

# بخش ۹: ارثبری (Inheritance) – ستون فقرات سلسله‌مراتب شیءگرا

تهییه شده توسط: سید سجاد پیراھش

# بمب ساعتی تکرار کد: کابوس یک توسعه دهنده

باید یک بازی بسازیم. ما سه نوع دشمن داریم: `Dragon`، `Orc` و `Goblin`. بدون ارثبری، کد ما این شکلی می‌شود:

Goblin.java

```
class Goblin {  
    int health = 50;  
    int attackPower = 10;  
  
    public void takeDamage(int amount) {  
        health -= amount;  
        System.out.println("Goblin took "  
            + amount + " damage!");  
    }  
  
    public void attack() {  
        // ... identical attack logic ...  
    }  
}
```

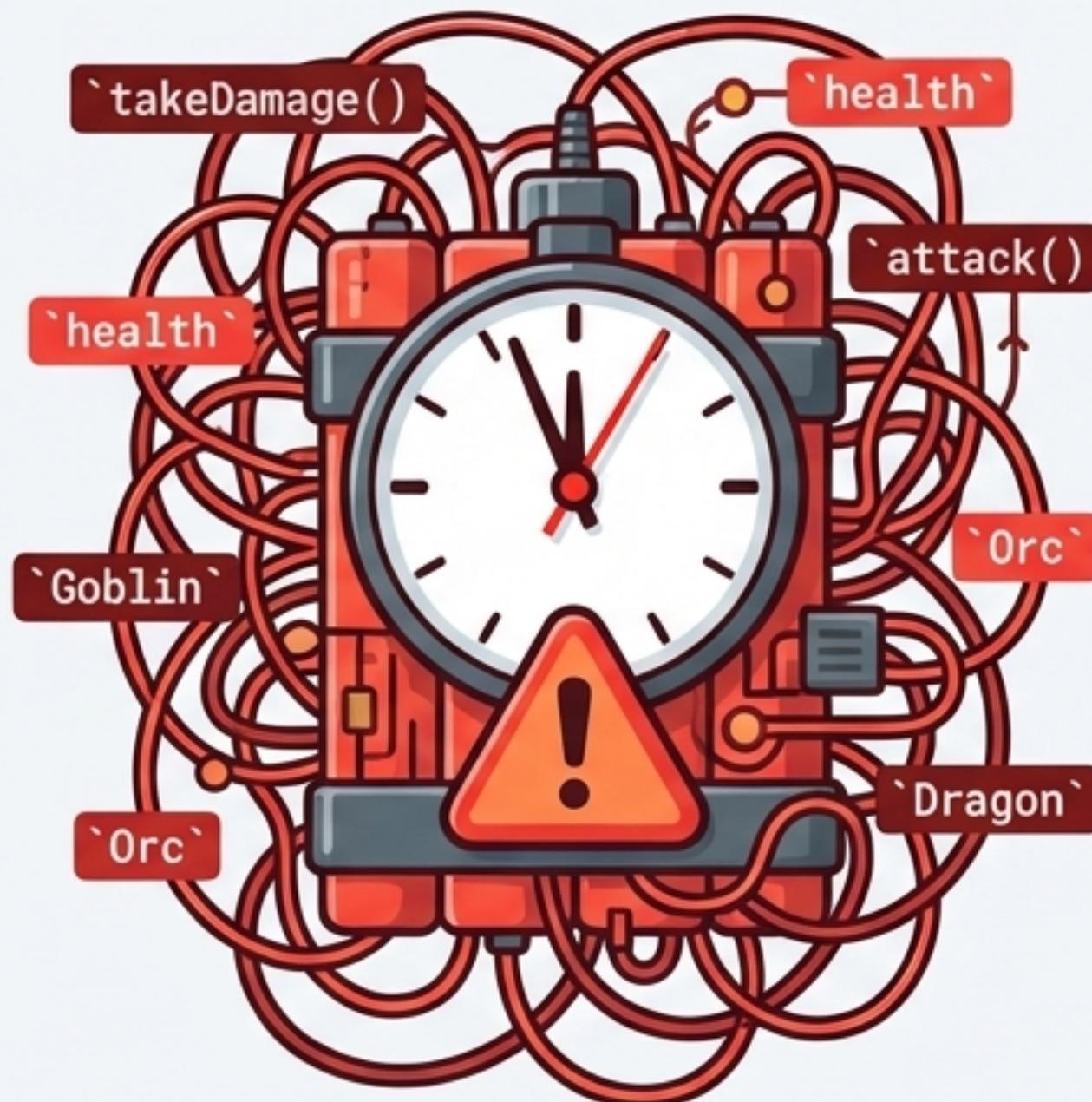
Orc.java

```
class Orc {  
    int health = 100;  
    int attackPower = 15;  
  
    public void takeDamage(int amount) {  
        health -= amount;  
        System.out.println("Orc took "  
            + amount + " damage!");  
    }  
  
    public void attack() {  
        // ... identical attack logic ...  
    }  
}
```

