When a Loop Is Still Better

Streams reward small, named steps and punish long anonymous ones. A one-line pipeline a colleague must decode at midnight is worse than four honest lines of loop.

Automation Foundations · ashishautomationlabs.store

Streams are not automatically an improvement. Prefer a plain loop when the body does several unrelated things, or when you need the index. Prefer it too when you need to break early in a way that is awkward to express, or when the stream version needs a comment to explain itself. A one-line stream that a colleague must decode at midnight is worse than four honest lines of loop. The goal is the clearest expression of the intent, and sometimes that is the loop you already know how to write.

That distinction is exactly what a code review caught once. A results summary was built from one chained stream running to eleven lines, with three nested lambdas. It's one statement and no mutable variables -- cleaner than a loop with three counters, on the surface. But read twice, nobody could say what it returns when the list is empty. That's not clean, that's compressed. The fix isn't to abandon the stream and go back to loops -- it's to split it: pull each lambda into a named method, so the pipeline reads as filter-by-failed, map-to-id, collect. The stream stays, and now the names do the explaining. Streams reward small, named steps and punish long anonymous ones. If a pipeline needs a comment to explain what it does, extract the lambdas into methods with names, or write the loop. Clever is not the goal; the next reader is.

One more question worth settling directly: are streams faster than loops? Not meaningfully, for the collection sizes a test suite deals with. Choose on readability. Streams can run in parallel with one method call, but that brings the thread-safety problems of Chapter 17, so don't reach for it casually.

This chapter is the modern half of a framework. Explicit waits take a condition as a lambda, which is the `waitUntil` you built earlier. Filtering `List<WebElement>` down to the visible or enabled ones is `stream().filter(...)`. Turning a list of elements into a list of their text is `map(WebElement::getText)`. Building a run summary is `filter` on failed results plus `collect` or `count`. And `Optional` shows up wherever a lookup may find nothing -- a missing config key, an element that is not on the page, a row that does not match.

Every rule in this chapter has a silent failure mode when ignored. The closing bug hunt collects three of them in one small program.