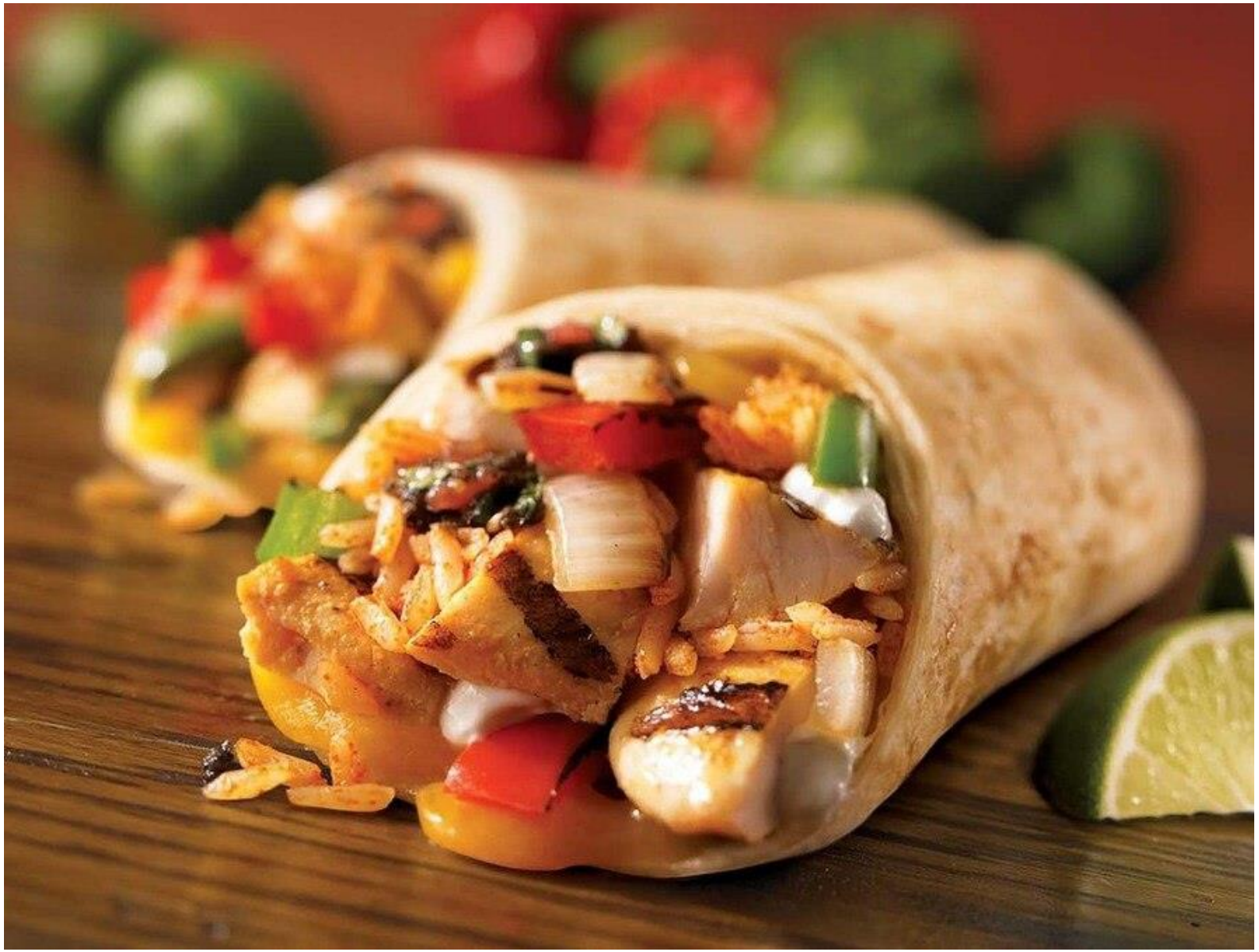


FUNKTIONALE PROGRAMMIERUNG

EMIL SCHLÄGER, MAXIM TAUSCH

AGENDA

1. Side Effects
2. Transparenz
3. Theoretische Grundlagen
4. Beispiel
5. FP in der Praxis

SIDE EFFECTS

SIDE EFFECTS

- Globale Zustandsänderungen
  - Können zu unerwartetem Verhalten führen
- Funktionen können Zustände außerhalb der Funktion ändern und externe Zustände als Input annehmen
- Wird bei objektorientierter Programmierung eingesetzt
- Funktionale Programmierung nutzt reine Funktionen

```java
static int counter = 6;

static void increment(int x) {
    counter += x;
}

public static void main(String[] args) {
    increment(1)
    System.out.println(counter); // Ausgabe: 7
}
```

SIDE EFFECTS

- Funktionale Programmierung vermeidet die Änderung oder Nutzung externer Zustände
- Reine Funktion: $f: A \to B, \quad x \mapsto y$
- Die Funktion nimmt keinen änderbaren Zustand an
- Die Funktion ändert nichts außerhalb der Funktion
- Anstatt einen Zustand zu ändern, erzeugt die Funktion einen neuen Zustand

```
static void increment(int x) {
    return x + 1
}

public static void main(String[] args) {
    int result = increment(6)
    System.out.println(result); // Ausgabe: 7
}
```

SIDE EFFECTS

- Funktionale Programmierung
  - Erlaubt keine globalen Mutationen
  - Betrachtet nur Funktionen
- Pur FP
  - Erlaubt gar keine Mutationen
  - for-loops nicht erlaubt, da Iterationsvariable inkrementiert werden muss

TRANSPARENZ

TRANSPARENZ

