OOPS Concepts implementation in Automation Framework

1. Inheritance

Inheritance is a concept in object-oriented programming where one class (called the child class) inherits the properties and methods of another class (called the parent class).

It helps us reuse common code and maintain a clean structure across the framework.

- In my automation framework, I've implemented inheritance mainly through the BasePage and BaseTest classes.

For example, the BasePage class acts as the parent class and contains all the reusable methods like:

- click()
- sendKeys()
- waitForElementToBeVisible()

It also initializes the WebDriver using:

- PageFactory.initElements(driver, this)

Then I have child classes like:

- LoginPage
- HomePage
- CartPage

These classes extend the BasePage, which means they automatically inherit all the common methods and WebDriver setup.
This avoids code duplication and keeps the framework modular and maintainable.

2. Polymorphism

Polymorphism means "many forms." It allows us to do the same action in different ways. In Java, there are two types of polymorphism:

1. Compile-time (Method Overloading)
2. Runtime (Method Overriding)

In my automation framework, I've used both types.

A) Method Overloading (Compile-Time):

This happens when multiple methods have the same name but different inputs (parameters).

I've used this in my utility classes, for example in a Helper class where I make overloaded methods for waiting on elements. One method accepts a custom timeout, while another uses a default timeout. This makes the code more flexible and easy to call.

Predefined Overloading methods I have used in my framework:

- System.out.println() → works with String, int, boolean, double, etc.
- Thread.sleep() → can take just milliseconds or milliseconds + nanoseconds
- String.valueOf() → works with int, double, char, boolean, etc.
- driver.switchTo().frame(int index) / driver.switchTo().frame(String nameOrId) / driver.switchTo().frame(WebElement element)
- driver.findElement(By.id) / driver.findElement(By.xpath) / driver.findElements(By.tagName)
- Actions class click() → click() / click(WebElement element)
- WebDriverWait until() → until(ExpectedConditions.elementToBeClickable) / until(ExpectedConditions.visibilityOfElementLocated)

Customized Overloading methos in Selenium Framework:

- waitForElement(WebDriver driver, WebElement element, int timeoutSeconds)
- waitForElement(WebDriver driver, WebElement element)
- clickElement(WebElement element)
- clickElement(WebElement element, int waitTime)

This way, I can choose the version of the method depending on the page behavior, like LoginPage needing a short wait but CartPage needing a longer wait.

B) Method Overriding (Run-Time):

This happens when a child class changes or provides its own version of a method that is already defined in the parent class.

In my framework, I've used this in test classes that extend a BaseTest class. BaseTest has common setup and teardown methods, but specific test classes override them to add their own page-level setup.

Predefined Overriding methods I have used in my framework:

- equals() → overridden to compare objects by value instead of reference.
- Selenium's findElement(By by) → overridden in RemoteWebDriver, ChromeDriver, FirefoxDriver
- driver.get("url") → implemented differently by ChromeDriver, FirefoxDriver, EdgeDriver
- driver.close() / driver.quit() → overridden by each browser driver
- driver.navigate().to(), back(), forward(), refresh() → each browser driver has its own implementation
- TestNG annotations like @BeforeMethod and @AfterMethod can be overridden in child test classes.

Customized Overriding in Framework:

- BaseTest has a generic setup() method to launch the browser.
- LoginTest overrides setup() to also open the login page.
- CheckoutTest overrides setup() to directly navigate to the checkout page.
- tearDown() can also be overridden for different cleanup needs.

3. Encapsulation

Encapsulation means "wrapping data (variables) and code (methods) together into a single unit."
In simple words, it is about hiding internal details and only exposing what is necessary through getter and setter methods.
This makes the code secure, reusable, and easy to maintain.

In Java, we achieve encapsulation by:

1. Declaring variables of a class as private
2. Providing public getter and setter methods to access and update the values

In my automation framework, I have used encapsulation in multiple ways:

- In Page Object Model (POM) classes, where WebElements are kept private and only public methods are exposed to interact with them.

- In Configuration or Utility classes, where sensitive data such as environment URLs, browser type, and timeouts are kept private and accessed only through public getter methods.

Using encapsulation in my automation framework keeps the code secure, controlled, and easier to maintain.

4. Abstraction

Abstraction means hiding the internal implementation and showing only the services to the user.
It focuses on what a system does, not on how it does it.
It is like using a TV remote — we press the button to change the channel but we don't need to know how the inside circuits work.

In programming, abstraction lets us focus on the main idea instead of the internal details. This makes the program easier to read, use, and maintain.

In Java, abstraction is done by:

1. Abstract classes (they can have normal methods and abstract methods without body)
2. Interfaces (they only have method names, the actual work is written in another class)

In my automation framework, I use abstraction in:

- Base classes like BaseTest and BasePage, which keep common methods like launching the browser or setup, while child classes add their own details.

- Interfaces for things like browser actions or logging, which different classes implement in their own way.

It helps reduce complexity, makes the code cleaner, and allows easy changes in the future.

5. Interfaces

Interface in Java means a blueprint of a class.
It is used to achieve 100% abstraction because it only contains method names (signatures) but no implementation.
A class that implements an interface must provide the body (implementation) of all the methods.

In simple words:
Interfaces define "what to do," and classes decide "how to do it."

In my automation framework, I have used interfaces to keep my code flexible and reusable.

A) Predefined Interfaces in Selenium / Java:

- WebDriver → implemented by ChromeDriver, FirefoxDriver, EdgeDriver etc.
- WebElement → implemented by different classes representing HTML elements.
- TakesScreenshot → used to capture screenshots.
- JavascriptExecutor → used to run JavaScript code inside the browser.
- Alert → to handle JavaScript alerts, popups.

B) Customized Interfaces in my Framework:

- BrowserActions → I created an interface to define methods like openBrowser(), closeBrowser(), navigateTo().
- Logger → an interface to define methods like logInfo(), logError(), logWarning().
- TestDataProvider → interface for reading test data from Excel, JSON, or Database.

By using interfaces:

- My framework supports multiple browsers (Chrome, Firefox, Edge) without changing test code.
- I can switch between different logging or reporting implementations easily.
- Test data can come from any source without affecting the test scripts.