Selenium WebDriver Fundamentals

Architecture, Locators, Waits & Browser Handling

A comprehensive training programme for QA engineers and test automation professionals

CHAPTER 1

Introduction to Selenium

Understanding the foundation of modern web test automation

What Is Selenium?

Definition

Selenium is an open-source suite of tools and libraries that enables automation of web browsers across different platforms and programming languages.

Selenium automates browsers. That's it! What you do with that power is entirely up to you. Primarily, it is used for automating web applications for testing purposes, but is certainly not limited to just that.

Selenium supports multiple programming languages including Java, Python, C#, JavaScript, and Ruby, making it accessible to teams with diverse technical backgrounds.

Why Selenium Is Used in Test Automation

Open Source & Free

No licensing costs, supported by a large community, continuous improvements and updates

Cross-Browser Support

Test across Chrome, Firefox, Safari, Edge, and Opera using the same codebase

Language Flexibility

Write tests in Java, Python, C#, Ruby, or JavaScript based on team expertise

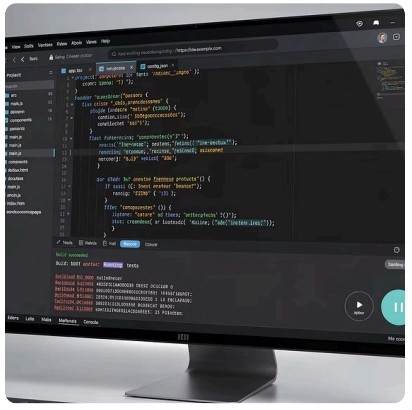
Platform Independent

Run tests on Windows, macOS, Linux, and even mobile browsers

Manual Testing vs Selenium Automation

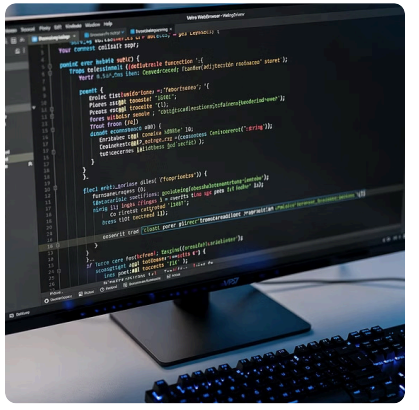
Aspect	Manual Testing	Selenium Automation
Execution Speed	Slow, depends on tester	Fast, runs in minutes
Repeatability	Time-consuming to repeat	Execute unlimited times
Initial Cost	Lower upfront cost	Higher initial investment
Long-term Cost	Expensive over time	Cost-effective for regression
Human Error	Prone to mistakes	Consistent execution
Exploratory Testing	Excellent for exploration	Limited to scripted tests
Best For	Usability, ad-hoc testing	Regression, smoke tests

Components of the Selenium Suite



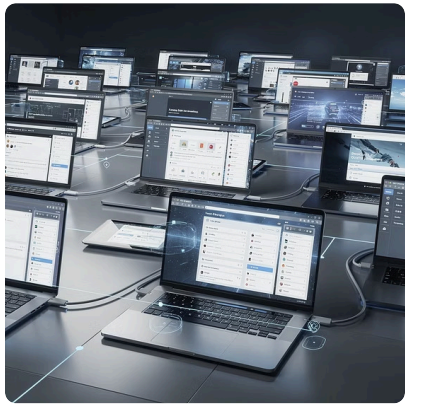
Selenium IDE

A browser extension for Firefox and Chrome that allows record and playback of user interactions. Ideal for quick prototyping and creating simple test cases without programming knowledge.



Selenium WebDriver

The core component that provides language-specific bindings to control browsers programmatically. Enables complex test scenarios through programming APIs in multiple languages.



Selenium Grid

A distributed testing tool that executes tests across multiple machines and browsers simultaneously. Enables parallel test execution and cross-browser testing at scale.

Real-World Automation Use Cases

Regression Testing

Automated execution of comprehensive test suites after each code change to ensure existing functionality remains intact. Typically runs overnight or after deployments.

Smoke & Sanity Testing

Quick verification of critical functionality after builds. Automated smoke tests can run in minutes, providing rapid feedback to development teams.

