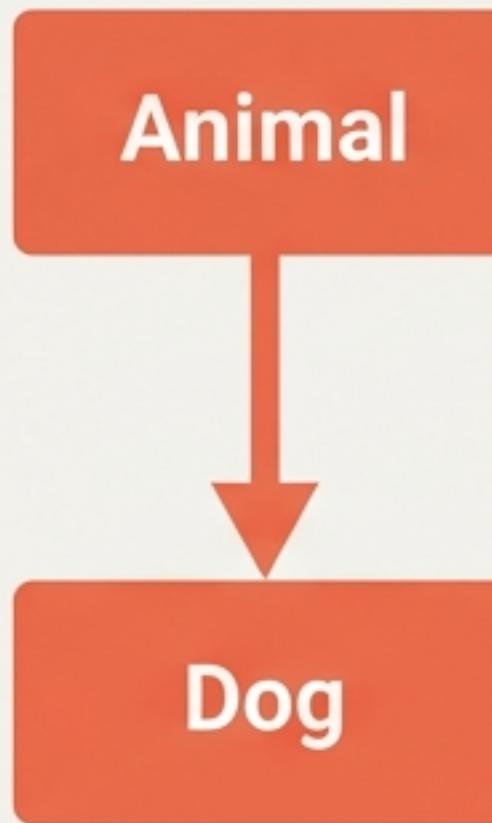


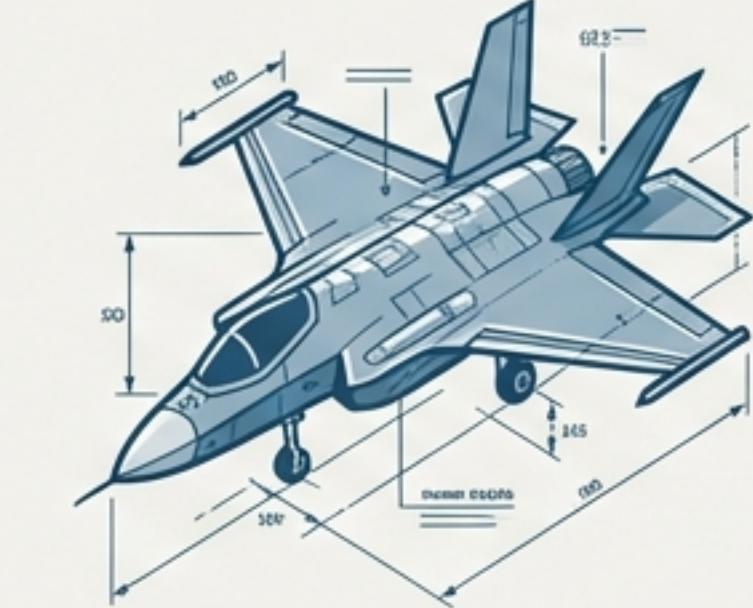
بخش ۱۴: رابطه‌ها (Interfaces) – قرارداد خالص رفتاری –

تهیه شده توسط: سید سجاد پیراهش

یک سوال اساسی...



ما می‌دانیم که "سگ" یک "حیوان" است. (رابطه IS-A)
این یعنی وراثت و یک خانواده مشترک.