Dragon.java

```
class Dragon {  
    int health = 500;  
    int attackPower = 50;  
  
    public void takeDamage(int amount) {  
        health -= amount;  
        System.out.println("Dragon took "  
            + amount + " damage!");  
    }  
  
    public void attack() {  
        // ... identical attack logic ...  
    }  
}
```

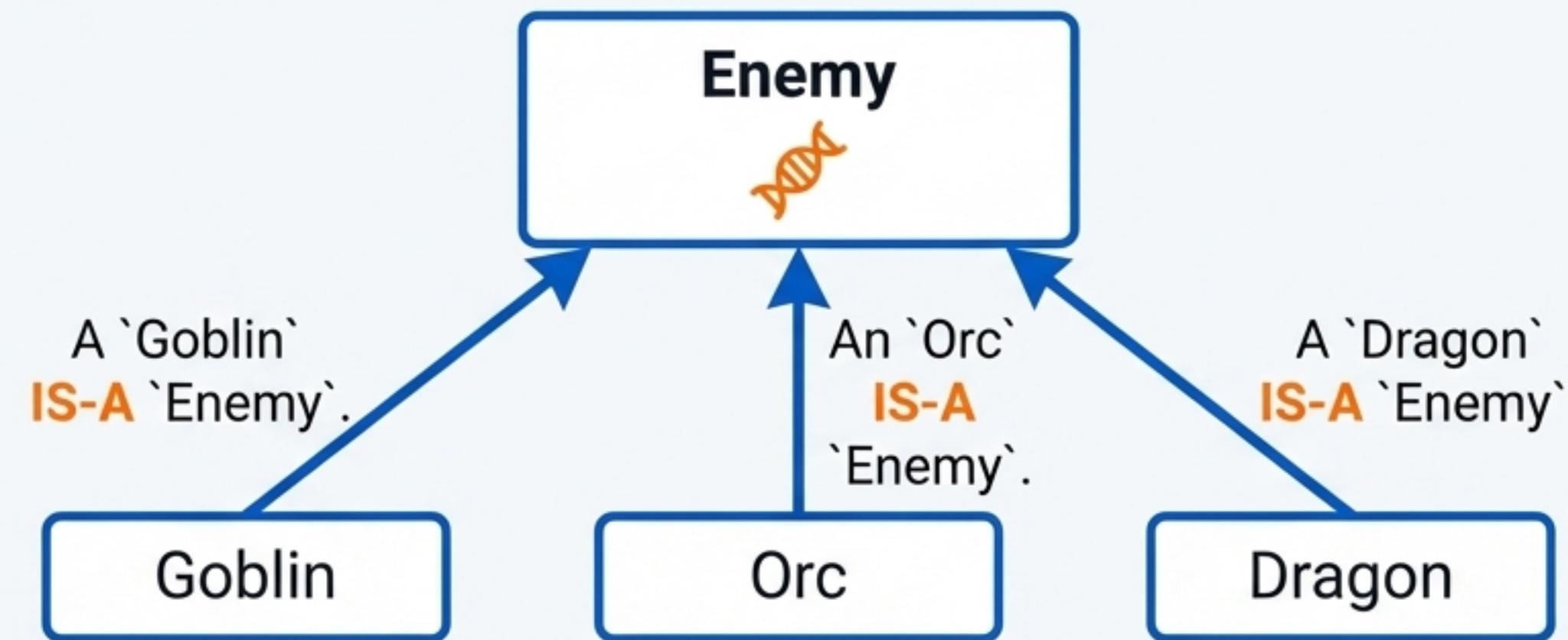
# تحلیل فاجعه: چرا این کد محکوم به شکست است؟



