Exception'lara Giriş

Bir exception'ın gerçekte ne olduğu -- neden var olduğu (dönüş-değeri-tabanlı hata sinyallemesinin sorunları), Java'nın checked exception tasarım kararının kısa tarihçesi, bir exception nesnesinin taşıdığı mesaj/cause/stack trace, yakalanmamış bir exception'ın programı nasıl sonlandırdığı, stack trace okumak ve propagation, ve yaygın runtime exception tetikleyicileri (NullPointerException, ArrayIndexOutOfBoundsException, NumberFormatException). Bu serinin 7 dersinin ilki, henüz try/catch mekaniği yok.

Başlangıç 20 dk

Son güncelleme: 5 Ekim 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/introduction-to-exceptions

---

Bu kursta şimdiye kadar yazdığın her program, işlerin yolunda gideceğini varsaydı -- bir dizi indeksinin geçerli olduğunu, bir string'in gerçekten bir sayı tuttuğunu, bir değerin kullanıldığı anda asla `null` olmadığını. Gerçek programlar bu varsayımı yapamaz. Bu, Java'nın bir şeyler ters gittiğinde ne yaptığını anlatan yedi dersin ilki -- en temel soruyla başlıyoruz: bir exception aslında NEDİR?

Exception Nedir?

Bir exception, bir program çalışırken beklenmedik bir şeyin olduğunu temsil eden bir nesnedir -- gerçek bir Java nesnesi, bir sınıfın örneği. Kod, üzerinden basitçe devam edemeyeceği bir sorunla karşılaştığında (sıfıra bölme, bir dizinin sonunun ötesini okuma, `null` olan bir referans üzerinde bir metot çağırma), Java Virtual Machine tam olarak neyin yanlış gittiğini tarif eden bir exception nesnesi oluşturur, ve onu FIRLATIR (throw) -- normal çalışmayı hemen durduran ve sorunla ilgilenmeye istekli bir şey arayan özel bir tür sıçrama.

Neden Var?

Exception'lar bir dil özelliği olarak var olmadan önce (C gibi eski dillerde eşdeğeri yoktu), başarısız olabilecek bir fonksiyon bu başarısızlığı DÖNÜŞ DEĞERİ üzerinden sinyallemek zorundaydı -- `-1` gibi özel bir sayı, ya da çağıranların HER SEFERİNDE kontrol etmeyi hatırlaması gereken bir out-parameter. Kontrol etmeyi unutmak kolaydı, sessizdi, ve gerçek hataların yaygın bir kaynağıydı. Exception'lar BAŞARISIZLIK YOLUNU BAŞARI YOLUNDAN tamamen ayırır: bir metodun dönüş değeri yalnızca başarıyı temsil etmek zorundadır, ve bir

başarısızlık, kontrol edilmemiş bir dönüş değerinin olabileceği gibi asla sessizce görmezden gelinemez -- yakalanmamış bir exception SESSİZ değil, GÜRÜLTÜLÜDÜR ("Bir Exception Yakalanmazsa Ne Olur?" bölümüne bakınız).

Tarihçe

Yapılandırılmış bir dil özelliği olarak exception handling, Java'dan onlarca yıl önceye dayanır -- PL/I (1964) ve daha sonra CLU ve Ada bununla denemeler yaptı, C++ ise 1980'lerin sonunda `try` / `catch` / `throw`'u ekledi. 1990'ların ortasında tasarlanan Java, bunu C++'ın gittiğinden daha ileri götüren, tartışmalı bir tasarım kararıyla genişletti: derleyicinin kabul etmeni ZORUNLU kıldığı bir kategori olan CHECKED exception'lar (tam kapsamı "Checked ve Unchecked Exception'lar" dersinde) -- Java'dan önce hiçbir ana akım dilin yapmadığı bir tercih, ve C# ile Kotlin gibi sonraki dillerin bilinçli olarak geri aldığı bir tercih. Bu tarihi, checked exception'lar derinlemesine işlenmeden ÖNCE, şimdiden bilmekte fayda var, çünkü Java'nın exception sisteminin neden böyle göründüğünü açıklıyor.