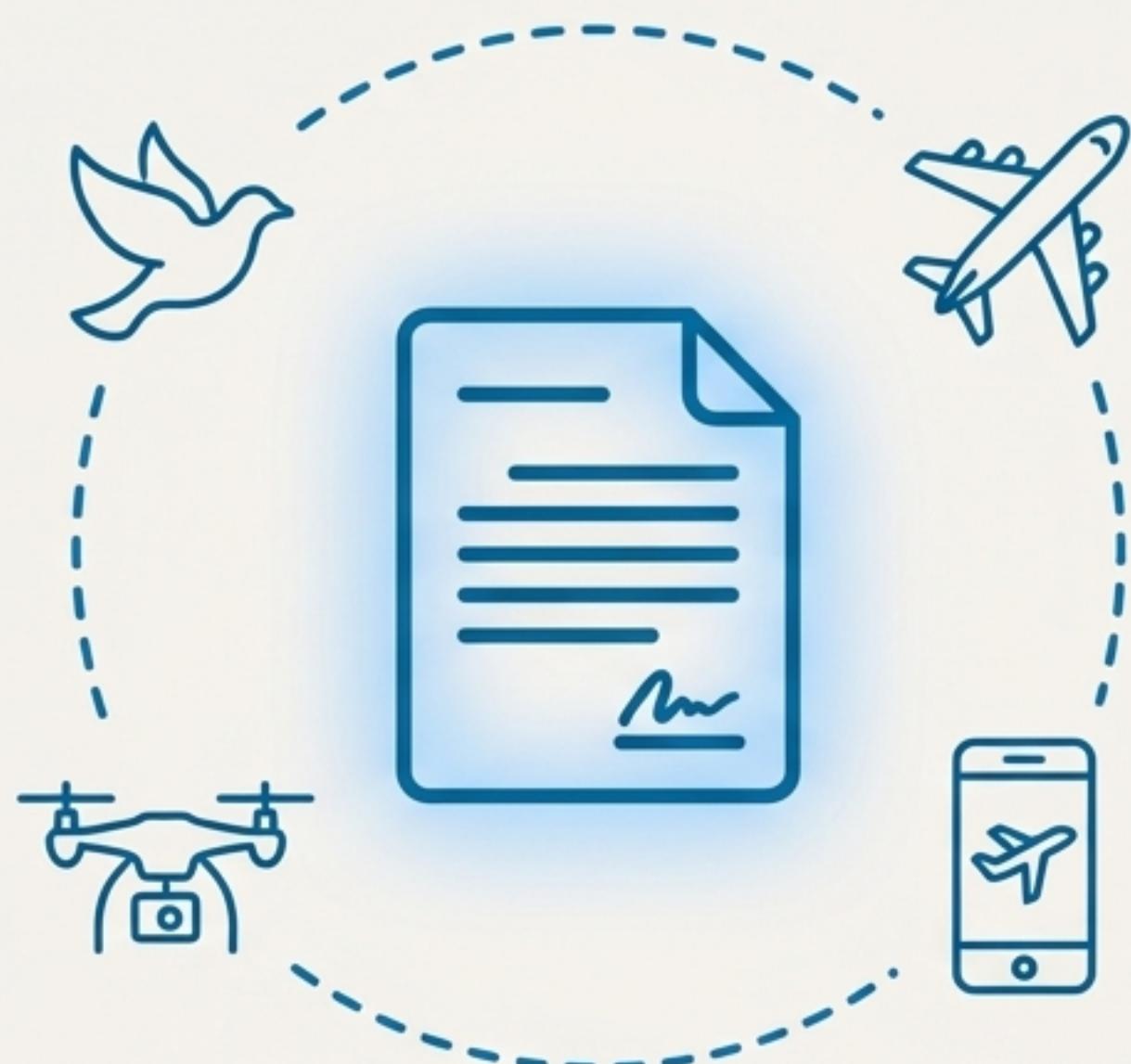


اما...

یک "پرنده" و یک "هوایپیما" هر دو می‌توانند پرواز کنند.
(رابطه CAN-DO)
آنها از یک خانواده نیستند و هیچ کد مشترکی ندارند.

چگونه این "توانایی" مشترک را مدل‌سازی کنیم؟

#معرفی Interface: قرارداد توانایی‌ها



یک Interface مانند یک عضویت در باشگاه است.

- **قانون باشگاه (Flyable Club):** "هر عضوی باید بتواند `fly()` کند."
- **شرط عضویت:** مهم نیست شما چه هستید (پرندگان، هواپیما، ...). اگر می‌خواهید عضو شوید، باید قوانین را رعایت کرده و متدهای `fly()` را پیاده‌سازی کنید.
- **مزیت بزرگ:** شما می‌توانید در چندین باشگاه به طور همزمان عضو باشید!

> رابطه‌ها به "چه کسی هستی" (وراثت) اهمیت نمی‌دهند، بلکه به "چه کاری می‌توانی انجام دهی" (رفتار) توجه دارند.

#کالبدشکافی یک Interface

یک قالب 100% انتزاعی برای رفتار است.

