Exception Handling Best Practices

Önceki altı dersin mekaniğinin üzerine kurulan pratikler -- exception'ları kontrol akışı için kullanma ve yutma anti-pattern'leri, catch bloklarını spesifikten genele doğru doğru sırada yakalamak, ve yalnızca gerçekten anlamlı bir tepki verilebilecek yerde yakalamak. Exception Handling serisinin 7. ve son dersi, tüm seriyi özetleyen bir kapanış bölümüyle.

Başlangıç 20 dk

Son güncelleme: 10 Kasım 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/exception-handling-best-practices

---

Altı ders sonunda, Java exception'larının mekaniğini iyice biliyorsun: nasıl yapılandırıldıkları, nasıl yakalanıp sonlandırıldıkları, hiyerarşinin nasıl işlediği, checked ve unchecked farkı, `throw` / `throws`, ve kendi türlerini tasarlamak. Mekaniği bilmek onları İYİ KULLANMAKLA aynı şey değil — bu kapanış dersi, yalnızca derlenen koddan bir ekibin production'da gerçekten güvenebileceği koda geçişi sağlayan alışkanlıklar ve anti-pattern'ler hakkında.

Bu Ders Neden Var?

Şimdiye kadar işlenen her mekanik geçerli Java'dır — derleyici hepsini kabul eder. Ama geçerli olmak akıllıca olmakla aynı şey değildir: boş bir `catch` bloğu sorunsuz derlenir ve production hatalarını debug etme yeteneğini sessizce yok eder; `Exception`'ı yakalamak sorunsuz derlenir ve gerçek hataları diğer her şeyle birlikte sessizce yutar. Bu ders, deneyimli Java geliştiricilerin zaten bildiğin kurallar üzerine uyguladığı pratikleri bir araya getiriyor.

Anti-Pattern: Exception'ları Kontrol Akışı İçin Kullanmak

Exception'lar İSTİSNAİ durumlar içindir — bir metodun normal, beklenen çalışmasının parçası olmayan sonuçlar. `throw` / `catch`'i sıradan bir kontrol akışını (bir şey bulunca döngüden çıkmak gibi) uygulamak için kullanmak, zaten `return` ya da `break`'in yaptığı işi, hata yayılması için tasarlanmış bir mekanizmayı kötüye kullanarak yapmaktır — gerçek bir çalışma zamanı maliyetiyle (bir stack trace oluşturmak bedava değildir) ve gerçek bir okunabilirlik maliyetiyle.

```
public class ExceptionsForControlFlowAntiPatternExample {

    public static void main(String[] args) {
        int[] numbers = {4, 8, 15, 16, 23, 42};

        System.out.println("bad (exception as control flow): " + findFirstOver20_bad
(numbers));
        System.out.println("good (ordinary control flow):    " + findFirstOver20_good
(numbers));
    }

    // BAD: throwing and catching an exception purely to break out of a loop
    // once a value is found. Nothing here is actually exceptional -- "found
    // a matching number" is a completely normal, expected outcome. Building
    // and unwinding a stack trace for it is wasted work and hides the real
    // control flow behind a try/catch that has nothing to do with error
    // handling.
    static int findFirstOver20_bad(int[] numbers) {
        try {
            for (int n : numbers) {
                if (n > 20) {
                    throw new FoundException(n);
                }
            }
        } catch (FoundException e) {
            return e.value;
        }
        return -1;
    }

    static class FoundException extends RuntimeException {
        final int value;
        FoundException(int value) {
            this.value = value;
        }
    }

    // GOOD: the same result using ordinary control flow -- no exception
    // involved at all, because nothing exceptional happened.
    static int findFirstOver20_good(int[] numbers) {
        for (int n : numbers) {
            if (n > 20) {
                return n;
            }
        }
        return -1;
    }
}
```

Beklenen Çıktı

```
bad (exception as control flow): 23
good (ordinary control flow):    23
```

`findFirstOver20_bad(...)`, yalnızca bir döngüden çıkmak için `FoundException`'ı fırlatıp yakalıyor — eşleşen bir sayı bulmak tamamen sıradan bir sonuçtur, istisnai değil.

`findFirstOver20_good(...)`, hiçbir exception mekanizması olmadan, düz bir `return` ile aynı sonucu üretiyor.

