Interface

Java'da davranış sözleşmeleri tanımlama; default/static/private metotlar ve sealed interface'ler.

Orta 50 dk

Son güncelleme: 9 Kasım 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/interface

---

Interface

Java'da **interface** (arayüz), bir tipin *ne yapabildiğini* tanımlayan ama *nasıl yaptığını* söylemeyen bir sözleşmedir (contract). Bir sınıf bir interface'i implement ettiğinde, o sözleşmeyi yerine getirmeyi taahhüt eder — çağıran kod, karşısındaki nesnenin gerçek sınıfını hiç bilmeden, yalnızca bu sözleşmeye güvenerek çalışabilir. Bu derste, temel soyut metotlardan başlayıp Java 8'in default/static metotlarına, Java 9'un private metotlarına ve Java 17'nin sealed interface'lerine kadar, interface'in zaman içinde nasıl evrildiğini uçtan uca işleyeceğiz.

Interface Nedir?

En basit haliyle bir interface, gövdesiz metot imzalarından oluşan bir tip tanımıdır — "bu tipten bir nesnede şu metotlar bulunmalı" der, ama metotların içini boş bırakır:

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

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

`Invoice`, `implements Payable` yazarak "ben bu sözleşmeyi yerine getiriyorum" demiş oluyor; `calculatePayment()`'ı gerçekten tanımlamazsa kod derlenmez. Karşılığında, `Payable` tipindeki bir değişken üzerinden çalışan herhangi bir kod, elindeki nesnenin `Invoice` mi, yoksa henüz yazılmamış başka bir `Payable` implementasyonu mu olduğunu hiç bilmek zorunda kalmaz.

Neden Var?

Interface'in çözdüğü temel problem **decoupling** — kodu, somut bir sınıfa değil, bir sözleşmeye bağımlı kılmak. "Bir `ArrayList` bekliyorum" yerine "bir `List` bekliyorum" demek, çağıran kodun yarın `LinkedList`'e geçmesini, hatta testte sahte (mock) bir implementasyon kullanmasını, hiçbir şeyi değiştirmeden mümkün kılar — bu ilkeye genelde *"implementasyona değil, arayüze göre programla"* denir.

İkinci önemli sebep, Java'nın tek kalıtım (single inheritance) kısıtlamasını aşmak. Bir sınıf yalnızca **bir** sınıftan `extends` ile miras alabilir — ama gerçek dünyada bir nesnenin birden fazla "rolü" olabilir: bir `Duck` hem uçabilir hem yüzebilir, ama ne "Uçan Şeyler" sınıfından ne de "Yüzen Şeyler" sınıfından aynı anda miras alabilir. İnterface'ler, sınıfların **tek durumdan (state)** ama **birden fazla sözleşmeden** miras almasına izin vererek bu sorunu çözer ("Çoklu Interface Implement Etmek" bölümünde bunu somut bir örnekle göreceğiz).

Tarihçe

Interface, Java'nın ilk gününden (JDK 1.0, 1996) beri var — ama başlangıçta çok katı bir kuralı vardı: yalnızca soyut metotlar ve sabitler içerebiliyordu, hiçbir metot gövdesi barındıramıyordu. Bu, yıllar içinde ciddi bir pratik soruna yol açtı: `java.util.List` gibi, binlerce üçüncü parti sınıfın implement ettiği bir interface'e tek bir yeni metot eklemek bile, o metodu implement etmeyen **her** sınıfın derlenmesini kırıyordu — bu yüzden JDK'nın kendisi bile koleksiyon arayüzlerine uzun yıllar yeni metot ekleyemedi.

Java 8 (2014), bu "interface evrimi" problemini **default metotlar** ile çözdü — bir interface artık gövdeli bir metot tanımlayabiliyor, mevcut implementasyonlar hiçbir şey yapmadan bu metodu otomatik miras alıyordu (`forEach`, `stream()` gibi metotlar tam olarak bu sayede `Collection`'a sonradan eklenebildi). Aynı sürümde **static metotlar** da geldi. Java 9 (2017), default metotlar arasında kod paylaşımını kolaylaştırmak için **private metotlar**ı ekledi. Java 17 (2021), interface'in implement edebileceği tipleri açıkça sınırlayan **sealed interface**'leri getirdi — bu projede kullandığımız Java 21, tüm bu katmanları bir arada barındırıyor ve hepsini bu derste sırayla işleyeceğiz.

