

بخش ۱۲: چندریختی (Polymorphism) – اوج قدرت شیء‌گرایی

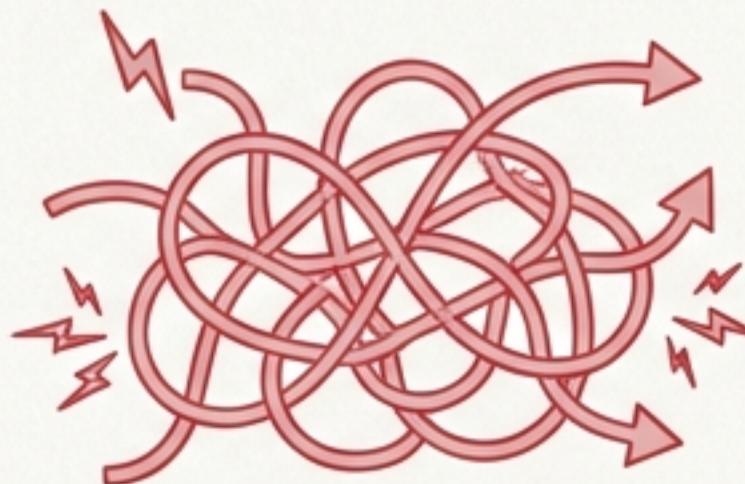
تهیه شده توسط: سید سجاد پیراھش

اگر شخصی از شما بپرسد: "قدرتمندترین ویژگی برنامه‌نویسی شیء‌گرایی چیست؟"، پاسخ یک کلمه است: **Upcasting**.. این مفهوم، اوج ترکیب وراثت، بازنویسی متدها و به ما اجازه می‌دهد کدهایی بنویسیم که انعطاف‌پذیر، قابل توسعه و قدرتمند هستند.



کابوس کدنویسی: مشکل بدون چندریختی

فرض کنید مدیر یک باغ وحش هستید و باید برای هر حیوان یک متدها دادن بنویسید.



کد اسپاگتی

مشکلات بدون چندریختی چندریختی

عدم انعطاف‌پذیری: با اضافه شدن هر حیوان جدید (مثلًاً یک پرنده)، باید کلاس `ZooKeeper` را دستکاری کنیم.

نقض آشکار اصل Open/Closed: کد ما برای تغییر باز است، نه برای توسعه.

نگهداری سخت و خوانایی ضعیف.



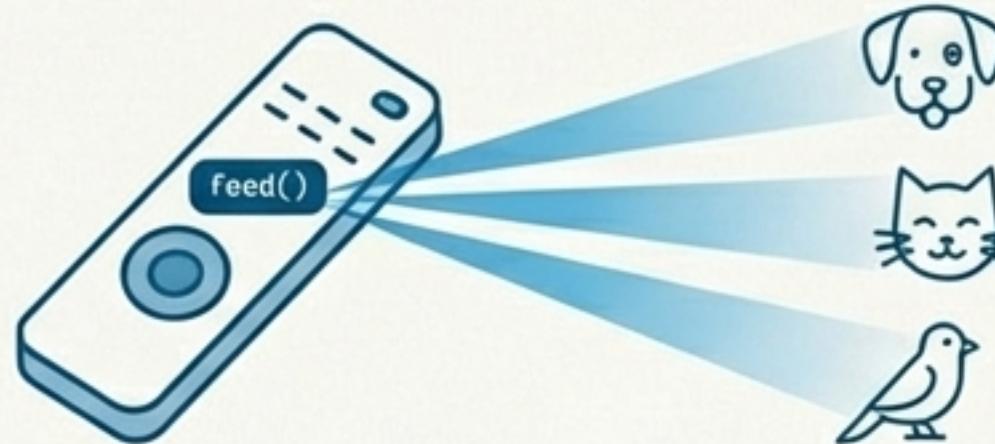
کد کابوس

```
// The "Before" state: rigid and repetitive
public class ZooKeeper {
    public void feedDog(Dog d) {
        System.out.println("Feeding the dog...");
        d.eat();
    }
    public void feedCat(Cat c) {
        System.out.println("Feeding the cat...");
        c.eat();
    }

    // What if we add a Bird? A Lion? A Tiger?
    // We have to keep changing this class!
}
```