Bir Exception'ın Anatomisi: Throwable, Mesaj ve Stack Trace

Her exception nesnesi -- spesifik sınıfından bağımsız olarak -- aynı üç bilgi için yer barındırır. Bir MESAJ: ne yanlış gittiğini anlatan, genellikle exception oluşturulurken ayarlanan, insan tarafından okunabilir bir string -- ama garanti değildir: mesaj verilmeden oluşturulan bir exception'da `getMessage()` `null` döndürür. Bir CAUSE (neden): bunu TETİKLEYEN başka bir exception'a opsiyonel bir referans (ne zaman ve neden kullanılacağı için "Exception Handling Best Practices" dersine bakınız). Ve bir STACK TRACE: exception nesnesinin oluşturulduğu anda tam olarak hangi metotların, hangi sırayla aktif olduğunun otomatik bir anlık görüntüsü.

```
// Every exception object carries three things worth knowing about before we
// ever write a single try/catch (that's the next lesson): a MESSAGE (a
// human-readable description), a CAUSE (an optional reference to another
// Throwable that triggered this one -- covered fully in "Exception Handling
// Best Practices"), and a STACK TRACE (a snapshot of exactly which method
// calls were active when the exception was created).
//
// We haven't covered try/catch yet, so this example uses a completely
// different, genuinely real mechanism to inspect an exception object:
// Thread.setDefaultUncaughtExceptionHandler. The JVM calls this handler with
// the exact Throwable it's about to report, right before terminating the
// program -- a legitimate way to EXAMINE an exception's anatomy without
// handling it.
public class ExceptionAnatomyExample {
    public static void main(String[] args) {
        Thread.setDefaultUncaughtExceptionHandler((thread, exception) -> {
            System.out.println("Caught by the handler, not by our own code:");
            System.out.println("Class: " + exception.getClass().getName());
            System.out.println("Message: " + exception.getMessage());
            System.out.println("First stack trace line: " + exception.getStackTrace()[0]);
        });

        int[] scores = {90, 85, 78};
        // Deliberately out of bounds -- this creates an
        // ArrayIndexOutOfBoundsException object, throws it, nothing catches
        // it, and the handler above receives it right before the JVM exits.
        System.out.println(scores[5]);
    }
}
```

> Java'daki her exception sınıfı (tam resim için "Exception Hiyerarşisi" dersine bakınız) nihayetinde `Throwable`'ı extend eder -- `getMessage()`, `getCause()` ve `getStackTrace()`'in gerçekte geldiği yer burasıdır -- spesifik exception bir `ArithmeticException`, bir `NullPointerException`, ya da bu serinin ilerisinde kendi yazacağın özel bir exception olsa bile bu doğrudur.

Bir Exception Yakalanmazsa Ne Olur?

Bir exception'ı nasıl handle edeceğimizi henüz işlemedik -- bu, hemen bir sonraki ders. Önce onu HİÇ handle etmeden ne olduğunu görmek, handle etmenin değerini çok daha net kılıyor.

```
// No try/catch anywhere in this file on purpose -- this lesson is about what
// happens BEFORE you learn to handle an exception (see "Try-Catch and
// Finally" for that). When an exception is thrown and nothing along the way
// catches it, the JVM does three specific things, in order: (1) it stops
// running the current thread at the exact point the exception was thrown,
// (2) it prints the exception's class, message, and full stack trace to
// standard error, (3) it terminates that thread -- for a single-threaded
// program like this one, that means the whole program exits, with a
// non-zero exit code (run "echo $?" right after this program exits on
// Linux/macOS to see it -- conventionally 1 for an uncaught exception).
public class UncaughtExceptionExample {
    public static void main(String[] args) {
        System.out.println("About to divide...");
        int result = divide(10, 0);
        // This line NEVER runs -- the exception thrown inside divide(...)
        // aborts main() before control ever returns here.
        System.out.println("Result: " + result);
    }

    private static int divide(int a, int b) {
        return a / b;   // b == 0 here -- the JVM itself creates and throws an
                        // ArithmeticException, we never wrote "throw" ourselves
    }
}
```

> Yakalanmamış bir exception yalnızca bir mesaj yazdırıp devam etmez -- üzerinde oluştuğu THREAD'i, onu fırlatan satırda, HEMEN sonlandırır. Bu kursun şimdiye kadar yazdığı tek-thread'li programlar için bu, tüm programın tam orada durması demektir, onu başlatan her neyse (bir shell, bir build aracı, başka bir program) başarısızlığı sinyalleyen sıfırdan farklı bir exit code'la birlikte.

Stack Trace Okumak: Çağrı Zincirinde Yayılma (Propagation)

