   - **Example:** Use groups attribute to categorize tests into logical groups for better organization.

2. **Use Data Providers for Data-Driven Testing**

   - **Advantage:** @DataProvider helps in running the same test method with different sets of data.
   - **Best Practice:** Keep data sets manageable and use meaningful test data.

3. **Leverage TestNG Annotations Wisely**

   - **Annotations:**
     - @BeforeClass and @AfterClass: Use for setup/teardown that should occur once per test class.
     - @BeforeMethod and @AfterMethod: Use for setup/teardown before and after each test method.
     - @Test: Use for defining test methods with attributes for dependency, priority, and expected exceptions.

4. **Implement Parallel Execution Carefully**

   - **Configuration:** Use the parallel attribute in your TestNG XML file for parallel test execution.
   - **Considerations:** Ensure thread safety in tests and avoid shared state or resources between tests.

5. **Utilize TestNG Listeners for Customization**

   - **Listeners:** Implement ITestListener, IReporter, and other listeners for custom logging, reporting, and handling test results.
   - **Example:** Use listeners to capture screenshots on failure or generate custom reports.

# Tips for Writing Efficient Tests:

1. **Keep Tests Independent and Atomic**

- **Independence:** Ensure tests do not rely on each other. Each test should be self-contained.
- **Atomicity:** Each test should focus on a single aspect of the functionality.

2. **Optimize Test Execution Time**

- **Strategies:** Use parallel execution and avoid unnecessary delays in tests.
- **Example:** Use implicit waits sparingly and prefer explicit waits to handle dynamic web elements.

3. **Manage Test Data Effectively**

- **Data Management:** Use @DataProvider or external data sources for data-driven tests.
- **Best Practice:** Keep test data in external files (e.g., Excel, CSV) or databases for flexibility and easy updates.

4. **Handle Test Failures Gracefully**

- **Error Handling:** Use assertions to validate expected outcomes and capture detailed error messages on failures.
- **Best Practice:** Implement retry logic for flaky tests using IRetryAnalyzer.

5. **Use Assertions Wisely**

- **Assertions:** Use appropriate assertions (assertTrue, assertEquals, assertNotNull, etc.) to verify conditions.
- **Best Practice:** Assert on key elements and states, and avoid over-asserting to keep tests clear and focused.

# Maintaining Test Suites:

1. **Regularly Review and Refactor Tests**

   - **Review:** Periodically review test cases to ensure they are still relevant and effective.
   - **Refactor:** Refactor test code to improve readability, maintainability, and remove redundant tests.

2. **Document Tests and Test Plans**

   - **Documentation:** Maintain clear documentation for test cases, including their purpose, dependencies, and expected outcomes.
   - **Best Practice:** Use descriptive names for test methods and groups to enhance readability.

3. **Integrate with CI/CD Pipelines**

   - **Integration:** Configure TestNG with continuous integration/continuous deployment (CI/CD) tools like Jenkins, GitLab CI, or Travis CI.
   - **Best Practice:** Automate test execution as part of the build pipeline and review test results regularly.

4. **Monitor Test Results and Metrics**

   - **Monitoring:** Use TestNG's built-in reports or custom reports to monitor test results and metrics.
   - **Best Practice:** Track test execution trends, failures, and success rates to identify and address issues promptly.

5. **Handle Test Dependencies with Care**

   - **Dependencies:** Use dependsOnMethods or dependsOnGroups to manage test dependencies, but avoid creating complex dependency chains.
   - **Best Practice:** Minimize test dependencies to keep tests modular and reliable