
Abstraction – Theory (For Explanation)

Definition:
Abstraction is a fundamental concept of Object-Oriented Programming that focuses on **showing only the essential features to the user** while hiding the internal implementation details. In other words, it tells the user **“what to do”** without exposing **“how it is done.”**

Implementation in Your Project:
In your automation project, abstraction is implemented in classes like **DriverFactory** and **WaitUtils**.

- In DriverFactory, test scripts only need to call methods like initDriver() and getDriver(). They **don’t need to know** how the WebDriver is configured with ChromeOptions, thread management, or SSL bypass.
- In WaitUtils, methods like waitForVisibility() or waitForClickability() abstract the internal complexity of Selenium waits and expected conditions.

Purpose:

---

1. Simplifies code usage for testers or developers.
2. Improves maintainability by centralizing logic in one place.
3. Promotes code reusability across multiple test cases.
4. Hides unnecessary details and keeps the test scripts clean and readable.

Advantages:

- Cleaner, readable, and maintainable code.
- Reduces code duplication.
- Internal changes do not affect the test scripts.
- Enhances scalability for large automation frameworks.

Disadvantages:

- Initial setup may require more planning and design.
- Excessive abstraction can make debugging slightly harder if overused.

Code Snippet – Real-Time Example from Your Project

Example 1: DriverFactory Abstraction

```
// Abstraction hides the complex WebDriver setup

DriverFactory.initDriver();
```

```
WebDriver driver = DriverFactory.getDriver(); // Only need to know this
```

Internally, initDriver() handles all ChromeOptions, thread safety, and WebDriverManager setup. The test script **doesn't care about these details**.

Encapsulation

Definition:
Encapsulation is a principle of object-oriented programming where the internal details (fields) of a class are hidden from outside access, and interaction is allowed only through public methods (getters/setters). It helps to control how the data is accessed and modified.

Purpose:

- To protect sensitive data from unintended modification.
- To provide a controlled interface to interact with class data.
- To improve code maintainability and readability.
- To reduce errors by restricting direct access to internal class fields.

Advantages:

1. **Data Security:** Internal variables cannot be accessed or modified directly from outside the class.
2. **Code Flexibility:** You can change the internal implementation without affecting external code.
3. **Ease of Maintenance:** Errors are reduced because all data manipulation goes through controlled methods.
4. **Reusability:** Encapsulated classes can be used safely in different parts of the project.

Real-Time Example in Your Project:
In your project, ConfigReader is a perfect example of encapsulation. The properties object is private, and you access values via public methods like getProperty(), getBrowser(), or getAppUrl(). This prevents direct modification of config data and ensures only valid access.

Implementation Code Snippet:

```
package utilities;


import java.io.FileInputStream;
import java.io.IOException;
import java.util.Properties;

public class ConfigReader {
    // Private variable - encapsulated
    private static Properties properties;
    private static final String CONFIG_FILE_PATH = "src/test/resources/data/config.properties";

    // Static block to load properties
    static {
        try (FileInputStream fileInputStream = new FileInputStream(CONFIG_FILE_PATH)) {
            properties = new Properties();
            properties.load(fileInputStream);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    // Public getter methods - controlled access
    public static String getProperty(String key) {
        return properties.getProperty(key);
    }

    public static String getBrowser() {
        return getProperty("browser");
    }

    public static String getAppUrl() {
        return getProperty("app.url");
    }
}
```

How it’s used in the project:

```
String browser = ConfigReader.getBrowser();  // Accessing browser property
String url = ConfigReader.getAppUrl();       // Accessing application URL
```

Here, the test scripts never touch the properties object directly—they only use the getter methods. This is encapsulation in action.

Polymorphism

Definition:
Polymorphism is a core concept of object-oriented programming where a single object, method, or operator can take multiple forms. In simpler words, it allows objects of different classes to be treated as objects of a common parent class, or a method to behave differently based on input or context.

Purpose:

- To allow flexibility and scalability in the code.
- To enable code reuse by designing methods that can work with multiple types of objects.
- To reduce complexity by providing a unified interface to different functionalities.
- To support dynamic behavior during runtime (runtime polymorphism) or compile-time decision making (compile-time polymorphism).

Advantages:

1. **Flexibility:** Methods or objects can operate in multiple ways depending on context.
2. **Code Reusability:** Same method name can be used in different classes with different functionality.
3. **Maintainability:** Reduces the need for multiple methods with different names.
4. **Extensibility:** New functionality can be added with minimal changes to existing code.

Real-Time Example in Your Project:
In your project, DriverFactory demonstrates polymorphism. You create different WebDriver instances (ChromeDriver, EdgeDriver) but treat them all as WebDriver type. This is runtime polymorphism because the exact driver type is decided at runtime based on the browser parameter.

Implementation Code Snippet:

```
package support;


import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import org.openqa.selenium.edge.EdgeDriver;


public class DriverFactory {


   public static WebDriver createDriver(String browser) {
      WebDriver driver;


      if (browser.equalsIgnoreCase("chrome")) {
         driver = new ChromeDriver();
      } else if (browser.equalsIgnoreCase("edge")) {
         driver = new EdgeDriver();
      } else {
         throw new IllegalArgumentException("Browser not supported");
      }


      return driver; // Returning as WebDriver type (polymorphic behavior)
   }
}
```

How it’s used in the project:

```
WebDriver driver = DriverFactory.createDriver("chrome"); // ChromeDriver instance
WebDriver driver2 = DriverFactory.createDriver("edge");  // EdgeDriver instance
```

Here, both driver and driver2 are **WebDriver type**, but at runtime, they behave differently based on the actual driver class (ChromeDriver or EdgeDriver).

Inheritance

Definition:
Inheritance is an object-oriented programming concept where one class (called the child or subclass) acquires the properties and behaviors (fields and methods) of another class (called the parent or superclass). In simple words, it lets you create a new class based on an existing class, so you can reuse code instead of rewriting it.

Purpose:

- To **promote code reusability** by sharing common code between classes.
- To **establish a hierarchical relationship** between classes.
- To **reduce redundancy** by keeping shared attributes and methods in a parent class.
- To allow **polymorphism**, because child classes can be treated as objects of the parent class.

Advantages:

1. **Code Reusability:** Avoids duplicating code across classes.
2. **Improved Maintainability:** Changes in the parent class automatically reflect in child classes.
3. **Logical Structure:** Helps in organizing code in a hierarchical and meaningful way.
4. **Polymorphism Support:** Child objects can be referenced by parent type.
5. **Extensibility:** New functionality can be added in child classes without modifying parent class.

Real-Time Example in Your Project:
In your project, TestNGTestRunner extends AbstractTestNGCucumberTests. Here, TestNGTestRunner inherits all behavior of the Cucumber TestNG integration provided by the abstract class, so you don’t have to write the whole test running logic yourself. You just add your setup, hooks, or reports.

Implementation Code Snippet:

```
package testRunner;
```

```
import io.cucumber.testng.AbstractTestNGCucumberTests;
import io.cucumber.testng.CucumberOptions;

@CucumberOptions(
    features = {".//src//test//resources//Features"},
    glue = {"stepDefinitions", "hooks"},
    plugin = {"pretty", "html:target/cucumber-reports/cucumber-report.html"},
    monochrome = true
)
public class TestNGTestRunner extends AbstractTestNGCucumberTests {

    // Custom setup for SSL bypass before tests
    @BeforeClass
    public void setupSSL() {
        SSLBypass.disableSSLValidation();
    }

    // After suite, open Allure report
    @AfterSuite
    public void generateReport() {
        AllureReportOpener.openAllureReport();
    }
}
```

Explanation:

- TestNGTestRunner **inherits** from AbstractTestNGCucumberTests.
- It automatically has all methods needed to run Cucumber scenarios with TestNG.
- You **add your own setup and teardown logic** without rewriting the core TestNG-Cucumber integration code.

Selenium Waits

Selenium Waits are mechanisms that pause the execution of a script until a certain condition is met or a maximum timeout occurs. They are crucial because web pages often take time to load elements, and trying to interact with elements before they appear leads to errors.

There are **three main types of waits** in Selenium:

1. Implicit Wait

Definition:
Implicit wait tells Selenium to wait for a certain amount of time when trying to find an element if it's not immediately available. It's applied globally for the WebDriver instance.

Purpose:

- To handle dynamically loaded elements without writing explicit wait code for every element.
- Selenium will repeatedly try to locate the element until the timeout occurs.

Advantages:

- Easy to implement.
- Applies to all elements automatically.

Disadvantages:

- Can slow down tests if elements are found quickly.
- Less flexible because it cannot handle specific conditions like visibility or clickability.

Example in Your Project:
You have implicit wait configured in your ConfigReader:

```
driver.manage().timeouts().implicitlyWait(Duration.ofSeconds(ConfigReader.getImplicitWait()));
```

This means all findElement calls will wait up to the configured time for the element to appear.

2. Explicit Wait

Definition:
Explicit wait is applied to a specific element and waits until a certain condition is met, like visibility, clickability, or presence.

Purpose:

- To wait for specific elements with conditions.
- More precise than implicit wait.

Advantages:

- More control over specific elements.
- Can wait for multiple conditions like visibility, clickability, text, etc.

Disadvantages:

- Needs to be applied individually to each element.
- Adds more code if used for many elements.

Example in Your Project:
You have WaitUtils which wraps explicit waits:

WaitUtils.waitForVisibility(driver, loginButton, ConfigReader.getExplicitWait());

WaitUtils.waitForClickability(driver, loginButton, ConfigReader.getExplicitWait());

This waits until the login button is visible and clickable before interacting with it.

3. Fluent Wait

Definition:
Fluent wait is a type of explicit wait that also allows you to set the polling frequency and ignore specific exceptions while waiting for a condition.

Purpose:

- To handle elements that appear intermittently or slowly.
- Customizes the waiting behavior.

Advantages:

- Can ignore specific exceptions like NoSuchElementException.
- Flexible polling interval instead of checking only at the end.

Disadvantages:

- Slightly more complex to implement than implicit/explicit waits.

