

RxAndroid

Reaktives Programmieren in Android

Leon Wollenweber und Stefan Neufeldt

Gliederung

- Einleitung
 - Was ist Reaktive Programmierung?
 - Anwendungsfälle
 - Bestandteile
- Grundlagen
 - Einführung in nebenläufige Funktionen mit Kotlin
 - Einführung in Kotlin Flow
- Flow-Operatoren
- Weitere Flow-Typen
 - Shared und State Flow
- Flows in Android UI

Was ist Reaktive Programmierung?

Reaktive Programmierung

- Programmierparadigma
- Konsument reagiert auf asynchron eintreffende Daten

Hintergrund

- Verbreitung von mobilen Geräten, die ...
 - miteinander vernetzt sind
 - über Mehrkernprozessoren verfügen
 - Daten aus der Cloud abrufen
 - Milliardenfach vorhanden sind

... und deren Software:

- unmittelbar auf Benutzereingaben reagieren soll
- sich ständig verändert und weiterentwickelt

Reactive Manifesto

- 2014 veröffentlichte Zusammenfassung von Designprinzipien
- Beschreibung und Zusammenfassung von Praktiken, die Unternehmen schon vorher genutzt haben um große Systeme aufzubauen
- Ein System sollte danach sein:
 1. Antwortbereit
 2. Widerstandsfähig
 3. Elastisch
 4. Nachrichtenorientiert

Reactive Manifesto

1. Antwortbereit:

- Das System antwortet zügig und mit konsistenter Geschwindigkeit.
- Verbesserung der Nutzbarkeit und hilft Probleme schnell zu erkennen.

Reactive Manifesto

2. Widerstandsfähig:

- Die Funktion von Komponenten ist voneinander isoliert
- Komponenten sind leicht und automatisiert neu erzeugbar
- Komponenten haben eine klare Aufgabenteilung.



Das System kann auch bei Ausfällen von Hard-/Software noch antworten.

Reactive Manifesto

3. Elastisch:

Komponenten können über Faktoren neu erzeugt werden und die Last auf diese verteilt werden.



Das System kann auf sich ändernde Lastbedingungen reagieren.

Reactive Manifesto

4. Nachrichtenorientiert

- Das System verwendet Nachrichten zur Weitergabe von Daten und Fehlern.
 - So kann die Isolation der Komponenten sichergestellt werden.
 - Gleichzeitig wird der Code auch Ortsunabhängig und kann also sowohl auf einem Gerät, als auch verteilt auf vielen Rechnern laufen.

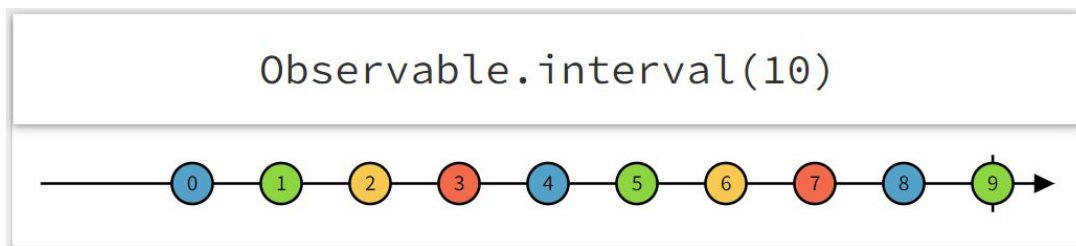
Bestandteile

Grundlagen Reaktiver Programmierung

1. Datenflüsse
2. Funktionale Programmierung
3. Asynchrone Beobachter

Datenflüsse

- Daten haben eine zeitliche Reihenfolge
- Die Gesamtmenge der Daten ist nicht zwangsläufig vorhersehbar



Funktionale Programmierung

Die Operatoren, die die Datenflüsse verarbeiten sind:

- Zustandslos
- haben eine Ein-/Ausgabe
- haben keine Seiteneffekte

Beobachter-Pattern

Die Konsumenten der Datenflüsse werden durch die Quellen (Observables) informiert, wenn ein neues Datenpaket gesendet wird, oder ein Fehler auftritt.

Anwendungsfälle

Nutzungsszenarien

- Backend-Services
 - Evtl. möchte man die Daten noch Filtern / Transformieren
 - Das Anfordern von Daten, z.B. von Datenbanken oder anderen Diensten kann längere Zeit brauchen
 - Die Anwendung soll mit wechselnden Datenmengen skalieren.
- User-Interfaces
 - wenn Daten aus mehreren Quellen dargestellt werden sollen
 - für mobile Applikationen interessant, da diese häufig asynchron Daten anfordern, diese transformieren und im UI darstellen.

