Optional

Optional<T>: bir değerin bulunmama ihtimalini tip sisteminde ifade etmek. of() /ofNullable()/empty() ile oluşturma, orElse() vs orElseGet() (eager vs lazy), orElseThrow(), map()/flatMap(), ifPresent()/ifPresentOrElse(), filter().

Orta 20 dk

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

---

Optional

"Terminal Operations" dersinde `reduce(accumulator)`, `min()`, `max()`, `findFirst()`, `findAny()` 'in `Optional<T>` döndürdüğünü görmüş, ama detayına girmemiştik. Bu ders tam olarak `Optional` 'ın kendisine odaklanıyor: bir değerin bulunmama ihtimalini tip sisteminde nasıl ifade ettiği, ve bunu nasıl güvenle kullanacağınız.

Optional Nedir?

`Optional<T>`, bir değeri **olabilir ya da olmayabilir** şeklinde saran bir kapsayıcı (wrapper) sınıftır. Amacı, bir metodun "bu değeri bulamayabilirim" ihtimalini, dönüş tipinde açıkça göstermektir -- çağıran kodun bunu görmezden gelip doğrudan `null` ile karşılaşmasını (ve bir `NullPointerException` 'a çarpmasını) engellemek.

Neden Var?

Java'da geleneksel olarak "değer yok" durumu `null` ile ifade edilir -- ama bir metodun imzasına bakarak `null` dönebileceğini anlayamazsınız; bunu ancak dokümantasyondan öğrenirsiniz (ya da acı yoldan, bir `NullPointerException` ile). `Optional<T>`, bu ihtimali dönüş tipinin **kendisine** taşır: bir metot `Optional<User>` döndürüyorsa, çağıran kodun bunu ele almak zorunda olduğu açıktır -- derleyici sizi zorlamaz, ama tip imzası niyeti nettir.

Tarihçe

`Optional<T>`, Java 8'de (2014) Stream API ile birlikte geldi -- Stream API'nin bazı terminal operation'larının (`reduce(accumulator)`, `min()`, `max()`, `findFirst()`, `findAny()`) boş bir stream için bir değer döndürememesi problemi, `Optional` 'ın asıl motivasyonlarından biriydi. `ifPresentOrElse()` gibi bazı metotlar sonradan, Java 9'da (2017) eklendi.

Optional Oluşturmak: of(), ofNullable(), empty()

Üç fabrika (factory) metodu bir `Optional` oluşturur. `Optional.of(value)`, değerin asla `null` olmayacağını iddia eder -- `null` verilirse anında `NullPointerException` fırlatır. `Optional.ofNullable(value)`, `null` olabilecek bir değeri güvenle sarar -- `null` ise boş bir `Optional` üretir. `Optional.empty()`, kasıtlı olarak boş bir `Optional` oluşturur.

```
import java.util.Optional;

// Optional<T> wraps a value that might be absent -- it exists to make that possibility
// visible in a method's return type, instead of silently returning null. Three factory
// methods create one: of() for a value known to be non-null, ofNullable() for a value
// that might be null, and empty() for a deliberately absent value.
class OptionalCreationExample {
    public static void main(String[] args) {
        Optional<String> present = Optional.of("hello");
        System.out.println(present.isPresent());
        System.out.println(present.isEmpty());
        System.out.println(present.get());

        // of() throws NullPointerException immediately if given null -- it's a
        // deliberate assertion that the value is never null.
        try {
            Optional.of(null);
        } catch (NullPointerException e) {
            System.out.println("caught: of(null) is not allowed");
        }

        // ofNullable() accepts null gracefully, producing an empty Optional instead.
        String maybeNull = null;
        Optional<String> fromNullable = Optional.ofNullable(maybeNull);
        System.out.println(fromNullable.isPresent());

        Optional<String> empty = Optional.empty();
        System.out.println(empty.isEmpty());

        // Calling get() on an empty Optional throws -- exactly the NullPointerException
        // risk Optional exists to make visible and force you to handle explicitly.
        try {
            empty.get();
        } catch (java.util.NoSuchElementException e) {
            System.out.println("caught: get() on empty Optional");
        }
    }
}
```

Beklenen Çıktı

```
true
false
hello
caught: of(null) is not allowed
false
true
caught: get() on empty Optional
```

Bir Optional'ın İçini Okumak: isPresent(), isEmpty(), get()

