



## بخش ۱۷: مدیریت استثناهای تدافعی – (Exception Handling)

```
codet strg: function() {  
}
```



تهیه شده توسط: سید سجاد پیراهاش



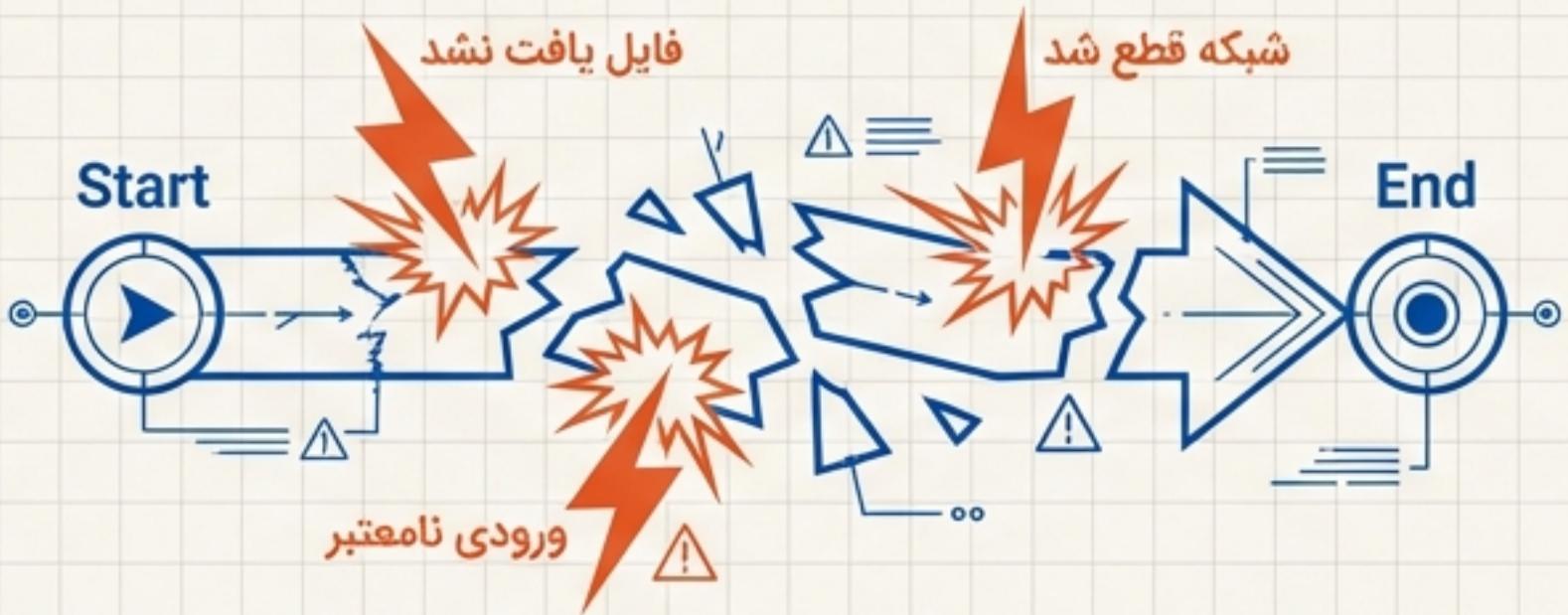
```
int sen() {  
    Exception(Enstten "Handling");  
  
    Sem1.strg = return.exists("XHandling");  
}
```

