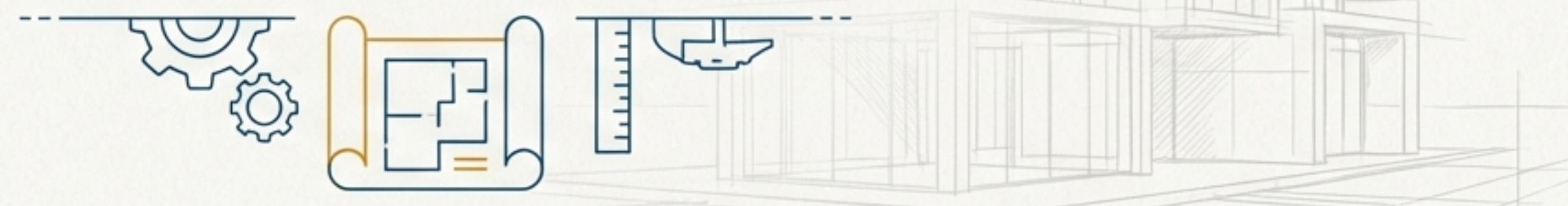




بخش ۳: final و static

اشتراک‌گذاری و تغییرناپذیری

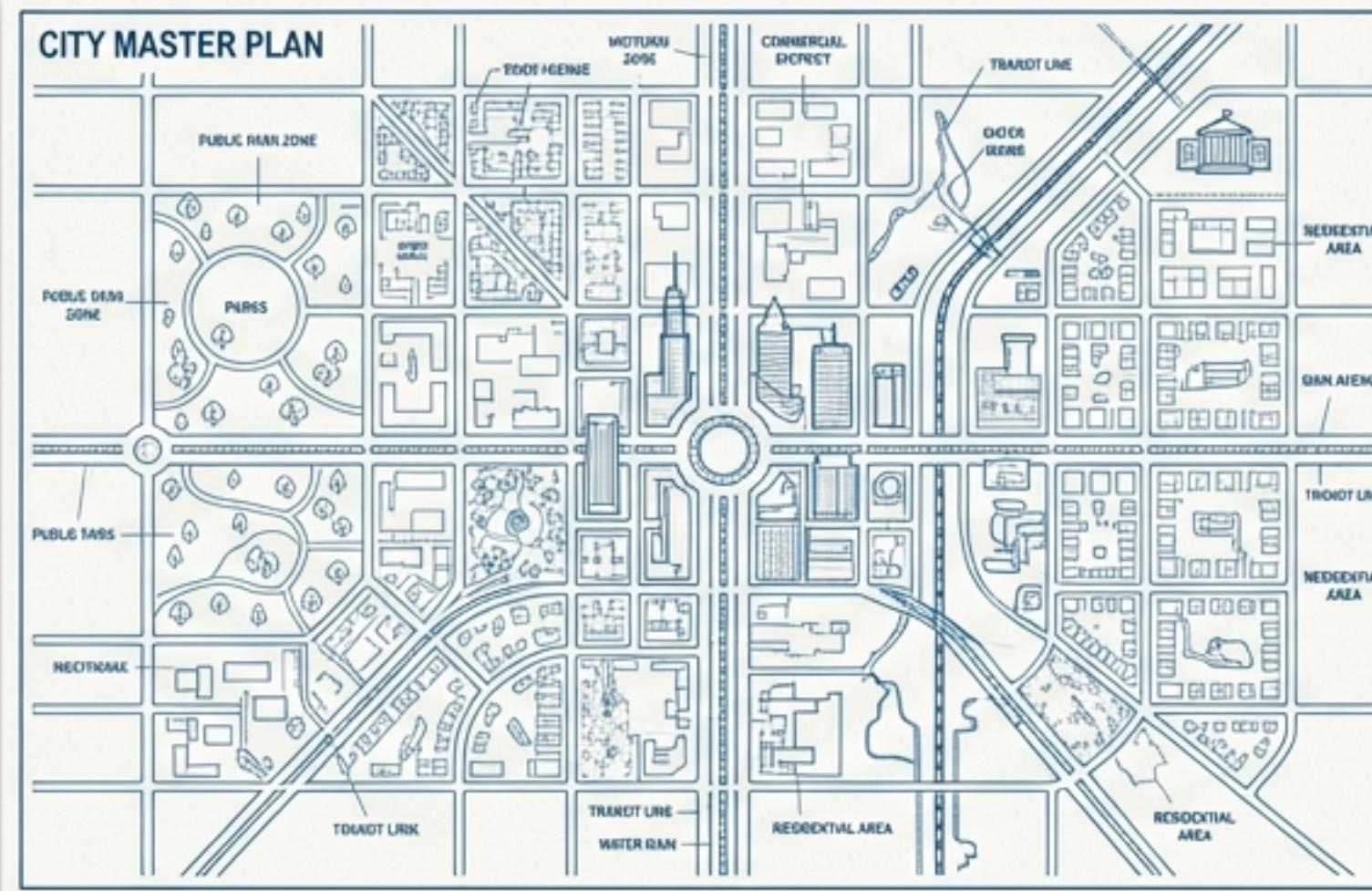
ابزارهای معمار: طراحی کلاس‌های قدرتمند و پایدار



