Interface

Defining behavior contracts in Java; default/static/private methods and sealed interfaces.

Intermediate 50 min

Last updated: November 9, 2026
Read online: https://www.learnforgex.com/en/topics/interface

Interface

In Java, an **interface** is a contract that defines what a type *can do*, without saying *how* it does it. When a class implements an interface, it commits to fulfilling that contract — calling code can then rely purely on the contract, without ever knowing the real class of the object it's holding. In this lesson we'll go end-to-end: starting from plain abstract methods, through Java 8's default/static methods and Java 9's private methods, all the way to Java 17's sealed interfaces.

What Is an Interface?

At its simplest, an interface is a type definition made of bodyless method signatures — it says "an object of this type must have these methods," but leaves the methods' insides empty:

```
interface Payable {
    double calculatePayment();
}

class Invoice implements Payable {
    @Override
    public double calculatePayment() {
        return 199.90;
    }
}
```

By writing `implements Payable`, `Invoice` is saying "I fulfill this contract" — if it doesn't actually define `calculatePayment()`, the code won't compile. In return, any code working through a `Payable`-typed variable never has to know whether the object behind it is an `Invoice` or some other `Payable` implementation that hasn't even been written yet.

Why Does It Exist?

The core problem an interface solves is **decoupling** — making code depend on a contract rather than a concrete class. Saying "I expect a `List`" instead of "I expect an `ArrayList`" lets the calling code switch to `LinkedList` tomorrow, or even use a mock implementation in a test, without changing anything else — this principle is usually called *"program to an interface, not an implementation."*

The second major reason is working around Java's single-inheritance restriction. A class can `extends` only **one** class — but in the real world, an object can play several "roles" at once: a `Duck` can both fly and swim, yet it can't inherit from a "Flying Things" class and a "Swimming Things" class at the same time. Interfaces solve this by letting a class inherit **state from only one place**, while inheriting as many **contracts** as it likes (we'll see this concretely in "Implementing Multiple Interfaces").

History

Interfaces have existed since Java's very first release (JDK 1.0, 1996) — but early on they had a strict rule: they could only contain abstract methods and constants, never a method body. Over the years this caused a real practical problem: adding even a single new method to an interface implemented by thousands of third-party classes — like `java.util.List` — would break compilation for **every** class that didn't implement that method; the JDK itself couldn't add new methods to the collection interfaces for years because of this.

Java 8 (2014) solved this "interface evolution" problem with **default methods** — an interface could now define a method with a body, and every existing implementer would automatically inherit that behavior without lifting a finger (`forEach` and `stream()` were able to be added to `Collection` after the fact precisely because of this). The same release added **static methods**. Java 9 (2017) added **private methods** to make it easier to share code between default methods. Java 17 (2021) introduced **sealed interfaces**, which let you explicitly restrict which types are allowed to implement an interface — the Java 21 this project targets carries all of these layers together, and we'll work through each one in turn.

Writing Your First Interface

An interface is declared with the `interface` keyword instead of `class`; a class that implements it uses `implements` and must provide an `@Override` for every one of the interface's abstract methods:

```
interface Payable {
    double calculatePayment();
}

class Invoice implements Payable {
    private final double amount;

    Invoice(double amount) {
        this.amount = amount;
    }

    @Override
    public double calculatePayment() {
        return amount;
    }
}

class FirstInterfaceDemo {
    public static void main(String[] args) {
        Payable invoice = new Invoice(199.90);
        System.out.println(invoice.calculatePayment()); // 199.9
    }
}
```

`Payable` says nothing beyond a single method signature — how `Invoice` computes that value is entirely its own business. Notice the line `Payable invoice = new Invoice(...)` in `FirstInterfaceDemo`: the variable's type is `Payable`, not `Invoice` — which means switching to a different `Payable` implementation later won't require changing a single line of calling code.

> If a class writes `implements` but leaves one of the interface's methods unimplemented, the compiler fails immediately — this is one of an interface's most valuable properties: breaking the contract is caught at **compile time**, not at runtime.

Abstract Methods

A bodyless method inside an interface is implicitly `public abstract` — even if you never type those two words, they still apply:

```
// Both declarations below are 100% equivalent — an interface method with no
// body is implicitly `public abstract`, whether you write those keywords or not.
interface Greeter {
    void greet(String name);
}

interface GreeterExplicit {
    public abstract void greet(String name);
}

class EnglishGreeter implements Greeter {
    // Must be `public` here — you can never REDUCE visibility when overriding,
    // and the interface method is implicitly public.
    @Override
    public void greet(String name) {
        System.out.println("Hello, " + name + "!");
    }
}

class AbstractMethodExample {
    public static void main(String[] args) {
        Greeter greeter = new EnglishGreeter();
        greeter.greet("Ada"); // Hello, Ada!
    }
}
```

Expected Output

```
Hello, Ada!
```

`Greeter` and `GreeterExplicit` define exactly the same thing. This has an important practical consequence: when a class implements the method, it can never **narrow** the access modifier — since the interface method is implicitly `public`, trying to make `EnglishGreeter.greet(...)` `protected` or package-private is a compile error; an overriding method must always be at least as accessible as the interface method it implements.

