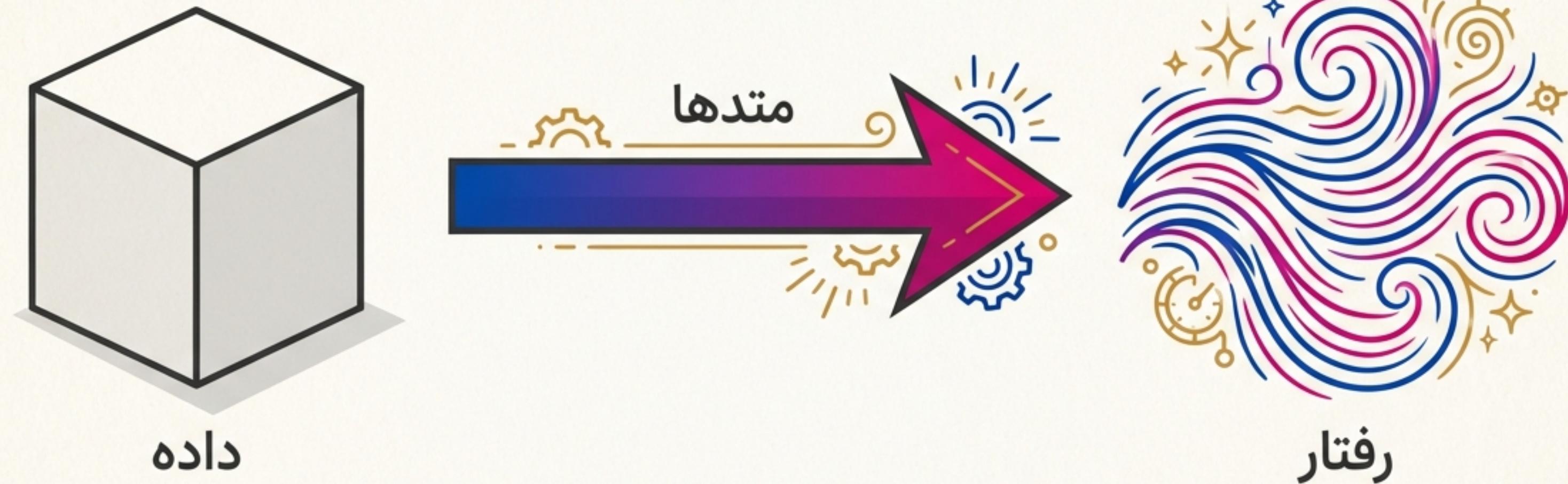


بخش ۵: متدها – تعریف رفتار اشیاء

تهیه شده توسط: سید سجاد پیراهش

اشیاء فقط داده نیستند، آن‌ها عمل می‌کنند.



اگر فیلدات "اسم‌ها" (دانش) یک شیء هستند، **متدها "فعال‌ها"** (اعمال) آن هستند. متدها پلی هستند که اطلاعات ثابت را به رفتار پویا تبدیل می‌کنند. اینجا جایی است که ما به کدهایمان روح می‌بخشیم.

کالبدشکافی یک متده‌های همیت و اجرا

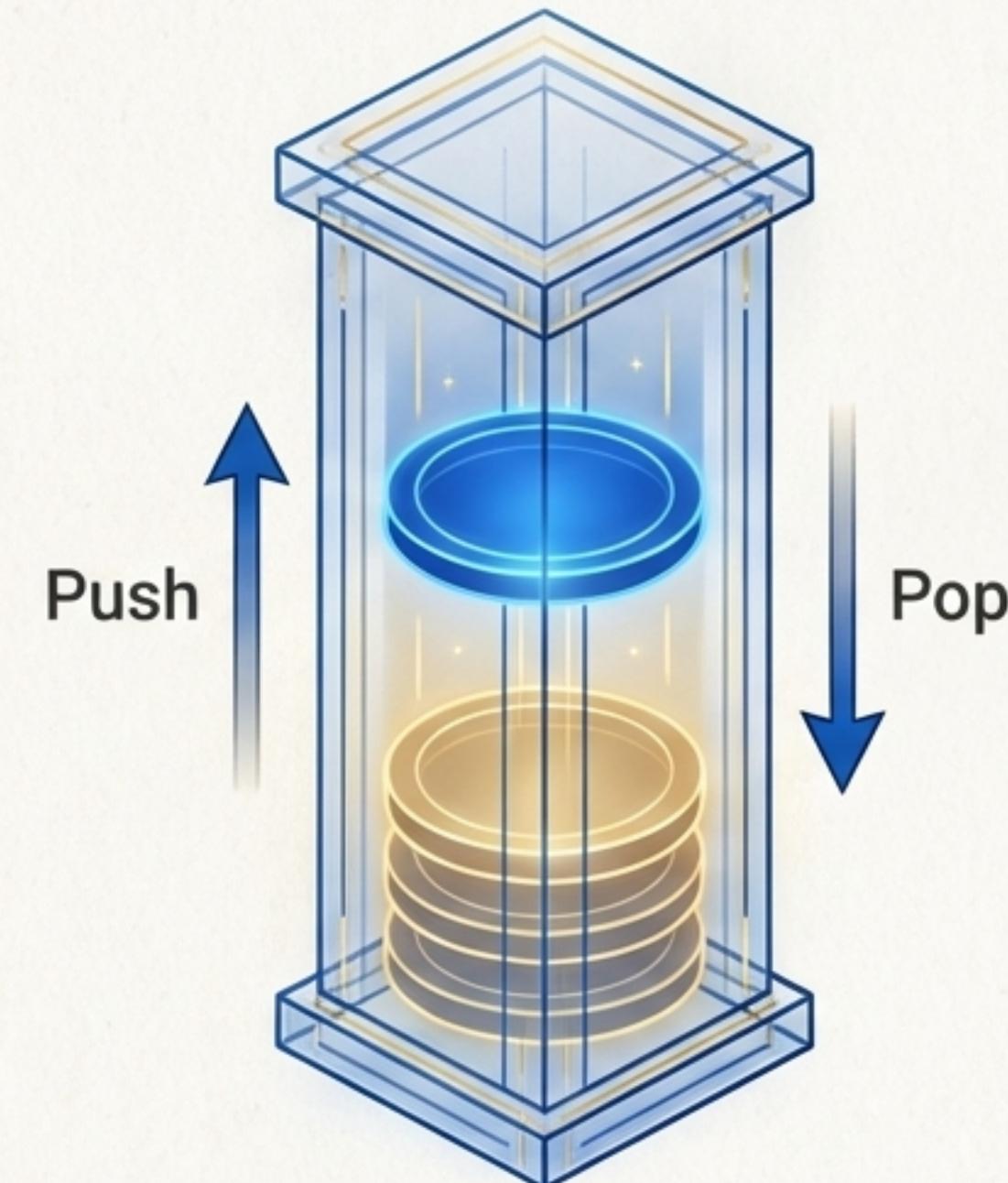
امضای متده‌ای مانند "کارت شناسایی" آن است. ماشین مجازی جاوا (JVM) از طریق آن متده‌ای را پیدا و تایید می‌کند.