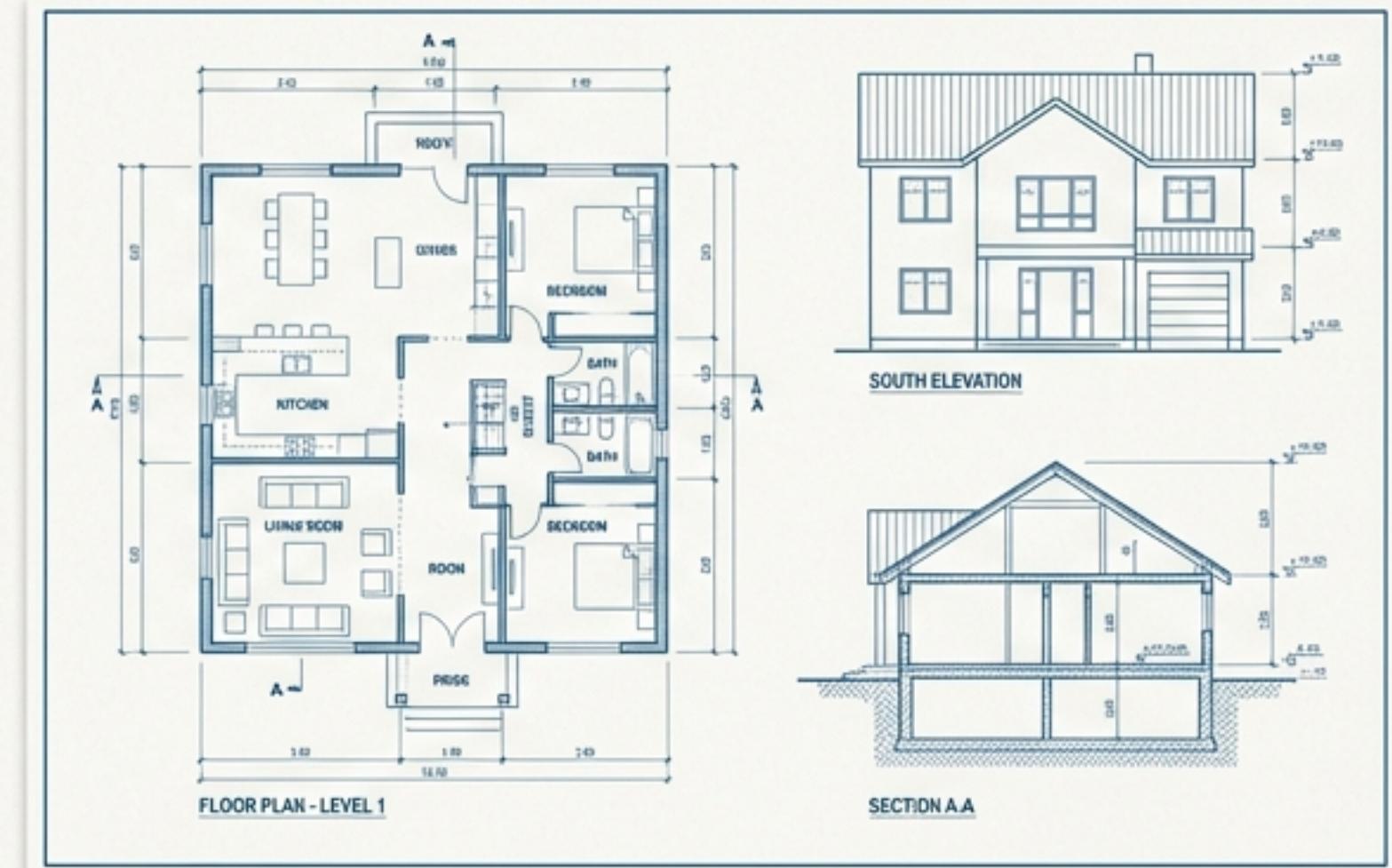
تهیه شده توسط: سید سجاد پیراهش

دو دنیا در یک کلاس: نقشه شخصی در مقابل نقشه عمومی

نقشه عمومی (Static)



کپی شخصی (Instance)



اعضای **static** به خود کلاس تعلق دارند، نه به یک شیء خاص. آن‌ها یک "نقشه عمومی" هستند که تمام اشیاء آن را به اشتراک می‌گذارند.

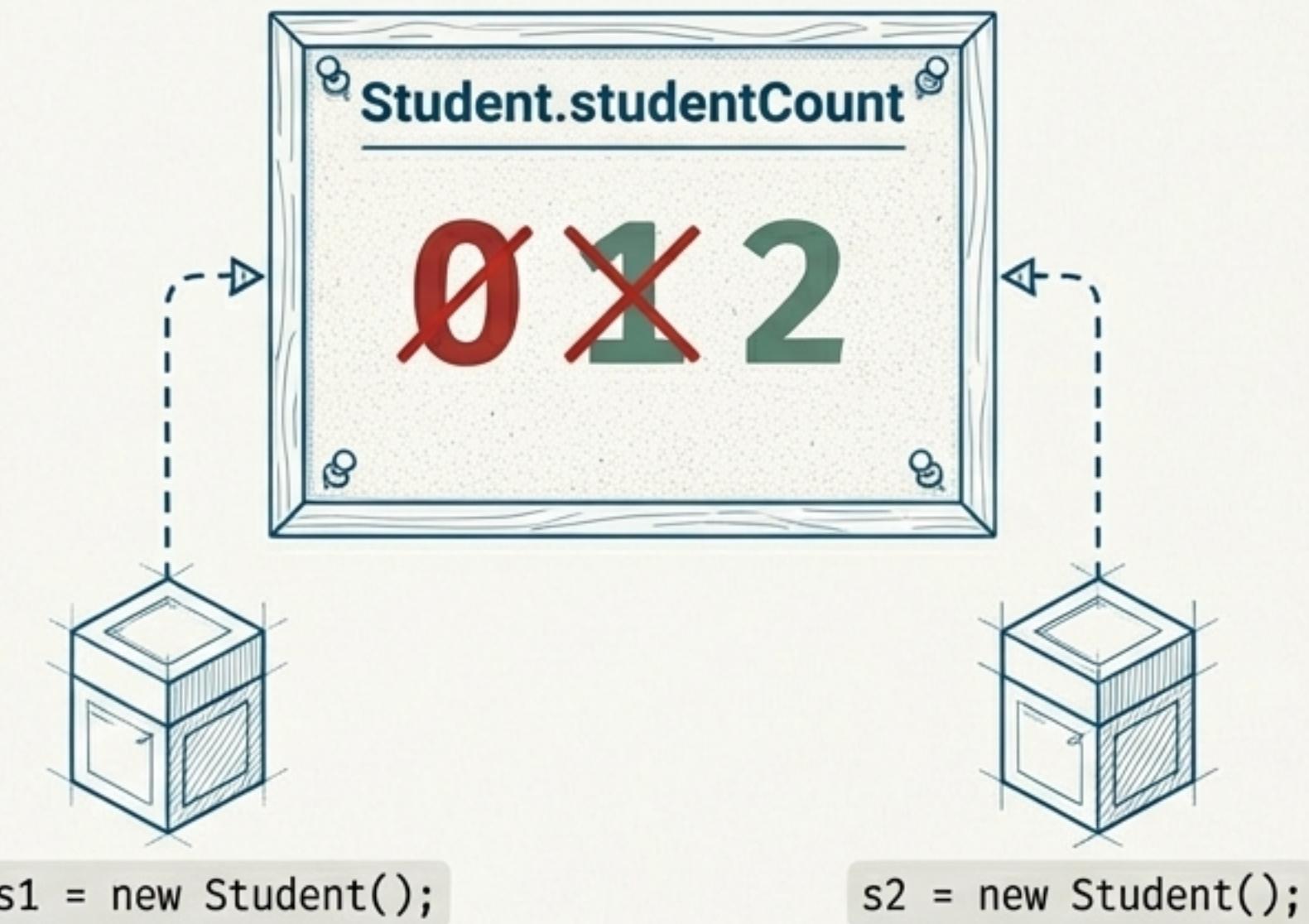
هر شیء که با **new** ساخته می‌شود (مثلًا `(new Student()`)، یک کپی کاملاً **شخصی** و **مستقل** از فیلدهای نمونه (`(name , id`) از متغیر `name` در دریافت می‌کند. اگر ۱۰۰ شیء بسازیم، ۱۰۰ مجزا در حافظه خواهیم داشت.