> Tek amacı bir döngüden ya da derin iç içe bir çağrıdan bir değer taşımak olan custom bir exception yazdığını fark edersen, bu exception'ları kontrol akışı için kullandığının güçlü bir işaretidir — kodu bunun yerine sıradan `return` değerleriyle yeniden yapılandır.

Anti-Pattern: Exception'ları Yutmak

Boş bir `catch` bloğu, Java'da yazabileceğin en zararlı şeylerden biridir: hata gerçekleşti, ama onun her izi — türü, mesajı, stack trace'i — çalışma o boş bloğa ulaştığı anda kayboluyor. Ortaya çıkan belirtiyi sonradan debug eden kişinin elinde hiçbir ipucu kalmaz.

```
public class SwallowingExceptionsAntiPatternExample {

    public static void main(String[] args) {
        System.out.println("bad result:  " + parsePort_bad("not-a-number"));
        try {
            parsePort_good("not-a-number");
        } catch (IllegalArgumentException e) {
            System.out.println("good result: rejected with a clear reason -- " + e.
getMessage());
        }
    }

    // BAD: an empty catch block swallows the exception entirely -- the
    // failure leaves no trace anywhere, and the caller gets a silently
    // wrong value (0) with no way to tell it apart from a genuinely
    // configured port 0.
    static int parsePort_bad(String text) {
        int port = 0;
        try {
            port = Integer.parseInt(text);
        } catch (NumberFormatException e) {
            // swallowed -- nothing here at all
        }
        return port;
    }

    // GOOD: if there's truly nothing useful this method can do about the
    // failure, don't catch it at all here -- let it propagate, or catch it
    // only to translate it into a clearer, more specific failure for this
    // method's own caller.
    static int parsePort_good(String text) {
        try {
            return Integer.parseInt(text);
        } catch (NumberFormatException e) {
            throw new IllegalArgumentException("'" + text + "' is not a valid port number",
e);
        }
    }
}
```

`parsePort_bad(...)`, `NumberFormatException`'ı yutuyor ve `0` döndürüyor — bir çağıran, gerçekten yapılandırılmış bir `0` portu ile sessizce atılmış bir ayrıştırma hatasını birbirinden ayırt edemez. `parsePort_good(...)` iki dürüst şeyden birini yapıyor: burada gerçekten yapılabilecek yararlı bir şey yoksa hiç yakalama (yayılsın); YAKALIYORSAN, hatayı kendi çağıranın için daha net bir şeye çevir — "Throw ve Throws"tan zaten bildiğin bir teknik, wrapping.

> Doğru ele almayı sonra bulacağım diye "geçici olarak" boş bir `catch` bloğu yazmak istersen, en azından içine bir yer tutucu olarak bir şey yazdır ya da logla — boş bir blok sonsuza kadar unutulması kolay bir şeydir.

Spesifik Yakalamak ve Doğru Sırada

Tek bir `try`'ın birden fazla `catch` bloğu olabilir, ve — "Try-Catch ve Finally"de gördüğün gibi — Java bunların en spesifikten en genele doğru sıralanmasını zorunlu kılar. Bu derleyici kuralı, gerçek bir tasarım hedefine hizmet eder: her `catch` bloğu, onu içeren geniş bir kategoriye değil, tam olarak adlandırdığı hataya tepki vermelidir.

```
public class CatchOrderAndSpecificityExample {

    public static void main(String[] args) {
        process("42");
        process("not-a-number");
        process(null);
    }

    static void process(String text) {
        try {
            System.out.println("length: " + text.length() + ", parsed: " + Integer.parseInt
(text));
        } catch (NumberFormatException e) {
            // Specific: this failure has one clear, targeted response.
            System.out.println("'" + text + "' is not a number");
        } catch (NullPointerException e) {
            // Also specific: a completely different failure, a different
            // targeted response -- listing it separately (instead of
            // folding it into a broad catch below) keeps each response
            // honest about what it's actually reacting to.
            System.out.println("no input was provided");
        } catch (RuntimeException e) {
            // A broad catch-all, placed LAST (the compiler would reject
            // it above the more specific catches, since it's a
            // supertype of both). Reserved for genuinely unanticipated
            // failures -- not a substitute for handling the two known
            // ones above.
            System.out.println("unexpected failure: " + e);
        }
    }
}
```