- Auswirkung einer Funktion ist bei OOP nicht sofort klar
  - Oft muss man die gesamte Klasse zuerst verstehen
- Bei FP gibt es einen klaren Input und Output
  - Auswirkung einer Funktion ist sofort sichtbar, da sie nichts außerhalb der Funktion ändert

TRANSPARENZ

```
static int counter = 6;

static void increment(int x) {
    counter += x;
}

public static void main(String[] args) {
    increment(1)
    System.out.println(counter); // Ausgabe: 7
}
```

```
static void increment(int x) {
    return x + 1
}

public static void main(String[] args) {
    int result = increment(6)
    System.out.println(result); // Ausgabe: 7
}
```

THEORETISCHE GRUNDLAGEN

(BZW: EIN ZEITLICH LIMITIERTER QUICK FIRE ÜBER EIN PAAR WICHTIGE BEGRIFFLICHKEITEN)

HIGHER-ORDER FUNCTION

= Funktion, die eine andere Funktion als Parameter nimmt oder zurück gibt

```
int applyFToX(int x, Function<Integer, Integer> f) {
    return f.apply(x);
}

int square(int x) {
    return x * x;
}

assert applyFToX(4, this::square) == 16;
```

ANONYME FUNKTION (LAMBDA FUNKTION)

= Funktion, die direkt an der Stelle initialisiert wird, wo sie genutzt wird

```
applyFToX(4, x -> x * x);
```

CURRYING

- Eine Funktion mit zwei Parametern kann interpretiert werden als eine HoF, wie folgt:

$$(A, B) \to C \quad \equiv \quad A \to (B \to C)$$

- Use Cases:
  - "Konstruktor" für Funktionen (s. Beispiel)
  - "Capturing" von Kontext

CURRYING - BEISPIEL

```
int power(int base, int exponent) {
    return (int) Math.pow(base, exponent);
}
```