> You don't need to write `abstract` for an interface method to be bodyless, but if you want to give that method a body **without** using `default` / `static` / `private`, it's never allowed — a plain method is either fully abstract or marked with one of the three special keywords we'll cover in the sections below.

Constant Fields

Fields declared in an interface are implicitly `public static final` too — meaning they are both constant and reachable directly as `InterfaceName.FIELD`, without needing an instance:

```
// Fields declared in an interface are implicitly `public static final` —
// writing those modifiers yourself is redundant (though legal).
interface PhysicsConstants {
    double GRAVITY = 9.81;          // really: public static final double GRAVITY = 9.81;
    double SPEED_OF_LIGHT = 299_792_458;
}

class FreeFall implements PhysicsConstants {
    double distanceAfter(double seconds) {
        return 0.5 * GRAVITY * seconds * seconds;
    }
}

class InterfaceConstantsExample {
    public static void main(String[] args) {
        System.out.println(PhysicsConstants.GRAVITY);           // 9.81 — accessible
without an instance
        System.out.println(new FreeFall().distanceAfter(2));    // 19.62
    }
}
```

By implementing `PhysicsConstants`, `FreeFall` can reach `GRAVITY` as if it were its own field — but this isn't **inheritance** in the state sense, because the field is already `static`; each implementer doesn't get its own copy, they all point to the exact same constant.

> Forgetting that an interface field is `final` and trying to assign it a new value later (`PhysicsConstants.GRAVITY = 10;`) is a compile error. An interface is never a suitable place to hold **mutable state** shared across implementations — use a class's `static` field for that instead.

Implementing an Interface

A class connects to an interface with the `implements` keyword and must provide every one of that interface's abstract methods. The same interface can be implemented by completely different classes in completely different ways — this is the foundation of **polymorphism**:

```
interface Shape {
    double area();

    double perimeter();
}

class Circle implements Shape {
    private final double radius;

    Circle(double radius) {
        this.radius = radius;
    }

    @Override
    public double area() {
```

```
        return Math.PI * radius * radius;
    }

    @Override
    public double perimeter() {
        return 2 * Math.PI * radius;
    }
}

class Rectangle implements Shape {
    private final double width;
    private final double height;

    Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }

    @Override
    public double area() {
        return width * height;
    }

    @Override
    public double perimeter() {
        return 2 * (width + height);
    }
}

class ShapeImplementationExample {
    public static void main(String[] args) {
        // Polymorphism through the interface reference: neither line below
        // needs to know whether it's holding a Circle or a Rectangle.
        Shape[] shapes = { new Circle(3), new Rectangle(4, 5) };

        for (Shape shape : shapes) {
            System.out.printf("area=%.2f perimeter=%.2f%n", shape.area(), shape.
perimeter());
        }
        // area=28.27 perimeter=18.85
        // area=20.00 perimeter=18.00
    }
}
```

`Circle` and `Rectangle` compute `area()` with wildly different formulas, but the loop in `ShapeImplementationExample` never has to know that — it only ever sees each object as a `Shape` and calls `area()` / `perimeter()`. Which method actually runs at runtime is decided by the JVM based on the object's real class (dynamic dispatch) — the `Shape` reference in the code doesn't even need to know that.

Implementing Multiple Interfaces

A class can `extends` only **one** class, but it can `implements` **any number** of interfaces at once — just separate them with commas:

```
interface Flyable {
    void fly();
}

interface Swimmable {
    void swim();
}

// A class can implement any number of interfaces — this is Java's answer to
// "multiple inheritance of type" (it only forbids multiple inheritance of
// STATE via `extends`, which classes are limited to one of).
class Duck implements Flyable, Swimmable {
    @Override
    public void fly() {
        System.out.println("The duck is flying");
    }

    @Override
    public void swim() {
        System.out.println("The duck is swimming");
    }
}

class MultipleInterfaceExample {
    public static void main(String[] args) {
        Duck duck = new Duck();
        duck.fly();
        duck.swim();

        // Each interface reference only sees its own contract.
        Flyable flyable = duck;
        flyable.fly();
    }
}
```

Expected Output

```
The duck is flying
The duck is swimming
The duck is flying
```

`Duck` fulfills both the `Flyable` and `Swimmable` contracts on its own — this is exactly where the single-inheritance workaround from "Why Does It Exist?" plays out. The same `duck` object can be used through a `Flyable` reference or through a `Swimmable` reference, whichever the situation calls for; each reference only ever sees the methods from its own contract.