۱. **نقض کامل اصل اصل DRY (Don't Repeat Yourself)**  
ما فقط کد را تکرار نکرده‌ایم؛ ما 'منطق' و 'دانش' را تکرار کرده‌ایم.  
این یعنی هر تغییر کوچکی باید در چندین جا اعمال شود.
۲. **نگهداری غیرممکن (Maintenance Nightmare)**  
تصور کنید بخواهید 'شانس جاخالی دادن' را اضافه کنید. باید تک تک کلاس‌ها را ویرایش کنید. فراموش کردن یکی از آن‌ها یعنی ایجاد یک باگ پنهان و خطرناک.
۳. **عدم وجود رابطه منطقی (No Logical Connection)**  
از دید کامپایلر، 'Orc' و 'Goblin' هیچ ربطی به هم ندارند. شما نمی‌توانید یک تابع بنویسید که 'هر نوع دشمنی' را بپذیرد. آن‌ها فقط بسته‌های کد جدا از هم هستند.

# راه حل: مهندسی ژنتیک برای کد!

ارت بری به ما می‌گوید: "به جای تکرار، ویژگی‌های مشترک را در یک کلاس عمومی‌تر و پایه‌ای‌تر قرار بده." این کلاس، کلاس والد (Superclass) نامیده می‌شود.



این فرآیند مانند استخراج DNA مشترک است. کلاس‌های فرزند (Subclasses) این را به ارت می‌برند و ویژگی‌های منحصر به‌فرد خود را به آن اضافه می‌کنند.

# کلمه کلیدی `'extends'`: به ارث بردن قدرت

## گام ۱: ساخت کلاس والد (Enemy)

```
public class Enemy {  
    int health;  
    int attackPower;  
  
    public void takeDamage(int amount) {  
        health -= amount;  
        System.out.println("Enemy took " + amount +  
" damage!");  
    }  
  
    public void attack() {  
        // ... shared attack logic ...  
    }  
}
```

## گام ۲: ساخت کلاس‌های فرزند (Subclasses)

```
public class Goblin extends Enemy {  
    // Goblin-specific logic here, if any.  
    // All the power of Enemy is already inherited!  
}  
  
public class Orc extends Enemy {  
    // Orc-specific logic here, if any.  
}
```

منطق مشترک فقط در یک مکان. کد تمیز، قابل نگهداری و قدرتمند.

# بازنویسی متدها (@Override): تخصص‌گرایی فرزندان

یک کلاس فرزند می‌تواند رفتار به ارث برده شده را تغییر دهد یا "بازنویسی" کند. این به فرزند اجازه می‌دهد تا یک نسخه تخصصی از یک متده داشته باشد.

## Enemy

```
public class Enemy {  
    public void shout() {  
        System.out.println("Arrrgh!");  
    }  
}
```

## Dragon

```
public class Dragon extends Enemy {  
    @Override // This is crucial!  
    public void shout() {  
        System.out.println("ROOOOAR!"); // A  
        much more fitting shout  
    }  
}
```

## Why @Override is Your Safety Net



به کامپایلر می‌گوید که قصد شما بازنویسی است. اگر املای متده را اشتباه بنویسید (مثلاً `shuot`)، کامپایلر به شما خطا می‌دهد و از یک تا بگ بزرگ جلوگیری می‌کند.



خوانایی کد را به شدت بالا می‌برد. هر کس کد شما را ببیند، فوراً متوجه می‌شود که این متده از کلاس والد بازنویسی شده است.

