Funktionale Programmierung in Dart

Eine Einführung in Monaden, Lambda-Kalkül und sichere Programmierung mit Dart.

Funktionale Programmierung in Dart mit *fpdart*

Eine Einführung in Monaden, Lambda-Kalkül und sichere Programmierung mit Dart.

Dart ist primär objektorientiert, aber **fpdart** bringt funktionale Konzepte.

Ziel: Fehlerfreie, deklarative und elegante Programmierung.

Foto von Andrea Piacquadio: https://www.pexels.com/de-de/foto/foto-einer-frau-die-denkt-941555/

2. Grundlagen der funktionalen Programmierung (Recap)

Kernprinzipien:

- **Immutability**: Daten sind unveränderlich.
- **Pure Functions**: Keine Seiteneffekte.
- **Higher-Order Functions**: Funktionen als Eingabe und Ausgabe.

➔Paradigma, das auf mathematischen Konzepten basiert.

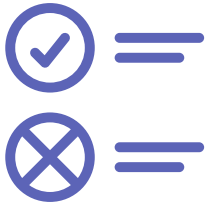
- **Lambda-Kalkül**: Grundlage der funktionalen Programmierung.
  - Beispiel: $(\lambda x.\ x + 1)(2) \rightarrow 3$
- **Monaden**: Verkapselung von Seiteneffekten (z. B. Fehlerbehandlung).

Einführung in **fpdart**

fpdart ist ein Paket, das funktionale Programmierkonzepte nach Dart bringt.

Task/TaskEither

Asynchrone Programmierung



Either

Funktionale Fehlerbehandlung.

Monaden

Elegantes Verkettungs-Handling.

Option

Umgang mit optionalen Werten.

Einführung in **fpdart**

Option | Either | Monaden | Task/TaskEither

```dart
Option<int> parseInt(String input) {
  final number = int.tryParse(input);
  return number ≠ null ? Option.of(number) : Option.none();
}
void main() {
  final result = parseInt("42").getOrElse(() ⇒ 0);
  print(result); // Output: 42
}
```

Option<T>

Enthält entweder einen Wert des T oder None für keinen Wert.

→ Vermeidung von Nullchecks

→ Eleganter Umgang mit fehlenden Werten

```dart
switch (option) {
  None() ⇒ 'None',
  Some(value: final a) ⇒ 'Some($a)',
}
```

Einführung in **fpdart**

Either | **Monaden** | **Task/TaskEither**

```dart
Either<String, int> divide(int a, int b) {
  return b == 0 ? Either.left("Cannot divide by zero") : Either.right(a ~/ b);
}
void main() {
  final result = divide(10, 2).match(
    (l) => "Error: $l",
    (r) => "Result: $r",
  );
  print(result); // Output: Result: 5
}
```

Either<L,R>

Enthält entweder einen Fehler des Typs **L**

oder

ein Ergebnis des Typs **R**

→ Klare und explizite Fehlerbehandlung ohne Ausnahmen.

Einführung in **fpdart**

Monaden | **Task/TaskEither**

Monaden sind abstrakte Datenstrukturen, die es ermöglichen, Daten in einem Kontext (z. B. optional, fehlerhaft, asynchron) sicher zu verarbeiten, ohne ihren ursprünglichen Zustand zu verändern.

```dart
// Funktion zur Validierung einer Benutzereingabe mit Fehler
Either<String, String> validateEmail(String email) {
  final emailRegex = RegExp(r"^[a-zA-Z0-9.]+@[a-zA-Z0-9]+\.[a-zA-Z]+");
  return emailRegex.hasMatch(email)
      ? Either.right(email)
      : Either.left("Invalid email format.");
}

// Funktion zur Speicherung der Benutzerdaten (Dummy)
Either<String, bool> saveUser(String email) {
  return Either.right(true);
}
```

--Kontext--
Left Error
Right Email

```dart
void main() {
  final userEmail = "user@example.com";

  final result = Either.right(userEmail)
      .flatMap(validateEmail)  // Schritt 1: Validierung
      .flatMap(saveUser);      // Schritt 2: Speichern

  result.match(
    (error) => print("Error: $error"),
    (success) => print("User saved successfully!"),
  );
}
```

Einführung in **fpdart**

Task/TaskEither

Task<T>: Asynchrone Berechnung, die bei Bedarf ausgeführt wird (lazy).
TaskEither<L,R>: Asynchrone Berechnung mit funktionaler Fehlerbehandlung (Left, Right).

Aspekt	**Future**	**Task (fpdart)**
Fehlerbehandlung	Durch Ausnahmen (try-catch)	Explizit durch Either (Left, Right)
Typensicherheit	Fehler sind untypisiert (Exception)	Fehler sind typisiert (Left)
Verkettung	Imperativ mit then() und catchError	Deklarativ mit map und flatMap
Lazy Evaluation	Berechnung startet sofort	Berechnung startet erst mit run()
Lesbarkeit	Schwer lesbar bei komplexen Abläufen	Klar und deklarativ

Einführung in **fpdart**

Task/TaskEither

Dart Future<int>

Error Handler

```dart
TaskEither<String, int> divideAsync(int a, int b) {
  return TaskEither.tryCatch(
    () async {
      if (b == 0) throw "Cannot divide by zero";
      return a ~/ b;
    },
    (error, _) => error.toString(),
  );
}
```

```dart
void main() async {
  final result = await divideAsync(10, 2)
      .flatMap((result1) => divideAsync(result1, 5))
      .run();

  result.match(
    (left) => print("Error: $left"),
    (right) => print("Result: $right"),
  );
}
```

Code Demo

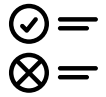
Code: https://inf-git.fh-rosenheim.de/studavrije7683/ws24-kp-avril/-/tree/main/docs/presentation/fp_dart-demo

Fragen & Feedback

Ressourcen

- Icons

  https://www.svgrepo.com/

  COLLECTION: Zwicon Line Icons
  LICENSE: CC Attribution License
  AUTHOR: zwicon

- Bilder

  https://www.pexels.com/

- Codedarstellung

  https://carbon.now.sh

- Quellen

  https://dart.dev/

  https://www.sandromaglione.com/articles/getting-started-with-fpdart-v1-functional-programming

  https://www.sandromaglione.com/articles/functional-programming-5-key-lessons-functional-programming-in-scala

  https://www.sandromaglione.com/articles/fpdart-functional-programming-in-dart-and-flutter

  https://www.sandromaglione.com/articles/option_type_and_null_safety_dart

  https://pub.dev/packages/fpdart