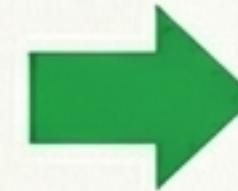
مسیر فروپاشی

راه حل جادویی: قدرت چندریختی



روش قدیمی

```
public class ZooKeeper {  
    public void feedDog(Dog d) {  
        System.out.println("Feeding the dog...");  
        d.eat();  
    }  
    public void feedCat(Cat c) {  
        System.out.println("Feeding the cat...");  
        c.eat();  
    }  
}
```



روش چندریختی

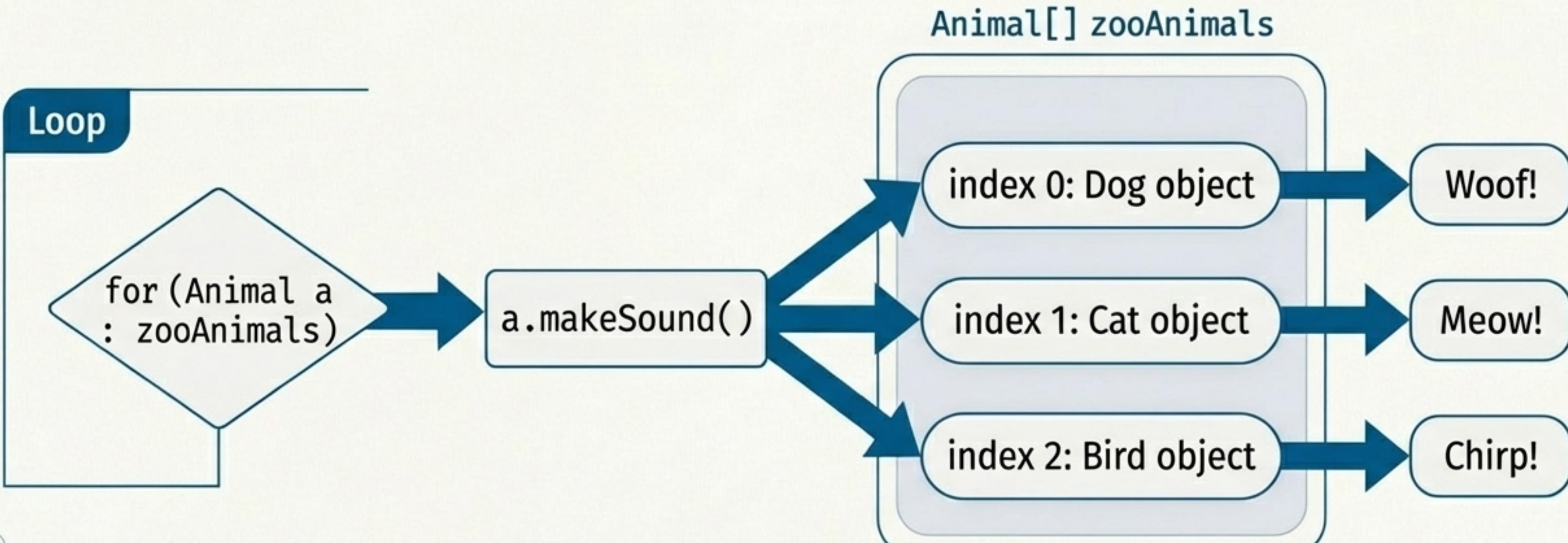
```
// The "After" state: clean, scalable, powerful  
public class ZooKeeper {  
    // One method to rule them all!  
    public void feedAnimal(Animal a) {  
        System.out.println("Feeding the animal...");  
        a.makeSound(); // The correct sound is made at runtime!  
        a.eat();  
    }  
}
```

یک متده برای همه. اضافه کردن Lion یا Bird یا دیگر نیازی به تغییر ZooKeeper ندارد. این قدرت واقعی OOP است.

