Try-Catch ve Finally

Bir exception'ı gerçekten handle etmenin mekanizması: temel try-catch bloğu, birden fazla catch bloğu (sırayla eşleşme, süper sınıf/alt sınıf sıralama kısıtı), | ile multi-catch (Java 7+), finally'nin koşulsuz olarak her zaman çalışması, ve finally içinde return /throw'un try/catch'in ürettiği her şeyi sessizce ezmesi. Exception Handling serisinin 2.'si.

Başlangıç 20 dk

Oluşturulma tarihi: 2 Ekim 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/try-catch-finally

---

"Exception'lara Giriş", hiçbir şey bir exception'ı handle etmediğinde ne olduğunu gösterdi -- program gürültülü bir şekilde, bir stack trace'le sonlanıyor. Bu ders, bu olmadan ÖNCE araya girmenin gerçek mekanizmasını kapsıyor: `try`, `catch`, ve `finally`.

Try ve Catch Nedir?

Bir `try` bloğu, bir kod bölümünü "bu bir exception fırlatabilir -- izle" olarak işaretler. Hemen ardından gelen bir `catch` bloğu, o `try` bloğunun İÇİNDE BELİRLİ bir tipte exception GERÇEKTEN oluşursa ne yapılacağını belirtir. Hiçbir exception fırlatılmazsa, her `catch` bloğu basitçe atlanır, ve çalışma hiç orada değillermiş gibi hemen onların ardından devam eder.

Neden Var?

"Exception'lara Giriş", exception'ların kendisinin bir dil özelliği olarak neden var olduğunu zaten kapsadı. `try` / `catch`, daha dar, daha pratik bir soruyu cevaplar: belirli bir kod parçasının başarısız OLABİLECEĞİNİ bildiğinde, bu başarısızlığı tüm programı sonlandırmasına izin vermek yerine, tam olarak ele almaya hazır olduğun yere nasıl sınırlarsın? Riskli kodu `try`'a, kurtarma mantığını `catch`'e sarmak, "bu başarısız olursa ne olur" handling'ini gerçekten başarısız olabilecek kodun fiziksel olarak yanında tutar.

Tarihçe

`try` / `catch` / `finally` keyword'leri olarak Java'dan öncesine dayanır -- Java'nın tasarımcıları bu tam üç parçalı şekli yeni bir tane icat etmek yerine doğrudan C++'ın exception handling sözdiziminden (1980'lerin sonunda C++'a eklendi) ödünç aldı. `finally`'nin kendisi, sonucundan bağımsız olarak cleanup kodunun çalışmasını garanti etmesi, Java'nın bu ödünç alınan şekle kendi eklemesi -- C++'ın doğrudan bir eşdeğeri yok, bunun yerine cleanup'ı garanti etmek için farklı bir desene (RAII) dayanıyor.

Temel try-catch Bloğu

```
// Compare this to UncaughtExceptionExample from "Introduction to Exceptions" --
// same idea (dividing by zero), but this time wrapped in a try-catch block.
// The exception is still thrown, still created with the same class and
// message -- the only difference is that something now intercepts it BEFORE
// it reaches main() uncaught, so the program keeps running instead of
// terminating.
public class BasicTryCatchExample {
    public static void main(String[] args) {
        System.out.println("About to divide...");
        int result = safeDivide(10, 0);
        // Unlike UncaughtExceptionExample, this line DOES run -- the
        // exception was handled, so execution continues normally after the
        // try-catch block.
        System.out.println("Result: " + result);
        System.out.println("Program continues normally.");
    }

    private static int safeDivide(int a, int b) {
        try {
            return a / b;   // throws ArithmeticException when b == 0
        } catch (ArithmeticException e) {
            // "e" is the SAME kind of exception object described in
            // "Introduction to Exceptions" -- it has a message, a class, and
            // a stack trace. Here we just read its message.
            System.out.println("Caught it: " + e.getMessage());
            return 0;   // a sensible fallback value
        }
    }
}
```

> Bir `catch` bloğundaki parametre (yukarıdaki `e`) gerçek, sıradan bir yerel değişkendir, yalnızca o `catch` bloğuna scope'lanmıştır -- üzerinde `Throwable`'ın tanımladığı herhangi bir metodu çağırabilirsin ("Exception'lara Giriş" dersinin "Bir Exception'ın Anatomisi: Throwable, Mesaj ve Stack Trace" bölümüne bakınız), en yaygın olarak `getMessage()`.

