

بخش ۱۱: بازنویسی متدها و کلاس – شخصی‌سازی سرنوشت موروثی

تهیه شده توسط: سید سجاد پیراهاش

چرا رفتار موروثی همیشه کافی نیست؟

وراثت به ما اجازه می‌دهد که دوباره استفاده کنیم، اما اگر رفتار موروثی برای فرزند مناسب نباشد چه؟ یک حیوان عمومی صدایی عمومی دارد. اما یک سگ... باید پارس کند!

صداي عمومي حيوان

```
class Animal {  
    public void makeSound() {  
        System.out.println("The animal makes a sound");  
    }  
}  
  
class Dog extends Animal {  
    // Inherits the generic sound... Not good!  
}  
  
Dog myDog = new Dog();  
myDog.makeSound(); // Output: The animal makes a sound
```



چگونه می‌توانیم این رفتار
را تخصصی کنیم؟

راه حل: بازنویسی متدها با قلم ویرایشگر شما:

بازنویسی (Override) یعنی کلاس فرزند، یک متده را از والد را با همان امضا (نام، پارامترها، نوع خروجی) دوباره پیاده‌سازی می‌کند تا رفتار آن را تغییر دهد.

```
class Dog extends Animal {  
    @Override // This tells the compiler: "I'm changing the parent's method!"  
    public void makeSound() {  
        System.out.println("The dog barks: Woof! Woof!");  
    }  
}
```

```
Dog myDog = new Dog();  
myDog.makeSound(); // Output: The dog barks: Woof! Woof!
```



چرا از `@Override` استفاده کنیم؟

- ایمنی: کامپایلر از اشتباهات تایپی در نام متده جلوگیری می‌کند.
- خوانایی: به وضوح نشان می‌دهد که این یک رفتار بازنویسی شده است، نه یک متده جدید.

قوانين طلایی بازنویسی: نقشه راه برای تغییر رفتار

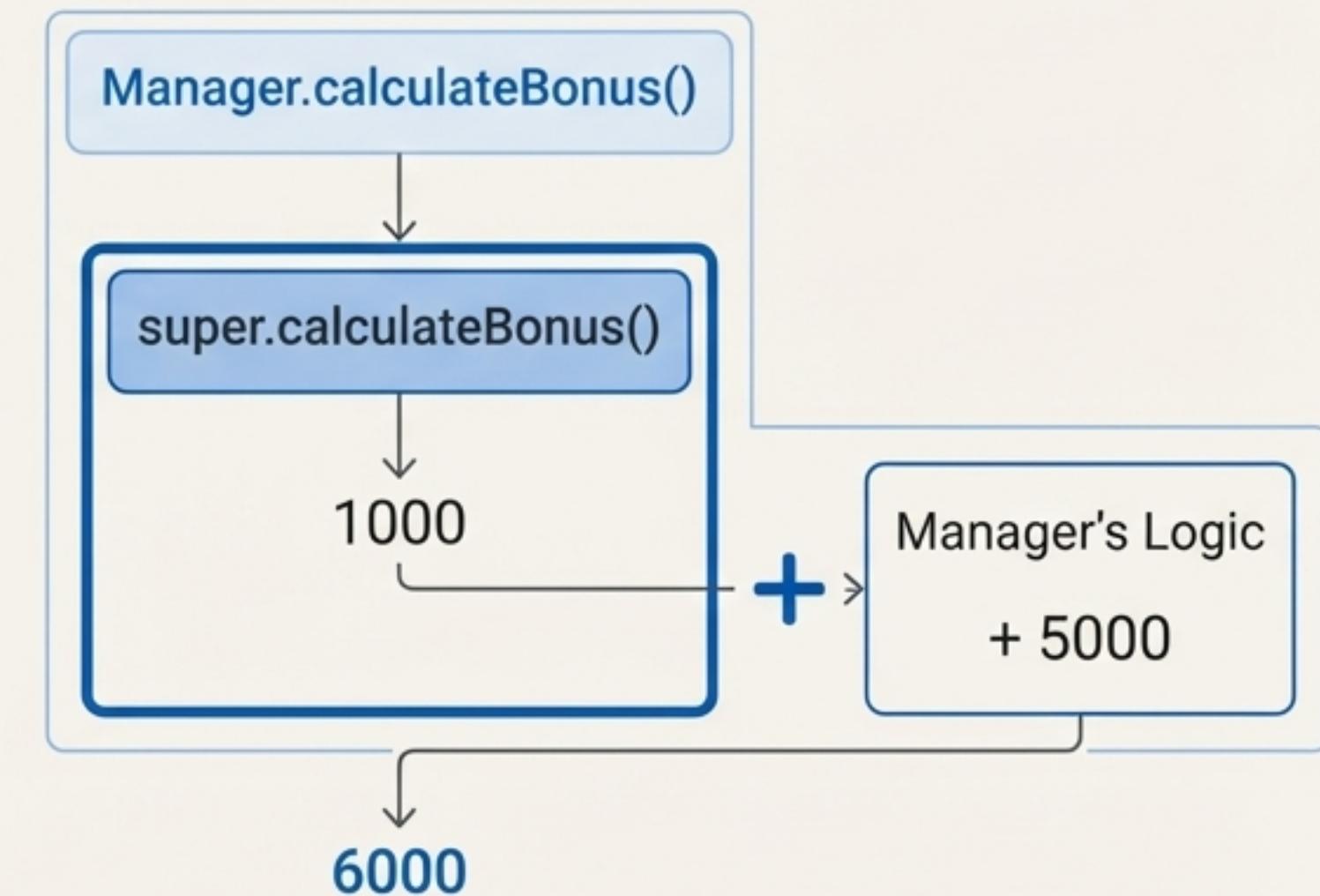
برای بازنویسی صحیح، باید این قوانین را رعایت کنید. نقض آنها منجر به خطای کامپایل می‌شود.

