Optional

Optional<T>: expressing the possibility of an absent value in the type system. Creating with of()/ofNullable()/empty(), orElse() vs. orElseGet() (eager vs. lazy), orElseThrow(), map()/flatMap(), ifPresent()/ifPresentOrElse(), filter().

Intermediate 20 min

Last updated: November 10, 2026
Read online: https://www.learnforgex.com/en/topics/optional

Optional

In the "Terminal Operations" lesson, you saw that `reduce(accumulator)`, `min()`, `max()`, `findFirst()`, and `findAny()` return `Optional<T>`, but we didn't go into detail. This lesson focuses on `Optional` itself: how it expresses the possibility of an absent value in the type system, and how to use it safely.

What Is Optional?

`Optional<T>` is a wrapper class that holds a value that **may or may not be present**. Its purpose is to make a method's "I might not be able to find this value" possibility explicit in its return type -- so the calling code can't ignore it and run straight into a raw `null` (and a `NullPointerException`).

Why Does It Exist?

Traditionally in Java, "no value" is expressed with `null` -- but you can't tell from a method's signature whether it might return `null`; you only find out from documentation (or the hard way, via a `NullPointerException`). `Optional<T>` moves that possibility into the return type **itself**: if a method returns `Optional<User>`, it's explicit that the calling code has to deal with it -- the compiler doesn't force you to, but the type signature makes the intent clear.

History

`Optional<T>` arrived in Java 8 (2014) alongside the Stream API -- the fact that some Stream terminal operations (`reduce(accumulator)`, `min()`, `max()`, `findFirst()`, `findAny()`) can't return a value for an empty stream was one of `Optional`'s original motivations. Some methods, like `ifPresentOrElse()`, were added later, in Java 9 (2017).

Creating an Optional: of(), ofNullable(), empty()

Three factory methods create an `Optional`. `Optional.of(value)` asserts the value is never `null` -- given `null`, it throws `NullPointerException` immediately. `Optional.ofNullable(value)` safely wraps a value that might be `null` -- producing an empty `Optional` if it is. `Optional.empty()` creates a deliberately empty `Optional`.

```
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");
        }
    }
}
```

Expected Output

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

Reading Inside an Optional: isPresent(), isEmpty(), get()

`isPresent()` and `isEmpty()` ask, as a `boolean`, whether a value exists. `get()` extracts the value directly -- but throws `NoSuchElementException` if the Optional is empty. Using this trio together (`if (opt.isPresent()) { opt.get() }`) technically works, but the methods covered in the rest of this lesson -- `orElse()` / `map()` / `ifPresent()` -- do the same job more safely and more concisely.

orElse() and orElseGet(): Supplying a Default Value

`orElse(value)` and `orElseGet(supplier)` both supply a default to use when the Optional is empty -- but they differ in **when** that default is computed. `orElse()` 's argument is **always** computed immediately, even if the Optional is present. `orElseGet()` 's `Supplier` is only invoked if the Optional turns out to be empty -- lazy evaluation, exactly what `Supplier<T>` (the "Built-in Functional Interfaces" lesson) exists for.

```
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";
    }
}
```

Expected Output

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

orElseThrow(): Throwing a Custom Exception

`orElseThrow()` has two forms. The no-argument form throws exactly the same `NoSuchElementException` as `get()` -- it just communicates intent more clearly. The form that takes a `Supplier<X extends Throwable>` lets you throw your own domain-specific exception instead. If the Optional is present, the `Supplier` is never called -- the same lazy-evaluation logic as `orElseGet()`.

```
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() and flatMap(): Transforming the Value Inside an Optional

`map(Function)` transforms the value **inside** an Optional, without needing an `isPresent()` check first -- if the Optional is empty, the function is never called and an empty Optional comes back.

`flatMap()` solves the same nesting problem `Stream.flatMap()` solved in the "Stream API Fundamentals" lesson: if the transformation function itself returns an `Optional`, `map()` produces an `Optional<Optional<T>>` -- an awkward, nested structure. `flatMap()` merges the inner Optional directly into the outer one.

