# Kotlin Coroutine Deep Dive

Thomas Künneth, MATHEMA Software GmbH

05.02.2020

https://github.com/tkuenneth/mind_the_thread

Asynchrone Programmierung

Unabhängig voneinander rechnen

Gemeinsam ein Ergebnis ermitteln

# Nebenläufigkeit

Mehrere Dinge gleichzeitig tun

Auf Ergebnis eines (entfernten) Prozeduraufrufs warten

und trotzdem mit anderen Programmteilen weiter machen

„Unabhängig voneinander rechnen“ und „Gemeinsam ein Ergebnis ermitteln“ *im großen Stil* funktioniert am besten mit echter Parallelisierung

Auf das Ergebnis eines (entfernten) Prozeduraufrufs warten und trotzdem mit anderen Programmteilen weiter machen

ist Gegenstand dieses Talks


**java_examples/BadButtonDemo.java**

# Blockierende Oberflächen

- Praktisch alle relevanten UI-Frameworks sind single-threaded
- Operationen > 1000 ms fallen dem Nutzer auf
- Deshalb:
  - Nur Aktualisieren der Oberfläche auf dem UI-Thread
  - Langläufer *woanders* ausführen

```
private void good(JLabel result) {
    new Thread(() -> {
        int i;
        do {
            i = (int) (Math.random() * Integer.MAX_VALUE);
            System.out.println(i);
        }
        while (i != Integer.MAX_VALUE / 3);
        final int ii = i;
        SwingUtilities.invokeLater(() -> {
            result.setText(Integer.toString(ii));
        });
    }).start();
}
```

**java_examples/BadButtonDemo.java**

- JavaScript: `async` in Verbindung mit `Promise`
- C#: `async` und `await` in Verbindung mit `Task`

- Kotlin: Koroutinen

Asynchrone Programmierung ist im Backend genauso wichtig

```
fun good(result: JLabel) {
    GlobalScope.launch {
        var i: Int
        do {
            i = (Math.random() * Int.MAX_VALUE).toInt()
            println(i)
        } while (i != Int.MAX_VALUE / 3)
        SwingUtilities.invokeLater {
            result.text = i.toString()
        }
    }
}
```

**ButtonDemo.kt**

A **coroutine** is an instance of [a] suspendable computation. It is conceptually similar to a thread, in the sense that it takes a block of code to run and has a similar life-cycle — it is created and started, but it is not bound to any particular thread. It may suspend its execution in one thread and resume in another one. Moreover, like a future or promise, it may complete with some result (which is either a value or an exception).

https://github.com/Kotlin/KEEP/blob/master/proposals/coroutines.md#terminology

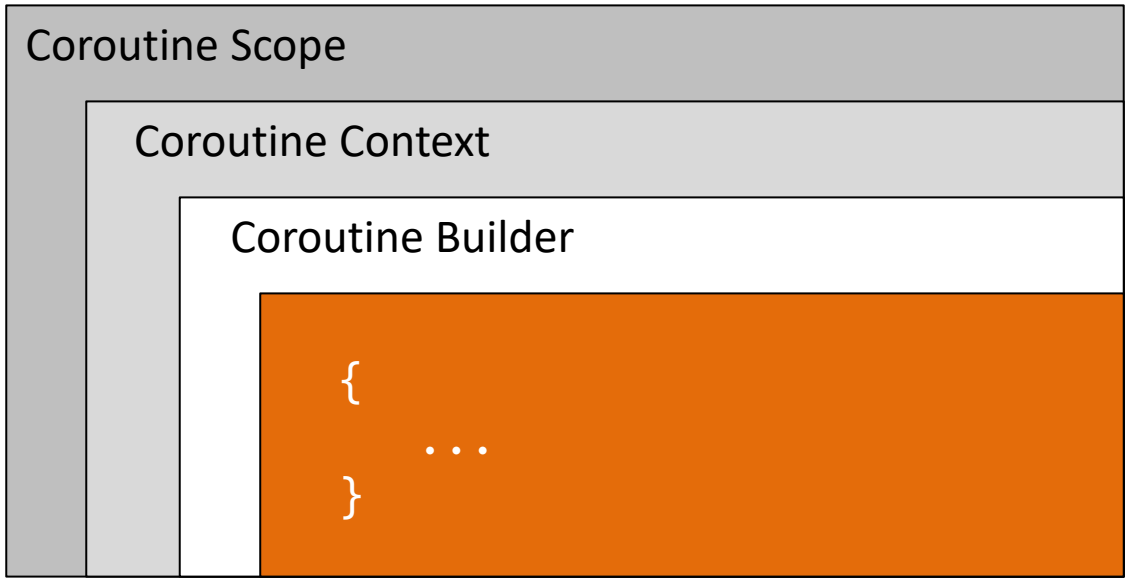
# TL;DR;