قانون	توضیح	مثال صحیح	مثال غلط
امضای یکسان	نام متدها، تعداد و نوع پارامترها باید دقیقاً یکسان باشند.	void eat() والد: void eat(String food) فرزند:	✗ void eat() والد: ✓ void eat(String food) فرزند:
دسترسی برابر یا بیشتر	سطح دسترسی در فرزند نمی‌تواند محدودتر از والد باشد. "قانون عدم تنزل".	protected void m() والد: public void m() فرزند:	✗ public void m() والد: ✓ private void m() فرزند:
استثناهای برابر یا کمتر	فرزند نمی‌تواند استثنای جدید یا عمومی‌تر از والد پرتاب کند.	throws IOException والد: throws FileNotFoundException فرزند:	✗ throws IOException والد: ✓ throws FileNotFoundException فرزند:
متدهای static قابلOverride	متدهای static به کلاس تعلق دارند نه به شیء. این عمل "Method Hiding" نام دارد.	-	✗ static void test() والد: static void test() فرزند:
متدهای final قابلOverride	کلمه کلیدی final در متدهای final والد، بازنویسی را ممنوع می‌کند.	-	✗ final void calc() والد: final void calc() فرزند:
متدهای private قابلOverride	کلاس فرزند به متدهای private والد دسترسی ندارد که بخواهد آنها را بازنویسی کند.	-	✗ private void secret() والد: private void secret() فرزند:

فراتر از جایگزینی: توسعه رفتار والد با `super`

گاهی نمی‌خواهیم رفتار والد را کاملاً حذف کنیم، بلکه می‌خواهیم چیزی به آن اضافه کنیم.
ما اجازه می‌دهد تا نسخه والد متده را از داخل فرزند فراخوانی کنیم.

```
class Employee {  
    public double calculateBonus(double salary) {  
        return salary * 0.10; // 10% base bonus  
    }  
  
class Manager extends Employee {  
    @Override  
    public double calculateBonus(double salary) {  
        // 1. First, get the base bonus from the parent.  
        double baseBonus = super.calculateBonus(salary);  
        // 2. Then, add the manager's special bonus.  
        return baseBonus + 5000;  
    }  
}
```



بازنویسی (Override) در مقابل سربارگذاری (Overload): دو مفهوم متفاوت

این دو مفهوم اغلب با هم اشتباه گرفته می‌شوند. بیایید تفاوت‌های کلیدی را بررسی کنیم.

