

MATHEMA

Non-blocking Code in Kotlin

Nebenläufigkeit mit Koroutinen und Flows

Werner Eberling

E-Mail: werner.eberling@mathema.de

Twitter: @Wer_Eb



KKON

Der Sprecher



Werner Eberling

Principal Consultant / Autor

Email: werner.eberling@mathema.de

Mastodon: [@Wer_Eb@social.dev-wiki.de](https://social.dev-wiki.de/@Wer_Eb)

X (fka. Twitter): [@Wer_Eb](https://twitter.com/Wer_Eb)



Was haben wir vor?

- Grundlagen
- Suspending Functions
- Scopes & Contexts
- Jobs & Cancelation
- Async/await
- Asynchronous Flows
- Channels
- Ausblick: Weiterführende Themen



Und vorallem

...

```
<!-- Navbar  
<div class="w3-top">  
  <div class="w3-bar w3-black w3-hide-large w3-right" href="javascript:void(0)" title="Toggle Navigation Menu"><i class="fa fa-bars">HOM  
  <a href="#" class="w3-bar-item w3-button w3-padding-large">BAND</a>  
  <a href="#band" class="w3-bar-item w3-button w3-padding-large">TOUR</a>  
  <a href="#tour" class="w3-bar-item w3-button w3-padding-large w3-hide-small">CONTACT</a>  
<div class="w3-dropdown-hover w3-hide-small">  
  <button class="w3-padding-large w3-button" title="More">MORE  
    fa fa-caret-down"></i></button>  
<div class="w3-dropdown-content w3-bar-block">  
  <a href="#" class="w3-bar-item w3-button w3-padding-large">
```


Fakten, Fakten, Fakten...

The Colorful History of the Internet and its Increasingly Problematic Future

A number of factors have pushed the development of the primary and secondary industries of today. Transistors, microprocessors, governmentally regulated institutions, technical institutions, academies, and children. Not single factor produced the World Wide Web that today connects so many global networks. Continued marketing exists in hardware (the mechanical parts comprising the computer network) and software (the programming language running the network) and doing the necessary (computational) operation have taken place, and these developments have had huge social and economic impacts on society.

The 1940s, 1950s, and 1960s. In the late 1940s and 1950s, when computers were made with vacuum tubes and occupied more than 90 square meters of space, there was indeed a limit to how big they could be. There were other problems with these early computers. The vacuum tubes could leak, the metal that emitted electrons in the vacuum tubes burned out, and all of these tubes required tons of power to run. Major improvements came in computer hardware technology with the development of transistors, invented by John Bardeen and Walter Brattain in 1947 and 1948 at Bell Telephone Laboratories, and the development of integrated circuits, invented by Jack Kilby at Texas Instruments and Robert Noyce at Fairchild Camera in 1959 and 1959. Integrated circuits resolved a number of vacuum tube technology's problems and did much to further the development of smaller computers with greater power. With this improved technology, instead of making transistors one by one, several transistors could be made at the same time and on the same piece of semiconductor. In addition to transistors, other electric components such as resistors, capacitors, and diodes could be made using the same process and materials (Flavland, 2003).

It is important to note that since the 1960s, the number of transistors per unit area has been doubling every one-and-a-half years—thus increasing computing power. This amazing progression of circuit fabrication is called Moore's Law, named after Gordon Moore, a pioneer in the integrated circuit field and founder of the Intel Corporation (ibid.).

Warum Kotlin?

- Soll eine moderne Alternative zu Java sein
- Syntax zu Java nicht kompatibel
- Interoperabilität sehr wichtig

Was ist Kotlin?

- Statisch typisiert
- Unterstützt sowohl objektorientierte, als auch funktionale Programmierparadigmen
- Läuft in der JVM
- Kann nach JavaScript transpiliert werden
- Unter Android bevorzugte Sprache
- Kann LLVM nutzen (Kotlin/Native)

Entwicklungsumgebungen

- IntelliJ
- Android Studio
- Eclipse
- Kommandozeile mit Standalone-Compiler
- Online (play.kotlinlang.org)

Der Klassiker (in Java)

```
public class HelloWorld {  
  
    public static void main(String[] args) {  
        System.out.println("Hallo KKON!");  
    }  
  
}
```

- Ganz schön viel Overhead für ein kleines „*Hello World!*“, oder?

