Throwing Your Own: Checked and Unchecked

Throwing your own exception with an expected-versus-actual message is the core of every assertion library. Checked vs. unchecked is just outside-your-control versus a bug in your own logic.

Automation Foundations · ashishautomationlabs.store

You are not limited to catching Java's exceptions. When your code detects a condition it cannot sensibly continue past, it should throw one -- and for a tester this is exactly how a check reports a genuine failure:

```
static void checkStatus(int code) {
    if (code != 200) {
        throw new IllegalStateException("Expected 200 but got " + code);
    }
    System.out.println("Status OK");
}

Status OK
Test failed: Expected 200 but got 503
```

Read the message carefully: `Expected 200 but got 503`. That is the expected-versus-actual framing you have used since your very first manual test case, now produced automatically. Every assertion method in every test framework does precisely this -- checks a condition and throws an exception carrying an expected-versus-actual message when it fails. You have just written the core of an assertion library. The rule for a good exception message is the rule for a good defect report: include the values. "Status check failed" is useless. "Expected 200 but got 503" can be triaged without reproducing anything.

Checked vs. unchecked

Java has two families of exception, and the difference is a common interview question with a simple answer. **Unchecked** exceptions are the ones you have met so far: `NullPointerException`, `ArrayIndexOutOfBoundsException`, `NumberFormatException`, `IllegalStateException`. They represent bugs in the logic, and the compiler does not force you to handle them. You could catch them, but the better fix is usually to correct the code so they cannot happen.

Checked exceptions represent conditions outside your program's control -- a missing file, a refused network connection. The compiler insists that you either catch them or declare that your method passes them on with `throws`. That is why file and network code in Java always seems to come wrapped in `try`.

You can define your own, which is what real frameworks do to describe their own failures clearly:

```
class TestDataException extends Exception {
    TestDataException(String message) {
        super(message);
    }
}
static void loadConfig(String file) throws TestDataException {
    throw new TestDataException("Config file not found: " + file);
}

Setup failed: Config file not found: missing.properties
```

The class extends `Exception`, which makes it checked -- so `loadConfig` must declare `throws TestDataException`, and callers must handle it. `super(message)` passes the message up to the built-in machinery so `getMessage()` works. A custom exception named for your domain makes a failure self-describing: `TestDataException` in a stack trace tells the next engineer where to look before they read a single line of your code.

Exceptions are the vocabulary of a failing test run in any framework you'll use next. A `NoSuchElementException` means the locator did not match anything on the page. A `TimeoutException` means the wait expired before the condition became true. A `StaleElementReferenceException` means the page changed under you after the element was found. Every one of these arrives as an exception object with a type, a message, and a stack trace -- the same three pieces you learned to read earlier in this chapter. Teardown lives in a `finally` or its annotation equivalent so the browser always closes. And your assertions throw exceptions with expected-versus-actual messages, exactly like `checkStatus` above. Learning a framework's assertions next will not mean learning a new idea, only new exception names.

Every tool in this chapter -- catching, ordering, `finally`, throwing your own -- only helps if it's used with discipline. The next article is what happens when it isn't.