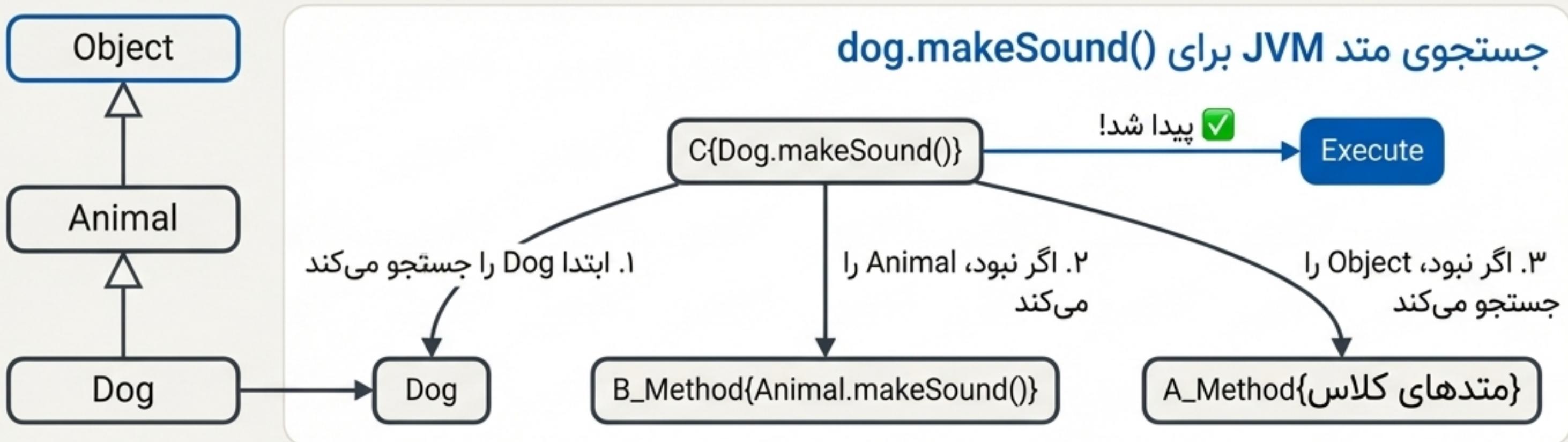
معیار	Override (بازنویسی)	Overload (سربارگذاری)
تعریف	نوشتن مجدد متدهای والد در فرزند	چند متدهای نام یکسان در یک کلاس
محل	در کلاس فرزند	در یک کلاس
رابطه با وراثت	وابسته به وراثت	مستقل از وراثت
امضا	باید دقیقاً یکسان باشد	باید متفاوت باشد
پارامترها	یکسان (تعداد و نوع)	متفاوت (تعداد یا نوع)
زمان تشخیص	Runtime (اتصال پویا)	Compile-time (اتصال ایستا)
هدف	تغییر رفتار متدهای موروثی	ارائه چند نسخه با ورودی متفاوت

Simple Rule of Thumb

- همان متدهای همنام، در همان کلاس، با پارامترهای متفاوت: Overload
- همان متدهای متفاوت، در کلاس فرزند، با رفتار جدید: Override

ارث بری از جد بزرگ: آشنایی با کلاس `Object`

در جاوا، هر کلاسی که می‌نویسید، به طور خودکار از کلاس `java.lang.Object` ارث می‌برد. این کلاس، ریشه تمام کلاس‌ها و به نوعی **DNA** کد منبع شماست.



چرا این مهم است؟! چون `Object` متدہای کلیدی دارد که رفتار بنیادی تمام اشیاء شما را کنترل می‌کنند. تسلط بر آنها برای نوشتن کد حرفه‌ای ضروری است.

تسلط بر سه متده بزرگ (۱): `toString()` – از آدرس حافظه تا اطلاعات خوانا

به طور پیش فرض، `toString()` یک خروجی بی معنی چاپ می کند:

~~Student@1f32e575~~

این به ما چه می گوید؟ هیچ!



کاربرد کلیدی: این متده بهترین دوست شما در هنگام **دیاگ کردن (debugging)** و **ثبت وقایع (logging)** است. همیشه آن را بازنویسی کنید.