An Interface Extending Another Interface

Just like classes, an interface can `extends` another interface (or, separated by commas, several at once) — the sub-interface then inherits all of the parent's abstract methods and adds its own:

```
interface Nameable {
    String name();
}

// An interface can extend another interface (or, unlike a class, several at
// once) — it inherits the abstract methods and adds its own.
interface Describable extends Nameable {
    String description();

    default String fullLabel() {
        return name() + " - " + description();
    }
}

class Product implements Describable {
    private final String name;
    private final String description;

    Product(String name, String description) {
        this.name = name;
        this.description = description;
    }

    @Override
    public String name() {
        return name;
    }

    @Override
    public String description() {
        return description;
    }
}

class InterfaceExtendsExample {
    public static void main(String[] args) {
        Describable product = new Product("Keyboard", "Mechanical, RGB");
        System.out.println(product.fullLabel()); // Keyboard - Mechanical, RGB

        // A Describable is also a Nameable, exactly like class inheritance.
        Nameable nameable = product;
        System.out.println(nameable.name()); // Keyboard
    }
}
```

Expected Output

```
Keyboard - Mechanical, RGB
Keyboard
```

`Describable` extends `Nameable`; when `Product` implements `Describable`, it's implicitly implementing `Nameable` too — it must provide `name()`, just as it must provide `description()`. Notice that the `fullLabel()` default method (we'll cover this keyword in detail in the next

section) freely calls `name()` and `description()` even though neither is implemented yet — an interface can safely use its own abstract methods inside its default methods, trusting that some implementation will supply them *later*.

Default Methods

The `default` keyword, introduced in Java 8, lets you give an interface method a **body** — this is exactly what solves the interface evolution problem mentioned in "History": adding a new default method to an existing interface never breaks any class that already implements it, because they all automatically inherit the new behavior:

```
interface Vehicle {
    void start();

    // Added years after Vehicle first shipped: every existing implementer
    // (like Bicycle below) keeps compiling untouched and simply inherits
    // this behavior for free.
    default void honk() {
        System.out.println("Beep!");
    }
}

class Car implements Vehicle {
    @Override
    public void start() {
        System.out.println("Car engine started");
    }
    // No honk() override needed — it uses Vehicle's default.
}

class SportsCar implements Vehicle {
    @Override
    public void start() {
        System.out.println("Sports car engine roars to life");
    }

    // A class is always free to override a default method with its own.
    @Override
    public void honk() {
        System.out.println("HOOONK!");
    }
}

class DefaultMethodExample {
    public static void main(String[] args) {
        Vehicle car = new Car();
        car.start();
        car.honk(); // Beep!

        Vehicle sportsCar = new SportsCar();
        sportsCar.start();
        sportsCar.honk(); // HOOONK!
    }
}
```

Expected Output

```
Car engine started
Beep!
Sports car engine roars to life
HOOONK!
```

When `honk()` is added to `Vehicle`, `Car` gets it for free without writing anything — it doesn't have to provide its own `honk()`. `SportsCar`, on the other hand, is free to replace that default with its own `@Override`, exactly like overriding in class inheritance. At runtime, a default method behaves like a **regular instance method** — it resolves polymorphically, and the calling code sees no difference at all.

> Default methods let you add **behavior** to a contract, but remember that an interface's core purpose is still describing "what can be done" — using default methods as the main home for business logic, rather than for shared/helper behavior, is a design trade-off we'll return to in "Best Practices."

Static Methods

Alongside `default`, Java 8 also added `static` methods to interfaces — unlike a default method, a static method **belongs to no implementer at all**; it's called directly through the interface name and can never be overridden:

```
interface DiscountPolicy {
    double apply(double price);

    // A static method belongs to the interface itself, not to any
    // implementer — it can never be overridden and is always called as
    // DiscountPolicy.xxx(), never through an instance.
    static DiscountPolicy percentageOff(double percent) {
        return price -> price * (1 - percent / 100);
    }

    static DiscountPolicy none() {
        return price -> price;
    }
}

class StaticMethodExample {
    public static void main(String[] args) {
        DiscountPolicy blackFriday = DiscountPolicy.percentageOff(30);
        System.out.println(blackFriday.apply(100.0)); // 70.0

        DiscountPolicy noDiscount = DiscountPolicy.none();
        System.out.println(noDiscount.apply(100.0)); // 100.0
    }
}
```

`DiscountPolicy.percentageOff(30)` is called **without** a `DiscountPolicy` instance — just like a class's static factory method. This pattern lets you keep an interface's related helper/factory methods right next to it, instead of moving them into a separate utility class (`DiscountPolicies` , say); `java.util.Comparator.comparing(...)` and `java.util.List.of(...)` are familiar real-world examples of this exact pattern.