- Koroutinen können unterbrechen und wieder fortgesetzt werden
- Sind wie leichtgewichtige Threads
- Kotlin nutzt für die Ausführung *irgendwie* Threads

```kotlin
import kotlinx.coroutines.GlobalScope
import kotlinx.coroutines.delay
import kotlinx.coroutines.launch

fun main() {
    print("Hello, ")
    GlobalScope.launch {
        delay(1000L)
        println("World!")
    }
    Thread.sleep(2000L)
}
```

**SimpleCoroutineDemo.kt**

- `delay()` pausiert Koroutine, hält aktuellen Thread aber nicht an
- Mischung von blockierenden und nicht blockierenden Aufrufen
- Unschön: Wir raten die Ausführdauer


Coroutine Scope
Coroutine Context
Coroutine Builder
{
...
}

# CoroutineScope

- Verwaltet und steuert eine oder mehrere Koroutinen
  - Kann Koroutinen starten und abbrechen
  - Wird bei Abbrüchen und Fehlern informiert
- Legt deren Gültigkeit fest
  - `GlobalScope`-Koroutinen sind nur bzgl. der Laufzeit der Anwendung eingeschränkt
  - Plattformen und Apps können eigene Scopes definieren

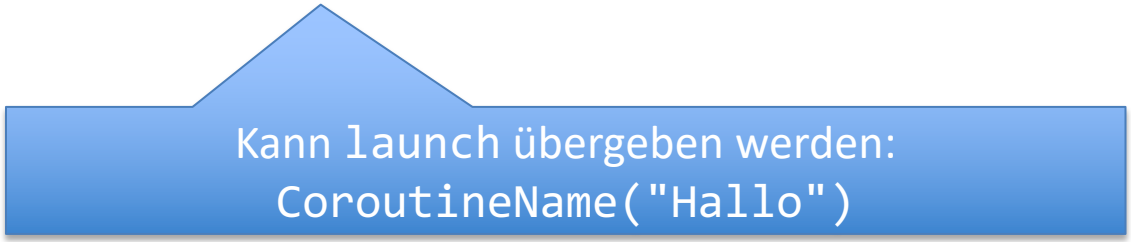
# CoroutineContext

- Jede Koroutine wird in einem bestimmten Kontext ausgeführt (`CoroutineContext`)
- Wird über `CoroutineScope.coroutineContext` zur Verfügung gestellt
- Als Index-basierte Menge von `Elementen` implementiert (eine Mischung aus Set und Map)
- Werte können mit + hinzugefügt werden

- Eine neue Koroutine erbt den Elternkontext
- Elternkontext = Defaults + geerbter Kontext + Argumente (z. B. aus `launch`)
- neuer `CoroutineContext` = Elternkontext + `Job()`

# Wichtige Elemente sind...

- CoroutineDispatcher
- Job
- CoroutineExceptionHandler
- CoroutineName



# Coroutine Dispatcher

- Legen den/die Threads für die Ausführung von Koroutinen fest
- Können Koroutinen auf einen Thread beschränken, in einem Pool ablaufen lassen, oder ohne Einschränkungen

- `Dispatchers.Default`
  Ausführung in einem Thread-Pool, der sich an der Zahl der Kerne orientiert
- `Dispatchers.Main`
  Ausführung nur auf dem Main Thread
- `Dispatchers.IO`
  Für lang laufende/blockierende I/O-Operationen
- `Dispatchers.Unconfined`
  Ausführung auf beliebigen Thread möglich (sollte nicht verwendet werden)

A **coroutine builder** [is] a function that takes some suspending lambda as an argument, creates a coroutine, and, optionally, gives access to its result in some form. For example, `launch{}`, `future{}`, and `sequence{}` [..] are coroutine builders. The standard library provides primitive coroutine builders that are used to define all other coroutine builders.