۱. سطح دسترسی (Access Modifier): (static, final)
۲. کلمات کلیدی دیگر (Return Type): تعلق به کلاس و عدم امکان بازنویسی.
۳. نوع خروجی (Method Name): متده‌ای که برای فراخوانی استفاده می‌شود.
۴. نام متده (Method Name): نامی که برای فراخوانی استفاده می‌شود.
۵. لیست پارامترها (Parameter List): داده‌های ورودی مورد نیاز متده.

```
public static final double calculateArea(double radius) {  
    return Math.PI * radius * radius;  
}
```

۶. بدنه متده (Method Body): منطق و دستورات اجرایی متده.

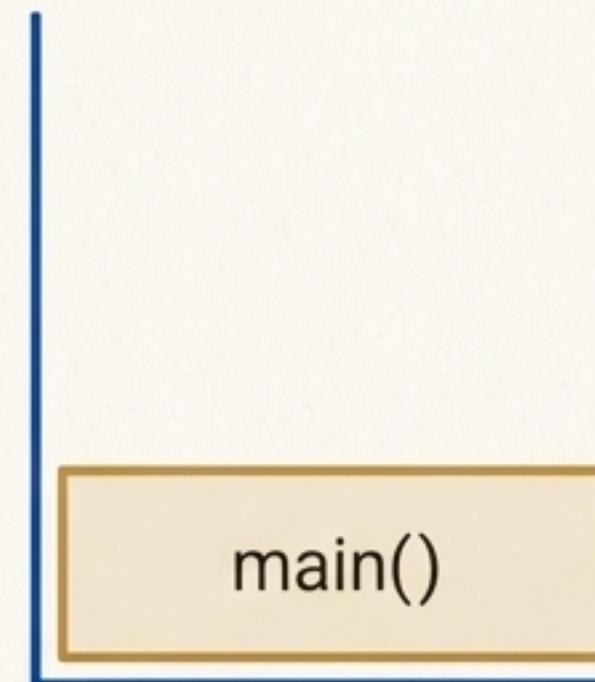
پشت پرده چه می‌گذرد؟ سفر به حافظه Stack



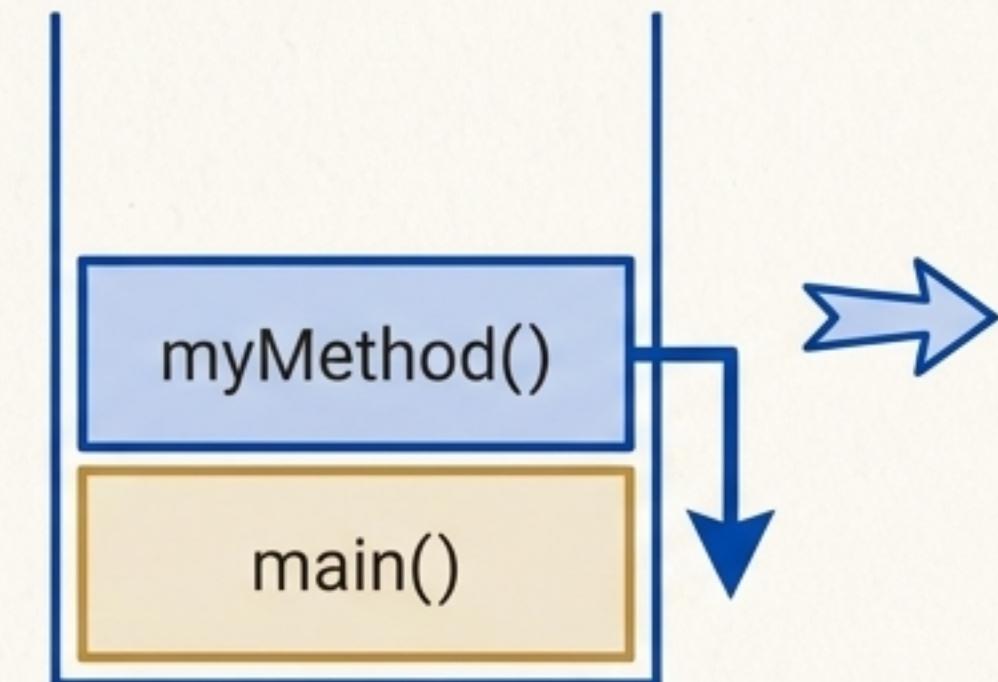
