Swift Concurrency

…und unser aller Erfahrung damit.

Kurze Einführung

- Concurrency:
  - Asynchrone Ausführung
  - Parallele Ausführung
- Wesentliche Konzepte:
  - async/await, Tasks, Actors
  - Continuations, Executors

async/await

- Asynchrone Funktionen:

```
func download(from: URL) async -> FileWrapper
```

- Aufrufbar in anderen Async-Funktionen (oder Main):

```
let file = await download(…)
```

- Await: Wartet an sog. Suspension Point
  - Anderer Async-Code wird evtl. ausgeführt
  - Invarianten könnten sich ändern!

async/await: Parallel

```
async let fileA = download(…)
async let fileB = download(…)
async let fileC = download(…)

let all = await [fileA, fileB, fileC]
```

- Parallele Ausführung
- Async-Expressions werden ohne *await* abgebrochen

async/await: Thread Pool

- Threads nicht sichtbar
- Intern: Fester Thread Pool (#Cores)
- Verhindert Thread-Explosion und CPU-Überlastung, aber:
  - (Zu viel) Blockierende I/O vermeiden
  - Lange synchrone Berechnungen: **Task.yield** nutzen
  - Starvation durch Ausschöpfen des Pools möglich
- Annahmen über Threads/Queues vermeiden (z.B. Reentrante Locks)

async/await (Un-)learings

- Scheinbares Busy Waiting völlig ok:

```
try await Task.sleep(.seconds(3))
```

Tasks

- Tasks sind keine Threads 🤓
- "Lebenslinie" einer kompletten Operation
- Operation: asynchron und/oder parallel
- Rückgabewert + Cancellation-Status

```
let someTask = Task {
    return await doSomething()
}

let result = await someTask.value()
```

Structured vs. Unstructured

- Structured Tasks:
  - Sub-Tasks zur Paralellisierung / Queuing via Task Group
  - Sub-Tasks erben u.a.:
    Task Local Storage, Cancellation-Status, Prioritäten, Actor-Isolation

- Unstructured Task: "Neue" Lebenslinie, selber Actor
  - Detached Task: Separater Actor

Actors

- Reference Type ohne Vererbung
- Swift stellt sicher:
  - Immer nur ein Task gleichzeitig pro Actor
  - Aufrufe außerhalb des Actors alle *async*
  - Sendable-Types für Argumente, Rückgabewerte etc.
- Spezieller Actor: @MainActor
- *nonisolated*

Sendable Types

- Problem: Ungeschützte Ref-Type Argumente / Return Values
- Sendable Types:
  - Erforderlich für Argumente/Rückgabewerte (ab Swift 6)
  - Value Types & @MainActor-Types per Default Sendable (meistens)
  - Reference Types:
    Erfordern *Sendable*-Protokoll (+ sichere Implementerung)

Actor Reentrancy

```
actor BackAccount {
    var balance: Int = 100

    func authorize() async { … }

    func withdraw(amount: Int) async {
        guard balance > 0 else { return }
        guard await authorize() { return }
        balance -= amount
    }
}
```

Zwischenzeitlicher withdraw-Aufruf:

balance > 0 gilt nicht mehr

Executor

- **Job**: Einzelner synchroner Abschnitt in einem Task
- **Executor:** Führt (irgendwie) Jobs aus
- **SerialExecutor:** Führt Jobs sequenziell aus (Actors)
  - Switching: Thread wird beibehalten
- Builtins:
  - MainActorExecutor
  - Default Concurrent Executor: Thread-Pool
  - Actor Executors: *SerialExecutor* auf Basis *Default Concurrent Executor*

Continuations

- Um async/await-APIs für Legacy Code zu bauen:

```
result = await withCheckedContinuation { continuation in
    DispatchQueue.main.async {
        ...
        continuation.resume()
    }
}
```

- *Checked Continuation:* Runtime-Checks gegen fehlendes/doppeltes Resume
- *Unsafe Continuation*: Ohne Check

AsyncSequence

- Datenstrom von asynchron erzeugten Werten:

```
for await x in stream {
    ...
}
```

- Gegenseite liefert Werte über continuation.yield()

AsyncSequence

- Verhältnis zu Combine:
  - Effektiv nur ein "Subscriber" gleichzeitig (ohne Runtime-Check)
  - Unterschiedliche Buffering-Policies
  - Eher für dauerhafte Producer/Consumer
  - Sequence bleibt retained
  - Umwandlung Publisher <> AsyncSequence
    (Probleme mit Actors!)