Beklenen Çıktı

```
length: 2, parsed: 42
'not-a-number' is not a number
no input was provided
```

`process(...)`, `NumberFormatException` ve `NullPointerException`'ı ayrı ayrı yakalıyor, her biri kendi hedeflenmiş tepkisiyle, ve yalnızca gerçekten beklenmeyen hatalar için geniş bir `catch (RuntimeException e)`'e geri düşüyor — en sona konmuş, çünkü derleyici bir süper tip yakalamasını alt tiplerinden önce konumlandırmayı reddeder. `Exception`'ı (ya da daha kötüsü `Throwable`'ı) bilinçli bir son çare olarak değil de alışkanlıktan yakalamak, tamamen farklı tepkiler gerektiren hataları tek bir kategoriye tıkıştırma eğilimindedir.

Yalnızca Gerçekten Ele Alabildiğini Yakala

Bir exception'ı YAKALAYABİLEN her metot bunu YAPMALI demek değildir. Bir `catch` bloğu ancak metodun hatayla ilgili anlamlı bir şey yapacak yeterli bağlama sahip olduğunda yerini hak eder — yeniden dene, bir varsayılana geri dön, kendi çağıranı için çevir. Metodun böyle bir tepkisi yoksa, orada yakalayıp hemen hiçbir yararlı şey yapmamak (ya da değiştirmeden yeniden fırlatmak) değer katmadan kod ekler.

```
public class OnlyCatchWhatYouCanHandleExample {

    public static void main(String[] args) {
        // The top of the call chain is where a retry decision actually
        // makes sense -- it's the only place with enough context (should
        // we ask the user again? give up? fall back to a default?) to
        // decide what "handling" the failure even means.
        try {
            System.out.println("price: " + readPriceWithRetry());
        } catch (IllegalStateException e) {
            System.out.println("giving up: " + e.getMessage());
        }
    }

    static double readPriceWithRetry() {
        int attempts = 0;
        while (attempts < 3) {
            attempts++;
            try {
                return parsePrice(attempts < 3 ? "bad-input" : "19.99");
            } catch (NumberFormatException e) {
                // This layer CAN meaningfully handle the failure -- it
                // knows how to retry, so it catches, reacts, and moves on
                // instead of letting the exception propagate needlessly.
                System.out.println("attempt " + attempts + " failed, retrying...");
            }
        }
        throw new IllegalStateException("could not read a valid price after 3 attempts");
    }

    static double parsePrice(String text) {
        // This method has no useful response to a malformed price -- it
        // isn't the one deciding whether to retry or give up, so it does
        // NOT catch NumberFormatException here. Catching it just to
        // immediately rethrow the same thing would add a try/catch that
        // does nothing.
        return Double.parseDouble(text);
    }
}
```

`parsePrice(...)`, bilinçli olarak `NumberFormatException`'ı YAKALAMIYOR — yeniden mi deneneceğine yoksa vazgeçilip vazgeçilmeyeceğine karar verecek bir dayanağı yok, bu yüzden exception'ın yayılmasına izin veriyor. `readPriceWithRetry(...)`, yakalamanın gerçekten ait olduğu yer, çünkü nasıl tepki vereceğini bilen katman burası: birkaç kez yeniden dene, sonra kendisini çağıran için net bir `IllegalStateException`'la vazgeç.

Hepsini Bir Araya Getirmek: Serinin Özeti

Bu seri boyunca, tam resmi katman katman inşa ettin: "Exception'lara Giriş" bir exception'ın ne olduğunu ve yakalanmadığında ne olduğunu işledi; "Try-Catch ve Finally" ele alma ve temizleme mekaniğini işledi; "Exception Hiyerarşisi", `Throwable`, `Error`, `Exception` ve `RuntimeException`'ın nasıl ilişkilendiğini ve `catch`'in süper tip üzerinden nasıl eşleştiğini

işledi; "Checked ve Unchecked Exception'lar" derleyici tarafından zorlanan sözleşmeyi ve hangisinin ne zaman seçileceğini işledi; "Throw ve Throws" çalışma zamanında bir hata yaratmakla derleme zamanında onu bildirmek arasındaki farkı işledi; "Özel (Custom) Exception Oluşturmak ve Fırlatmak" kendi türlerini tasarlamayı işledi. Bu dersin dört pratiği, tam da bu araç kutusunu bir ekibin güvenebileceği koda dönüştüren şeydir: sıradan kontrol akışı yeterliyken `throw`/`catch`'e başvurma, bir `catch` bloğunun ters giden şeyin kanıtını asla silmesine izin verme, spesifik türleri doğru sırada yakala, ve yalnızca gerçekten tepki verebildiğin yerde yakala.