Birden Fazla catch Bloğu: Sırayla Eşleşme

Tek bir `try`'ı, her biri FARKLI bir exception tipini ele alan birkaç `catch` bloğu izleyebilir -- Java bunları yukarıdan aşağıya kontrol eder ve yalnızca eşleşen İLKİNİ çalıştırır.

```
// A single try block can be followed by SEVERAL catch blocks -- Java checks
// them TOP TO BOTTOM and runs the FIRST one whose type matches the thrown
// exception, then skips every catch block after it. Only ONE catch block
// ever runs per exception, never more than one.
public class MultipleCatchBlocksExample {
    public static void main(String[] args) {
        parseAndDivide("100", "0");     // triggers the ArithmeticException catch
        parseAndDivide("abc", "5");     // triggers the NumberFormatException catch
        parseAndDivide("100", "5");     // triggers neither -- prints the result
    }

    private static void parseAndDivide(String numeratorText, String denominatorText) {
        try {
            int numerator = Integer.parseInt(numeratorText);
            int denominator = Integer.parseInt(denominatorText);
            System.out.println(numeratorText + " / " + denominatorText + " = " + (numerator
/ denominator));
        } catch (ArithmeticException e) {
            // Matches ONLY division-related failures (denominator == 0).
            System.out.println("Cannot divide: " + e.getMessage());
        } catch (NumberFormatException e) {
            // Matches ONLY invalid number text -- NOTE: this catch block
            // must come AFTER any catch for a SUPERCLASS of
            // NumberFormatException, or the compiler rejects it as
            // unreachable (see "Common Mistakes").
            System.out.println("Not a valid number: " + e.getMessage());
        }
    }
}
```

> Sıra önemli. Bir SÜPER SINIF için `catch` bloğu (`RuntimeException` gibi), alt sınıflarından biri için `catch` bloğundan (`NumberFormatException` gibi) ÖNCE gelseydi, alt sınıfın `catch` bloğu erişilemez olurdu -- süper sınıfınki her zaman önce eşleşirdi. Derleyici bu spesifik hatayı yakalar ve build'i reddeder ("süper sınıf"ın burada tam olarak ne anlama geldiği için "Exception Hiyerarşisi" dersine bakınız).

Multi-Catch: | ile Birden Fazla Exception Tipini Aynı Blokta Yakalamak

İki ya da daha fazla FARKLI exception tipi gerçekten AYNI handling koduna ihtiyaç duyduğunda, ayrı, yinelenen `catch` blokları yazmak (`MultipleCatchBlocksExample`'da olduğu gibi) hiçbir sebep olmadan kendini tekrar eder. Java 7'de eklenen multi-catch, tek bir `catch` bloğunun `|` ile ayrılmış birkaç tipi listelemesine izin verir.

```
// Multi-catch (Java 7+): when two or more DIFFERENT exception types need the
// EXACT SAME handling code, "|" lets one catch block handle all of them,
// instead of duplicating the same body in separate catch blocks (compare to
// MultipleCatchBlocksExample, where the two blocks needed genuinely
// DIFFERENT handling).
public class MultiCatchExample {
    public static void main(String[] args) {
        parseAndDivide("100", "0");     // ArithmeticException -- caught below
        parseAndDivide("abc", "5");     // NumberFormatException -- caught below, SAME
handling
        parseAndDivide("100", "5");     // neither -- prints the result
    }

    private static void parseAndDivide(String numeratorText, String denominatorText) {
        try {
            int numerator = Integer.parseInt(numeratorText);
            int denominator = Integer.parseInt(denominatorText);
            System.out.println(numeratorText + " / " + denominatorText + " = " + (numerator
/ denominator));
        } catch (ArithmeticException | NumberFormatException e) {
            // Both exception types land here -- "e" is typed as their
            // nearest common supertype for anything they share (in this
            // case, RuntimeException -- see "Exception Hierarchy" for what
            // that means precisely). The two types in a multi-catch can
            // NEVER be one a subclass of the other -- the compiler rejects
            // that as redundant.
            System.out.println("Could not complete the calculation: " + e.getMessage());
        }
    }
}
```