# کلمه کلیدی 'super': گفتگو با والد

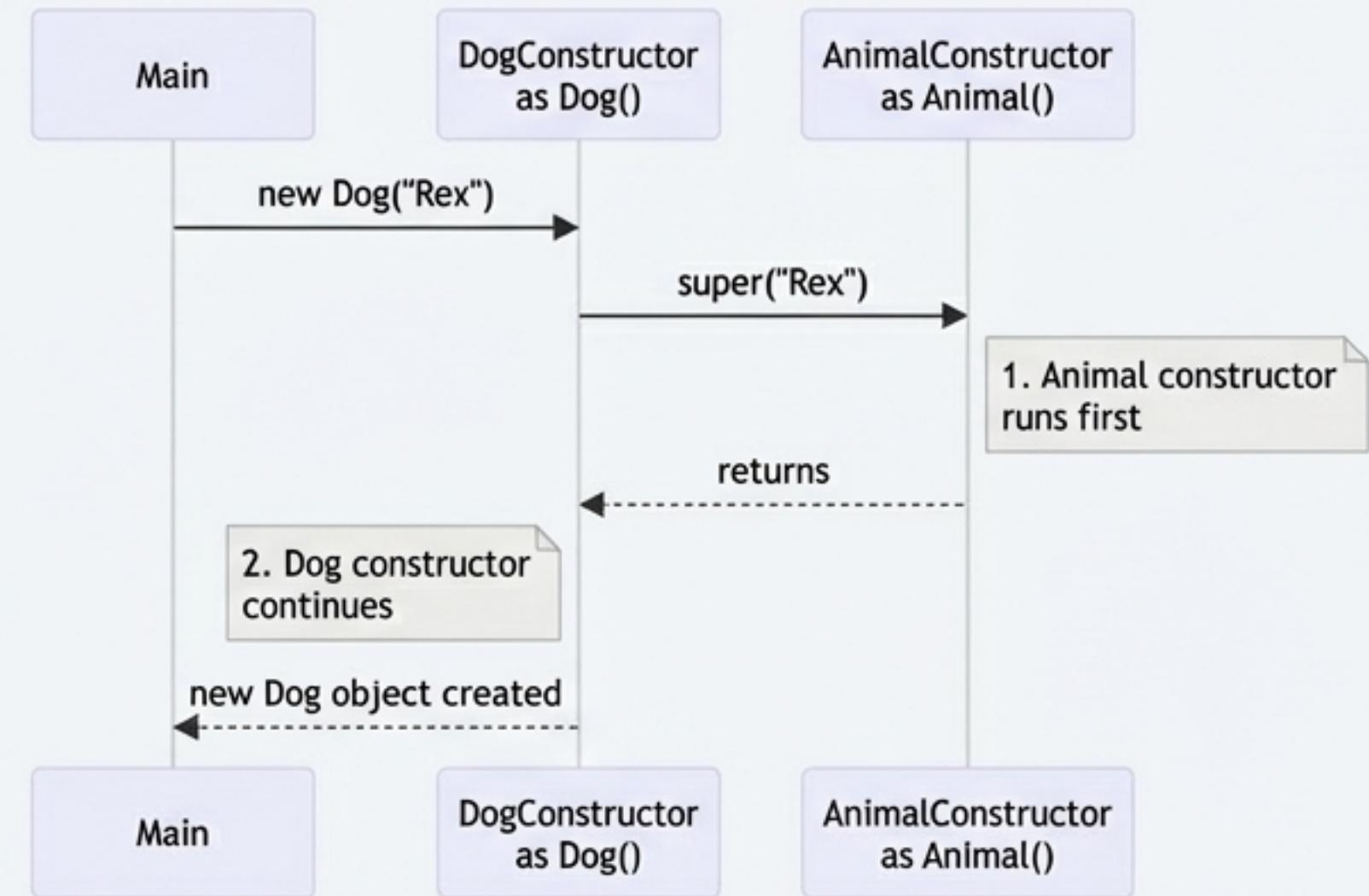
وقتی یک شیء Dog می‌سازیم، بخش Animal آن چگونه ساخته می‌شود؟

با استفاده از `super()`. این کلمه کلیدی دو کاربرد اصلی دارد:

- .1. `super()` : فراخوانی سازنده والد.
- .2. `super.method()` : فراخوانی متده والد.

