Exception Hiyerarşisi

Throwable, Error, Exception ve RuntimeException arasındaki miras ilişkisi -- her exception'ın tek bir ortak kökten geldiği, catch bloklarının bir türün herhangi bir atasını da hedefleyebildiği (polimorfik yakalama), ve instanceof ile bu hiyerarşinin çalışma zamanında nasıl sorgulanacağı. Exception Handling serisinin 3.'sü.

Başlangıç 20 dk

Son güncelleme: 9 Kasım 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/exception-hierarchy

---

"Exception'lara Giriş" konusunda bir exception'ın ne olduğunu, nasıl fırlatıldığını ve nasıl yayıldığını gördün; "Try-Catch ve Finally" konusunda onu nasıl yakalayıp temizlik yapacağını öğrendin. Ama şu ana kadar `ArithmeticException`, `NumberFormatException`, `ArrayIndexOutOfBoundsException` gibi sınıfları birbirinden bağımsız isimler gibi kullandın. Aslında hiçbiri bağımsız değil — hepsi tek bir ortak ağacın dalları, ve bu ağacın şeklini bilmek hangi `catch` bloğunun neyi yakalayacağını, ne zaman `catch (Exception e)` yazmanın tehlikeli olduğunu ve neden bazı hatalar (`StackOverflowError` gibi) hiçbir zaman normal bir `catch` bloğuyla yakalanmaması gerektiğini anlamanı sağlar.

Exception Hiyerarşisi Nedir?

Java'daki her exception ve her hata, `java.lang.Throwable` sınıfından türer. `Throwable`'ın iki doğrudan alt sınıfı vardır: `Error` ve `Exception`. `Exception`'ın da kendi alt sınıfı olan `RuntimeException` vardır. Bu üç sınıf (`Throwable`, `Error`, `Exception`) ve özellikle `RuntimeException`, şimdiye kadar gördüğün hemen hemen her exception'ın ortak atasıdır — `NumberFormatException` bir `IllegalArgumentException`'dır, o da bir `RuntimeException`'dır, o da bir `Exception`'dır, o da bir `Throwable`'dır.

Neden Var?

Bir hiyerarşi olmasaydı, her exception türü birbirinden tamamen bağımsız, ilişkisiz bir sınıf olurdu — bir metodun fırlatabileceği 10 farklı exception türünü tek tek yakalamak için 10 ayrı `catch` bloğu yazman gerekirdi, hem de aralarında hiçbir ortak davranışı (`getMessage()`, `getStackTrace()` gibi) paylaşmadan. Hiyerarşi, hem ortak bir arayüz (her `Throwable`'ın mesajı ve stack trace'i vardır) hem de **polimorfik yakalama** imkânı sağlar: bir `catch (RuntimeException e)` bloğu, `RuntimeException`'ın altındaki HERHANGİ bir alt sınıfı tek satırda yakalayabilir.

Tarihçe

Java'nın ilk sürümünden (1.0, 1996) beri `Throwable` / `Error` / `Exception` / `RuntimeException` dörtlüsü aynı temel yapıda. Tasarımcılar bilinçli olarak `Error`'ı ayrı bir dal yaptı: `Exception` uygulama kodunun tepki verebileceği durumları temsil ederken, `Error` JVM'in kendisiyle ilgili, genellikle kurtarılamaz durumları temsil eder. Bu ayrım, Java'nın "checked exception" felsefesinin de temelidir (bu, "Checked ve Unchecked Exception'lar" konusunda ayrıntılı işlenecek) — ama hiyerarşinin kendisi, checked/unchecked ayrımından bağımsız, daha temel bir sınıflandırmadır.

Throwable: Hiyerarşinin Kökü

`Throwable`, `catch` edilebilen veya `throw` edilebilen HER ŞEYİN ortak atasıdır — `Error` de, `Exception` de, `RuntimeException` de dahil. `getMessage()`, `getStackTrace()`, `printStackTrace()`, `getCause()` gibi metodların hepsi `Throwable`'da tanımlıdır, bu yüzden hangi somut exception türüyle çalışıyor olursan ol, hepsi bu metodları paylaşır.

```
// Every exception you've seen so far -- ArithmeticException, NumberFormatException,
// ArrayIndexOutOfBoundsException, IllegalArgumentException -- is a REAL class,
// and every one of those classes extends SOMETHING, all the way up to a single
// shared root: Throwable. This example walks that chain using reflection
// (getSuperclass(), see the Reflection lesson for the full API) to print it
// out directly, instead of just describing it.
public class ThrowableHierarchyWalkExample {
    public static void main(String[] args) {
        printHierarchy(new NumberFormatException("example"));
        System.out.println();
        printHierarchy(new ArithmeticException("example"));
        System.out.println();
        printHierarchy(new StackOverflowError());
    }

    private static void printHierarchy(Throwable throwable) {
        System.out.println("Hierarchy for " + throwable.getClass().getSimpleName() + ":");
        Class<?> current = throwable.getClass();
        while (current != null) {
            System.out.println("  " + current.getName());
            current = current.getSuperclass();
        }
        // Output for NumberFormatException, for example:
        //   java.lang.NumberFormatException
        //   java.lang.IllegalArgumentException
        //   java.lang.RuntimeException
        //   java.lang.Exception
        //   java.lang.Throwable
        //   java.lang.Object
    }
}
```