Example (Custom, similar to your project style):

```
Wait<WebDriver> wait = new FluentWait<>(driver)
        .withTimeout(Duration.ofSeconds(30))
        .pollingEvery(Duration.ofSeconds(2))
        .ignoring(NoSuchElementException.class);


WebElement loginButton = wait.until(ExpectedConditions.elementToBeClickable(By.id("loginBtn")));
loginButton.click();
```

Here, Selenium checks every 2 seconds for the login button for up to 30 seconds, ignoring NoSuchElementException until it's clickable.

Summary:

- **Implicit Wait:** Global wait for all elements.
- **Explicit Wait:** Wait for a specific condition for a specific element.
- **Fluent Wait:** Customizable explicit wait with polling and exception handling.

Selenium Alerts

Definition:
An **alert** is a pop-up dialog box that appears on the browser and requires user interaction. It can show messages, warnings, or confirmations. Alerts are part of the browser, not the DOM, so you cannot locate them with normal locators. Selenium provides the Alert interface to handle them.

Purpose:

- To handle pop-ups like **JavaScript alerts, confirmations, or prompts**.
- Allows automated tests to interact with alerts without manual intervention.

Advantages:

- Automates pop-up handling.
- Helps validate alert messages in tests.

- Can accept, dismiss, or send input to alerts.

Disadvantages:

- Cannot locate alert with standard locators.
- Must switch to the alert before interacting.

Types of Alerts

1. **Simple Alert:** Shows a message with an OK button.
2. **Confirmation Alert:** Shows OK and Cancel buttons.
3. **Prompt Alert:** Accepts user input in a text box.

Selenium Methods for Alerts

- driver.switchTo().alert() → Switch to alert.
- alert.accept() → Click OK.
- alert.dismiss() → Click Cancel.
- alert.getText() → Get alert message.
- alert.sendKeys("text") → Enter text in prompt.

Example from Your Project (Car Insurance Form Scenario)

Suppose after submitting a form with invalid data, a JavaScript alert pops up with an error message.

```
import org.openqa.selenium.Alert;
import org.openqa.selenium.WebDriver;


WebDriver driver = DriverFactory.getDriver();
driver.get(ConfigReader.getAppUrl());


// Assume this triggers an alert
driver.findElement(By.id("submitBtn")).click();
```

```
// Switch to alert
Alert alert = driver.switchTo().alert();


// Get alert text
String alertMessage = alert.getText();
System.out.println("Alert says: " + alertMessage);


// Accept the alert
alert.accept();
```

Explanation in your words:

- First, I submit the form which triggers a pop-up alert.
- I switch Selenium’s focus to the alert because it’s not part of the normal web page.
- I read the alert message to validate it.
- Finally, I accept it to continue the test.

Selenium Components

Definition:
Selenium is a widely used **open-source tool for automating web application testing** across multiple browsers and platforms. It provides a set of components that serve different purposes in test automation, ensuring that tests are faster, repeatable, and consistent. Each component focuses on specific areas, from simple record-and-playback testing to complex browser interactions and distributed test execution.

Main Components of Selenium:

1. **Selenium WebDriver:**
    - The core and most commonly used component.
    - Enables direct interaction with web browsers using programming languages like Java, Python, or C#.
    - Supports all major browsers (Chrome, Edge, Firefox) and allows you to perform actions such as click, type, navigate, handle alerts, and work with dynamic elements.
    - In your project, `DriverFactory` uses WebDriver to initialize and control the browser for automation.

2. **Selenium IDE (Integrated Development Environment):**
   - A browser plugin for Chrome and Firefox that allows **record-and-playback testing**.
   - Ideal for beginners who want to create simple test scripts without writing code.
   - Limited in terms of advanced test logic and cross-browser support.
3. **Selenium Grid:**
   - Allows **parallel execution of tests** across multiple browsers and machines.
   - Useful for large projects to reduce execution time and validate cross-browser compatibility.
   - In your project, the `testng.xml` file shows how tests can be run on both Chrome and Edge, which is conceptually similar to Grid execution.
4. **Selenium RC (Remote Control) – Legacy Component:**
   - Older version of Selenium that allowed tests to interact with browsers via a server.
   - Deprecated and replaced by WebDriver due to its limitations and slower execution.

Purpose:

- Automate repetitive tasks in web applications.
- Ensure tests are consistent, reliable, and faster than manual testing.
- Support cross-browser and cross-platform testing.
- Integrate with frameworks like TestNG, Cucumber, and reporting tools like Allure or ExtentReports.

Advantages:

- Open-source and free.
- Supports multiple browsers and platforms.
- Allows advanced automation scenarios like alerts, frames, waits, and dynamic element handling.
- Integrates with CI/CD pipelines for continuous testing.

Disadvantages:

- Cannot automate desktop applications.
- Selenium IDE has limited flexibility and advanced testing capabilities.
- Requires programming knowledge for WebDriver-based automation.
- Reporting is not built-in; you must integrate external tools.

Explanation in Your Words:
Selenium has different components to handle testing at various levels. WebDriver is used in my project for browser automation, Selenium IDE is useful for quick test recording, Selenium Grid helps run tests in parallel, and RC is an older method replaced by WebDriver. Together, they make testing faster, reliable, and easier to manage across browsers.

XPath vs Dynamic XPath

Definition:

- **XPath:** XPath is a syntax used to locate elements in an XML or HTML document. In Selenium, we use it to identify web elements on a webpage when ID, name, or class locators are not available or reliable.
- **Dynamic XPath:** Dynamic XPath is an XPath expression that can handle **changing or dynamic web elements**, whose attributes like ID, class, or text may vary every time the page loads.

Key Differences:

Feature	XPath (Static)	Dynamic XPath
Definition	Fixed XPath used to locate elements with stable attributes.	XPath that uses functions or patterns to handle elements with dynamic/changing attributes.
Stability	Works for elements whose attributes do not change.	Works for elements whose attributes, IDs, or positions change dynamically.
Use Case	Ideal for elements with unique IDs or fixed structure.	Ideal for elements generated dynamically or with changing IDs, text, or classes.
Construction	Usually uses absolute or simple relative paths.	Uses contains(), starts-with(), ends-with(), or other XPath functions to locate elements.
Maintenance	Low maintenance if the page structure is stable.	High maintenance if page structure changes significantly; needs careful crafting.

Purpose:

- **XPath:** Quickly locate elements for interaction (click, sendKeys, getText).

- **Dynamic XPath:** Handle web elements whose attributes or positions change during runtime, preventing tests from breaking.

Example (Explained in Human Words):

- **XPath:** Suppose a login button has ID loginBtn. Static XPath would be //button[@id='loginBtn']. Every time the page loads, this works.
- **Dynamic XPath:** Suppose the login button ID changes like loginBtn_123, loginBtn_456. We use Dynamic XPath: //button[starts-with(@id,'loginBtn')] so it works regardless of the number at the end.

Advantages of Dynamic XPath:

- Flexible and handles changing elements.
- Reduces test failures due to dynamic web pages.

Disadvantages:

- Slightly slower than static XPath because Selenium evaluates conditions at runtime.
- Complex expressions can be harder to maintain.

What is pom.xml?

pom.xml stands for **Project Object Model** file. It is the core configuration file of a **Maven project**. Maven is a build automation tool that helps manage dependencies, build lifecycle, plugins, and project information. Every Maven project has one pom.xml at its root.

In simple words, think of pom.xml as the **brain of your project**. It tells Maven what your project is, what it needs, how to build it, and how to run tests.

Purpose of pom.xml:

1. **Dependency Management:**
   It lists all the libraries (like Selenium, TestNG, Cucumber, Allure) your project needs. Maven automatically downloads them and adds to your project classpath.
2. **Build Management:**
   It defines how the project is compiled, tested, and packaged (like a JAR or WAR file).

3. **Plugin Management:**
   It manages plugins like Surefire for running tests, Compiler plugin for Java versions, or Allure for reports.
4. **Project Information:**
   Stores project metadata like groupId, artifactId, version, name, and URL.
5. **Profiles and Environments:**
   You can define multiple profiles for different environments (like QA, Dev, Production) to run tests differently.

Advantages of pom.xml:

- **Automatic dependency handling:** No need to manually download jars.
- **Consistency:** Ensures the same build across different machines.
- **Integration:** Works smoothly with CI/CD tools like Jenkins.
- **Scalability:** Supports multi-module projects easily.

Disadvantages:

- **Learning curve:** Understanding Maven lifecycle and syntax takes time.
- **Complexity:** For large projects with many dependencies and plugins, pom.xml can become long and confusing.

Human-word Example from Your Project:

In your project's pom.xml, you are managing:

- **Dependencies:**
  - Selenium (selenium-java) → for browser automation.
  - Cucumber (cucumber-java, cucumber-testng) → for BDD scenarios.
  - TestNG → for test execution and annotations.
  - Allure & ExtentReports → for reporting.
  - Apache POI → for Excel operations.
  - WebDriverManager → for automatic driver setup.
- **Plugins:**
  - Maven Compiler Plugin → sets Java version (11 or 17).

    - Maven Surefire Plugin → runs TestNG tests defined in testng.xml.
    - Allure Maven Plugin → generates Allure reports automatically.

Example in human words:

"pom.xml in my project is like a control center. It tells Maven which Java version to use, what external libraries to download like Selenium and Cucumber, how to run my tests through TestNG, and even how to generate reports automatically using Allure and ExtentReports."

What is SDLC?

SDLC stands for **Software Development Life Cycle**. It is a **structured process used to design, develop, test, and deploy software** efficiently and with minimal errors. Think of it as a roadmap or blueprint that guides a project from **idea to final product**.

The main idea is to break the software development process into **phases** so that each stage is well-planned, executed, and tested.

Purpose of SDLC:

1. Ensures **quality software** with minimal bugs.
2. Provides a **structured approach** to development.
3. Helps in **better planning and resource management**.
4. Makes it easier to **track progress** and maintain software.
5. Reduces risks of **project failure** or delays.

Phases of SDLC:

1. **Requirement Gathering:**
   Collecting all business and technical requirements from stakeholders. Example: “The login page should accept username and password.”
2. **System Design:**
   Planning the architecture, database design, and UI/UX. Example: Deciding to use Selenium for testing, TestNG for execution, and Excel for test data.
3. **Implementation / Coding:**
   Writing the actual code. Example: Creating Java classes, Cucumber step definitions, and POM structure.