İlk Interface'ini Yazmak

Bir interface, `class` yerine `interface` anahtar kelimesiyle tanımlanır; onu implement eden bir sınıf, `implements` ile bunu belirtir ve interface'in tüm soyut metotlarını `@Override` ile gerçeklemek zorundadır:

```
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`, tek bir metot imzası dışında hiçbir şey söylemiyor — `Invoice`'un o metodu nasıl hesapladığı tamamen kendi işi. `FirstInterfaceDemo` içinde `Payable invoice = new Invoice(...)` satırına dikkat et: değişkenin tipi `Invoice` değil, `Payable` — bu, ileride başka bir `Payable` implementasyonuna geçmenin, çağıran kodda tek bir satır bile değiştirmeyeceği anlamına gelir.

> Bir sınıf `implements` yazıp interface'in bir metodunu eksik bırakırsa, derleyici hemen hata verir — bu, interface'in en değerli özelliklerinden biri: sözleşmeyi bozmak çalışma zamanında değil, **derleme zamanında** yakalanır.

Soyut Metotlar

Bir interface içindeki gövdesiz bir metot, hiçbir şey yazmasan bile örtük olarak (implicitly) `public abstract`'tır — bu iki kelimeyi elle yazman gerekmez, yazsan da anlam değişmez:

```
// 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!
    }
}
```

Beklenen Çıktı

```
Hello, Ada!
```

`Greeter` ile `GreeterExplicit` tamamen aynı şeyi tanımlıyor. Bunun pratikteki önemli bir sonucu var: bir sınıf bu metodu implement ederken, erişim belirleyicisini **daraltamaz** — interface metodu örtük olarak `public` olduğu için, `EnglishGreeter.greet(...)` 'i `protected` ya da paketin içine kapalı (package-private) yazmaya kalkışsan derleme hatası alırsın; override eden metot en az interface'teki kadar erişilebilir olmak zorundadır.

> Bir interface metodunun gövdesiz olması için `abstract` anahtar kelimesini **yazmana gerek yok**, ama bu metoda bir gövde eklemek istiyorsan (`default` / `static` / `private` kullanmadan) buna hiç izin verilmez — düz bir metot ya tamamen gövdesiz (soyut) ya da aşağıdaki bölümlerde göreceğimiz üç özel anahtar kelimeden biriyle işaretlenmiş olmalıdır.

Constant Alanlar

Bir interface'te tanımlanan alanlar da, tıpkı metotlar gibi, örtük olarak `public static final` 'dır — yani hem sabittirler hem de bir örneğe ihtiyaç duymadan doğrudan `InterfaceAdı.ALAN` şeklinde erişilirler:

```
// 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
    }
}
```

`FreeFall`, `PhysicsConstants`'ı implement ederek `GRAVITY`'ye sanki kendi alanıymış gibi doğrudan erişebiliyor — ama bu bir **kalıtım** değil, çünkü alan zaten `static`; her implementasyon kendi kopyasını almaz, hepsi aynı tek sabiti paylaşır.

> Bir interface alanının `final` olduğunu unutup ona sonradan bir değer atamaya çalışmak (`PhysicsConstants.GRAVITY = 10;`) derleme hatasıdır. Bir interface, implementasyonlar arasında paylaşılan **değişebilir (mutable) durum** tutmak için asla uygun bir yer değildir — bunun için sınıfların `static` alanlarını kullan.

Interface Implement Etmek

Bir sınıf, `implements` anahtar kelimesiyle bir interface'e bağlanır ve interface'in tüm soyut metotlarını gerçeklemek zorundadır. Aynı interface'i birden fazla sınıf, birbirinden tamamen farklı şekillerde implement edebilir — bu, **polimorfizmin** temelidir:

```
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` ve `Rectangle`, `area()`'yı taban tabana zıt formüllerle hesaplıyor, ama `ShapeImplementationExample`'daki döngü bunu hiç bilmiyor — elindeki her nesneyi yalnızca bir `Shape` olarak görüyor ve `area()` / `perimeter()` çağırıyor. Çalışma zamanında hangi metodun **gerçekten** çalışacağına JVM, nesnenin gerçek sınıfına bakarak karar veriyor (dinamik gönderim / dynamic dispatch) — koddaki `Shape` referansı bunu bilmene bile gerek bırakmıyor.