Beklenen Çıktı

```
Hierarchy for NumberFormatException:
  java.lang.NumberFormatException
  java.lang.IllegalArgumentException
  java.lang.RuntimeException
  java.lang.Exception
  java.lang.Throwable
  java.lang.Object

Hierarchy for ArithmeticException:
  java.lang.ArithmeticException
  java.lang.RuntimeException
  java.lang.Exception
  java.lang.Throwable
  java.lang.Object

Hierarchy for StackOverflowError:
  java.lang.StackOverflowError
  java.lang.VirtualMachineError
  java.lang.Error
  java.lang.Throwable
  java.lang.Object
```

Bu örnek, `getSuperclass()` ile (bkz. "Reflection" konusu) bir exception'ın sınıf zincirini gerçekten yukarı doğru yürüyerek yazdırıyor — `NumberFormatException`'ın zinciri `RuntimeException` ve `Exception` üzerinden `Throwable`'a çıkarken, `StackOverflowError`'ın zinciri hiç `Exception`'a uğramadan doğrudan `Error` üzerinden `Throwable`'a çıkıyor. İki dal, yalnızca en tepede birleşiyor.

Error: JVM'in Kendi Sorunları

`Error` ve alt sınıfları (`StackOverflowError`, `OutOfMemoryError` gibi), uygulama kodunun normal şartlarda önleyemeyeceği veya kurtaramayacağı, JVM seviyesindeki ciddi durumları temsil eder. `StackOverflowError`, çağrı yığınının (call stack) taştığı durumda fırlatılır — genellikle sonsuz veya çok derin bir özyineleme (recursion) yüzünden.

```
// StackOverflowError is a REAL, common Error every Java developer eventually
// sees -- not from broken hardware or a JVM bug, but from something as
// ordinary as a recursive method with no base case. It extends Error, NOT
// Exception (see ThrowableHierarchyWalkExample for what that chain actually
// looks like) -- a structural signal that this represents a problem with the
// RUNTIME ENVIRONMENT itself (the call stack ran out of space), not a
// recoverable application-level condition.
public class StackOverflowErrorExample {
    public static void main(String[] args) {
        try {
            recurseForever(0);
        } catch (StackOverflowError e) {
            // Technically legal -- Error extends Throwable, and any Throwable
            // CAN be caught. But see "Best Practices": catching an Error is
            // almost never the right response, because the JVM is often in a
            // degraded state by the time one is thrown (the stack was
            // already nearly exhausted). This catch exists here purely to
            // demonstrate that it's possible, not to recommend it.
            System.out.println("Caught: " + e.getClass().getName());
        }
    }

    private static void recurseForever(int depth) {
        // No base case, no termination condition -- every call adds another
        // frame to the call stack (see "Reading a Stack Trace: Propagation
        // Through the Call Chain" in "Introduction to Exceptions" for what a
        // stack frame is) until the JVM has nowhere left to put one.
        recurseForever(depth + 1);
    }
}
```

Beklenen Çıktı

```
Caught: java.lang.StackOverflowError
```

Bu örnek `catch (StackOverflowError e)` yazıyor ve teknik olarak çalışıyor — ama bu, "Best Practices" bölümünde göreceğin gibi neredeyse hiçbir zaman doğru yaklaşım değildir.

> `Error`'ı yakalamak dilin izin verdiği bir şeydir ama JVM bir `Error` fırlattığında çoğu zaman zaten bozulmuş bir durumdadır (örneğin çağrı yığını neredeyse tükenmiş olabilir) — bu yüzden `Error`'ı yakalayıp normal akışa devam etmeyi denemek, sorunu gizlemekten öteye geçmez.

Exception: Uygulama Seviyesi Sorunlar

`Exception`, `Error`'ın aksine, uygulama kodunun makul biçimde tepki verebileceği durumları temsil eder — kullanıcının girdiği geçersiz bir sayı, bulunamayan bir dosya, sıfıra bölme gibi. `Exception`'ın kendisi iki geniş dala ayrılır: `RuntimeException` (ve onun tüm alt sınıfları) ile `RuntimeException` OLMAYAN diğer her şey. Bu ikinci grup, derleyicinin `throws` ile bildirilmesini veya yakalanmasını zorunlu tuttuğu **checked exception**'lardır;

`RuntimeException` dalı ise **unchecked**'tir. Bu ayrımın kendisi, ne zaman hangisinin kullanılacağı ve neden ikisinin de var olduğu — "Checked ve Unchecked Exception'lar" konusunun tamamı bu soruyu ele alıyor, burada yalnızca hiyerarşideki yerini not ediyoruz.