فراخوانی `super()` باید اولین دستور در سازنده کلاس فرزند باشد.

```
class Animal {  
    String name;  
    Animal(String name) {  
        this.name = name;  
        System.out.println("1. Animal constructor called for: " +  
    }  
}  
  
class Dog extends Animal {  
    Dog(String name) {  
        super(name); // Must be the first line!  
        System.out.println("2. Dog constructor called.");  
    }  
}
```



# قدرت کامل `super` : فراتر از سازنده‌ها

\* گاهی نمی‌خواهیم رفتار والد را  
جایگزین کنیم، بلکه می‌خواهیم به آن  
\***اضافه\*** کنیم.

```
class Animal {  
    void eat() {  
        System.out.println("Animal eating generic food.");  
    }  
}  
class Dog extends Animal {  
    @Override  
    void eat() {  
        super.eat(); // First, do what an animal does...  
        System.out.println("Dog is now eating bones."); // ...then add specialized behavior.  
    }  
}
```

خروجی

1. Animal eating generic food.
2. Dog is now eating bones.

## `super` در برابر `this`

ویرگی	`this`	`super`
اشاره به	شیء فعلی	شیء والد
فراخوانی سازنده	سازنده دیگر در همان کلاس را فراخوانی <b>this()</b> می‌کند.	سازنده کلاس <b>super()</b> والد را فراخوانی می‌کند.
کاربرد اصلی	رفع ابهام بین فیلدهای کلاس و پارامترهای متدها.	دسترسی به اعضای بازنویسی شده در کلاس والد.

# قوانين دسترسی: چه چیزهایی به ارث می‌رسد؟

ارثبری بدون درک سطوح دسترسی می‌تواند به شدت خطرناک باشد. این کلمات کلیدی، "رازهای خانوادگی" کد شما را مشخص مشخص می‌کنند.

سطح دسترسی والد	کلمه کلیدی	دسترسی مستقیم در فرزند (پکیج دیگر)	توضیح و فلسفه
خصوصی	<b>private</b>		راز مطلق کلاس. حتی فرزند حق دانستن آن را ندارد. <code>private int secretCode;</code>
پکیج	<b>(default)</b>		راز خانوادگی در یک محله. فرزندان هم پکیج می‌دانند، اما فرزندان پکیج دیگر نه.
محفظت شده	<b>protected</b>		نقشه شیرین ارثبری! این "راز خانوادگی برای تمام نسل‌ها" است و دقیقاً برای این کار طراحی شده. <code>protected String heirloom;</code>
عمومی	<b>public</b>		دانش عمومی. همه، از جمله تمام فرزندان، آن را می‌دانند. <code>public String name;</code>

# قانون تک فرزندی: چرا جاوا اجازه ارث بری چندگانه را نمی دهد؟

در جاوا، یک کلاس فقط می تواند از یک کلاس والد ارث بری کند. (class D extends A, B) غیرممکن است).

## The "Why": The Diamond Problem (مشکل الماس)

این یک مشکل کلاسیک در زبان هایی مثل C++ است.