حافظه Stack مانند یک دسته بشقاب است. وقتی وقتی متدهای فراخوانی می‌شود، یک "فریم" (یک بشقاب جدید) با متغیرهای محلی اش روی پشته قرار می‌گیرد. وقتی کارش تمام می‌شود، بشقاب برداشته می‌شود. این یک فرآیند LIFO (Last-In, First-Out) است.

ردپای اجرا: یک فریم در Stack

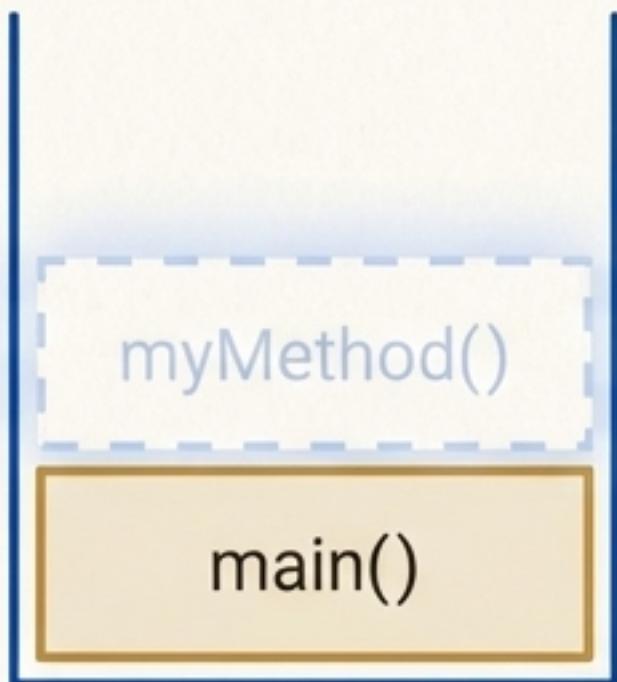
قبل از فراخوانی



هنگام اجرای `myMethod`



بعد از اتمام `myMethod`



هر فریم یک محیط ایزوله است. به همین دلیل است که یک متغیر در `myMethod` با متغیری با نام مشابه در `main` تداخل پیدا نمی‌کند. با پایان `myMethod`, فریم آن از `Stack` حذف (pop) می‌شود و کنترل به `main` بازمی‌گردد.

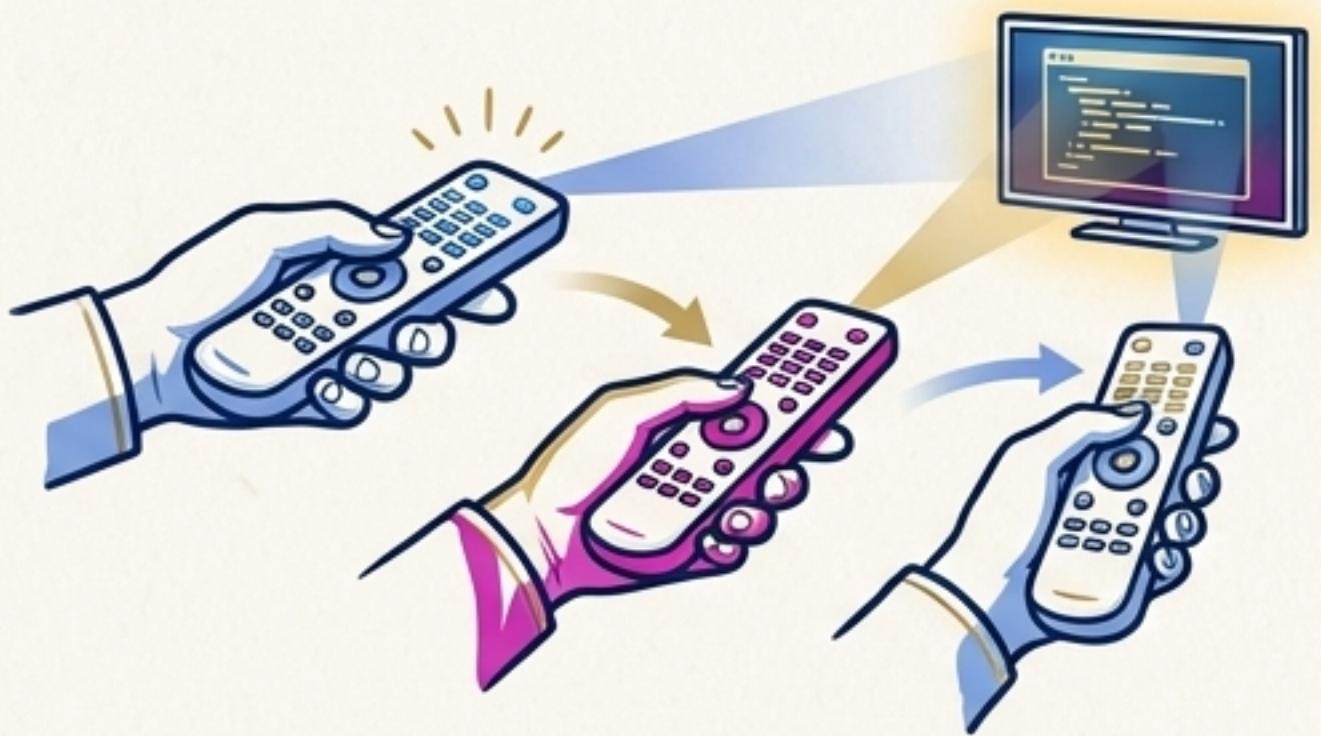
راز پاس دادن آرگومان‌ها: همیشه "Pass-by-Value"

