TestNG

1. Use of listeners in testNG, Different Types of listeners in Selenium testNG Automation Framework?

Ans. **Modify**/**Listen** the **behaviour** of **test execution**

- onTestStart, success, failure, skipped, start, finish (ITestListeners interface)
- taking screenshot on test failure
- IAnnotationTransformer (Modify Test priority/retry failed test cases/Runtime)

2. How to run a single test multiple times in testNG?

Ans. **Invocation count** at @Test annotation

3. @**BeforeSuite** → Runs once before all tests in the suite.

   @**BeforeTest** → Runs once before any test cases in the <test> tag of testng.xml.

   @**BeforeClass** → Runs once before the first method in a test class.

   @**BeforeMethod** → Runs before each @**Test method** in a test class.

   @**BeforeGroups** → Runs before any test method in the **specified group**(s).

   @**Test** → Marks a method as a TestNG test case.

   @**DataProvider** → Supplies test data to test methods.

   @**Parameters** → Passes parameters from **testng.xml** to test methods.

   @**Listeners** → Used to define custom listeners for test execution monitoring.

4. How can we perform data driven testing in testNG?

Ans. We can perform data driven test by-

- @**DataProvider** annotation which has return type **Object[][] Array**
- Using **Excel (Apache POI), @DataProvider** reads from excel sheet
- **CSV/JSON/XML**
- @**Paramters** from textNG.xml **<parameter name="" value="">**

5. How to read and write data from excel in testNG?

Ans. Apache POI, XSSFWorkbook, XSSFSheet, **class ExcelReader**, File InputStream.

6. How to exclude a seperate test method while running multiple tests?
7. How to Implement Parallel test running?

Ans. use the <**exclude**> tag the testng.xml file inside and use parallel attribute in <**suite** >tag with **thread-count**

```
<suite name="MyTestSuite" parallel="tests" thread-count="2">
  <test name="MyTest">
    <classes>
      <class name="TestExample">
        <methods>
          <exclude name="testMethod2"/>
        </methods>
      </class>
    </classes>
  </test>
</suite>
```

8. How to set Priority in testNG?

Ans. **@Test(priority = -1)**

9. Assertions – difference **between assert and verify**

Ans.

Feature	**Assert (Hard Assertion)**	**Verify (Soft Assertion)**
Definition	Stops test execution if assertion fails	Continues execution even if assertion fails
Class Used	org.testng.Assert	org.testng.asserts.SoftAssert
Impact on Execution	If assertion fails, **the test method stops immediately**	All assertions are executed, and **failures are reported at the end**
Use Case	**Critical** validations (e.g., login success)	**Non-critical checks** (e.g., UI elements)

10. Different **Waits**

Ans.

- **Implicit Wait** – **Global wait** applied to all elements. Waits for a fixed time before throwing an exception.

- **Explicit Wait** – Waits for a **specific condition** (like visibility, clickability).

- **Fluent Wait** – Similar to Explicit Wait but **checks periodically** and can **ignore exceptions**.

Use Implicit for general waits, Explicit for condition-based waits, and Fluent for highly dynamic elements.

11. Rerun failed tests in testNG

Ans.

Method	Description	Best Use Case
IRetryAnalyzer	Retries failed tests automatically **within the same run**	When tests fail due to temporary issues (e.g., network lag, timeout)
testng-failed.xml	Manually re-executes only failed tests in a separate run	When rerunning failed tests **after execution**

12. Page Factory class

Ans.

✅ **Page Factory initializes elements lazily & optimizes POM.**
✅ Uses **@FindBy annotations** instead of driver.findElement().
✅ **PageFactory.initElements(driver, this);** initializes elements automatically.

🔥 **Makes Selenium automation faster & structured!**

13. Faker Library

Ans.