RuntimeException'ın Alt Ağacını Yakalamak: catch ve Polimorfizm

Bir `catch` bloğu, tam olarak fırlatılan sınıfı değil, o sınıfın herhangi bir ATASINI da hedef alabilir — çünkü `NumberFormatException` bir `RuntimeException`'DIR (is-a ilişkisi). Bu, farklı somut exception türlerini TEK bir `catch` bloğuyla yakalamanı sağlar.

```
// A catch block doesn't have to name the exact exception type that was
// thrown -- it can name any TYPE IN ITS HIERARCHY, and it will still catch
// it, because a NumberFormatException genuinely IS-A RuntimeException (that's
// what "extends" means). This example throws three completely different
// concrete exception types and catches all of them with a SINGLE
// catch (RuntimeException e) block.
public class CatchingBySupertypeExample {
    public static void main(String[] args) {
        attempt(() -> Integer.parseInt("not-a-number"));
        attempt(() -> { int[] values = new int[2]; int x = values[5]; });
        attempt(() -> { int x = 10 / 0; });
    }

    private static void attempt(Runnable riskyOperation) {
        try {
            riskyOperation.run();
        } catch (RuntimeException e) {
            // Whichever concrete type was actually thrown, e.getClass()
            // reveals it -- the catch block itself only cares that it's
            // SOME RuntimeException.
            System.out.println("Caught " + e.getClass().getSimpleName()
                    + " via catch (RuntimeException e)");
        }
    }
}
```

> **Beklenen Çıktı**
>
> ```
> Caught NumberFormatException via catch (RuntimeException e)
> Caught ArrayIndexOutOfBoundsException via catch (RuntimeException e)
> Caught ArithmeticException via catch (RuntimeException e)
> ```

Bu örnekte `NumberFormatException`, `ArrayIndexOutOfBoundsException` ve `ArithmeticException` — üçü de birbirinden tamamen farklı somut sınıflar — tek bir `catch (RuntimeException e)` bloğuyla yakalanıyor, çünkü üçü de `RuntimeException`'ın bir alt sınıfı.

> Birden fazla `catch` bloğu yazarken (bkz. "Birden Fazla catch Bloğu: Sırayla Eşleşme") en ÖZEL (spesifik) türü en üste, en GENEL türü en alta koymalısın — aksi halde genel blok, ondan sonraki özel bloğu asla çalıştırılamaz hâle getirir ve derleyici bunu hata olarak işaretler.

instanceof ile Hiyerarşiyi Çalışma Zamanında Kontrol Etmek

`instanceof` operatörü, bir nesnenin belirli bir sınıfın (veya onun herhangi bir atasının) örneği olup olmadığını çalışma zamanında sorar — `catch` bloğunun statik olarak yaptığı eşleştirmenin aynısını, kod içinde açıkça kontrol etmeni sağlar.

```
// instanceof asks "is this object a member of this branch of the hierarchy?"
// at runtime -- true for the exact class AND every ancestor. This is useful
// when a single catch block needs to react differently depending on how
// specific the caught exception actually is, without writing a separate
// catch clause for every possible subtype.
public class InstanceofHierarchyCheckExample {
    public static void main(String[] args) {
        describe(new NumberFormatException("example"));
        describe(new IllegalArgumentException("example"));
        describe(new ArithmeticException("example"));
        describe(new Exception("example"));
    }

    private static void describe(Exception exception) {
        // Checked broadest-relevant-to-narrowest is not required here since
        // each branch just prints a label -- but in real code, ordering
        // instanceof checks (or catch clauses) from MOST specific to LEAST
        // specific matters, or a broader check would shadow a narrower one.
        if (exception instanceof NumberFormatException) {
            System.out.println(exception.getClass().getSimpleName() + " -> is a
NumberFormatException");
        } else if (exception instanceof IllegalArgumentException) {
            System.out.println(exception.getClass().getSimpleName() + " -> is an
IllegalArgumentException, but not a NumberFormatException");
        } else if (exception instanceof RuntimeException) {
            System.out.println(exception.getClass().getSimpleName() + " -> is some other
RuntimeException");
        } else {
            System.out.println(exception.getClass().getSimpleName() + " -> is not a
RuntimeException at all");
        }

        // Every exception here also passes this check, since Exception sits
        // above all of them in the hierarchy -- instanceof against a common
        // ancestor is always true for every descendant.
        System.out.println("  instanceof Exception: " + (exception instanceof Exception));
    }
}
```

