Interview Questions: Exceptions, Modern Java, and the Switch Itself

Ten closing questions: checked vs unchecked, finally, empty catch blocks, missing data files, lambdas, streams, Optional -- and the two questions every manual-to-automation switcher gets asked directly.

Automation Foundations · ashishautomationlabs.store

Q36 · ●●● -- What is the difference between checked and unchecked exceptions?

Weak answer: "Checked ones must be handled and unchecked ones do not."

Strong answer: "Checked exceptions model conditions outside the program's control -- a missing file, a refused connection -- and the compiler forces you to either catch them or declare `throws`. Unchecked ones, like `NullPointerException` or `NumberFormatException`, represent bugs in the logic, and the compiler does not force handling because the right fix is usually to correct the code. Worth knowing for automation: Selenium's exceptions are all unchecked, which is why you can call `findElement` without a `try`/`catch` around every line."

Follow-up: *"Give me an example of each that you have hit."*

Where in this book: Chapter 9.

Q37 · ●●● -- What does `finally` do, and does it always run?

Weak answer: "It runs at the end, whether or not there is an exception."

Strong answer: "It runs on both paths -- when the `try` block completed normally and when it threw -- which makes it the place for cleanup. The automation example is closing the browser: without it, a test that fails halfway leaves a browser process running, and after forty failures the machine is out of memory and later tests fail for reasons unrelated to the application. For anything I open, I now prefer try-with-resources, which closes it automatically and removes the chance of forgetting."

Follow-up: *"What is try-with-resources and when can you use it?"*

Where in this book: Chapter 9, with try-with-resources in Chapter 15.

Q38 · ●●● -- What is wrong with an empty catch block?

Weak answer: "It hides the error."

Strong answer: "It destroys the evidence at exactly the moment it was available. The failure happened, nothing was recorded, and the run continues as though everything worked. In a data-driven suite that is the worst possible outcome: forty rows fail to parse, forty rows are skipped in silence, and the suite reports PASS. Catching is not the problem -- catching narrowly, reporting what was skipped, and failing the run when the skip count is above zero is the discipline. A suite that hides failures is worse than no suite, because people trust it."

Follow-up: *"So what should the catch block do instead?"*

Where in this book: Chapter 9.

Q39 · ●●■ -- How would you handle bad rows in a data-driven test?

Weak answer: "Wrap it in a `try`/`catch` so the run does not stop."

Strong answer: "Put the `try` inside the loop, wrapping one row, so a bad row is reported and the remaining rows still run -- placing it outside the loop means the first failure abandons everything after it. Catch the specific exception I expect rather than `Exception`, log the row number and the offending value, and count the failures. Then fail the suite at the end if that count is above zero. The run survives dirty data and still tells the truth, which is the combination that matters."

Follow-up: *"Why catch the specific type rather than* `Exception`*?"*

Where in this book: Chapter 9.

Q40 · ●●■ -- What happens if your test data file is missing?

Weak answer: "You should handle the exception so the suite does not crash."

Strong answer: "It has to fail the run, loudly, with the file path and the working directory in the message. The tempting alternative is to catch it and return an empty list so nothing crashes. That is the most dangerous thing you can do: zero rows means zero tests run, and the suite reports green. A pipeline that checks nothing and shows a tick is worse than one showing a cross, because a cross gets investigated. A missing data file is a setup failure, not a valid state."

Follow-up: *"Why include the working directory in the message?"*

Where in this book: Chapter 15.

Q41 · ●●■ -- What is a lambda?

Weak answer: "A short way to write a function."

Strong answer: "It is a block of behavior you can pass to a method as a value, written as parameters, an arrow, and a body -- `r -> r.equals("FAIL")`. It fits any interface with exactly one abstract method, and it supplies that method's body. The reason it matters for automation is that

an explicit wait is precisely this: you hand a condition to a method, and the method re-evaluates it until it becomes true. Once you see that, waits stop being magic."

Follow-up: *"What kind of interface can a lambda be assigned to?"*

Where in this book: Chapter 16.

Q42 · ●●■ -- What does a stream do?

Weak answer: "It processes collections in a functional way."

Strong answer: "It is a pipeline over a collection: start with `stream()`, chain steps like `filter` and `map`, and finish with something that produces a result, such as `collect` or `count`. It replaces the loop-plus-if-plus-accumulator shape with one readable statement -- filtering failures out of three hundred results, for instance. The catch worth knowing is that intermediate steps do nothing on their own: without a terminal operation the pipeline never runs, silently."

Follow-up: *"What is the difference between an intermediate and a terminal operation?"*

Where in this book: Chapter 16.

Q43 · ●■■ -- What problem does `Optional` solve?

Weak answer: "It avoids `null`."

Strong answer: "It makes a possibly-absent value visible in the type, so the person writing the code is asked 'what if this is missing?' at the moment they write it, rather than at 2 a.m. when a pipeline fails. A search that may find nothing returns an `Optional`, and I read it with `orElse` to supply a fallback or `ifPresent` to act only when there is a value. Calling `get()` without checking simply swaps a `NullPointerException` for a `NoSuchElementException`, which is `Optional` used as decoration."

Follow-up: *"Where would you use it in a framework?"*

Where in this book: Chapter 16.

The two questions every switcher is asked

Q44 · ●●● -- You come from a manual testing background. Why should we hire you for an automation role?

Weak answer: "I have learned Java and Selenium and I am a fast learner."

Strong answer: "Because the hard part of automation is deciding what to check and recognizing when a result is lying to you, and that is what I have been doing for years. I know which flows

break, which data causes trouble, and what a suspicious pass looks like -- which is exactly the judgment that keeps a suite trustworthy. What I have added is the engineering: I write Java, I have built page objects and data-driven tests, and I understand why a static driver breaks in parallel. Someone who can code but cannot design a test case writes automation that runs and proves nothing."

Follow-up: *"Tell me about a time your testing instinct caught something automation missed."*

Where in this book: the whole book, and the Bug Hunts in particular.

Q45 · ●●■ -- How did you learn Java?

Weak answer: "I did an online course and some tutorials."

Strong answer: "I worked through the fundamentals properly rather than copying framework code -- types, collections, objects, exceptions, and the OOP concepts -- and I ran every example rather than reading it. The habit I built that helped most was predicting a program's output before running it, then explaining any difference, which is the same expected-versus-actual discipline I already used in testing. That is why I can read framework code now instead of pattern-matching it, and why I can explain what `WebDriver driver = new ChromeDriver();` is actually doing."

Follow-up: *"What was the hardest concept, and how did you get past it?"*

Where in this book: the predict-then-run habit from Chapter 1 onwards.

That's Level 1 in full -- forty-five questions, six articles, one weak answer and one strong answer each. It closes the loop this whole series opened: Chapter 1 asked you to trust a line you couldn't read yet, and Q45 above is the same habit that got you here -- predict, run, explain the difference. Level 2 exists for when automation experience is behind you rather than ahead of you; that's a different day's work.