Checked ve Unchecked Exception'lar

Checked exception'ın derleyici tarafından zorlanan throws/catch sözleşmesi ile unchecked exception'ın (RuntimeException ailesi) bu zorunluluğu hiç taşımaması arasındaki fark -- ikisi arasında ne zaman hangisinin seçileceği, checked bir exception'ı cause parametresiyle unchecked'e sarmalamak, ve override edilen bir metotta throws bildiriminin yalnızca DARALTILABİLECEĞİ kısıtı. Exception Handling serisinin 4.'sü.

Başlangıç 20 dk

Oluşturulma tarihi: 2 Ekim 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/checked-vs-unchecked-exceptions

---

"Exception Hiyerarşisi" konusunda `Exception`'ın iki geniş dala ayrıldığını gördün: `RuntimeException` ve onun DIŞINDA kalan her şey. Bu ayrım, isimlerinin ima ettiğinden çok daha somut bir sonuç doğuruyor — derleyicinin seni gerçekten zorladığı bir kural. Bu derste bu kuralın tam olarak ne olduğunu, neden var olduğunu, ve profesyonel Java kodunda ikisi arasında nasıl seçim yapılacağını göreceksin.

Checked ve Unchecked Exception'lar Nedir?

Checked exception, `Exception`'ın altında olup `RuntimeException`'ın DIŞINDA kalan her sınıftır (`IOException`, `SQLException` gibi). Bir metot checked bir exception fırlatabiliyorsa, bunu `throws` ile imzasında bildirmek ZORUNDADIR — ve o metodu çağıran her kod, bu exception'ı ya `catch` etmek ya da kendi imzasında `throws` ile bildirmek ZORUNDADIR. **Unchecked exception** ise `RuntimeException`'ın (ve `Error`'ın) kendisi ve tüm alt sınıflarıdır — bunlar için ne `throws` bildirmek ne de `catch` etmek zorunludur; derleyici hiçbir şey talep etmez.

Neden Var?

Bu ayrımın amacı, bir API'nin kullanıcısına iki farklı sözleşme sunabilmektir. Checked exception, "bu işlem başarısız olabilir ve BU BAŞARISIZLIK SENİN KONTROLÜNÜN DIŞINDA, ama makul biçimde beklenebilir bir durum — bunu görmezden gelemezsin" der (dosya bulunamaması, ağ bağlantısının kopması gibi). Unchecked exception ise genellikle bir PROGRAMLAMA HATASINI temsil eder (null bir referansa erişmek, geçersiz bir indekse erişmek) — bunu her çağrı noktasında `catch` etmeye zorlamak, kodu okunamaz hale getirir ve asıl hatayı (kaynak kodundaki bir bug'ı) gizler.

Tarihçe

Java 1.0'dan (1996) beri checked exception mekanizması var — bu, dilin en tartışmalı tasarım kararlarından biri olarak kabul edilir. C++ gibi diller exception'ları hiç zorunlu kılmazken, Java'nın tasarımcıları güvenilirliği artırmak için checked exception'ı bilinçli olarak seçti. Zamanla topluluk, checked exception'ın HER durumda doğru araç olmadığını fark etti — bu yüzden Java'nın kendi standart kütüphanesi bile zamanla `RuntimeException` tabanlı alternatifler ekledi (`java.io.UncheckedIOException` gibi, Java 8'de eklendi).