Bir exception yalnızca en üst seviyede belirmez -- sorunun gerçekte olduğu yerde, genellikle birkaç metot çağrısı derinlikte oluşturulur, ve bir şey onu handle edene ya da en tepeye ulaşana kadar, kendisini çağıran her metot boyunca, bir çerçeve (frame) seferinde yukarı doğru YAYILIR (propagate).

```
// PROPAGATION: when a method doesn't handle an exception (again, we haven't
// covered how to yet -- see "Try-Catch and Finally"), the exception moves
// UP to whichever method called it, then to whichever method called THAT
// one, and so on, until either something handles it or it reaches main()
// and the program terminates (see UncaughtExceptionExample). This is called
// "unwinding the stack" -- each frame the exception passes through gets
// added to its stack trace, in order.
//
// Run this and read the stack trace top to bottom: the FIRST line is where
// the exception was actually created (validateQuantity), and each line
// after it is one more frame it propagated through on its way back to
// main() -- processOrder, then main itself. The stack trace is a direct,
// readable record of the call chain at the moment of failure.
public class PropagationThroughCallChainExample {
    public static void main(String[] args) {
        processOrder("Keyboard", -3);
    }

    private static void processOrder(String productName, int quantity) {
        System.out.println("Processing order for: " + productName);
        validateQuantity(quantity);   // the exception thrown here has to pass
                                      // THROUGH this method to reach main()
        System.out.println("Order processed.");   // never reached
    }

    private static void validateQuantity(int quantity) {
        if (quantity <= 0) {
            // We ARE writing "throw" here -- a brief, necessary preview.
            // "Throw and Throws" covers this keyword properly; here it's
            // only a vehicle to demonstrate propagation.
            throw new IllegalArgumentException("Quantity must be positive, got: " +
quantity);
        }
    }
}
```

Exception'lar Pratikte Neden Oluşur?

Gerçek Java kodunda sürekli karşılaşacağın exception'lar egzotik değil -- küçük, tekrar eden bir günlük hata ve kenar durum kümesinden geliyorlar.

```
// A survey of exceptions that occur NATURALLY in everyday code, without
// anyone deliberately writing "throw" -- the JVM itself creates and throws
// these the moment it detects the problem. Only ONE of these can actually
// run and terminate the program per execution (nothing after an uncaught
// exception runs) -- main() calls divideByZero() below; comment it out and
// uncomment a different call to see each one in turn.
public class CommonExceptionTriggersExample {
    public static void main(String[] args) {
        divideByZero();
        // accessInvalidArrayIndex();
        // dereferenceNull();
        // parseInvalidNumber();
        // castToWrongType();
    }

    private static void divideByZero() {
        int result = 10 / 0;   // ArithmeticException: "/ by zero"
        System.out.println(result);
    }

    private static void accessInvalidArrayIndex() {
        int[] values = {1, 2, 3};
        System.out.println(values[3]);   // ArrayIndexOutOfBoundsException --
                                         // valid indices here are 0, 1, 2
    }

    private static void dereferenceNull() {
        String text = null;
        System.out.println(text.length());   // NullPointerException -- the
                                              // single most common exception
                                              // in real Java code
    }

    private static void parseInvalidNumber() {
        int quantity = Integer.parseInt("not-a-number");   // NumberFormatException --
                                                           // extremely common when
                                                           // parsing user/file input
        System.out.println(quantity);
    }

    private static void castToWrongType() {
        Object value = "a String, not an Integer";
        Integer number = (Integer) value;   // ClassCastException
        System.out.println(number);
    }
}
```

Temel Terminoloji

Bu sürecin tamamını kesin olarak anlatan bir avuç kelime var, ve bu seri bunları buradan itibaren tutarlı bir şekilde kullanıyor. Bir exception'ı FIRLATMAK (throw), nesneyi oluşturmak ve JVM'e bir handler aramaya başlamasını sinyallemektir (`ExceptionAnatomyExample`'ın dizi erişimi bunu örtük olarak yapar; `PropagationThroughCallChainExample`'ın `throw new IllegalArgumentException(...)`'ı bunu açıkça yapar -- keyword'ün kendisi için "Throw ve

Throws" dersine bakınız). Bir exception'ı YAKALAMAK (catch), onu araya giren ve programın sonlanmasına izin vermek yerine ne yapılacağına karar veren kod yazmaktır (bir sonraki ders olan "Try-Catch ve Finally"ye bakınız). YAYILMA (propagation), yakalanmamış bir exception'ın onu fırlatan metottan, yukarıda gösterildiği gibi, onu çağıran her metot boyunca hareket etmesidir. Bir STACK TRACE, o yayılmanın yazdırılmış kaydıdır. Ve CHECKED vs UNCHECKED, derleyicinin olası bir exception'ı kabul etmeni zorunlu kılıp kılmadığını tarif eder -- kendi başına bir ders olacak kadar önemli bir ayrım ("Checked ve Unchecked Exception'lar").