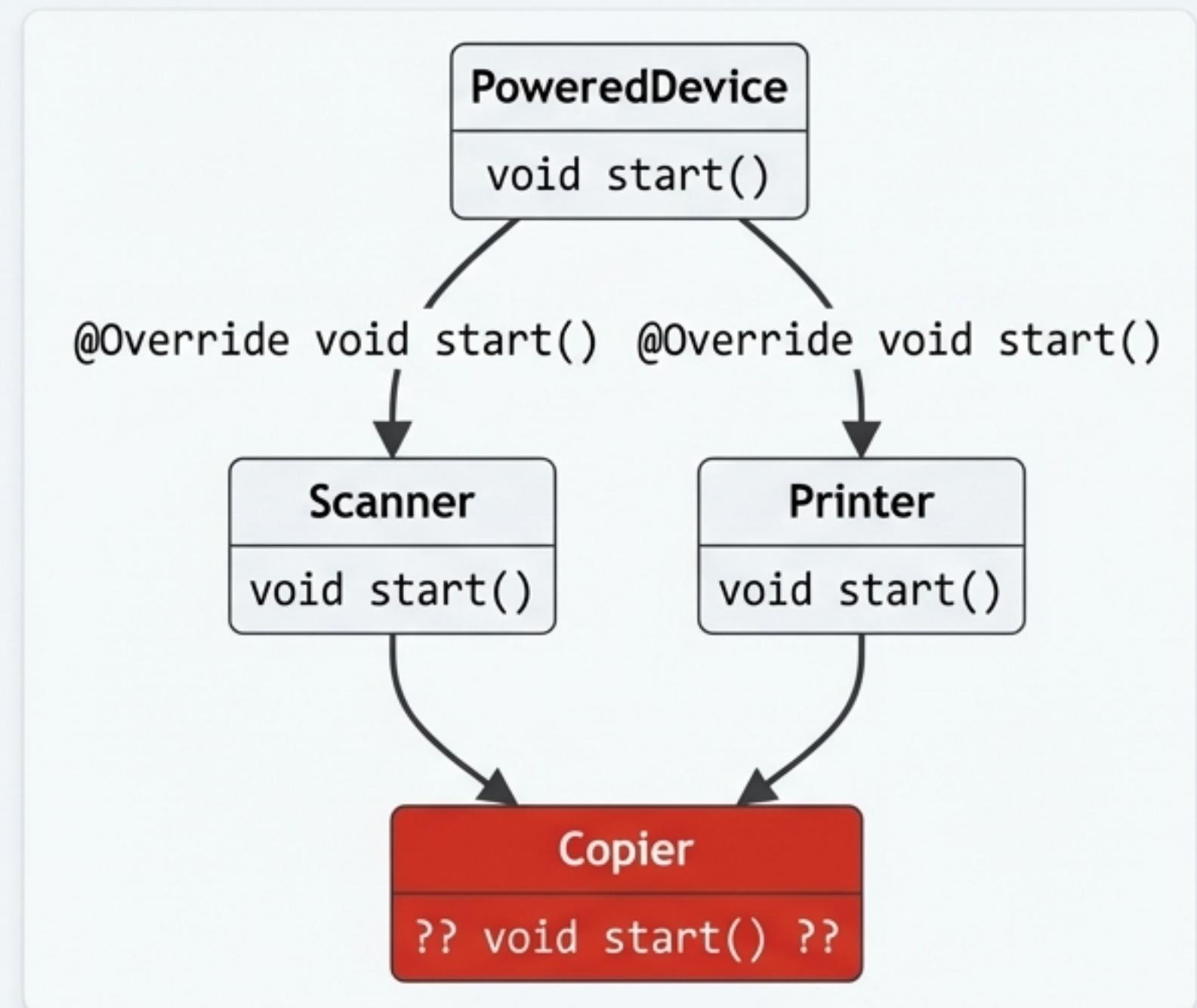
تصور کنید کلاس Scanner و Printer هر تصور کنید کلاس Printer سارند: هر دو یک متده است start(). اگر کلاس Copier از هر دوی آنها ارث ببرد، وقتی کامپایلر گیج می شود!

اگر کلاس Copier از هر دوی آنها ارث ببرد، وقتی کامپایلر گیج می شود!

## Java's Solution

جاوا با محدود کردن ارث بری به یک والد، این ابهام را از ریشه حذف می کند و زندگی را برای توسعه دهنده گان ساده تر می کند.

راه های دیگر برای رسیدن به قابلیت های مشابه (مانند Interface) بعداً بررسی خواهند شد.



# دو راهی بزرگ طراحی: ارثبری (IS-A) یا ترکیب (HAS-A)?

اصل طلایی: ترکیب را بر ارثبری ترجیح بده (Favor Composition over Inheritance).