فیلدهای static: تابلوی اعلانات عمومی

”یک کپی برای همه.“ یک متغیر واحد و مشترک برای تمام نمونه‌های یک کلاس، فارغ از اینکه صفر یا یک میلیون میلیون شیء از آن ساخته شود.

شمارنده اشیاء

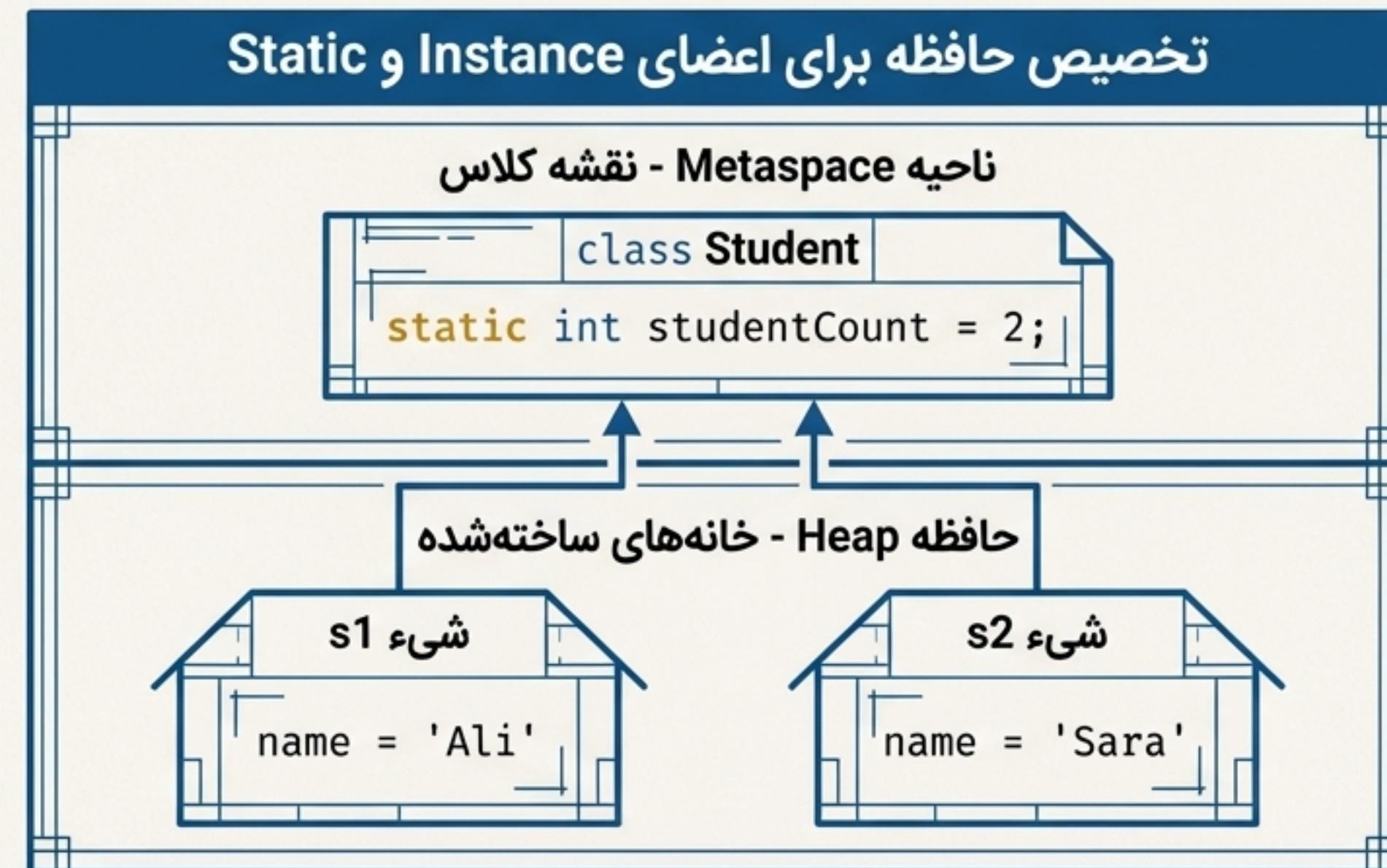
```
1 public class Student {  
2     public String name;  
3     public static int studentCount = 0; // Sha  
4  
5     public Student(String name) {  
6         this.name = name;  
7         studentCount++; // Accesses the single  
8     }  
9 }  
10  
11 // --- In main method ---  
12 Student s1 = new Student("Ali");  
13 Student s2 = new Student("Sara");
```



هر دو شیء، یک متغیر `studentCount` را می‌بینند و تغییر می‌دهند.