Best Practices

- Exception'ları gerçekten istisnai durumlar için ayır — kodunun doğrudan kontrol edebileceği beklenen, sıradan sonuçlar için asla.
- Bir `catch` bloğunu asla gerçekten boş bırakma; en azından hatanın gerçekleştiğini not et.
- `catch` bloklarını en spesifikten en genele doğru listele, ve geniş bir `catch (Exception e)`'i alışkanlık değil bilinçli bir son çare olarak ele al.
- Yalnızca metodun hataya gerçek, yararlı bir tepkisi olduğu yerde yakala — aksi hâlde, bunu yapabilecek bir katmana yayılmasına izin ver.
- Farklı bir türle yakalayıp yeniden fırlattığında, orijinalini her zaman "Throw ve Throws"ta işlendiği gibi `cause` olarak koru.

Yaygın Hatalar

- Yalnızca bir döngüden ya da iç içe bir çağrıdan çıkmak için tam bir custom exception ("bulundu" sinyali gibi) tasarlamak.
- `catch (Exception e) {}`'i "geçici olarak" yazıp bir daha düzeltmeye hiç dönmemek.
- Daha dar bir tip henüz değerlendirilmeden önce geniş bir süper tipi yakalayıp, gerçekten farklı hataları tek bir genel tepkiye sıkıştırmak.
- Bir exception'ı yalnızca loglamak ve tam olarak aynısını değiştirmeden yeniden fırlatmak için yakalamak — bu, çağıranın zaten kendi başına yapamayacağı hiçbir şey katmayan bir `try`/`catch` ekler.

Özet, Cheat Sheet ve Terimler Sözlüğü

Özet

- Exception'lar istisnai durumlar içindir, sıradan kontrol akışını uygulamak için değil.
- Boş bir `catch` bloğu bir hatanın kanıtını yok eder — asla sessizce boş bırakma.
- `catch` blokları spesifik olmalı ve en spesifikten en genele sıralanmalı; geniş bir yakalama bilinçli bir son çaredir.
- Yalnızca bir metodun hataya gerçekten tepki verebildiği yerde yakala — aksi hâlde yayılmasına izin ver.

- Tüm seri tutarlı tek bir araç kutusu inşa eder: exception'ların ne olduğu, ele alma ve temizleme, hiyerarşi, checked vs unchecked, throw vs throws, ve kendi türlerini tasarlamak.

Cheat Sheet

```
// Yapma: kontrol akışı olarak exception
try {
    for (int n : numbers) {
        if (n > 20) throw new FoundException(n);
    }
} catch (FoundException e) { return e.value; }

// Yap: sıradan kontrol akışı
for (int n : numbers) {
    if (n > 20) return n;
}

// Yapma: yutmak
try {
    return Integer.parseInt(text);
} catch (NumberFormatException e) { /* hiçbir şey */ }

// Yap: çevir, silme
try {
    return Integer.parseInt(text);
} catch (NumberFormatException e) {
    throw new IllegalArgumentException("invalid: " + text, e);
}

// Yap: spesifik önce, geniş sonra
catch (NumberFormatException e) { ... }
catch (NullPointerException e) { ... }
catch (RuntimeException e) { ... } // son çare
```

Terimler Sözlüğü

- **Kontrol akışı olarak exception**: hata yayılması yerine sıradan, beklenen mantığı uygulamak için `throw` / `catch` kullanan bir anti-pattern.
- **Exception'ı yutmak**: onu yakalayıp genelde boş bir `catch` bloğuyla tüm izini yok etmek.
- **Yakalama spesifikliği**: `catch` bloklarını en dar, en hedeflenmiş türden en genele doğru sıralamak.
- **Ele alınabilir (handleable)**: belirli bir metodun yalnızca geçirmek yerine gerçekten tepki verecek (yeniden dene, varsayılana dön, çevir) yeterli bağlama sahip olduğu bir hata.