The Shared Driver and the Lost Increment

The last Bug Hunt in the book. A DriverManager that looks like the real thing, with four defects -- one traceable to a chapter you've already read.

Automation Foundations · ashishautomationlabs.store

This is the last Bug Hunt, and it revisits the whole book. This `DriverManager` looks like the real thing and has four defects. Read it before reading on.

```
public class DriverManager {
    public static String driver;
    public static int activeSessions = 0;
    public static void startDriver(String session) {
        driver = session;
        activeSessions++;
    }
    public static String getDriver() {
        return driver;
    }
    public static void quitDriver() {
        driver = null;
    }
}
```

Four defects, each traceable to a chapter you have read.

`public static String driver` is the shared mutable field from this chapter. Under parallel execution two tests overwrite each other's driver, and one drives the other's browser. It must be a `ThreadLocal`.

`public static int activeSessions` is shared mutable state too, and `activeSessions++` is not one instruction -- it reads, adds, and writes. Two threads can read the same value before either writes, so increments are silently lost. The count is wrong, and nothing reports it.

The fields are `public`, so any test can write `DriverManager.driver = null` directly and bypass the methods entirely. Chapter 7's encapsulation applies: private fields with methods that control access.

`quitDriver()` sets the reference to `null` but there is no `remove()`. Once the field becomes a `ThreadLocal`, forgetting `remove()` retains the value on a pooled thread and hands it to the next test.

The corrected shape is short:

```
public class DriverManager {
    private static final ThreadLocal<String> driver = new ThreadLocal<>();
    public static void startDriver(String session) {
        driver.set(session);
    }
    public static String getDriver() {
        return driver.get();
    }
    public static void quitDriver() {
        driver.remove();
    }
}
```

Private field, one holder shared, one value per thread, and a `remove()` in the method teardown already calls. That is the whole fix, and it is the class most professional frameworks have.

Checkpoint

- What does the compiler do with `List<String>` that it cannot do with a

raw `List`?

- What does `<T>` declare in a generic method signature, and where does

it go?

- In one sentence, what is type erasure?
- Why does a static driver break under parallel execution but pass when

run one test at a time?

- Name the three `ThreadLocal` methods and say when each is called in a

test lifecycle.

- Give the two ways to make state safe under threads.

You can write type-safe code and explain why the angle brackets you have used since Chapter 5 exist at all. You can write a generic method that serves any type without casting, and answer the type-erasure question that follows it in interviews. You understand what a thread is, why shared mutable state is the root of parallel failures, and why a suite can pass for two years and then break the week it goes parallel. And you can fix it with a `ThreadLocal` behind an accessor, remembering the `remove()` that most people forget.

The whole book, in one place

This is the last chapter of the core book, so the summary is larger than usual.

You started unable to read `public static void main(String[] args)`. You can now decode every word of it and explain why each part must be what it is. You can store and type data, judge it with conditions, repeat with loops, and hold it in arrays, lists, maps, and sets. You can design your own classes with real encapsulation, define what equality means for them, and organize them into packages with access levels you chose on purpose. You can handle failure without hiding it, parse the messy text a real system returns, and read your test data from files that other people maintain. You can express filtering and transformation with streams, and you know that Selenium's explicit wait is a lambda handed to a loop. You can read the type hierarchy behind `WebDriver driver = new ChromeDriver();` and say precisely why it is written that way.

Along the way you met seventeen Bug Hunts, and they all taught one lesson from different angles: a program that runs is not a program that is correct. A `=` that should have been `==`. An `else if` chain in the wrong order. A test comparing text with `==` that passed by luck. A constructor missing `this`. An empty catch block reporting PASS over skipped rows. A missing data file producing a green suite that tested nothing. That instinct -- to distrust a passing result until you know why it passed -- is the difference between someone who writes automation and someone who can be trusted with it.

What is left is not Core Java. It is Selenium, TestNG or JUnit, REST Assured, build tools, and CI. Every one of them will now read as a library written in a language you know, rather than as magic to be copied. That was the promise on the first page, and it is kept.

Go and build the framework.