Cross-Browser Validation

Testing web applications across multiple browser versions and platforms to ensure consistent user experience. Selenium Grid enables parallel execution across browsers.

Data-Driven Testing

Execute the same test scenario with multiple data sets from external sources like Excel, CSV, or databases. Ideal for testing forms with various input combinations.

Knowledge Check: Introduction to Selenium

1

Which Selenium component is best for distributed test execution?

A) Selenium IDE
B) Selenium WebDriver
C) Selenium Grid
D) Selenium RC

Answer: C) Selenium Grid – enables parallel execution across multiple machines and browsers

2

What is the primary advantage of Selenium over commercial tools?

A) Better technical support
B) Open-source and no licensing costs
C) Record and playback features
D) Built-in reporting

Answer: B) Open-source and no licensing costs – Selenium is free with strong community support

Interview Keywords to Remember

Browser Automation

Programmatic control of web browsers

Open Source

Free, community-driven development

Cross-Browser

Support for multiple browser types

WebDriver API

Programming interface for browser control

CHAPTER 2

Selenium Architecture

Understanding how WebDriver communicates with browsers

What Is Selenium WebDriver Architecture?

Selenium WebDriver follows a client-server architecture where your test script acts as the client and the browser acts as the server. Communication happens through a browser-specific driver that translates WebDriver commands into browser-native actions.

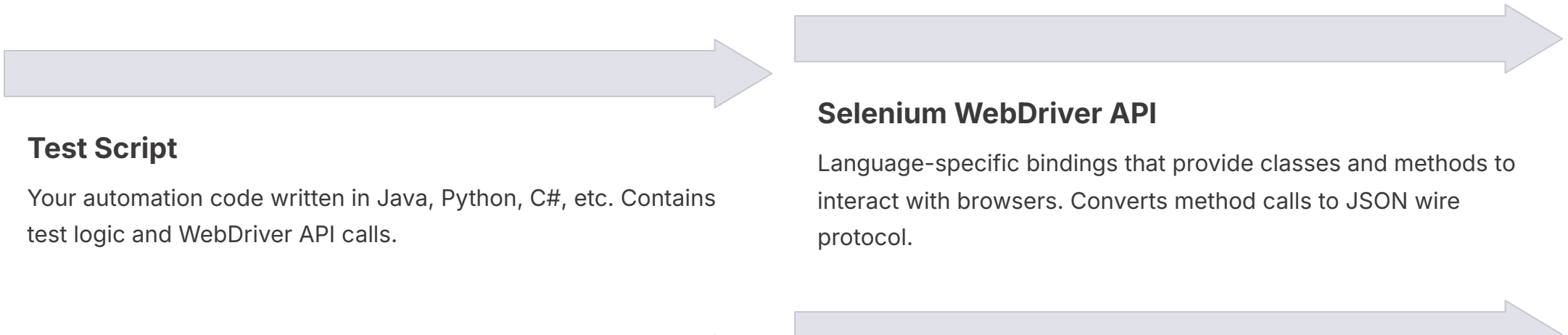
This architecture enables language independence – you can write tests in Java, Python, C#, or JavaScript, and they all communicate with browsers using the same WebDriver protocol. The architecture ensures that Selenium can support new browsers by simply adding new driver implementations.

> **Key Concept:** WebDriver is not a single tool but a collection of language bindings and driver implementations that work together to automate browsers.

Selenium Architecture: Client-Server Model

The architecture operates in four distinct layers. Your test script calls WebDriver API methods, which convert these calls into HTTP commands. The browser driver receives these commands and translates them into browser-specific instructions, which the browser then executes.

Architecture Components Explained



Test Script

Your automation code written in Java, Python, C#, etc. Contains test logic and WebDriver API calls.

Selenium WebDriver API

Language-specific bindings that provide classes and methods to interact with browsers. Converts method calls to JSON wire protocol.

Browser Driver

Executable file (ChromeDriver, GeckoDriver) that acts as a bridge between WebDriver and the browser. Browser-specific implementation.

Browser

The actual web browser (Chrome, Firefox, Edge) where your application runs and test actions are performed.