کپی مقدار



برای **انواع اولیه**، شما یک کپی از داده را پاس می‌دهید.

کپی آدرس



برای **انواع ارجاعی**، شما یک کپی از آدرس حافظه را پاس می‌دهید.

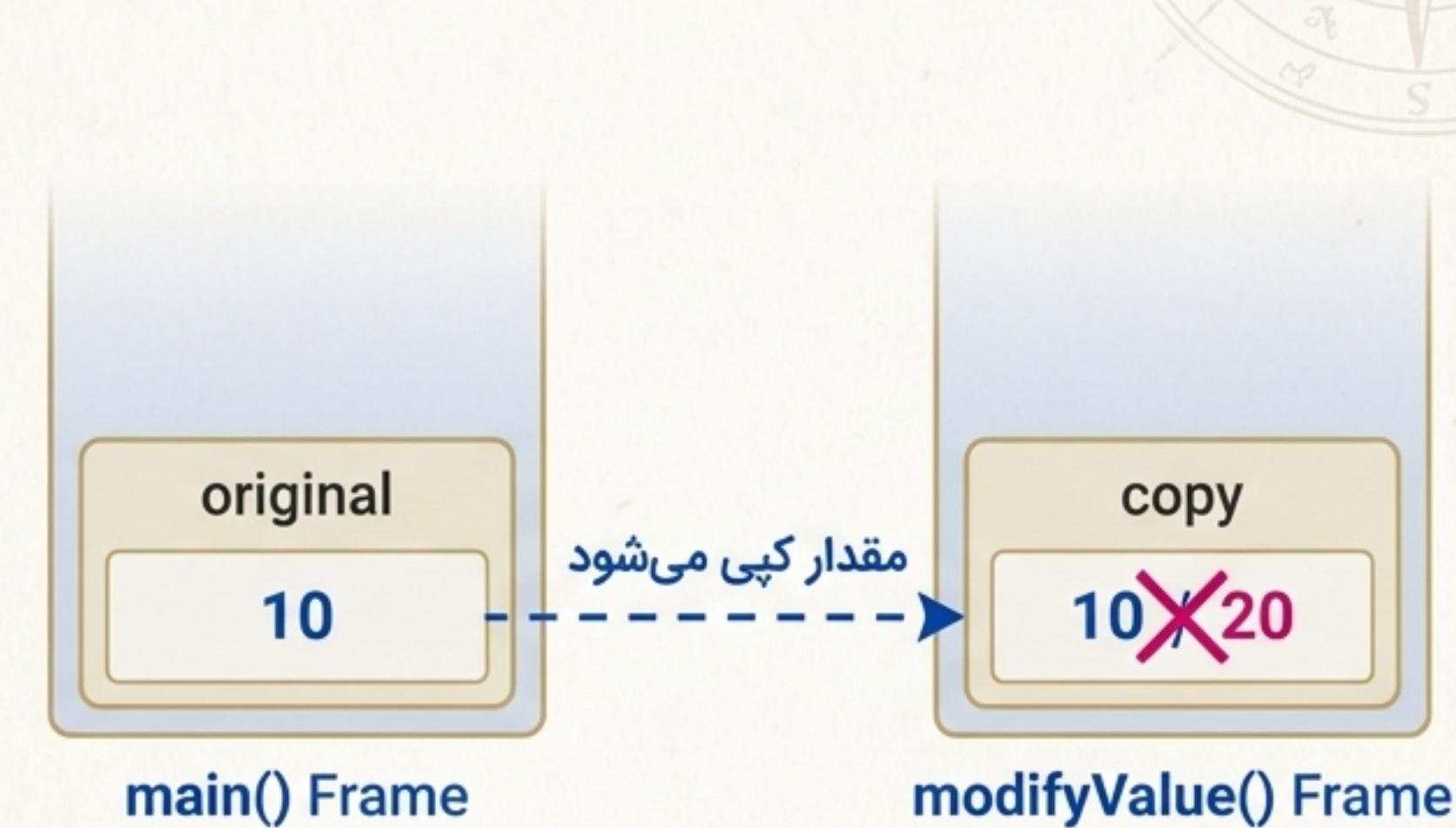
جاوا همیشه از مکانیزم "پاس با مقدار" (Pass-by-Value) استفاده می‌کند. سردرگمی از اینجا ناشی می‌شود که "مقدار" پاس داده شده برای انواع اولیه و انواع ارجاعی متفاوت است.