Und jetzt in Kotlin...

```
fun main() = println("Hallo KKON!")
```

- Beliebige Datei (CamelCase) mit Endung .kt
- `main` ist Toplevel-Funktion
- Keine Strichpunkte nötig ;)

Beispiele und Materialien

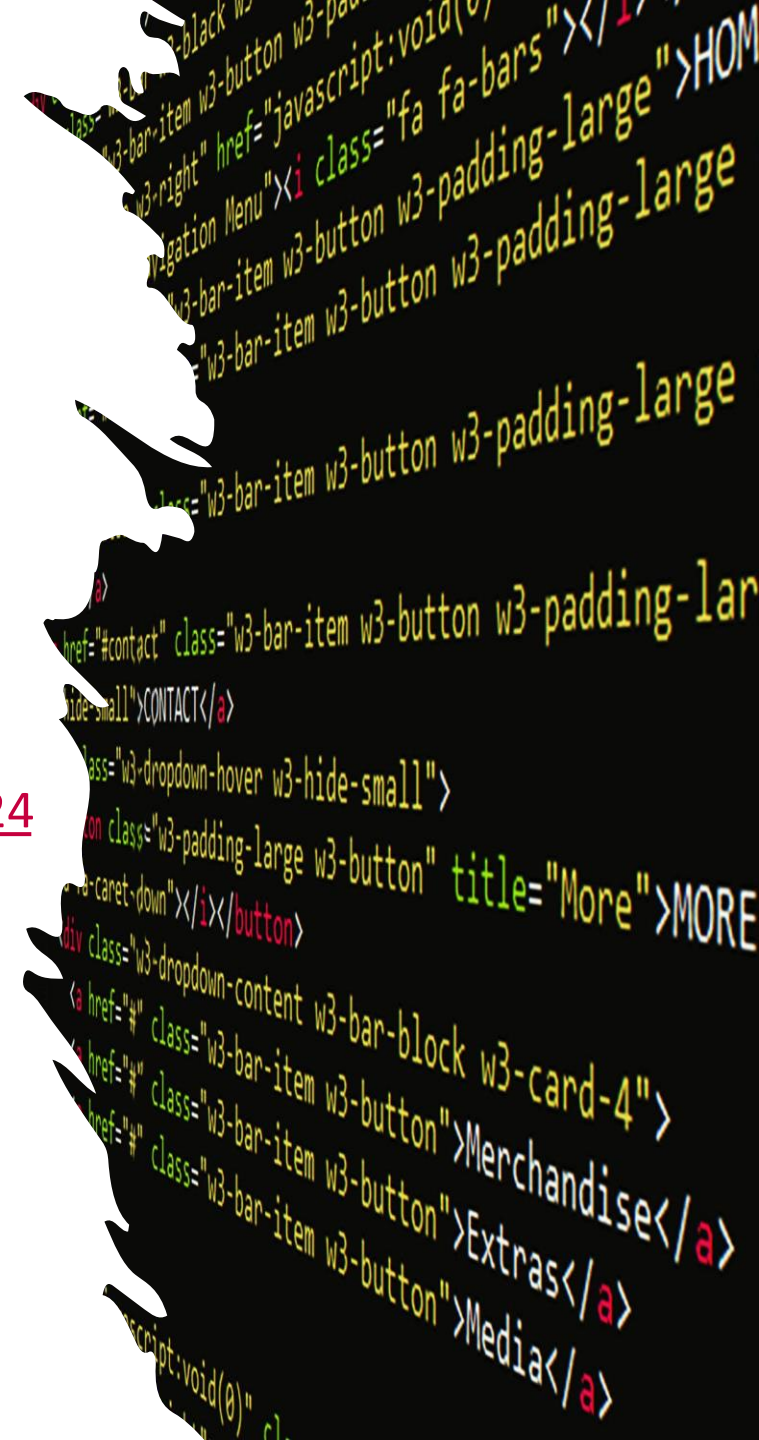
git clone

<https://github.com/wern/kotlin-non-blocking-coroutines-flows-ws-kkon-2024>

Alternativ per Download:

<https://tinyurl.com/ws-kotlin-kkon24>

Passwort: Kcrf-KKON2024



Nebenläufigkeit mit Koroutinen

Nebenläufigkeit in Java und in Kotlin

```
public class BlockingDemo {  
  
    public static void main(String[] args) {  
        Thread t = new Thread(() -> {  
            try {  
                Thread.sleep(1000);  
                System.out.println("KKON!");  
            } catch (InterruptedException e) {}  
        });  
        t.start();  
        System.out.print("Willkommen zur ");  
        try {  
            t.join();  
        } catch (InterruptedException e) {}  
    }  
}
```

Java

```
fun main() = runBlocking {  
    launch {  
        delay(1000L)  
        println("KKON!")  
    }  
    println("Willkommen zur")  
}
```

Kotlin

Benötigt
org.jetbrains.kotlinx:kotlinx-coroutines-core:1.6.21

Koroutinen in wenigen Worten

```
fun main() = runBlocking {  
    launch {  
        delay(1000L)  
        println("KKON!")  
    }  
    println("Willkommen zur")  
}
```

- Koroutinen können unterbrochen und wieder fortgesetzt werden
- Kotlin nutzt für die Ausführung „irgendwie“ Threads

Koroutinen – Suspending Functions

```
fun main() = runBlocking {  
    launch {  
        delayedKKON()  
    }  
    println("Willkommen zur")  
}  
  
suspend fun delayedKKON() {  
    delay(1000L)  
    println("KKON!")  
}
```

