Overriding, and Your First Sight of Polymorphism

Java picks a method from the object's actual class, not the variable's declared type. That's polymorphism -- and @Override is what turns a broken attempt at it into a compile error instead of a silent bug.

Automation Foundations · ashishautomationlabs.store

A subclass can replace a method it inherited. Give the method the same name and the same parameters, and mark it `@Override`:

```
class BaseTest {
    void setUp() {
        System.out.println("[BASE] Opening browser");
    }
}
class ApiTest extends BaseTest {
    @Override
    void setUp() {
        System.out.println("[API ] No browser needed - starting REST client");
    }
}

BaseTest t = new ApiTest();
t.setUp();

[API ] No browser needed - starting REST client
```

Look closely at those two lines of calling code, because something important happened. The variable is declared as `BaseTest`, but the object created was an `ApiTest` -- and the method that ran was `ApiTest`'s. Java chooses the method based on the actual object, not the declared type of the variable. That behavior is called **polymorphism**, it is the engine of the whole next chapter, and this is your first sight of it.

`@Override` deserves a moment. It is optional, and you should never omit it. It tells the compiler "I intend to replace a parent method" -- and if you have made a mistake, so that no parent method matches, the compiler refuses to build. Without it, that same mistake compiles happily and produces a silent bug -- which is exactly what the next article is about. One annotation, an entire class of defect prevented.

> *Given the* `BaseTest`/`ApiTest` *above: in* `BaseTest t = new ApiTest();`*, which class's* `setUp` *runs? What must be the first statement in a subclass constructor that calls its parent? What does* `@Override` *protect you from?*

>

> *Answer -- `ApiTest`'s, because Java picks the method from the actual object. `super(...)`. And `@Override` protects you from a method you meant as an override that matches no parent method -- the compiler rejects it instead of silently creating a new one.*

Overriding looks simple in an example this small. The trap that catches almost everyone shows up the moment the replacement method's signature drifts even slightly from the original.