Faker faker = new Faker();

System.out.println(faker.name().fullName());

Selenium 4.25.0

Java version 21

CUCUMBER

1. Why we use BDD?

Ans. **BDD (Behavior-Driven Development)** is an approach that enhances collaboration between **developers, testers, and business stakeholders** by using **plain English** to define test scenarios. It extends **TDD (Test-Driven Development)** with a focus on **business requirements**. **GHERKIN syntax – given when then**

2. How can we make sure that step definitions were written for the steps in feature file?

Ans. executing your test and verify output in the console

- **Run the test → Cucumber suggests missing step definitions in the console.**
- **Implement the suggested steps in a Step Definition class.**
- **Re-run the test → Check if all steps execute without errors in the console.**

3. Ideal way of Declaring Asserts (in **Steps**/Pages)?

Ans.

- Step definitions handle **test verification**, while **Page Objects only interact with UI elements**.

- Keeping assertions in steps ensures **separation of concerns** (POM for actions, Steps for verification).

4. Why Use Page Object Model (POM)?

Ans. POM improves test maintainability and reusability.

- Separates **UI interactions** (Page Objects) from **test logic** (Step Definitions).
- Reduces duplicate code by centralizing element locators and actions.
- Easier maintenance when UI changes – **update only in the Page Class.**

5. Why Use Hooks in Cucumber?

Ans. **Hooks (@Before & @After) manage pre- and post-test setup automatically.**

- **@Before** – Runs **before** every scenario (e.g., launching the browser, setting up data).
- **@After** – Runs **after** every scenario (e.g., closing browser, cleaning up data)

Asserts in Steps – Keep assertions in step definitions to separate test logic from UI interactions.

POM Usage – Helps maintain test structure, improves reusability, and makes updates easier.

Hooks in Cucumber – Automate setup/teardown, improving test consistency.

6. How to generate reports

Ans. **Use extent.properties and add ExtentCucumberAdapter in CucumberOptions Plugin**

7. Screenshot

Ans. Use **TakesScreenshot Interface** and attach it to failed scenarios in **@After** hook.

8. Utility files

Ans. Use **ConfigReader to fetch properties and DriverManager to handle WebDriver.**

9. Scenario/Scenario outline

Ans. Scenario runs **once**, Scenario Outline runs **multiple times with different Input via Examples: table**

10. How can we pass the data in Examples: block to Scenario outline (ans - using angular brackets)

Ans. **Use < > Angular Brackets**

11. How can we declare input data in scenario steps

Ans. Use **hardcoded values in steps OR < > placeholders in Scenario Outline**.

12. How to rerun failed tests in cucumber?

Ans.

- Use **dryRun = true to check missing step definitions before execution**.
- Enable **rerun:target/failed_scenarios.txt in plugin** to store failed tests.
- Create a **separate test runner with @target/failed_scenarios.txt to rerun failures.**

REST ASSURED API

1. Various Response Status Codes

Ans. 200 - Ok, 201 – Created, 204 – No content,

300 – redirection codes, 301 – moved permanently, 302 – temporarily moved,

400 – Bad Request, 401 – Unauthorized Authentication, 403 – Forbidden,

404 – Not Found, 429 – too many requests (client side)

500 – Internal Api server error, 502 – Bad Gateway, 503 – service unavailable,

504 – gateway timeout, 505 – https version not supported.

2. given(), when(), then()

Ans.

given() → **Prepares request** (headers, params, body). given().header("Content-Type", "application/json")

when() → **Makes the actual Api request** (GET, POST, PUT, DELETE). when().get("/users/2")

then() → **Validates response** (status code, body, headers). then().statusCode(200)

3. Request Payloads (Request Body)

Ans.

A request payload (also called request body) is the data sent in an API request, typically in JSON format for POST, PUT, and PATCH requests.

Types of Request Payloads

Raw JSON String (directly in body())

Using a HashMap (automatically converted to JSON)

Using a POJO (Plain Old Java Object) with Jackson or Gson

Using an External JSON File

4. Put and Patch

Ans.

PUT → Full update (Replaces entire resource).

PATCH → Partial update (Only modifies specified fields).

Use PATCH when updating a few fields, PUT for complete replacements!

5. Key Value Pairs

Ans.