https://github.com/Kotlin/KEEP/blob/master/proposals/coroutines.md#terminology

A **suspending lambda** [is a] a block of code that have to run in a coroutine. It looks exactly like an ordinary lambda expression but its functional type is marked with `suspend` modifier. Just like a regular lambda expression is a short syntactic form for an anonymous local function, a suspending lambda is a short syntactic form for an anonymous suspending function. It may suspend execution of the code without blocking the current thread of execution by invoking suspending functions. For example, blocks of code in curly braces following `launch`, `future`, and `sequence` functions [...] are suspending lambdas.

https://github.com/Kotlin/KEEP/blob/master/proposals/coroutines.md#terminology

# Coroutine Builder: TL;DR;

- Starten Koroutinen
- Beispiele: `launch`, `async`, …
- Erweitern `CoroutineScope`
- Können Parameter erhalten
  - Start Modus
  - Kontexterweiterungen

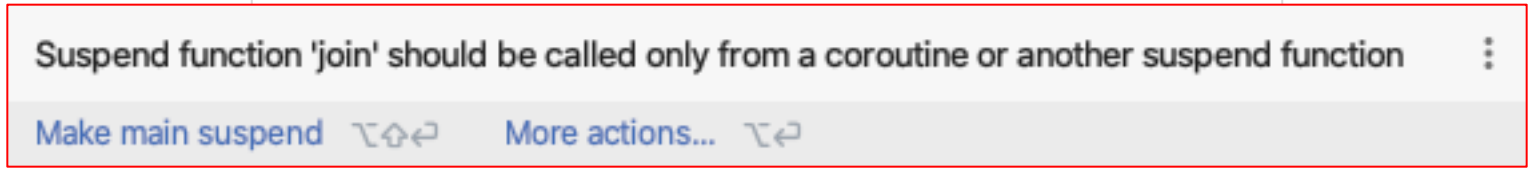
# Start Modi

- `DEFAULT`
  Sofort mit Ausführung beginnen
- `ATOMIC`
  ähnlich `DEFAULT`; kann erst nach Beginn der Ausführung abgebrochen werden
- `LAZY`
  Start nur bei Bedarf (z. B. Zugriff auf Ergebnis)
- `UNDISPATCHED`
  Sofortige Ausführung auf aktuellen Thread, aber Unterbrechung sobald suspension point erreicht wird

```java
public class BlockingDemo {

    public static void main(String[] args) {
        Thread t = new Thread(() -> {
            try {
                Thread.sleep(1000);
                System.out.println("World!");
            } catch (InterruptedException e) { /**/ }
        });
        System.out.print("Hello, ");
        t.start();
        try {
            t.join();
        } catch (InterruptedException e) { /**/ }
    }
}
```

**java_examples/BlockingDemo.java**

```kotlin
import kotlinx.coroutines.GlobalScope
import kotlinx.coroutines.delay
import kotlinx.coroutines.launch

fun main() {
    print("Hello, ")
    val job = GlobalScope.launch {
        delay(1000L)
        println("World!")
    }
    job.join()
}
```



A **suspending function** is a function that is marked with `suspend` modifier. It may suspend execution of the code without blocking the current thread of execution by invoking other suspending functions. A suspending function cannot be invoked from a regular code, but only from other suspending functions and from suspending lambdas [...]. For example, `await()` and `yield()` [...] are suspending functions that may be defined in a library. The standard library provides primitive suspending functions that are used to define all other suspending functions.

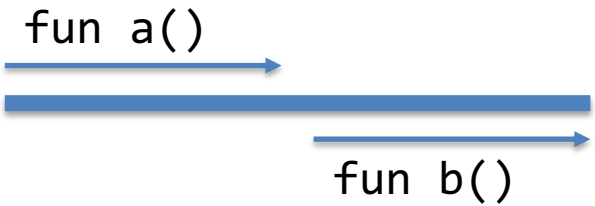
https://github.com/Kotlin/KEEP/blob/master/proposals/coroutines.md#terminology

TL;DR;