4. **Testing:**
Validating software against requirements. Example: Using Selenium + TestNG + Cucumber to test login functionality.
5. **Deployment:**
Releasing the software to production or staging. Example: Deploying a web application to a QA server.
6. **Maintenance:**
Fixing bugs, updating features, and ensuring smooth operation. Example: Updating a test script when a new browser version is released.

Advantages of SDLC:

- **Better planning and control** of the project.
- **Improves quality** by thorough testing in each phase.
- **Reduces cost** by identifying issues early.
- **Clear documentation** helps new team members understand the project easily.

Disadvantages of SDLC:

- Can be **time-consuming** if phases are strictly followed.
- May become **rigid** in traditional models like Waterfall.
- Changes in requirements may require **rework** in earlier phases.

Human-word Example from Your Project:

"In my project, SDLC starts with understanding the requirements for car insurance forms. Then we design the automation framework using Selenium and Cucumber. Next, we implement step definitions and page classes. After that, we test scenarios using TestNG and generate reports with Allure and ExtentReports. Finally, we maintain scripts to adapt to changes in the website or browser versions."

What is STLC?

STLC stands for **Software Testing Life Cycle**. It is a **structured process followed to test software systematically** and ensure it meets the required quality standards. Unlike SDLC, which covers the whole software development, STLC **focuses only on testing activities**.

The goal of STLC is to **detect defects early, ensure functionality works as expected, and confirm software quality** before release.

Purpose of STLC:

1. Ensures **all features are tested** properly.
2. Helps **identify defects early**, reducing cost and effort.
3. Provides **structured approach for testing** with clear phases.
4. Improves **software reliability and user satisfaction**.
5. Makes testing **repeatable, measurable, and documented**.

Phases of STLC:

1. **Requirement Analysis:**
    - Understand functional and non-functional requirements.
    - Example: For your project, check requirements like valid login, invalid login error, or insurance form validations.
2. **Test Planning:**
    - Decide the **strategy, tools, resources, and timelines** for testing.
    - Example: Using Selenium, TestNG, Cucumber, and Excel for test data.
3. **Test Case Design:**
    - Create **detailed test cases and scenarios**.
    - Example: “Verify that submitting a blank email shows an error message.”
4. **Test Environment Setup:**
    - Prepare **browsers, servers, databases, and tools** needed for testing.
    - Example: Setting up ChromeDriver with WebDriverManager and SSL bypass for your site.
5. **Test Execution:**
    - Execute test cases manually or using automation scripts.
    - Example: Run Selenium scripts through TestNG and generate Allure/ExtentReports.

6. **Defect Reporting:**
   - Log any **bugs found during execution** with details for developers.
   - Example: Invalid phone number field does not show error—log in JIRA/Excel.
7. **Test Closure:**
   - Evaluate **testing completion, coverage, and quality metrics**.
   - Prepare a **test summary report**.
   - Example: Close the testing cycle after verifying all functional flows in car insurance forms.

Advantages of STLC:

- **Ensures better software quality** by detecting defects early.
- **Structured process** makes testing organized and predictable.
- Provides **clear documentation** for future maintenance.
- Helps in **measuring testing effectiveness**.
- **Reduces cost and risk** by preventing production defects.

Disadvantages of STLC:

- Requires **time and effort** for documentation.
- Can be **rigid** if there are frequent requirement changes.
- Needs **experienced testers** to interpret requirements correctly.

Human-word Example from Your Project:

"In my automation project, STLC starts by analyzing the insurance form requirements. Then I plan tests using Selenium and TestNG, design test cases for login, email, and phone validations. I set up ChromeDriver and SSL bypass for the test environment. Next, I execute scripts, capture screenshots, generate reports, and log defects if any. Finally, I close the cycle by ensuring all functionalities are working and reporting the testing metrics."

What is Bug Life Cycle?

The **Bug Life Cycle** is the **process that a defect or bug goes through from its discovery until it is fixed and closed**. It ensures that every issue in the software is properly tracked and resolved.

In other words, it's the **journey of a bug** from being found by a tester to being fixed by a developer and finally confirmed as resolved.

Purpose of Bug Life Cycle:

1. Ensures **defects are tracked systematically**.
2. Helps maintain **quality and stability** of the software.
3. Provides **clear communication between testers and developers**.
4. Allows management to **monitor bug status and progress**.
5. Prevents bugs from being **ignored or left unresolved**.

Phases of Bug Life Cycle:

1. **New / Open:**
    - When a tester finds a defect, it is logged as **New**.
    - Example: Login form allows invalid email but no error message.
2. **Assigned:**
    - The bug is **assigned to a developer** to fix.
3. **Acknowledged / Accepted:**
    - Developer confirms that they understand the bug and will work on it.
4. **In Progress / Fixed:**
    - Developer works on the bug and marks it as **Fixed** in the system.
5. **Retest / Verified:**
    - Tester re-tests the bug to **verify the fix**.
    - If it works fine, the status moves to **Closed**.
6. **Reopened (if needed):**
    - If the bug is **not fixed properly**, it is **reopened** and goes back to the developer.
7. **Closed / Resolved:**
    - Bug is fixed and verified. No further action is needed.

8. **Deferred / Rejected (optional):**
    - Sometimes a bug may be **deferred for a future release** or **rejected** if it's not valid.

Advantages of Bug Life Cycle:

- Provides **clarity on the status of defects**.
- Helps in **prioritizing bugs** for fixing.
- Improves **communication between testers and developers**.
- Tracks bug **history for future reference**.
- Reduces the chance of **unresolved defects going into production**.

Disadvantages of Bug Life Cycle:

- Can become **complicated if there are too many bugs**.
- Requires **discipline in updating status** regularly.
- Needs **proper tools** (like JIRA, Bugzilla) for effective tracking.

Human-word Example from Your Project:

"In my car insurance automation project, if I find that the phone number field accepts letters, I log it as New. The bug is then assigned to a developer. After the developer fixes it, I retest the functionality. If the phone field now works correctly, I mark the bug as Closed. If it still fails, I reopen it. This process ensures every defect is tracked and resolved efficiently."

What is Waterfall Model?

The **Waterfall Model** is one of the **earliest and simplest software development life cycle (SDLC) models**. It is a **linear and sequential approach** where each phase must be **completed before moving to the next phase**. Think of it like a waterfall—once water flows down to the next step, it **cannot go back easily**.

Purpose of Waterfall Model:

1. Provides a **clear and structured approach** to software development.
2. Helps in **planning and scheduling** by defining all phases upfront.

3. Suitable for **projects with well-defined requirements** that are unlikely to change.
4. Makes it easy to **track progress** because each phase has specific deliverables.

Phases of Waterfall Model:

1. **Requirement Analysis:**
   - Gather all **project requirements** from the client.
   - Example: Collect details about car insurance forms, user details, and validation rules.
2. **System Design:**
   - Create **architecture, data flow diagrams, and design documents**.
   - Example: Design page structure for insurance policy form automation.
3. **Implementation / Coding:**
   - Developers **write the actual code** based on the design.
   - Example: Implement Selenium scripts for form fields, validations, and login.
4. **Integration and Testing:**
   - Testers **validate the software for bugs or errors**.
   - Example: Run Cucumber tests for invalid phone numbers and check error messages.
5. **Deployment:**
   - The software is **released to the production environment**.
   - Example: The automation framework is deployed to run on different browsers via TestNG.
6. **Maintenance:**
   - Fix any **post-deployment issues** and make necessary updates.
   - Example: Update scripts if the insurance portal changes its UI or validation rules.

Advantages of Waterfall Model:

- **Simple and easy to understand** because it is linear.

- **Well-defined stages** make it easy to manage and monitor.
- Documentation is **detailed** at every phase.
- Works well for **small projects with stable requirements**.

Disadvantages of Waterfall Model:

- **Not flexible**; difficult to handle requirement changes once a phase is completed.
- **Late testing** can lead to discovering critical bugs very late.
- Not suitable for **complex or long-term projects** with evolving requirements.

Human-word Example from Your Project:

"In my car insurance automation project, I first collected all requirements like user details, car info, and form validations. Then I designed the automation framework structure, wrote Selenium scripts for each page, tested the scripts using Cucumber, deployed it for Chrome and Edge via TestNG, and finally maintained it by updating scripts when the portal changed. Each step followed the next in order, just like the Waterfall Model."

Agile Model – Complete Explanation

Agile Model is a **flexible and iterative software development approach** that focuses on delivering **small, functional pieces of software frequently**. It emphasizes **customer collaboration, adaptability, and continuous improvement** rather than following a strict, linear plan.

Purpose of Agile Model:

1. Deliver **working software quickly** in small increments.
2. **Adapt to changing requirements** anytime during the project.
3. Encourage **collaboration between teams and customers**.
4. Ensure **early detection and resolution of defects**.
5. Improve **team efficiency and project visibility**.

Core Agile Components & Frameworks

1. Scrum:

- The most widely used Agile framework.
- Works in **sprints**, usually 1–4 weeks.
- **Roles in Scrum:**
  - **Product Owner:** Defines features, manages the product backlog.
  - **Scrum Master:** Facilitates the team, removes obstacles.
  - **Development Team:** Builds the product incrementally.
- **Ceremonies in Scrum:**
  - **Sprint Planning:** Decide what work will be done in the sprint.
  - **Daily Stand-up (Daily Scrum):** Short daily meeting to track progress.
  - **Sprint Review:** Demo completed features to stakeholders.
  - **Sprint Retrospective:** Discuss what went well and improvements for next sprint.

2. Kanban:

- Focuses on **visualizing work** and **continuous delivery**.
- Uses a **Kanban board** with columns like *To Do, In Progress, Done*.
- Limits work in progress (WIP) to improve flow.

3. Extreme Programming (XP):

- Emphasizes **technical excellence**, like continuous testing, pair programming, and TDD (Test-Driven Development).

4. Lean:

- Focuses on **reducing waste** and improving efficiency.