Key-Value pairs represent data in JSON, headers, query params, and form data.

They store request & response data in structured formats.

Common use cases: JSON payloads, authentication headers, API parameters.

6. Headers, why we use headers?

Ans.

Headers provide additional information (**metadata**) about the **request or response**, helping the server and client communicate effectively.

Why Are Headers Used?

Authentication & Security → (Authorization: Bearer token123)

Data Format Specification → (Content-Type: application/json)

Caching Control → (Cache-Control: no-cache)

Language & Encoding → (Accept-Language: en-US)

7. Query parameters, path parameters

Ans. Both used to pass information in the api requests, but ::

Parameter Type	Definition	Example	Use Case
Query Parameter	Passed after ? in the URL	GET/users?page=2&sort=asc	Filters, pagination, sorting
Path Parameter	Part of the URL itself	GET/users/123	Fetching a specific resource

8. **POJO**

Ans.

POJO (Plain Old Java Object) is a simple Java class used to map API request & response data.

Helps in serialization (Java → JSON) & deserialization (JSON → Java).

Improves test maintainability, avoiding hardcoded JSON.

- Use pojo class for request payload
- Use pojo in rest assured for api request
- Convert JSON response to pojo

9. Bearer -tokens

Ans.

Bearer tokens authenticate API requests via the Authorization header.

The token is usually obtained from a login API and used for subsequent requests.

Bearer tokens follow OAuth 2.0 authentication and improve security.

10. How we validate the response (jsonPath, shema, POJO)?

Ans.

JsonPath → Extract specific data from JSON response

Schema Validation → Check if response follows expected structure

POJO Deserialization → Convert JSON response to Java Object for verification

11. Uses of getter setter method in POJO class?

Ans.

Getters **retrieve** values, setters **modify** values while keeping fields private (**Encapsulation**).

They enable JSON serialization & deserialization in API automation (RestAssured).

Using getters & setters ensures controlled access and data validation.

Mobile Automation

1. Difference between mobile and web testing?

Ans.

✅ **Mobile testing** focuses on **native, hybrid, or web apps** on **Android/iOS**.
✅ **Web testing** ensures **websites/web apps** work across **browsers & devices**.
✅ **Challenges in mobile testing**: Network variations, gestures, battery usage, OS fragmentation.
✅ **Automation tools**: **Appium** for mobile, **Selenium/Playwright** for web.
✅ **Different testing strategies** are needed for mobile & web testing!

2. Different types of mobile apps?

Ans.

* Native apps (apps built specifically for a mobile operating system).

* Hybrid apps (apps that combine web and native technologies).

* web apps (websites accessed through a mobile browser/ no installation).

3. What is Appium?

Ans. Open-source tool used for mobile automation.

supports both iOS and Android platforms(use the same or similar test script)

Appium uses the WebDriver protocol

Appium inspector is a gui interface

4. Simulator/Emulator?

Ans.

- **Simulator** → **Mimics** OS behavior but does not simulate hardware (e.g., Xcode Simulator). Ios, faster, ui testing.
- **Emulator** → **Replicates** both hardware & software, allowing full testing (e.g., Android Studio, Genymotion). Android , **Full testing** (UI + hardware interactions)
- Emulators are more powerful, but simulators are faster for UI testing.
- Choose based on testing needs!

5. Capabilities, why we use them?

Ans.

Capabilities are key-value pairs used to configure the test environment for automation tools like Appium & Selenium.

Specify Target Device & Platform → Example: platformName, deviceName, browserName.

Control App Behavior → Example: appActivity, appPackage for Android apps.

Enable Testing Features → Example: noReset (to retain app state), **automationName (to define automation engine).**

Cross-Browser & Cross-Device Testing → Helps run tests on different environments seamlessly.

6. 6. Gesture - tap, long tap, scroll, swipe
7. 7. Multiple gestures - zoom in/out, how we can do multiple gestures 2 sequnce pointers are used

Ans.

📌 Single Touch Gestures:

✓ **Tap** → Single quick touch (e.g., clicking a button).

