Interview Questions: Java Basics and Syntax

Eight questions every switcher-from-manual interview asks in some form. For each: a weak answer that isn't wrong, a strong answer that names the mechanism, and where in this book to go re-run the code.

Automation Foundations · ashishautomationlabs.store

Most interview preparation fails in the same way. You read a list of questions, recognize every answer, feel ready, and then produce something woolly out loud when it counts. Recognizing an answer and being able to say it are different skills, and only one of them is tested.

So this and the five articles after it are built around a distinction rather than a list. Every question shows a **weak answer** and a **strong answer**. The weak answer is not wrong -- that's the point. It's the true-but-shallow reply that sounds like someone who has read about Java, and it's what most candidates say. The strong answer is the same knowledge, delivered the way an interviewer is actually listening for: it names the mechanism, not just the outcome; it gives a concrete example instead of staying abstract; and it ends by connecting the idea to test automation, because you're interviewing as a tester, not as a developer.

These six articles cover **Level 1 -- Making the switch**: the fundamentals you're asked when moving from manual testing into automation. They are not easy questions; they are the *first* questions, and they're where these interviews are actually won and lost. (A deeper Level 2 -- Senior SDET -- exists in the source book for once you have automation experience behind you; it isn't published here yet.)

The dots rate how often a question comes up: ●●● asked in almost every interview, ●●■ common, ●■■ occasional. To rehearse: cover the answers, read the question aloud, answer it aloud -- standing up if you can -- then compare. The gap between what you said and the strong answer is your actual preparation list, and it will be shorter than you fear. Where an answer refers to code, go back to the chapter and run it again. Every claim here is backed by a program in this series that you have already executed, and an answer you have watched run is one you can defend under pressure.

Q1 · ●●● -- Explain `public static void main(String[] args)`

Weak answer: "It is the main method where the program starts."

Strong answer: "It is the method the JVM calls to start the program, and every word is required for that to work. `public` so the JVM can reach it from outside the class. `static` so it can be called

without creating an object first -- there is nothing to create yet at that point. `void` because there is nothing to return to. And `String[] args` receives command-line arguments, which is how a framework takes a browser or environment name at run time rather than hard-coding it."

Follow-up: *"Why does it have to be static?"*

Where in this book: Chapter 8.

Q2 · ●●● -- What is the difference between a primitive type and a reference type?

Weak answer: "Primitives store values and objects store references."

Strong answer: "A primitive variable holds the value itself -- `int count = 5` puts `5` in the box. A reference variable holds an arrow pointing at an object stored elsewhere, so `String name = "priya"` puts an address in the box, not the text. That difference explains two things testers hit constantly: a reference can be `null` while a primitive cannot, and comparing references with `==` compares the arrows rather than the contents, which is why `.equals` exists."

Follow-up: *"So what happens when you pass each one into a method?"*

Where in this book: Chapter 2, and Chapter 8 for pass-by-value.

Q3 · ●●■ -- What does the `new` keyword actually do?

Weak answer: "It creates a new object."

Strong answer: "It does three things in order. It allocates memory for the object, runs the constructor to put it into a valid starting state, and returns a reference to it, which is what gets stored in your variable. That is why `new ChromeDriver()` both builds the object and starts a browser session -- the constructor is doing real work, not just filling in fields."

Follow-up: *"What happens if you never call* `new` *and use the variable anyway?"*

Where in this book: Chapter 7.

Q4 · ●●● -- What is `null`, and what is a `NullPointerException`?

Weak answer: "Null means empty, and an NPE happens when something is null."

Strong answer: "`null` means a reference variable is not pointing at any object -- the box holds no arrow. It is not zero and not an empty string. A `NullPointerException` is thrown when you try to use that reference as though it pointed somewhere, such as calling a method on it. In test automation it usually means something upstream returned nothing: a config key that was not found, an element lookup that failed, or a field that was never initialized. The fix is almost always to check for `null` at the point the value arrives, rather than at the point it explodes."

Follow-up: *"How would you avoid one when reading a value that might be missing?"*

Where in this book: Chapters 2, 3, and 6.

Q5 · ●●■ -- What does `static` mean?

Weak answer: "It means you do not need an object to call it."

Strong answer: "A static member belongs to the class rather than to any instance, so there is exactly one of it for the whole program and you call it on the class name. Utility methods are the natural use -- a `UrlBuilder.forEnv(...)` needs no state. The part worth knowing is the danger: a static field whose value changes is shared by everything, including every thread, which is exactly why a static `WebDriver` breaks under parallel execution."

Follow-up: *"So when is static the right choice?"*

Where in this book: Chapter 8, with the parallel problem in Chapter 17.

Q6 · ●●■ -- What is the difference between `==` and `.equals()`?

Weak answer: "`==` compares references and `.equals()` compares values."

Strong answer: "`==` asks whether two references point at the same object. `.equals()` asks whether two objects are meaningfully the same, and you define what that means for your own classes. For Strings this matters constantly, because two Strings with identical text can be separate objects. The trap is that `==` sometimes returns `true` by luck -- Java pools identical literals -- so a test comparing text with `==` can pass for months and then fail the day the value is built at run time instead of written literally."

Follow-up: *"Why does `==` return `true` for two identical String literals but `false` when one is built at run time?"*

Where in this book: Chapter 6.

Q7 · ●●■ -- What are a package and an import?

Weak answer: "A package groups classes and an import lets you use them."

Strong answer: "A package is a named group of classes matching a folder on disk, and it becomes part of the class's real name -- `com.shopcart.utils.UrlBuilder` rather than just `UrlBuilder`. That prevents two classes with the same short name from colliding. An import tells the compiler which one you mean so you can write the short name. In a framework, packages are how you separate tests, pages, utilities, and models so a newcomer can find anything without a guide."

Follow-up: *"Do classes in the same package need imports for each other?"*

Where in this book: Chapter 13.

Q8 · ●●■ -- How do you read a stack trace?

Weak answer: "It shows the error and where it happened."

Strong answer: "Three pieces. The first line gives the exception type and the message -- that is what went wrong and why. Below it is the call path, most recent call first. Most of those lines are inside Java's own libraries, so the one that matters is usually the deepest line naming my file, because that is where my code caused it. So I read the top line for the reason, then scan down to my own class name for the location. As a tester that habit turns a wall of red text into a defect report with steps."

Follow-up: *"The failure is in your code but the top line names a Java library class. Where do you look?"*

Where in this book: Chapter 9.