ThreadLocal: One Value Per Thread

ThreadLocal keeps the field shared but gives each thread its own value inside it. Skipping remove() in teardown hands the next pooled test a dead reference and a slow memory leak.

Automation Foundations · ashishautomationlabs.store

`ThreadLocal<T>` is a container that holds a separate value for each thread. Every thread that calls `get()` sees only what it put there with `set()`. Change the field and nothing else:

```
public class TH2 {
    static final ThreadLocal<String> driver = new ThreadLocal<>();
    static void runTest(String name, String session) {
        driver.set(session);
        System.out.println(name + " set driver to   : " + session);
        sleep(150);
        System.out.println(name + " now sees driver : " + driver.get());
        driver.remove();
    }
}
```

```
TestA set driver to   : chrome-session-1
TestB set driver to   : chrome-session-2
TestA now sees driver : chrome-session-1
TestB now sees driver : chrome-session-2
main thread sees : null
```

Same structure, same timing, correct result. TestA sees `session-1` and TestB sees `session-2`, because each thread now has its own value in its own compartment. Note the field is still `static`, and that is on purpose. You want one shared *holder* that every test class can reach. What is no longer shared is the value inside it.

The last line deserves its own attention: the main thread sees `null`, because it never called `set()`. A `ThreadLocal` gives nothing to a thread that has not put something in, which is why a driver retrieved on the wrong thread comes back as `null` rather than as somebody else's browser.

Three methods, and one discipline. `set(value)` stores this thread's value. `get()` retrieves it. And `remove()` clears it, which you must call when the test finishes. What actually happens if you forget `remove()`? Two things, both bad and both slow to appear. Test frameworks reuse threads from a pool, so the next test on that thread inherits the old value -- often a closed browser. And the values stay held for the life of those threads, so a long run steadily consumes memory. Put `remove()` in teardown and neither happens; in a real framework the `set` belongs in setup and the `remove` belongs in teardown, alongside the browser `quit` from Chapter 9.

`ThreadLocal` fixed one field. The idea behind why it worked applies to every piece of shared state in a framework, not just the driver.