Java Programming and Selenium Web Automation

Homework – Week 8 – TestNG and JUnit

# Compare Junit vs TestNG.

|  |  |
| --- | --- |
| **TestNG** | **JUnit** |
| **Open-source** framework for both **unit** and **end-to-end** testing | **Open-source** **unit** testing framework for Java |
| **Allows** **parallel testing** | **Does not support** **parallel** testing |
| Test cases **can** be **grouped** together | **Cannot** **group** tests |
| **In-built HTML reports** (can also use Maven to generate reports) | **No in-built HTML reporting** (use Maven to generate reports) |
| **Special annotation** for **data driven** testing | More **difficult** to execute **data driven** testing |
| **Supports** **dependency** testing | **Doesn’t support** **dependency** testing |

# What is TestNG?

**TestNG**is an **automation testing framework** for all types of testing such as **unit**, **functional**, **end-to-end** and **integration**.

It is **written** in **Java** programming language and needs a **JDK 5** or **higher** to work. The “**NG**” stands for **“Next Generation**” because it is a framework developed on the lines of Junit and NUnit but with **more** **features**.

With TestNG, you can **generate** **HTML** **reports** easily and come to know how many test cases **passed**, **failed,** or **skipped**. TestNG has more annotations than JUnit and they are much easier to understand and use.

Other testing tools like TestNG are **JUnit**, **NUnit**, **HtmlUnit**.

# Why is TestNG popular?

**TestNG** is a **popular** because it offers **several features** that make it **easier** to **write** and **manage** tests.

* **Parallel testing**:
* **Run** tests **concurrently**.
  + **Reduce** test **time**.
* **Support** for a **wide range** of **test types**:
* **Unit**, **Integration**, **Functional**, **End-to-End**, etc.
  + Write **specific use-case** tests.
* **Easy to write** and **manage** tests:
* **Simple** and **intuitive** **annotations** to define tests and groups.
* **Better reporting**:
* **Detailed** **reports** (passed, failed, and skipped tests).
  + **Identify** **issues** and **fix** them quickly.
* **Integration** with **build tools**:
* Like **Maven** and **Gradle**
  + **Automate** the testing process.
  + **Continuous Integration**/**Continuous Delivery** (**CI/CD**) **pipeline**.

# How is TestNG is used?

Here are the **typical steps** involved in using TestNG:

1. **Set up** a TestNG **project**:
   * **Create** a new Java **project**.
   * **Add** the TestNG **library** to the project's build path.
2. **Create** test **classes**:

* **Create** classes that contain test **methods**.
* **Annotate** with TestNG **annotations** (@Test, @BeforeSuite, @AfterSuite, @BeforeClass, @AfterClass, @BeforeMethod, @AfterMethod, and @DataProvider)

1. **Configure** the test **suite**:

* **Create** an XML **configuration** file that
* **Specifiy** the tests, methods, and any test parameters or data.

1. **Run** the test **suite**:

* From **command line** or from **within an IDE**.

1. **Analyze** the test **results**:

* **Information** on test cases: status, execution time, failures, or errors.

# Explain Maven Dependencies

**Maven Dependencies** are external libraries or modules that a Maven project uses to execute its functionality.

They are **specified** in the project's **pom.xml file**, and Maven uses them to **download** the **required libraries** from a central or a local repository.

**→** Easily **manage** their project's **dependencies**.

→ **Ensures** project runs **correctly**, **without** the need to **manually** download and configure each library.

# Explain TestNG Annotation

**TestNG Annotations** are used to provide **metadata** and **instructions** to TestNG on **how to run tests**.

They help in the organization and execution of test cases.

They are placed **before methods** or **classes** and provide a way to **define** the order and **behavior** of tests, groups, data providers, setup, teardown methods, etc.

Some **built-in** annotations are:

|  |  |
| --- | --- |
| @Test  @BeforeSuite  @AfterSuite  @BeforeTest  @AfterTest | @BeforeGroups  @AfterGroups  @BeforeClass  @AfterClass  @BeforeMethod  @AfterMethod |

# Explain Before Test and Before Method.

**@BeforeTest** and **@BeforeMethod** are annotations used to **define methods** that should be **executed before** each **test case** or **test method**, respectively.

**@BeforeTest:**

* **Used** to define a method to be **executed** **before any test case** belonging to the same <test> tag in the TestNG XML file.
* **To set up** any global configurations or resources that are **needed by the tests**.

**@BeforeMethod:**

* **Used** to define a method to be executed **before each test method** within a class.
* **To set up** any test data or objects that are **needed for the test method**.

**Both methods** can be used to **set up** the **test** **environment** and **ensure** that each test case or test method is executed in a **predictable** and **consistent** manner.

# Explain After Test and After Method.

**@AfterTest** and **@AfterMethod** are annotation methods used to **specify the actions** that should be taken **after** the **execution** of a **test case** or a **test method**.

**@AfterTest**

* **Used** to specify a block of code to be **executed** **after all the test methods** belonging to the same <test> tag have been **executed**.
* **To perform** cleanup operations **after** the **execution** of **all the test methods** in a test case.