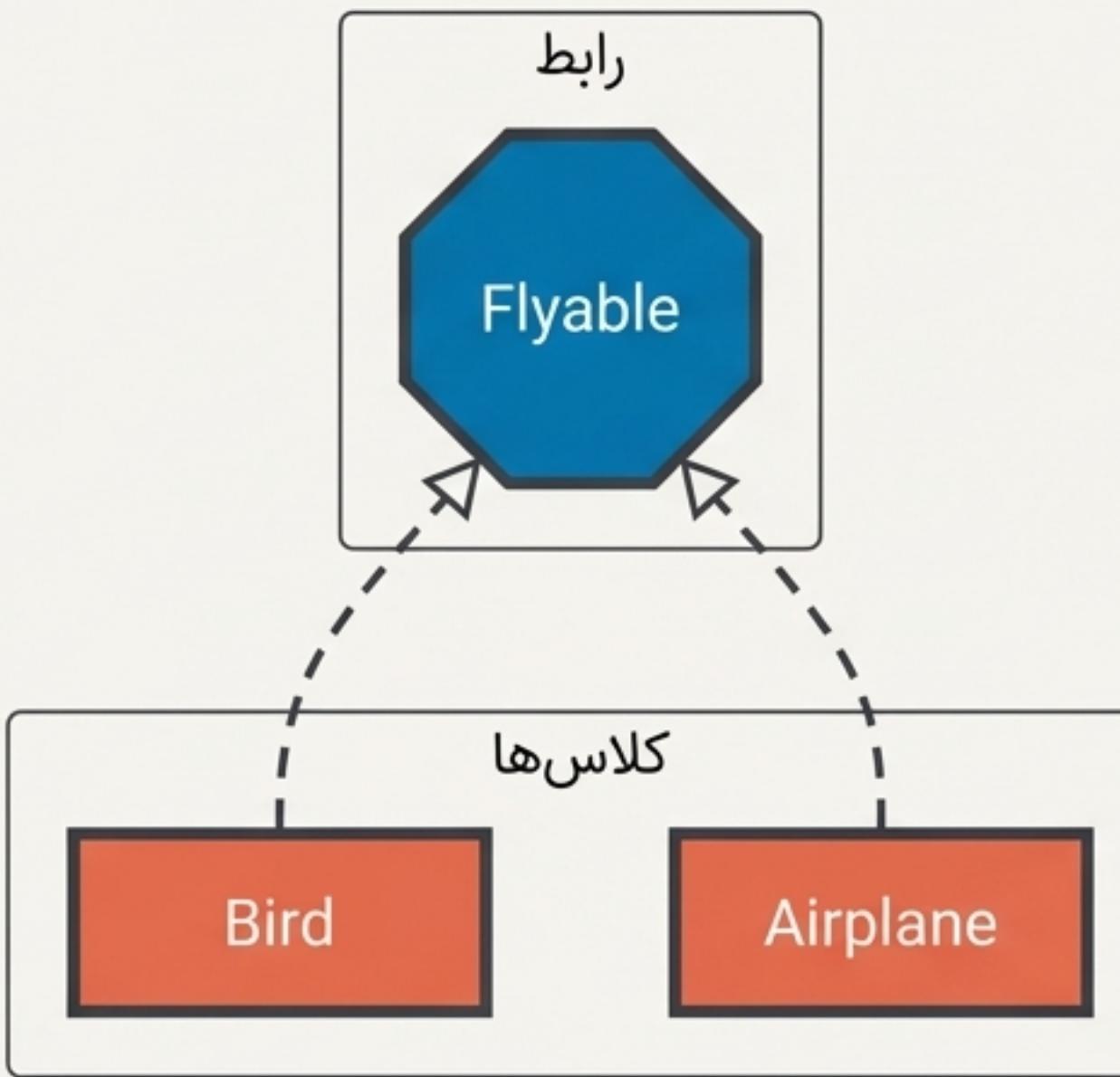
```
public interface Flyable {  
    //          // فیلد: به طور خودکار "public static final" است  
    //          // این یک ثابت است، نه یک متغیر!  
    int MAX_ALTITUDE = 40000;  
  
    //          // متدها: به طور خودکار "public abstract" است  
    //          // فقط امضا، بدون هیچ پیادهسازی.  
    void fly();  
    void land();  
}
```

قوانين کلیدی:

- **متدها:** همیشه `abstract` و `public` نوشتن نیست (نیازی به نوشتند).
- **فیلددها:** همیشه `public static final` (ثابت).
- **بدون سازنده (Constructor):** چون نمیتوان از آن شیء ساخت.
- **بدون وضعیت (State):** هیچ فیلد غیر ثابتی ندارد.