- *Suspending Functions* können nur in Koroutinen verwendet werden
 - Kapseln Verhalten wie „normale“ Funktionen
 - Können den Prozessor aufgeben
 - Können weitere *Suspending Functions* rufen (z.B. `delay()`)

Hands-on

```
<!-- Navbar -->
<div class="w3-top">
  <a class="w3-bar-item w3-button w3-padding-large w3-hide-large w3-right" href="javascript:void(0)" title="Toggle Navigation Menu"><i class="fa fa-bars"></i></a>
  <a href="#" class="w3-bar-item w3-button w3-padding-large" title="HOME">HOME</a>
  <a href="#band" class="w3-bar-item w3-button w3-padding-large" title="BAND">BAND</a>
  <a href="#tour" class="w3-bar-item w3-button w3-padding-large" title="TOUR">TOUR</a>
  <a href="#contact" class="w3-bar-item w3-button w3-padding-large w3-hide-small" title="CONTACT">CONTACT</a>
  <div class="w3-dropdown-hover w3-hide-small">
    <button class="w3-padding-large w3-button" title="More"><i class="fa fa-caret-down"></i></button>
    <div class="w3-dropdown-content w3-bar-block">
      <a href="#" class="w3-bar-item w3-button w3-padding-large" title="More">MORE</a>
    </div>
  </div>
</div>
```



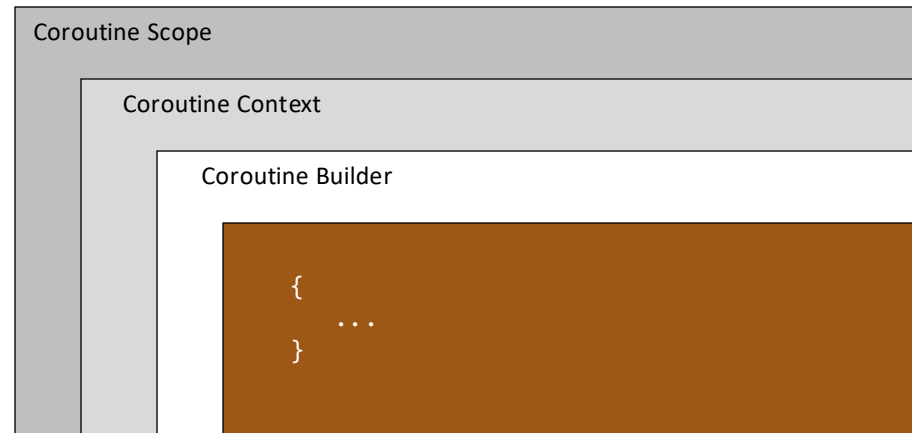

Strukturierte Nebenläufigkeit

Strukturierte Nebenläufigkeit - Scopes

```
fun main() = runBlocking {  
    launch {  
        delay(1000L)  
        println("KKON!")  
    }  
    println("Willkommen zur")  
}
```

- Koroutinen laufen immer innerhalb eines Scopes
Start u.a. mit `runBlocking{...}` oder `coroutineScope{...}`
- `runBlocking` als Brücke zwischen „normalem“ und „suspendable“ Code
- `coroutineScope` innerhalb von *Suspending Functions*

Strukturierte Nebenläufigkeit



- Scopes

Beschränkung der Laufzeit der Koroutinen (nur innerhalb des Scopes)

Keine „wild laufenden“ Threads (auch nach Fehlern)

- Contexts

Ablaufkontext der Koroutine (Welcher Thread arbeitet die Koroutine ab?)

- Builder

Erzeugung konkreter Koroutinen

Koroutinen – Parallele Ausführung

```
fun main() = runBlocking {  
    launch {  
        delayedKKONWithNewJobs()  
    }  
    println("Willkommen zur")  
}  
suspend fun delayedKKONWithNewJobs() = coroutineScope {  
    launch {  
        delay(2000L)  
        println("2022")  
    }  
    launch {  
        delay(1000L)  
        println("digital")  
    }  
    println("KKON")  
}
```

- Neuer Scope mit zwei parallelen Koroutinen

Innerer Scope endet erst, wenn beide Koroutinen beendet wurden!

Koroutinen – Was passiert bei Fehlern?

```
fun main() = runBlocking {  
    launch {  
        delayedKKONWithNewJobs()  
    }  
    println("Willkommen zur")  
}  
suspend fun delayedKKONWithNewJobs() = coroutineScope {  
    launch {  
        delay(2000L)  
        println("2022")  
    }  
    launch {  
        delay(1000L)  
        throw RuntimeException("Bang!")  
    }  
    println("KKON")  
}
```

- Unbehandelte Fehler führen zum Abbruch des Scopes und aller enthaltenen Koroutinen (=> Strukturierte Nebenläufigkeit)