Derleyicinin Zorladığı Sözleşme: Checked Exception'lar

Checked bir exception fırlatabilen bir metot, bunu `throws` ile açıkça bildirmek zorundadır — ve bu metodu çağıran HER kod, ya bir `catch` bloğuyla karşılamalı ya da kendi `throws` bildirimine eklemelidir. Bu, isteğe bağlı bir öneri değil, derleme zamanında zorlanan bir kuraldır.

```
import java.io.IOException;

// A checked exception is anything that extends Exception but NOT RuntimeException
// (see "Exception Hierarchy" for that split). The compiler enforces a real contract
// around it: if a method can throw one, every caller must either catch it or declare
// it with `throws` in its own signature -- there is no way to call this method and
// silently ignore that possibility, the way you can with an unchecked exception.
public class CheckedExceptionHandlingExample {
    public static void main(String[] args) {
        try {
            System.out.println("Loaded: " + readSetting("timeout"));
            System.out.println("Loaded: " + readSetting("missing"));
        } catch (IOException e) {
            // Forced by the compiler -- removing this catch (or a `throws IOException`
            // on main) would not compile.
            System.out.println("Caught checked exception: " + e.getMessage());
        }
    }

    // IOException is used here purely as a familiar, real checked exception to
    // demonstrate the COMPILER'S enforcement -- no actual file I/O happens (see the
    // File Reading lesson for real IOException scenarios from disk access).
    private static String readSetting(String key) throws IOException {
        if (key.equals("missing")) {
            throw new IOException("setting not found: " + key);
        }
        return "30s";
    }
}
```

Bu örnekte `readSetting(...)` , `throws IOException` bildirdiği için `main` 'in onu ya `catch` etmesi ya da kendisinin de `throws IOException` bildirmesi gerekiyor — üçüncü bir seçenek yok, derleyici izin vermiyor.

Unchecked Exception'lar: RuntimeException Ailesi

`RuntimeException`'ın (ve alt sınıflarının) tam tersi bir sözleşmesi var: hiçbir `throws` bildirimi ya da `catch` bloğu ZORUNLU değil. Bu, "Exception Hiyerarşisi" konusunda gördüğün `ArithmeticException`, `NumberFormatException` gibi sınıfların hepsinin ortak özelliği.

```
// An unchecked exception is any RuntimeException (or subclass of it). The compiler
// places NO obligation on a caller to catch it or declare it -- this file compiles
// and runs perfectly fine with no try/catch anywhere and no `throws` on main, even
// though divide(...) can clearly fail.
public class UncheckedExceptionExample {
    public static void main(String[] args) {
        System.out.println("Before calling divide(...)");
        int result = divide(10, 0);
        System.out.println("This line never runs: " + result);
    }

    private static int divide(int a, int b) {
        return a / b; // throws ArithmeticException -- unchecked, no `throws` clause needed
    }
}
```

Bu örnek, hiçbir `try`/`catch` ya da `throws` bildirimi olmadan derleniyor ve çalışıyor — `divide(...)`'ın bir `ArithmeticException` fırlatabileceği gerçeği, derleyici için hiçbir fark yaratmıyor.

> Bir exception'ın checked mi unchecked mi olduğunu anlamanın en hızlı yolu: sınıf `RuntimeException`'ı (ya da `Error`'ı) miras alıyor mu diye bakmaktır — alıyorsa unchecked'tir, almıyorsa (ve `Exception`'ın altındaysa) checked'tir.

Hangisini Ne Zaman Kullanmalı?

Genel kabul gören yaklaşım şudur: eğer çağıran kodun makul bir şekilde KURTARABİLECEĞİ bir durum söz konusuysa (örneğin bir dosyanın bulunamaması — kullanıcıdan farklı bir yol istemek gibi bir tepki mümkün) checked exception mantıklıdır. Eğer durum bir PROGRAMLAMA HATASINI (geçersiz bir argüman, null bir referans) ya da çağıranın gerçekte hiçbir şey yapamayacağı bir durumu temsil ediyorsa, unchecked exception daha uygundur. Modern Java kütüphanelerinin çoğu (Spring dahil) checked exception kullanımını bilinçli olarak sınırlı tutar, çünkü her API sınırında zorunlu `catch`/`throws` zinciri kod tabanını hızla şişirebilir.