✓ **Long Tap** (Press & Hold) → Touch and hold (e.g., selecting text).

✓ **Scroll** → Move content vertically (e.g., reading a long page).

✓ **Swipe** → Quick slide in any direction (e.g., swiping images).

📌 Multiple Touch Gestures:

✓ **Zoom** In/Out → Performed using **two** fingers (pinch open/close).

✓ How? → **Two sequence pointers are used to track multiple touches in automation.**

🔥 Used in Appium with TouchAction or MultiTouchAction for automation!

8. Locators used

Ans.

Common Locators (Selenium & Appium):

ID → By.id("elementID") (Unique & fast)

Name → By.name("elementName")

Class Name → By.className("className")

Tag Name → By.tagName("button") (For web)

Link Text → By.linkText("Full Link Text")

Partial Link Text → By.partialLinkText("Partial Text")

CSS Selector → By.cssSelector(".class #id") (Web only)

XPath → By.xpath("//div[@class='example']")

Android-Specific Locators (Appium):

Accessibility ID → driver.findElementByAccessibilityId("Submit")

UIAutomator Selector → driver.findElementByAndroidUIAutomator("new UiSelector().text(\"OK\")")

XPath → //android.widget.Button[@text='OK']

XPath Axis - following-sibling:::

Used to locate elements that share the same parent and come after a specific element.

Example: Select the next div after a specific span:

//span[text()='Username']/following-sibling::input

XPath Axes like following-sibling::, parent::, preceding-sibling:: are powerful for dynamic elements!

Manual testing is done to make sure ui components and design is implemented in a user friendly

1. What is a Class and object?

✅ A **class** is a blueprint for creating objects. It contains **variables (fields) and methods (functions). Object is an instance of a class**

✓ **Objects store data & perform actions.**
✓ Created using the new keyword.
✓ Multiple objects of the same class can exist, each with its own state.

2. What is an Interface?

✅ An **interface** is a contract that defines methods **without implementation.** A class must implement all methods of an interface.

3. Static Variables & Methods

✓ **Static variable** → One copy shared across all objects, stored in class memory.
✓ **Static method** → Belongs to the class, can be called without creating an object.
✓ **Instance variables CANNOT be used in static methods.**

4. Abstraction

✅ **Hides implementation, shows only functionality.**
✓ **Abstract class** → Can have both abstract & non-abstract methods.
✓ **Interface** → 100% abstraction (until Java 8), all methods are abstract by default.

5. Checked vs. Unchecked Exceptions

✓ **Checked Exceptions** → **Compile-time**, must be handled (e.g., IOException).
✓ **Unchecked Exceptions** → **Runtime**, optional handling (e.g., NullPointerException).

6. Handling Exceptions

✓ **Checked** → Use try-catch or throw inside a method.
✓ **Unchecked** → Use throws in method signature.

7. Switching to a New Tab (Window Handle)

```
Set<String> handles = driver.getWindowHandles();

for (String handle : handles) {

   driver.switchTo().window(handle);

}
```

```
ChromeOptions options = new ChromeOptions();
```

options.addArguments("--incognito");

WebDriver driver = new ChromeDriver(options);

Static Variables vs. Instance Variables

✓ **Static** → Single copy for all objects.
✓ **Instance** → Each object has its own copy.

🔟 **Key Rules About Classes & Interfaces**

✓ **A class can extend only ONE class** but **can implement multiple interfaces.**
✓ **Interfaces cannot have instance variables or constructors.**
✓ **Abstract classes must have at least one abstract method.**
✓ **All interface variables are public static final.**

✓ **Always prefer ID** (Fastest & most reliable).
✓ **Use CSS Selector before XPath** (CSS is faster).
✓ **XPath should be the last option** (Slower, but useful for dynamic elements).

✓ **BasePage reduces redundancy** by handling common methods.
✓ **BasePage constructor initializes WebDriver**, avoiding duplicate code.
✓ **All page classes extend BasePage**, making test scripts cleaner & reusable.