

# Dart

- Grundlegendes zu Flutter
- Dart

# Grundlegendes zu Flutter

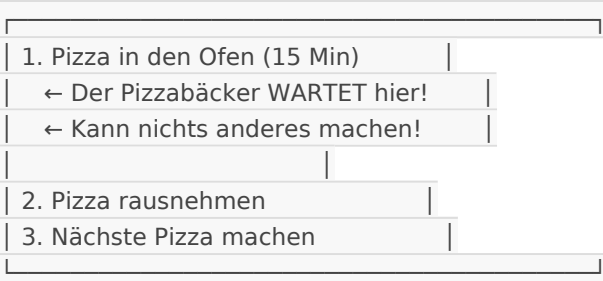
## ☐☐ Async/Await - Vollständige Erklärung

### Das Problem: Warum ist Dateiarbeit langsam?

Stell dir vor, du machst eine Pizza:

text

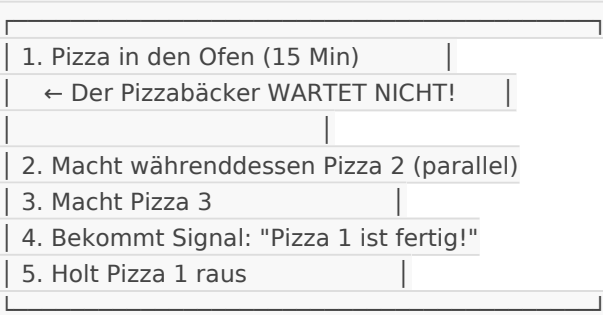
SYNCHRON (blockierend - FALSCH für Dateien):



Resultat: Sehr ineffizient!

text

ASYNCHRON (nicht-blockierend - RICHTIG für Dateien):



Resultat: Sehr effizient!

---

# Code-Beispiel: Sync vs Async

## ❑ SYNCHRON (Falsch für Dateien):

```
dart
void main() {
  print('Start');

  // Annahme: Datei braucht 2 Sekunden
  final file = File('data/users.json');
  final content = file.readAsStringSync(); // ← BLOCKIERT!

  print('Datei gelesen: $content');
  print('Fertig');
}
```

// Output:

// Start

// [warte 2 Sekunden... nichts läuft!]

// Datei gelesen: ...

// Fertig

// Die App war 2 Sekunden "eingefroren"!

**Problem:** Während die App wartet, kann sie NICHTS anderes machen!

---

## ❑ ASYNCHRON (Richtig für Dateien):

```
dart
void main() async { // ← main() wird async!
  print('Start');

  final file = File('data/users.json');
  final content = await file.readAsString(); // ← BLOCKIERT NICHT!
```

```
print('Datei gelesen: $content');
print('Fertig');
}
```

```
// Output:
// Start
// [warte 2 Sekunden... aber die App läuft weiter!]
// Datei gelesen: ...
// Fertig
```

```
// Die App war NICHT eingefroren!
```

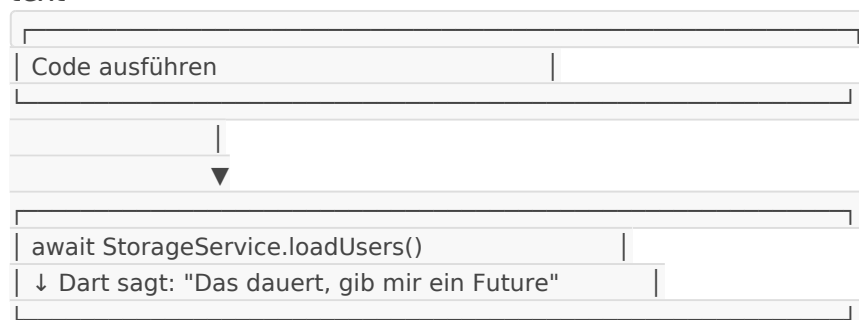
**Vorteil:** Die App kann während des Wartens andere Dinge machen!