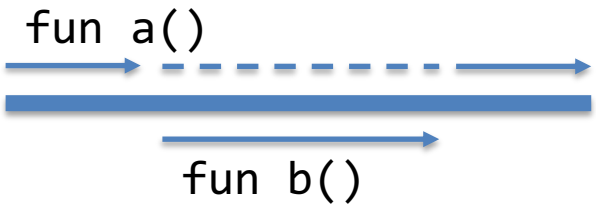
- suspend functions sind Grundbausteine von Koroutinen
- Machen irgendwann eine Pause oder müssen auf ein Ergebnis warten
- Blockieren aber nicht den aktuellen Thread
- Werden aus Koroutinen oder anderen suspending functions aufgerufen
- Liefern wie normale Funktionen ein Ergebnis

```
import kotlinx.coroutines.delay

suspend fun main() {
    print("Hello, ")
    hello()
}

suspend fun hello() {
    delay(1000)
    println("World!")
}
```

**SimpleSuspendingFunctions.kt**




normale, blockierende Funktionen




unterbrechbare Funktion

A **suspension point** is a point during coroutine execution where the execution of the coroutine may be suspended. Syntactically, a suspension point is an invocation of suspending function, but the actual suspension happens when the suspending function invokes the standard library primitive to suspend the execution.

A **continuation** is a state of the suspended coroutine at suspension point. It conceptually represents the rest of its execution after the suspension point.

https://github.com/Kotlin/KEEP/blob/master/proposals/coroutines.md#terminology

**Unter Umständen wird eine suspend function auf einem anderen Thread fortgesetzt als den, auf dem sie unterbrochen wurde.**
**Dieses Verhalten wird durch Coroutine Dispatcher gesteuert.**

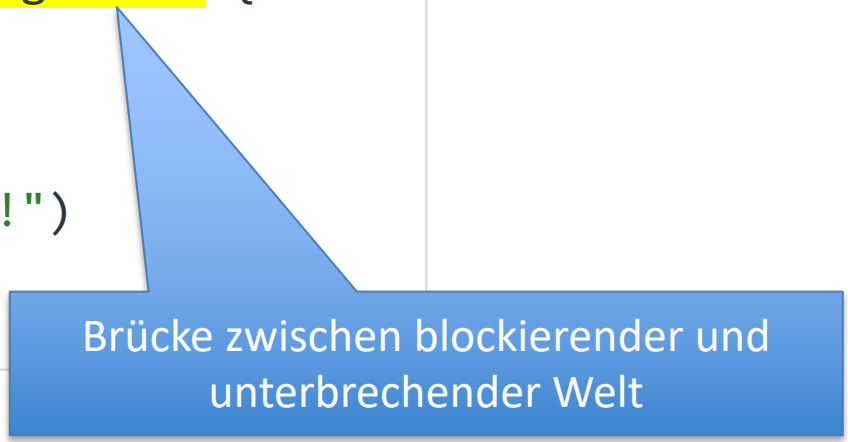
```
import kotlinx.coroutines.GlobalScope
import kotlinx.coroutines.delay
import kotlinx.coroutines.launch
import kotlinx.coroutines.runBlocking

fun main() {
    print("Hello, ")
    val job = GlobalScope.launch {
        delay(1000L)
        println("World!")
    }
    runBlocking {
        job.join()
    }
}
```

Ein blockierender Coroutine Builder

**RunBlockingJoinDemo.kt**

```
import kotlinx.coroutines.delay
import kotlinx.coroutines.launch
import kotlinx.coroutines.runBlocking

fun main() = runBlocking<Unit> {
    print("Hello, ")
    launch {
        delay(1000L)
        println("World!")
    }
}
```

**RunBlockingDemo.kt**



```
fun main() {
    runBlocking {
        for (i in 1..10) {
            launch {
                delay(i * 1000L)
                println("$i finished")
            }
        }
        println("Already there")
    }
    println("All done")
}
```

**SeveralCoroutines.kt**

```
fun main() = runBlocking<Unit> {
    coroutineScope {
        for (i in 1..10) {
            launch {
                delay(i * 1000L)
                println("$i finished")
            }
        }
        println("Already there")
    }
    println("All done")
}
```


**CoroutineScope.kt**

- `runBlocking` und `coroutineScope` warten auf die Ausführung ihres Blocks und Kinder
- `runBlocking` blockiert den aktuellen Thread, `coroutineScope` unterbricht nur die Ausführung
- Auf dem aktuellen Thread können andere Aktionen ausgeführt werden

# Ausführung abbrechen

- `launch` liefert `Job`-Objekt
  - Steuert den Lebenszyklus
  - Ermöglicht Hierarchien
- `cancel` initiiert den Abbruch
- Mit `join` auf Abschluss warten
- `launch` ist fire and forget

