The Real Selenium Hierarchy

SearchContext, WebDriver, RemoteWebDriver, and ChromeDriver are ordinary Java -- two interfaces and two classes. Reading that hierarchy answers 'why is WebDriver an interface?' for good.

Automation Foundations · ashishautomationlabs.store

You can now read Selenium's own design as ordinary Java. `SearchContext` is the topmost interface, declaring only `findElement` and `findElements`. `WebDriver` is an interface that extends it, adding the browser-level methods such as `get`, `close`, and `quit`. `RemoteWebDriver` is a class that implements those interfaces, supplying real bodies for every one of those methods. And `ChromeDriver`, `FirefoxDriver`, and `EdgeDriver` extend `RemoteWebDriver`.

So the famous line contains everything in this chapter and the last. `WebDriver` is an interface used as the declared type. `ChromeDriver` is a class several levels down that ultimately satisfies that contract. The assignment is an upcast, and every method call on `driver` is resolved at run time against the real object.

Write it the other way -- `ChromeDriver driver = new ChromeDriver();` -- and it still compiles and still runs. But now every method that receives that variable is tied to Chrome, and switching browsers means editing code rather than changing a configuration value. The interface on the left is what makes the suite portable.

Where declaring the concrete type actually costs you

A framework declares `ChromeDriver driver = new ChromeDriver();` in `BaseTest`, and a helper method takes a `ChromeDriver` parameter. It runs correctly, and declaring the real type is honest -- you get Chrome-specific methods without casting, and today the suite only tests on Chrome. The problem shows up the day a client asks for Edge: the change is not one line in a config file, it is every signature that mentions `ChromeDriver`. That is a day of work, once -- plus a regression risk across the whole suite, to avoid typing one word today. And it costs the ability to run the same suite on a remote grid, where the object is a `RemoteWebDriver` rather than a `ChromeDriver`. Declare the interface, create the implementation. `WebDriver driver = new ChromeDriver();` costs nothing today and keeps every future option open. Reach for the concrete type only when you genuinely need a method the interface does not offer, and confine that to one place.

Interfaces are the joints a framework bends at. `WebDriver` lets one suite drive any browser. `WebElement` is an interface, which is why the same code handles a button and a text field. A `DriverFactory` returns a `WebDriver`, hiding which browser was built. Page Objects are usually classes extending an abstract `BasePage`, because they share real state -- the driver -- while

capability contracts such as `Reportable` or `Retryable` are interfaces. When you write a helper, take the interface as the parameter type: a method that accepts `WebDriver` works with everything, and a method that accepts `ChromeDriver` works with one thing.

Every idea in this chapter compounds into that one hierarchy. The closing bug hunt shows exactly how easy it is to write code that quietly throws all of it away.