

بخش ۲: انواع داده، کالبدشکافی حافظه (Heap و Stack) و فیلدها

تهییه شده توسط: سید سجاد پیراھش

قانون اول جاوا: نوع دهنی ایستا (Statically-Typed)، تور ایمنی شما

کامپایلر شما را موظف می‌کند قبل از استفاده از هر متغیر، نوع داده آن را صراحتاً اعلام کنید. این یک مزیت است، نه محدودیت.



کشف خطا در اولین فرصت

کامپایلر قبل از اجرا، تلاش برای تخصیص نوع داده اشتباه (مثلًا `(`age = "thirty")`) را شناسایی و از باگ‌های بی‌شمار جلوگیری می‌کند.



عملکرد و بهینگی

وقتی JVM دقیقاً می‌داند هر متغیر چقدر حافظه نیاز دارد، می‌تواند حافظه را به بهینه‌ترین شکل ممکن مدیریت کند.

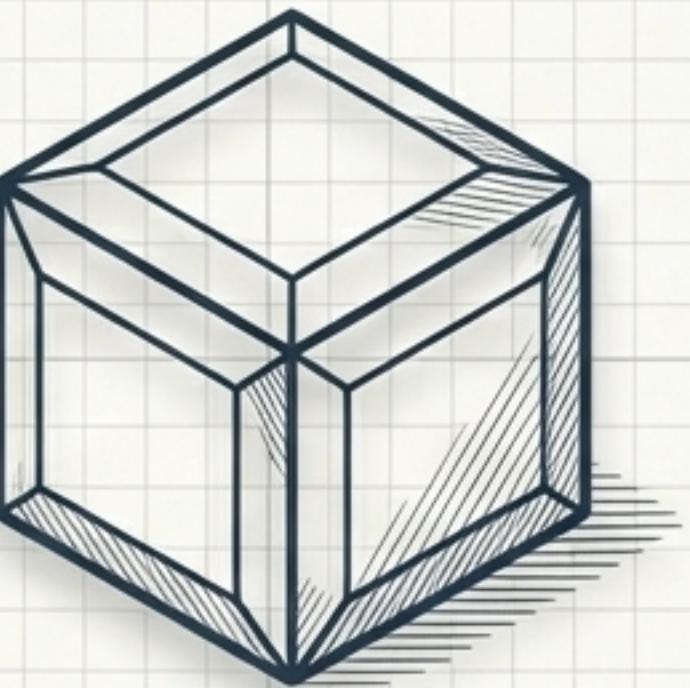


خوانایی و نگهداری کد

نوع‌های مشخص، مانند مستندات زنده عمل کرده و کد را برای دیگران قابل فهم‌تر می‌کنند.