# دنیای ایدهآل در برابر دنیای واقعی

## دنیای ایدهآل

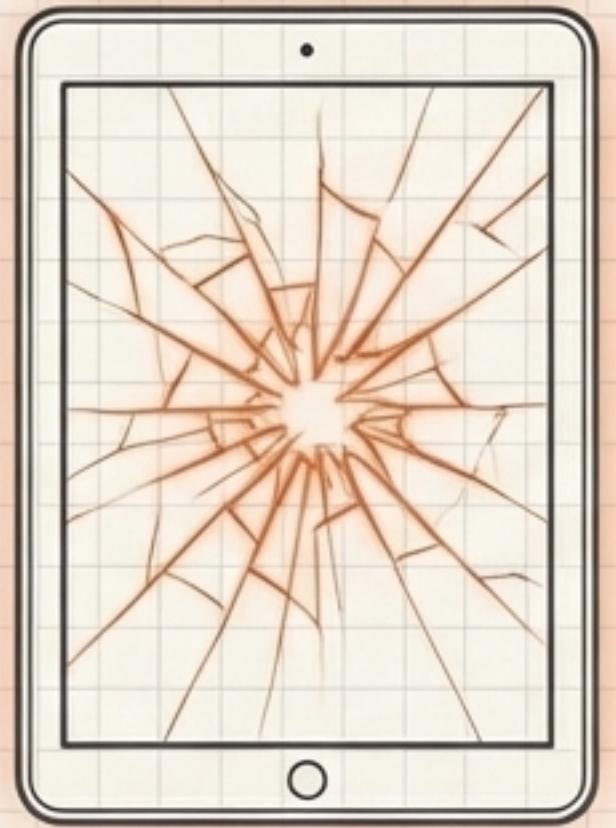


## دنیای واقعی

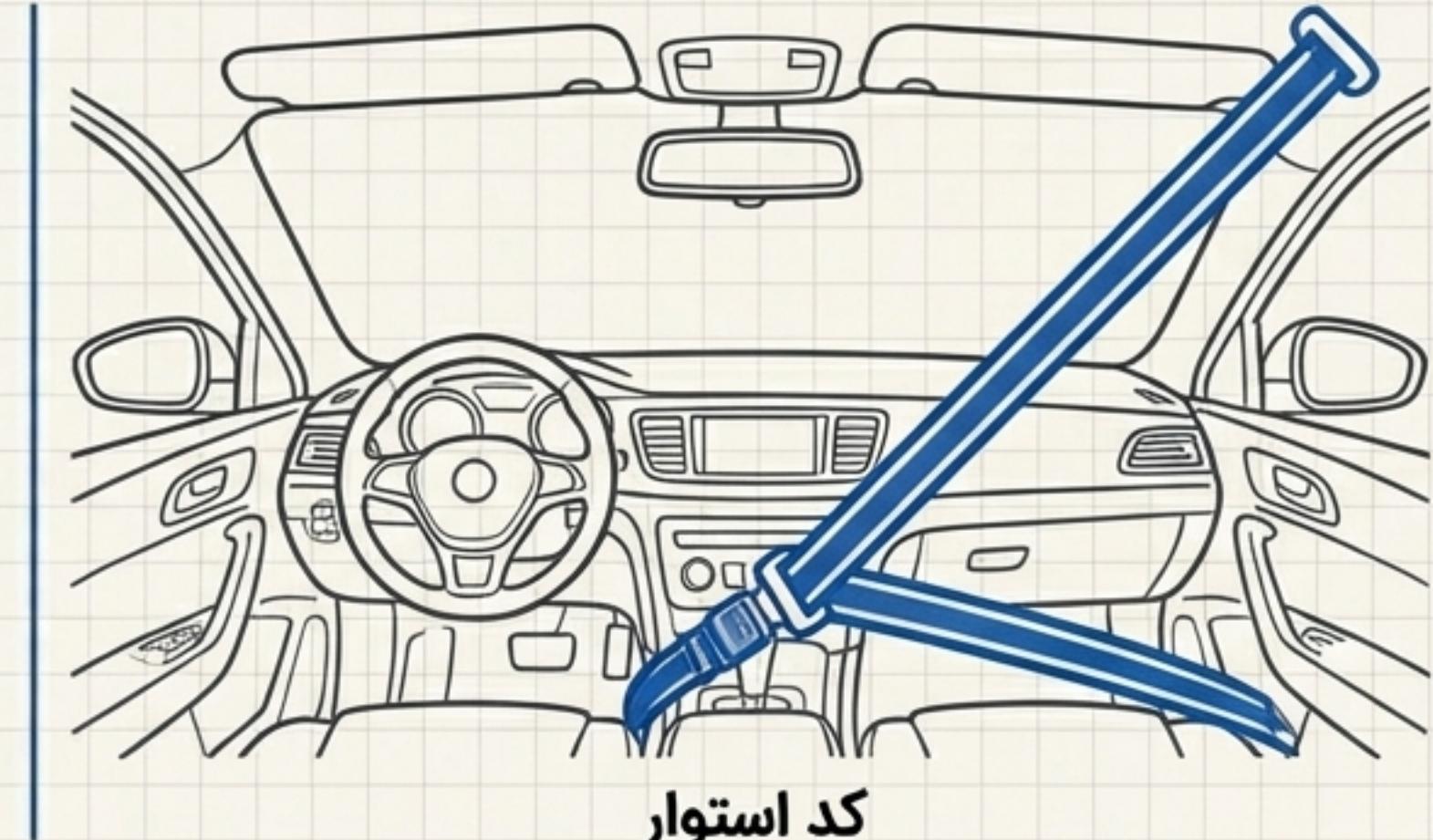


- تا به حال، ما در یک دنیای ایدهآل کد می‌زدیم: همه چیز طبق انتظار کار می‌کند.
- اما در دنیای واقعی، برنامه شما تحت محاصره است: فایل‌ها حذف می‌شوند، کاربران ورودی اشتباه می‌دهند.
- کد بدون دفاع، کدی شکننده است. با اولین حمله (خطا)، فرو می‌ریزد.
- به این "اشتباهات" در زمان اجرای برنامه، استثنای (Exception) گفته می‌شود.