Hands-on

```
<!-- Navbar -->
<div class="w3-top">
  <a class="w3-bar w3-black w3-hide-large w3-right" href="javascript:void(0)" title="Toggle Navigation Menu"><i class="fa fa-bars"></i></a>
  <a href="#" class="w3-bar-item w3-button w3-padding-large">HOME</a>
  <a href="#band" class="w3-bar-item w3-button w3-padding-large">BAND</a>
  <a href="#tour" class="w3-bar-item w3-button w3-padding-large">TOUR</a>
  <a href="#contact" class="w3-bar-item w3-button w3-padding-large w3-hide-small">CONTACT</a>
  <div class="w3-dropdown-hover w3-hide-small">
    <button class="w3-padding-large w3-button" title="More">MORE</button>
    <div class="w3-dropdown-content w3-bar-block">
      <a href="#" class="w3-bar-item w3-button w3-padding-large">...</a>
    </div>
  </div>
</div>
```




Dispatcher

Dispatcher – Wer macht eigentlich die Arbeit?

```
fun main() = runBlocking {  
    launch {  
        delayedKKON()  
    }  
    printlnWithThreadInfo("Willkommen zur")  
}  
  
suspend fun delayedKKON() {  
    delay(1000L)  
    printlnWithThreadInfo("KKON!")  
}
```

- *Coroutine Dispatcher* definieren in welchem Thread-Kontext eine Koroutine abläuft

Wird normalerweise vom umgebenden Scope übernommen (hier: Main-Thread)

Dispatcher – Wer macht eigentlich die Arbeit?

```
fun main() = runBlocking {  
    launch(Dispatchers.Default){  
        delayedKKON()  
    }  
    printlnWithThreadInfo("Willkommen zur")  
}  
  
suspend fun delayedKKON() {  
    delay(1000L)  
    printlnWithThreadInfo("KKON!")  
}
```

- Beim Bau der Koroutine kann der gewünschte Dispatcher explizit gesetzt werden

Dispatcher – Mögliche Optionen

- Das Dispatchers Objekt definiert die standardmäßig verfügbaren Dispatcher

Default: Nutzung des Default Thread Pools

IO: Spezieller Dispatcher für Aufgaben mit Blocking IO

Main: Ablauf im UI-Main Thread (nur mit entsprechender Library Unterstützung)

Unconfined: Keine Bindung an einen konkreten Thread/Thread-Pool

- Zusätzlich kann auch explizit neuer Context mit eigenem Thread gestartet werden

```
launch(newSingleThreadContext("MyOwnThread")) { ... }
```

Dispatcher – Explizite Context-Wechsel

```
fun main() = runBlocking {  
    val outerContext = coroutineContext  
    launch(Dispatchers.Default) {  
        delay(1000L)  
        printlnWithThreadInfo("Doing my work...")  
        withContext(outerContext) {  
            printlnWithThreadInfo("KKON!")  
        }  
    }  
    printlnWithThreadInfo("Willkommen zur")  
}
```

- Beim Aufruf einer *Suspending Function* kann der *Context* mittel `withContext()` explizit gewechselt werden
 Aufruf erfolgt dann in einem anderen Thread

Hands-on

```
<!-- Navbar -->
<div class="w3-top">
  <a class="w3-bar w3-black w3-hide-large w3-right" href="javascript:void(0)" title="Toggle Navigation Menu"><i class="fa fa-bars"></i></a>
  <a href="#" class="w3-bar-item w3-button w3-padding-large">HOME</a>
  <a href="#band" class="w3-bar-item w3-button w3-padding-large">BAND</a>
  <a href="#tour" class="w3-bar-item w3-button w3-padding-large">TOUR</a>
  <a href="#contact" class="w3-bar-item w3-button w3-padding-large w3-hide-small">CONTACT</a>
  <div class="w3-dropdown-hover w3-hide-small">
    <button class="w3-padding-large w3-button" title="More">MORE</button>
    <div class="w3-dropdown-content w3-bar-block">
      <a href="#" class="w3-bar-item w3-button w3-padding-large">...</a>
    </div>
  </div>
</div>
```


Kontrolle mit Jobs



"Close-up of a man holding remote control of robotic vacuum cleaner to start cleaning" by Marco Verch is licensed with CC BY 2.0.

To view a copy of this license, visit <https://creativecommons.org/licenses/by/2.0/>

Link to original photography: <https://foto.wuestenigel.com/close-up-of-a-man-holding-remote-control-of-robotic-vacuum-cleaner-to-start-cleaning/>

Jobs – Wenn es „expliziter“ sein soll

```
fun main() = runBlocking {  
    val job = launch {  
        delayedKKON()  
    }  
    println("Willkommen zur")  
    job.join()  
    println("2022")  
}  
suspend fun delayedKKON() {  
    delay(1000L)  
    println("KKON")  
}
```

- launch liefert bei der Erzeugung der Koroutine ein Job Objekt zurück
 - Ermöglicht Abfrage des Status der Koroutine
 - Kann zum expliziten Warten auf die Beendigung der Koroutine genutzt werden