انواع اولیه (Primitives): یک کپی بی خطر

```
public void main() {  
    int original = 10;  
    modifyValue(original);  
    // original is STILL 10 here  
}
```

```
public void modifyValue(int copy) {  
    copy = 20; // Only affects the copy  
}
```

یک کپی از مقدار واقعی آرگومان به پارامتر متده می شود.
هر تغییری روی پارامتر در داخل متده، هیچ تأثیری روی متغير اصلی در خارج از متده نخواهد داشت.



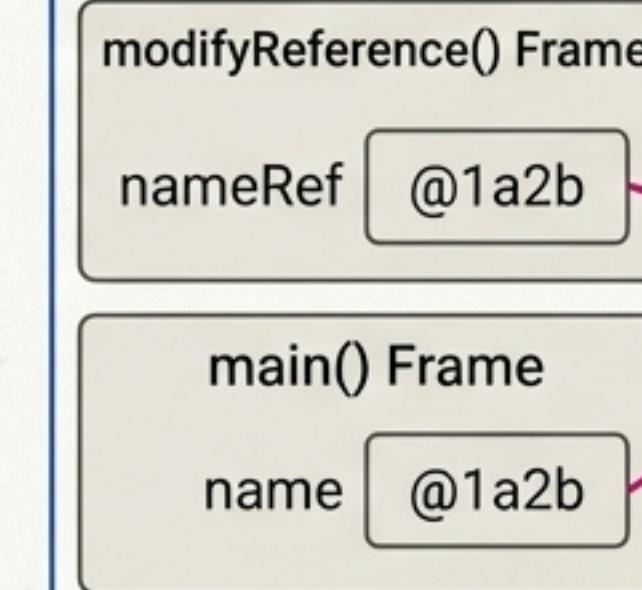
تغییرات در داخل، در بیرون بی تأثیر است.

انواع ارجاعی (References): یک کنترل از راه دور مشترک

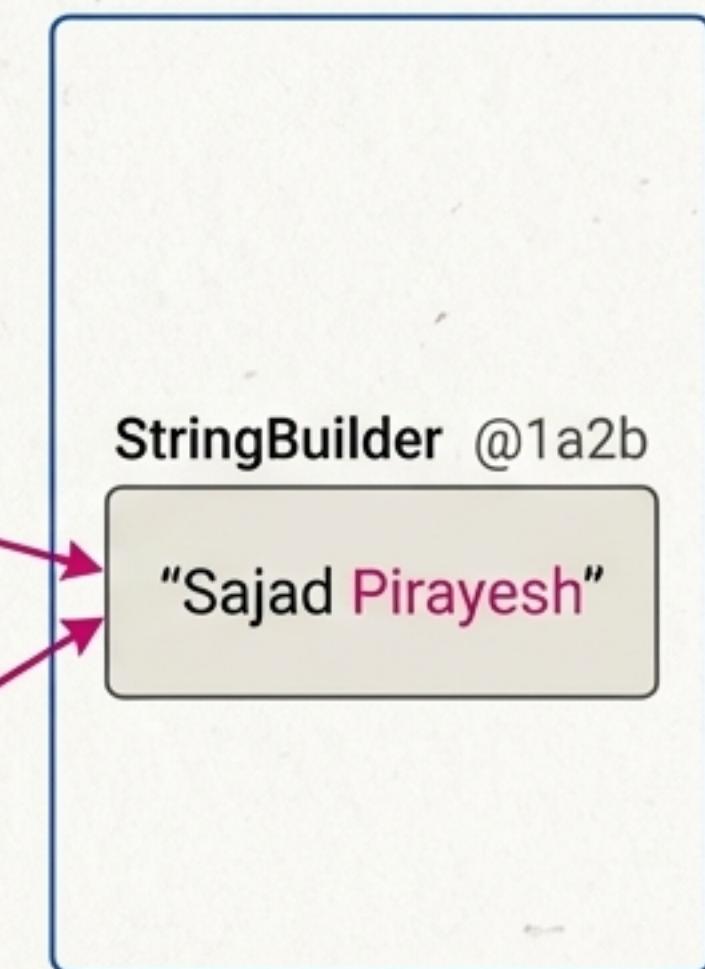
```
public void main() {  
    StringBuilder name = new StringBuilder("Sajad");  
    modifyReference(name);  
    // name is now "Sajad Pirayesh"  
}  
  
public void modifyReference(StringBuilder nameRef) {  
    nameRef.append(" Pirayesh");  
}
```

در اینجا نیز یک **کپی از مقدار** پاس داده می‌شود، اما این مقدار، **آدرس حافظه** شیء است. بنابراین، پارامتر داخل متده و متغیر خارج از متده، هر دو به یک شیء یکسان در حافظه Heap اشاره می‌کنند. شما می‌توانید از این "کنترل از راه دور" کپی شده برای تغییر وضعیت داخلی شیء اصلی استفاده کنید.