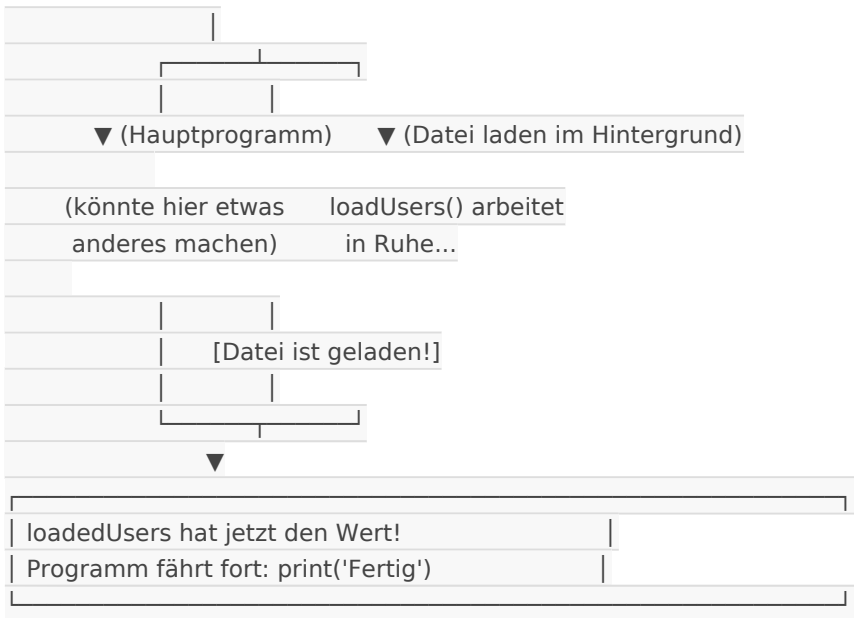
## ☐☐ Was bedeuten die Keywords?

Keyword	Bedeutung	Beispiel
<code>async</code>	"Diese Methode ist langsam, kann warten"	<code>Future&lt;String&gt; getData() async { ... }</code>
<code>await</code>	"Warte hier, bis Ergebnis kommt"	<code>final data = await loadFile();</code>
<code>Future&lt;T&gt;</code>	"Ein Ergebnis vom Typ T kommt später"	<code>Future&lt;int&gt; getAge() { ... }</code>

## ☐☐ Visualisierung: Der Fluss von Async/Await

text





# Praktisches Beispiel: Async testen

Erstelle `bin/async_demo.dart`:

```
dart
import 'dart:io';

// Beispiel 1: Synchroner Funktion (NICHT async)
void syncFunction() {
  print('Sync Start');
  sleep(Duration(seconds: 2)); // Blockiert die App!
  print('Sync Ende (nach 2 Sekunden)');
}

// Beispiel 2: Asynchrone Funktion (mit async/await)
Future<void> asyncFunction() async {
  print('Async Start');
  await Future.delayed(Duration(seconds: 2)); // Blockiert NICHT!
  print('Async Ende (nach 2 Sekunden)');
}

// Beispiel 3: Mehrere Async-Funktionen (parallel!)
Future<void> multipleAsync() async {
  print('\n=== Mehrere Async-Tasks parallel ===');

  // Alle 3 starten gleichzeitig!
  final task1 = doTask('Task 1', 1);
  final task2 = doTask('Task 2', 2);
```

```

final task3 = doTask('Task 3', 3);

// Warte auf alle
await Future.wait([task1, task2, task3]);

print('Alle Tasks fertig!');
}

Future<void> doTask(String name, int seconds) async {
  print('$name startet...');
  await Future.delayed(Duration(seconds: seconds));
  print('$name fertig!');
}

void main() async {
  print('=== SYNCHRON (blockierend) ===');
  syncFunction(); // Freezt die App!

  print('\n=== ASYNCHRON (nicht-blockierend) ===');
  await asyncFunction(); // App läuft weiter!

  await multipleAsync();
}

```

### Starte das Programm:

bash

```
dart bin/async_demo.dart
```

### Erwartete Ausgabe:

text

```

=== SYNCHRON (blockierend) ===
Sync Start
Sync Ende (nach 2 Sekunden)

=== ASYNCHRON (nicht-blockierend) ===
Async Start
Async Ende (nach 2 Sekunden)

=== Mehrere Async-Tasks parallel ===
Task 1 startet...
Task 2 startet...
Task 3 startet...
Task 3 fertig!
Task 2 fertig!
Task 1 fertig!
Alle Tasks fertig!