دو دنیای متفاوت داده: سرنخ‌های اصلی تحقیق

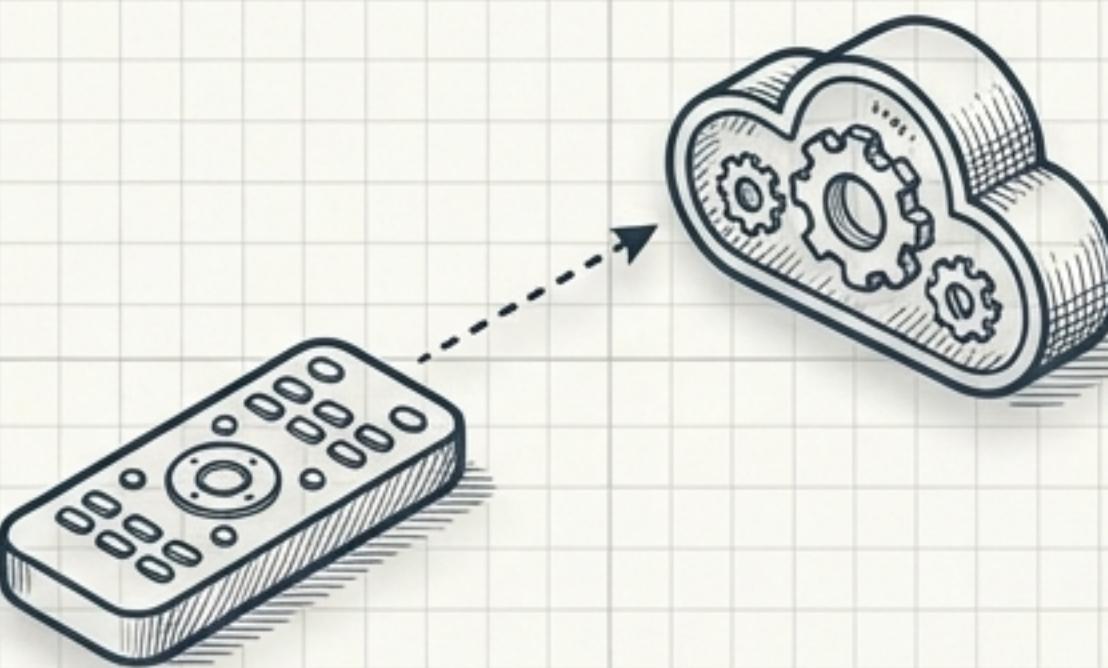
داده‌های اولیه (Primitive)



ارزش خالص. متغیر، خود مقدار است.

```
int age = 30;
```

داده‌های ارجاعی (Reference)



یک کنترل از راه دور. متغیر، آدرسِ مقدار است.

```
String name = new String("Ali");
```

کالبدشکافی داده‌های اولیه: ۸ عنصر بنیادین

boolean



Memory: ~1 بیت

وضعیت‌های منطقی:
isLoggedIn = true;

int



Memory: 4 بایت

پراستفاده‌ترین عدد صحیح:
int score = 100;

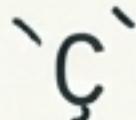
double

پیش‌فرض

Memory: 8 بایت

پیش‌فرض برای اعداد اعشاری دقیق:
double price = 19.99;

char



Memory: 2 بایت

نگهداری یک کاراکتر یونیکد:
char grade = 'A';

long



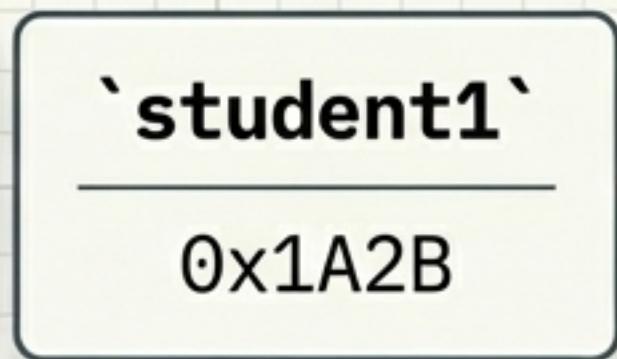
Memory: 8 بایت

اعداد صحیح بسیار بزرگ (با پسوند L):
long population = 8_000_000_000L;

این‌ها ۸ نوع داده بنیادین هستند که مقدار واقعی را مستقیماً در خود ذخیره می‌کنند.

کالبدشکافی داده‌های ارجاعی: آدرس‌هایی به دنیای اشیاء

متغیر ارجاعی، خود شیء را نگه نمی‌دارد، بلکه فقط آدرس (Reference) آن را در حافظه Heap ذخیره می‌کند. هر نوعی غیر از ۸ نوع اولیه (کلاس‌ها، آرایه‌ها) از این دسته است.



متغیر 'student1' فقط یک آدرس را در خود دارد.

خطر: اشتباه میلیارد دلاری!