**`implements` کلمه کلیدی قرارداد: امضای##

یک کلاس با `implements` قول می‌دهد که تمام قوانین (متدهای) رابط را پیاده‌سازی کند.

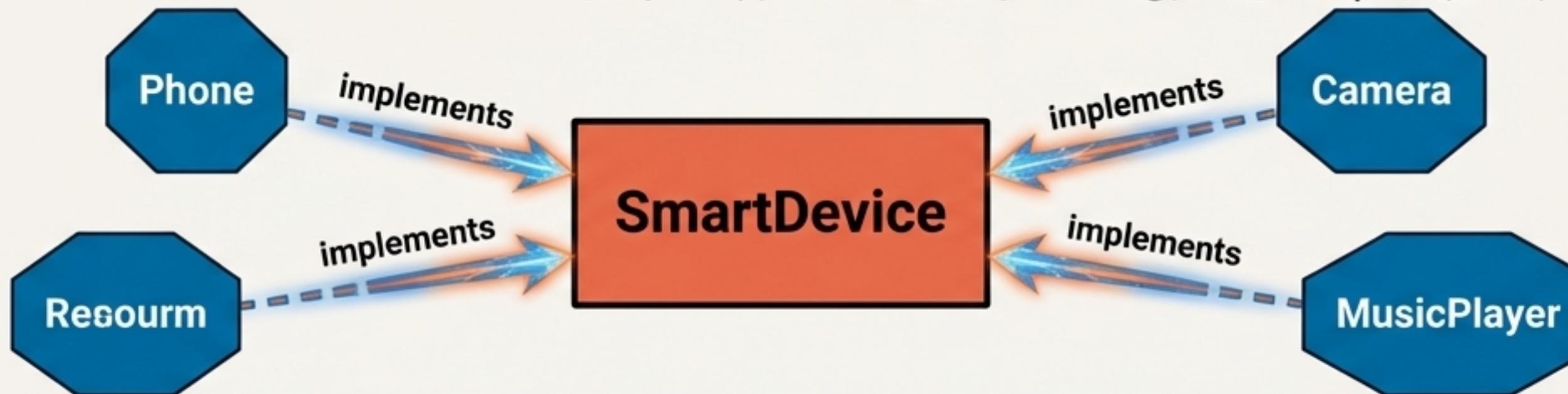


```
پرنده، با ارث بری از خانواده حیوانات //
1 public class Bird extends Animal implements Flyable {
2     @Override
3     public void fly() {
4         System.out.println("پرنده با بال زدن پرواز می‌کند");
5     }
6     @Override
7     public void land() {
8         System.out.println("پرنده روی شاخه فرود می‌آید.");
9     }
10 }
11 }
```

```
هوایپیما، یک ماشین //
1 public class Airplane implements Flyable {
2     @Override
3     public void fly() {
4         System.out.println("هوایپیما با موتور جت پرواز می‌کند");
5     }
6     @Override
7     public void land() {
8         System.out.println("هوایپیما روی باند فرود می‌آید.");
9     }
10 }
11 }
```

ابر قدرت نهایی: شکستن محدودیت وراثت یگانه!

در جاوا، یک کلاس فقط می‌تواند "یک پدر داشته باشد (extends). این یک محدودیت بزرگ است. اما... یک کلاس می‌تواند هر تعداد Interface را `implements` کند! این یعنی به ارث بردن چندین "نوع" یا "توانایی" به طور همزمان.



```
public interface Phone { void call(String number); }  
public interface Camera { void takePicture(); }  
public interface MusicPlayer { void play(String song); }
```

```
// این کلاس هر سه توانایی را دارد !  
public class SmartDevice implements Phone, Camera, MusicPlayer {  
    ... پیاده‌سازی تمام متدها // ...  
}
```

#*جدال نهایی: چه زمانی از کدام استفاده کنیم؟*

این مهمترین تصمیم در طراحی شیءگر است.