finally: Her Zaman Çalışan Blok

Son `catch`'ten sonra (ya da hiç `catch` olmadan, doğrudan `try`'dan sonra) yerleştirilen bir `finally` bloğu, her durumda çalışır -- `try` bloğu başarılı olsun, bir exception yakalansın, ya da bir exception her `catch` bloğunu yakalanmadan geçip gitsin.

```
// finally runs in EVERY case -- whether the try block succeeds, throws an
// exception that gets caught, or (as the last call below shows) throws an
// exception that DOESN'T get caught at all. This is what makes finally the
// right place for cleanup code that must run no matter what happened.
public class FinallyAlwaysRunsExample {
    public static void main(String[] args) {
        System.out.println("--- Case 1: no exception ---");
        process(10, 2);

        System.out.println("--- Case 2: caught exception ---");
        process(10, 0);

        System.out.println("--- Case 3: uncaught exception ---");
        processWithoutCatch(10, 0);   // finally STILL runs, then the
                                      // exception propagates anyway (see
                                      // "Introduction to Exceptions")
    }

    private static void process(int a, int b) {
        try {
            System.out.println("Result: " + (a / b));
        } catch (ArithmeticException e) {
            System.out.println("Caught: " + e.getMessage());
        } finally {
            System.out.println("finally: cleanup runs either way");
        }
    }

    private static void processWithoutCatch(int a, int b) {
        try {
            System.out.println("Result: " + (a / b));
        } finally {
            // try WITHOUT a catch, paired only with finally, is completely
            // legal -- finally still runs, then the exception (having no
            // catch to stop it) continues propagating exactly as if there
            // were no try block at all.
            System.out.println("finally: cleanup runs even without a catch block");
        }
    }
}
```

> `finally`, bir dosyayı ya da bir veritabanı bağlantısını kapatmak gibi bir resource-cleanup deseninin üzerine inşa edildiği şey -- `try`-with-resources (gerçekten kullanıldığı yerde, "File Reading" dersinde kapsanıyor) aslında derleyicinin senin için otomatik olarak yazdığı `finally`-tabanlı bir cleanup'tır, temelde farklı bir mekanizma değil.

finally ve return'ün Etkileşimi: İnce Bir Ayrıntı

`finally`'nin koşulsuz çalışmasının gerçekten şaşırtıcı bir sonucu var: `finally`'nin KENDİSİ bir `return` (ya da bir `throw`) içeriyorsa, `try` ya da `catch` bloğunun döndürmek üzere olduğu her şeyi sessizce EZER, tamamen atarak -- zaten yayılmakta olan gerçekten yakalanmamış bir exception dahil.

```
// A genuinely surprising, real Java behavior -- and exactly why "Best
// Practices" recommends never putting a return (or another throw) inside a
// finally block. If finally itself returns a value, that value SILENTLY
// REPLACES whatever the try or catch block was about to return -- and if
// finally throws, it SILENTLY REPLACES whatever exception was already
// propagating, discarding it completely.
public class FinallyOverridingReturnExample {
    public static void main(String[] args) {
        System.out.println("brokenDivide(10, 0) returned: " + brokenDivide(10, 0));
        // Prints -1, NOT the ArithmeticException you might expect --
        // finally's "return -1" swallowed the exception entirely. The
        // catch block never even got a chance to run its own return.
    }

    private static int brokenDivide(int a, int b) {
        try {
            return a / b;              // throws ArithmeticException when b == 0
        } catch (ArithmeticException e) {
            System.out.println("Caught: " + e.getMessage());
            return -1;
        } finally {
            // This return DISCARDS the catch block's "return -1" (in this
            // case, coincidentally the same value) -- but more importantly,
            // it would ALSO discard a genuinely uncaught, unrelated
            // exception from anywhere in the try/catch, silently, with no
            // trace of it ever happening. That's the real danger, not the
            // duplicate value here.
            return -1;
        }
    }
}
```

> Bu, Java'nın kazara izin verdiği bir bug ya da kenar durum DEĞİL -- yalnızca `finally`'nin koşulsuz çalışmasının `return` / `throw` ile nasıl etkileşime girdiği. Bu tam olarak neden "Best Practices"in `finally` içinde bir `return` ya da `throw`'u pratik bir kısayol DEĞİL, kaçınılması gereken bir şey olarak ele almanı önerdiği.