How Commands Travel from Script to Browser

01

Script Execution

Test script calls a WebDriver method like driver.findElement()

02

API Translation

WebDriver API converts the method call to HTTP request (JSON format)

03

Driver Processing

Browser driver receives HTTP request and translates to browser commands

04

Browser Execution

Browser performs the action and returns response back through the chain

Each WebDriver command follows this request-response cycle. For example, when you call `driver.get("https://example.com")`, it creates an HTTP POST request to the driver, which instructs the browser to navigate to that URL.

Understanding this flow helps debug issues – if a command fails, you can identify whether the problem is in your script, the WebDriver binding, the driver, or the browser itself.

Simple Architecture Example

Here's how a basic element interaction flows through the architecture:

```
// 1. Your test script (Java)
WebDriver driver = new ChromeDriver();
driver.get("https://example.com");
WebElement loginButton = driver.findElement(By.id("login"));
loginButton.click();

// Behind the scenes:
// 2. WebDriver API converts to HTTP request
// POST http://localhost:9515/session/abc123/element
// Body: {"using":"id", "value":"login"}

// 3. ChromeDriver receives request
// Translates to Chrome DevTools Protocol command

// 4. Chrome browser executes the click action
// Returns success response back to your script
```

Knowledge Check: Selenium Architecture

1

Which component directly communicates with the browser?

A) Test script
B) WebDriver API
C) Browser driver
D) JSON wire protocol

Answer: C) Browser driver – it acts as the bridge between WebDriver commands and browser-specific actions

2

What protocol does WebDriver use for communication?

A) FTP
B) HTTP/JSON
C) SOAP
D) SMTP

Answer: B) HTTP/JSON – WebDriver sends commands as HTTP requests with JSON payloads

Interview Trap: Is Selenium a Tool or Framework?

Common Interview Question

"Is Selenium a testing tool or a testing framework?"

Correct Answer: Selenium WebDriver is a library/API, not a complete framework. It provides browser automation capabilities but does not include test management, reporting, or assertion capabilities.

Why This Matters: Understanding this distinction shows architectural knowledge. Selenium must be integrated with testing frameworks like TestNG, JUnit, or pytest to create a complete test automation framework.

Key Points

- Selenium = browser automation library
- TestNG/JUnit = testing framework
- Together = complete solution
- Selenium has no built-in assertions
- Requires external reporting tools

CHAPTER 3

WebDriver Setup

Configuring your automation environment for success

Prerequisites for Selenium Setup

Programming Language

Java JDK 8+, Python 3.7+, or C# .NET installed on your system

IDE or Text Editor

IntelliJ IDEA, Eclipse, PyCharm, or Visual Studio Code for writing test scripts

Web Browser

Chrome, Firefox, or Edge browser installed (latest stable version recommended)

WebDriver Binaries

Browser-specific driver executables matching your browser version

Understanding Browser Drivers

Browser	Driver Name	Download Source	Executable
Google Chrome	ChromeDriver	chromedriver.chromium.org	chromedriver.exe
Mozilla Firefox	GeckoDriver	github.com/mozilla	geckodriver.exe
Microsoft Edge	EdgeDriver	developer.microsoft.com	msedgedriver.exe
Safari	SafariDriver	Built-in (macOS)	safaridriver

Critical Requirement: The driver version must match your browser version. Chrome 120 requires ChromeDriver 120. Version mismatches cause session creation failures.

WebDriver Binary Management

Manual Driver Management

Traditional approach where you download the driver executable manually and specify its path in your test code:

```
System.setProperty(
  "webdriver.chrome.driver",
  "C:/drivers/chromedriver.exe"
);
WebDriver driver = new ChromeDriver();
```

Challenges: Manual updates needed when browser updates, path configuration required, different paths across team environments.

WebDriverManager (Recommended)

Automated driver management library that downloads and configures the correct driver version automatically:

```
WebDriverManager.chromedriver().setup();
WebDriver driver = new ChromeDriver();
```

Benefits: Automatic version detection, downloads correct driver, handles path configuration, works across environments seamlessly.

