Interview Questions: OOP

Seven questions on the four OOP concepts, extends, overloading vs overriding, why WebDriver is an interface, and access modifiers -- with a real framework example for each, not just a definition.

Automation Foundations · ashishautomationlabs.store

Q29 · ●●● -- What are the four OOP concepts, with an example of each?

Weak answer: "Encapsulation, inheritance, polymorphism, and abstraction."

Strong answer: "Encapsulation is an object controlling its own state -- private locators in a Page Object, exposed through action methods. Inheritance is a class building on another -- every test class extending a `BaseTest` that holds setup and teardown. Polymorphism is one reference behaving as whichever object it actually holds -- a `WebDriver` variable running Chrome, Firefox, or Edge without the test changing. And abstraction is exposing what something does while hiding how -- `WebDriver` is an interface declaring `get` and `quit`, and the browser classes supply the real work. Naming them is easy; the examples are what interviewers are listening for."

Follow-up: *"Give me an example of each from a framework you have worked on."*

Where in this book: Chapters 7, 11, and 12.

Q30 · ●●● -- What does `extends` do?

Weak answer: "It means one class inherits from another."

Strong answer: "It creates an is-a relationship: the subclass receives the parent's fields and methods and can use them as its own. That is why a test class extending `BaseTest` can call `setUp()` without containing any browser code -- the parent has it. Constructors are the exception, because they are not inherited; the child calls `super(...)` instead, and the parent constructor always completes first. Java allows only one parent class, which is why interfaces exist for the times a class needs to satisfy several contracts."

Follow-up: *"In what order do the parent and child constructors run?"*

Where in this book: Chapter 11.

Q31 · ●●● -- What is the difference between overloading and overriding?

Weak answer: "Overloading is the same name with different parameters, and overriding is redefining a parent method."

Strong answer: "Overloading means several methods sharing a name with different parameter lists, resolved by the compiler. It has nothing to do with inheritance. Overriding means a subclass replacing a parent method with the same name and the same parameters, and it is resolved at run time from the actual object. The trap is mixing them up: if I change a parameter type while intending to override, I have quietly created an overload, the parent's version still answers the call, and my code never runs. That is why I put `@Override` on every intended override -- then the compiler rejects the mistake instead of hiding it."

Follow-up: *"What exactly does* `@Override` *protect you from?"*

Where in this book: Chapter 11.

Q32 · ●●● -- Why is `WebDriver driver = new ChromeDriver();` written that way?

Weak answer: "Because `WebDriver` is the parent of `ChromeDriver`."

Strong answer: "`WebDriver` is an interface -- a contract listing what a browser must be able to do -- and `ChromeDriver` is a class that ultimately satisfies it. Declaring the interface on the left means the rest of my code only depends on the contract. The same test then runs on Chrome, Firefox, or Edge by changing which object is created, usually from a config value. Writing `ChromeDriver` on the left compiles and runs identically today, but welds every method signature to Chrome, and switching browsers becomes an edit across the framework rather than a config change."

Follow-up: *"So which method actually runs when you call* `driver.get(url)`*?"*

Where in this book: Chapter 12.

Q33 · ●●■ -- What is an interface, in plain terms?

Weak answer: "A contract that classes implement."

Strong answer: "It lists what something must be able to do, without saying how. The methods have no bodies, so an interface cannot be instantiated -- `new WebDriver()` is meaningless because there is no code behind it. A class signs the contract with `implements` and must supply every method, or it will not compile. The payoff is that code written against the interface works with every class that implements it, which is exactly how one test script drives three different browsers."

Follow-up: *"Can an interface have a constructor?"*

Where in this book: Chapter 12.

Q34 · ●●■ -- What are the access modifiers?

Weak answer: "`public`, `private`, `protected`, and default."

Strong answer: "Four levels, widening outwards. `private` is visible only inside its own class. Package-private, which you get by writing no modifier at all, adds the rest of the same package. `protected` adds subclasses even in other packages, which is what a base class uses to share with its children. And `public` is visible everywhere. My habit is to start at `private` and widen only when something genuinely needs the access -- a framework where everything is public has not made a decision, it has skipped one."

Follow-up: *"Which one do you get if you write nothing?"*

Where in this book: Chapter 13.

Q35 · ●■■ -- Why would you use an enum instead of String constants?

Weak answer: "Because it is more type-safe."

Strong answer: "Because the set of values is fixed and the compiler knows all of them. With Strings, `"PROD"`, `"prod"`, and a typo all compile, and a comparison silently takes the wrong branch -- a real risk when the value decides whether destructive tests run. With an enum, a misspelled constant fails to compile, `values()` documents the whole set in one line, and `==` is safe. An enum can also carry data and methods, so an `Environment` constant can hold its own URL rather than needing a lookup elsewhere."

Follow-up: *"What happens if you read an invalid value from a config file with* `valueOf`*?"*

Where in this book: Chapter 13.