Checked Exception'ı Unchecked'e Sarmalamak (Wrapping)

Bazen çağıran kodun imzası checked bir exception bildiremez (örneğin bir interface metodunu override ederken — bir sonraki bölümde göreceksin) ya da böyle bir bildirim API'yi gereksiz yere karmaşıklaştırır. Bu durumda, checked exception'ı yakalayıp unchecked bir exception içine SARMALAMAK (wrap) yaygın bir çözümdür.

```
import java.io.IOException;

// A common, real pattern: catch a checked exception and re-throw it wrapped inside
// an unchecked one, when the surrounding method's signature can't (or shouldn't)
// declare `throws IOException` itself -- for example, because it overrides an
// interface method that doesn't declare it (see the next example in this lesson).
public class WrappingCheckedAsUncheckedExample {
    public static void main(String[] args) {
        loadRequiredSetting("missing");
    }

    // No `throws` clause needed here -- RuntimeException is unchecked, so this
    // compiles even though readSetting(...) itself can throw a checked exception.
    private static void loadRequiredSetting(String key) {
        try {
            readSetting(key);
        } catch (IOException e) {
            // The constructor's second argument is the ORIGINAL exception, passed
            // as the "cause" -- nothing about the checked exception's message or
            // stack trace is lost, it's just no longer something the caller is
            // FORCED to handle.
            throw new RuntimeException("failed to load required setting: " + key, e);
        }
    }

    private static String readSetting(String key) throws IOException {
        if (key.equals("missing")) {
            throw new IOException("setting not found: " + key);
        }
        return "30s";
    }
}
```

Burada `IOException`, `RuntimeException`'ın constructor'ının ikinci argümanı olan `cause` parametresiyle sarmalanıyor — orijinal exception'ın bilgisi (mesaj, stack trace) kaybolmuyor, yalnızca artık çağıranın ZORUNLU olarak ele alması gerekmiyor.

> Bir checked exception'ı sarmalarken orijinal exception'ı `cause` olarak GEÇMEYİ unutma (`new RuntimeException(message, e)` gibi) — aksi halde asıl hatanın stack trace'i kaybolur ve hata ayıklamak çok zorlaşır.

Override Edilen Metotlarda throws Kısıtı

Bir metodu override ederken, üst sınıfın/interface'in bildirdiği checked exception'lardan DAHA AZINI (ya da daha DAR bir alt tipini) bildirebilirsin — ama asla DAHA FAZLASINI ya da daha GENİŞ bir checked tipini bildiremezsin. Bu kural derleyici tarafından zorlanır.

```
import java.io.FileNotFoundException;
import java.io.IOException;

// When overriding a method, you may declare FEWER (or NARROWER) checked exceptions
// than the method you're overriding -- but never MORE, and never a BROADER checked
// exception type. This is enforced by the compiler, not just a convention.
public class OverridingThrowsRestrictionExample {
    interface SettingsSource {
        String read(String key) throws IOException;
    }

    // Narrows IOException down to FileNotFoundException (one of its subclasses) --
    // legal, because any caller that already handles IOException also handles this.
    static class StrictSettingsSource implements SettingsSource {
        @Override
        public String read(String key) throws FileNotFoundException {
            if (key.equals("missing")) {
                throw new FileNotFoundException("setting not found: " + key);
            }
            return "30s";
        }
    }

    // Drops the `throws` clause entirely -- also legal, since declaring nothing is
    // narrower than declaring IOException.
    static class InMemorySettingsSource implements SettingsSource {
        @Override
        public String read(String key) {
            return "30s";
        }
    }

    public static void main(String[] args) throws IOException {
        SettingsSource source = new StrictSettingsSource();
        System.out.println(source.read("timeout"));

        SettingsSource memorySource = new InMemorySettingsSource();
        System.out.println(memorySource.read("timeout"));

        // If StrictSettingsSource.read tried to declare `throws Exception` (broader
        // than IOException), this file would NOT compile -- the restriction is
        // checked by the compiler at the override site, not at any call site.
    }
}
```

Bu örnekte `StrictSettingsSource`, arayüzün bildirdiği `IOException`'ı onun bir alt sınıfı olan `FileNotFoundException`'a DARALTIYOR; `InMemorySettingsSource` ise `throws` bildirimini tamamen KALDIRIYOR — ikisi de geçerli, çünkü ikisi de çağıranın zaten beklediği `IOException` sözleşmesinin bir alt kümesi.