Çoklu Interface Implement Etmek

Bir sınıf, `extends` ile yalnızca **bir** sınıftan miras alabilirken, `implements` ile **birden fazla** interface'i aynı anda implement edebilir — virgülle ayırman yeterli:

```
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();
    }
}
```

> **Beklenen Çıktı**
>
> ```
> The duck is flying
> The duck is swimming
> The duck is flying
> ```

`Duck`, hem `Flyable` hem `Swimmable` sözleşmesini tek başına yerine getiriyor — "Neden Var?" bölümünde değindiğimiz, Java'nın tek kalıtım kısıtlamasını aşma problemi tam olarak burada çözülüyor. Aynı `duck` nesnesi, ihtiyaca göre bir `Flyable` referansı üzerinden de, bir `Swimmable` referansı üzerinden de kullanılabilir; her referans yalnızca kendi sözleşmesindeki metotları görür.

Interface'in Interface'i Genişletmesi

Tıpkı sınıflar gibi, bir interface de `extends` ile başka bir interface'i (hatta virgülle ayırarak birden fazlasını) genişletebilir — bu durumda alt interface, üst interface'in tüm soyut metotlarını devralır ve kendi metotlarını ekler:

```
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
    }
}
```

Beklenen Çıktı

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

`Describable`, `Nameable`'ı genişletiyor; `Product` sınıfı `Describable`'ı implement ettiğinde, aslında dolaylı olarak `Nameable`'ı da implement etmiş oluyor — `name()`'i gerçeklemek zorunda kalıyor, tıpkı `description()` gibi. `fullLabel()` default metodunun (bir sonraki bölümde bu anahtar kelimeyi detaylandıracağız), henüz implement edilmemiş `name()` ve `description()`'ı doğrudan çağırabildiğine dikkat et — bir interface, kendi soyut metotlarını (implementasyonun *ileride* sağlayacağını varsayarak) default metotlarının içinde güvenle kullanabilir.

Default Metotlar

Java 8 ile gelen `default` anahtar kelimesi, bir interface metoduna **gövde** yazmana izin verir — "Tarihçe" bölümünde bahsettiğimiz interface evrimi problemini çözen tam olarak budur: var olan bir interface'e yeni bir default metot eklediğinde, onu implement eden hiçbir sınıfın derlemesi bozulmaz, hepsi bu davranışı otomatik miras alır:

```
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!
    }
}
```

Beklenen Çıktı

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

`Vehicle`'a `honk()` eklendiğinde, `Car` hiçbir şey yapmadan bu metodu bedava kazanıyor — kendi `honk()` implementasyonunu yazmak zorunda değil. `SportsCar` ise dilerse bu

varsayılanı, tıpkı bir sınıf kalıtımındaki gibi, kendi `@Override`'ıyla değiştirebiliyor. Bir default metot, çalışma zamanında **normal bir instance metodu** gibi davranır — polimorfik olarak çözümlenir, çağıran kod hiçbir farkı hissetmez.

> Default metotlar sözleşmeye **davranış** eklemeni sağlasa da, bir interface'in temel amacının hâlâ "ne yapılabildiğini" tanımlamak olduğunu unutma — default metotları business logic'in ana yeri haline getirmek yerine, ortak/yardımcı davranışlar için kullanmak "Best Practices" bölümünde değineceğimiz bir tasarım tavsiyesidir.