# کمربند ایمنی برنامه شما



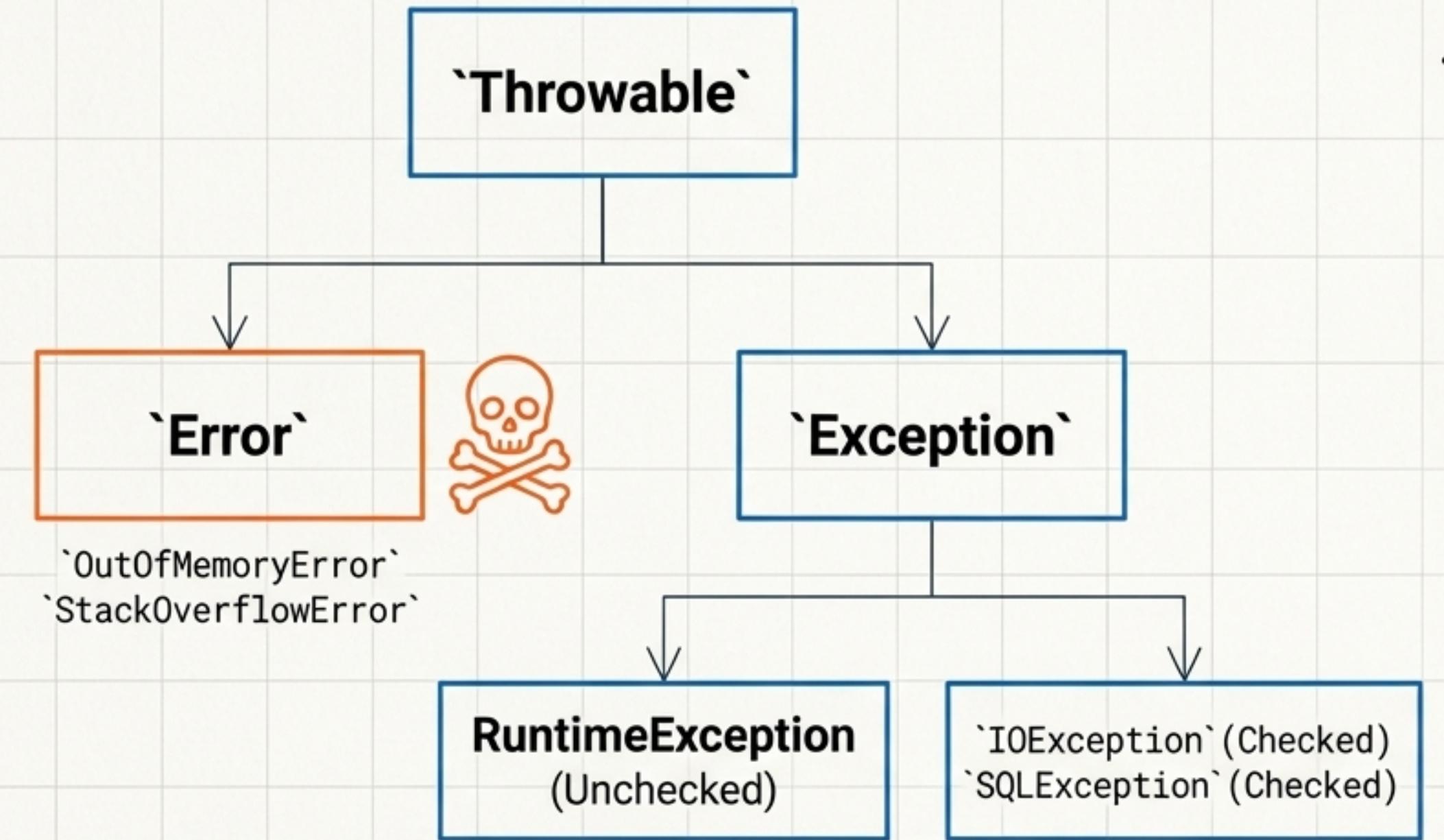
کد شکننده



کد استوار

- مدیریت استثناهای (Exception Handling) فقط یک سینتکس نیست؛ یک فلسفه است.
- این هنر پیش‌بینی شکست و طراحی کدی است که حتی در شرایط بحرانی، کنترل را از دست ندهد و در یک وضعیت پایدار باقی بماند.
- این کمربند ایمنی برنامه شماست. شما آن را نمی‌نویسید، چون انتظار تصادف دارید؛ شما آن را می‌نویسید تا در صورت وقوع تصادف، تصادف، جان سالم به در ببرید.