Private Methods

Java 9 added **private methods** so that when two (or more) default methods share the same helper logic, that logic doesn't have to be duplicated — a private interface method (optionally `private static` as well) can only be called by the default/static methods of that same interface; it never leaks outside:

```
interface Invoice {
    double subtotal();

    double taxRate();

    // Two default methods share the same rounding logic. Extracting that
    // logic into a `private` method (Java 9+) lets them reuse it without
    // exposing it as part of the interface's public API.
    default double totalWithTax() {
        return round(subtotal() * (1 + taxRate()));
    }

    default double taxAmount() {
        return round(subtotal() * taxRate());
    }

    private double round(double value) {
        return Math.round(value * 100.0) / 100.0;
    }
}

class SimpleInvoice implements Invoice {
    private final double subtotal;
    private final double taxRate;

    SimpleInvoice(double subtotal, double taxRate) {
        this.subtotal = subtotal;
        this.taxRate = taxRate;
    }

    @Override
    public double subtotal() {
        return subtotal;
    }

    @Override
    public double taxRate() {
        return taxRate;
    }
}

class PrivateMethodExample {
    public static void main(String[] args) {
        Invoice invoice = new SimpleInvoice(199.99, 0.18);
        System.out.println(invoice.taxAmount());    // 36.0
        System.out.println(invoice.totalWithTax()); // 236.0
    }
}
```

`totalWithTax()` and `taxAmount()` both use the same rounding logic (`round(...)`) — rather than copying it into both, we extract it into a `private` method and share it. `round(...)` is **not part of `Invoice`'s public API at all** — a class implementing `SimpleInvoice` can never see it, and because it can't see it, it can neither override it nor accidentally use it with a different meaning.

> Private methods only arrived in Java 9+ — they work fine at this project's Java 21 target, but if you ever need code portable to an older Java version (like 8), you'd have to pull that shared logic out into a separate `static` helper class instead.

Diamond Problem and Its Resolution

What happens when a class implements two interfaces that both supply a **different default method with the same signature**? Java never guesses which one you meant — it forces the class to override the method explicitly:

```
interface Flyer {
    default String move() {
        return "flying";
    }
}

interface Swimmer {
    default String move() {
        return "swimming";
    }
}

// Duck inherits two DIFFERENT default implementations of move() from two
// unrelated interfaces — the compiler refuses to guess which one you meant,
// so overriding move() here is mandatory, not optional.
class Duck implements Flyer, Swimmer {
    @Override
    public String move() {
        // Explicitly picking (and combining) both parents' behavior via
        // InterfaceName.super.method() — plain super.method() does not
        // work here, it's ambiguous the same way the field would be.
        return Flyer.super.move() + " and " + Swimmer.super.move();
    }
}

class DiamondProblemExample {
    public static void main(String[] args) {
        Duck duck = new Duck();
        System.out.println(duck.move()); // flying and swimming
    }
}
```

Expected Output

```
flying and swimming
```

`Duck` inherits a conflicting `move()` default from both `Flyer` and `Swimmer` — the code won't compile unless you override `move()` yourself (since it's ambiguous whether `Flyer` or `Swimmer` was intended). The fix is the `InterfaceName.super.methodName()` syntax, which lets

you explicitly pick which parent interface's behavior to use (or, as in the example, combine both) — plain `super.move()` doesn't work here, because unlike class inheritance there's no single "parent," just two equal interfaces.

> This situation is historically known as the "diamond problem" — the same name as a much more severe ambiguity issue that arises from **multiple class inheritance** in C++. Java sidesteps that version entirely by never allowing multiple inheritance for classes; this lighter version, which only appears when interface **behavior** (a default method) collides, forces the compiler to make you resolve it explicitly instead of silently picking one for you.

Interface vs. Abstract Class

Even though default methods have brought interfaces much closer to abstract classes since Java 8, some differences remain permanent:

- **State:** An abstract class can hold instance fields; an interface can only declare `public static final` constants (as we saw in "Constant Fields") — it can never have a mutable instance-level field.
- **Constructors:** An abstract class can define a constructor (for shared initialization logic); an interface can never have one.
- **Number of parents:** A class can `extends` only **one** abstract class, but it can `implements` as many interfaces as it likes (as we saw in "Implementing Multiple Interfaces").
- **Purpose:** An abstract class exists to share the **implementation** of closely related types (an "is-a" relationship, a single hierarchy); an interface exists to let even completely unrelated types conform to the same **contract** (a "can-do" relationship — both a `String` and a `Player` can implement `Comparable`, with no hierarchical relationship between them whatsoever).