Static Metotlar

Java 8, `default` ile birlikte `static` metotları da interface'lere ekledi — bir default metodun aksine, bir static metot **hiçbir implementasyona ait değildir**, doğrudan interface adı üzerinden çağrılır ve asla override edilemez:

```
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)`, bir `DiscountPolicy` **örneği olmadan** çağrılıyor — tıpkı bir sınıfın static factory metodu gibi. Bu kalıp, bir interface'in kendi ilgili yardımcı/factory metotlarını, ayrı bir utility sınıfına (`DiscountPolicies` gibi) taşımadan, doğrudan yanına koymanı sağlar; `java.util.Comparator.comparing(...)` ve `java.util.List.of(...)` bu deseni gerçek dünyada kullanan tanıdık örneklerdir.

Private Metotlar

Java 9, iki (ya da daha fazla) default metot aynı yardımcı mantığı paylaştığında bu mantığı tekrar etmek zorunda kalmamak için **private metotları** ekledi — bir private interface metodu (isteğe bağlı olarak `private static` da olabilir), yalnızca aynı interface'in default /static metotları tarafından çağrılabilir, dışarıya hiç sızmaz:

```
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()` ve `taxAmount()`, ikisi de aynı yuvarlama mantığını (`round(...)`) kullanıyor — bu mantığı her ikisine de kopyalamak yerine, `private` bir metoda çıkarıp paylaşıyoruz. `round(...)`, `Invoice`'un public API'sinin **hiçbir parçası değil** — `SimpleInvoice`'u implement eden bir sınıf onu asla göremez, göremediği için de override edemez ya da yanlışlıkla farklı bir anlamda kullanamaz.

> Private metotlar yalnızca Java 9+ ile gelir — bu projedeki Java 21 hedefinde sorunsuz çalışır, ama daha eski bir Java sürümüne (8 gibi) taşınabilir kod yazman gerekiyorsa, bu paylaşılan mantığı ayrı bir `static` yardımcı sınıfa çıkarmak zorunda kalırsın.

Diamond Problem ve Çözümü

Bir sınıf, aynı imzaya sahip **iki farklı default metot** sağlayan iki interface'i aynı anda implement ederse ne olur? Java, hangisini seçeceğini asla kendiliğinden **tahmin etmez** — bu durumda sınıfın metodu açıkça override etmesini zorunlu kılar:

```
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
    }
}
```

Beklenen Çıktı

```
flying and swimming
```

`Duck`, hem `Flyer`'dan hem `Swimmer`'dan çakışan bir `move()` default'u miras alıyor — `move()`'u override etmezsen kod derlenmez (`Flyer`'ın mı yoksa `Swimmer`'ın mı kastedildiği belirsiz olduğu için). Çözüm, `InterfaceAdı.super.metotAdı()` söz dizimiyle üst interface'lerden

hangisinin (ya da örnekteki gibi ikisinin birden) kullanılacağını açıkça belirtmek — düz `super.move()` burada işe yaramaz, çünkü sınıf kalıtımındaki gibi tek bir "üst" yok, birbirine eşit iki interface var.

> Bu durum tarihsel olarak "elmas problemi" (diamond problem) diye anılır — C++'ta **çoklu sınıf kalıtımı** yüzünden ortaya çıkan, çok daha ciddi bir belirsizlik problemiyle aynı isimdir. Java, sınıflar için çoklu kalıtıma hiç izin vermeyerek o versiyonunu baştan engelliyor; interface'lerdeki bu daha hafif versiyon ise yalnızca **davranış** (default metot) çakıştığında ortaya çıkar ve derleyici seni sessizce yanlış bir seçim yapmak yerine açıkça çözmeye zorlar.

Interface vs Abstract Class

Java 8'den beri default metotlarla interface'ler abstract class'lara epey yaklaşsa da, aralarında hâlâ kalıcı farklar var:

- **Durum (state):** Bir abstract class instance alanları tutabilir; bir interface yalnızca `public static final` sabitler tanımlayabilir ("Constant Alanlar" bölümünde gördük) — instance düzeyinde değişebilir bir alanı asla olamaz.
- **Constructor:** Bir abstract class constructor tanımlayabilir (ortak başlatma mantığı için); bir interface'in hiçbir zaman constructor'ı olamaz.
- **Kalıtım sayısı:** Bir sınıf yalnızca **bir** abstract class'tan `extends` ile miras alabilir, ama istediği kadar interface `implements` edebilir ("Çoklu Interface Implement Etmek" bölümünde gördük).
- **Amaç:** Abstract class, birbiriyle yakından ilişkili tiplerin ortak **implementasyonunu** paylaşmak içindir ("is-a" ilişkisi, tek bir hiyerarşi); interface, birbiriyle hiç ilişkili olmayan tiplerin bile aynı **sözleşmeye** uyabilmesini sağlamak içindir ("can-do" ilişkisi — `Comparable`'ı hem bir `String` hem bir `Player` implement edebilir, ikisi arasında hiçbir hiyerarşik ilişki yok).