```

**Beobachtung:** Task 3 endet zuerst (nur 1 Sekunde), obwohl sie nicht zuerst startete! Das zeigt, dass sie **parallel** laufen! ☐☐

---

# ☐ Die 3 Zustände eines Future

Ein `Future` hat immer einen von 3 Zuständen:

text

```
Future Status
1. PENDING (wartet noch)
   await loadUsers()
   ↓ (Datei wird gelesen...)
2. COMPLETED (fertig!)
   ↓ (Datei komplett gelesen)
   loadedUsers = [User(...), ...]
3. ERROR (Fehler!)
   ↓ (Datei nicht gefunden)
   catch (e) { print(e); }
```

## ⚠️ Wichtige Regeln:

**Regel 1:** Du kannst **nur** `await` **innerhalb von** `async` **Funktionen** nutzen!

dart

```
// ☐ RICHTIG:
Future<void> myFunction() async {
  await Future.delayed(Duration(seconds: 1));
}
```

// ☐ FALSCH:

```
void myFunction() {
  await Future.delayed(Duration(seconds: 1)); // ERROR!
}
```

**Regel 2:** Wenn du `await` brauchst, muss die Funktion `async` sein!

dart

```
// ☐ RICHTIG:
void main() async { // ← main() wird async!
  await StorageService.loadUsers();
}
```

```
// ❌ FALSCH:  
void main() {  
  await StorageService.loadUsers(); // ERROR!  
}
```

**Regel 3:** `async` macht deine Funktion automatisch zu einem `Future` !

```
Future<String> getData() async { // ← Gibt automatisch Future<String> zurück!  
  return "Daten"; } // Ist gleichbedeutend mit:  
  
Future<String> getData2() { // ← Musst du manuell Future sagen  
  return Future.value("Daten"); }
```

## ☐ Kurze Analogie:

Synchron:

Du: "Barista, mach mir einen Kaffee"

Barista: "Ja, warte hier... [5 Minuten warten]... Fertig!"

Du: "Okay, danke!" ← Du konntest nur herumsitzen!

Asynchron:

Du: "Barista, mach mir einen Kaffee" (mit Future-Ticket)

Barista: "Ja, kein Problem, ich ruf dich auf!"

Du: "Gut, ich setz mich hin und lies Zeitung" ← Du machst etwas anderes! [5 Minuten später]

Barista: "Dein Kaffee ist fertig!" (Future resolved!)

Du: "Danke!" ← Du warst nicht blockiert!

## Methoden und Klassen

Eine typische Funktion in Dart besteht aus dem Rückgabewert, dem Namen, optionalen Parametern und dem Funktionskörper.

# Aufbau einer Funktion

dart

Copy code

```
Rueckgabewert funktionsName(Parameter) {  
  // Funktionskörper  
  return wert; // nur wenn Rueckgabewert nicht void ist  
}
```

Beispiel:

dart

Copy code

```
int addiere(int a, int b) {  
  return a + b;  
}
```

# Aufbau einer Klasse

Eine Klasse bündelt Werte (Felder), einen Konstruktor und Methoden.

dart


Copy code

```
class Auto {  
  // Felder  
  final String marke;  
  int kilometerstand;  
  
  // Konstruktor  
  Auto(this.marke, this.kilometerstand);  
  
  // Methode  
  void fahre(int kilometer) {  
    kilometerstand += kilometer;  
  }  
}
```

```
// Eine Methode mit Rückgabewert
String beschreibung() {
    return 'Marke: $marke, Kilometerstand: $kilometerstand';
}
}
```

# Nutzung der Klasse

dart

Copy code 

```
void main() {
    final auto = Auto('Tesla', 0); // Konstruktor
    auto.fahre(120);               // Methode
    print(auto.beschreibung());    // Methode mit Rückgabewert
}
```

So siehst du den typischen Aufbau: Felder speichern den Zustand, der Konstruktor initialisiert ihn, und Methoden ändern oder lesen ihn aus.

# Dart

Bei einem Dart Programm Argument auf Konsolenebene mitgeben:

```
void main(List<String> arguments) {  
  print('Hello world: ${notepad_app.calculate()}!');  
}
```

arguments ist eine liste die Argumente entgegennimmt von außen.

Das bedeutet konkret:

- Wenn du dein Dart-Programm in der Konsole z.B. startest mit `dart run meinprogramm.dart arg1 arg2 arg3`, dann ist arguments eine Liste mit den Strings `["arg1", "arg2", "arg3"]`.
- Du kannst diese Argumente nutzen, um das Verhalten deines Programms dynamisch zu steuern, z.B. verschiedene Modi, Dateinamen oder Einstellungen von außen zu übergeben.
- Wenn du aber keine Argumente übergibst, ist die Liste einfach leer.