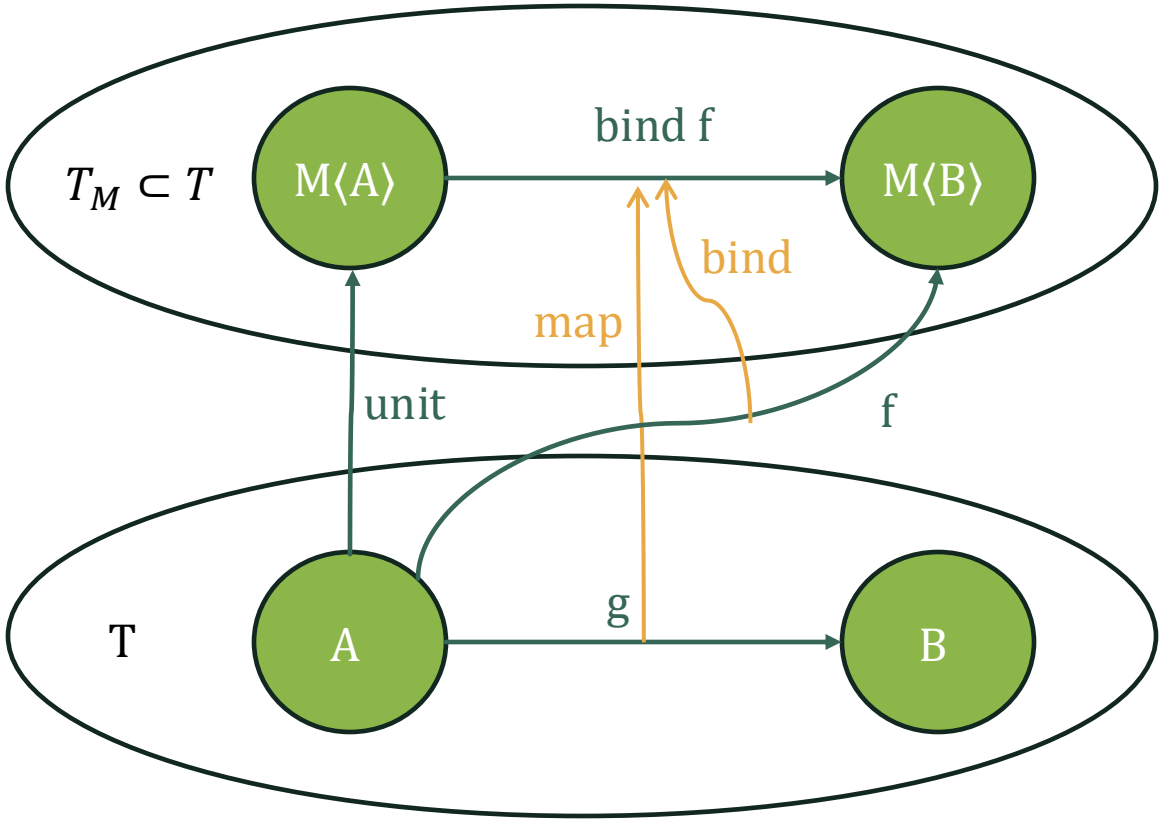
```
Function<Integer, Integer> cPower(int exponent) {
    return base -> (int) Math.pow(base, exponent);
}
```

```
Function<Integer, Integer> square = cPower(2);
Function<Integer, Integer> cube = cPower(3);

assert square.apply(2) == power(2, 2);
assert cube.apply(2) == power(2, 3);
```

MONADEN

- **Definition** (basierend auf Kleisli-Tripel)
- Ein Typ-Konstruktor M⟨T⟩ mit den folgenden assoziierten Funktionen:
  - unit: $\mathrm{T} \rightarrow \mathrm{M}\langle\mathrm{T}\rangle$
  - bind: $(M\langle A\rangle, \mathrm{f}: A \rightarrow M\langle B\rangle) \rightarrow M\langle B\rangle$
  - map: $(\mathrm{g}: \mathrm{A} \rightarrow \mathrm{B}) \rightarrow (\mathrm{M}\langle\mathrm{A}\rangle \rightarrow \mathrm{M}\langle\mathrm{B}\rangle)$
    - Zeigt Funktiorialität der Monade