> Pratik bir kural: paylaşacağın şey **davranış ve ortak alanlar** ise (birbirine gerçekten akraba tipler) abstract class'ı düşün; paylaşacağın şey yalnızca bir **yetenek sözleşmesi** ise (birbirine hiç akraba olmayan tipler bile uyabilsin istiyorsan) interface'i tercih et. Modern Java'da, sadece davranış paylaşmak için default metotlu bir interface çoğu zaman abstract class'a göre daha esnektir, çünkü çoklu implement etmeye izin verir.

Functional Interface ve Lambda

Tam olarak **tek** soyut metoda sahip bir interface'e *functional interface* (fonksiyonel arayüz) denir — default ve static metotlar bu sayıya dahil edilmez, istediğin kadar olabilir. `@FunctionalInterface` annotation'ı zorunlu değildir ama derleyiciye "bu interface'in tek soyut

metodu olmasını garanti et" demeni sağlar; birisi yanlışlıkla ikinci bir soyut metot eklerse, derleme hatası hemen orada yakalanır:

```
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
    }
}
```

Beklenen Çıktı

```
true
false
true
true
```

Bir lambda ifadesi (`value -> value != null && !value.isBlank()`), ayrı bir sınıf yazmadan, doğrudan bir functional interface'in **örneği** olarak kullanılabilir — Java, lambda'nın imzasını (parametre sayısı/tipleri, dönüş tipi) hedef interface'in tek soyut metoduyla eşleştirir. `java.util.function` paketi (`Predicate`, `Function`, `Supplier`, `Consumer` gibi), en sık ihtiyaç duyulan şekilleri hazır sağlar — kendi functional interface'ini yazmadan önce, ihtiyacının bunlardan biriyle zaten karşılanıp karşılanmadığını kontrol etmek genelde daha iyi bir ilk adımdır.

> Bir interface'in **iki veya daha fazla** soyut metodu varsa, ona asla lambda atayamazsın — derleyici hangi metodu implement ettiğini bilemez. `@FunctionalInterface` eklemeden yazdığın bir interface'e sonradan ikinci bir soyut metot eklersen, onu lambda ile kullanan tüm kod sessizce (derleme zamanında, açık bir hatayla) bozulur; annotation'ı eklemek bu hatayı ekleme anında yakalar, aylar sonra değil.

Sealed Interface

Java 17 ile gelen `sealed` anahtar kelimesi, bir interface'i implement edebilecek tiplerin **tam listesini** `permits` ile önceden belirlemene izin verir — derleyici, bu listenin dışında hiçbir tipin (başka bir pakette, başka bir modülde bile) bu interface'i implement edemeyeceğini garanti eder:

```
// 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()));
    }
}
```

Beklenen Çıktı

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

`PaymentMethod`'un yalnızca `CreditCard`, `BankTransfer` ve `CashOnDelivery` tarafından implement edilebileceğini derleyici biliyor — bu yüzden `switch` ifadesinde bu üç dalı kapsadığımızda, hiçbir `default` dalına ihtiyaç duymuyoruz; derleyici bunun **tüketici (exhaustive)** olduğunu kanıtlayabiliyor. Yarın dördüncü bir ödeme yöntemi eklemek istersen, önce `permits` listesine ekleyip sonra bu `switch`'i güncellemek **zorunda** kalırsın — derleyici seni unutmaya karşı korur, tıpkı bir enum'a yeni bir değer eklendiğinde olduğu gibi.

> `permits` ile izin verilen her tip, `final`, `sealed` ya da `non-sealed` olmak zorundadır — yani hiyerarşinin "kapalı" mı yoksa yeniden "açık" mı olacağı her seviyede açıkça belirtilir. Örnekteki `record`'lar buna otomatik uyar, çünkü record'lar zaten örtük olarak `final`'dır; ayrıca `final` yazmana gerek yoktur.

