Özel (Custom) Exception Oluşturmak ve Fırlatmak

Kendi exception türünü Exception (checked) ya da RuntimeException (unchecked) genişleterek tanımlamak -- ekstra bağlam (context) taşıyan alanlar eklemek, Throwable'ın dört standart constructor'ını yansıtmak, ve ortak bir custom base sınıfıyla kendi küçük exception hiyerarşini kurmak. Exception Handling serisinin 6.'sı.

Başlangıç 20 dk

Son güncelleme: 10 Kasım 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/custom-exceptions

---

Bu seri boyunca şimdiye kadar fırlattığın her exception, hazır bir Java türüydü — `IllegalStateException`, `IllegalArgumentException`, `IOException`. Bunlar işe yarıyor, ama yalnızca genel isimlerinin ve bir metin mesajının izin verdiği kadar konuşabiliyorlar. Bu ders, KENDİ exception türlerini tasarlamayı ele alıyor: bir hatayı kendi uygulamanın kelime dağarcığıyla tanımlayan, ve bir mesajın taşıyabileceğinden daha fazlasını taşıyabilen sınıflar.

Custom (Özel) Exception Nedir?

Custom exception, basitçe SENİN yazdığın, `Exception`'ı (checked yapan) ya da `RuntimeException`'ı (unchecked yapan) genişleten bir sınıftır — onu gerçek, kullanılabilir bir exception türü yapmak için başka hiçbir şeye gerek yoktur. Tanımlandıktan sonra fırlatılabilir, yakalanabilir, ve hazır exception'lar için zaten gördüğün AYNI `catch` / `throws` /hiyerarşi kurallarına tabi olur.

Neden Var?

`IllegalStateException` gibi genel bir exception, kodun güvenilir biçimde tepki veremeyeceği bir metin mesajının ötesinde, TAM OLARAK neyin ters gittiği hakkında çağırana neredeyse hiçbir şey söylemez. Custom bir exception türü, bir `catch` bloğuna eşleşecek somut bir şey verir — `catch (InsufficientFundsException e)`, mesaj kontrolü yapan bir `catch (Exception e)` 'nin asla olamayacağı kadar açıktır — ve genel bir exception'ın alanı olmayan yapılandırılmış veriyi (geçersiz bir değer, bir hata kodu) taşıyabilir.

Minimal Bir Custom Exception

En basit custom exception, mesajı superclass'a ileten bir constructor dışında hiçbir şey eklemez.

```
public class BasicCustomExceptionExample {

    // A custom checked exception: extends Exception (not RuntimeException),
    // so every caller of a method that throws it must catch it or declare
    // it with "throws" -- the same compiler-enforced contract you saw with
    // built-in checked exceptions like IOException.
    static class InsufficientFundsException extends Exception {
        InsufficientFundsException(String message) {
            super(message);
        }
    }

    public static void main(String[] args) {
        try {
            withdraw(100.0, 250.0);
        } catch (InsufficientFundsException e) {
            System.out.println("Withdrawal failed: " + e.getMessage());
        }
    }

    static void withdraw(double balance, double amount) throws InsufficientFundsException {
        if (amount > balance) {
            throw new InsufficientFundsException(
                    "cannot withdraw " + amount + ", balance is only " + balance);
        }
        System.out.println("withdrew " + amount);
    }
}
```

`InsufficientFundsException`, `Exception`'ı genişletir, bu yüzden checked'tir — `withdraw(...) throws InsufficientFundsException` bildirmek zorundadır, ve `main` onu ya `catch` etmeli ya da yine bildirmelidir — tıpkı "Checked ve Unchecked Exception'lar"da hazır checked exception'larla gördüğün gibi.

> Kural olarak, her custom exception'ın sınıf adı `Exception` ile biter — `InsufficientFunds` değil, `InsufficientFundsException`. Bu derleyici tarafından zorlanmaz, ama her Java kod tabanının beklediği güçlü bir okunabilirlik kuralıdır.