# شناخت دشمن: سلسله مراتب `Throwable`

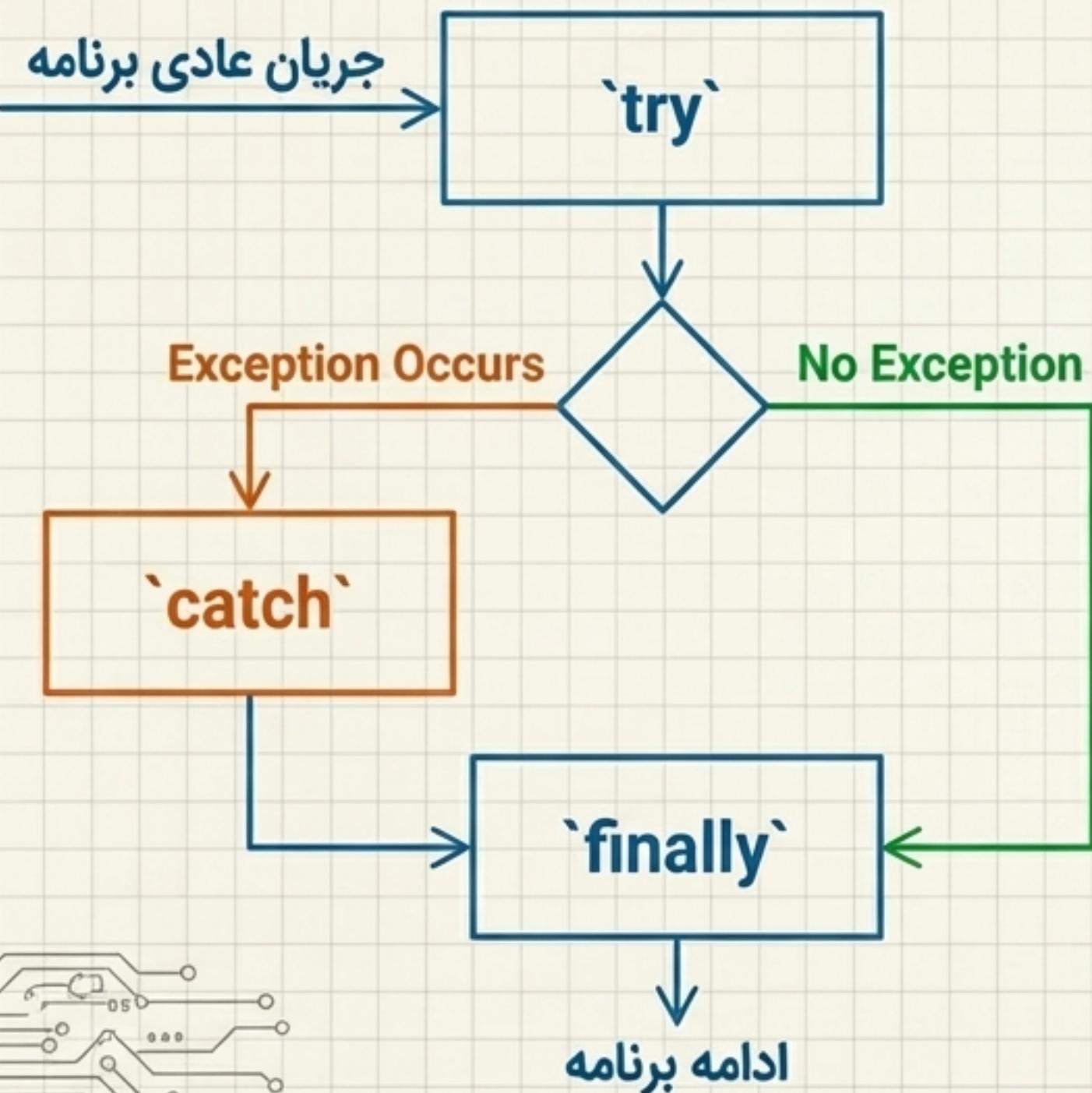


- **Throwable**: فرمانده کل خطاها در جاوا.  
همه خطاها اشیائی هستند که از این کلاس ارث بری می‌کنند.
- **Error**: فجایع سیستمی غیرقابل بازیابی.  
این‌ها خطاها جدی هستند که برنامه نباید (و نمی‌تواند) آن‌ها را مدیریت کند. ما با این‌ها نمی‌جنگیم. مأموریت لغو شده است.
- **Exception**: حوادث قابل مدیریت.  
این میدان نبرد ماست. این‌ها خطاها بی هستند که برنامه می‌تواند و باید آن‌ها می‌تواند و باید آن‌ها را مدیریت کند.

# دو نوع تهدید: خطاها قابل پیش‌بینی در برابر باگ‌های برنامه‌نویس

معیار	Checked Exception (استثنای بررسی شده)	Unchecked Exception (استثنای بررسی نشده)
مفهوم	خطاهایی که از کنترل مستقیم برنامه خارج هستند (مشکلات فایل، شبکه).	خطاهای برنامه‌نویسی و منطقی (باگ).
بررسی در کامپایل	اجباری.  کامپایلر شما را مجبور به مدیریت نمی‌کند.	اختیاری.  کامپایلر شما را مجبور به مدیریت نمی‌کند.
کلاس والد	.(`Exception` (به جز `RuntimeException`	`RuntimeException`
مثال‌های رایج	`IOException`, `SQLException`	`NullPointerException`, `ArithmaticException`
فلسفه	فرمانده، دشمن خارجی ممکن است حمله کند. آماده باش!	یک خائن در صفوف ماست (یک باگ در کد). او را پیدا و حذف کن.
نحوه مدیریت	باید `catch` شوند یا با `throws` واگذار شوند.	معمولًاً نباید `catch` شوند؛ باید از وقوعشان جلوگیری شود.