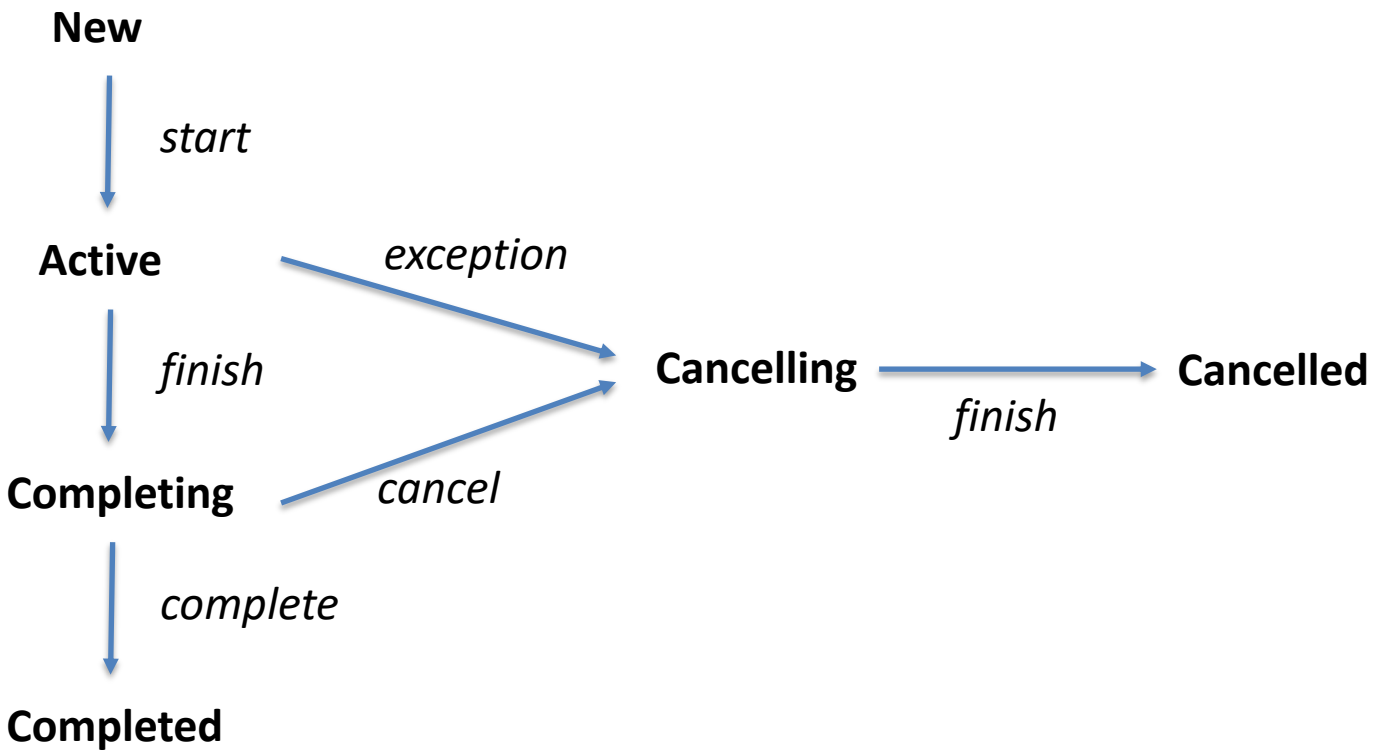
```
fun main() = runBlocking {
    println(measureTimeMillis {
        val job = launch {
            println("Enter")
            delay(10000)
            println("Exit")
        }
        delay(3000)
        job.cancelAndJoin()
        println("Cancelled")
    })
}
```

**CancelDemo.kt**

# Generell gilt...

- Das Abbrechen eines Scopes mit `cancel()` bricht auch die Kindjobs ab
- Vorhandene Scopes z. B. unter Android führen `cancel()` automatisch aus
- Ein mit `cancel()` abgebrochenes Kind führt nicht zum Abbruch anderer Kinder des Jobs

# ´Lebenszyklus eines Jobs



# Eigenschaften

- `isActive`
  Aktiv
- `isCancelled`
  Abgebrochen
- `isCompleted`
  Abgeschlossen

```
fun main() = runBlocking {
    println(measureTimeMillis {
        val job = launch {
            println("Enter")
            var count = 0
            while (true) {
                println("${++count}")
            }
            println("Exit")
        }
        delay(3000)
        job.cancelAndJoin()
        println("Cancelled")
    })
}
```