Gerçek Dünya Kullanım Alanları

Buraya kadar öğrendiğimiz her mekanizma, JDK'nın ve popüler framework'lerin temel tasarım araçlarından biridir:

- `Comparable<T>` (java.lang, JDK 1.2'den beri), bir tipe tek bir "doğal sıralama" kazandırır — implement eden her tip, `Collections.sort()`, `Arrays.sort()`, `TreeSet`, `TreeMap` tarafından otomatik olarak anlaşılır:

```
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)]
    }
}
```

Beklenen Çıktı

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

- **`Runnable` / `Callable`**, bir iş parçasını ("şunu çalıştır") somut bir sınıftan bağımsız şekilde temsil eder — thread'ler ve `ExecutorService` bunları bilir, senin hangi sınıfı kullandığını hiç umursamaz.
- **Koleksiyon hiyerarşisi** (`Collection`, `List`, `Map`, `Set`), neredeyse tamamen interface'lerden oluşur — kodun `ArrayList`'e değil `List`'e bağımlı olması, "Neden Var?" bölümünde bahsettiğimiz decoupling'in en yaygın gerçek dünya örneğidir.

- **Spring**, `@Service` / `@Repository` ile işaretlenen sınıfları genelde bir interface üzerinden enjekte eder (`UserService` interface'i + `UserServiceImpl` sınıfı) — bu, testte gerçek implementasyon yerine bir sahte (mock) `UserService` vermeyi, hiçbir çağıran kodu değiştirmeden mümkün kılar.
- `Serializable` / `Cloneable` gibi, hiç metot içermeyen "marker interface"ler, bir tipe herhangi bir davranış eklemez — yalnızca JVM'e/kütüphaneye "bu tip şu özel muameleyi kabul ediyor" diye işaret verir (`instanceof` ile kontrol edilir).

Best Practices

- **İmplementasyona değil, arayüze göre programla** — değişken/parametre/dönüş tiplerinde, mümkün olduğunca somut sınıf yerine interface kullan ("Neden Var?" bölümünde değindik).
- **Interface Segregation Principle (ISP):** bir interface'i küçük ve odaklı tut; implementasyonların ihtiyaç duymadığı metotları implement etmeye zorlayan "şişkin" bir interface yerine, birkaç dar interface'i (istersen "Interface'in Interface'i Genişletmesi" bölümündeki gibi) birleştirmek genelde daha esnektir.
- Default metotları **yardımcı/ortak davranış** için kullan, business logic'in ana yeri haline getirme (bkz. "Default Metotlar" bölümündeki tip).
- Kendi functional interface'ini yazmadan önce `java.util.function` paketinde hazır bir eşleniği olup olmadığını kontrol et (bkz. "Functional Interface ve Lambda").
- Kapalı, sonlu bir tip kümesi tasarlıyorsan (özellikle `switch` ile birlikte kullanılacaksa) `sealed` kullanmayı değerlendir — derleyici desteğinden (exhaustiveness kontrolü) faydalanırsın (bkz. "Sealed Interface").
- Interface isimlerinde C#'taki gibi `I` öneki (`IShape`) kullanmak Java konvansiyonuna aykırıdır; bunun yerine `Shape` / `Comparable` / `-able` sonekli isimler ya da doğrudan rol adı tercih edilir.

Yaygın Hatalar

1. Bir interface alanının instance başına ayrı bir kopyası olacağını sanmak. Tüm interface alanları örtük olarak `static` 'tir — hepsi tek bir paylaşılan değeri işaret eder, her implementasyon kendi kopyasını almaz (bkz. "Constant Alanlar").

2. Diamond problem'de derleyicinin "mantıklı" bir seçim yapacağını beklemek. İki interface aynı imzalı çakışan default metotlar sağladığında, override etmek **zorunludur**; derleyici asla kendiliğinden birini seçmez (bkz. "Diamond Problem ve Çözümü").

3. `@FunctionalInterface` olmayan (birden fazla soyut metotlu) bir interface'e lambda atamaya çalışmak. Lambda yalnızca **tam olarak tek** soyut metotlu interface'lere atanabilir (bkz. "Functional Interface ve Lambda").