Cancelation – den Job wieder abbrechen

```
fun main() = runBlocking {  
    val job = launch {  
        delayedKKON()  
    }  
    println("Willkommen zum")  
    delay(10L)  
    job.cancel()  
    println("Heute nicht ;)")  
}
```

```
suspend fun delayedKKON() {  
    delay(1000L)  
    println("KKON")  
}
```

- Das Job Objekt ermöglicht auch den Abbruch der Koroutine
 - Bricht auch alle „Subjobs“ ab
 - Benötigt die „Kooperation“ der Koroutine
 - Alle suspending Library Funktionen sind „cancelable“
 - Eigener Code muss den Context Status abfragen (`CoroutineScope.isActive`)

Cancellation – auf den Abbruch reagieren

```
fun main() = runBlocking {  
    val job = launch { //... }  
    //...  
    job.cancel()  
}  
suspend fun delayedKKON() {  
    try {  
        delay(1000L)  
        println("KKON")  
    } catch (e : CancellationException) {  
        println("Abbruch KKON :(")  
    }  
}
```

- Library Funktionen werfen `CancellationException` im Falle eines Abbruchs
Gilt bei Nicht-Abfangen als „normales“ Verhalten der Koroutine

Timeouts – Abbruch auch ganz automatisch

```
fun main() = runBlocking {  
    withTimeout(100L) {  
        delayedKKON()  
    }  
}  
  
suspend fun delayedKKON() {  
    try {  
        delay(1000L)  
        println("KKON")  
    } catch (e : CancellationException) {  
        println("Abbruch KKON :(")  
    }  
}
```

- Beim Erzeugen via `withTimeout()` kann einer Koroutine eine max. Laufzeit angegeben werden, nach der diese abgebrochen wird

Hands-on

```
<!-- Navbar -->
<div class="w3-top">
  <a class="w3-bar w3-black w3-hide-large w3-right" href="javascript:void(0)" title="Toggle Navigation Menu"><i class="fa fa-bars"></i></a>
  <a href="#" class="w3-bar-item w3-button w3-padding-large">HOME</a>
  <a href="#band" class="w3-bar-item w3-button w3-padding-large">BAND</a>
  <a href="#tour" class="w3-bar-item w3-button w3-padding-large">TOUR</a>
  <a href="#contact" class="w3-bar-item w3-button w3-padding-large w3-hide-small">CONTACT</a>
  <div class="w3-dropdown-hover w3-hide-small">
    <button class="w3-padding-large w3-button" title="More">MORE</button>
    <div class="w3-dropdown-content w3-bar-block">
      <a href="#" class="w3-bar-item w3-button w3-padding-large">...</a>
    </div>
  </div>
</div>
```




Ergebnisse zurückliefern

Rückgabewerte mit `async` und `await`

```
fun main() = runBlocking {  
    val audience = async { countAudience() }  
    val presenter = async { countPresenter() }  
    println("${audience.await()} Zuhörer und ${presenter.await()} Sprecher anwesend.")  
}  
suspend fun countAudience() : Int {  
    delay(1000L)  
    return 14  
}  
suspend fun countPresenter() : Int {  
    delay(1000L)  
    return 1  
}
```

- `async` erzeugt eine Koroutine, ähnlich wie `launch`
Statt eines Jobs wird der Rückgabewert als `Deferred` zurückgegeben
- Mittels `await` kann auf das Beenden der Koroutine und den Rückgabewerte gewartet werden

Mehrere Werte zurückgeben

```
fun main() = runBlocking {  
    println("Teilnehmer:")  
    val participants = async { participantSuspendList() }  
    launch {  
        // Start paralleler Verarbeitungen  
    }  
    participants.await().forEach { println(it) }  
}  
  
suspend fun participantSuspendList() : List<String> {  
    participantDatabase.forEach {  
        delay(100L) // Komplexe Verarbeitung der Eintraege  
    }  
    return participantDatabase  
}
```

- Asynchrone Aufrufe können auch mehrere Ergebnisse zurückliefern
- Mittels `await` wird aber auf das Gesamtergebnis gewartet
Reduziert evtl. die Parallelität der Ergebnisverarbeitung
Werte können nicht „so früh wie möglich“ verarbeitet werden

Werte parallel Erzeugen mit asynchronen Flows

```
fun main() = runBlocking {  
    println("Teilnehmer:")  
    launch {  
        // Start paralleler Verarbeitungen  
    }  
    participantFlow().collect(){ println(it) }  
}  
  
fun participantFlow() = flow {  
    participantDatabase.forEach {  
        delay(1000L) // aufwendige Datenverarbeitung  
        emit(it) // Ausgabe des naechsten Wertes  
    }  
}
```

- `flow` erzeugt eine asynchronen Flow

Der *Flow Builder* ist **keine** *Suspending Function*

Ein *Flow* kann *Suspending Functions* aufrufen

Werte werden **einzeln** durch `emit()` zurückgegeben und mit `collect()` abgerufen

