The Overload That Never Overrode

This subclass wrote three things meant to replace the parent's behavior. Not one of them took effect -- and the object is a CheckoutTest in every single line.

Automation Foundations · ashishautomationlabs.store

This program compiles and runs, and prints three lines. The subclass wrote three things, and not one of them took effect. Read first, then run.

```
class BaseTest {
    String environment = "UAT-2";
    void setUp() {
        System.out.println("[BASE] Opening browser on " + environment);
    }
    void logResult(String result) {
        System.out.println("[BASE] Result: " + result);
    }
}
class CheckoutTest extends BaseTest {
    String environment = "PROD";
    void setUp(String browser) {
        System.out.println("[CHECKOUT] Opening " + browser);
    }
    void logResult(Object result) {
        System.out.println("[CHECKOUT] Result: " + result);
    }
}

BaseTest t = new CheckoutTest();
t.setUp();
t.logResult("PASS");
System.out.println("Env used: " + t.environment);
```

```
[BASE] Opening browser on UAT-2
[BASE] Result: PASS
Env used: UAT-2
```

Three defects, and the object is a `CheckoutTest` in every one of them.

`setUp(String browser)` takes a parameter the parent's version does not. That is an overload, not an override, so the call to `t.setUp()` found only the parent's method. The checkout-specific setup never ran.

`logResult(Object result)` looks closer, but `Object` is not `String`, so this too is an overload rather than an override. The variable is declared `BaseTest`, so the compiler chose the parent's `logResult(String)` and the subclass version was never a candidate.

`environment = "PROD"` did not override the parent's field -- fields are not overridden at all. Declaring a field with the same name in a subclass _hides_ it, and which one you get depends on the declared type of the variable. The variable is a `BaseTest`, so you read the parent's `"UAT-2"`. A checkout test that believes it is running against production is reporting on UAT.

Every one of these is cured by the same two habits. Write `@Override` on any method you intend to override, and the first two defects become compile errors instead of silent ones. And never redeclare a parent's field in a subclass -- set the inherited field's value in a constructor instead.

Checkpoint

- What does a subclass receive from its superclass, and what does it not

receive?

- In what order do a parent and child constructor run, and what forces that

order?

- Give the two-part definition that separates overriding from overloading.
- What does `@Override` do that helps you, and what happens without it?
- What is `protected`, and why does a base class need it?
- Which class sits at the root of every Java hierarchy, and name two methods it

gives you.

Reliable questions, all from this chapter: the difference between overriding and overloading (same signature in a subclass versus a different parameter list; inheritance versus not); what `super` does (calls the parent constructor, which must be the first statement, or a parent version of an overridden method); whether you can override a static method (no -- it is hidden, not overridden); what `protected` means (visible to this class and its subclasses); which class is the parent of every Java class (`Object`, which is where `toString`, `equals`, and `hashCode` come from); and why Java does not allow multiple inheritance of classes (to avoid inheriting conflicting implementations -- interfaces solve the same need safely).

You can read the first line of any test class in any framework and know exactly what it means. You can write a base class holding shared setup, extend it, and call `super` to run the parent's constructor or its version of a method you replaced. You can override behavior on purpose, protect yourself with `@Override`, and tell overriding from overloading well enough to explain the difference under interview pressure. You can choose `protected` where a base class needs to share with its children. And you know that `Object` has been above your classes all along, which is why Chapter 7's `toString` and `equals` were overrides rather than inventions.

There is one line left in this book that you still cannot fully read, and it is the most famous line in Java test automation:

```
WebDriver driver = new ChromeDriver();
```

You now know enough to be suspicious of it. `WebDriver` is the declared type and `ChromeDriver` is the object, which is the polymorphism you just met. But `WebDriver` is not a class, and `ChromeDriver` does not extend it the way `LoginTest` extends `BaseTest`. Something else is going on, and it is the reason the same test script runs on Chrome, Firefox, and Edge without a single change. That is the interface, and it is Chapter 12.