راه حل: بازنویسی متده برای ارائه اطلاعات مفید

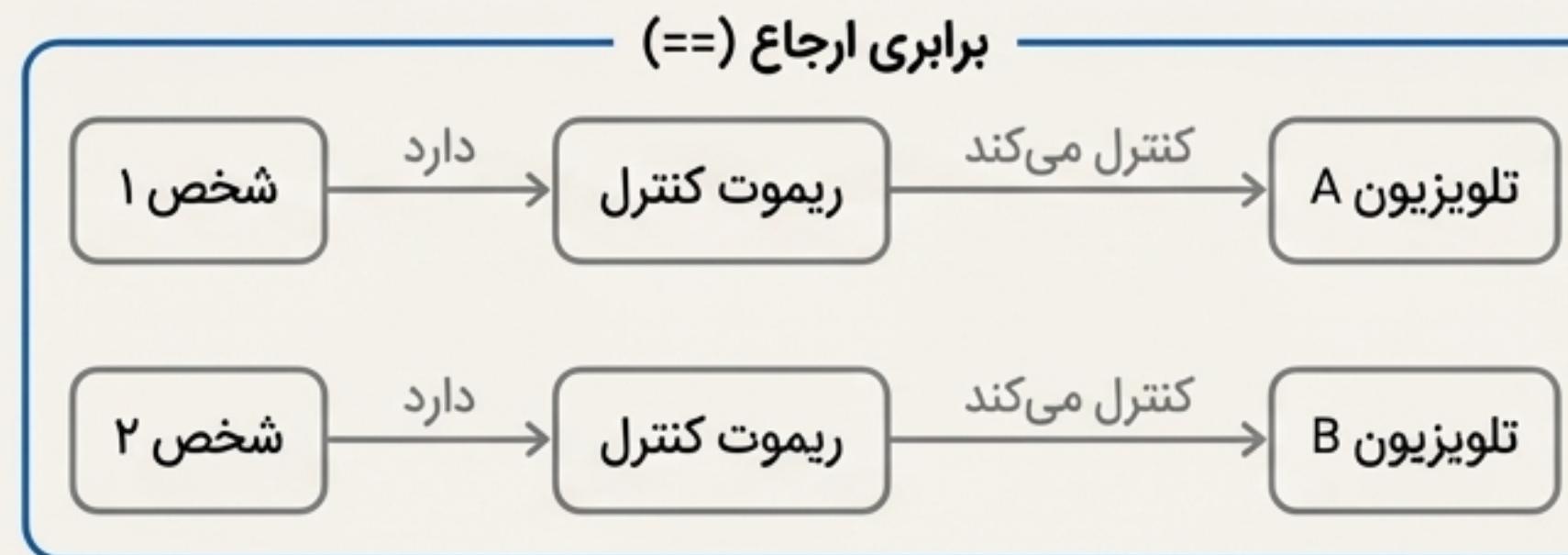
```
public class Student {  
    private String name;  
    private int studentId;  
  
    // ... سازنده ...  
  
    @Override  
    public String toString() {  
        return "Student[ID=" + studentId + ", Name='" +  
            + name + "']";  
    }  
}
```

```
Student s = new Student("Ali", 981001);  
System.out.println(s);
```

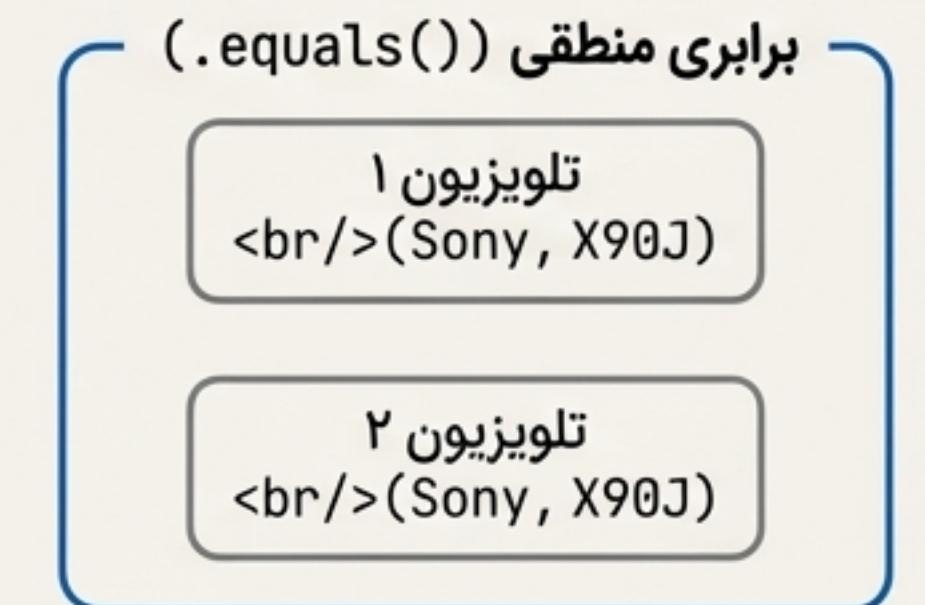
Student[ID=981001, Name='Ali']

