Lernskript: Der `Optional<T>`-Typ

`Optional<T>` wurde in Java 8 eingeführt, um eine der häufigsten Fehlerquellen in Java zu bekämpfen: die `NullPointerException`.

---

1. Das Konzept: Ein Container für Werte

Ein `Optional<T>` ist ein **Container-Objekt**, das einen Wert vom Typ `T` entweder **enthält** oder **nicht enthält** (leer ist). Es macht im Typsystem explizit, dass ein Wert fehlen könnte, und zwingt den Entwickler, diesen Fall zu behandeln.

Klassisches Problem (vor Java 8): Methoden gaben oft `null` zurück, was zu `NullPointerExceptions` führte, wenn der Aufrufer eine `null`-Prüfung vergaß.

```
// Klassisch - gefährlich
Buch meinBuch = bibliothek.findeBuchNachTitel("Gibt es nicht");
System.out.println(meinBuch.getAutor()); // Wirft NullPointerException!



Moderne Lösung (seit Java 8): Die Methode gibt ein Optional<Buch> zurück.
Generated java


// Modern - sicher
Optional<Buch> optBuch = bibliothek.findeBuchNachTitel("Gibt es nicht");


2. Erstellen und Verwenden von Optional
Erstellung

    Optional.of(wert): Erstellt ein Optional mit wert. Wirft eine Exception, wenn
wert null ist.

    Optional.ofNullable(wert): Erstellt ein Optional mit wert oder ein leeres
Optional, wenn wert null ist. (Sicherer).

    Optional.empty(): Erstellt explizit ein leeres Optional.

Umgang mit dem Wert (Funktionaler Stil)

Der falsche Weg ist .get(), da es eine NoSuchElementException wirft, wenn das
Optional leer ist und somit den Sicherheitsvorteil zunichte macht.

Die richtigen Wege sind:

    Prüfen und Handeln:

        ifPresent(Consumer<T> action): Führt eine Aktion nur aus, wenn ein Wert
```

vorhanden ist.

ifPresentOrElse(Consumer<T> action, Runnable emptyAction): Führt eine von zwei Aktionen aus, je nachdem, ob ein Wert vorhanden ist oder nicht.

Standardwerte liefern:

orElse(T andererWert): Gibt den enthaltenen Wert oder den anderenWert zurück.

orElseGet(Supplier<T> supplier): Gibt den enthaltenen Wert oder das Ergebnis des Lambda-supplier zurück (wird nur bei leerem Optional ausgeführt).

orElseThrow(): Gibt den Wert zurück oder wirft eine NoSuchElementException.

Werte transformieren (Chaining):

map(Function<T, R> mapper): Wendet eine Funktion auf den enthaltenen Wert an und gibt das Ergebnis in einem neuen Optional zurück.

filter(Predicate<T> predicate): Gibt das Optional nur zurück, wenn sein Wert eine Bedingung erfüllt.

3. Vertiefung (JavaMasta's Profi-Tipps)
Optional vs. Leere Collections

Eine Methode, die eine Collection zurückgibt (z.B. List<Buch>), sollte bei keinem Ergebnis eine leere Collection zurückgeben, niemals null oder ein Optional<List<Buch>>. Dies erspart dem Aufrufer null-Prüfungen. Optional ist für singuläre Objekte gedacht.
Chaining mit flatMap

Wenn man Optional-Werte hat, die wiederum Optional-Werte enthalten (z.B. ein Optional<Benutzer>, der ein Optional<Adresse> hat), verwendet man flatMap, um die verschachtelten Optionals zu einer einzigen, flachen Ebene zu kombinieren.
Performance: orElse vs. orElseGet

orElse(methode()) führt methode() immer aus, auch wenn das Optional nicht leer ist. orElseGet(() -> methode()) führt das Lambda nur dann aus, wenn das Optional wirklich leer ist. Bei aufwändigen Standardwert-Erzeugungen ist orElseGet daher deutlich performanter.