4. Private interface metotlarını Java 8 sürümünde de kullanılabilir sanmak. Private metotlar yalnızca Java 9+ ile gelir; static ve default metotlar ise Java 8'den beri vardır — bu

ikisini karıştırmak, eski bir JDK hedefinde beklenmedik derleme hatalarına yol açar (bkz. "Tarihçe" ve "Private Metotlar").

5. Sealed bir interface'e yeni bir implementasyon eklerken `permits` listesini güncellemeyi unutmak. Yeni tip, `permits` listesine eklenmediği sürece derlenmez — bu kısıtlama kasıtlıdır, atlanacak bir hata değildir (bkz. "Sealed Interface").

6. Interface'i, aslında birbirine hiç ilişkisi olmayan metotları bir araya toplayan "her şeyi yapan" tek bir sözleşme haline getirmek. Bu, Interface Segregation Principle'ı ihlal eder ve implementasyonları gereksiz metotlarla implement etmeye zorlar (bkz. "Best Practices").

Özet, Cheat Sheet ve Terimler Sözlüğü

Interface, Java'nın JDK 1.0'dan beri sahip olduğu, bir tipin sözleşmesini implementasyonundan ayıran temel yapı taşıdır. Öne çıkan noktalar:

- Bir interface metodu, gövdesizse örtük olarak `public abstract`'tır; bir interface alanı her zaman örtük olarak `public static final`'dır
- Bir sınıf `extends` ile yalnızca bir sınıftan, ama `implements` ile istediği kadar interface'ten miras alabilir
- Bir interface, `extends` ile başka bir (hatta birden fazla) interface'i genişletebilir
- Java 8: `default` (gövdeli, override edilebilir, otomatik miras alınır) ve `static` (interface'e ait, asla override edilemez) metotlar
- Java 9: `private` (yalnızca aynı interface'in default/static metotları tarafından çağrılabilen, dışa kapalı) metotlar
- İki default metot çakıştığında override zorunludur; `InterfaceAdı.super.metot()` ile belirli bir üst interface'in davranışı seçilebilir
- Tam olarak tek soyut metotlu bir interface *functional interface*'tir — lambda hedefi olarak kullanılabilir
- Java 17: `sealed` + `permits`, implement edebilecek tipleri kapalı bir listeyle sınırlar, `switch` ile birlikte derleyici destekli tüketici (exhaustive) kontrol sağlar

Hızlı referans:

```
// Temel tanım ve implementasyon
interface Shape {
    double area();                      // örtük: public abstract

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

    default String describe() {         // Java 8: gövdeli, override edilebilir
        return "area=" + area();
    }

    static Shape unit() {               // Java 8: interface'e ait, örneksiz çağrılır
        return () -> 1.0;
    }

    private double round(double v) {    // Java 9: yalnızca içeriden çağrılabilir
        return Math.round(v * 100) / 100.0;
    }
}

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

// Çoklu implement + interface genişletme
interface A { }
interface B { }
class C implements A, B { }
interface D extends A, B { }

// Diamond problem çözümü
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 { }
```

Terimler Sözlüğü

Interface (arayüz) — Bir tipin davranışını, implementasyondan bağımsız olarak tanımlayan sözleşme; `interface` anahtar kelimesiyle bildirilir.

Soyut metot (abstract method) — Gövdesi olmayan, örtük olarak `public abstract` sayılan interface metodu; onu implement eden her sınıf bir gövde sağlamak zorundadır.

`default` metot — Java 8 ile gelen, gövdesi olan ve implementasyonlar tarafından override edilebilen interface metodu.

`static` metot — Java 8 ile gelen, doğrudan interface adı üzerinden çağrılan, hiçbir implementasyona ait olmayan ve override edilemeyen interface metodu.

`private` **metot** — Java 9 ile gelen, yalnızca aynı interface'in default/static metotları tarafından çağrılabilen, dışarıya kapalı yardımcı metot.

Diamond problem — Bir sınıfın, iki farklı interface'ten aynı imzalı çakışan default metotlar miras almasıyla ortaya çıkan, açık override gerektiren belirsizlik durumu.