# سنگر دفاعی شما: بلوک `try-catch-finally`



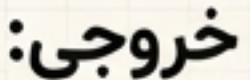
- منطقه خطر (The Danger Zone). کدی که "ممکن است" استثنای ایجاد کند در این این بلوک قرار می‌گیرد.
- سنگر ایمنی (The Safety Net). اگر در منطقه خطر استثنایی رخ دهد، اجرای 'try' متوقف شده و برنامه به اینجا پناه می‌آورد. هر نوع خطر، سنگر مخصوص خود را دارد.
- گروه پاکسازی (The Cleanup Crew). این بلوک همیشه اجرا می‌شود، چه استثنایی رخ بددهد چه ندهد. برای آزادسازی منابع حیاتی (مثل بستن فایل‌ها) ضروری است.

# تمرین میدانی: خنثی‌سازی تقسیم بر صفر

## کد شکننده (Fragile Code)

```
public void unsafeDivide(int a, int b) {  
    int result = a / b; // CRASHES if b is 0!  
    System.out.println("Result: " + result);  
}
```

خروجی:  
Exception in thread "main"  
java.lang.ArithmeticException: / by zero



## کد استوار (Robust Code)

```
public void safeDivide(int a, int b) {  
    try {  
        int result = a / b;  
        System.out.println("Result: " + result);  
    } catch (ArithmetricException e) {  
        System.err.println("Error: Division by zero");  
        // Stack Trace  
        e.printStackTrace();  
    }  
}
```

;"خطا: تقسیم بر صفر مجاز نیست"  
نمایش ردپای خطأ (Stack Trace) برای دیباق کردن //  
e.printStackTrace();

خروجی: خطأ: تقسيم بر صفر مجاز نیست.

# عملیات پاکسازی: تضمین آزادسازی منابع با `finally`

وقتی با منابع خارجی مانند فایل‌ها یا اتصالات شبکه کار می‌کنیم، بسیار مهم است که آن‌ها را پس از اتمام کار ببندیم. بلوک `finally` تضمین می‌کند که این پاکسازی همیشه انجام می‌شود، حتی اگر خطایی رخ دهد.

```
Reader reader = null;
try {
    reader = new FileReader("myFile.txt");
    // ... code to read from the file ...
} catch (IOException e) {
    e.printStackTrace();
} finally {
    if (reader != null) {
        try {
            reader.close(); // ALWAYS runs!
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}
```

# ارتقاء زرادخانه: پاکسازی خودکار با `try-with-resources`

## روش کلاسیک با `finally`

```
Reader reader = null;
try {
    reader = new FileReader("myFile.txt");
    // ... code to read from the file ...
} catch (IOException e) {
    e.printStackTrace();
} finally {
    if (reader != null) {
        try {
            reader.close(); // ALWAYS runs!
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}
```