چند ریختی واقعاً به چه معناست؟

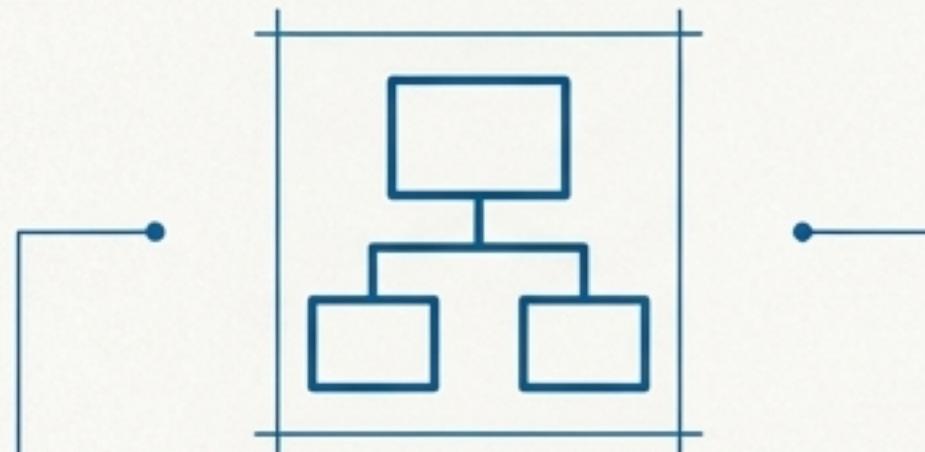
"یک رابط، چندین پیاده‌سازی". یک رفرنس از نوع **والد** می‌تواند به اشیاء از انواع مختلف **فرزندان** اشاره کند و رفتار مناسب در زمان اجرا **فراخوانی** می‌شود.

Poly = چند (Many)
Morph = شکل (Form)
Polymorphism = داشتن چندین شکل



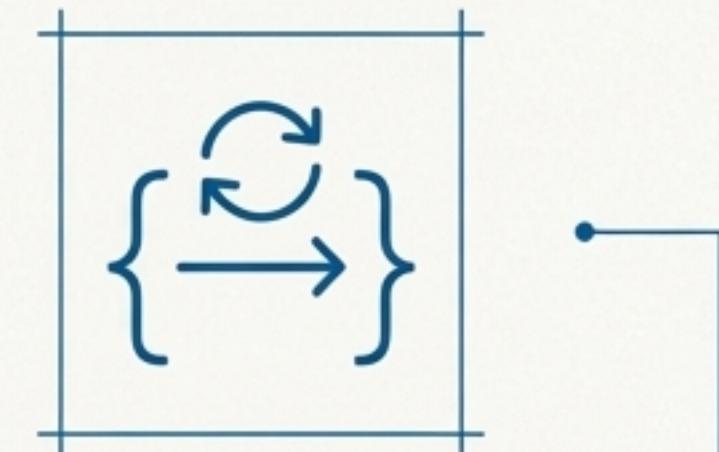
سه ستون اصلی که چندریختی را ممکن می‌سازند

برای اینکه جادوی چندریختی کار کند، هر سه شرط زیر باید برقرار باشند. بدون حتی یکی از آنها، این قدرت را از دست می‌دهید.



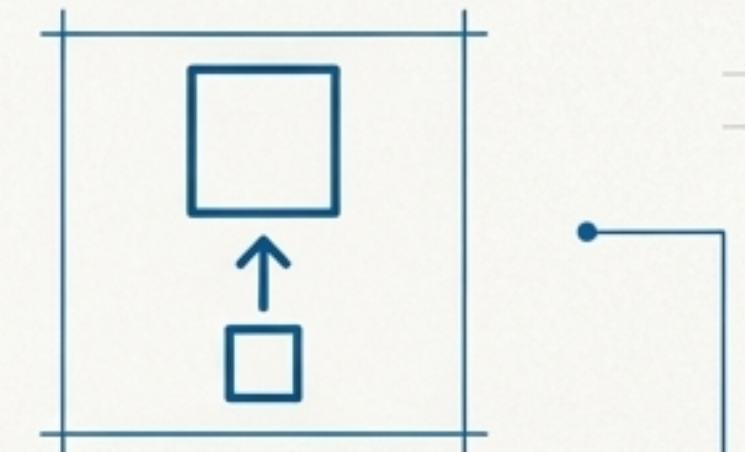
وراثت (Inheritance)

باید یک رابطه "IS-A" وجود داشته باشد. `Dog` یک `Animal` است. این پایه و اساس همه چیز است.