`InterfaceAdı.super.metot()` — Diamond problem'i çözmek için, belirli bir üst interface'in default metot implementasyonunu açıkça seçmeye yarayan söz dizimi.

Functional interface — Tam olarak tek soyut metoda sahip interface; lambda ifadelerinin ve metot referanslarının hedefi olabilir. `@FunctionalInterface` annotation'ı bunu derleme zamanında garanti eder.

`sealed` / `permits` — Java 17 ile gelen, bir interface'i implement edebilecek tiplerin tam listesini önceden sınırlayan anahtar kelimeler.

Marker interface — Hiçbir metot içermeyen, yalnızca bir tipe JVM/kütüphane tarafından tanınan özel bir özellik "işaretleyen" interface (`Serializable`, `Cloneable` gibi).

Interface Segregation Principle (ISP) — SOLID prensiplerinden biri; bir implementasyonu, ihtiyaç duymadığı metotları implement etmeye zorlayan geniş interface'ler yerine, küçük ve odaklı interface'ler tercih edilmesini savunur.

Ek: Mini Proje — Basit Plugin/Strateji Sistemi

Şimdiye kadar öğrendiklerimizi ("Çoklu Interface Implement Etmek", "Functional Interface ve Lambda") birleştirip, çalışma zamanında yeni davranışlar "takılabilen" küçük bir plugin sistemi yazalım. Fikir basit: bir `NotificationChannel` sözleşmesi tanımlıyoruz, kayıt defteri (registry) yalnızca bu sözleşmeye güveniyor — hangi somut sınıfların (ya da lambda'ların) kayıtlı olduğunu hiç bilmeden:

```
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`'nin kodu, `EmailChannel` ya da `SmsChannel`'ın var olduğunu hiç bilmiyor — hatta üçüncü kanalın bir sınıf bile olmadığına, doğrudan bir lambda olduğuna dikkat et. Gerçek dünyada Spring'in `@Component` taradığı arayüz implementasyonları, ya da bir ödeme sağlayıcısını (Stripe/iyzico gibi) tek bir `PaymentGateway` interface'i arkasına gizleyen bir e-ticaret sistemi, tam olarak bu deseni kullanır.

> Bu mini projede reflection'a hiç ihtiyaç duymadık — Reflection dersindeki "Basit Dependency Injection Container" mini projesiyle karıştırma: o container, hangi tipin hangi tipe ihtiyaç duyduğunu **çalışma zamanında keşfediyordu**; burada ise hangi implementasyonun kullanılacağına programcı `register(...)` çağrısıyla **açıkça** karar veriyor. İkisi de "somut sınıfı sabitlememe" fikrine hizmet ediyor, ama farklı mekanizmalarla.

Ek: Mini Proje — Event Bus (Yayınla/Abone Ol)

Son mini proje, `java.util.function.Consumer`'ı genişleten kendi functional interface'imizi (`OrderPlacedListener`) tanımlayıp, bunun üzerine minimal bir yayınla/abone ol (publish/subscribe) mekanizması kuruyor — "Functional Interface ve Lambda" ile "Interface'in Interface'i Genişletmesi" bölümlerinde gördüğümüz iki fikri birleştiriyor:

```
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>`, yeni bir soyut metot eklemiyor — yalnızca `Consumer`'ın `accept(T)`'sini daha anlamlı bir isimle sarmalıyor; bu yüzden hâlâ geçerli bir functional interface ve hâlâ bir lambda ile implement edilebiliyor. `EventBusDemo`'daki iki `subscribe(...)` çağrısına dikkat et: `EventBus`, kaç dinleyici olduğunu, onların ne yaptığını hiç bilmiyor — yalnızca her yayınlanan olayı, kayıtlı her `OrderPlacedListener`'a sırayla iletiyor.

> Bu desen, Spring'in `ApplicationEventPublisher` / `@EventListener` mekanizmasının, gerçek dünya karmaşıklığından (asenkron işleme, hata yönetimi, event hiyerarşisi) arındırılmış çekirdek halidir — az sayıda gevşek bağlı (loosely coupled) bileşenin birbirine doğrudan referans tutmadan haberleşmesi gerektiğinde tercih edilir.