**CancelNotWorkingDemo.kt**

# Koroutinen sind kooperativ

- Regelmäßig `delay()`, `yield()` oder `ensureActive()` aufrufen
- Abbruch-Wünsche mit `isActive` erfragen

```
...
val job = launch {
    println("Enter")
    var count = 0
    while (isActive) {
        println("${++count}")
        yield()
    }
    println("Exit")
}
...
```

- Alle suspending functions in kotlinx.coroutines sind abbrechbar
- Auch eigene suspend functions sollten abbrechbar sein

Koroutinen sollten sich beenden,
wenn sie nicht mehr benötigt werden

# CancellationException

- Wird von suspend functions geworfen, um Abbruch anzuzeigen
- Mit ihr kann zwischen Abbrüchen und anderen Exceptions unterschieden werden

```
fun main(): Unit = runBlocking<Unit> {
    println(measureTimeMillis {
        val delay = 2000
        try {
            withTimeout(delay + 1000L) {
                val current = System.currentTimeMillis()
                while ((System.currentTimeMillis() - current) < delay) {
                    yield()
                }
            }
        } catch (e: TimeoutCancellationException) {
            println("I timed out")
        } finally {
            println("finally")
        }
    })
}
```

**TimeoutDemo.kt**

```
internal fun CoroutineContext.checkCompletion() {
    val job = get(Job)
    if (job != null && !job.isActive) throw job.getCancellationException()
}
```

```
fun main() = runBlocking<Unit> {
    println("Took ${measureTimeMillis {
        println("result: ${myFun1() + myFun2()}")
    }} ms")
}

suspend fun myFun1(): Int {
    delay(1000)
    return 1
}

suspend fun myFun2(): Int {
    delay(2000)
    return 2
}
```

**ReturnAValueDemo.kt**

# Deferred

- Nicht blockierende, abbrechbare Future
- Wird mit `async` (Builder) oder `CompletableDeferred()` erzeugt
- Dieselben Zustände wie `Job`
- Ergebnis wird mit `await()` erfragt (wirft im Fehlerfall eine Exception)

```
fun main() = runBlocking<Unit> {
    async { waiter(2000) }
    println("Result: ${async {
        waiter(3000)
    }.await()}")
}

suspend fun waiter(timeToWait: Long): Long {
    println("Waiting for $timeToWait ms")
    delay(timeToWait)
    return timeToWait
}
```

**AsyncDemo.kt**

Was man alles mit Koroutinen machen kann...

```kotlin
fun main() {
    val fibs = fibList(10)
    for (i in (1..fibs.size)) {
        println("fib($i) = ${fibs[i - 1]}")
    }
}

fun fibList(n: Int): List<Int> {
    val result: MutableList<Int> = ArrayList()
    for (i in (1..n)) {
        result.add(if (i <= 2) 1 else result[i - 2] + result[i - 3])
    }
    return result
}
```

**Fibonacci.kt**

```kotlin
fun main() = runBlocking {
    fibFlow(10).collectIndexed { index, value ->
        println("fib(${index + 1}) = ${value}")
    }
}

fun fibFlow(n: Int): Flow<Int> = flow {
    var minus2: Int
    var minus1 = 1
    var current = 1
    for (i in 1..n) {
        if (i > 2) {
            minus2 = minus1
            minus1 = current
            current = minus1 + minus2
        }
        emit(current)
    }
}
```

**FibonacciFlow.kt**

# Flow Operatoren

- `transform` (map reduce, Zwischenwerte, ...)
- Einsammeln der Flow-Werte (terminal)
  - `collect`
  - `toList` und `toSet`
  - `first` und `single`
  - `reduce` und `fold`

# Fortgeschrittene Flow Konzepte

- `flowOn` ändert Ausführungskontext
- `buffer` ändert den Charakter eines F.lows von sequentiell zu nebenläufig
- `conflate` erlaubt das Auslassen von Zwischenwerten
- Multiple Flows mit `zip` und und `combine`