Abstract Class (کلاس انتزاعی)

- **هدف***: تعریف یک *هویت خانوادگی (IS-A).
- **کاربرد***: وقتی میخواهید کد و وضعیت مشترک را بین کلاس‌های مرتبط به اشتراک بگذارید.
- **مثال***: `Animal` پایه مشترکی برای `Dog` و `Cat` است. آنها وضعیت مشترک (`age` و رفتار مشترک (`sleep()` دارند.

Interface (رابط)

- **هدف***: تعریف یک *توانایی یا نقش (CAN-DO).
- **کاربرد***: وقتی میخواهید یک قرارداد رفتاری را برای کلاس‌های نامرتبط تعریف کنید.
- **مثال***: `Saveable` توانایی ذخیره شدن را به `User`, `Order` و `Product` می‌دهد، هرچند این کلاس‌ها هیچ ربطی به هم ندارند.

مقایسه فنی: جزئیات مهم هستند

معیار	Interface (رابط)	Abstract Class (کلاس انتزاعی)
هدف اصلی	قرارداد رفتاری (Can-Do)	پایه مشترک (Is-A)
وراثت چندگانه	پشتیبانی کامل	فقط وراثت یگانه
متدهای با بدنه	فقط با <code>default</code> (از جاوا ۸)	کاملاً پشتیبانی می‌شود
فیلدها	<code>public static final</code> ثابت)	هر نوع فیلدی ممکن است
(Constructor) سازنده	ندارد	(<code>super()</code> برای
سطح دسترسی متدها	public فقط	public , protected , default
کلمه کلیدی	implements	extends

#رابطها در دنیای واقعی: استانداردهای صنعتی

این مفاهیم فقط تئوری نیستند. شما هر روز از آنها استفاده می‌کنید!



Comparable<T>

باشگاه: "قابل مقایسه بودن"

قانون: متد `(compareTo()`) را پیاده‌سازی کن.

نتیجه: به جاوا می‌فهماند که چگونه اشیاء شما را مرتب کند. `(Collections.sort())`

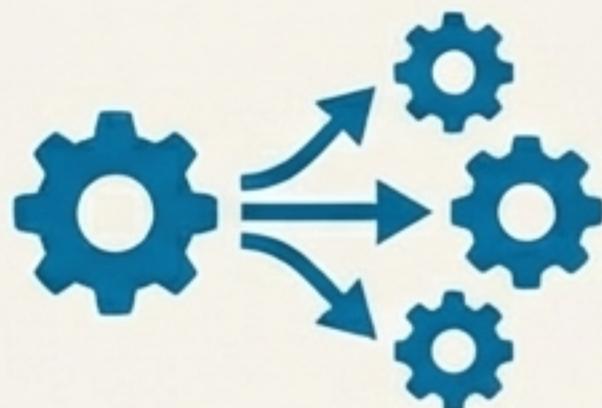


Serializable

باشگاه: "قابل سریالایز شدن" (تبدیل به بايت)

قانون: هیچ متدی ندارد! (یک Marker Interface است)

نتیجه: به JVM اجازه می‌دهد شیء شما را در فایل یا از طریق شبکه ارسال کند.



Runnable

باشگاه: "قابل اجرا بودن در یک ترد"

قانون: متد `(run())` را پیاده‌سازی کن.

نتیجه: کدی که می‌تواند به صورت موازی اجرا شود.

#نکته حرفه‌ای: اصل تفکیک رابط‌ها (ISP)

یک قانون طلایی: رابط‌های کوچک و متمرکز بهتر از یک رابط غولپیکر و همه‌کاره هستند.

Bad Design



Fat Interface

: (Fat Interface) X

```
interface SuperMachine {  
    void print();  
    void scan();  
    void fax();  
}
```

> مشکل: یک پرینتر ارزان قیمت که اسکنر ندارد، مجبور است متدهای `scan()` را به صورت خالی پیاده‌سازی کند!