Agile Workflow / Phases:

1. **Requirement Gathering / Backlog Creation:**
   - All features are listed in a **product backlog**.
2. **Sprint Planning:**
   - Team selects features for the upcoming sprint.
3. **Design & Development:**
   - Build features incrementally using **best coding practices**.
4. **Testing / Quality Assurance:**

    - Continuous testing ensures defects are caught early.

5. **Review / Demo:**
    - Stakeholders see working features and provide feedback.
6. **Retrospective:**
    - Team discusses **successes and improvements** for the next sprint.
7. **Release / Deployment:**
    - Small increments can be deployed frequently.

Advantages of Agile Model:

- **Flexibility:** Can adapt to requirement changes anytime.
- **Faster delivery:** Working software is delivered in short cycles.
- **Customer involvement:** Continuous feedback ensures software meets expectations.
- **Higher quality:** Continuous testing and iterative development catch defects early.
- **Team collaboration:** Daily stand-ups and retrospectives improve communication.

Disadvantages of Agile Model:

- Requires **high customer involvement**, otherwise feedback loop fails.
- Can be **chaotic** if team lacks discipline.
- **Documentation may be limited**, which can make future maintenance difficult.
- **Needs experienced teams** to manage sprints effectively.

Real-Time Example from Your Project:

"In my car insurance automation project, I used Agile with Scrum. Each sprint focused on one module, like login or car details page. Daily stand-ups helped track progress, and at the end of each sprint, I demonstrated working Selenium scripts to stakeholders. Based on their feedback, I adjusted scripts for new validations. Kanban board helped me visualize tasks, and retrospectives ensured continuous improvement in the next sprint. Agile allowed me to deliver features quickly and adapt to UI changes easily."

V-Model – Explanation

The **V-Model**, also called the **Verification and Validation model**, is a **sequential software development process** where **each development phase is directly associated with a corresponding testing phase**. It's called "V" because if you draw the development and testing activities in a diagram, it forms a V shape: the left side represents **development**, and the right side represents **testing/validation**.

Purpose of V-Model:

1. Ensure **systematic development and testing**.
2. Detect **defects early** by linking requirements with test cases.
3. Provide **clear documentation** for each phase.
4. Maintain **high-quality software** through structured verification and validation.

Phases of V-Model:

Left Side – Development Phases:

1. **Requirement Analysis:** Gather and document system requirements.
2. **System Design:** Define architecture and modules.
3. **High-Level Design (HLD):** Specify module relationships.
4. **Low-Level Design (LLD):** Detail individual module design.
5. **Implementation / Coding:** Developers write code for modules.

Right Side – Testing Phases (Validation):

1. **Unit Testing:** Tests individual modules (matches LLD).
2. **Integration Testing:** Tests module interactions (matches HLD).
3. **System Testing:** Tests the complete system (matches system design).
4. **Acceptance Testing:** Validates requirements and user needs (matches requirement analysis).

Advantages of V-Model:

- **Early defect detection** due to parallel testing planning.
- **Clear structure** and well-documented phases.

- Works well for **small to medium projects** with **fixed requirements**.
- **Simple and easy to use** because it's sequential.

Disadvantages of V-Model:

- **Not flexible**; changes in requirements are costly.
- **Not ideal for large projects** or those with evolving requirements.
- Testing starts **late** compared to Agile.
- Requires **detailed documentation** at every stage, which can slow progress.

Spiral Model – Explanation

The **Spiral Model** is a **risk-driven software development process** that combines elements of both **iterative development** and **waterfall model**. It emphasizes **risk assessment at every phase** and allows **repeated refinement** of the system through **multiple iterations or spirals**. It's called "Spiral" because the process moves in a spiral shape, with each loop representing a development cycle.

Purpose of Spiral Model:

1. Minimize **risks** by identifying and addressing them early.
2. Allow **incremental development** of large and complex projects.
3. Provide flexibility to **accommodate requirement changes**.
4. Ensure continuous **feedback from stakeholders**.

Phases of Spiral Model:

Each spiral loop consists of **four main phases:**

1. **Planning Phase:**
    - Requirements are gathered, objectives are defined, and project scope is outlined.
2. **Risk Analysis Phase:**
    - Potential risks (technical, financial, operational) are identified and mitigation strategies are planned.
3. **Engineering / Development Phase:**
    - Actual software design, coding, and testing are performed.

    - Includes prototyping to validate functionalities.

4. **Evaluation Phase:**
    - Customer or stakeholder reviews are conducted.
    - Feedback is collected to refine the next iteration.

Each loop of the spiral produces an updated version of the software until the final product is complete.

Advantages of Spiral Model:

- Focuses on **risk management**, reducing the chance of project failure.
- Supports **incremental delivery**, so parts of the system are available early.
- Flexible in handling **changes in requirements**.
- **Customer involvement** ensures the final product meets expectations.

Disadvantages of Spiral Model:

- **Complex and costly** compared to simpler models.
- Requires **risk assessment expertise**; not suitable for small projects.
- Can **take longer** if too many iterations are needed.

Functional Testing and Non-Functional Testing – Explanation

In software testing, understanding the difference between **functional** and **non-functional** testing is crucial because they focus on **different aspects of the application**.

Functional Testing:

Functional testing verifies that the **software performs the functions it is supposed to do** according to the **requirements or specifications**. It focuses on **what the system does**.

Purpose:

- Ensure that **features work as expected**.
- Validate the **business logic, workflows, and user actions**.

Examples in real projects:

- Checking login functionality: valid credentials should allow access.
- Verifying registration forms accept correct inputs and display error messages for invalid inputs.
- Testing car insurance premium calculation logic in your project.

Advantages:

- Confirms the application meets **functional requirements**.
- Detects errors in **specific features**.
- Can be **automated** using Selenium, Cucumber, or TestNG.

Disadvantages:

- Doesn't check **performance, scalability, or usability**.
- Limited to **functional correctness**, ignoring system behavior under stress.

Non-Functional Testing:

Non-functional testing verifies **how the system performs** rather than what it does. It focuses on **quality attributes** like performance, reliability, usability, and security.

Purpose:

- Ensure the system meets **performance and quality standards**.
- Assess **user experience, security, and stability** under different conditions.

Examples in real projects:

- Checking **website load time** when 1000 users try to access the insurance portal simultaneously.
- Testing **SSL certificate handling** using your SSLBypass utility.
- Measuring **browser compatibility** across Chrome, Edge, and Firefox.

Advantages:

- Ensures system is **stable, secure, and responsive**.
- Helps identify bottlenecks before production.

Disadvantages:

- More **complex and time-consuming** than functional testing.
- Often requires **specialized tools** like JMeter for performance or LoadRunner for load testing.

In short:

- **Functional testing** = "Does it work?"
- **Non-functional testing** = "How well does it work?"

Handling Exceptions in Your Project – Explanation

In software testing automation, **exceptions** occur when something unexpected happens at runtime, like an element not found, a timeout, or a file not accessible. Handling these exceptions properly ensures the **test execution continues smoothly** and provides meaningful **logs or reports**.

How Exceptions Are Handled in Your Project:

1. **Try-Catch Blocks:**
    - Most of your utility methods (like in ScreenShot, ExcelUtil, ConfigReader) use try-catch blocks.
    - Example: When reading/writing Excel data, if the file is missing or a cell is null, the IOException is caught, and a meaningful message is printed.
    - This prevents the test from **crashing unexpectedly**.
2. **Custom Runtime Exceptions:**
    - In some utility methods, after catching an exception, you rethrow it as a RuntimeException with a descriptive message.
    - Example:
    - throw new RuntimeException("Screenshot capture failed: " + e.getMessage());
    - This helps in **debugging quickly** because the error clearly explains what failed.
3. **WebDriver Waits and Selenium Exceptions:**
    - In WaitUtils, you handle TimeoutException when elements are not visible or clickable within the specified time.
    - This ensures that the test waits for elements properly instead of failing immediately.
4. **SSL Handling:**
    - In SSLBypass, exceptions from SSLContext initialization are caught, preventing SSL errors from stopping your tests on HTTPS sites.

5. **Logging:**
    - ConfigReader uses **Log4j** to log errors or warnings when property files fail to load or a key is missing.
    - This helps **track where exceptions happened** without stopping execution.
6. **Graceful Test Failure:**
    - If a test fails due to an exception, you capture a **screenshot** using ScreenShot.screenShotTC() before failing, which is helpful for reporting in **Allure** or **ExtentReports**.

Advantages of Exception Handling in Your Project:

- Prevents tests from stopping abruptly.
- Provides clear logs and messages for debugging.
- Supports automated report generation with screenshots.
- Handles Selenium-specific issues like element not found or SSL issues.

Usage of Selenium – Explanation

Selenium is primarily used for **automating web applications** to test their functionality, performance, and behavior across different browsers. Its main goal is to **reduce manual testing efforts**, increase **test accuracy**, and allow **repeated execution** of test cases quickly.

In your project, Selenium is used for:

1. **Web Browser Automation:**
    - Opening browsers like Chrome and Edge (DriverFactory) and performing actions such as clicking buttons, entering text, selecting options, and submitting forms.
2. **Cross-Browser Testing:**
    - With parameters in testng.xml, you can run the same test on multiple browsers (Chrome, Edge) without changing the code.
3. **Synchronization with Web Elements:**
    - Using explicit and implicit waits (WaitUtils) to ensure elements are loaded before interacting. This avoids flaky tests due to page load delays.
4. **Handling Alerts and Popups:**

    - Selenium provides methods to accept, dismiss, or read alert messages (driver.switchTo().alert()).
5. **Capturing Screenshots:**
    - With ScreenShot class, Selenium captures screenshots whenever a test fails or for reporting purposes.
6. **Data-Driven Testing:**
    - Selenium interacts with data from Excel (ExcelUtil) or XML (XmlReader) to test multiple scenarios automatically without manually entering values.
7. **Integration with Frameworks:**
    - Works seamlessly with **TestNG**, **Cucumber**, **Allure**, and **ExtentReports** in your project to generate reports, execute BDD scenarios, and log test results.
8. **SSL and Security Testing:**
    - Using SSLBypass, Selenium allows automation on sites with invalid SSL certificates.

Purpose:

- To automate repetitive browser tasks.
- To ensure consistent and fast execution of tests.
- To improve coverage and detect defects early.