Hands-on

```
<!-- Navbar -->
<div class="w3-top">
  <a class="w3-bar w3-black w3-hide-large w3-right" href="javascript:void(0)" title="Toggle Navigation Menu"><i class="fa fa-bars"></i></a>
  <a href="#" class="w3-bar-item w3-button w3-padding-large">HOME</a>
  <a href="#band" class="w3-bar-item w3-button w3-padding-large">BAND</a>
  <a href="#tour" class="w3-bar-item w3-button w3-padding-large">TOUR</a>
  <a href="#contact" class="w3-bar-item w3-button w3-padding-large w3-hide-small">CONTACT</a>
  <div class="w3-dropdown-hover w3-hide-small">
    <button class="w3-padding-large w3-button" title="More">MORE</button>
    <div class="w3-dropdown-content w3-bar-block">
      <a href="#" class="w3-bar-item w3-button w3-padding-large">...</a>
    </div>
  </div>
</div>
```


An aerial photograph of a river delta, showing a complex network of water channels and green, marshy land. The text "Daten im Fluss" is overlaid in the center-left. The image is dark and moody, with a focus on the intricate patterns of the water and land.

Daten im Fluss

Flows sind „cold“

```
fun main() = runBlocking {  
    println("Teilnehmer:")  
    val partFlow = participantFlow()  
  
    println("Flow erzeugt...")  
    delay(500)  
    println("...nichts passiert!")  
  
    // Erst collect() startet den Flow!  
    partFlow.collect { println(it) }  
    println("Alle aufgelistet.")  
}
```

```
fun participantFlow() = flow {  
    participantDatabase.forEach {  
        delay(1000L)  
        println("Liefere $it")  
        emit(it)  
    }  
}
```

- Flows sind grundsätzlich „cold“

Werden nicht selbst aktiv, d.h. die Erzeugung startet den Flow noch nicht
Erst bei Verbrauch (z.B. mit `collect()`) wird der Flow abgearbeitet

- Ausnahme: SharedFlows

Einfluss nehmen

```
fun main() = runBlocking {  
    println("Teilnehmer:")  
    participantFlow()  
        .onStart { println("Flow startet...") }  
        .map { p -> p.length }  
        .take(2)  
        .collect { println(it) }  
}
```

```
fun participantFlow() = flow {  
    participantDatabase.forEach {  
        delay(1000L)  
        println("Liefere $it")  
        emit(it)  
    }  
}
```

- Operatoren können auf gelieferte Werte oder das Verhalten des Flows Einfluss nehmen

Intermediate Operatoren verändern das Verhalten und liefern einen neuen Flow
Terminale Operatoren starten die Abarbeitung und sammeln die Ergebnisse ein

Flows kombinieren

```
fun main() = runBlocking {  
    println("Teilnehmer:")  
    positionFlow()  
        .zip(participantFlow())  
        { pos, part -> "$pos. $part" }  
    .collect { println(it) }  
}
```

```
fun participantFlow() = flow {  
    participantDatabase.forEach {  
        delay(1000L)  
        emit(it)  
    }  
}  
  
fun positionFlow() = (1..4).asFlow()
```

- Flows können zu einem neuen, kombinierten Flow verbunden werden
Regel für die Bildung der Werte des neuen Flows muss dabei mit angegeben werden

Verarbeitung abbrechen – Suspending Flows

```
fun main() = runBlocking {  
    println("Teilnehmer:")  
    withTimeout(350) {  
        participantFlow()  
        .collect { println(it) }  
    }  
}
```

```
fun participantFlow() = flow {  
    participantDatabase.forEach {  
        delay(1000L)  
        emit(it)  
    }  
}
```

- Abarbeitung des Flows kann mit Timeout versehen werden
 - Abbruch nach Ablauf
 - Benötigt Unterstützung durch die verwendeten Suspend Functions
- Auch explizites `cancel` möglich
 - Siehe nächstes Beispiel

Verarbeitung abbrechen – Busy Flows

```
fun main() = runBlocking {  
    println("Teilnehmer:")  
  
    participantFlow()  
        .collect {  
            if( it.length > 4 ) cancel()  
            println(it)  
        }  
  
    positionFlow()  
        .cancellable()  
        .collect {  
            if ( it > 2) cancel()  
            println(it)  
        }  
}
```

```
fun participantFlow() = flow {  
    participantDatabase.forEach {  
        emit(it)  
    }  
}  
  
fun positionFlow() = (1..4).asFlow()
```

- Abbruch bei Busy Flows nicht garantiert!