نقشه شهر حافظه: Metaspace در مقابل Heap

فیلدهای نمونه (شخصی) در حافظه `Heap` و درون هر شیء زندگی می‌کنند. اما فیلدهای static (عمومی) در یک ناحیه ویژه به نام `Metaspace` (در نسخه‌های جدید JVM) ذخیره می‌شوند که فراداده (Metadata) کلاس در آن قرار دارد.



هر دو شیء s1 و s2 در Heap کپی شخصی خود از name را دارند، اما هر دو به یک نسخه مشترک از studentCount اشاره می‌کنند.

متدهای static: جعبه ابزار همگانی

توابعی که به کلاس تعلق دارند و برای فراخوانی آنها نیازی به ساختن شیء نیست. اینها ابزارهای عمومی هستند که به وضعیت (state) یک شیء خاص وابستگی ندارند.

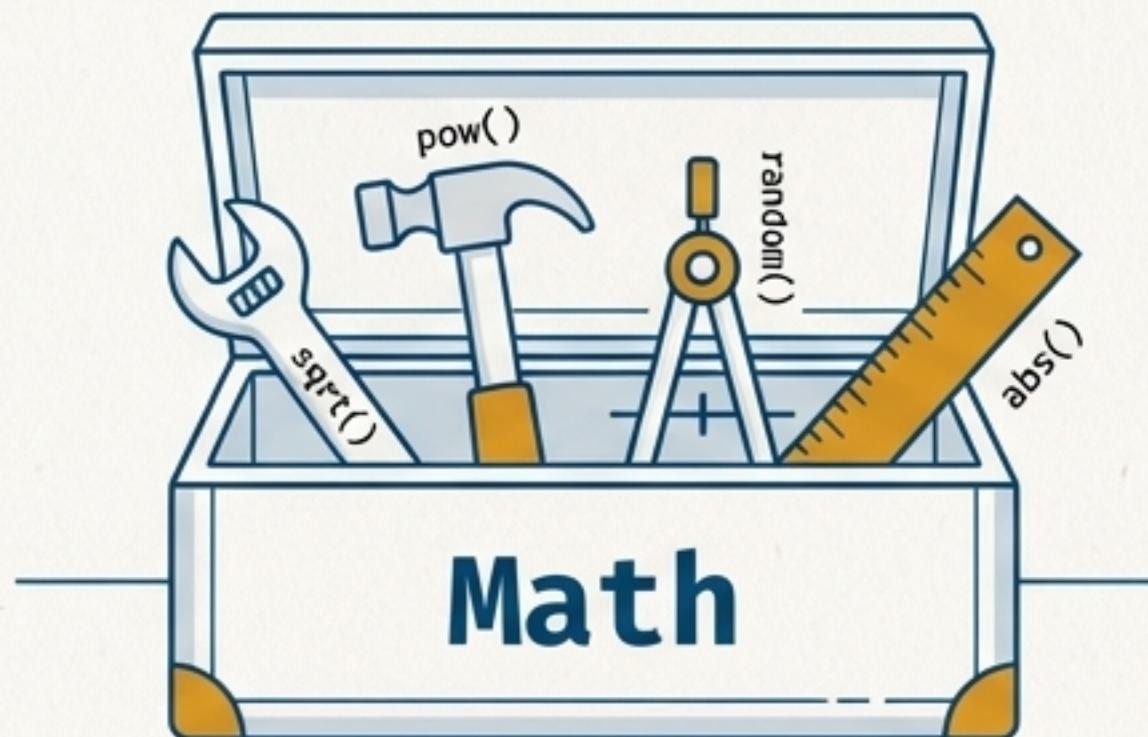
قانون طلایی:

یک متدهای static نمی‌تواند به اعضای غیر استاتیک (مانند فیلد name یا کلمه کلیدی this) دسترسی داشته باشد.



چرا؟ چون متدهای static به هیچ شیء خاصی گره نخورده است. وقتی (Student.getAverageGrade()) JVM را صدا می‌زنید، نمی‌داند منظور شما نمرات کدام دانشجو است. هیچ "this" وجود ندارد.

مثال کلاسیک:



```
Math.sqrt(25);
```

شما هرگز (new Math()) نمی‌سازید تا جذر یک عدد را حساب کنید. این یک ابزار جهانی است.

بلوک‌های static: مراسم افتتاحیه کلاس

یک بلوک کد که با کلمه `static` مشخص شده و فقط یک بار، در اولین لحظه‌ای که کلاس توسط JVM بارگذاری (Load) می‌شود، اجرا می‌گردد.