`isPresent()` ve `isEmpty()`, bir değerin olup olmadığını `boolean` olarak sorar. `get()`, değeri doğrudan çıkarır -- ama Optional boşsa `NoSuchElementException` fırlatır. Bu üçlüyü birlikte kullanmak (`if (opt.isPresent()) { opt.get() }`) teknik olarak çalışır, ama bu dersin geri kalanında görülecek `orElse()` / `map()` / `ifPresent()` gibi metotlar aynı işi daha güvenli ve daha kısa yapar.

orElse() ve orElseGet(): Varsayılan Değer Sağlamak

`orElse(değer)` ve `orElseGet(supplier)`, Optional boşsa kullanılacak bir varsayılan sağlar -- ama varsayılanın **ne zaman** hesaplandığı konusunda ayrılırlar. `orElse()`'in argümanı **her zaman**, Optional dolu olsa bile, hemen hesaplanır. `orElseGet()`'in `Supplier`'ı yalnızca Optional gerçekten boşsa çağrılır -- tembel (lazy) değerlendirme (Built-in Functional Interfaces dersindeki `Supplier<T>`'ın tam olarak var olma sebebi).

```
import java.util.Optional;

// orElse(value) and orElseGet(supplier) both provide a default when the Optional is
// empty -- but they differ in WHEN the default is computed. orElse()'s argument is
// evaluated EAGERLY, every time, whether the Optional is present or not. orElseGet()'s
// Supplier is only invoked LAZILY, if the Optional actually turns out to be empty.
class OrElseExample {
    public static void main(String[] args) {
        Optional<String> present = Optional.of("value");

        System.out.println(present.orElse(computeDefault("orElse, present")));
        System.out.println(present.orElseGet(() -> computeDefault("orElseGet, present")));

        Optional<String> empty = Optional.empty();
        System.out.println(empty.orElse(computeDefault("orElse, empty")));
        System.out.println(empty.orElseGet(() -> computeDefault("orElseGet, empty")));
    }

    // Printing a message lets us SEE whether this actually ran or not.
    static String computeDefault(String label) {
        System.out.println("computing default: " + label);
        return "default";
    }
}
```

Beklenen Çıktı

```
computing default: orElse, present
value
value
computing default: orElse, empty
default
computing default: orElseGet, empty
default
```

orElseThrow(): Özel Bir İstisna Fırlatmak

`orElseThrow()`'un iki biçimi vardır. Argümansız hali, `get()` ile tamamen aynı `NoSuchElementException`'ı fırlatır -- yalnızca niyeti daha net ifade eder. Bir `Supplier<X extends Throwable>` alan hali, kendi alan-özel (domain-specific) istisnanızı fırlatmanızı sağlar. Optional doluysa, `Supplier` hiç çağrılmaz -- `orElseGet()` ile aynı tembel değerlendirme mantığı.

```
import java.util.NoSuchElementException;
import java.util.Optional;

// orElseThrow() has two forms: no-argument, which throws a plain
// NoSuchElementException (identical to what get() throws), and one-argument, which
// takes a Supplier<X extends Throwable> so you can throw a specific, meaningful
// exception for your own domain.
class OrElseThrowExample {
    public static void main(String[] args) {
        Optional<String> empty = Optional.empty();

        try {
            empty.orElseThrow();
        } catch (NoSuchElementException e) {
            System.out.println("caught: no-arg orElseThrow()");
        }

        try {
            empty.orElseThrow(() -> new IllegalStateException("user not found"));
        } catch (IllegalStateException e) {
            System.out.println("caught: " + e.getMessage());
        }

        // On a present Optional, orElseThrow() just returns the value -- the Supplier
        // is never invoked, exactly like orElseGet().
        Optional<String> present = Optional.of("Ahmet");
        System.out.println(present.orElseThrow(() -> new IllegalStateException("never
thrown")));
    }
}
```

map() ve flatMap(): Optional İçindeki Değeri Dönüştürmek

`map(Function)`, Optional'ın **içindeki** değeri dönüştürür, önce `isPresent()` kontrolü yapmanıza gerek kalmadan -- Optional boşsa, fonksiyon hiç çağrılmadan boş bir Optional döner.