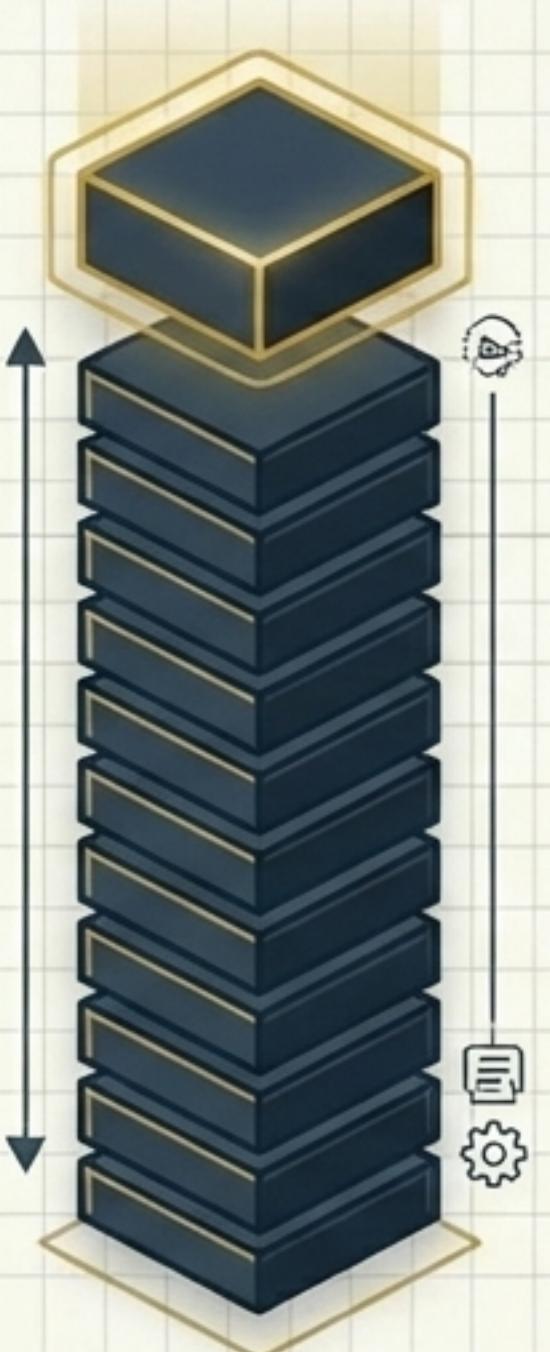
یک ارجاع null به 'هیچ‌کجا' اشاره می‌کند. فراخوانی هر متodi روی آن (null.someMethod()) منجر به خطای مهلك در زمان اجرا می‌شود.

صحنه اجرا: معرفی دو ناحیه حیاتی حافظه

حافظه Stack (پشته)

میز کار موقت و سریع یک متد.

- بسیار سریع
- LIFO (Last-In, First-Out)
- ✗ عمر کوتاه
- متغیرهای محلی و ارجاع‌ها