کاربرد:

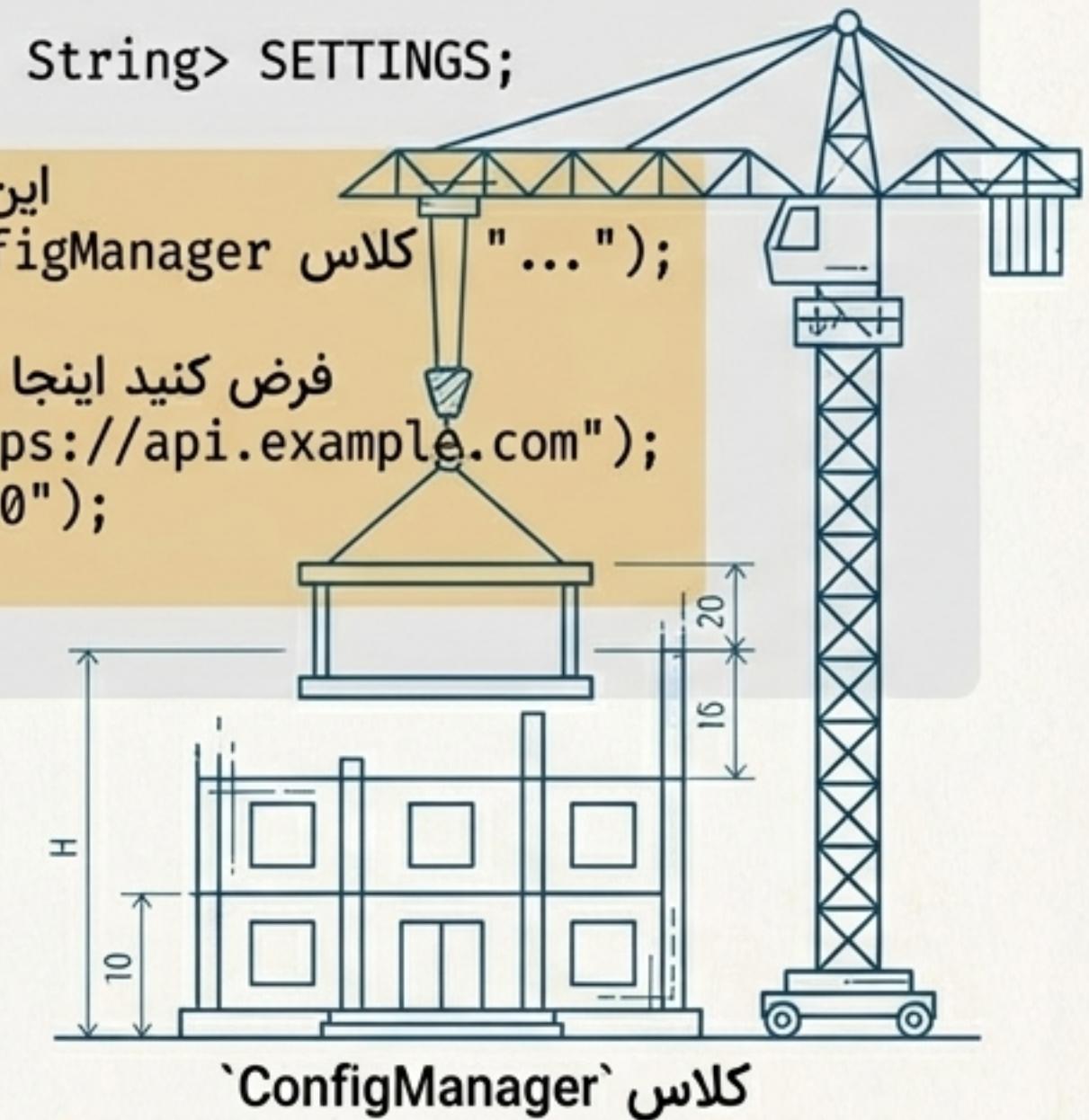
عالی برای مقداردهی اولیه پیچیده و یکباره برای فیلد های استاتیک استاتیک. مثلاً:

- خواندن تنظیمات از یک فایل کانفیگ.

- برقراری یک اتصال اولیه به پایگاه داده.

- آماده سازی یک `Map` استاتیک با مقادیر پیش فرض.

```
class ConfigManager {  
    public static final Map<String, String> SETTINGS;  
  
    static { // این بلوک فقط یک اجرا می‌شود  
        System.out.println("است ConfigManager ...");  
        SETTINGS = new HashMap<>();  
        فرض کنید اینجا تنظیمات از یک فایل خوانده می‌شود //  
        SETTINGS.put("API_URL", "https://api.example.com");  
        SETTINGS.put("TIMEOUT", "5000");  
    }  
}
```



قلعه `final` : قوانین تغییرناپذیری

اکنون که با ابزارهای اشتراک‌گذاری آشنا شدیم، به سراغ ابزارهای استحکام و پایداری می‌رویم. `final` یک قول به کامپایلر و دیگر توسعه‌دهندگان است: "این مقدار، پس از تعیین، هرگز تغییر نخواهد کرد."

تمثیل: استفاده از `final` مانند 'نوشتن روی سنگ' است؛ یک تعهد دائمی.

متغیرهای final: نقطه بی‌بازگشت

قانون اصلی: "تک تخصیص" (Single Assignment). یک متغیر `final` فقط و فقط یک بار می‌تواند مقداردهی شود (یا در زمان تعریف، یا در سازنده).