Advantages:

- Open-source and free.
- Supports multiple programming languages (Java in your case).
- Works with all major browsers.
- Integrates with CI/CD tools for continuous testing.

Cucumber BDD – Full Theory Explanation with Gherkin Words and Scenario

Cucumber is a **Behavior Driven Development (BDD) automation tool** that allows teams to write test cases in a **human-readable format** using **Gherkin language**. The main idea of BDD is to focus on the **behavior of the application** rather than its implementation. This helps bridge the gap between **technical and non-technical team members**, such as developers, testers, product owners, and business analysts. In Cucumber, tests are written as **feature files**, which describe the functionality of the application in plain English. The actual code is implemented separately in **step**

definition classes, which means that the test logic is separated from the programming logic, allowing for **better readability, reusability, and maintainability**.

Cucumber supports several **Gherkin keywords** that define the structure of scenarios:

- **Feature:** Defines the functionality being tested.
- **Scenario:** Represents a specific test case for that feature.
- **Given:** Describes the initial context or preconditions before an action is performed.
- **When:** Defines the action or event that triggers the behavior.
- **Then:** Describes the expected outcome or result of the action.
- **And / But:** Used to combine multiple steps for readability and flow.
- **Scenario Outline:** Used for **data-driven testing**, where the same scenario runs with multiple sets of inputs.
- **Examples:** Provides the data for Scenario Outline execution.
- **Background:** Defines common preconditions that are shared across multiple scenarios in a feature file.
- **Tags:** Allow selective execution of scenarios, e.g., @Smoke, @Regression, @Functional.

Purpose:
Cucumber is designed to ensure that **everyone on the team understands what the software is supposed to do**. It allows collaboration between developers, testers, and business stakeholders, ensuring that the application behavior matches requirements.

Advantages:

- Tests are **easy to read and understand** even by non-technical members.
- Promotes **collaboration** between technical and non-technical teams.
- Allows **reusable step definitions**, reducing duplicate code.
- Supports **data-driven testing** using Scenario Outline and Examples.
- Provides **integration with automation tools** like Selenium and reporting tools like Allure or ExtentReports.

Simple Real-Time Scenario Example (Feature File in Gherkin Language):

```
Feature: Login Functionality


  Background:
    Given User is on the login page

  @Smoke
  Scenario: Successful login with valid credentials
    Given User enters valid username "testuser"
    And User enters valid password "password123"
    When User clicks on the login button
    Then User should be redirected to the dashboard
    And A welcome message should be displayed

  @Regression
  Scenario Outline: Unsuccessful login with invalid credentials
    Given User enters username "<username>"
    And User enters password "<password>"
    When User clicks on the login button
    Then User should see an error message "Invalid credentials"

    Examples:
      | username   | password    |
      | wrongUser  | wrongPass  |
      | testuser   | wrongPass  |
```

Theory: Handling Excel Files in Selenium Automation with Apache POI

In Selenium automation projects, Excel files are commonly used for **data-driven testing**, where test input data, expected results, or configuration details are stored in Excel sheets. The **Apache POI library** is the most widely used Java library for reading and writing Microsoft Excel files (.xls and .xlsx). It provides a simple API to access sheets, rows, and cells in Excel, allowing automation scripts to dynamically read test data or write test results.

Purpose:

- Centralize test data separately from code.

- Facilitate testing multiple scenarios with different inputs.
- Enable automated logging of test results in a structured manner.

Advantages:

- Supports both old (.xls) and modern (.xlsx) Excel formats.
- Allows dynamic reading and writing of data.
- Reduces manual effort and errors in test data handling.

Disadvantages:

- Large Excel files may impact performance.
- Requires proper exception handling for file operations.
- Complex formatting or formulas may require additional handling.

Code Snippet: Reading and Writing Excel

Reading Data:

```
import java.io.FileInputStream;
import java.io.IOException;
import org.apache.poi.ss.usermodel.*;
import org.apache.poi.xssf.usermodel.XSSFWorkbook;

public class ExcelReadExample {
    public static void main(String[] args) throws IOException {
        String filePath = "src/test/resources/ProjectData.xlsx";
        FileInputStream fi = new FileInputStream(filePath);
        Workbook workbook = new XSSFWorkbook(fi);
        Sheet sheet = workbook.getSheet("Sheet1");
        Row row = sheet.getRow(1);
        Cell cell = row.getCell(1);

        String value = "";
        if(cell.getCellType() == CellType.STRING)
            value = cell.getStringCellValue();
        else if(cell.getCellType() == CellType.NUMERIC)
            value = String.valueOf((int)cell.getNumericCellValue());

        System.out.println("Cell Value: " + value);

        workbook.close();
        fi.close();
    }
}
```

Writing Data:

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import org.apache.poi.ss.usermodel.*;
import org.apache.poi.xssf.usermodel.XSSFWorkbook;

public class ExcelWriteExample {
    public static void main(String[] args) throws IOException {
        String filePath = "src/test/resources/ProjectData.xlsx";
        FileInputStream fi = new FileInputStream(filePath);
        Workbook workbook = new XSSFWorkbook(fi);
        Sheet sheet = workbook.getSheet("Sheet1");
        Row row = sheet.getRow(1);
        if(row == null) row = sheet.createRow(1);

        Cell cell = row.createCell(2);
        cell.setCellValue("Test Passed");

        FileOutputStream fo = new FileOutputStream(filePath);
        workbook.write(fo);


        workbook.close();
        fi.close();
        fo.close();


        System.out.println("Data written successfully!");
    }
}
```

---

1. Reading Multiple Values from Excel (Data-Driven Approach)

Theory:

In data-driven testing, you don’t just read a single value; you often need **multiple rows and columns** to test different scenarios. Instead of hardcoding test data, Excel sheets can contain multiple test cases, each with different inputs.

Purpose:

- Run the same test scenario multiple times with different inputs automatically.
- Reduce code duplication.
- Maintain a single source of truth for all test data.

How it works:

1. Open the Excel file using **Apache POI**.
2. Select the sheet containing test data.
3. Loop through all **rows and columns** to read each value.
4. Store data in a **2D array, list, or map** so it can be fed into test methods.

Advantages:

- Scalable for multiple test cases.
- Easy to update test data without touching code.
- Supports different data types like numeric, string, boolean.

Disadvantages:

- Large datasets may increase test execution time.

- Complex Excel formulas may require extra handling.

Code Snippet (Your Project Style with ExcelUtil)

```
public static String[][] readMultipleData(String sheetName) {
  try {
    fi = new FileInputStream(xfile);
    workbook = new XSSFWorkbook(fi);
    sheet = workbook.getSheet(sheetName);

    int rowCount = sheet.getPhysicalNumberOfRows();
    int colCount = sheet.getRow(0).getLastCellNum();

    String[][] data = new String[rowCount-1][colCount]; // skip header row

    for(int i=1; i<rowCount; i++) {
      Row row = sheet.getRow(i);
      for(int j=0; j<colCount; j++) {
        Cell cell = row.getCell(j);
        if(cell != null) {
          if(cell.getCellType() == CellType.NUMERIC)
            data[i-1][j] = String.valueOf((int)cell.getNumericCellValue());
          else
            data[i-1][j] = cell.toString();
        }
      }
    }

    workbook.close();
    fi.close();
    return data;
    } catch (Exception e) {
        e.printStackTrace();
        return null;
    }
}
```

✅ This method reads all rows except the header, returning a **2D array**. You can feed it into **Cucumber step definitions** or **TestNG @DataProvider** for multiple test iterations.

---

2. Writing Feature File Steps for Stakeholders (BDD Style)

Theory:

In **Cucumber BDD**, feature files are meant to be **readable by non-technical stakeholders**, like business analysts or product owners. The focus should be **on behavior, not implementation**.

Best Practices:

- Avoid hardcoding test values in steps. Use **placeholders or examples**.
- Use **Given-When-Then** structure to describe behavior clearly.
- Provide **scenarios with multiple examples** in a **Scenario Outline**.

Gherkin Keywords:

- Feature: High-level functionality.
- Scenario: A single test case.
- Scenario Outline: Template for multiple data-driven test cases.
- Examples: Table of values for Scenario Outline.
- Given: Precondition or setup.
- When: Action performed by the user.
- Then: Expected outcome.
- And/But: Additional steps for readability.

Example Feature File for Car Insurance Project

```
Feature: Car Insurance Quote
```

```
As a customer
I want to get a car insurance quote
So that I can buy insurance online easily

Scenario Outline: Get insurance quote for multiple cars
  Given the customer is on the insurance homepage
  When the customer enters "<City>", "<CarBrand>", and "<FuelType>"
  And the customer enters name "<Name>" and mobile number "<MobileNumber>"
  Then the system should display a valid insurance quote