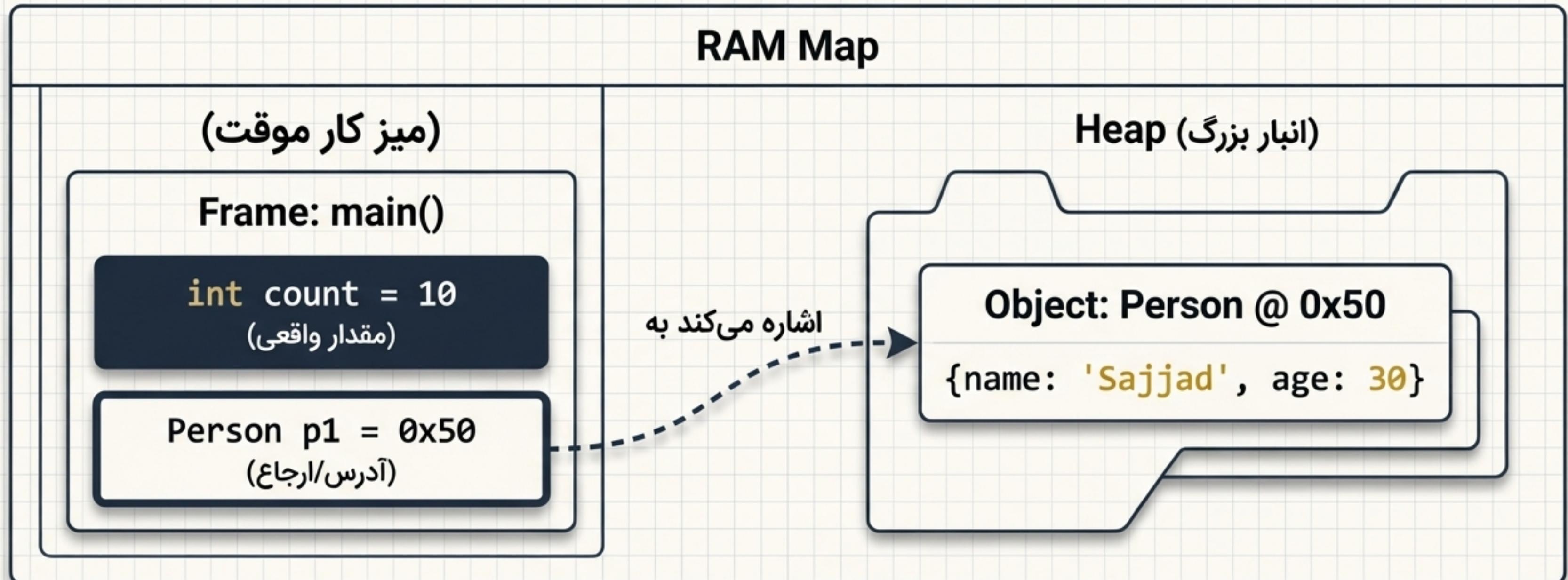
حافظه Heap (هیپ)

انبار بزرگ و اشتراکی برنامه.

- بزرگ
- پویا
- عمر طولانی
- تمام اشیاء و آرایه‌ها



نقشه حافظه: تعامل Heap و Stack در عمل



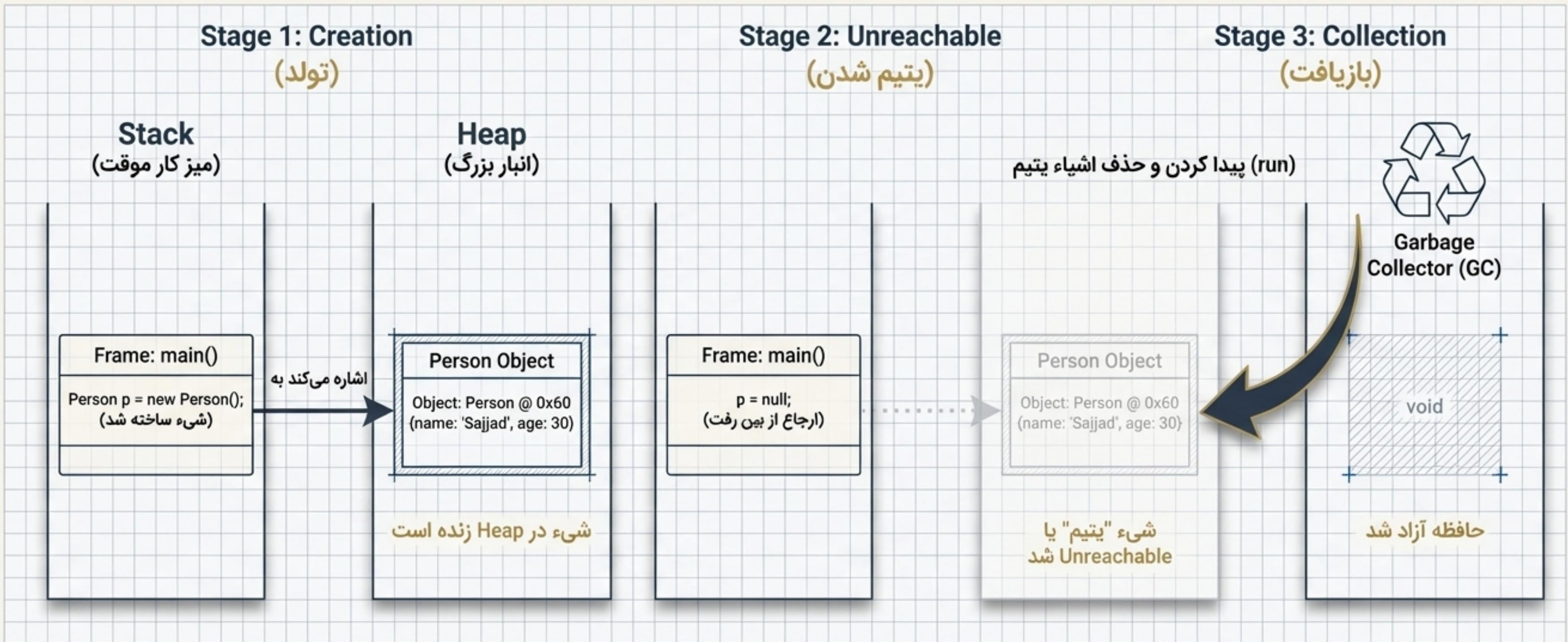
مقادیر اولیه (int) مستقیماً در Stack زندگی می‌کنند. متغیرهای ارجاعی (p1) در Stack موجود در Heap را نگه می‌دارند.

