Default Methods, and the Abstract Class Choice

A default method gives every implementer a body for free, but an interface still can't hold fields or a constructor -- which is exactly the line that separates it from an abstract class.

Automation Foundations · ashishautomationlabs.store

Interfaces originally held signatures only. Modern Java also allows a **default method**: a signature with a body, which every implementing class receives for free and may override.

```
interface Browser {
    void open(String url);
    default void openHome() {
        System.out.println("[default] using the shared home URL");
        open("https://uat.shopcart.internal/home");
    }
}
```

```
[default] using the shared home URL
[Firefox] navigating to https://uat.shopcart.internal/home
```

`FirefoxBrowser` never wrote `openHome`, yet it has one. Note what the default method does: it calls `open`, which the implementing class did provide. A default method can build on the contract's own promises. Defaults exist mainly so a widely-used interface can gain a method without breaking every class that already implements it. Use them for convenience behavior that genuinely suits all implementers, not as a way to smuggle shared state into an interface -- an interface still holds no fields.

Can an interface have a constructor? No -- constructors build objects, and an interface is never instantiated. This is a common interview question, and the reason is the answer.

Does that make an interface with defaults the same as an abstract class? No. An interface still cannot hold instance fields or a constructor, and a class can implement many interfaces while extending only one. Defaults added convenience, not state.

Interface or abstract class?

Chapter 11 gave you inheritance; this chapter gives you contracts. An **abstract class** sits between the two. It cannot be instantiated, it may hold fields and finished methods, and it may also declare abstract methods that subclasses must supply:

```
abstract class AbstractTest {
```

```
    void run() {
        setUp();
        check();
        System.out.println("[BASE] done");
    }
    void setUp() {
        System.out.println("[BASE] shared setup");
    }
    abstract void check();
}
class CheckoutTest extends AbstractTest {
    @Override
    void check() {
        System.out.println("[TEST] checking the cart total");
    }
}

[BASE] shared setup
[TEST] checking the cart total
[BASE] done
```

`AbstractTest` fixed the order of a test run and left one hole for the subclass to fill. That is a genuinely useful base class.

The choice comes down to two questions. How many can you have? A class extends one parent but implements any number of interfaces -- so if a type needs several contracts, interfaces are the only option. Do you need shared state? An abstract class can hold fields and constructors; an interface cannot.

The practical rule for a framework: use an interface to describe a capability that unrelated types might have, and an abstract class to share real setup among types that truly are related. `BaseTest` is an abstract class. `Browser` is an interface.

Everything so far has been about designing the contract. The chapter's biggest payoff is seeing that design already at work, in a hierarchy you use every day without having read it as ordinary Java.