بازنویسی متدها (Method Overriding)

هر کلاس فرزند باید رفتار خاص خود را پیاده‌سازی کند. متد `makeSound()` در `Dog` با `Cat` متفاوت است.



ارجاع به بالا (Upcasting)

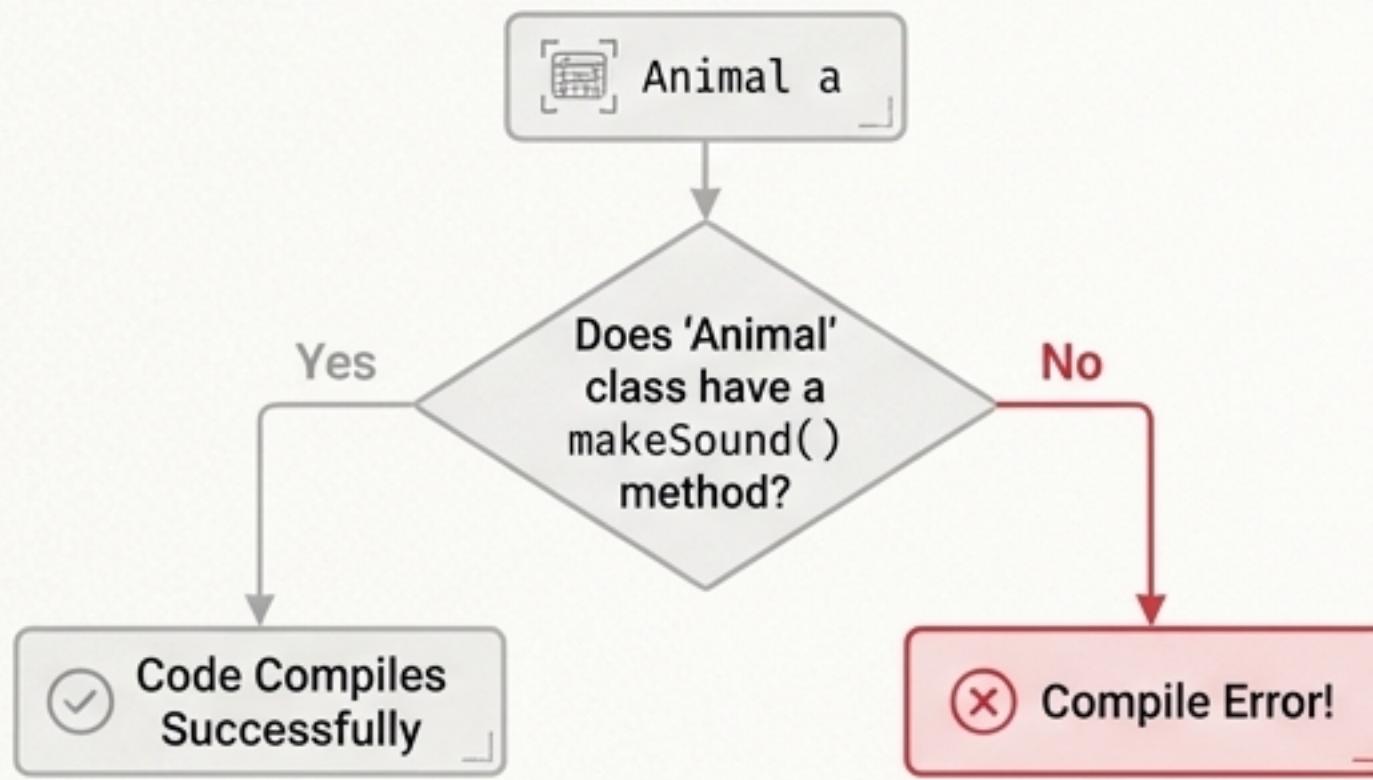
توانایی ارجاع دادن یک رفرنس از نوع والد به یک شیء از نوع فرزند. `Animal a = new Dog();` این همان چیزی است که به ما اجازه می‌دهد کد عمومی بنویسیم.