تسلط بر سه متده بزرگ (۲): equals() – برابری منطقی در مقابل برابری ارجاع

- `==`: آیا این دو متغیر به یک شیء یکسان در حافظه اشاره می‌کنند؟ (آیا دو نفر، یک ریموت کنترل برای یک تلویزیون دارند؟)
- `equals()`: آیا این دو شیء، با وجود اینکه ممکن است در حافظه جدا باشند، از نظر محتوا و منطق برابر هستند؟ (آیا دو تلویزیون، از یک برنده و مدل هستند؟)



نتیجه
`Remote1 == Remote2 --> false`



نتیجه
`TV1.equals(TV2) --> true`



نکته حیاتی: پیاده‌سازی پیش‌فرض `equals()` در کلاس `Object` دقیقاً مانند `==` عمل می‌کند.
برای مقایسه محتوا، باید آن را بازنویسی کنید.

سلط بر سه متده بزرگ (۳): قرارداد و hashCode() و equals()

قرارداد حیاتی: اگر دو شیء بر اساس `a.equals(b)` برابر هستند، `a.hashCode()` نیز *باید* برابر باشد.

(عكس این موضوع لزوماً صحیح نیست: اشیاء متفاوت می‌توانند که هش یکسان داشته باشند - یک «تصادم» یا (collision).

چرا این اهمیت دارد؟

- کالکشن‌هایی مانند `HashSet` و `HashMap` برای دسته‌بندی و پیدا کردن سریع اشیاء استفاده می‌کنند.
- آن‌ها ابتدا `hashCode()` را چک می‌کنند و فقط اگر کدها برابر بودند، برای اطمینان نهایی `equals()` را فراخوانی می‌کنند.

آزمون نقض قرارداد

فرض کنید `equals()` را بازنویسی کرده‌ایم اما `hashCode()` را نه. چه اتفاقی برای این کد می‌افتد؟

```
HashMap<Point, String> map = new HashMap<>();  
Point p1 = new Point(1, 2);  
map.put(p1, "Origin");
```

چرا این `null` برمی‌گرداند؟

// `map.get(new Point(1, 2))` will return null! WHY?

p1

`hashCode() → 12345 (default)`



`new Point(1, 2)`

`hashCode() → 67890 (default)`

پاسخ: نقض قرارداد باعث می‌شود `HashMap` شیء شما را پیدا نکند، چون کدهای هش متفاوت هستند، حتی اگر اشیاء از نظر منطقی برابر باشند!

بهترین رویه: چرا `hashCode` و `equals` تولید شده توسط IDE بهتر است؟

نوشتن دستی این متدها مستعد خطا است. IDE‌های مدرن (IntelliJ, Eclipse) کدی بهینه و امن تولید می‌کنند.

۱. صحت قرارداد (Contract Correctness): همیشه تضمین می‌کند که اگر `equals` صحیح باشد، `hashCode` نیز صحیح خواهد بود.



۲. مدیریت Null (Null Handling): به درستی مقادیر `null` را در فیلدها مدیریت می‌کند تا جلوگیری شود.



۳. کارایی هش (Hash Performance): از الگوریتم‌های اثبات شده (مانند استفاده از عدد اول ۳۱) برای تولید کدهای هش با توزیع مناسب استفاده می‌کند که منجر به تصادم (collision) کمتر در `HashMap` می‌شود.



```
result = 31 * result + (field == null ? 0 : field.hashCode());
```

```
Objects.hash(field1, field2),  
Objects.equals(a, b)
```

۴. کد تمیزتر (Cleaner Code): از کلاس‌های کمکی مانند `java.util.Objects` استفاده می‌کند.