Best Practices

- **Bir stack trace'i YUKARIDAN AŞAĞIYA oku, aşağıdan yukarıya değil** -- en üstteki satır exception'ın gerçekte oluşturulduğu yerdir, ki bu genellikle gerçek sorunun olduğu yerdir, programın sonlandığı yer değil.
- **Yakalanmamış bir exception'ı sadece bir çökme değil, bilgi olarak ele al** -- sınıf adı, mesaj ve stack trace birlikte, tek bir satır handling kodu yazmadan ÖNCE bile, neyin nerede yanlış gittiğini neredeyse her zaman tam olarak söyler.
- **Yaygın exception sınıflarını görür görmez tanımayı öğren** (`NullPointerException`, `ArrayIndexOutOfBoundsException`, `NumberFormatException`, `ArithmeticException`, `ClassCastException` -- `CommonExceptionTriggersExample`'a bakınız) -- bunlar, gerçekte karşılaşacağın exception'ların ezici çoğunluğunu oluşturur.
- **Bu ders exception'ların handle edilebileceğinden bahsetti diye henüz try /catch'e yönelme** -- bir exception'ın gerçekte NE OLDUĞUNU, ve handle edilmediğinde ne olduğunu anlamak, bir sonraki ders mekanikleri tanıtmadan önce üzerinde durmaya değer.

Yaygın Hatalar

- **Bir exception'ın her zaman KENDİ kodundaki bir hata anlamına geldiğini varsaymak.** Birçok exception, gerçekten olağandışı ama geçerli bir durumu temsil eder (henüz var olmayan bir dosya, sayı olmayan bir kullanıcı girdisi) -- Java'nın kendi tip sisteminin bu ayrımı nasıl yansıttığı için "Checked ve Unchecked Exception'lar" dersine bakınız.
- **Stack trace'i görmezden gelip yalnızca exception'ın mesajını okumak.** Mesaj tek başına, sorunun NEREDE olduğunu bulmak için genellikle yeterli değildir -- onu tam olarak belirleyen stack trace'tir.
- **Bir exception'ın mevcut metodun geri kalanını sessizce "atladığını" düşünmek.** Sessizce atlamaz -- yayılan her metotta, throw noktasından sonraki her satır gerçekten hiç çalışmaz, bu yüzden `PropagationThroughCallChainExample`'ın "Order processed." satırı hiçbir zaman yazdırılmaz.
- **Bir exception'ın fırlatılmasını, programın basitçe bir hata mesajı yazdırıp devam etmesiyle karıştırmak.** Bir handler olmadan, çalışma throw noktasının

ötesinde HİÇ devam etmez -- "Bir Exception Yakalanmazsa Ne Olur?" bölümündeki uyarıya bakınız.

Özet, Cheat Sheet ve Terimler Sözlüğü

Bir exception, çalışma sırasında beklenmedik bir şeyi temsil eden bir nesnedir -- bir sorun tespit edildiği anda JVM tarafından (ya da bu serinin ilerisinde kendi kodun tarafından) oluşturulur ve fırlatılır. Her exception, hepsi `Throwable`'dan miras alınan bir mesaj, opsiyonel bir cause, ve bir stack trace taşır. Bir handler olmadan, yakalanmamış bir exception thread'ini hemen sonlandırır, yolda kendisini çağıran her metot boyunca yukarı doğru yayılır, ve tam olarak nerede olduğunun okunabilir bir kaydı olarak bir stack trace bırakır. Sonraki altı ders doğrudan bu dört fikrin üzerine inşa ediyor -- exception'ları handle etmek, ait oldukları sınıf hiyerarşisini anlamak, checked/unchecked ayrımı, kendi exception'ını fırlatmak, ve yardımcı olan exception-handling kodunu sorunları yalnızca gizleyen koddan ayıran pratikler.

Hızlı referans:

```
// JVM bunu otomatik olarak fırlatır -- bizim yazdığımız bir "throw" yok:
int result = 10 / 0;                    // ArithmeticException: / by zero

// Kendimiz de birini fırlatabiliriz (tam kapsam "Throw ve Throws"ta):
throw new IllegalArgumentException("quantity must be positive");

// Bir Throwable'ı incelemek (try/catch işlenmeden önce, gayri resmi olarak):
exception.getMessage();                 // insan tarafından okunabilir açıklama
exception.getClass().getName();         // tam exception tipi
exception.getStackTrace();              // nerede olduğu, çerçeve çerçeve
```

Terimler Sözlüğü

Exception — Bir programın çalışması sırasında olan beklenmedik bir şeyi temsil eden bir nesne.

Throw (Fırlatmak) — Bir exception nesnesi oluşturma ve JVM'e bir handler aramaya başlamasını sinyalleme eylemi.

Uncaught Exception (Yakalanmamış Exception) — Hiçbir kodun araya girmediği, JVM'in üzerinde oluştuğu thread'i sonlandırıp stack trace'ini yazdırmasına neden olan bir exception.

Propagation (Yayılma) — Yakalanmamış bir exception'ın, onu fırlatan metottan, onu çağıran her metot boyunca yukarı doğru hareket etmesi.

Stack Trace — Bir exception oluşturulduğu anda aktif olan her metot çağrısının yazdırılmış, sıralı kaydı.