Environment Setup Steps

Install Java/Python

Download and install JDK 8+ or Python 3.7+. Set JAVA_HOME or Python in system PATH.

Add Selenium Dependencies

Maven: Add selenium-java dependency. Python: Run `pip install selenium`

Configure Browser Driver

Use WebDriverManager or download driver manually and add to system PATH.

Verify Installation

Run a simple script to launch browser and verify everything works correctly.

Sample Setup Code

Basic Selenium script to verify your setup is working correctly:

```
// Java with WebDriverManager
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import io.github.bonigarcia.wdm.WebDriverManager;

public class SetupTest {
 public static void main(String[] args) {
 // Automatically download and setup ChromeDriver
 WebDriverManager.chromedriver().setup();

 // Create WebDriver instance
 WebDriver driver = new ChromeDriver();

 // Navigate to website
 driver.get("https://www.google.com");

 // Print title to verify
 System.out.println("Page title: " + driver.getTitle());

 // Close browser
 driver.quit();
 }
}
```

Common Setup Issues and Fixes

SessionNotCreatedException

Cause: Driver version doesn't match browser version
Fix: Update ChromeDriver to match Chrome version or use WebDriverManager for automatic matching

WebDriverException: driver executable not found

Cause: Driver executable not in system PATH or incorrect path specified
Fix: Add driver location to PATH environment variable or use absolute path in setProperty()

Connection refused: localhost

Cause: Driver failed to start or port conflict
Fix: Check firewall settings, ensure driver has execute permissions, verify port availability

NoClassDefFoundError

Cause: Selenium dependencies not added to project classpath
Fix: Verify Maven/Gradle dependencies are downloaded, rebuild project with dependencies

Knowledge Check: WebDriver Setup

1

What is the primary benefit of using WebDriverManager?

A) Faster test execution
B) Automatic driver version management
C) Built-in test reporting
D) Cross-browser compatibility

Answer: B) Automatic driver version management – WebDriverManager downloads and configures the correct driver automatically

2

Which exception indicates driver and browser version mismatch?

A) WebDriverException
B) NoSuchElementException
C) SessionNotCreatedException
D) TimeoutException

Answer: C) SessionNotCreatedException – commonly thrown when driver version doesn't match browser version

Trainer Notes: Live Setup Demonstration

Demo Checklist

- Show Chrome version check (chrome://version)
- Demonstrate WebDriverManager setup
- Run basic script with browser launch
- Show common error and fix
- Verify with different browser (Firefox)

Key Teaching Points: Emphasise the importance of version compatibility. Show both manual and automated driver management approaches. Demonstrate how to troubleshoot SessionNotCreatedException by checking versions.

Interactive Element: Ask learners to check their Chrome version and verify it matches their ChromeDriver. Have them run the sample script and share results.