به IDE خود اعتماد کنید. از قابلیت `Alt+Insert or 'Generate...'` برای تولید این متدها استفاده کنید.

جعبه کد: یک کلاس `Point` حرفه‌ای و کامل

این کلاس تمام مفاهیم را در کنار هم نشان می‌دهد: یک `equals()`، یک `toString()` منطقی، و یک `hashCode()` مفید، که به قرارداد پاییند است.

```
public final class Point {  
    private final int x;  
    private final int y;  
  
    public Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
    // Getter methods for x and y...  
    @Override  
    public String toString() {  
        return "Point(" + x + ", " + y + ")";  
    }  
    @Override  
    public boolean equals(Object o) {  
        if (this == o) return true;  
        if (o == null || getClass() != o.getClass()) return false;  
        Point point = (Point) o;  
        return x == point.x && y == point.y;  
    }  
    @Override  
    public int hashCode() {  
        // Using the modern, safe Objects.hash() helper method.  
        return Objects.hash(x, y);  
    }  
}
```

خروجی خوانا برای دیباگ

بررسی برابری محتوایی (مقایسه
(y و x

تضمین قرارداد با `equals`

چالش شما: طراحی کلاس `Product`

کلاسی به نام `Product` با فیلدهای `productId` (String) و `name` (String) طراحی کنید.

لیست الزامات

- یک سازنده (constructor) برای مقداردهی اولیه فیلدها بنویسید.
- متد (toString) را بازنویسی کنید تا خروجی خوانایی مانند ""Product[id=A-123, name=Laptop]"" تولید کند.
- متد `()` را بازنویسی کنید. دو محصول برابرند اگر و فقط اگر آنها یکسان باشد.
- متد (hashCode) را بازنویسی کنید تا با (equals) سازگار باشد.

مورد تست

```
Product p1 = new Product("A-123", "Laptop");
Product p2 = new Product("A-123", "Gaming Laptop");
Product p3 = new Product("B-456", "Mouse");

// p1.equals(p2) should be true.
// p1.equals(p3) should be false.
// A HashSet containing p1 should report that it contains p2.
```

خلاصه نهایی: متدهای کلیدی کلاس `Object`

این جدول، راهنمای سریع شما برای تصمیم‌گیری در مورد بازنویسی متدهای `Object` است.

نکات مهم	باید Override شود؟	هدف	متدها
برای دیباگ و لاغ حیاتی است.	<input checked="" type="checkbox"/> توصیه می‌شود	نمایش متنی شیء	<code>toString()</code>
باید با <code>hashCode()</code> همراه باشد.	<input checked="" type="checkbox"/> اگر نیاز به مقایسه محتوا دارد	مقایسه محتوای	<code>equals()</code>
equals-hashCode قرارداد حیاتی است.	<input checked="" type="checkbox"/> اگر <code>equals</code> را بازنویسی کردید	کد هش برای کالکشن‌ها	<code>hashCode()</code>
معمولًاً جایگزین‌های بهتری مانند copy constructors وجود دارد.	 پیچیده	کپی کردن شیء	<code>clone()</code>
قابل بازنویسی نیست.	 است <code>final</code>	اطلاعات کلاس در زمان اجرا	<code>getClass()</code>

شما اکنون سرنوشت اشیاء خود را کنترل می‌کنید

@Override: شما می‌توانید رفتار موروثی را با دقت و ایمنی تغییر دهید.



super: شما می‌توانید رفتار والد را توسعه دهید، نه فقط جایگزین کنید.



toString(): شما به اشیاء خود صدا می‌بخشید تا در دیباگ به شما کمک کنند.



equals() & hashCode(): شما هويت منطقی اشیاء خود را اعریف می‌کنید و تضمین می‌کنید که در کالکشن‌ها به درستی رفتار می‌کنند.



با تسلط بر این مفاهیم، شما دیگر فقط کد نمی‌نویسید؛ شما کلاس‌هایی قابل اعتماد، حرفه‌ای و آماده تولید می‌سازید که به درستی با اکوسیستم جاوا تعامل دارند.

در بخش بعدی: قدرت واقعی OOP با **Polymorphism**.