CODE-BEISPIEL: MAYBE-MONADE

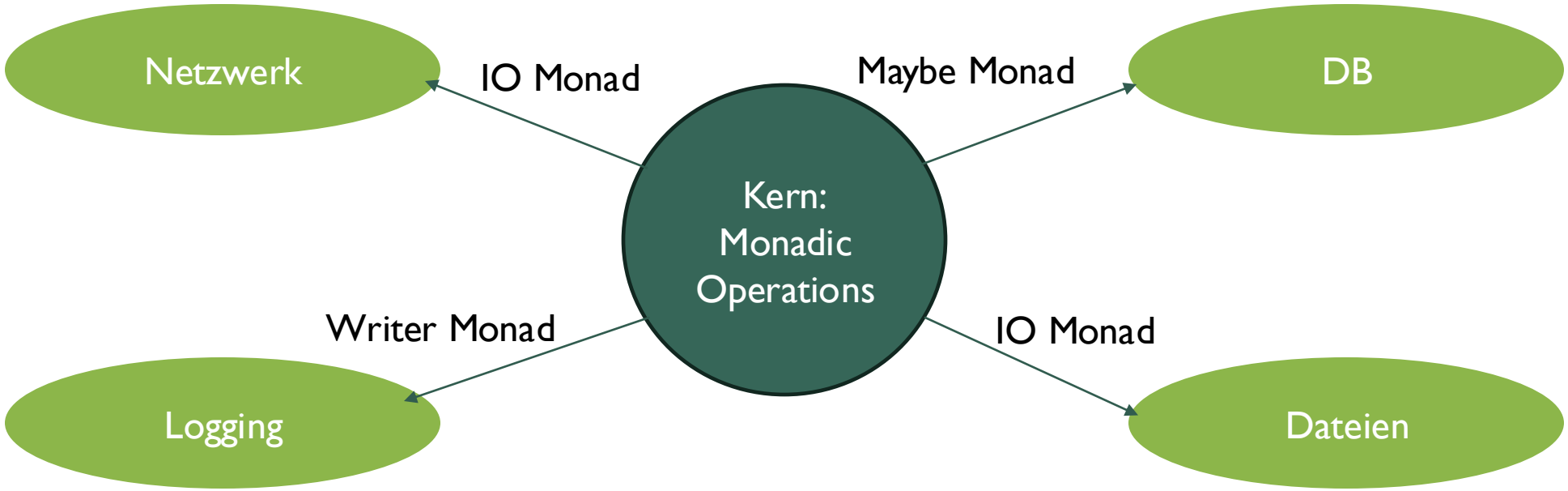
SIMPELSTE MONADE: ENKAPSULIERUNG DER MÖGLICHEN NICHT-EXISTENZ EINES WERTES

WEITERE BEKANNTE MONADEN

- **IO Monad**: Zukünftige Existenz eines Wertes
- **Writer Monad**: Aggregieren von Outputs (z.B. logs)
- **State Monad**: Enkapsulieren einer Funktion mit notwendigen Side Effects
- **List Monad:** Enkapsulieren von Operationen mit mehreren Ergebnissen
- **Result Monad:** Error Handling ohne Exceptions

FP IN DER PRAXIS

- Notwendige Side-Effects an den Rand auslagern (IO Monads)
- Dazwischen: Pur-Funktionale Operationen auf Monaden



QUELLEN

- B. Milewski, Category Theory for Programmers. 2019. Accessed: Dec. 08, 2025. [Online]. Available: http://www.blurb.com/b/9621951-category-theory-for-programmers-new-edition-hardco
- E. Moggi, "Notions of computation and monads," Information and Computation, vol. 93, no. 1, pp. 55–92, July 1991, doi: 10.1016/0890-5401(91)90052-4
- K. Slechten, "A gentle introduction to monads." Accessed: Nov. 04, 2025. [Online]. Available: https://kristofsl.medium.com/a-gentle-introduction-to-monads-bc583d41d95
- https://learn.microsoft.com/de-de/dotnet/standard/linq/functional-vs-imperative-programming

FRAGEN?