Time Allocation: 10-12 minutes for complete demonstration with Q&A

CHAPTER 4

Locators in Selenium

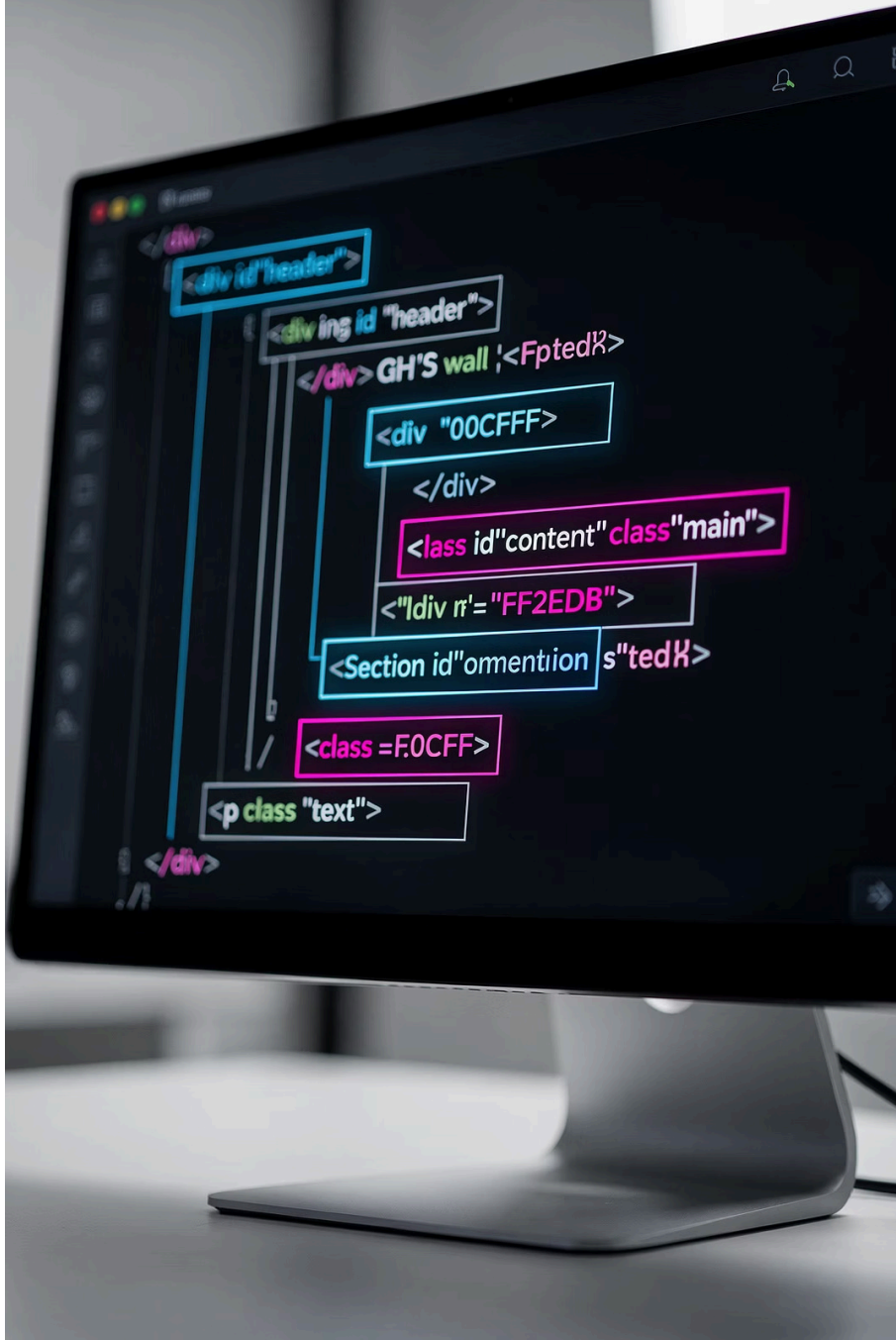
The foundation of element identification and interaction

What Are Locators?

Locators are mechanisms used to identify and locate web elements on a webpage. Every interaction in Selenium – clicking a button, entering text, reading values – requires first finding the element using a locator.

Think of locators as addresses that help Selenium find elements in the HTML DOM (Document Object Model). Just as a postal address uniquely identifies a house, a good locator uniquely identifies a web element.

> **Critical Concept:** The quality of your locators directly impacts test reliability. Robust locators create stable tests; weak locators lead to flaky, unreliable automation.



Why Locators Are Critical in Automation

Test Stability

Strong locators remain valid even when page layout or styling changes. Tests break less frequently, reducing maintenance overhead.

Execution Speed

Efficient locators like ID and name find elements instantly. Complex XPath can slow down test execution significantly.

Code Readability

Clear, semantic locators make test scripts easier to understand and maintain. Good locators serve as documentation.