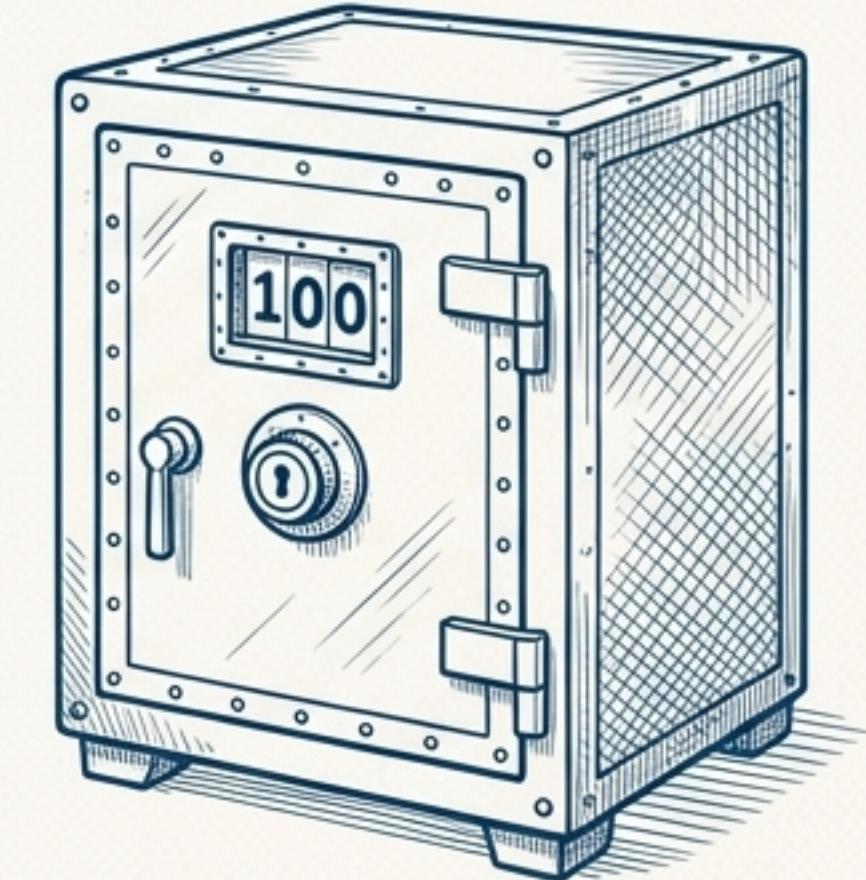
```
final List<String> USERS = new ArrayList<>();  
USERS.add("Sajjad"); // کاملاً مجاز! ✓
```

این ارجاع (آدرس حافظه) است که قفل شده، نه محتوای لیست.



```
final int MAX_SCORE = 100;  
MAX_SCORE = 101; // خطای کامپایل! ✗
```

خود مقدار `100` قفل شده است.



تله ارجاع `final` : آدرس قفل است، نه محتوا!

این مهمترین نکته در مورد `final` است. وقتی یک متغیر ارجاعی (مانند یک شیء یا لیست) را `final` می‌کنید، شما فقط متغیر را از اشاره کردن به یک شیء ***جديد*** منع می‌کنید. اما شیء اصلی که به آن اشاره می‌شود، کاملاً قابل تغییر باقی می‌ماند.

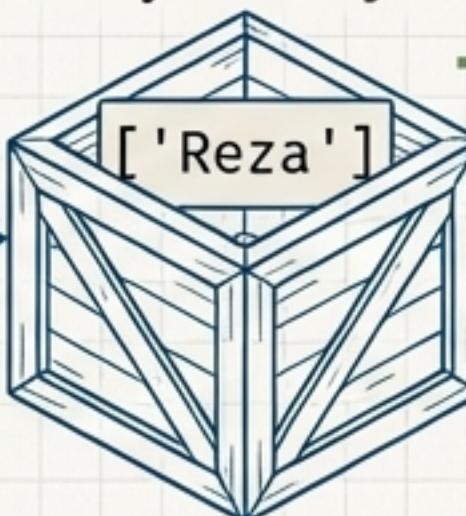
آناتومی یک ارجاع final

حافظه Stack



حافظه Heap

ArrayList Object

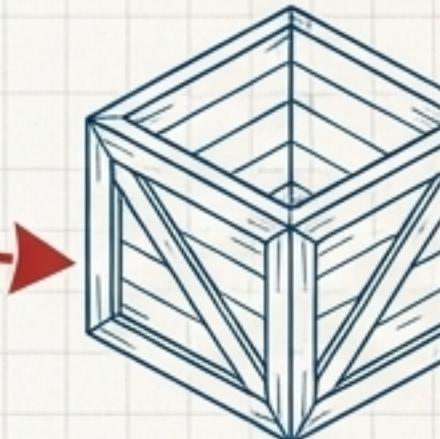


USERS.add('Nima')

['Reza', 'Nima']

تغییر محتوای شیء

USERS = new ArrayList<>()
تلاش برای تغییر ارجاع



می‌توانیم محتوای لیست را تغییر دهیم , اما نمی‌توانیم USERS را به یک لیست جدید منتب کنیم .

ترکیب طلایی: `public static final` برای ثابت‌های جهانی

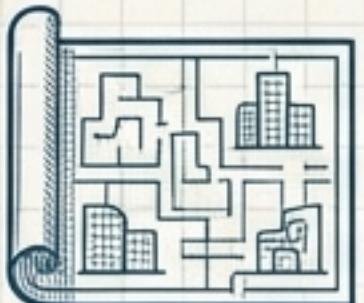
این ترکیب، استاندارد طلایی معماری نرم‌افزار برای تعریف مقادیر ثابتی است که در کل پروژه باید یکسان و غیرقابل تغییر باقی بمانند.

تجزیه ترکیب:



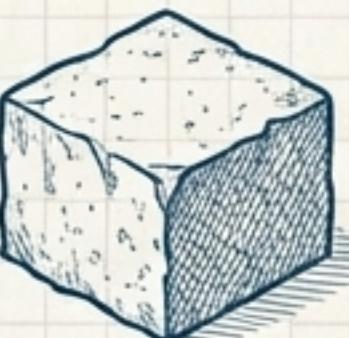
public

قابل دسترس از هر کجای پروژه.
(کلید عمومی ساختمان)



static

متعلق به کلاس، بدون
نیاز به ساخت شیء.
(بخشی از نقشه اصلی)



final

مقدار آن پس از تعیین،
هرگز تغییر نمی‌کند.
(ساخته شده از بتن مسلح)