Polymorphism

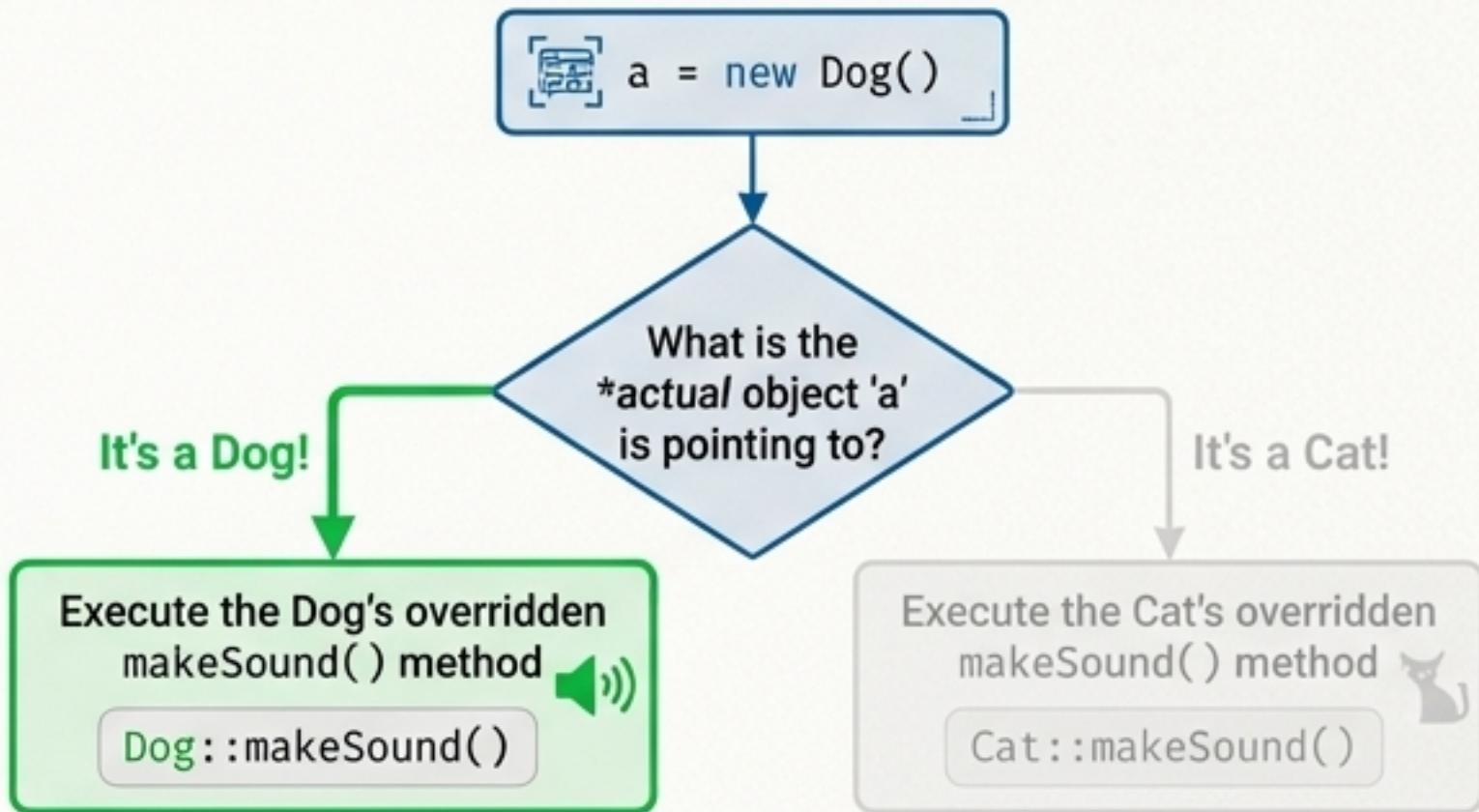
موتور اصلی: ارسال پویای متد (Dynamic Method Dispatch)