> A practical rule of thumb: if what you're sharing is **behavior and common fields** (types that are genuinely related), reach for an abstract class; if what you're sharing is just a **capability contract** (you want even unrelated types to be able to conform), prefer an interface. In modern Java, an interface with default methods is often more flexible than an abstract class for pure behavior sharing, since it still allows multiple implementation.

Functional Interfaces and Lambdas

An interface with **exactly one** abstract method is called a *functional interface* — default and static methods don't count toward that number, you can have as many as you want.

The `@FunctionalInterface` annotation isn't required, but it tells the compiler to guarantee that this interface has exactly one abstract method; if someone accidentally adds a second abstract method, the compile error is caught right there:

```
import java.util.function.Predicate;

// Exactly one abstract method (SAM) — default/static methods don't count
// toward that limit, so a functional interface can still have plenty of them.
@FunctionalInterface
interface Validator<T> {
    boolean isValid(T value);

    default Validator<T> negate() {
        return value -> !isValid(value);
    }
}

class FunctionalInterfaceExample {
    public static void main(String[] args) {
        // A lambda IS an instance of the functional interface — no named
        // implementing class needed.
        Validator<String> notBlank = value -> value != null && !value.isBlank();

        System.out.println(notBlank.isValid("hello")); // true
        System.out.println(notBlank.isValid("   "));   // false
        System.out.println(notBlank.negate().isValid("   ")); // true

        // java.util.function.Predicate is the exact same shape (T -> boolean)
        // as our own Validator — in real code, prefer the JDK's built-in
        // functional interfaces over hand-rolled ones whenever they fit.
        Predicate<String> notBlankPredicate = value -> value != null && !value.isBlank();
        System.out.println(notBlankPredicate.test("hi")); // true
    }
}
```

Expected Output

```
true
false
true
true
```

A lambda expression (`value -> value != null && !value.isBlank()`) can be used directly as an **instance** of a functional interface, without writing a separate class — Java matches the lambda's shape (parameter count/types, return type) against the target interface's single abstract method. The `java.util.function` package (`Predicate`, `Function`, `Supplier`, `Consumer`, and so on) already provides the most commonly needed shapes — checking whether one of these already fits your need is usually a better first step than writing your own functional interface.

> You can never assign a lambda to an interface with **two or more** abstract methods — the compiler has no way to know which method it's implementing. If you later add a second abstract method to an interface you wrote without `@FunctionalInterface`, every piece of code using it with a lambda breaks silently (with a clear compile error, but possibly months later); adding the annotation catches that mistake the moment it's introduced, not months down the line.

Sealed Interfaces

The `sealed` keyword, introduced in Java 17, lets you declare the **complete list** of types allowed to implement an interface up front with `permits` — the compiler then guarantees that no type outside that list, even in another package or module, can ever implement it:

```
// A sealed interface names, up front, the complete and closed set of types
// allowed to implement it — no unknown fourth implementer can ever show up
// at runtime from some other package.
sealed interface PaymentMethod permits CreditCard, BankTransfer, CashOnDelivery {
}

record CreditCard(String cardNumber) implements PaymentMethod {
}

record BankTransfer(String iban) implements PaymentMethod {
}

final class CashOnDelivery implements PaymentMethod {
}

class SealedInterfaceExample {
    static String describe(PaymentMethod method) {
        // Because PaymentMethod is sealed, the compiler knows these three
        // branches are the ONLY possibilities — no `default` branch needed,
        // and adding a fourth implementer elsewhere would fail to compile
        // here until this switch is updated.
        return switch (method) {
            case CreditCard creditCard -> "Card ending in " + creditCard.cardNumber().
substring(
                    creditCard.cardNumber().length() - 4);
            case BankTransfer bankTransfer -> "Transfer to " + bankTransfer.iban();
            case CashOnDelivery ignored -> "Cash on delivery";
        };
    }

    public static void main(String[] args) {
        System.out.println(describe(new CreditCard("4111111111111234")));
        System.out.println(describe(new BankTransfer("TR330006100519786457841326")));
        System.out.println(describe(new CashOnDelivery()));
    }
}
```

Expected Output

```
Card ending in 1234
Transfer to TR330006100519786457841326
Cash on delivery
```

The compiler knows `PaymentMethod` can only be implemented by `CreditCard`, `BankTransfer`, and `CashOnDelivery` — so once the `switch` expression covers those three branches, no `default` branch is needed at all; the compiler can prove the switch is **exhaustive**. If you want to add a fourth payment method tomorrow, you're **forced** to add it to the `permits` list first and then update this `switch` — the compiler protects you from forgetting, exactly the way adding a new enum constant does.