ارثبری قدرتمند است، اما فقط برای روابط 'هست یک' (IS-A) واقعی استفاده می‌شود.  
برای بقیه موارد، ترکیب (Composition) یا رابطه 'دارد یک' (HAS-A) انتخاب بهتری است.

Inheritance (IS-A)	Composition (HAS-A)	معیار
مالکیتی یا عملکردی ('Car' یک 'Engine' دارد)	هویتی و ذاتی ('Dog' یک 'Animal' است)	نوع رابطه
بسیار پایین (کلاس‌ها از جزئیات هم بی‌خبرند)	بسیار بالا (فرزند به شدت به والد وابسته است)	وابستگی
زیاد (دینامیک). می‌توان اجزاء را در زمان اجرا تعویض کرد.	کم (استاتیک). رابطه در زمان کامپایل ثابت است.	انعطاف‌پذیری
در اکثر موارد دیگر؛ برای استفاده مجدد از عملکرد و ساختارهای انعطاف‌پذیر.	فقط زمانی که رابطه "IS-A" صد درصد برقرار است.	چه زمانی؟

# تله‌های طراحی: اشتباهات رایج در ارثبری



## 1. استفاده اشتباه از IS-A مشکل مربع-مستطیل

تفکر غلط: Square یک Rectangle است. پس:  
`.class Square extends Rectangle`

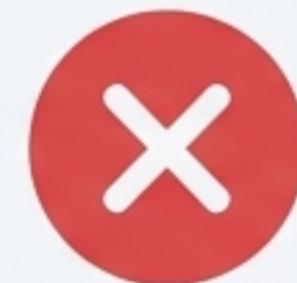
مشکل: `setWidth()` متدهای Rectangle دارد. تغییر عرض یک Square، "مربع بودن" آن را از بین می‌برد و قرارداد والد را نقض می‌کند. این نقض اصل Liskov است.



## 2. ارثبری فقط برای استفاده مجدد از کد

سناریوی غلط: چون کلاس UserRegistration به ارسال ایمیل نیاز دارد، منویسیم:  
`.class UserRegistration extends EmailSender`

چرا غلط است؟ UserRegistration یک EmailSender نیست.  
راه حل صحیح: ترکیب (Composition). کلاس UserRegistration یک EmailSender دارد.



## 3. فراموش کردن () super و سازنده پیشفرض

مشکل: اگر والد سازنده بدون آرگومان (پیشفرض) نداشته باشد و شما در فرزند () super را صراحةً صدا نزنید، کد شما کامپایل نخواهد شد.

# زمان چالش: مهارت‌های خود را آزمایش کنید

تئوری کافیست! این تمرین‌ها را حل کنید تا مفاهیم در ذهن شما حک شوند.

## 1. چالش زنجیره سازنده‌ها

خروجی دقیق کد زیر چیست؟ مراحل فرآخوانی را روی کاغذ دنبال کنید.

```
class A { A() { System.out.println("A"); } }
class B extends A { B() { System.out.println("B"); } }
class C extends B { C() { System.out.println("C"); } }
// In main: new C();
```

## 2. طراحی سلسله‌مراتب حساب بانکی

یک کلاس Account (والد)، و دو فرزند SavingsAccount (با نرخ سود) و CheckingAccount (با سقف اضافه‌برداشت) طراحی کنید. متدهای withdraw را در هر کلاس به درستی بازنویسی کنید.

## 3. کارآگاه روابط: IS-A یا HAS-A؟

برای جفت‌های زیر، نوع رابطه را مشخص کنید:

- (Person, Address)
- (Student, Person)
- (University, Department)