وقتی می‌نویسیم `Animal a = new Dog(); a.makeSound();` ماشین مجازی جاوا (JVM) چگونه تصمیم می‌گیرد کدام متد `makeSound()` را اجرا کند؟

(Compile-Time) دید کامپایلر



JVM (Runtime) دید



کامپایلر فقط نوع رفرنس (`Animal`) را برای اطمینان از وجود متد بررسی می‌کند. اما در زمان اجرا، JVM به شیء واقعی (`Dog`) نگاه می‌کند و متد بازنویسی شده در آن شیء را اجرا می‌کند. این فرآیند **Virtual Method Invocation** نامیده می‌شود.

دو نوع چندریختی در جاوا

در حالی که تمرکز ما بر روی قدرت چندریختی در زمان اجرا است، نوع دیگری نیز وجود دارد که در زمان کامپایل اتفاق می‌افتد.

| نوع | زمان تشخیص | mekanizm | مثال | کاربرد اصلی |
|--|--------------|-------------------------------|---|---|
| Static Polymorphism (Compile-time) | زمان کامپایل | Overloading | <code>print(int x) print(String s)</code> | داشتن چندین نسخه از یک متدهای پارامترهای متفاوت برای راحتی کار. |
| Dynamic Polymorphism (Runtime) | زمان اجرا | Overriding + Upcasting | <code>Animal a = new Dog(); a.makeSound();</code> | قلب چندریختی. ایجاد کدهای انعطاف‌پذیر که رفتارشان در زمان اجرا مشخص می‌شود. |

اثبات در عمل: طراحی یک سیستم پرداخت انعطاف‌پذیر

یک فروشگاه آنلاین را تصور کنید که نیاز به پشتیبانی از روش‌های پرداخت مختلف دارد.

```
// The abstraction  
abstract class Payment { abstract void process(double amount); }  
  
// Concrete implementations  
class CreditCardPayment extends Payment {  
    @Override void process(double amount) { /* Logic for credit card API... */ }  
}  
class PayPalPayment extends Payment {  
    @Override void process(double amount) { /* Logic for PayPal API... */ }  
}  
class BitcoinPayment extends Payment {  
    @Override void process(double amount) { /* Logic for Bitcoin wallet... */ }  
}
```