Good Design



Lean Interfaces

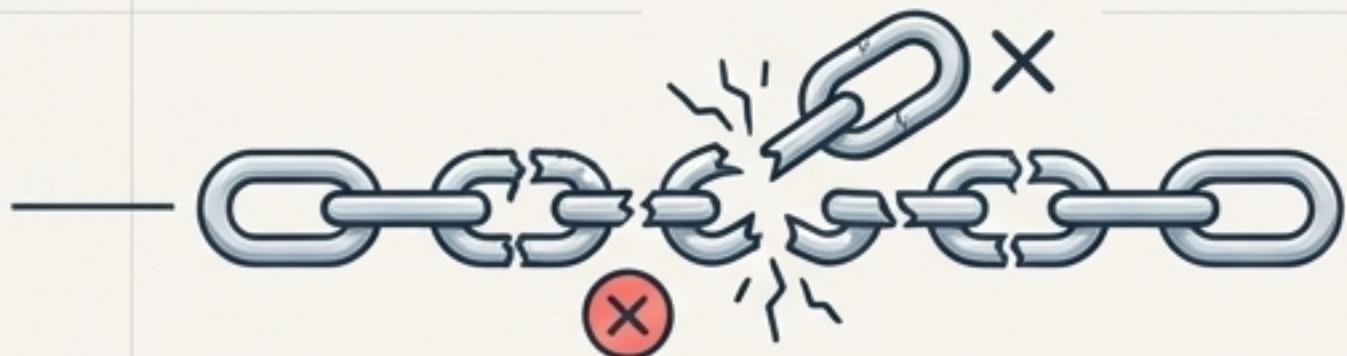
: (Lean Interfaces) ✓

```
interface Printer { void print(); }  
interface Scanner { void scan(); }  
interface Fax { void fax(); }  
  
// یک دستگاه چندکاره، هر سه باشگاه را عضو می‌شود.  
class AllInOne implements Printer, Scanner, Fax { ... }
```

یک پرینتر ساده فقط عضو باشگاه خودش است. //
class SimplePrinter implements Printer { ... }

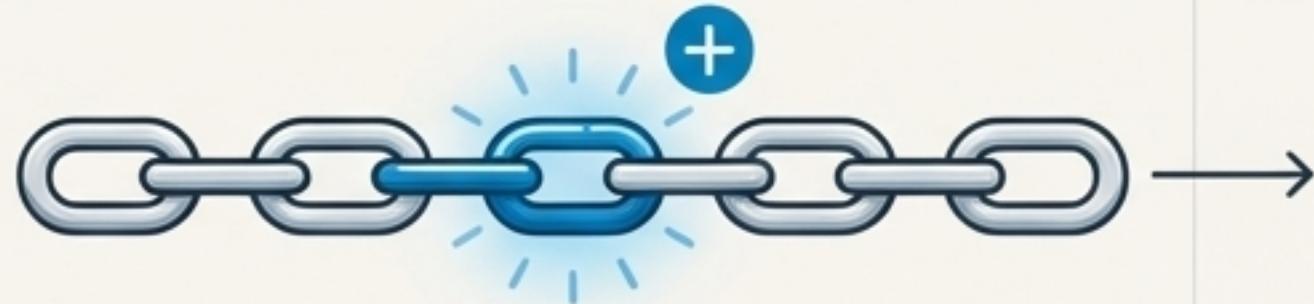
#تکامل رابط‌ها: چطور قوانین باشگاه را آپدیت کنیم؟*

Before Java 8



مشکل قدیمی: اگر یک متدهای جدید به یک رابط اضافه می‌کردید، تمام کلاس‌های پیاده‌ساز دچار خطای کامپایل می‌شدند!