Flow Builder `flow{}` unterstützt Cancellation unabhängig von Suspensions

Bei anderen Flow Buildern explizite Deklaration via `cancellable()` nötig

Am Ende des Flusses

```
fun main() = runBlocking {  
    println("Teilnehmer:")  
  
    try {  
        participantFlow()  
        .collect {  
            println(it)  
        }  
    } finally {  
        println("Fertig.")  
    }  
}
```

```
fun main() = runBlocking {  
    println("Teilnehmer:")  
  
    participantFlow()  
        .onCompletion {  
            println("Flow ${ if(it == null)  
                            "erfolgreich."  
                            else  
                            "fehlerhaft!"})")  
        }  
        .catch { e -> println("Ups... '$e'")}  
        .collect { println(it) }  
}
```

- Auf das Ende der Abarbeitung kann ebenfalls imperativ oder deklarativ reagiert werden

Deklarative Lösung ermöglicht implizite Prüfung auf Fehler

Flows nebenläufig ausführen

```
fun main() = runBlocking {  
    println("Teilnehmer:")  
    participantFlow()  
        .onEach { println(it) }  
        .launchIn(this)  
    println("Weiter geht's...")  
}
```

```
fun participantFlow() = flow {  
    participantDatabase.forEach {  
        delay(1000L)  
        emit(it)  
    }  
}
```

- Die Abarbeitung von Flows mit `collect()` blockiert den weiteren Ablauf des

Aufrufers

Auslagerung in eigene Koroutine ermöglicht eine nebenläufige Abarbeitung
Verarbeitung muss in `onEach()` ausgelagert werden

Hands-on

```
<!-- Navbar -->
<div class="w3-top">
  <a class="w3-bar w3-black w3-hide-large w3-right" href="javascript:void(0)" title="Toggle Navigation Menu"><i class="fa fa-bars"></i></a>
  <a href="#" class="w3-bar-item w3-button w3-padding-large">HOME</a>
  <a href="#band" class="w3-bar-item w3-button w3-padding-large">BAND</a>
  <a href="#tour" class="w3-bar-item w3-button w3-padding-large">TOUR</a>
  <a href="#contact" class="w3-bar-item w3-button w3-padding-large w3-hide-small">CONTACT</a>
  <div class="w3-dropdown-hover w3-hide-small">
    <button class="w3-padding-large w3-button" title="More">MORE</button>
    <div class="w3-dropdown-content w3-bar-block">
      <a href="#" class="w3-bar-item w3-button w3-padding-large">...</a>
    </div>
  </div>
</div>
```




Koroutinen verbinden

Channels – Non blocking Kommunikationskanäle

```
val chars = "Hello KKON digital 2022!".toCharArray()

fun main() = runBlocking {
    val channel = Channel<Char>()
    launch {
        chars.forEach {
            delay(400L) // aufwendige Datenverarbeitung
            channel.send(it) // Ausgabe des naechsten Wertes
        }
    }
    repeat(chars.size) { print(channel.receive()) }
}
```

- Um Daten aus einer Koroutinen zu liefern können *Channels* verwendet werden
- *Channels* arbeiten ähnlich wie *BlockingQueues*, nur **suspendable**
Können auch Daten Puffern => `Channel<Char>(10)`

Channels – Mehr gibt's nicht

```
val chars = "Hello KKON digital 2022!".toCharArray()

fun main() = runBlocking {
    val channel = Channel<Char>()
    launch {
        chars.forEach {
            delay(400L) // aufwendige Datenverarbeitung
            channel.send(it) // Ausgabe des naechsten Wertes
        }
        channel.close()
    }
    for(c in channel) { print(c) }
}
```

- Um mitzuteilen, dass keine weiteren Daten vorhanden sind, kann ein *Channel* explizit geschlossen werden.

Der Empfänger wird hierüber durch einen speziellen Token benachrichtigt

Producer – Koroutinen mit impliziter Channel Erzeugung

```
val chars = "Hello KKON digital 2022!".toCharArray()

fun main() = runBlocking {
    val charProducer = produce {
        chars.forEach {
            delay(400L)
            send(it)
        }
    }
    charProducer.consumeEach { print(it) }
}
```

- Der *CoroutineBuilder* `produce` erzeugt eine neue Koroutine, die Ihre Rückgabewerte über über einen `ReceiveChannel` zurückliefert
Passende *ExtensionFunctions* vereinfachen zusätzlich die Datenverarbeitung

Pipelines – Kommunikationkette von Koroutinen (I)

```
fun CoroutineScope.produceWords(charProducer : ReceiveChannel<Char>) :  
    ReceiveChannel<String> = produce {  
        val wordBuffer = StringBuilder()  
        charProducer.consumeEach {  
            if(it == ' ') {  
                send(wordBuffer.toString())  
                wordBuffer.clear()  
            } else {  
                wordBuffer.append(it)  
            }  
        }  
        if(!wordBuffer.isEmpty()) { send(wordBuffer.toString())}  
    }
```

- Bei der Erzeugung von *Producer* Koroutinen können vorhandene *Channels* als Parameter übergeben werden, ...