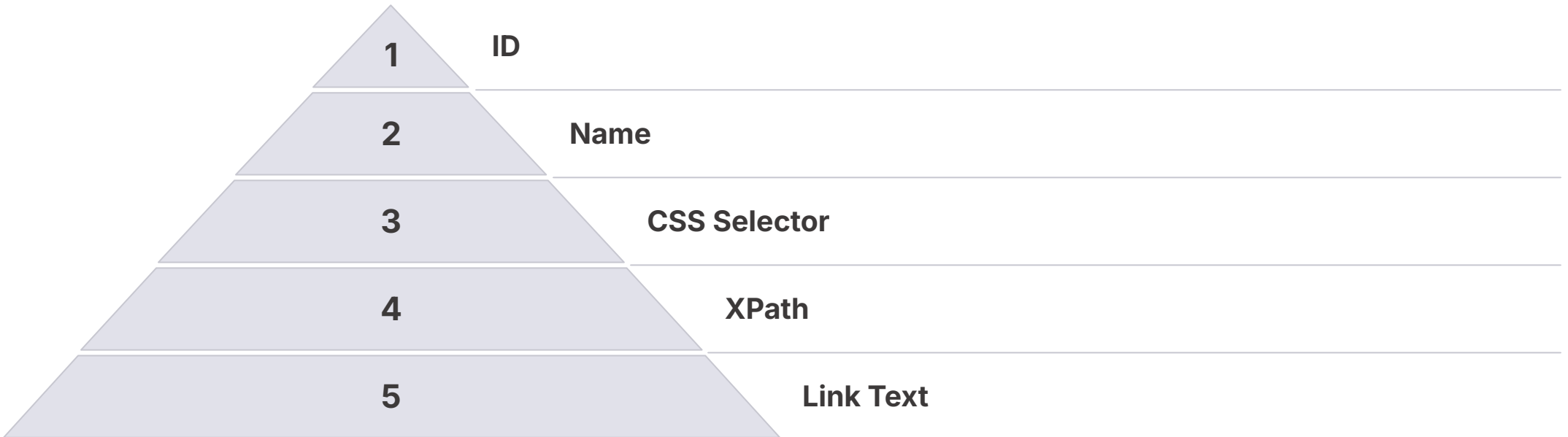
Dynamic Elements

Proper locator strategies handle dynamically generated IDs and elements that change with each page load.

Selenium Locator Types

Locator	Syntax	Use Case
id	By.id("loginButton")	Fastest and most reliable when ID is unique and stable
name	By.name("username")	Common for form elements with name attributes
className	By.className("btn-primary")	Good for elements with distinctive CSS classes
tagName	By.tagName("input")	Finding elements by HTML tag, less specific
linkText	By.linkText("Click Here")	Exact text match for hyperlinks
partialLinkText	By.partialLinkText("Click")	Partial text match for links
xpath	By.xpath("//input[@id='user']")	Most flexible, can traverse DOM, slower
cssSelector	By.cssSelector("#user")	Faster than XPath, CSS-based selection

Locator Priority and Best Practices



Prefer locators at the top of the pyramid. ID is fastest and most reliable. Use XPath and CSS selectors when simpler locators aren't available, but write them carefully to avoid brittleness.

Locator Examples: ID, Name, ClassName

HTML Code

```
<input
  id="email"
  name="userEmail"
  class="form-control"
  type="text"
  placeholder="Enter email"
/>

<button
  id="submitBtn"
  class="btn btn-primary"
>
  Submit
</button>
```

Selenium Locators

```
// By ID (preferred)
driver.findElement(By.id("email"));
driver.findElement(By.id("submitBtn"));

// By Name
driver.findElement(By.name("userEmail"));

// By ClassName
driver.findElement(
 By.className("form-control")
);
driver.findElement(
 By.className("btn-primary")
);

// Note: className cannot have spaces
// Use "btn-primary", not "btn btn-primary"
```

XPath: Relative vs Absolute

Absolute XPath

Starts from root HTML element, traverses entire DOM path. Extremely brittle – breaks if any parent element changes.

```
/html/body/div[1]/div[2]/form/input[1]
```

Problems: Long and unreadable, breaks easily with page structure changes, not recommended for automation.

Relative XPath

Starts from anywhere in DOM using //. More flexible and maintainable. Uses element attributes for identification.

```
//input[@id='username']
//button[text()='Login']
//div[@class='error']//span
```

Benefits: Shorter and readable, survives page structure changes, focuses on element characteristics rather than position.