**@AfterMethod**

* **Used** to specify a block of code that will be executed **after the execution of each test method**.
* **To perform** cleanup operations **after** the **execution** of **each test method**.
* **It ensures** that the **state of the system** is the same for each test method, and that there are **no conflicts** between test methods.

**Both methods** are useful for **maintaining the integrity** of the **test environment**, **ensuring** that the **test cases** are **independent** of each other, and that the **system** is in a **consistent** **state** **before** and **after** each test.

# Explain @Test.

**@Test annotation** is used to **indicate** that a method is a **test** **method** that needs to be executed.

**TestNG** **runs** the method annotated with **@Test** as a **separate test case**.

It can be **used** with various attributes to **configure** how the test method should be executed.

* **Example**:
  + - * **priority** attribute: to **specify** the **order** in which **test methods** should be executed.
      * **dataProvider** attribute: to **provide** **data** to the **test** **method**.

# Explain Setup methods.

**Setup methods** are used to **perform** the necessary **setup** or **preparation** **before** **running** a **test** **method** in the class.

These methods are marked with the **@Before** annotation.

They are often used to **initialize variables**, **objects** or **resources** that are required by the test method.

* **Example**:
  + - * if a test **method** **requires** **access** to a **database**, the setup method can **establish** the database **connection** and **perform** **initialization**.

**Setup methods** ensures the **test code** can be **organized**:

* **Easier** to **read** and **maintain**.
* **Reduce** code **duplication** by **centralizing** the common **setup** **logic** in **one** **place**.

# Write a simple example.

**Example:**

public class **LoginTest** {

@BeforeClass

public void **setupClass**() {

// code to set up the test environment for the entire test class

}

@BeforeMethod

public void **setupMethod**() {

// code to set up the test environment for each individual test method

}

@Test

public void **testValidLogin**() {

// code to test a valid login scenario

}

@Test

public void **testInvalidLogin**() {

// code to test an invalid login scenario

}

}

# What’s TestNG Suite?

**TestNG Suite** is a **collection** of **TestNG test cases** that can be executed **together**.

It is a way to **group** **tests**, **define** **dependencies** between them, and **configure** **test** execution **parameters** for the **entire** **suite**.

A TestNG Suite file is an **XML file** that contains all the **information** required to **run** a **suite**, including the **test classes**, **test methods**, and **parameters**.

It helps **organize** and **execute** **tests** in a more **efficient** and **streamlined** way.

# How to parameterize TestNG Tests?

Using **@Parameters** annotation, test can be **parameterized**.

First, the **parameters** need to be **defined** in the **testng.xml file** or in a **separate** **properties** **file**.

Then, the **parameter** names are **specified** in the @Parameters annotation in the test method.

**Example:**

***Define*** *the usernames and passwords in a* ***separate******file***:

# credentials.properties

username1=alice

password1=passw0rd

username2=bob

password2=abc123

*In* ***testng.xml file*:**

<suite>

<test>

<parameter name="username" value="${username1}" />

<parameter name="password" value="${password1}" />

<classes>

<class name="com.example.LoginTest" />

</classes>

</test>

</suite>

*In* ***Java*:**

public class **LoginTest** {

@Test

@Parameters({"username", "password"})

public void **testLogin**(String username, String password) {

// Do login test with the specified username and password

}

}

# TestNG Pros and Cons

|  |  |
| --- | --- |
| Pros | Cons |
| **Comprehensive** framework for **testing** | **Steep** learning curve for **beginners** |
| **Supports** multiple programming **languages** | **Requires** **additional** **setup** and **configuration** |
| **Provides** rich set of **annotations** | **Limited** **integration** with IDEs |
| **Allows** for **parallel** test **execution** | **Relatively** **slower** **execution** speed |
| **Provides** **support** for **data-driven** **testing** | **Documentation** can be **incomplete** or **outdated** |
| **Enables** **configuration** of test **suite** | **Difficulty** in **debugging** **complex** test cases |

# How to generate TestNG Report, show steps

1. **Add** TestNG **dependency** in project's pom.xml file.
2. **Create** TestNG **test classes** with all the **required** annotations and tests.
3. **Run** TestNG **tests** using any **preferred method** (command line, IDE, or Maven).
4. TestNG **generates** an **HTML report** by **default**.

Usually located in the "**test-output**" folder in project directory.

1. **Open** the HTML report in any **web browser** to **view** the **detailed** **results**.

# How to generate code coverage Report, show steps

Use a **popular tool** called **JaCoCo**.

1. **Add** the **JaCoCo** **plugin** to your project’s pom.xml file:

<build>

<plugins>

<plugin>

<groupId>org.jacoco</groupId>

<artifactId>jacoco-maven-plugin</artifactId>

<version>0.8.7</version>

<executions>

<execution>

<goals>

<goal>prepare-agent</goal>

</goals>

</execution>

<execution>

<id>report</id>

<phase>prepare-package</phase>

<goals>

<goal>report</goal>

</goals>

</execution>

</executions>

</plugin>

</plugins>

</build>

1. **Run** tests with **JaCoCo** coverage.