Pipelines – Kommunikationkette von Koroutinen (II)

```
val chars = "Hello KKON digital 2022!".toCharArray()

fun main() = runBlocking {
    val charProducer = produce {
        chars.forEach {
            delay(400)
            send(it)
        }
    }
    val wordProducer = produceWords(charProducer)
    wordProducer.consumeEach { println(it) }
}
```

- ... um eine Verarbeitungskette von Koroutinen (*Pipeline*) zu erzeugen

Fan Out – Daten mit mehreren Koroutinen parallel verarbeiten

```
val chars = "Hello KKON digital 2022!".toCharArray()

fun main() = runBlocking {
    val charProducer = produce {
        chars.forEach {
            delay(400)
            send(it)
        }
    }
    launchWordConsumer(1, wordProducer)
    launchWordConsumer(2, wordProducer)
}
```

- Datenverarbeitung erfolgt „fair“
(nach FiFO Semantik)
- Auch Fan-In möglich

```
fun CoroutineScope.launchWordConsumer(idx: Int,
    wordProducer: ReceiveChannel<String>) = launch {
    for (word in wordProducer) {
        println("$word ($idx. Consumer)")
    }
}
```

Aktoren – Sicher gemeinsamer Zustand mit Channels (I)

```
var counter = 0

fun main() = runBlocking {
    withContext(Dispatchers.Default) {
        startCoroutines(100, 1000) {
            counter++
        }
    }
    println("Counter = $counter")
}
```

```
suspend fun startCoroutines(n : Int,
    r: Int, action: suspend () -> Unit) {
    coroutineScope {
        repeat(n) {
            launch {
                repeat(r) { action() }
            }
        }
    }
}
```

- Parallel Änderung des gemeinsamen Zustands (counter) ohne jede Synchronisierung
=> Update werden vermutlich verloren gehen (counter < 100.000)

Aktoren – Sicher gemeinsamer Zustand mit Channels (II)

```
sealed class CounterMsg
object IncrementCounter : CounterMsg()
class GetCounterValue(val counterValue: CompletableDeferred<Int>) : CounterMsg()

fun CoroutineScope.counterActor() = actor<CounterMsg> {
    var counter = 0
    for (msg in channel) {
        when (msg) {
            is IncrementCounter -> counter++
            is GetCounterValue -> msg.counterValue.complete(counter)
        }
    }
}
```

- Der *CoroutineBuilder* actor erzeugt eine neue Koroutine, die Ihre Eingabewerte über einen `SendChannel` erhält
Kommunikation erfolgt über Austausch definierter Nachrichten Typen

Aktoren – Sicher gemeinsamer Zustand mit Channels (III)

```
fun countSafe() = runBlocking<Unit> {  
    val counter = counterActor()  
    withContext(Dispatchers.Default) {  
        startCoroutines(100, 1000) {  
            counter.send(IncrementCounter)  
        }  
    }  
  
    val response = CompletableDeferred<Int>()  
    counter.send(GetCounterValue(response))  
    println("Counter = ${response.await()}")  
    counter.close()  
}
```

- Auch bei hoch parallelem Aufruf des Aktors erfolgt die eigentliche Abarbeitung der Anfragen sequentiell
=> Es gehen keine Updates mehr verloren (counter == 100.000)

Hands-on

```
<!-- Navbar -->
<div class="w3-top">
  <a class="w3-bar w3-black w3-hide-large w3-right" href="javascript:void(0)" title="Toggle Navigation Menu"><i class="fa fa-bars">HOM
  <a href="#" class="w3-bar-item w3-button w3-padding-large">HOME
  <a href="#band" class="w3-bar-item w3-button w3-padding-large"
">BAND</a>
  <a href="#tour" class="w3-bar-item w3-button w3-padding-large"
">TOUR</a>
  <a href="#contact" class="w3-bar-item w3-button w3-padding-large
w3-hide-small">CONTACT</a>
<div class="w3-dropdown-hover w3-hide-small">
  <button class="w3-padding-large w3-button" title="More">MORE
    fa fa-caret-down"></i></button>
  <div class="w3-dropdown-content w3-bar-block" style="display: none;">
    <a href="#" class="w3-bar-item w3-button w3-padding-large">HOME
```


Wie geht es von
hier weiter?



Weiterführende Themen

- Start Modi
- Exception Handling
- Shared State
- ...

 **Kotlin**
<https://kotlinlang.org/docs/coroutines-guide.html>

Referenzen

- Beispiele & Folien
 - <https://github.com/wern/kotlin-non-blocking-coroutines-flows-ws-kkon-2024>
- Kotlin Dokumentation
 - <https://kotlinlang.org/docs/coroutines-overview.html>
- Koroutinen und Java Virtual Threads
 - <https://www.youtube.com/watch?v=zluKcazgkV4>



MATHEMA

Vielen Dank! Fragen?

Werner Eberling

E-Mail: werner.eberling@mathema.de

Twitter: @Wer_Eb

Beispiele:

<https://github.com/wern/kotlin-non-blocking-coroutines-flows-ws-kkon-2024>

www.mathema.de



Beispiele?
Scan me ;)