Beklenen Çıktı

```
NumberFormatException -> is a NumberFormatException
  instanceof Exception: true
IllegalArgumentException -> is an IllegalArgumentException, but not a NumberFormatException
  instanceof Exception: true
ArithmeticException -> is some other RuntimeException
  instanceof Exception: true
Exception -> is not a RuntimeException at all
  instanceof Exception: true
```

Bu, özellikle tek bir geniş `catch (Exception e)` bloğu içinde, yakalanan nesnenin GERÇEKTE hangi tür olduğuna göre farklı davranmak gerektiğinde kullanışlıdır — ama çoğu durumda, ayrı `catch` bloklarıyla aynı sonucu daha okunaklı elde edebileceğini unutma.

Best Practices

- Bir `catch` bloğu yazarken elinden geldiğince SPESİFİK ol — `catch (RuntimeException e)` yerine gerçekten beklediğin türü (`catch (NumberFormatException e)`) yakala; geniş bir tür yalnızca gerçekten birden fazla türü aynı şekilde ele almak istediğinde mantıklı.
- `Error`'ı (veya `Throwable`'ı doğrudan) yakalamaktan kaçın — neredeyse her zaman JVM zaten kurtarılamaz bir durumdadır, yakalamak sorunu gizler.
- Hiyerarşiyi hatırlamak için IDE'nin "go to superclass" (üst sınıfa git) özelliğini veya `getSuperclass()` zincirini (bu konudaki ilk örnekte gösterildiği gibi) kullan; ezberlemeye çalışmak yerine gerektiğinde doğrula.
- Birden fazla `catch` bloğu yazarken, en spesifik türü en üste koy — derleyici sıra yanlışsa zaten hata verir, ama bu alışkanlığı baştan doğru kurmak okunabilirliği artırır.

Yaygın Hatalar

- `catch (Exception e)` yazıp içine hiçbir şey koymadan geçmek — bu, hem `Error` alt sınıflarını DEĞİL (`Exception` , `Error`'ı kapsamaz) ama TÜM checked ve unchecked exception'ları sessizce yutar, hata ayıklamayı imkânsız hâle getirir.
- `catch (Throwable t)` yazmak — bu, `Error`'ı da kapsar ve neredeyse hiçbir zaman doğru bir seçim değildir.
- Bir `StackOverflowError`'ı yakalayıp normal akışa devam etmeye çalışmak — yığın zaten tükenmiş durumdayken bu, yeni ve daha öngörülemez hatalara yol açabilir.
- Hiyerarşiyi karıştırıp `RuntimeException`'ın `Exception`'dan AYRI bir dal olduğunu düşünmek — aslında `RuntimeException` , `Exception`'ın kendisinin bir alt sınıfıdır, kardeşi değil.

Özet, Cheat Sheet ve Terimler Sözlüğü

Özet

- Her exception ve hata `Throwable`'dan türer; `Throwable`'ın iki dalı vardır: `Error` ve `Exception`.
- `Error`, JVM seviyesindeki kurtarılamaz durumları temsil eder (`StackOverflowError` gibi) — normalde yakalanmamalı.
- `Exception`, uygulama kodunun tepki verebileceği durumları temsil eder; `RuntimeException` onun bir alt sınıfıdır.
- `catch`, tam eşleşen türü değil, o türün herhangi bir atasını da hedefleyebilir (polimorfik yakalama).
- `instanceof`, bir nesnenin hiyerarşideki yerini çalışma zamanında sorgulamanı sağlar.

Cheat Sheet

```
// Throwable
//   ├── Error (StackOverflowError, OutOfMemoryError, ...)
//   └── Exception
//         ├── RuntimeException (NumberFormatException, ArithmeticException, ...)
//         └── (checked exception'lar -- bkz. Checked vs Unchecked Exceptions)

try {
    riskyOperation();
} catch (RuntimeException e) {   // spesifik türlerin hepsini kapsar
    // ...
}

if (something instanceof RuntimeException) {
    // ...
}
```

Terimler Sözlüğü

- **Throwable**: `catch`/`throw` edilebilen her şeyin ortak atası.
- **Error**: JVM seviyesinde, genellikle kurtarılamaz durumları temsil eden `Throwable` alt sınıfı.
- **Hiyerarşi**: Sınıflar arasındaki miras (inheritance) ağacı; burada `Throwable` kökü, `Error`/`Exception` dalları oluşturur.
- **Polimorfik yakalama**: Bir `catch` bloğunun, tam eşleşen türün yanı sıra o türün herhangi bir alt sınıfını da yakalayabilmesi.
- **instanceof**: Bir nesnenin belirli bir sınıfın (veya atasının) örneği olup olmadığını çalışma zamanında sorgulayan operatör.