Stack



Heap



→ تغییر در وضعیت داخلی شیء

آزمون اول: خروجی را پیش‌بینی کنید

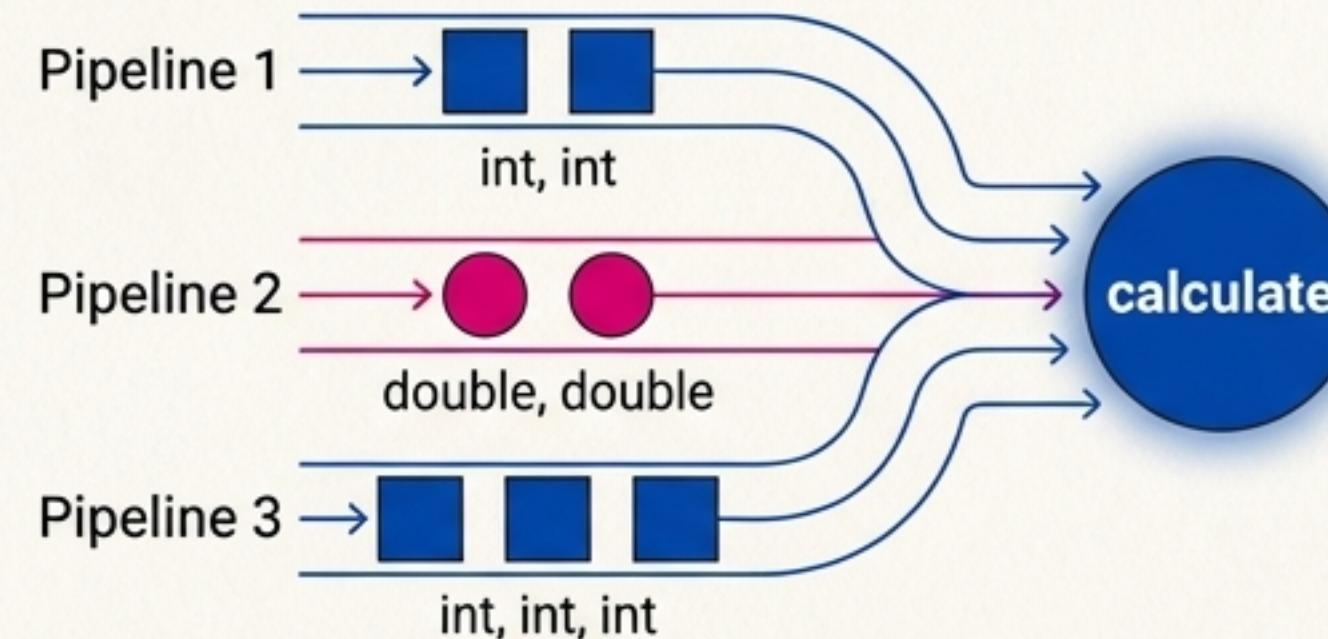
با توجه به مفاهیم Pass-by-Value، کد زیر چه چیزی در خروجی چاپ می‌کند؟

```
public class Prediction {  
    public static void main(String[] args) {  
        StringBuilder name = new StringBuilder("Ali");  
        int score = 100;  
        modify(name, score);  
        System.out.println("Name: " + name + ", Score: " + score);  
    }  
  
    public static void modify(StringBuilder nameRef, int scoreVal) {  
        nameRef.append(" Reza");  
        scoreVal = 0;  
    }  
}
```

به تفاوت بین پاس دادن یک `StringBuilder` و یک `int` فکر کنید!

خروجی؟

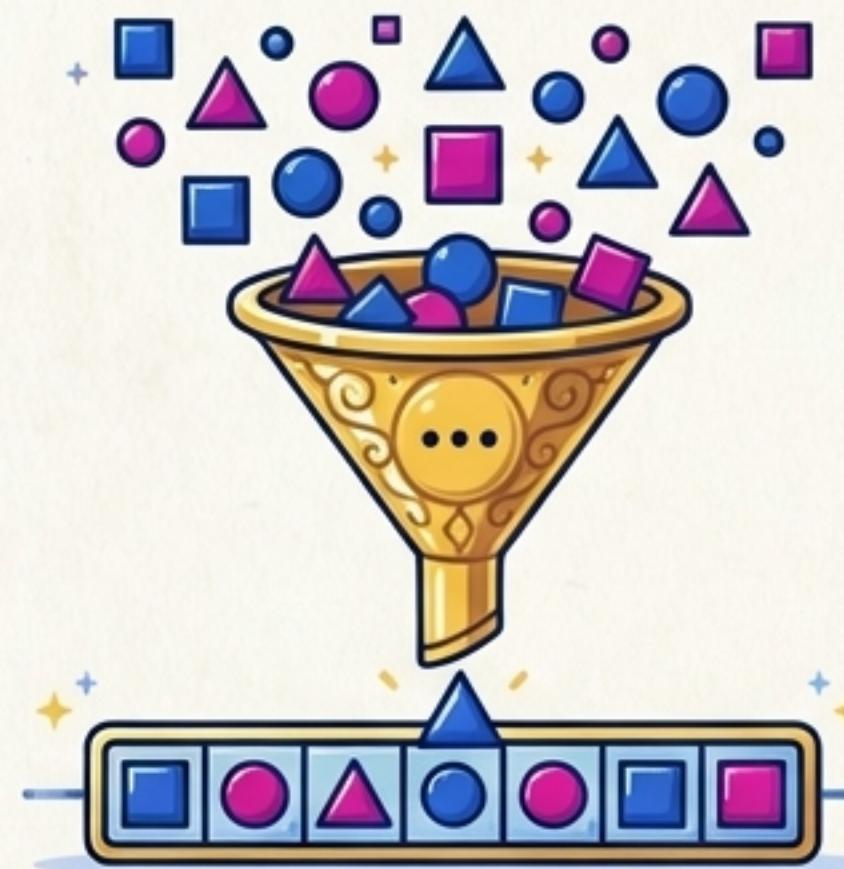
سربارگذاری (Overloading): یک نام، چندین قدرت