CSS Selector Examples

CSS selectors provide a powerful alternative to XPath with generally faster performance:

Pattern	CSS Selector	Description
ID	#username	Select by ID (# symbol)
Class	.btn-primary	Select by class (. symbol)
Tag	input	Select by tag name
Attribute	input[name='email']	Select by attribute value
Child	div > input	Direct child selector
Descendant	form input	Any descendant selector
Multiple classes	.btn.btn-primary	Element with both classes
Starts with	input[id^='user']	Attribute starts with value

Simple Locator Examples in Context

```
// Login form automation using different locators

// ID - Best choice when available
driver.findElement(By.id("username")).sendKeys("testuser");
driver.findElement(By.id("password")).sendKeys("password123");

// CSS Selector - Good alternative
driver.findElement(By.cssSelector("#username")).sendKeys("testuser");
driver.findElement(By.cssSelector("input[type='password']")).sendKeys("password123");

// Name attribute - Common for forms
driver.findElement(By.name("email")).sendKeys("test@example.com");

// Link Text - Perfect for links
driver.findElement(By.linkText("Forgot Password?")).click();

// Partial Link Text - When full text is too long
driver.findElement(By.partialLinkText("Forgot")).click();

// XPath - When other locators don't work
driver.findElement(By.xpath("//button[text()='Login']")).click();
driver.findElement(By.xpath("//input[@placeholder='Enter username']")).sendKeys("user");
```

Knowledge Check: Locators

1

Which locator is fastest and most reliable?

A) XPath
B) CSS Selector
C) ID
D) ClassName

Answer: C) ID – fastest lookup in DOM, most reliable when IDs are unique and stable

2

What is the main problem with absolute XPath?

A) It's too slow
B) It breaks easily with page changes
C) It doesn't support attributes
D) It only works in Chrome

Answer: B) It breaks easily with page changes – absolute XPath depends on complete DOM structure

3

Which locator would you use for a link with text "Sign Up Now"?

A) By.id("signup")
B) By.linkText("Sign Up Now")
C) By.tagName("a")
D) By.className("link")

Answer: B) By.linkText("Sign Up Now") – linkText is specifically designed for hyperlinks with exact text match

Interview Keywords: Locators

Robust Locators

Stable identifiers that survive page changes

Dynamic Elements

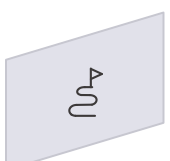
Elements with changing IDs or attributes

DOM Traversal

Navigating parent-child relationships

Locator Performance

ID fastest, complex XPath slowest



Relative XPath

Preferred over absolute for maintainability

CHAPTER 5

Web Elements & Actions

Interacting with elements once they're located

What Is a WebElement?

A WebElement represents an HTML element on a webpage. Once you locate an element using a locator, Selenium returns a WebElement object that you can interact with.

WebElement is an interface in Selenium that provides methods to perform actions on DOM elements like clicking buttons, entering text, reading values, and checking element states.

```
// Find element - returns WebElement
WebElement loginButton =
  driver.findElement(By.id("login"));

// Now you can interact with it
loginButton.click();
```

Key Concept

Every element interaction in Selenium follows a two-step process:

- **Step 1:** Locate the element using a locator strategy
- **Step 2:** Perform an action using WebElement methods

Understanding this distinction helps debug issues – is the element not found (locator problem) or not responding (interaction problem)?

Common WebElement Methods

click()

Clicks on an element (button, link, checkbox, radio button). Most commonly used method.

```
element.click();
```

sendKeys()

Enters text into input fields. Can also send special keys like ENTER or TAB.

```
element.sendKeys("text");
```

getText()

Retrieves visible text from an element. Returns inner text as a String.

```
String text = element.getText();
```

clear()

Clears text from input fields. Use before sendKeys() to ensure clean input.

```
element.clear();
```

isDisplayed()

Checks if element is visible on page. Returns boolean true/false.

```
boolean visible = element.isDisplayed();
```

isEnabled()

Checks if element is enabled and interactable. Returns boolean.

```
boolean enabled = element.isEnabled();
```

isSelected()

Checks if checkbox or radio button is selected. Returns boolean.

```
boolean selected = element.isSelected();
```

getAttribute()

Retrieves attribute value like "href", "value", "class" from element.

```
String value = element.getAttribute("value");
```