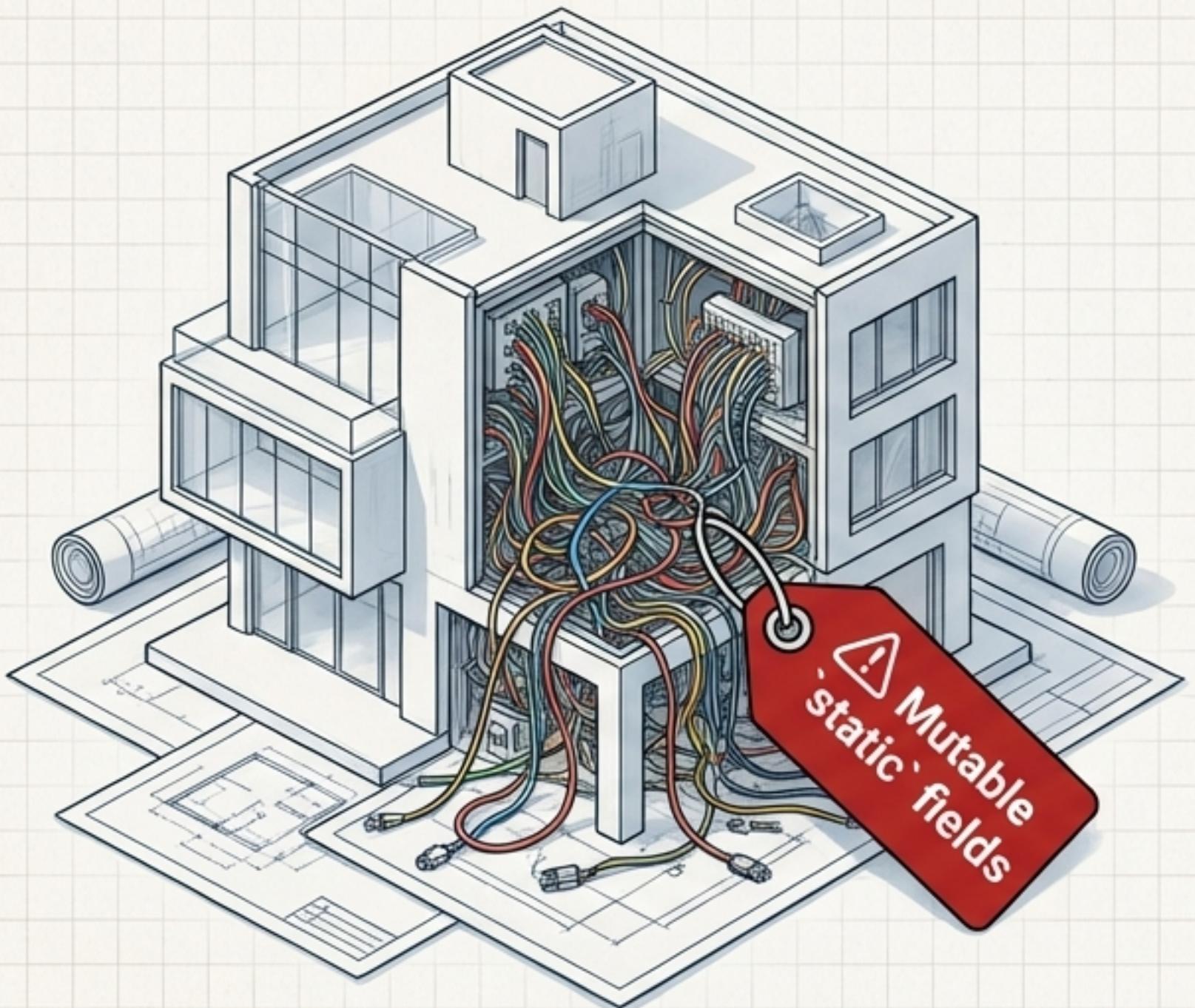
public static final

ثابت‌های بازی

```
public class GameConstants {  
    public static final int  
        MAX_PLAYERS = 4;  
    public static final String  
        GAME_VERSION = "v2.1.0";  
    public static final float  
        DEFAULT_SPEED = 1.5f;  
}
```

**نکته مهم: به قرارداد نام‌گذاری `UPPER_SNAKE_CASE` برای این ثابت‌ها توجه کنید.

هشدار معمار: بُوی بِدَّ گَد از وضعیت سراسری (Global State)



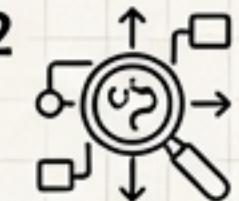
استفاده بیش از حد از فیلدهای `static` که قابل تغییر (mutable) باشند، یک ضدالگو (anti-pattern) محسوب می‌شود و به آن "وضعیت سراسری" یا 'Global State' می‌گویند.

چرا این یک مشکل است؟

1. تست‌پذیری را نابود می‌کند: تست‌ها بر روی یکدیگر تأثیر می‌گذارند.
اگر یک تست، مقدار یک متغیر استاتیک را تغییر دهد، تست بعدی با یک وضعیت غیرمنتظره شروع به کار می‌کند.



2. استدلال را دشوار می‌کند: وقتی هر بخشی از برنامه می‌تواند یک متغیر را تغییر دهد، پیدا کردن منشأ باغ‌ها بسیار پیچیده می‌شود.



راهنمای معمار:

از `static` برای ثابت‌ها (`final`) و توابع ابزاری بی‌حالت (`stateless`) استفاده کنید. در مورد وضعیت استاتیک قابل تغییر، بسیار محتاط باشید.