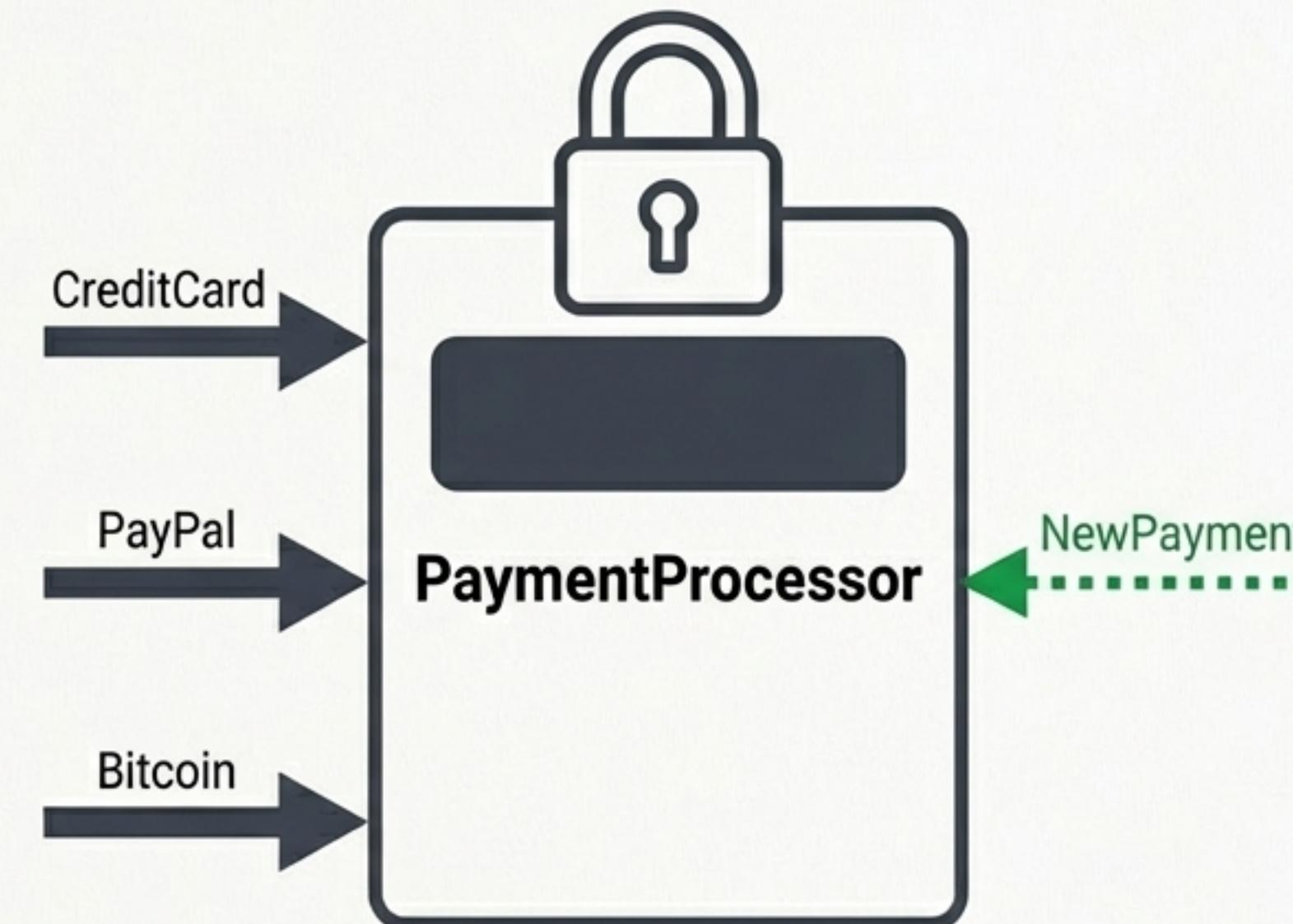
```
// The polymorphic processor  
public class PaymentProcessor {  
    // This method doesn't care about the details!  
    public void executePayment(Payment paymentMethod, double amount) {  
        System.out.println("Processing payment...");  
        paymentMethod.process(amount); // Dynamic dispatch happens here!  
    }  
}
```

برای اضافه کردن یک روش پرداخت جدید (مثلًا ApplePayPayment)، نیازی به تغییر حتی یک خط از کلاس PaymentProcessor نیست.

ابرقدرت معماری: اصل باز-بسته (Open-Closed Principle)

تعریف (Definition)

- **:Open for Extension** شما باید بتوانید قابلیت قابلیت‌های جدیدی به سیستم اضافه کنید (مثلاً کلاس‌های فرزند جدید).
- **Closed for Modification** شما نباید برای این کار مجبور به **تغییر کدهای موجود** و تست شده (مانند کلاس ZooKeeper یا PaymentProcessor) شوید.



ارتباط (The Connection)

چند ریختی مکانیزم اصلی برای پیاده‌سازی اصل Open-Closed برنامه‌نویسی بر اساس **انتزاع (Abstraction)** به جای **پیاده‌سازی‌های مشخص (Concrete Implementations)** ما کدی می‌نویسیم که پایدار است و در عین حال به راحتی توسعه می‌یابد.

تفاوت پنیادین: نگاهی مقایسه‌ای

| معیار | کد بدون چندریختی (switch یا if-else) | کد با چندریختی |
|---------------------------|---|---|
| پیچیدگی (Complexity) | بالا و فزاینده. هر نوع جدید یک if جدید است. | پایین. منطق اصلی ساده و متمرکز است. |
| نگهداری (Maintenance) | کابوس. تغییر در یک بخش، کل سیستم را تهدید می‌کند. | آسان. کلاس‌ها مستقل هستند. |
| توسعه‌پذیری (Scalability) | بسیار ضعیف. افزودن قابلیت جدید در دنگ است. | عالی. فقط کلاس جدید را اضافه کنید. |
| خوانایی (Readability) | ضعیف. پر از شرط‌های تو در تو. | بالا. object.doSomething(). بسیار واضح است. |
| Open-Closed | نقض شده | رعايت شده |

تله رایج: مقاومت در برابر چندریختی

یک اشتباه رایج، استفاده از `instanceof` و Type Checking به جای اعتماد به چندریختی است. این کار تمام مزایای آن را از بین می‌برد.