```
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();
    }
}
```

Expected Output

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

ifPresent() and ifPresentOrElse(): Applying a Side Effect

`ifPresent(Consumer)` runs a side effect only if a value is present -- the counterpart of `if (value != null) { ... }`, without an explicit `null` check. `ifPresentOrElse(Consumer, Runnable)` adds a branch for the empty case too -- something `ifPresent()` alone can't express.

```
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"));
    }
}
```

Expected Output

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

filter(): Filtering the Value Inside an Optional With a Condition

`filter(Predicate)` keeps the value only if it satisfies 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). `filter()` combines naturally with `map()` and `orElse()` into a validation chain -- with no explicit `isPresent()` / `get()` call anywhere.

```
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);
    }
}
```

Expected Output

```
Ahmet
too short
AHMET
```

Best Practices

- **Use `Optional` only as a return type.** Using `Optional` as a field type, a method parameter, or a collection element type is widely discouraged by the community -- `Optional`'s design intent is purely to communicate "this method might not return a value."
- **Don't call `get()` without checking `isPresent()` first** -- or better, avoid `get()` entirely; `orElse()` / `orElseGet()` / `orElseThrow()` / `map()` / `ifPresent()` cover nearly every case.
- **Use `orElse()` when the default is cheap, and `orElseGet()` when computing it is expensive or has a side effect** -- remember that `orElse()`'s argument is always computed.
- **Stop returning `null` in new code you write, in favor of `Optional<T>`** -- but if you're working with a third-party API that can already return `null`, wrapping it with `Optional.ofNullable()` makes that possibility visible in the rest of your code.

Common Mistakes

- **Calling `get()` without checking first.** On an empty Optional, `get()` fails at runtime just like `null.toString()` would -- just with a different exception type (`NoSuchElementException`).
- **Forgetting that `orElse()`'s argument is always computed.** Passing an expensive computation or a side effect to `orElse()` runs it even when the Optional is present -- `orElseGet()` is the right tool in that case.
- **Using `Optional` as a field type.** `Optional` isn't `Serializable` and wasn't designed for this purpose; a `null` check or a separate design (like a default value) is more appropriate for a class field.
- **Confusing `map()` and `flatMap()`.** If the transformation function returns an `Optional` and you used `map()`, you're left with a useless `Optional<Optional<T>>` -- the exact same mistake as `flatMap()` in Streams.

Summary, Cheat Sheet, and Glossary

`Optional<T>` is a wrapper that expresses the possibility of an absent value in the type system: it's created with `of()` / `ofNullable()` / `empty()`, resolved to a default or exception with `orElse()` / `orElseGet()` / `orElseThrow()`, transformed with `map()` / `flatMap()`, filtered with `filter()`, and acted on with `ifPresent()` / `ifPresentOrElse()`.

Quick reference:

```
Optional.of(value)                 // NPE if given null
Optional.ofNullable(value)           // empty Optional if null
Optional.empty()                       // deliberately empty

opt.orElse(defaultValue)                 // always computed
opt.orElseGet(() -> ...)                   // only invoked if empty
opt.orElseThrow(() -> new X())               // only invoked if empty

opt.map(fn)                                    // transforms inside, untouched if empty
opt.flatMap(fnReturningOptional)                 // flattens a nested Optional
opt.filter(predicate)                              // turns to empty if condition fails
opt.ifPresent(consumer)                              // only runs if present
```

Glossary

Optional — A wrapper class that explicitly expresses the possibility of an absent value in the type system.

Present — The state of an `Optional` actually containing a value.

Empty — The state of an `Optional` containing no value.

Eager evaluation — Computing an expression immediately, regardless of whether its result is actually needed; `orElse()`'s argument behaves this way.

Lazy evaluation — Computing an expression only when it's actually needed; `orElseGet()`'s `Supplier` behaves this way.