> Every type allowed in `permits` must be `final`, `sealed`, or `non-sealed` — meaning whether the hierarchy stays "closed" or reopens is spelled out explicitly at every level. The `record`s in the example satisfy this automatically, since records are already implicitly `final`; you don't need to write `final` yourself.

Real-World Use Cases

Every mechanism we've covered so far is one of the JDK's and popular frameworks' core design tools:

- `Comparable<T>` (java.lang, since JDK 1.2) gives a type a single "natural ordering" — any type that implements it is automatically understood by `Collections.sort()`, `Arrays.sort()`, `TreeSet`, and `TreeMap`:

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

// Comparable<T> is one of the JDK's oldest interfaces (java.lang, since
// JDK 1.2): implementing it gives a type a single, "natural" ordering that
// Collections.sort(), Arrays.sort(), TreeSet, and TreeMap all understand
// automatically — no separate sorting code needed anywhere.
class Player implements Comparable<Player> {
    private final String name;
    private final int score;

    Player(String name, int score) {
        this.name = name;
        this.score = score;
    }

    @Override
    public int compareTo(Player other) {
        return Integer.compare(this.score, other.score);
    }

    @Override
    public String toString() {
        return name + "(" + score + ")";
    }
}

class ComparableImplementationExample {
    public static void main(String[] args) {
        List<Player> players = new ArrayList<>(List.of(
                new Player("Ada", 42),
                new Player("Linus", 17),
                new Player("Grace", 88)));

        Collections.sort(players);
        System.out.println(players); // [Linus(17), Ada(42), Grace(88)]
    }
}
```

Expected Output

```
[Linus(17), Ada(42), Grace(88)]
```

- **`Runnable` / `Callable`** represent a unit of work ("run this") independently of any concrete class — threads and `ExecutorService` know about them without ever caring which class you used.
- **The collection hierarchy** (`Collection`, `List`, `Map`, `Set`) is almost entirely made of interfaces — code depending on `List` rather than `ArrayList` is the most common real-world example of the decoupling discussed in "Why Does It Exist?"

- **Spring** typically injects classes marked `@Service` / `@Repository` through an interface (a `UserService` interface plus a `UserServiceImpl` class) — this makes it possible to hand a test a mock `UserService` instead of the real implementation, without changing any calling code.
- **Marker interfaces** like `Serializable` / `Cloneable`, which contain no methods at all, add no behavior to a type by themselves — they only signal to the JVM/library "this type opts into special treatment" (checked with `instanceof`).

Best Practices

- **Program to an interface, not an implementation** — prefer an interface over a concrete class for variable/parameter/return types wherever you can (as discussed in "Why Does It Exist?").
- **Interface Segregation Principle (ISP):** keep an interface small and focused; instead of one "bloated" interface that forces implementers to provide methods they don't need, a few narrow interfaces (optionally combined the way we did in "An Interface Extending Another Interface") are usually more flexible.
- Use default methods for **shared/helper behavior**, not as the main home for business logic (see the tip in "Default Methods").
- Before writing your own functional interface, check whether `java.util.function` already has a matching one (see "Functional Interfaces and Lambdas").
- If you're modeling a closed, finite set of types — especially one you'll use with `switch` — consider `sealed`, so you benefit from compiler-checked exhaustiveness (see "Sealed Interfaces").
- Prefixing interface names with `I` (`IShape`), as in C#, goes against Java convention; prefer plain names (`Shape` / `Comparable`), an `-able` suffix, or a direct role name instead.

Common Mistakes

1. Assuming an interface field gets a separate copy per instance. All interface fields are implicitly `static` — they all point to one shared value, no implementer gets its own copy (see "Constant Fields").

2. Expecting the compiler to make a "sensible" choice in the diamond problem. When two interfaces supply conflicting default methods with the same signature, overriding is **mandatory**; the compiler never picks one on its own (see "Diamond Problem and Its Resolution").

3. Trying to assign a lambda to an interface that isn't `@FunctionalInterface` (has more than one abstract method). A lambda can only be assigned to an interface with **exactly one** abstract method (see "Functional Interfaces and Lambdas").

4. Assuming private interface methods are available in Java 8 too. Private methods only arrived in Java 9+; static and default methods have been around since Java 8 — mixing

these two up leads to unexpected compile errors when targeting an older JDK (see "History" and "Private Methods").

5. Forgetting to update the `permits` list when adding a new implementation to a sealed interface. A new type won't compile until it's added to the `permits` list — this restriction is intentional, not a bug to work around (see "Sealed Interfaces").

6. Turning an interface into a single "does everything" contract that bundles together methods with no real relationship to each other. This violates the Interface Segregation Principle and forces implementers to provide methods they don't actually need (see "Best Practices").

Summary, Cheat Sheet, and Glossary

The interface has been Java's fundamental building block for separating a type's contract from its implementation since JDK 1.0. Key takeaways:

- A bodyless interface method is implicitly `public abstract`; an interface field is always implicitly `public static final`
- A class can `extends` only one class but can `implements` as many interfaces as it likes
- An interface can `extends` another interface (or several at once)
- Java 8: `default` methods (have a body, overridable, automatically inherited) and `static` methods (belong to the interface, never overridable)
- Java 9: `private` methods (callable only from the same interface's default/static methods, invisible outside it)
- When two default methods collide, overriding is mandatory; `InterfaceName.super.method()` picks a specific parent's behavior
- An interface with exactly one abstract method is a *functional interface* — usable as a lambda target
- Java 17: `sealed` + `permits` restrict which types may implement an interface to a closed list, enabling compiler-checked exhaustive `switch`es

Quick reference:

```
// Basic definition and implementation
interface Shape {
    double area();                      // implicit: public abstract

    double DEFAULT_SIDES = 4;           // implicit: public static final

    default String describe() {         // Java 8: has a body, overridable
        return "area=" + area();
    }

    static Shape unit() {               // Java 8: belongs to the interface, no instance
needed
        return () -> 1.0;
    }

    private double round(double v) {    // Java 9: callable only from inside
        return Math.round(v * 100) / 100.0;
    }
}

class Circle implements Shape {
    public double area() { return 3.14; }
}

// Multiple implementation + interface extension
interface A { }
interface B { }
class C implements A, B { }
interface D extends A, B { }

// Diamond problem resolution
interface X { default String who() { return "X"; } }
interface Y { default String who() { return "Y"; } }
class Z implements X, Y {
    public String who() { return X.super.who() + Y.super.who(); }
}

// Sealed interface
sealed interface Result permits Success, Failure { }
record Success(String value) implements Result { }
record Failure(String reason) implements Result { }
```

Glossary

Interface — A contract defining a type's behavior independently of its implementation; declared with the `interface` keyword.

Abstract method — A bodyless interface method, implicitly `public abstract`; any class implementing the interface must supply a body for it.

`default` **method** — An interface method with a body, introduced in Java 8, that implementers can override.

`static` **method** — An interface method introduced in Java 8, called directly through the interface name, belonging to no implementer and never overridable.

`private` **method** — An interface helper method introduced in Java 9, callable only from the same interface's default/static methods and invisible outside it.

Diamond problem — The ambiguity that arises when a class inherits conflicting default methods with the same signature from two different interfaces, requiring an explicit override.

`InterfaceName.super.method()` — Syntax used to resolve the diamond problem by explicitly selecting a specific parent interface's default method implementation.

Functional interface — An interface with exactly one abstract method; can be the target of lambda expressions and method references. The `@FunctionalInterface` annotation enforces this at compile time.

`sealed` / `permits` — Keywords introduced in Java 17 that restrict, up front, the complete list of types allowed to implement an interface.

Marker interface — An interface with no methods at all, used only to "flag" a type for special treatment recognized by the JVM/a library (`Serializable`, `Cloneable`).

Interface Segregation Principle (ISP) — One of the SOLID principles; argues for several small, focused interfaces rather than one wide interface that forces implementers to provide methods they don't need.

Appendix: Mini Project — A Simple Plugin/Strategy System

Let's combine what we've learned so far ("Implementing Multiple Interfaces," "Functional Interfaces and Lambdas") to build a small plugin system whose behavior can be "plugged in" at runtime. The idea is simple: we define a `NotificationChannel` contract, and the registry relies on nothing but that contract — it never knows which concrete classes (or lambdas) are registered:

```
import java.util.HashMap;
import java.util.Map;

// The contract every plugin has to satisfy — the registry below never
// needs to know about a concrete plugin's class, only this interface.
interface NotificationChannel {
    void send(String message);
}

class EmailChannel implements NotificationChannel {
    @Override
    public void send(String message) {
        System.out.println("[email] " + message);
    }
}

class SmsChannel implements NotificationChannel {
    @Override
    public void send(String message) {
        System.out.println("[sms] " + message);
    }
}

