# The instanceof Ladder That Broke Polymorphism

*This program compiles and runs, and prints two lines instead of three. One whole browser is missing from the report -- and both defects quietly convert a flexible design back into a rigid one.*

Automation Foundations · ashishautomationlabs.store

This program compiles and runs. It prints two lines, and one whole browser is missing from the report. Read first, then run.

```
public class BugHunt12 {
    public static void main(String[] args) {
        report(new ChromeBrowser());
        report(new FirefoxBrowser());
        ChromeBrowser only = new ChromeBrowser();
        only.open("https://uat.shopcart.internal");
    }
    static void report(Browser b) {
        if (b instanceof ChromeBrowser) {
            System.out.println("Reporting for Chrome");
        }
    }
}
```

```
Reporting for Chrome
[Chrome ] navigating to https://uat.shopcart.internal
```

Two defects, and both defeat the whole point of the chapter.

`report` takes a `Browser`, which is correct -- and then immediately throws that away by asking `instanceof ChromeBrowser`. The Firefox call produced no output whatsoever. No error, no warning, no report line. Every new browser added to the suite will be silently skipped until someone remembers to extend the chain. This is the classic anti-pattern: an `instanceof` ladder inside a method that already had polymorphism available. The fix is to put the behavior on the contract -- add a `report()` method to `Browser` -- and let each class supply its own version. Then a new browser cannot be forgotten, because the compiler will not let it exist without one.

`ChromeBrowser only = new ChromeBrowser();` declares the concrete type. It runs correctly today, so nothing looks wrong. The cost is invisible: that variable, and everything it is passed to, is now welded to Chrome. Declaring `Browser only` instead costs one word and keeps the suite portable.

Neither line is an error. Both quietly convert a flexible design back into a rigid one, which is exactly how frameworks decay.

## Checkpoint

- What does an interface contain, and why can it not be instantiated?
- Explain what `Browser b = new ChromeBrowser();` does in terms of declared type and actual object.
- What is runtime polymorphism, and which of the two types decides the method that runs?
- Give the two questions that decide between an interface and an abstract class.
- Why is `List<String> x = new ArrayList<>();` better style than `ArrayList<String> x = new ArrayList<>();`?
- Name the four levels of Selenium's hierarchy from `SearchContext` down to `ChromeDriver`, and say which are interfaces.

This chapter is the highest-value interview material in the book. Why is `WebDriver` an interface? (So any browser class can satisfy one contract, and a test written against it runs on all of them.) What is the difference between an abstract class and an interface? (One superclass versus many interfaces; abstract classes hold fields, constructors, and finished methods.) What is runtime polymorphism? (Method selection from the actual object at the moment of the call.) Why write `List` on the left and `ArrayList` on the right? (Programming to an interface -- the implementation can change without touching the code that uses it.) Can an interface have a constructor? (No.) What happens if a class does not implement every method of its interface? (It fails to compile, unless it is declared abstract.)

You can read and explain the most famous line in Java test automation, and you can defend it in an interview. You can define an interface as a contract, implement it in several classes, and write code that works with all of them without naming any. You understand upcasting, and you know that the declared type limits what you may call while the object decides what runs. You can choose between an interface and an abstract class with a reason rather than a habit. And the `List` on the left of your collection declarations finally makes sense.

That completes the object-oriented core. Encapsulation came in Chapter 7 with `private` fields, inheritance in Chapter 11, and abstraction and polymorphism here. Those four ideas are what interviewers mean by "OOP concepts," and you have met every one of them through code you ran yourself.

Your framework can now bend at the joints. What it cannot do yet is find anything. Twelve test classes, some page objects, a handful of utilities, and everything sits in one folder with no structure and no rules about what may see what. A growing suite needs organization before it needs more features -- packages, access levels, and constants that are not scattered strings. That is Chapter 13.