`flatMap()`, "Stream API Temelleri" dersinde `Stream.flatMap()`'in çözdüğü aynı iç içelik problemini çözer: dönüştürme fonksiyonunun kendisi bir `Optional` döndürüyorsa, `map()` bir `Optional<Optional<T>>` üretir -- kullanışsız, iç içe bir yapı. `flatMap()`, iç Optional'ı doğrudan dış Optional'a birleştirir.

```
import java.util.Optional;

// map() transforms the value INSIDE an Optional, without you having to check
// isPresent() first -- if the Optional is empty, map() just returns empty, the
// function is never called. flatMap() solves the same nesting problem it solves for
// Stream (the Stream Fundamentals lesson): if the mapping function itself returns an
// Optional, map() would produce an Optional<Optional<T>> -- flatMap() flattens that.
class OptionalMapFlatMapExample {
    public static void main(String[] args) {
        Optional<String> name = Optional.of("ahmet");

        Optional<String> upper = name.map(String::toUpperCase);
        System.out.println(upper.get());

        Optional<String> empty = Optional.empty();
        Optional<String> stillEmpty = empty.map(String::toUpperCase);
        System.out.println(stillEmpty.isEmpty());

        // If the lookup itself can fail, it naturally returns an Optional -- map()
        // would nest it: Optional<Optional<String>>.
        Optional<Optional<String>> nested = name.map(OptionalMapFlatMapExample::findEmail);
        System.out.println(nested.get().get());

        // flatMap() merges the inner Optional into the outer one instead of nesting.
        Optional<String> flat = name.flatMap(OptionalMapFlatMapExample::findEmail);
        System.out.println(flat.orElse("not found"));
    }

    // A lookup that might not find anything -- naturally returns Optional itself.
    static Optional<String> findEmail(String username) {
        return username.equals("ahmet") ? Optional.of("ahmet@example.com") : Optional.
empty();
    }
}
```

Beklenen Çıktı

```
AHMET
true
ahmet@example.com
ahmet@example.com
```

ifPresent() ve ifPresentOrElse(): Yan Etki Uygulamak

`ifPresent(Consumer)`, yalnızca bir değer varsa bir yan etki çalıştırır -- açık bir `null` kontrolüne gerek kalmadan `if (value != null) { ... }`'in karşılığı. `ifPresentOrElse(Consumer, Runnable)`, boş durum için de bir dal ekler -- `ifPresent()` tek başına bunu ifade edemez.

```
import java.util.Optional;

// ifPresent(Consumer) runs a side effect ONLY if a value is present -- the Optional
// equivalent of "if (value != null) { ... }", without an explicit null check.
// ifPresentOrElse(Consumer, Runnable), added in Java 9, adds an "else" branch for the
// empty case, which ifPresent() alone can't express.
class IfPresentExample {
    public static void main(String[] args) {
        Optional<String> present = Optional.of("Ahmet");
        Optional<String> empty = Optional.empty();

        present.ifPresent(name -> System.out.println("found: " + name));
        empty.ifPresent(name -> System.out.println("this never prints: " + name));

        present.ifPresentOrElse(
                name -> System.out.println("present branch: " + name),
                () -> System.out.println("empty branch"));
        empty.ifPresentOrElse(
                name -> System.out.println("present branch: " + name),
                () -> System.out.println("empty branch"));
    }
}
```

Beklenen Çıktı

```
found: Ahmet
present branch: Ahmet
empty branch
```

filter(): Optional İçindeki Değeri Koşulla Süzmek

`filter(Predicate)`, değeri yalnızca koşulu sağlıyorsa tutar -- aksi halde dolu bir Optional'ı boşa çevirir. Zaten boş bir Optional'a hiç dokunmaz (Predicate yalnızca bir değer varsa test edilir). `filter()`, `map()` ve `orElse()` ile birlikte doğal bir doğrulama (validation) zinciri oluşturur -- hiçbir yerde açık bir `isPresent()` / `get()` çağrısına gerek kalmaz.

```
import java.util.Optional;

// filter(Predicate) keeps the value only if it matches the condition -- otherwise it
// turns a present Optional into an empty one. It never touches an already-empty
// Optional (the Predicate is only tested when a value exists).
class OptionalFilterExample {
    public static void main(String[] args) {
        Optional<String> name = Optional.of("Ahmet");

        Optional<String> longEnough = name.filter(n -> n.length() > 3);
        System.out.println(longEnough.orElse("too short"));

        Optional<String> tooShort = name.filter(n -> n.length() > 10);
        System.out.println(tooShort.orElse("too short"));

        // filter() combines naturally with map() and orElse() into a short validation
        // chain -- no explicit isPresent()/get() calls needed anywhere.
        String result = Optional.of("ahmet")
                .filter(n -> !n.isBlank())
                .map(String::toUpperCase)
                .orElse("INVALID");
        System.out.println(result);
    }
}
```