// A tiny strategy/plugin registry: new channels can be registered from
// anywhere (even a different package/module) as long as they implement
// NotificationChannel — the registry's code never has to change.
class PluginRegistry {
    private final Map<String, NotificationChannel> channels = new HashMap<>();

    void register(String key, NotificationChannel channel) {
        channels.put(key, channel);
    }

    void broadcast(String message) {
        for (NotificationChannel channel : channels.values()) {
            channel.send(message);
        }
    }

    void sendVia(String key, String message) {
        NotificationChannel channel = channels.get(key);
        if (channel == null) {
            throw new IllegalArgumentException("No channel registered for: " + key);
        }
        channel.send(message);
    }
}
```

```
class PluginRegistryDemo {
    public static void main(String[] args) {
        PluginRegistry registry = new PluginRegistry();
        registry.register("email", new EmailChannel());
        registry.register("sms", new SmsChannel());

        // A lambda works too — a functional interface plugin, registered
        // right at the call site, no separate class required.
        registry.register("console", message -> System.out.println("[console] " + message));

        registry.broadcast("Server restarting in 5 minutes");
        // [email] Server restarting in 5 minutes
        // [sms] Server restarting in 5 minutes
        // [console] Server restarting in 5 minutes

        registry.sendVia("sms", "Restart complete");
        // [sms] Restart complete
    }
}
```

`PluginRegistry`'s code has no idea `EmailChannel` or `SmsChannel` exist — notice that the third channel isn't even a class, it's a plain lambda. In the real world, the interface implementations Spring scans via `@Component`, or an e-commerce system that hides a payment provider (Stripe/iyzico) behind a single `PaymentGateway` interface, uses exactly this pattern.

> This mini project needed no reflection at all — don't confuse it with the "Simple Dependency Injection Container" mini project from the Reflection lesson: that container **discovered at runtime** which type needed which other type; here, the programmer decides **explicitly**, with a `register(...)` call, which implementation gets used. Both serve the same idea — not hardcoding the concrete class — but through different mechanisms.

Appendix: Mini Project — Event Bus (Publish/Subscribe)

The final mini project defines its own functional interface (`OrderPlacedListener`) by extending `java.util.function.Consumer`, then builds a minimal publish/subscribe mechanism on top of it — combining two ideas from "Functional Interfaces and Lambdas" and "An Interface Extending Another Interface":

```
import java.util.ArrayList;
import java.util.List;
import java.util.function.Consumer;

// A functional interface used as an event listener contract. Extending
// Consumer<OrderPlacedEvent> instead of writing a brand-new interface from
// scratch reuses a shape the whole JDK already understands.
@FunctionalInterface
interface OrderPlacedListener extends Consumer<OrderPlacedEvent> {
}

record OrderPlacedEvent(String orderId, double amount) {
}

// A minimal publish/subscribe event bus: subscribers register a listener
// (often just a lambda), and every published event fans out to all of them —
// the core idea behind Spring's ApplicationEventPublisher, scaled down to
// its essence.
class EventBus {
    private final List<OrderPlacedListener> listeners = new ArrayList<>();

    void subscribe(OrderPlacedListener listener) {
        listeners.add(listener);
    }

    void publish(OrderPlacedEvent event) {
        for (OrderPlacedListener listener : listeners) {
            listener.accept(event);
        }
    }
}
```

```
class EventBusDemo {
    public static void main(String[] args) {
        EventBus bus = new EventBus();

        bus.subscribe(event -> System.out.println("[email] Order confirmed: " + event.
orderId()));
        bus.subscribe(event -> {
            if (event.amount() > 500) {
                System.out.println("[fraud-check] High value order flagged: " + event.
orderId());
            }
        });

        bus.publish(new OrderPlacedEvent("ORD-1001", 129.90));
        // [email] Order confirmed: ORD-1001

        bus.publish(new OrderPlacedEvent("ORD-1002", 899.00));
        // [email] Order confirmed: ORD-1002
        // [fraud-check] High value order flagged: ORD-1002
    }
}
```

`OrderPlacedListener extends Consumer<OrderPlacedEvent>` adds no new abstract method — it just wraps `Consumer`'s `accept(T)` under a more meaningful name; so it's still a valid functional interface, and still implementable with a lambda. Notice the two `subscribe(...)` calls in `EventBusDemo`: `EventBus` has no idea how many listeners exist or what they do — it just forwards every published event, in order, to each registered `OrderPlacedListener`.

> This pattern is the core of Spring's `ApplicationEventPublisher` / `@EventListener` mechanism, stripped of real-world complexity (asynchronous processing, error handling, event hierarchies) — it's the right choice when a small number of loosely coupled components need to communicate without holding direct references to each other.