The Wrong Way ✗

```
// Anti-Pattern: Violates OCP
public void drawShapes(Shape[] shapes) {
    for (Shape s : shapes) {
        if (s instanceof Circle) {
            // draw circle logic...
        } else if (s instanceof Rectangle) {
            // draw rectangle logic...
        } // Must add a new 'if' for Triangle!
    }
}
```

نیاز به تغییر در کد اصلی ↗

The Right Way ✓

```
// Correct Polymorphic Way
public void drawShapes(Shape[] shapes) {
    for (Shape s : shapes) {
        s.draw(); // Let the object decide
        how to draw itself!
    }
}
```

چندریختی در عمل: خود شی تصمیم می‌گیرد

استفاده از `instanceof` یک "بوی کد" (Code Smell) است که نشان می‌دهد احتمالاً معماری شما نیاز به بازنگری دارد.



دانش خود را به چالش بکشید: تمرین‌های طراحی

برای هر سناریو، یک ساختار کلاسی با استفاده از وراثت و چندریختی طراحی کنید.



Challenge 1: Enemy System for a Game

- **Base Class:** `Enemy` با متدهای `attack()` و `takeDamage(int amount)`.
- **Subclasses:** `Goblin` (حمله سریع، آسیب کم) (`Orc`, `Dragon` (حمله آهسته، آسیب زیاد کم)، (حمله آتشین)).
- **Goal:** `Game` بنویسید یک کلاس `Enemy[]` را مدیریت کند و یک متد `triggerAllAttacks()` داشته باشد.



Challenge 2: Notification Service

- **Base Class:** `Notification` با متد `send(String message)`.
- **Subclasses:** `EmailNotification`, `SMSNotification`, `PushNotification`.
- **Goal:** `NotificationService` بنویسید که از طریق تمام کانال‌ها ارسال کند. `sendToAll(Notification[] channels, String msg)` (بساز که یک پیام را بسازید).



Runtime را پیش‌بینی کنید: آزمون

با توجه به مفاهیم Dynamic Method Dispatch، خروجی کد زیر چیست و چرا؟

```
class A {  
    public void m1() { System.out.println("A's m1"); }  
    public void m2() { System.out.println("A's m2"); }  
}  
class B extends A {  
    @Override  
    public void m1() { System.out.println("B's m1"); }  
    public void m3() { System.out.println("B's m3"); }  
}  
public class Main {  
    public static void main(String[] args) {  
        A obj1 = new B();  
        obj1.m1(); // Line 1  
        obj1.m2(); // Line 2  
  
        // obj1.m3(); // What happens if we uncomment this? Why?  
    }  
}
```

خروجی خط ۱ چیست؟

1

خروجی خط ۲ چیست؟

2

چرا خط obj1.m3() یک خطای کامپایل ایجاد می‌کند؟

3

خلاصه نهایی: قدرت چندریختی در دستان شما



یک رفرنس، چندین رفتار: قلب چندریختی.



سه شرط اصلی: وراثت + بازنویسی متدها + Upcasting.



تصمیم در زمان اجرا (**Runtime Binding**): JVM بر اساس شیء واقعی تصمیم می‌گیرد، نه نوع رفرنس.



اصل Open-Closed: کد شما برای توسعه باز و برای تغییر بسته است.



اجتناب از `instanceof`: به جای بررسی نوع، به چندریختی اعتماد کنید.



کد انعطاف‌پذیر و قابل نگهداری: این هدف نهایی و بزرگترین دستاورده استفاده از چندریختی است.

شما اکنون قدرتمندترین ابزار در جعبه ابزار شیء‌گرایی را در اختیار دارید. از آن برای ساختن نرم‌افزارهای زیبا، انعطاف‌پذیر و ماندگار استفاده کنید.