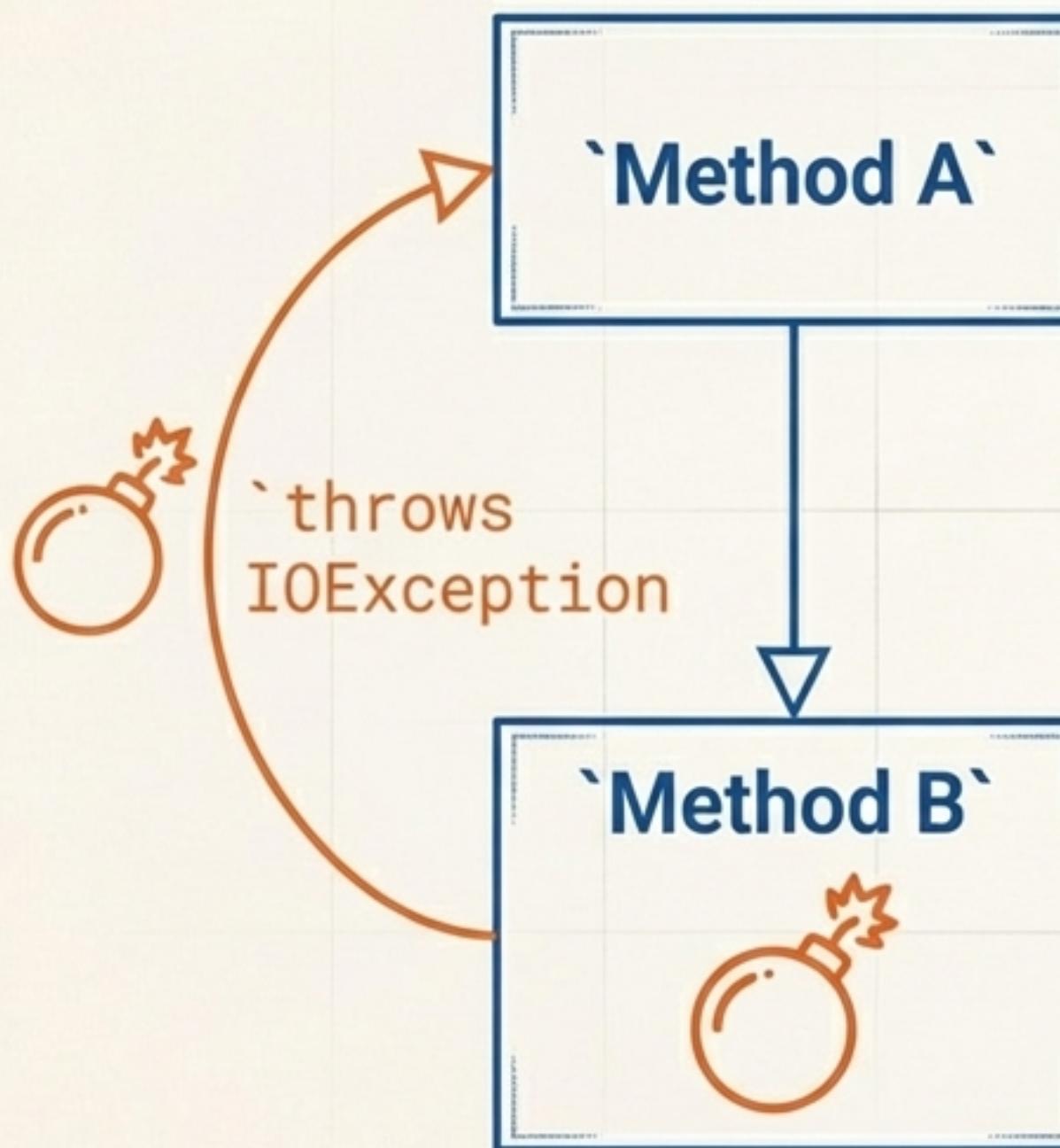
## روش مدرن با `try-with-resources`

```
// The modern, cleaner way
try (Reader reader = new FileReader("myFile.txt")) {
    // ... code to read from the file ...
    // reader.close() is called automatically!
} catch (IOException e) {
    e.printStackTrace();
}
```



- بلوک `finally` قدرتمند است، اما می‌تواند کد را پیچیده و تودرتو کند.
- جاوا 7 یک راه هوشمندانه‌تر معرفی کرد: [Try-with-Resources](#). این ساختار به طور خودکار منابعی را که اینترفیس [AutoCloseable](#) را پیاده‌سازی کرده‌اند، می‌بندد.
- کد تمیزتر، امن‌تر و خواناتر است. این استاندارد مدرن برای کار با منابع است.

# واگذاری مسئولیت: کلمه کلیدی `throws`



- گاهی اوقات، یک متده نمی‌داند (یا نباید بداند) چگونه یک خطای خاص را مدیریت کند.
- کلمه کلیدی `throws` به کامپایلر اعلام می‌کند: "من این خطر را مدیریت نمی‌کنم و مسئولیت آن را به هرکنی که مرا فراخوانی کرده واگذار می‌کنم."
- ردپای پشته (Stack Trace): اگر استثنا تا بالا واگذار شود و هیچ متده آن را `catch` نکند، برنامه کرش می‌کند و یک "ردپای پشته" چاپ می‌کند. این ردپا، زنجیره فراخوانی متدهاست که نشان می‌دهد خطا از کجا شروع شده و چگونه به بالا  منتشر (Propagate) یافته است.