  Examples:
    | City      | CarBrand | FuelType | Name      | MobileNumber |
    | Mumbai    | Honda    | Petrol   | Rahul     | 9876543210   |
    | Pune      | Toyota   | Diesel   | Priya     | 9123456780   |
    | Bengaluru | Suzuki   | Petrol   | Akash     | 9988776655   |
```

Why this works:

- Stakeholders can **read it like a story**.
- Multiple test cases are handled with a **single Scenario Outline**.
- No need to highlight single values manually; Excel or examples table can feed data automatically.

Technologies Implemented in Your Project

In my project, I have implemented **Selenium WebDriver**, **Cucumber BDD**, **TestNG**, **Apache POI**, **Allure Reporting**, and **Java** as the core technologies. **Selenium WebDriver** is used to automate browser actions like clicking buttons, filling forms, handling alerts, and navigating web pages. It supports multiple browsers and is highly flexible for testing web applications. **Cucumber BDD** is used to write human-readable feature files in **Gherkin language**, which allows stakeholders to understand the test scenarios without reading code. **TestNG** is integrated for test execution, parallel execution, grouping tests, and handling annotations like @BeforeClass and @AfterSuite. For reading and writing test data, **Apache POI** is used to interact with Excel files in a data-driven approach. **Allure Reporting** is implemented to generate detailed test reports with logs, screenshots, and steps, which makes results easy to analyze. Finally, the project is written in **Java**, which provides object-oriented features like **abstraction, encapsulation, inheritance, and polymorphism**, making the framework modular, reusable, and maintainable.

Purpose:

- Automate functional tests efficiently.
- Make test cases understandable to non-technical stakeholders.
- Generate clear, detailed reports.
- Handle multiple datasets dynamically.
- Ensure cross-browser compatibility.

Advantages:

- Reduces manual testing effort.
- Improves accuracy and speed of testing.
- Makes collaboration with business teams easier.
- Enables scalable and maintainable automation framework.
- Provides detailed reports for quick analysis.

TestNG in Your Project

TestNG is a testing framework inspired by JUnit but more powerful and flexible. It is widely used in Selenium automation projects because it supports annotations, grouping, parameterization, parallel execution, and report generation. TestNG helps organize test cases efficiently, execute them in a controlled way, and produce detailed execution reports.

Key Features of TestNG

1. **Annotations** – Special keywords that control the execution flow of test methods.
2. **Grouping** – Allows you to group tests logically and run related tests together.
3. **Parallel Execution** – Runs multiple tests simultaneously to save time.
4. **Testng.xml** – Configuration file to define test suites, tests, browsers, parameters, and execution order.
5. **Parameterized Tests** – Pass multiple data values to a test method without modifying the code.
6. **Reports** – Generates HTML reports with detailed results of passed, failed, and skipped tests.

TestNG Annotations and Their Purpose

Annotation	Purpose
@BeforeSuite	Runs once before the entire suite execution. Used to set up global resources.
@AfterSuite	Runs once after the entire suite execution. Used for cleanup, reports, etc.
@BeforeTest	Executes before each <test> in testng.xml. Useful for test-level setup.
@AfterTest	Executes after each <test> in testng.xml. Cleanup after test.
@BeforeClass	Runs before the first method in the current class. Used to initialize resources for that class.
@AfterClass	Runs after all methods in the current class. Used for class-level cleanup.
@BeforeMethod	Executes before each test method. Useful for preconditions like opening a URL.
@AfterMethod	Executes after each test method. Useful for postconditions like taking a screenshot.
@Test	Marks a method as a test case. Supports parameters like priority, groups, dependsOnMethods.
@DataProvider	Provides data for data-driven testing to a test method.

TestNG File (testng.xml)

Purpose:

- Organize tests into suites and tests.
- Define execution order.
- Pass parameters.
- Enable parallel execution.
- Group tests logically.

Example:

```
<suite name="AutomationSuite" parallel="tests" thread-count="2">
  <test name="ChromeTests" parallel="methods">
    <parameter name="browser" value="chrome"/>
    <groups>
      <run>
        <include name="functional"/>
        <exclude name="smoke"/>
      </run>
    </groups>
    <classes>
      <class name="testRunner.TestNGTestRunner"/>
    </classes>
  </test>

  <test name="EdgeTests">
    <parameter name="browser" value="edge"/>
    <classes>
      <class name="testRunner.TestNGTestRunner"/>
    </classes>
  </test>
</suite>
```

Explanation:

- parallel="tests" → Runs multiple <test> blocks in parallel.
- thread-count="2" → Maximum 2 parallel threads.
- parameter → Pass browser value dynamically.
- groups → Run only specific groups of tests (e.g., functional) and skip others (e.g., smoke).
- classes → Include the test classes to execute.

Grouping and Parallel Execution

- **Grouping**: Helps to run only specific test sets. For example, functional tests vs. smoke tests.
- **Parallel Execution**: Reduces execution time by running tests simultaneously in multiple threads.

Code Example with Groups:

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

public class SampleTest {

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

    @Test(groups = {"smoke"})
    public void homepageTest() {
        System.out.println("Smoke test: Homepage load");
    }

    @Test(groups = {"functional", "regression"})
    public void addToCartTest() {
        System.out.println("Functional + Regression test: Add to Cart");
    }
}
```

Execution:

- Using groups in testng.xml, we can execute only functional tests, skipping smoke tests.

Advantages of TestNG

1. Supports flexible test configuration using annotations.

2. Can run tests in parallel to save execution time.
3. Groups and parameterization provide modular test execution.
4. Generates detailed HTML reports.
5. Handles dependencies between tests.
6. Integrates smoothly with Selenium, Maven, and CI/CD tools like Jenkins.

In Java, Strings are objects that represent sequences of characters, and there are **two main ways to create Strings**:

---

1. Using String Literal

- A string literal is enclosed in double quotes " ".
- When you create a string this way, Java stores it in the **String Pool** to optimize memory.
- If the same literal is used elsewhere, Java reuses the existing object instead of creating a new one.

Example:

String str1 = "Hello";

String str2 = "Hello"; // str1 and str2 refer to the same object in the String Pool

Key Point:

- Memory-efficient because duplicates are not created in the String Pool.

---

2. Using new Keyword

- Using new String() always creates a **new object** in the heap memory, even if the same string exists in the pool.
- Useful when you explicitly want a new String object.

Example:

String str3 = new String("Hello");

String str4 = new String("Hello"); // str3 and str4 are different objects in heap

Key Point:

- Less memory-efficient because it creates a new object each time.
- Useful when you need an object reference independent of the String Pool.

Summary

Method	Memory Location	Reuse	Example
String literal	String Pool	Yes	String s = "Hello";
new keyword	Heap	No	String s = new String("Hello");

✅ **Purpose & Advantage:**

- Strings are immutable in Java (cannot be changed once created).
- Using literals is faster and saves memory, whereas new allows explicit control over object creation.

Collections Framework in Java

The **Collections Framework** in Java is a unified architecture that provides **interfaces, classes, and algorithms** to store, access, and manipulate groups of objects efficiently. It replaces older ways of handling groups of objects (like arrays) with a **more flexible, dynamic, and reusable structure**.

---

1. Main Interfaces

These define the **contract** for data structures:

1. **Collection** – The root interface; stores **groups of objects**. Subinterfaces include **List, Set, Queue**.
    - Does **not include Map**.
2. **List** – Ordered collection, allows duplicates. Common implementations:
    - **ArrayList**: Dynamic array, fast random access.
    - **LinkedList**: Doubly linked list, faster insertion/deletion in middle.
    - **Vector**: Synchronized ArrayList (legacy, rarely used now).
      Purpose: When order matters and duplicates are allowed.
3. **Set** – Unordered collection, **no duplicates allowed**. Common implementations:
    - **HashSet**: Uses hash table; fast lookup; no order.
    - **LinkedHashSet**: Maintains insertion order.

   - **TreeSet**: Sorted order using natural ordering or comparator.
     Purpose: To store unique elements efficiently.

4. **Queue** – FIFO (First In First Out) structure. Common implementations:
   - **LinkedList**: Implements Queue interface.
   - **PriorityQueue**: Orders elements based on priority (natural or custom).
     Purpose: Task scheduling, buffer management.

5. **Deque** – Double-ended queue; elements can be added/removed from both ends.
   - Implemented by **ArrayDeque** and **LinkedList**.
     Purpose: Stack + Queue operations in one.

6. **Map** – Stores **key-value pairs**, not part of Collection interface.
   - **HashMap**: Unordered, allows null key/value.
   - **LinkedHashMap**: Maintains insertion order.
   - **TreeMap**: Sorted by key.
   - **Hashtable**: Synchronized, legacy.
     Purpose: Fast lookup using keys.

---

2. Utilities in Collections

The **Collections class** provides **static methods** for common tasks:

- sort(): Sort a List.
- reverse(): Reverse elements.
- shuffle(): Randomly shuffle elements.
- max() / min(): Get maximum/minimum element.
- binarySearch(): Find an element in sorted list.
- synchronizedList(), unmodifiableList(): Thread-safety and immutability.

---

3. Comparator vs Comparable

1. **Comparable** – Used when class objects need **natural ordering**.
   - Implement compareTo() in your class.
   - Only **one way to compare**.
2. class Employee implements Comparable<Employee> {

3. int salary;
4. public int compareTo(Employee e) {
5. return this.salary - e.salary; // ascending order
6. }
7. }
8. **Comparator** – Used for **custom sorting** without modifying the class.
    - Implement compare() method.
    - Multiple ways to sort.
9. class SalaryComparator implements Comparator<Employee> {
10. public int compare(Employee e1, Employee e2) {
11. return e2.salary - e1.salary; // descending order
12. }
13. }

Difference: Comparable is **internal**, Comparator is **external**; Comparable allows single natural sort, Comparator allows multiple custom sorts.

---

4. Importance and Purpose of Collections

- **Efficiency:** Provides faster insertion, deletion, search, sorting.
- **Reusability:** Standardized interfaces mean you can switch implementations easily.
- **Flexibility:** Dynamic resizing (ArrayList), multiple ordering options (TreeSet, PriorityQueue).
- **Thread Safety:** Some implementations like Vector and Hashtable are synchronized; also Collections class provides wrappers.
- **Readability:** Standard methods like add, remove, contains, sort make code clean.

---

5. Inheritance Overview

Collection (Interface)

├─ List

| ├─ ArrayList

```
|   ├─ LinkedList
|   └─ Vector
├─ Set
|   ├─ HashSet
|   ├─ LinkedHashSet
|   └─ TreeSet
└─ Queue
    ├─ LinkedList
    └─ PriorityQueue

Map (Interface)
 ├─ HashMap
 ├─ LinkedHashMap
 ├─ TreeMap
 └─ Hashtable
```

✅ **In simple human words:**
Collections framework is like a **toolbox for handling groups of data**. You can choose the right tool (List, Set, Map, Queue) depending on your requirement: preserve order, avoid duplicates, fast lookup, or priority sorting. Utility methods and comparators make these tools even smarter and flexible for real-world applications.

---

1. Array vs ArrayList

Feature	**Array**	**ArrayList**
Size	Fixed at creation	Dynamic, grows automatically
Type	Can hold primitives directly	Only objects (for primitives, use wrapper classes like Integer)
Performance	Fast for simple access	Slightly slower due to dynamic resizing
Methods	Limited (length, index access)	Rich API: add, remove, contains, sort, etc.

Feature	**Array**	**ArrayList**
Memory	Slightly less overhead	Slightly more overhead due to object wrappers
Use Case	When size is known and fixed	When size changes frequently

In short: Arrays are **simple and static**, ArrayLists are **flexible and dynamic**.