# Channels

- Mit `Deferred` können einzelne Werte zwischen Koroutinen ausgetauscht werden
- `Channels` übertragen Datenströme
- Konzeptionell der `BlockingQueue` ähnlich

```
fun main() = runBlocking<Unit> {
    val channel = Channel<Int>()
    val job = launch {
        while (isActive) {
            channel.send((Math.random() * 10).toInt())
        }
    }
    repeat(5) {
        println(channel.receive())
    }
    job.cancelAndJoin()
    channel.close()
}
```

**ChannelDemo.kt**

```
fun main() = runBlocking<Unit> {
    val channel = Channel<Int>()
    launch {
        repeat(5) {
            channel.send((Math.random() * 10).toInt())
        }
        channel.close()
    }
    for (i in channel) {
        println(i)
    }
}
```

**ChannelDemo2.kt**

```
fun main() = runBlocking<Unit> {
    myProducer().consumeEach { println(it) }
}

fun CoroutineScope.myProducer(): ReceiveChannel<Int>
    = produce {
      repeat(5) {
          send((Math.random() * 10).toInt())
      }
}
```

**ChannelDemo3.kt**

# Mehr mit Producern

- Fan out
  - Mehrere Koroutinen erhalten Werte vom selben `Channel`
  - Teilen sich die Arbeit
- Fan in
  - Mehrere Koroutinen senden Daten zum selben `Channel`
- Gepufferte `Channels`

# Vom Umgang mit Fehlern

- Ausnahmen nach `launch` werden wie nicht gefangene Ausnahmen in Threads behandelt
- Abgestürzte Kind-Koroutinen brechen die Elternkoroutine mit der korrespondierenden Ausnahme ab
- Bei `async` sollten Ausnahmen explizit behandelt werden

- Nicht gefangene Exceptions können mit `CoroutineExceptionHandlern` getrackt werden
- Alternative: `runCatching()`

```
import kotlinx.coroutines.launch
import kotlinx.coroutines.runBlocking

fun main() = runBlocking<Unit> {
    val job = launch() {
        println("Hello launch")
        throw RuntimeException("Hello crash")
    }
    job.join()
    // not printed!!!
    println("join() finished")
}
```

**ExceptionDemo.kt**

```
fun main() = runBlocking {
    val deferred = GlobalScope.async {
        println("Started")
        delay(1000)
        throw IllegalArgumentException("I want to crash")
    }
    try {
        deferred.await()
    } catch (e: Exception) {
        println(e.message)
    } finally {
        println("I am finally here")
    }
    println("Finished")
}
```

**ExceptionDemo3.kt**

```
import kotlinx.coroutines.CoroutineExceptionHandler
import kotlinx.coroutines.GlobalScope
import ...

fun main() = runBlocking {
    val handler = CoroutineExceptionHandler { _, exception ->
        println("Caught $exception")
    }
    val job = GlobalScope.launch(handler) {
        println("Hello launch")
        throw RuntimeException("Hello crash")
    }
    job.join()
    println("join() finished")
}
```

**ExceptionDemo2.kt**

# SupervisorJob

- Ein Fehler oder Abbruch führt nicht zum Abbruch anderer Kinder bzw. des Parents

```
val scope =
CoroutineScope(SupervisorJob())
```

- Nicht gefangene Ausnahmen werden nach oben propagiert
- SupervisorJob funktioniert nur, wenn er der direkte Parent einer Koroutine ist

# Finale

- Koroutinen werden durch eigene Bibliothek (`kotlinx.coroutines`) implementiert
- Auf Sprachebene „nur“ ein Schlüsselwort (`suspend`)
- Einige Features sind (noch) experimentell

- Koroutinen fühlen sich wie Threads an
- Verhalten sich aber anders
  - *Blockierend* vs. *unterbrechend*
  - Erfordern Mitarbeit des Entwicklers (*kooperativ*)
- Aufpassen, wenn Thread-basierte Mechanismen „einfach so“ übernommen werden

- API fühlt sich nicht immer intuitiv an
  - Wann welcher Dispatcher?
  - Wann welcher Start Mode?
- Default-Werte machen Verhalten manchmal schwer nachvollziehbar

# Hinweise für eigene Experimente

- Umgang mit veränderlichen Zustand mit Aktoren
- Select-Ausdrücke (experimentelles Feature)

- In den Code schauen, um Koroutinen genau zu verstehen

# Vielen Dank

thomas.kuenneth@mathema.de
@tkuenneth

https://github.com/tkuenneth/mind_the_thread