After Java 8



راه حل (از جاوا ۸): متدهای default method یک اجازه به شما اجازه می‌دهد یک متدهای پیاده‌سازی پیش‌فرض را به رابط اضافه کنید.

```
public interface Vehicle {  
    void start();  
    void stop();  
  
    // این متدهای جدید است!  
    // کلاس‌های قدیمی مجبور به پیاده‌سازی آن نیستند.  
    default void honk() {  
        System.out.println("Beep Beep!");  
    }  
}  
  
public class Car implements Vehicle {  
    // ... start() و stop() پیاده‌سازی ...  
    // نیازی به پیاده‌سازی honk() نیست، از نسخه پیش‌فرض استفاده می‌کند.  
}
```

> این ویژگی، تکامل API‌ها را بدون شکستن کدهای قبلی ممکن می‌سازد.

#آزمون دانش: کدام ابزار برای کدام کار؟

برای هر سناریو، بهترین گزینه (Abstract Class یا Interface) را انتخاب کنید.



1. سناریو: تعریف انواع مختلف کارمندان (`Manager`, `Developer`) که همگی دارای `calculateSalary` و متدهای `employeeId` مشترک هستند.

پاسخ: Abstract Class (هویت و کد مشترک)



3. سناریو: مدل‌سازی انواع حساب بانکی (`Savings`, `Checking`) که ویژگی‌های مثل `balance` و `accountNumber` دارند.

پاسخ: Abstract Class (خانواده مرتبط)



2. سناریو: تعریف توانایی "ذخیره شدن در دیتابیس" (`Saveable`) برای اشیاء کاملاً نامرتبط (User, Product, Order).

پاسخ: Interface (توانایی خالص رفتاری)



4. سناریو: تعریف قابلیت "قابل مرتب‌سازی" (`Sortable`) برای هر شیئی که بخواهد در بخواهد در یک لیست مرتب شود.

پاسخ: Interface (یک قرارداد رفتاری گسترده)

#چالش کدنویسی: مرتبسازی انسان‌ها!

ماموریت شما:

1. کلاس Person زیر را در نظر بگیرید.
2. رابط implements را Comparable<Person> کنید.
3. متد compareTo(Person other) را طوری پیاده‌سازی کنید که Person ها بر اساس سن (age) از کم به زیاد مرتب شوند.
4. در main، یک لیست از Person ها بسازید و با Collections.sort() مرتب کنید.

```
// شروع کد
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class Person /* ??? */ {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // getters, toString...
}
```

راه حل چالش # Comparable

```
public class Person implements Comparable<Person> {
    private String name;
    private int age;

    public Person(String name, int age) { /* ... constructor ... */ }

    public int getAge() { return age; }

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

    // اینجا عضویت در باشگاه "قابل مقایسه بودن" تایید می شود!
    @Override
    public int compareTo(Person other) {
        if (this.age < other.age) {
            return -1;
        } else if (this.age > other.age) {
            return 1;
        } else {
            return 0;
        }
    }

    public static void main(String[] args) {
        List<Person> people = new ArrayList<>();
        people.add(new Person("Ali", 30));
        people.add(new Person("Zahra", 25));
        people.add(new Person("Reza", 35));

        Collections.sort(people); // جادو اینجا اتفاق می افتد!

        System.out.println(people);
        // خروجی: [Zahra (age 25), Ali (age 30), Reza (age 35)]
    }
}
```

[Zahra (age 25), Ali (age 30), Reza (age 35)]

خلاصه بخش ۱۴: جعبه ابزار جدید شما

شما اکنون به یکی از قدرتمندترین ابزارهای طراحی در جاوا مجهز شده‌اید.

یک قرارداد ۱۰۰% انتزاعی برای تعریف توانایی‌ها (Can-Do) 

: کلمه کلیدی برای عضویت در باشگاه و امضای قرارداد 

وراثت چندگانه: بزرگترین مزیت رابطها. یک کلاس می‌تواند چندین نقش را بپذیرد 

قوانين ساخت: فیلدها همیشه public abstract و متدها (به طور سنتی) public static final هستند 

: راهی برای تکامل رابطها بدون شکستن کدهای قدیمی 

انتخاب هوشمندانه: از Interface برای تعریف "توانایی" و Abstract Class برای تعریف "هویت خانوادگی" استفاده کنید 

در بخش بعدی: تکامل یک طراحی — از کلاس انتزاعی به رابط