Best Practices

- **`try` bloklarını gerçekten başarısız olabilecek spesifik koda odaklı tut** -- gerekenden çok daha fazla kodu sarmak, bir exception'ın gerçekçi olarak hangi satırdan gelebileceğini söylemeyi zorlaştırır.

- **Birden fazla `catch` bloğunu en spesifikten en genele sırala** -- "Birden Fazla catch Bloğu"ndaki uyarıya, ve hangi tipin daha genel olduğunu nasıl akıl yürüteceğin için "Exception Hiyerarşisi" dersine bakınız.
- **Özdeş `catch` gövdelerini yinelemek yerine multi-catch'i tercih et** -- `MultiCatchExample`'a bakınız -- ama yalnızca handling mantığı gerçekten özdeşse, sadece benzer değil.
- **Bir `finally` bloğunun İÇİNE asla bir `return` ya da `throw` koyma** -- "finally ve return'ün Etkileşimi: İnce Bir Ayrıntı" bölümüne bakınız -- `finally`'i yalnızca neyin döndürüldüğünü ya da fırlatıldığını etkilemeyen cleanup için kullan.

Yaygın Hatalar

- **Bir süper sınıfın `catch` bloğunu bir alt sınıfınkinden ÖNCE sıralamak.** Derleyici bunu erişilemez kod olarak doğrudan reddeder -- "Birden Fazla catch Bloğu: Sırayla Eşleşme" bölümündeki uyarıya bakınız.
- **`finally`'nin bir `try` bloğu return ettiğinde çalışmadığını varsaymak.** Hâlâ çalışır, `return` ifadesi değerlendirildikten ile kontrolün gerçekten metottan çıkması arasında -- `FinallyAlwaysRunsExample`'a bakınız.
- **`finally` içindeki bir `return`'ün devam eden bir exception'ı sessizce attığını fark etmemek.** Hiçbir uyarı, orijinal sorunun hiçbir izi yok -- `FinallyOverridingReturnExample` 'a bakınız.
- **Bir metodun tüm gövdesini "her ihtimale karşı" tek bir dev `try` bloğuna sarmak.** Bu, sonradan bir exception'ın gerçekte hangi spesifik satıra karşı koruma sağladığını söylemeyi çok zorlaştırır.

Özet, Cheat Sheet ve Terimler Sözlüğü

`try`, başarısız olabilecek kodu işaretler; `catch`, bir eşleşme bulunana kadar yukarıdan aşağıya kontrol edilerek, gerçekleşirse belirli bir exception tipi için ne yapılacağını belirtir; multi-catch (`|`), tek bir `catch` bloğunun birbiriyle ilgisiz birkaç tipi özdeş handling ile ele almasına izin verir; `finally` her durumda koşulsuz olarak çalışır, bu da onu cleanup için doğru yer yapar -- ama `return` ya da `throw` için ASLA, çünkü bu, `try`/`catch`'in üretmek üzere olduğu her şeyi sessizce ezer.

Hızlı referans:

```
try {
    riskyOperation();
} catch (SpecificException e) {
    // spesifik durumu handle et
} catch (AnotherException | YetAnotherException e) {
    // multi-catch: iki ilgisiz tip için özdeş handling
} finally {
    // her zaman çalışır -- yalnızca cleanup, burada asla return/throw yok
}
```

Terimler Sözlüğü

try — Bir exception fırlatabilecek kodu işaretleyen bir blok.

catch — Önceki bir `try` bloğunun içinde fırlatılan belirli bir exception tipinin nasıl handle edileceğini belirten bir blok.

Multi-Catch — `|` kullanarak, birbiriyle ilgisiz birkaç exception tipini özdeş şekilde handle eden tek bir `catch` bloğu.

finally — Bir exception oluşmuş ya da yakalanmış olsun olmasın, `try` / `catch`'ten sonra koşulsuz olarak çalışan bir blok.