سربارگذاری به ما اجازه می‌دهد تا چندین متده باشیم، به شرطی که **لیست پارامترهای آنها متفاوت باشد** (از نظر تعداد یا نوع). این یک نوع "چندریختی زمان کامپایل" (Compile-Time Polymorphism) است، تامپاین، زیرا کامپایلر دقیقاً می‌داند کدام نسخه را بر اساس آرگومان‌های شما فراخوانی کند.

معیار	Overriding (بازنویسی)	(سربارگذاری) Overloading
هدف	تغییر رفتار یک متده از کلاس والد	ارائه چند نسخه از یک متده با ورودی‌های متفاوت
محل تعریف	در کلاس فرزند	در یک کلاس
امضای متده	باید دقیقاً یکسان باشد	باید متفاوت باشد (نام یکسان)
زمان تشخیص	Run-Time (Dynamic Polymorphism)	Compile-Time (Static Polymorphism)
ارتباط	کاملاً وابسته به وراثت است	هیچ ارتباطی با وراثت ندارد

لازار (...):Varargs برای همه اندازه‌ها



(Varargs) راهی ساده برای نوشتن متدى است که تعداد نامشخصی از آرگومان‌ها را می‌پذیرد. جاوا به صورت خودکار این آرگومان‌ها را در یک آرایه قرار می‌دهد.

: (Rules)

- در هر متى، فقط یك پارامتر Varargs می‌تواند وجود داشته باشد.
- پارامتر Varargs باید آخرین پارامتر در لیست باشد.

مثال جمع‌کننده (Aggregator)

```
public static int sumAll(int... numbers) {  
    int sum = 0;  
    for (int num : numbers) {  
        sum += num;  
    }  
    return sum;  
}
```

// How to call it:

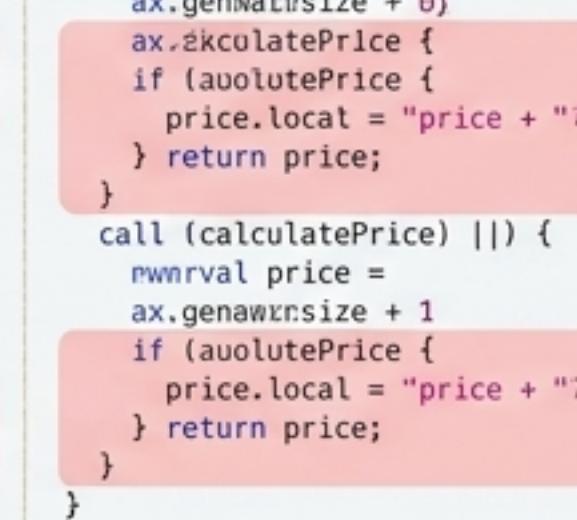
```
sumAll(1, 2); // returns 3  
sumAll(10, 20, 30, 40); // returns 100  
sumAll(); // returns 0
```

مهمترین قانون کیمیاگری کد: خودت را تکرار نکن! (DRY)

(WET - Write Everything Twice) تکرار کد



```
public func calculatePrice() {  
    ranoval price =  
        ax.genaatesize +  
        if (auilotePrice {  
            price.local = "price + ");  
        } return price;  
        f return price?;  
    }  
  
privanc calculatePrice() {  
    removal price =  
        ax.genesrrsize +  
        if (ouilotePrice {  
            price.local = "price + ");  
        } return price;  
    }  
}
```



```
public func calculatePrice() {  
    rematvat price =  
        ax.genmatwsize + 0)  
        ax.ekculatePrlce {  
            if (auolutePrice {  
                price.local = "price + ?";  
            } return price;  
        }  
  
    call (calculatePrice) ||| {  
        mwnrval price =  
            ax.genawrnsize + 1  
            if (auolutePrice {  
                price.local = "price + ");  
            } return price;  
        }  
}
```

(DRY - Don't Repeat Yourself) کد تمیز

```
public fun calculatePrice() {  
    var prices = 100;  
    let peinePices = "Price - KABEET";  
    return price;  
}
```

```
calculatePrice(), calculatePrice();  
calculatePrice(), calculatePrice();
```



اصل "سنگ بنای کدنویسی تمیز و حرفه‌ای است. متدها ابزار اصلی شما برای دستیابی دستیابی به DRY هستند. استفاده مجدد از کد خطاهای را کاهش می‌دهد، به روزرسانی‌ها را ساده می‌کند و منطق برنامه را قابل فهم‌تر می‌سازد.