مقایسه رو در رو: Stack در برابر Heap

ویژگی	Stack	Heap
سرعت	بسیار سریع 	کندتر 
اندازه	کوچک و محدود (مثلاً 1MB~)	بزرگ و انعطاف‌پذیر (GBs)
محتوی	متغیرهای محلی و ارجاع‌ها	تمام اشیاء و آرایه‌ها
چرخه حیات	عمر متد (کوتاه)	تا زمان جمع‌آوری زباله (بلند)
مدیریت	JVM خودکار توسط	Garbage Collector توسط
خطای رایج	`StackOverflowError`	`OutOfMemoryError`

چرخه حیات در Heap: از تولد تا بازیافت توسط Garbage Collector

وقتی هیچ ارجاعی از Stack (یا از شیء دیگری) به یک شیء در Heap وجود نداشته باشد، آن شیء 'یتیم' (Unreachable) است. Garbage Collector (GC) این اشیاء را پیدا کرده و حافظه‌شان را آزاد می‌کند.



پرونده ویژه: تفاوت حیاتی فیلدها و متغیرهای محلی

ویژگی	Fields (فیلدها)	Local Variables (متغیرهای محلی)
محل تعریف	داخل کلاس، خارج متدها	داخل متدها** یا بلوک کد
محل ذخیره	به عنوان بخشی از شیء (Heap ➔)	(درون فریم متدها) Stack ➔
چرخه حیات	عمر شیء	عمر متدها
مقدار پیش فرض	دارد (0, false, null)	ندارد (باید حتماً مقداردهی شود)
Access Modifiers	مجاز (private, public...)	غیرمجاز

تجهیزات ویژه: کلاس‌های پوششی (Wrapper Classes)

چرا به آن‌ها نیاز داریم؟

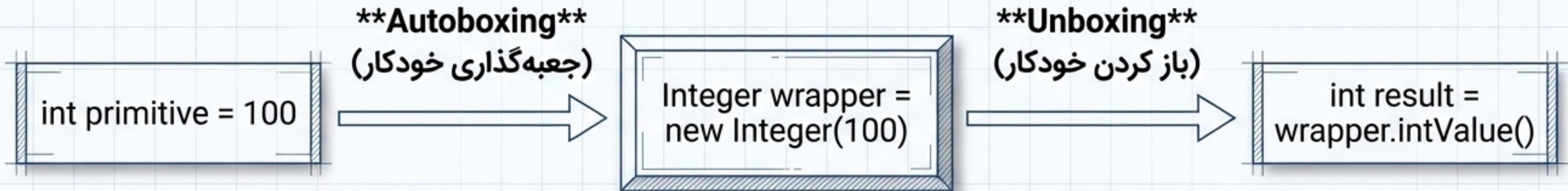
کالکشن‌ها مانند `ArrayList` فقط اشیاء کالکشن‌ها مانند `ArrayList` فقط اشیاء (Reference Types) را می‌پذیرند.