آزمون: کدام کد کامپایل نمی‌شود؟

به قطعه‌کدهای زیر نگاه کنید و پیش‌بینی کنید کدامیک باعث خطای کامپایل می‌شود.

```
// static method accesses instance field
class Test {
    String name;
    public static void printName() {
        System.out.println(this.name); // ?
    }
}
```

(خطای کامپایل)

```
// re-assigning a final primitive
final int x = 10;
x = 20; // ?
```

(خطای کامپایل)

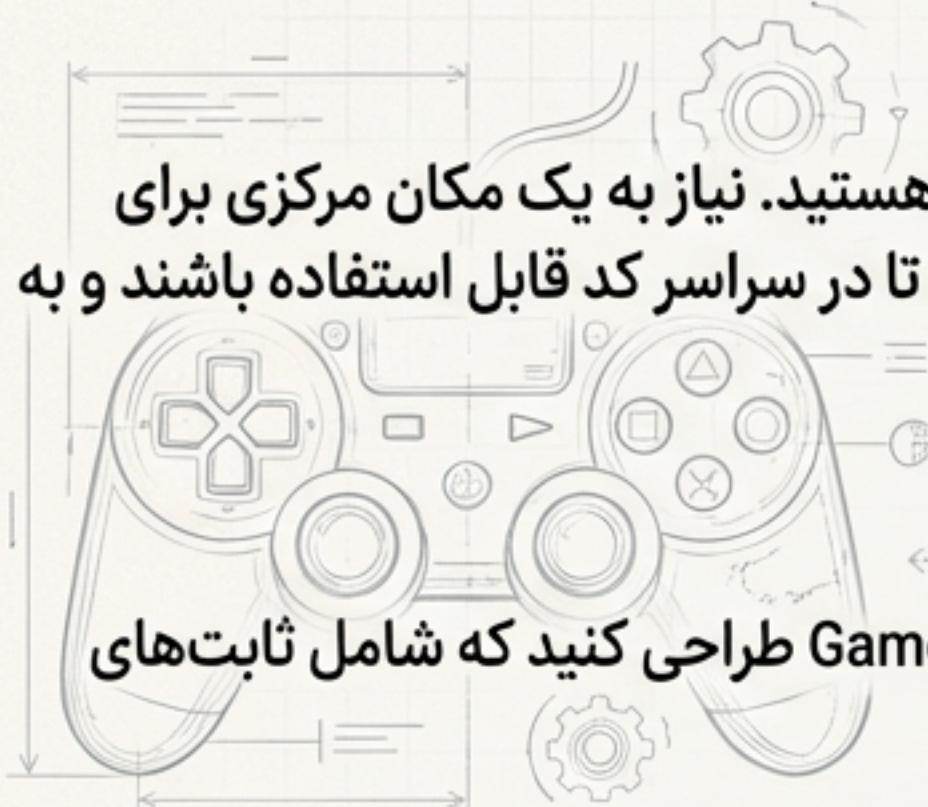
```
// re-assigning a final reference
final List<String> list = new ArrayList<>();
list = new LinkedList<>(); // ?
```

(خطای کامپایل)

```
// modifying the object of a final reference
final List<String> list = new ArrayList<>();
list.add("Hello"); // ?
```

(کامپایل می‌شود)

کارگاه طراحی: ثابت‌های یک بازی



چالش:

شما در حال طراحی یک بازی هستید. نیاز به یک مکان مرکزی برای نگهداری ثابت‌های بازی دارید تا در سراسر کد قابل استفاده باشند و به اشتباه تغییر نکنند.

وظیفه:

یک کلاس به نام GameConstants طراحی کنید که شامل ثابت‌های زیر باشد:

- حداکثر تعداد جان‌ها (MAX_LIVES): عدد صحیح، برابر با 3.
- نسخه بازی (GAME_VERSION): رشته، برابر با '1.0-beta'.
- حالت اشکال‌زدایی (DEBUG_MODE): بولین، برابر با false.

الزامات:

این مقادیر باید از هر کجای برنامه، بدون نیاز به ساختن شیء، قابل دسترس و مطلقاً غیرقابل تغییر باشند.

نقشه پیاده‌سازی:



```
public class GameConstants {  
    // دسترسی سراسری  
    public static final int MAX_LIVES = 3;  
    // غیرقابل تغییر  
    public static final String GAME_VERSION  
        = "1.0-beta";  
    public static final boolean DEBUG_MODE  
        = false;  
}
```

خلاصه نقشه



'final'

- تک تخصیص (Write-once)
- تغییرناپذیر
- نقطه بی بازگشت
- تله ارجاع (آدرس در مقابل محتوا)
- نوشتن روی سنگ



'static'

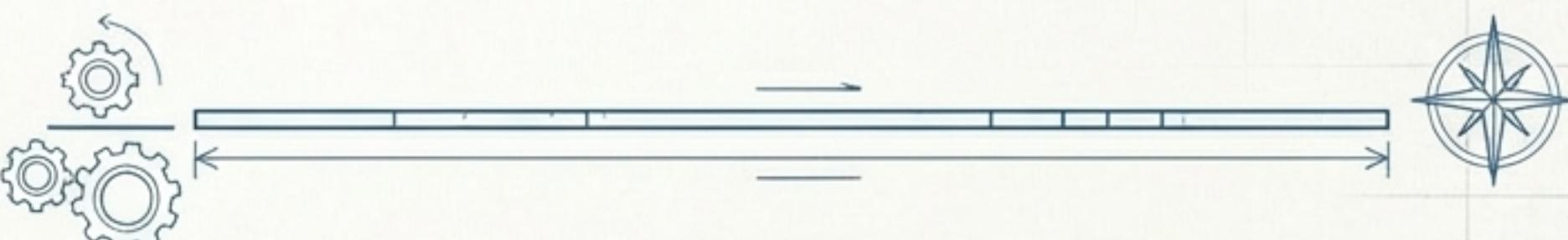
- اشتراکی
- سطح کلاس (Class-level)
- حافظه Metaspace
- دسترسی با `ClassName.member`
- تابلوی اعلانات عمومی

ترکیب طلایی: `public static final .UPPER_SNAKE_CASE`



شما اکنون معمار کدهای پایدارتر هستید

با تسلط بر `static` و `final`، شما کنترل دقیقی بر اشتراک‌گذاری داده، مدیریت وضعیت و پایداری ساختارهای نرم‌افزاری خود به دست آورده‌اید. این ابزارها به شما قدرت می‌دهند تا کدهایی بنویسید که نه تنها کار می‌کنند، بلکه قابل اعتماد، قابل پیش‌بینی و مستحکم هستند.



نگاه به آینده:

در بخش بعدی: آرایه‌ها – مدیریت
مجموعه‌های داده