---

2. Hashtable vs HashMap

Feature	**Hashtable**	**HashMap**
Thread Safety	Synchronized (thread-safe)	Not synchronized (not thread-safe)
Null Keys/Values	Not allowed	One null key allowed, multiple null values allowed
Performance	Slower due to synchronization	Faster
Legacy	Legacy class	Modern, preferred
Use Case	Multi-threaded legacy code	Modern single-threaded apps, or use ConcurrentHashMap for threads

In short: Hashtable = legacy + thread-safe; HashMap = modern + flexible.

---

3. List vs Set

Feature	**List**	**Set**
Duplicates	Allowed	Not allowed
Order	Maintains insertion order	HashSet: unordered, TreeSet: sorted, LinkedHashSet: insertion order
Indexing	Supports index (get, set)	No index
Common Implementations	ArrayList, LinkedList, Vector	HashSet, TreeSet, LinkedHashSet

Feature	List	Set
Use Case	When order and duplicates matter	When uniqueness is required

In short: List = order + duplicates; Set = unique elements only.

---

4. Vector vs ArrayList

Feature	Vector	ArrayList
Thread Safety	Synchronized (thread-safe)	Not synchronized
Performance	Slower due to synchronization	Faster
Legacy	Legacy class	Modern class
Resizing	Doubles size when full	Grows by 50% of current size
Use Case	Legacy multi-threaded apps	Modern single-threaded apps

In short: Vector = legacy thread-safe; ArrayList = modern and faster.

---

5. Quick Summary of Differences

- **Array vs ArrayList:** Fixed size vs dynamic size.
- **Hashtable vs HashMap:** Thread-safe legacy vs modern flexible map.
- **List vs Set:** Allows duplicates vs unique elements.
- **Vector vs ArrayList:** Synchronized vs unsynchronized; legacy vs modern.

✅ Human words explanation for interview:

- Use **Array** for simple, fixed-size data.
- Use **ArrayList** when your list grows dynamically.
- Use **HashMap** for key-value storage in modern apps; **Hashtable** only for legacy thread-safe code.
- Use **Set** when you need **unique values**, List when you care about order and duplicates.
- **Vector** is old-school, thread-safe; **ArrayList** is modern and faster.

---

1. HashMap vs TreeMap

- **HashMap**:
    - Stores **key-value pairs** without any order.
    - Allows **one null key** and multiple null values.
    - **Faster** for basic operations like get(), put().
    - Ideal when you **don't care about order** of keys.
- **TreeMap**:
    - Stores **key-value pairs in sorted order** (natural ordering of keys or custom Comparator).
    - **Does NOT allow null keys**, but allows null values.
    - Slower than HashMap due to **red-black tree structure**.
    - Useful when you need **sorted map** for reporting or logical order.

2. ArrayList vs LinkedList

- **ArrayList**:
    - Backed by an **array**, supports fast **random access** using index.
    - Slower for **insertion/deletion in the middle** (elements shift).
    - Uses **less memory** compared to LinkedList.
- **LinkedList**:
    - Stores elements as **nodes**, each pointing to the next/previous.
    - Fast **insertion/deletion** at start/middle/end.
    - Slower **access by index** because it traverses nodes.
    - Uses **more memory** due to node pointers.

3. Set Types

- **HashSet**:
    - No duplicates.
    - **No order maintained**.
    - Allows one null value.
    - Fast operations due to hashing.
- **LinkedHashSet**:
    - No duplicates.
    - Maintains **insertion order**.

    - Slightly slower than HashSet because of linked list overhead.
- **TreeSet**:
    - No duplicates.
    - **Sorted order** (natural or custom Comparator).
    - Does **not allow null**.
    - Slower due to red-black tree operations.

Write the code to access the second element from a String

ChatGPT said:

You can access the second element (character) of a String in Java using the charAt() method. Here's a simple example:

```
public class StringExample {
    public static void main(String[] args) {
        String text = "HelloWorld";

        // Access the second character (index 1, because indexing starts at 0)
        char secondChar = text.charAt(1);

        System.out.println("The second character is: " + secondChar);
    }
}
```

Output:

```
The second character is: e
```

Explanation:

- Strings in Java are **0-indexed**, so the first character is at index 0 and the second character is at index 1.
- charAt(index) retrieves the character at the specified index.

Data-Driven Workflow for JSON in Automation

1. **Purpose**:
    - To separate test data from test logic.
    - Makes tests maintainable and reusable.

    - Allows running multiple test scenarios with different input data automatically.

2. **JSON Structure**:
   A JSON file stores data in **key-value pairs** or arrays. Example:

```
[
 {
  "username": "user1",
  "password": "pass1",
  "city": "Mumbai"
 },
 {
  "username": "user2",
  "password": "pass2",
  "city": "Pune"
 }
]
```

3. **Steps in Workflow**:
    1. **Create JSON file**: Place all test inputs in JSON format under src/test/resources/data/.
    2. **Read JSON in Java**: Use a library like **Jackson** or **JSON-simple** to parse the JSON file into Java objects.
    3. **Use in Step Definitions**: Iterate through JSON objects and feed the data into your Selenium actions.
    4. **Write Back Results (Optional)**: If your test calculates outputs (like validation messages), write back to JSON or a log file for reporting.

Implementation in Java (Your Project Style)

Add dependency (Maven) for JSON parsing:

```
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-databind</artifactId>
```

```
    <version>2.17.1</version>
</dependency>
```

Java Utility Class to Read JSON:

```
package utilities;

import com.fasterxml.jackson.databind.ObjectMapper;
import java.io.File;
import java.util.List;
import java.util.Map;

public class JSONUtil {

    static String jsonFilePath = System.getProperty("user.dir") +
"/src/test/resources/data/testData.json";

    public static List<Map<String, String>> readJSON() {
        List<Map<String, String>> testData = null;
        try {
            ObjectMapper mapper = new ObjectMapper();
            testData = mapper.readValue(new File(jsonFilePath), List.class);
        } catch (Exception e) {
            e.printStackTrace();
        }
        return testData;
    }

    public static void writeJSON(List<Map<String, String>> data) {
        try {
            ObjectMapper mapper = new ObjectMapper();
            mapper.writerWithDefaultPrettyPrinter().writeValue(new File(jsonFilePath), data);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Usage in Step Definition (Data-Driven Test):

```
package stepDefinitions;

import io.cucumber.java.en.Given;
import utilities.JSONUtil;
import java.util.List;
import java.util.Map;

public class LoginSteps {

    @Given("I perform login with JSON data")
    public void loginWithJSONData() {
        List<Map<String, String>> testData = JSONUtil.readJSON();

        for (Map<String, String> data : testData) {
            String username = data.get("username");
            String password = data.get("password");

            System.out.println("Logging in with: " + username + " / " + password);
            // Here you call Selenium methods to enter username/password and click login
        }
    }
}
```

Advantages

- Centralized test data makes maintenance easy.
- Supports running multiple scenarios automatically.
- Flexible to integrate with Excel, JSON, or DB later.

Disadvantages

- JSON structure must be correct; parsing fails if malformed.
- Writing back results is optional and may complicate tests if not managed carefully.

Ways to Click an Element in Selenium

In Selenium, clicking an element can be done using **multiple approaches**, depending on the element's behavior and web page complexity:

1. **Standard Click Method**:
   Using WebElement.click(). This is the simplest way when the element is visible and clickable.
2. WebElement button = driver.findElement(By.id("submitBtn"));
3. button.click();
4. **Using JavaScript Executor**:
   Some elements may not respond to standard click due to overlays or dynamic loading. In that case, we can use JS:
5. WebElement button = driver.findElement(By.id("submitBtn"));
6. ((JavascriptExecutor)driver).executeScript("arguments[0].click();", button);
7. **Using Actions Class**:
   Useful for hover menus or complex user interactions:
8. WebElement button = driver.findElement(By.id("submitBtn"));
9. Actions actions = new Actions(driver);
10. actions.moveToElement(button).click().perform();
11. **Using Keys (ENTER/RETURN)**:
    When the element is a form input or link and pressing Enter triggers a click:
12. WebElement input = driver.findElement(By.id("username"));
13. input.sendKeys(Keys.ENTER);

Purpose: Ensures interaction with elements in all scenarios, even if the page is dynamic or has hidden layers.

Advantage: Offers flexibility; you can choose the method that works depending on page behavior.

Disadvantage: Using JS Executor or Actions adds overhead and is less readable than a normal click.

Difference Between findElement() and findElements()

Feature	**findElement()**	**findElements()**
Return Type	Single WebElement	List<WebElement>
Number of Elements	Returns **first matching element only**	Returns **all matching elements**
If Element Not Found	Throws NoSuchElementException	Returns **empty list**
Use Case	When you expect **one element**	When you want to **iterate through multiple elements**

Example:

```
// findElement()
WebElement firstButton = driver.findElement(By.tagName("button"));
firstButton.click();


// findElements()
List<WebElement> buttons = driver.findElements(By.tagName("button"));
for(WebElement btn : buttons) {
   System.out.println(btn.getText());
}
```

Purpose: Helps to interact with single or multiple elements appropriately.

Advantage:

- findElement() is straightforward for single interaction.
- findElements() avoids exceptions and allows bulk operations.

Disadvantage:

- Using findElement() incorrectly can break tests if the element is not present.
- findElements() may return an empty list, so you need a check before looping.

---

3. How will you find an Element?

In Selenium, finding elements is done using **locators**. Locators help Selenium identify the element on the page. Common locator types:

1. **By ID** (most preferred, unique):

```
WebElement searchBox = driver.findElement(By.id("searchInput"));
```

2. **By Name**:

```
WebElement email = driver.findElement(By.name("username"));
```

3. **By Class Name**:

```
WebElement button = driver.findElement(By.className("btn-primary"));
```

4. **By Tag Name**:

```
List<WebElement> links = driver.findElements(By.tagName("a"));
```

5. **By Link Text / Partial Link Text**:

```
WebElement homeLink = driver.findElement(By.linkText("Home"));

WebElement partialLink = driver.findElement(By.partialLinkText("About"));
```

6. **By CSS Selector**:

```
WebElement button = driver.findElement(By.cssSelector(".btn-primary"));
```

7. **By XPath**:

```
WebElement searchBox = driver.findElement(By.xpath("//input[@name='q']"));
```

Purpose: Selenium uses these locators to identify elements in the DOM for interaction.

---

4. How will you handle if you got 2 buttons when you find an element and Req is to click both buttons?

If the page has multiple elements matching the same locator, we use **findElements()** which returns a **list of WebElements**. Then iterate over the list and click each button.

Example using your project style with DriverFactory and waits:

```
List<WebElement> buttons =
DriverFactory.getDriver().findElements(By.className("btn"));

for(WebElement btn : buttons) {
   WaitUtils.waitForClickability(DriverFactory.getDriver(), btn,
ConfigReader.getExplicitWait());

   btn.click();

}
```

Explanation:

- findElements() returns all buttons.
- WaitUtils ensures the button is clickable before clicking.
- Loop allows clicking **both buttons** safely.

Advantage: Avoids NoSuchElementException and allows interaction with multiple elements efficiently.

5. Write XPath for search bar in www.google.com

Google search bar can be located using different XPaths:

1. **Using name attribute** (most stable):

```
//input[@name='q']
```

2. **Using type and title attributes**:

```
//input[@type='text' and @title='Search']
```

3. **Relative XPath with form parent**:

```
//form[@role='search']//input[@name='q']
```

6. Which library you are using for Excel?

In my project, I am using **Apache POI** library to handle Excel files. Apache POI provides classes to read, write, and manipulate **.xls** and **.xlsx** files.

- **Dependencies in Maven (your project):**

```
<dependency>

   <groupId>org.apache.poi</groupId>

   <artifactId>poi</artifactId>

   <version>5.4.1</version>

</dependency>

<dependency>

   <groupId>org.apache.poi</groupId>
```

```
  <artifactId>poi-ooxml</artifactId>
  <version>5.4.1</version>
</dependency>
```

Purpose:

- Read test data for **data-driven testing**.
- Write test results or logs back into Excel.
- Format cells with colors or styles for reporting.

Advantage:

- Supports both .xls and .xlsx.
- Allows reading/writing and formatting.
- Works seamlessly with automation frameworks.

---

7. How will you find last row and last column?

To find **last row** and **last column** in a sheet using Apache POI:

Example code snippet:

```
FileInputStream fi = new FileInputStream("src/test/resources/ProjectData.xlsx");
XSSFWorkbook workbook = new XSSFWorkbook(fi);
XSSFSheet sheet = workbook.getSheet("Sheet1");

// Last row index
int lastRow = sheet.getLastRowNum(); // returns index (0-based)

// Last column index in a specific row (e.g., first row)
int lastCol = sheet.getRow(0).getLastCellNum() - 1; // getLastCellNum() returns count, so subtract 1 for index

System.out.println("Last Row: " + lastRow);
System.out.println("Last Column: " + lastCol);

workbook.close();
```

fi.close();

Explanation:

- getLastRowNum() gives the index of the last populated row.
- getLastCellNum() gives total cells in a row (1-based), so subtract 1 for zero-based index.

Advantage: Useful for **dynamic data-driven testing** without hardcoding row/column values.

---

8. Have you used BDD? Explain where you used and how it works.

Yes, I have used **BDD (Behavior-Driven Development)** in my project.

Where I used:

- Implemented **Cucumber framework** for functional automation testing.
- Features like **Car Insurance workflow**, login/registration, and form validation are written in **.feature files**.

How it works:

1. **Feature File:** Written in plain English using **Gherkin keywords** (Given, When, Then, And, But) for better readability.
   - Example:
2. Feature: Car Insurance Form
3. Scenario: User submits invalid phone number
4. Given User is on Car Insurance page
5. When User enters invalid mobile number
6. And clicks Submit button
7. Then Error message should be displayed
8. **Step Definitions:** Java methods map each Gherkin step to Selenium code.
9. @When("User enters invalid mobile number")
10. public void enterInvalidMobile() {
11. WebElement mobileField = DriverFactory.getDriver().findElement(By.id("mobile"));
12. mobileField.sendKeys("12345");
13. }

14. **Runner Class:** TestNGTestRunner runs Cucumber scenarios using TestNG.
15. **Reporting:** Allure and ExtentReports generate readable test reports for stakeholders.

Advantages:

- Non-technical stakeholders can **understand requirements** via feature files.
- Improves **collaboration** between QA, developers, and business teams.
- Easy to maintain **reusable steps** across scenarios.

---

10. How will you handle if you haven't found any item on your page?

In Selenium, when an element is **not present**, trying to interact with it directly using `findElement()` will throw a **`NoSuchElementException`**. To handle this, you can use:

1. **Try-Catch block** – gracefully handle the exception:

```
try {
    WebElement item =
DriverFactory.getDriver().findElement(By.id("itemId"));
    item.click();
} catch (NoSuchElementException e) {
    System.out.println("Item not found on the page, skipping
operation.");
}
```

2. **Check element size with `findElements()`** – returns a list, even if empty:

```
List<WebElement> items =
DriverFactory.getDriver().findElements(By.id("itemId"));
if (items.size() > 0) {
    items.get(0).click();
} else {
    System.out.println("Item not found on the page.");
}
```

Explanation:

- `findElement()` throws an exception if the element is absent.
- `findElements()` returns an empty list, allowing safe handling without exceptions.

Advantage:

- Prevents test failure due to missing elements.
- Makes automation scripts **robust and dynamic**.

---

11. Do you know Regular Expressions in Java? Pattern to validate only numbers

Yes, **Regular Expressions (Regex)** in Java are used to match text patterns. They are widely used for **validating inputs**, like numbers, emails, or phone numbers.

Java Pattern and Matcher classes:

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class RegexExample {
    public static void main(String[] args) {
        String input = "12345";
        String regex = "^[0-9]+$";  // Only numbers

        Pattern pattern = Pattern.compile(regex);
        Matcher matcher = pattern.matcher(input);

        if (matcher.matches()) {
            System.out.println("Input contains only numbers.");
        } else {
            System.out.println("Invalid input. Only numbers 
allowed.");
        }
    }
}
```

Explanation:

- `^` – Start of string
- `[0-9]` – Digits from 0 to 9
- `+` – One or more occurrences
- `$` – End of string

Advantage:

- Validates inputs **before sending them to the server**.
- Reduces **manual error checking**.
- Can be reused for multiple fields like phone, age, pin code, etc.

---

1. Class & Object

A **class** is like a blueprint or template that defines **attributes (variables)** and **behaviors (methods)** of an object. It itself does not occupy memory.

An **object** is a real-world instance of a class. It is created using the new keyword and occupies memory. Objects can access the properties and methods defined in the class.

Purpose:

- Classes provide structure to organize code.
- Objects allow you to use that structure in real scenarios.

Example in your project context:
In your Selenium project, DriverFactory is a **class**, and DriverFactory.getDriver() returns the **object (WebDriver)** to perform actions like clicking or sending keys.

---

2. What is Exception

An **exception** is an unexpected event or error that occurs during program execution. It disrupts the normal flow of the program.

Types:

- **Checked Exception:** Must be handled in code using try-catch or throws. (e.g., IOException)
- **Unchecked Exception:** Runtime errors that may or may not be handled. (e.g., NullPointerException)

Purpose in your project:

- Exceptions in Selenium, like NoSuchElementException or TimeoutException, help handle scenarios where elements are missing or pages take too long to load.

Example:

```
try {

   WebElement element = DriverFactory.getDriver().findElement(By.id("username"));

   element.sendKeys("admin");

} catch (NoSuchElementException e) {

   System.out.println("Element not found!");

}
```

---

3. Difference between break & continue

Feature	break	continue
Purpose	Exits the current loop completely.	Skips the current iteration, continues next iteration.
Use-case	Stop looping when a condition is met.	Skip unwanted values without stopping the loop.
Example	Stop searching when a match is found.	Skip invalid records in a list.

Code Example:

```
for(int i=1; i<=5; i++) {
   if(i == 3) {
      continue; // skip 3
   }
   if(i == 4) {
      break; // exit loop
   }
   System.out.println(i);
}
// Output: 1, 2
```

---

1. Data Types in Java

Explanation:
Data types in Java define the kind of data a variable can hold. Java has two main categories:

- **Primitive Data Types:** These store simple values directly and include int, float, double, char, boolean, byte, short, and long. For example, int age = 25; stores the number 25 in memory.
- **Non-Primitive (Reference) Data Types:** These store references to objects rather than the actual value. Examples are String, Arrays, Classes, and Interfaces.

Purpose: Using data types ensures proper memory allocation and type safety, preventing errors like assigning a string to a number.

Advantages:

- Efficient memory usage
- Type checking at compile time
- Easier to understand and maintain code

Example:

```
int age = 30; // primitive

String name = "Sudarshan"; // non-primitive
```

---

2. Type Casting in Java

Explanation:
Type casting is converting a variable from one data type to another. There are two types:

- **Implicit Casting (Widening):** Automatic conversion from smaller to larger data type (int → long).
- **Explicit Casting (Narrowing):** Manual conversion from larger to smaller type (double → int) using parentheses.

Purpose: Allows flexibility in calculations, storing values, and ensuring compatibility between data types.

Advantages:

- Helps in mathematical operations
- Enables data manipulation without errors
- Useful when reading data from files or APIs

Example:

```
int i = 100;

double d = i;  // implicit casting

double price = 99.99;

int p = (int) price; // explicit casting
```

---

3. Access Modifiers in Java

Explanation:
Access modifiers control **who can access a class, method, or variable**. Java has four main types:

1. **private** – Accessible only within the same class.
2. **default (no modifier)** – Accessible within the same package.
3. **protected** – Accessible within package + subclasses.
4. **public** – Accessible from anywhere.

Purpose: Protects data, enforces encapsulation, and controls visibility.

Advantages:

- Prevents unauthorized access
- Helps maintain modular and secure code

- Enables abstraction

Example (Project Context):

```
public class ConfigReader {   // public class accessible everywhere
    private static Properties properties;  // private, encapsulated
    public static String getAppUrl() {     // public, accessible to all classes
        return properties.getProperty("app.url");
    }
}
```