## 4. طراحی سلسله‌مراتب اشکال هندسی

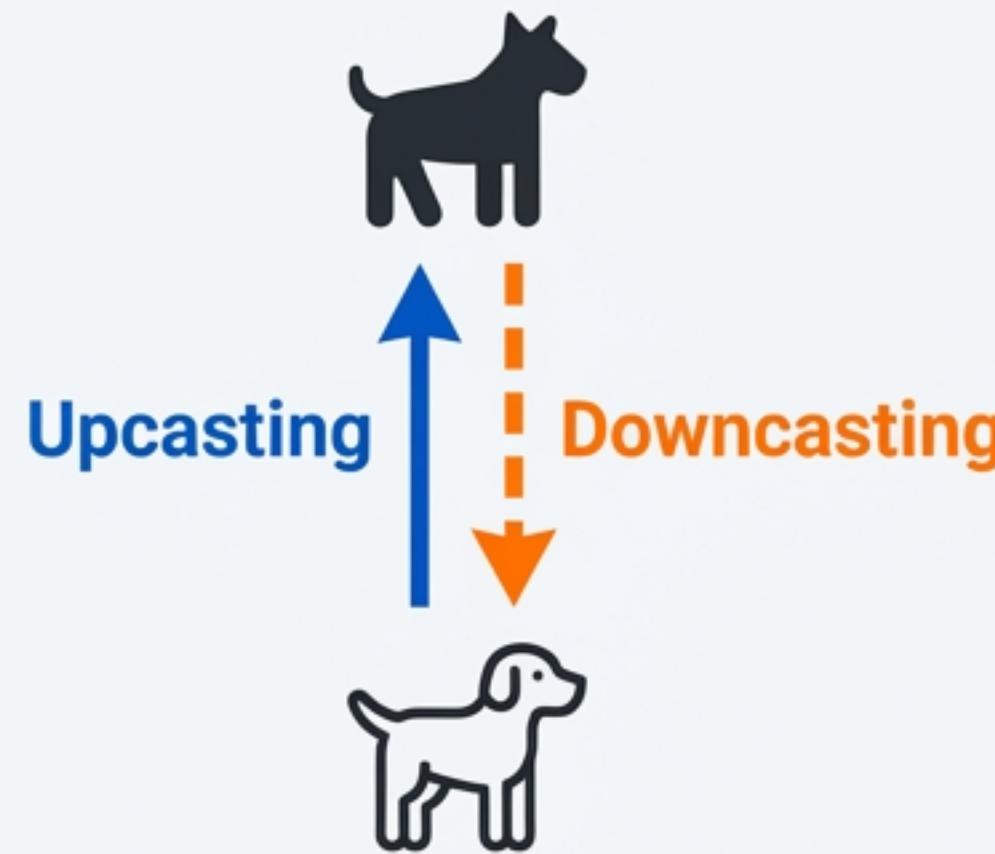
کلاس Shape (والد) و فرزندان Rectangle و Circle را بسازید. متدهای getArea() را در هر کدام بازنویسی کنید.

# جعبه ابزار ارثبری شما

- : مکانیزمی برای مدلسازی رابطه 'هست یک' و استفاده مجدد از کد. **Inheritance (IS-A)**
- : کلمه کلیدی برای ایجاد وراثت. (جاوا فقط از وراثت یگانه پشتیبانی می‌کند). **extends**
- : فراخوانی سازنده والد، که باید اولین دستور در سازنده فرزند باشد. **super()**
- : دسترسی به رفتار بازنویسی شده در کلاس والد. **super.method()**
- : آنوتیشنی برای تضمین صحت بازنویسی متدهای خوانایی کد. **@Override**
- : سطح دسترسی ایدهآل برای اعضایی که باید توسط فرزندان دیده شوند. **protected**
- : اصل طراحی کلیدی: ترکیب را بر ارثبری ترجیح دهید، مگر اینکه با یک رابطه IS-A واقعی رو برو باشید. **Composition over Inheritance**

# در بخش بعدی...

حالا که می‌دانیم چگونه سلسله‌مراتب بسازیم، وقت آن است که از قدرت واقعی آن استفاده کنیم.



## – هنر تغییر دید – Downcasting و Upcasting

- چگونه می‌توانیم با یک شیء `Dog` مانند یک `Animal` رفتار کنیم؟
- این قابلیت چگونه به ما اجازه می‌دهد کدی بنویسیم که با انواع مختلفی از اشیاء به صورت یکسان کار کند؟