Ekstra Bağlam (Context) Taşımak

Custom bir exception'ın hazır bir exception'a göre gerçek avantajı, kendi ALANLARINI (fields) taşıyabilmesidir — yalnızca mesaj metninin ötesinde, bir `catch` bloğunun geri okuyabileceği veri.

```
public class CustomExceptionWithContextExample {

    // A custom unchecked exception: extends RuntimeException, so no
    // "throws" declaration is required at call sites. It carries an EXTRA
    // field beyond the inherited message -- the invalid value itself --
    // giving a catch block structured data to act on, not just text.
    static class InvalidOrderQuantityException extends RuntimeException {
        private final int quantity;

        InvalidOrderQuantityException(int quantity) {
            super("order quantity must be positive, was " + quantity);
            this.quantity = quantity;
        }

        int getQuantity() {
            return quantity;
        }
    }

    public static void main(String[] args) {
        try {
            placeOrder(-3);
        } catch (InvalidOrderQuantityException e) {
            System.out.println(e.getMessage());
            System.out.println("rejected quantity was: " + e.getQuantity());
        }
    }

    static void placeOrder(int quantity) {
        if (quantity <= 0) {
            throw new InvalidOrderQuantityException(quantity);
        }
        System.out.println("order placed for " + quantity);
    }
}
```

`InvalidOrderQuantityException`, kalıtsal aldığı mesajın yanı sıra reddedilen `quantity`'yi kendi alanında saklar. `main`'deki `catch` bloğu, bu değeri doğrudan geri almak için `e.getQuantity()` çağırır — bir mesaj metnini ayrıştırmaya gerek kalmaz.

Hazır Constructor Şekillerini Yansıtmak

`Throwable`'ın kendisi dört constructor sunar: argümansız, yalnızca mesaj, mesaj+cause, ve yalnızca cause. İyi tasarlanmış bir custom exception genelde bu dördünü de yansıtır, böylece çağıranlar onu hazır exception'ları kullandıkları gibi kullanabilir.

```
public class CustomExceptionConstructorsExample {

    // Mirroring the four constructors Throwable itself offers is a common
    // convention -- it lets callers of your exception choose exactly the
    // same shapes they already use with built-in exceptions: no details,
    // a message only, a message with a cause, or a cause only.
    static class ReportGenerationException extends RuntimeException {
        ReportGenerationException() {
            super();
        }

        ReportGenerationException(String message) {
            super(message);
        }

        ReportGenerationException(String message, Throwable cause) {
            super(message, cause);
        }

        ReportGenerationException(Throwable cause) {
            super(cause);
        }
    }

    public static void main(String[] args) {
        try {
            throw new ReportGenerationException("PDF renderer returned no pages");
        } catch (ReportGenerationException e) {
            System.out.println("message-only: " + e.getMessage());
        }

        try {
            Exception original = new IllegalStateException("renderer not initialized");
            throw new ReportGenerationException("report generation failed", original);
        } catch (ReportGenerationException e) {
            System.out.println("message+cause: " + e.getMessage() + " <- " + e.getCause());
        }
    }
}
```

`ReportGenerationException`, her constructor'ın argümanlarını doğrudan `super(...)`'a iletir. Bu, "Throw ve Throws"taki desenin — yakalanan bir exception'ın farklı, daha anlamlı bir türle yeniden fırlatılması (wrapping) — kendi exception türlerinle de sorunsuz çalışmasını sağlayan şeydir: orada başvuracağın şey mesaj-ve-cause constructor'ıdır.

Kendi Exception Hiyerarşini Kurmak

`IOException` ve `SQLException`'ın ikisinin de ortak `Exception` türünün altında yer alması gibi, kendi exception'ların da ortak bir custom base sınıfı paylaşabilir — çağırana geniş yakalama (ortak sorun) ya da dar yakalama (tek bir spesifik neden) arasında seçim şansı verir.

```
public class CustomExceptionHierarchyExample {

    // A small hierarchy of your own, built the same way Java's own
    // exception classes are: one shared base type, several specific
    // subclasses. Code that only cares about "some payment problem
    // happened" can catch the base type; code that needs to react
    // differently to each cause can catch the specific subclasses instead.
    static class PaymentException extends RuntimeException {
        PaymentException(String message) {
            super(message);
        }
    }

    static class CardDeclinedException extends PaymentException {
        CardDeclinedException(String message) {
            super(message);
        }
    }

    static class PaymentGatewayTimeoutException extends PaymentException {
        PaymentGatewayTimeoutException(String message) {
            super(message);
        }
    }

    public static void main(String[] args) {
        // Catching by the shared base type, exactly like catching by
        // RuntimeException in "Exception Hierarchy" -- both subclasses
        // match, without listing each one.
        for (String scenario : new String[] {"declined", "timeout"}) {
            try {
                charge(scenario);
            } catch (PaymentException e) {
                System.out.println("payment failed (" + e.getClass().getSimpleName() + "):
" + e.getMessage());
            }
        }
    }

    static void charge(String scenario) {
        if (scenario.equals("declined")) {
            throw new CardDeclinedException("card was declined by the issuer");
        }
        throw new PaymentGatewayTimeoutException("gateway did not respond in time");
    }
}
```

Beklenen Çıktı

```
payment failed (CardDeclinedException): card was declined by the issuer
payment failed (PaymentGatewayTimeoutException): gateway did not respond in time
```

`CardDeclinedException` ve `PaymentGatewayTimeoutException`, ikisi de `PaymentException`'ı genişletir. `main`'deki döngü yalnızca `PaymentException`'ı yakalar ve iki somut hatayı da TEK bir `catch` bloğuyla ele alır — "Exception Hiyerarşisi"nde hazır türlerle gördüğün süper tip

üzerinden polimorfik eşleştirmenin AYNISI, şimdi kendi tasarladığın bir hiyerarşiye uygulanmış hâli.