# اشتباهات مرگبار: چگونه سنگر خود را به تله تبدیل کنیم

1. قورت دادن استثنا (Empty Catch Block):  
این معادل دیدن یک آتشسوزی و خاموش کردن آژیر خطر است. شما مشکل را پنهان می‌کنید،  
نه حل. برنامه در یک وضعیت نامعتبر به کار خود ادامه می‌دهد.



2. شکار با تور ماهیگیری بزرگ (Catching `Exception`):  
این کار باگ‌های غیرمنتظره (مانند `NullPointerException`) را در کنار خطاهای قابل پیش‌بینی پنهان  
می‌کند. دقیق باشید. فقط خطاهایی را که انتظار دارید و می‌دانید چگونه مدیریت کنید، بگیرید.



3. استفاده از استثناهای برای کنترل جریان (Control Flow):  
استفاده از `try-catch` برای منطق عادی برنامه، مانند استفاده از موشک برای باز کردن یک در است.  
بسیار کند، ناخوانا و غیراصولی است. استثناهای برای شرایط استثنایی هستند.



# آزمون پیش‌بینی: چه استثنایی پرتاب خواهد شد؟

کویز ۱

```
String s = null;  
System.out.println(s.length());
```

پیش‌بینی:  
`NullPointerException`

کویز ۲

```
int[] arr = new int[5];  
arr[5] = 10;
```

پیش‌بینی:  
`ArrayIndexOutOfBoundsException`

کویز ۳

```
Object x = new Integer(0);  
String s = (String)x;
```

پیش‌بینی:  
`ClassCastException`

# چالش `finally` خروجی این کد چیست؟

```
public static int testFinally() {  
    try {  
        System.out.println("Entering try block");  
        return 1; // Attempting to return  
    } catch (Exception e) {  
        System.out.println("Entering catch block");  
        return 2;  
    } finally {  
        System.out.println("Executing finally block");  
    }  
}
```

سوال: وقتی متدها `testFinally()` را فراخوانی می‌کنیم، چه متنی در کنسول چاپ می‌شود و چه مقدار عددی بازگردانده می‌شود؟

"Entering try block"  
"Executing finally block"

بازگردانده می‌شود: ۱

پاسخ:

چاپ می‌شود:

# تحلیل و اصلاح: این کد از نظر فلسفی چه مشکلی دارد؟

```
public void processData(String[] data) {  
    try {  
        String item = data[0];  
        System.out.println(item.toUpperCase());  
    } catch (NullPointerException | ArrayIndexOutOfBoundsException e) {  
        آیا این ایده خوبی است //  
        System.err.println("An input error occurred!");  
    }  
}
```

کد برای تحلیل

- آیا **NullPointerException** و **ArrayIndexOutOfBoundsException** پیش‌بینی هستند یا بگهای برنامه‌نویسی؟
- آیا باید آن‌ها را **catch** کرد یا باید با کدنویسی بهتر از وقوع آن‌ها جلوگیری نمود؟

```
public void processData(String[] data) {  
    // Defensive Check: Prevent the error, don't just catch it.  
    if (data != null && data.length > 0 && data[0] != null) {  
        String item = data[0];  
        System.out.println(item.toUpperCase());  
    } else {  
        System.err.println("Invalid data provided.");  
    }  
}
```

راه حل تدافعی

# خلاصه مأموریت: از کد شکننده به کد استوار



استثنایاً رویدادهایی هستند که جریان عادی برنامه را مختل می‌کنند.



ساختار اصلی شما برای محاصره کد خطرناک، مدیریت خطا، و پاکسازی است



برای خطاها خارجی است (باید مدیریت شوند): **Checked**  
برای باگ‌های داخلی است (باید از آن‌ها جلوگیری شود): **Unchecked**



مسئولیت را واگذار می‌کند؛ آن را هوشمندانه به کار ببرید.



استاندارد مدرن و برتر برای کار با منابع است.



هرگز، هرگز، هرگز `catch` را خالی نگذارید! حداقل خطا را لاغ کنید (e.printStackTrace()).



شما دیگر فقط یک معمار نیستید. شما فرمانده دژی هستید که می‌توانید از آن در  
برابر هرجو مرجدنیای واقعی دفاع کنید.