Best Practices

- Checked exception'ı yalnızca çağıranın GERÇEKTEN kurtarabileceği durumlar için kullan; programlama hataları için `RuntimeException`'ı tercih et.

- Bir checked exception'ı sarmalarken orijinal exception'ı her zaman `cause` olarak geç, asla bilgiyi kaybetme.
- API tasarlarken checked exception kullanımını gerekçelendir — her checked exception, o API'yi kullanan HERKESE `catch` / `throws` yükü bindirir.
- Override edilen bir metotta hangi checked exception'ların bildirilebileceğini unutursan, derleyici zaten seni durduracak — ama bu kısıtı önceden bilmek, arayüz tasarımı yaparken doğru kararı vermeni sağlar.

Yaygın Hatalar

- Her checked exception'ı hiçbir şey yapmadan `catch (IOException e) {}` ile boş bırakmak — bu, "Yaygın Hatalar" bölümlerinde tekrar tekrar göreceğin, hatayı tamamen sessizce yutan bir anti-pattern.
- Checked exception'ları hiç düşünmeden `catch (Exception e)` gibi aşırı geniş bir tipe yakalamak — bu, hem checked hem unchecked her şeyi (isteyerek ya da istemeyerek) aynı bloğa toplar.
- Bir exception'ı sarmalarken `cause` parametresini geçmeyi unutmak — orijinal hatanın stack trace'i kaybolur.
- Checked exception'ın her zaman "daha iyi" ya da "daha profesyonel" olduğunu düşünmek — aslında modern Java pratiği genellikle unchecked exception'ı tercih eder, checked exception yalnızca gerçekten kurtarılabilir durumlar için saklanır.

Özet, Cheat Sheet ve Terimler Sözlüğü

Özet

- Checked exception, `RuntimeException`'ın dışında kalan `Exception` alt sınıflarıdır — `throws` bildirmek veya `catch` etmek ZORUNLUDUR.
- Unchecked exception, `RuntimeException`'ın (ve `Error`'ın) kendisi ve alt sınıflarıdır — hiçbir bildirim zorunlu değildir.
- Checked exception genellikle kurtarılabilir, dış kaynaklı durumları; unchecked exception genellikle programlama hatalarını temsil eder.
- Bir checked exception'ı `cause` parametresiyle unchecked bir exception'a sarmalamak yaygın ve güvenli bir tekniktir.
- Override edilen bir metot, üst sınıfın bildirdiği checked exception'lardan yalnızca DAHA AZINI/DAR bir alt tipini bildirebilir.

Cheat Sheet

```
// Checked  -- throws bildirmek/catch etmek ZORUNLU
void readFile() throws IOException { ... }

// Unchecked -- hiçbir bildirim zorunlu değil
void divide(int a, int b) { return a / b; }

// Sarmalama (wrapping)
try {
    readFile();
} catch (IOException e) {
    throw new RuntimeException("failed", e); // cause'u unutma
}

// Override kısıtı: yalnızca DARALTABİLİRSİN
interface Source { String read() throws IOException; }
class Strict implements Source {
    public String read() throws FileNotFoundException { ... } // OK, alt tip
}
```

Terimler Sözlüğü

- **Checked exception**: `RuntimeException`'ın dışında kalan `Exception` alt sınıfı; `throws` / `catch` zorunludur.
- **Unchecked exception**: `RuntimeException` (veya `Error`) ve alt sınıfları; hiçbir bildirim zorunlu değildir.
- **Sarmalama (wrapping)**: Bir exception'ı yakalayıp başka bir exception'ın `cause`'u olarak yeniden fırlatmak.
- **cause**: Bir exception'ın kendisine yol açan orijinal exception'a referansı.
- **Daraltma (narrowing)**: Override edilen bir metodun, üst sınıfın bildirdiği checked exception'dan daha dar bir alt tip (ya da hiçbirini) bildirmesi.