> "Ne olur ne olmaz" diye derin bir custom exception hiyerarşisi kurma. Kodunun gerçekten farklı tepki vermesi gereken her gerçek hata için bir exception türüyle başla — gerçek bir ihtiyaç ortaya çıktığında bir alt sınıf ekle.

Best Practices

- Her custom exception türünü, her Java kod tabanının beklediği kurala uyarak `Exception` sonekiyle adlandır.
- Çağıran gerçekten hatadan kurtulabiliyor ve bunu ele almaya zorlanmalıysa `Exception`, aksi hâlde `RuntimeException` genişlet — "Checked ve Unchecked Exception'lar"daki aynı kılavuz kendi türlerin için de geçerlidir.
- Bir `catch` bloğunun ihtiyaç duyabileceği her yapılandırılmış veri için, yalnızca mesaj metnine kodlamak yerine alan ekle.
- `Throwable`'ın standart constructor'larını (argümansız, mesaj, mesaj+cause, cause) yansıt, böylece exception'ın wrapping ile temiz bir şekilde birleşir.

Yaygın Hatalar

- `Exception` ya da `RuntimeException` yerine doğrudan `Throwable`'ı genişletmek — bu, checked/unchecked ayrımını tamamen atlar ve neredeyse hiçbir zaman istediğin şey değildir.
- `super(message)` (ya da `super(message, cause)`) çağırmayı unutup `getMessage()`'ın sebepsiz yere `null` dönmesine yol açmak.
- Hiçbir kodun gerçekten farklı yakalamaya ihtiyacı olmadığı halde, gereğinden derin ya da geniş bir custom exception hiyerarşisi tasarlamak.
- Hazır bir exception (`IllegalArgumentException` gibi) zaten tam olarak gerekeni söylerken custom bir exception'a başvurmak — her hata yepyeni bir tür gerektirmez.

Özet, Cheat Sheet ve Terimler Sözlüğü

Özet

- Custom exception, `Exception`'ı (checked) ya da `RuntimeException`'ı (unchecked) genişleterek yazdığın bir sınıftır.
- Custom exception'lar, bir `catch` bloğunun kesin eşleşmesine ve bir mesaj metninin ötesinde yapılandırılmış veri taşımasına izin verir.
- Kural olarak, custom exception sınıf adları her zaman `Exception` ile biter.

- `Throwable`'ın dört standart constructor'ını yansıtmak, custom exception'ı wrapping ile uyumlu tutar.
- Ortak bir custom base sınıfı, çağıranların geniş ya da dar yakalamasına izin verir — hazır hiyerarşilerin kullandığı aynı polimorfik eşleştirme.

Cheat Sheet

```
// Minimal checked custom exception
class InsufficientFundsException extends Exception {
    InsufficientFundsException(String message) {
        super(message);
    }
}

// Unchecked, ekstra bağlamla
class InvalidOrderQuantityException extends RuntimeException {
    private final int quantity;
    InvalidOrderQuantityException(int quantity) {
        super("invalid quantity: " + quantity);
        this.quantity = quantity;
    }
    int getQuantity() { return quantity; }
}

// Kendi küçük hiyerarşin
class PaymentException extends RuntimeException {
    PaymentException(String message) { super(message); }
}
class CardDeclinedException extends PaymentException {
    CardDeclinedException(String message) { super(message); }
}
```

Terimler Sözlüğü

- **Custom exception**: `Exception` ya da `RuntimeException`'ı genişleterek tanımladığın bir sınıf.
- **Bağlam (context, bir exception'da)**: kalıtsal alınan mesajın ötesinde, bir custom exception'ın taşıdığı ve bir `catch` bloğunun okuyabildiği ekstra alanlar.
- **Constructor yansıtma**: bir custom exception'a `Throwable`'ınkiyle aynı dört constructor şeklini vermek.
- **Custom hiyerarşi**: altında kendi alt sınıfların olan, geniş ya da dar yakalanabilen, ortak bir base exception türü.