`ArrayList<int>`


راه حل چیست؟

برای هر نوع primitive یک کلاس معادل وجود دارد که آن را "می‌پوشاند" (wraps).

`ArrayList<Integer>` 



تبديل خودکار بین Unboxing و Autoboxing، به ترتیب primitive و Wrapper نام دارد.

هشدارهای حیاتی: تلههای کار با Wrapper ها



Forensic Blueprint



Trap 1: The `null` Ambush

Unboxing خودکار روی یک Wrapper با `null` منجر به `NullPointerException` می‌شود.

```
Integer myScore = null;  
int score = myScore; ← BOOM!  
  
int score = myScore; // BOOM! NullPointerException
```



Trap 2: The `==` Deception

آدرس حافظه را مقایسه می‌کند، نه مقدار را. برای اعداد بزرگ‌تر از 127، آدرس‌ها متفاوت خواهد بود.

```
Integer a = 1000;  
Integer b = 1000;
```

```
System.out.println(a == b); // false  
System.out.println(a.equals(b)); // true
```

برای مقایسه مقدار Wrapper‌ها، **همیشه** از متدهای استفاده کنید. `equals()`

تمرین عملی ۱: پیش‌بینی خروجی (بازجویی از کد)

خروجی دو قطعه کد زیر چیست؟ پاسخ خود را بر اساس تفاوت ذخیره‌سازی در Stack و Heap توجیه کنید.

Primitive (Value Copy)

مقدار 10 در Stack

```
ooo  
int x = 10;  
int y = x; // یک کپی از مقدار 10 ساخته می‌شود  
y = 20;  
System.out.println(x); // خروجی چیست
```

کپی مستقیم مقدار



پاسخ مورد انتظار:

?

Reference (Reference Copy)

شیء در Heap، آدرس در Stack

```
ooo  
Person p1 = new Person("Ali");  
Person p2 = p1; // هر دو شیء کپی می‌شوند  
p2.setName("Reza");  
System.out.println(p1.getName()); // خروجی چیست
```

