protected, and the Object Behind Every Class

protected is the access level a base class needs to share with its children without opening itself to everyone. And every class you've ever written already extends Object, whether you typed extends or not.

Automation Foundations · ashishautomationlabs.store

Chapter 7 gave you `private`, which hides a field from everything outside its class -- including subclasses. That is often too strict for a base class, whose entire purpose is to be built upon. `protected` is the middle setting: available to this class and to any subclass, but not to unrelated code.

```
class BaseTest {
    protected String runId = "RUN-8891";
    private String secret = "internal-token";
    protected String getSecret() {
        return secret;
    }
}
class CartTest extends BaseTest {
    void report() {
        System.out.println("Run id : " + runId);
        System.out.println("Secret : " + getSecret());
    }
}
```

```
Run id : RUN-8891
Secret : internal-token
```

`CartTest` reads `runId` directly because it is `protected`. It cannot touch `secret` directly at all -- that field is `private`, and the only way in is the `protected` method the parent chose to expose. That is the parent controlling exactly what it shares, which is the point. A base class with everything public has no boundary; a base class with everything private cannot be extended usefully. `protected` is the setting that says "for my subclasses."

Every class already extends Object

Here is a fact that explains something from Chapter 7. Write a class with no `extends` at all, and Java quietly gives it one anyway: every class in Java inherits from a class called `Object`.

That is why `toString` and `equals` existed on your classes before you wrote them -- you were not inventing those methods in Chapter 7, you were overriding them. Watch the inherited version:

```
class Plain {
}
class Nice {
    @Override
    public String toString() {
        return "Nice[a readable object]";
    }
}

Default toString : Plain@b7f23d9
Overridden       : Nice[a readable object]
```

The first line is `Object`'s own `toString`: the class name, an `@`, and a hexadecimal number derived from the object. That number differs on every run, so do not expect to reproduce it exactly -- which is itself the lesson, because it is useless in a test report. The second line is what a five-line override buys you. So the inheritance rules you learned in this chapter were already running underneath Chapter 7. `Object` sits at the root of every hierarchy, supplying `toString`, `equals`, and `hashCode`, and your job is to override the ones that matter for your type.

How deep should an inheritance chain go?

Shallow. Two levels is common and comfortable; four is a warning sign. A framework has `BaseTest`, then `BaseWebTest extends BaseTest`, then `BaseCheckoutTest extends BaseWebTest`, and a new `GuestCheckoutTest extends` that. Four levels, and nothing in any one of them is duplicated -- every level adds something real. But open `GuestCheckoutTest` and try to say where `setUp` actually comes from. You cannot, not without opening four files. The IDE will tell you, and the person debugging a flaky run at midnight still has to hold four classes in their head. Most deep chains like this are sharing helper code, not describing what the test genuinely _is_ -- and a helper can be a utility object the test holds, rather than a parent it inherits from. Inherit when the child genuinely is a kind of the parent; reach for a helper object when you only want to share code. A deep chain is usually code sharing wearing an inheritance costume, and it costs the next reader more than it saved you.

Inheritance is the backbone of a framework's structure precisely because of what this chapter covered. `BaseTest` holds the driver, the config, and the setup and teardown that every test needs, so a test class carries only its own checks. `BasePage` holds the driver reference and shared element helpers, and each Page Object extends it. A `BaseApiTest` sets up the REST client the same way. The `@Override` habit matters here more than anywhere: an overridden setup that silently does not override is a test running with the wrong preconditions, and that produces failures nobody can reproduce.

Everything in this chapter has been about one class built on another. There is one more relationship left in Java's toolkit, and it looks similar on the surface but works completely differently -- it's what makes `WebDriver` not a class at all.