چالش طراحی: کلاس ماشین حساب

یک کلاس `Calculator` طراحی کنید که شامل متدهای زیر باشد:

- `add(double a, double b)`
- `subtract(double a, double b)`
- `multiply(double a, double b)`
- `divide(double a, double b)`

برای متدهای `divide`، اگر مخرج صفر بود، مقدار `0.0` را برگردانید.

به بهترین سطح دسترسی و نوع خروجی برای هر متدهای فکر کنید.



جعبه ابزار شما: مفاهیم کلیدی متدها



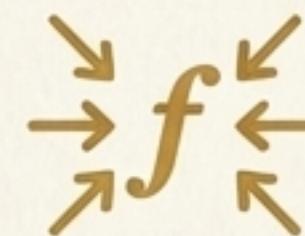
متدها: بلوک های کد قابل استفاده مجدد که رفتار شیء را تعریف می کنند.



امضا: هویت متدها (نام + لیست پارامترها).



.References، کپی آدرس برای Primitives: کپی مقدار برای Pass-by-Value

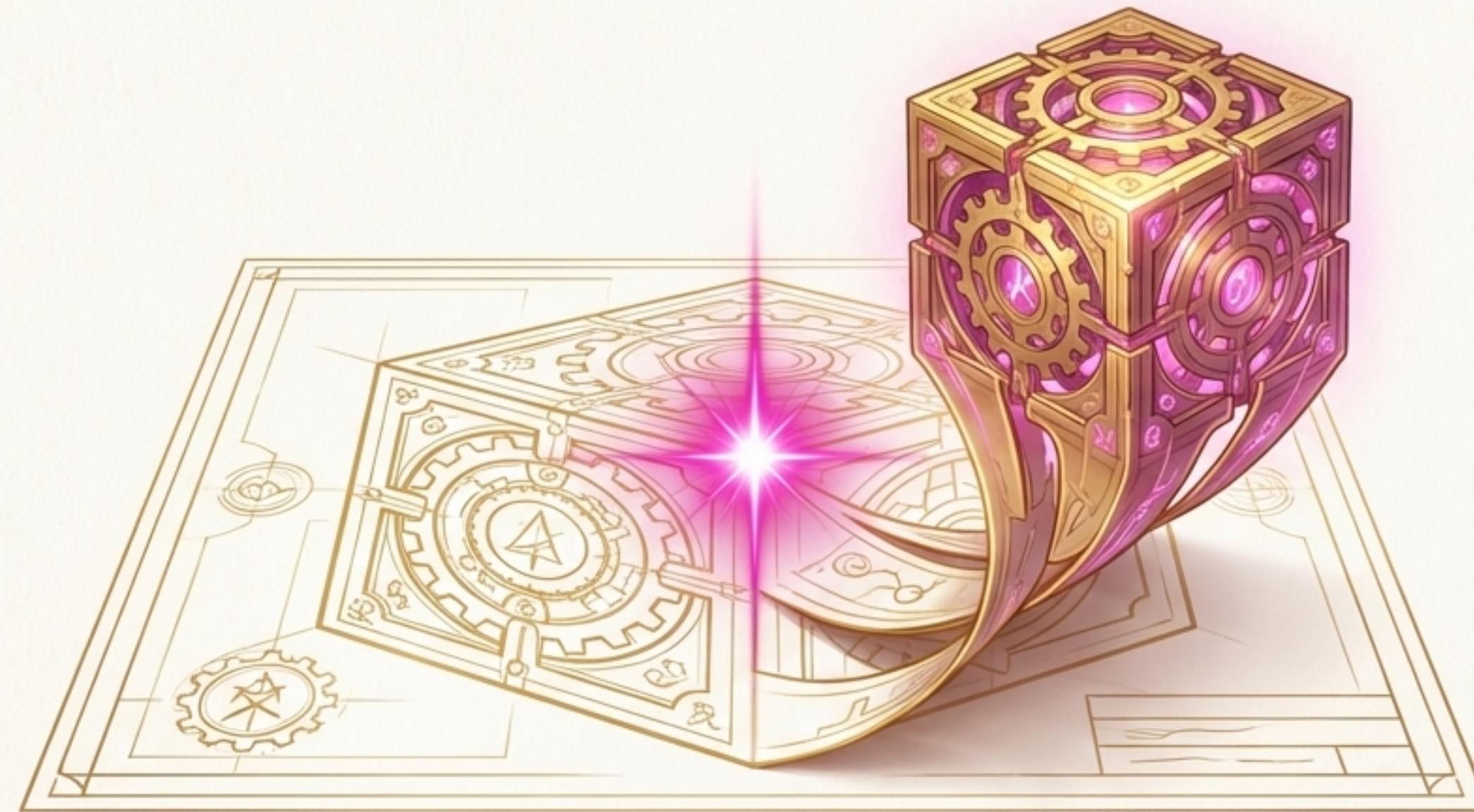


: نام یکسان، پارامترهای متفاوت، در یک کلاس.



: پذیرش تعداد نامشخصی از آرگومانها به عنوان یک آرایه.

در بخش بعدی...



سازنده‌ها (Constructors): لحظه تولد یک شیء