کپی آدرس، اشاره به همان شیء



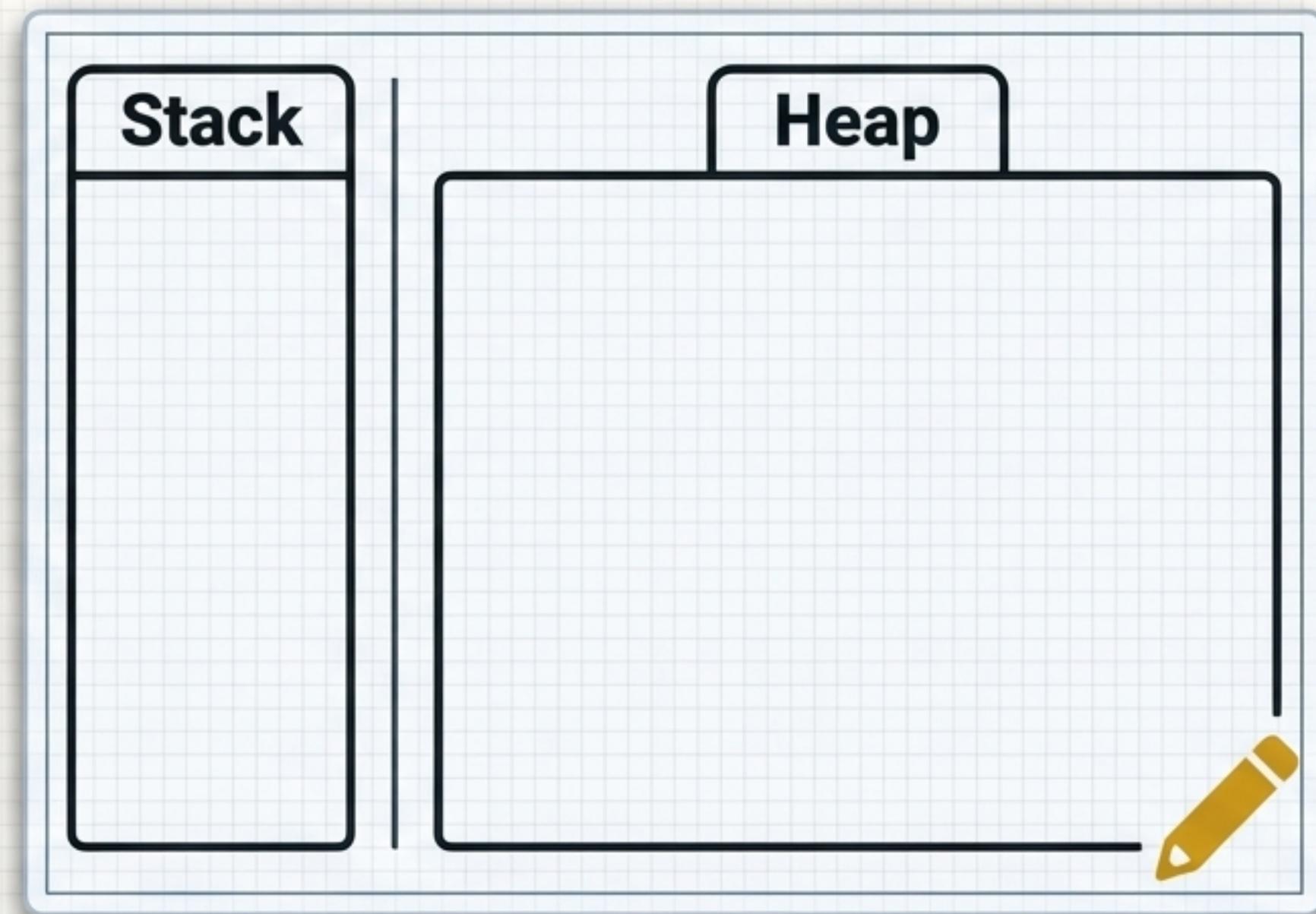
پاسخ مورد انتظار:

?

تمرین عملی ۲: بازسازی صحنه (ترسیم نقشه حافظه)

با توجه به کد زیر، وضعیت نهایی Stack و Heap را در لحظه‌ای که با فلش مشخص شده، ترسیم کنید.
کدام شیء 'یتیم' و آماده جمعآوری زباله است؟

```
public void memoryTest() {  
    Student s1 = new Student("Nima");  
    Student s2 = new Student("Sara");  
    int finalScore = 95;  
    s1 = s2;  
    // وضعیت حافظه در این نقطه چگونه است?  
}
```



خلاصه پرونده: یافته‌های کلیدی



Primitive vs. Reference:* تفاوت حیاتی بین "مقدار واقعی" و "آدرس".



: 'میز کار موقت' (متغیر محلی) در برابر 'انبار بزرگ' (اشیاء).



: مدیریت خودکار حافظه در Heap با حذف اشیاء بدون ارجاع.



تفاوت در محل ذخیره سه د محل ذخیره ((Heap/Stack)) چرخه حیات و داشتن مقدار پیش‌فرض.



Wrapper Traps: برای مقایسه از `equals()` استفاده کنید و همیشه مراقب `null` باشید.

در بخش بعدی:

به دنیای `static` و `final` سفر می‌کنیم تا مفاهیم اشتراک‌گذاری داده بین اشیاء و تغییر تغییرناپذیری را کالبدشکافی کنیم. →