Beklenen Çıktı

```
Ahmet
too short
AHMET
```

Best Practices

- `Optional`**'ı yalnızca dönüş tipi olarak kullanın.** Bir alan (field) tipi, metot parametresi, ya da koleksiyon eleman tipi olarak `Optional` kullanmak, topluluk tarafından yaygın olarak önerilmez -- `Optional`'ın tasarım amacı yalnızca "bu metot bir değer döndürmeyebilir" ihtimalini iletmektir.
- `get()`**'i, önce** `isPresent()` **kontrolü yapmadan çağırmayın** -- ya da daha iyisi, `get()`'i hiç kullanmayın; `orElse()` / `orElseGet()` / `orElseThrow()` / `map()` / `ifPresent()` neredeyse her durumu kapsar.
- **Varsayılan değer ucuzsa** `orElse()`**, hesaplaması pahalıysa veya yan etkiliyse** `orElseGet()` **kullanın** -- `orElse()`'in argümanının her zaman hesaplandığını unutmayın.
- `Optional<T>` **yerine** `null` **döndürmeyi bırakın** yeni yazdığınız metotlarda -- ama zaten `null` dönebilen bir üçüncü parti API ile çalışıyorsanız, `Optional.ofNullable()` ile sarmak, bu ihtimali kodunuzun geri kalanında görünür kılar.

Yaygın Hatalar

- **`get()`'i kontrolsüz çağırmak.** Boş bir Optional'da `get()`, tıpkı `null.toString()` gibi çalışma zamanı hatası verir -- yalnızca farklı bir istisna tipiyle (`NoSuchElementException`).
- **`orElse()`'in argümanının her zaman hesaplandığını unutmak.** Pahalı bir hesaplamayı veya bir yan etkiyi `orElse()`'e vermek, Optional dolu olsa bile o hesaplamayı/yan etkiyi çalıştırır -- bu durumda `orElseGet()` doğru araçtır.
- **`Optional`'ı bir alan tipi olarak kullanmak.** `Optional`, `Serializable` değildir ve bu amaç için tasarlanmamıştır; bir sınıfın alanı için `null` kontrolü veya ayrı bir tasarım (örneğin bir varsayılan değer) daha uygundur.
- **`map()` ile `flatMap()`'i karıştırmak.** Dönüştürme fonksiyonu bir `Optional` döndürüyorsa ve `map()` kullandıysanız, elinizde işe yaramaz bir `Optional<Optional<T>>` kalır -- tıpkı Stream'deki `flatMap()` hatasının aynısı.

Özet, Cheat Sheet ve Terimler Sözlüğü

`Optional<T>`, bir değerin bulunmama ihtimalini tip sisteminde ifade eden bir sarmalayıcıdır: `of()` / `ofNullable()` / `empty()` ile oluşturulur, `orElse()` / `orElseGet()` / `orElseThrow()` ile varsayılan/istisna ile sonlandırılır, `map()` / `flatMap()` ile içindeki değer dönüştürülür, `filter()` ile koşulla süzülür, ve `ifPresent()` / `ifPresentOrElse()` ile yan etki uygulanır.

Hızlı referans:

```
Optional.of(value)              // null verilirse NPE
Optional.ofNullable(value)        // null ise boş Optional
Optional.empty()                    // kasıtlı olarak boş

opt.orElse(defaultValue)              // her zaman hesaplanır
opt.orElseGet(() -> ...)                // yalnızca boşsa çağrılır
opt.orElseThrow(() -> new X())            // yalnızca boşsa çağrılır

opt.map(fn)                                 // içi dönüştürülür, boşsa dokunulmaz
opt.flatMap(fnReturningOptional)              // iç içe Optional'ı düzleştirir
opt.filter(predicate)                           // koşulu sağlamıyorsa boşa çevirir
opt.ifPresent(consumer)                           // yalnızca doluysa çalışır
```

Terimler Sözlüğü

Optional — Bir değerin bulunmama ihtimalini tip sisteminde açıkça ifade eden sarmalayıcı sınıf.

Present (dolu) — Bir `Optional`'ın gerçek bir değer içerdiği durum.

Empty (boş) — Bir `Optional`'ın hiçbir değer içermediği durum.

Eager evaluation (istekli değerlendirme) — Bir ifadenin, sonucuna gerçekten ihtiyaç olup olmadığına bakılmaksızın hemen hesaplanması; `orElse()`'in argümanı bu şekilde davranır.

Lazy evaluation (tembel değerlendirme) — Bir ifadenin yalnızca gerçekten gerektiğinde hesaplanması; `orElseGet()`'in `Supplier`'ı bu şekilde davranır.