Nutzung in Unternehmen

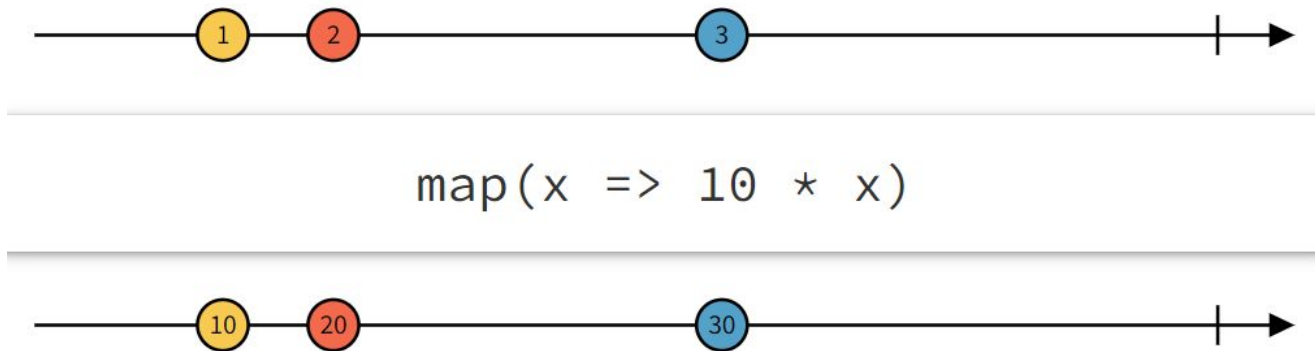


Einige Firmen, die ReactiveX-Frameworks einsetzen.

Operatoren

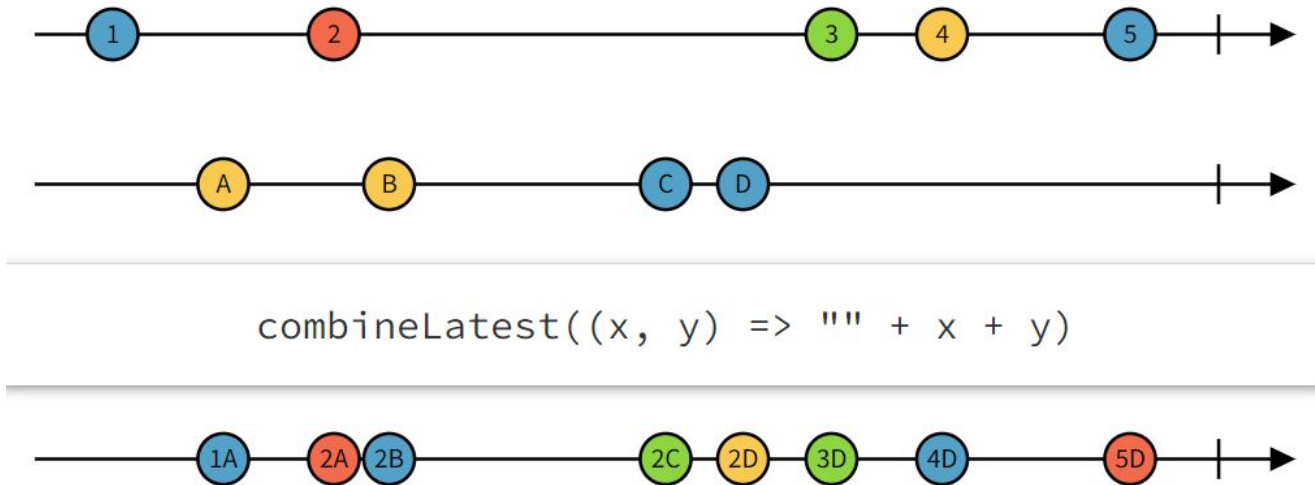
Map

Map wendet eine Funktion auf jedes Element des Datenflusses an.



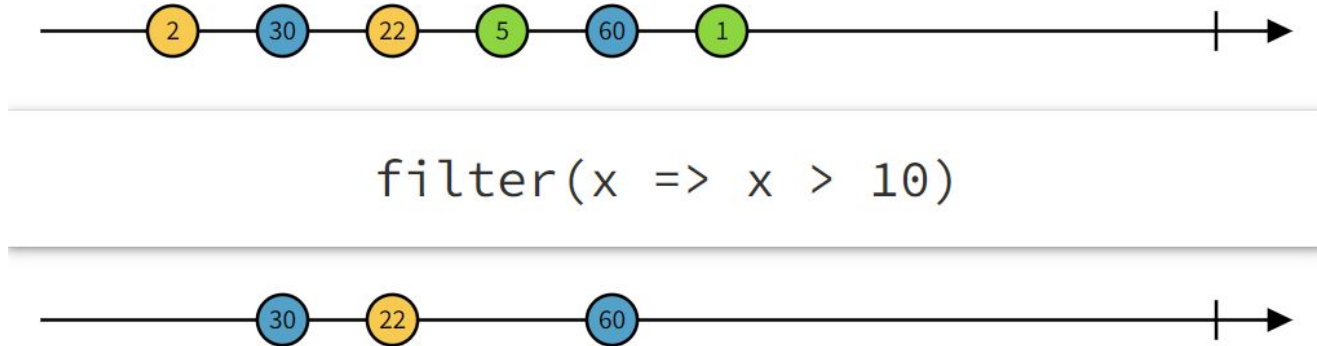
Combine

Fügt zwei Flows zusammen indem jedes emittierte Element mit dem neuesten Element des anderen Flows kombiniert wird.



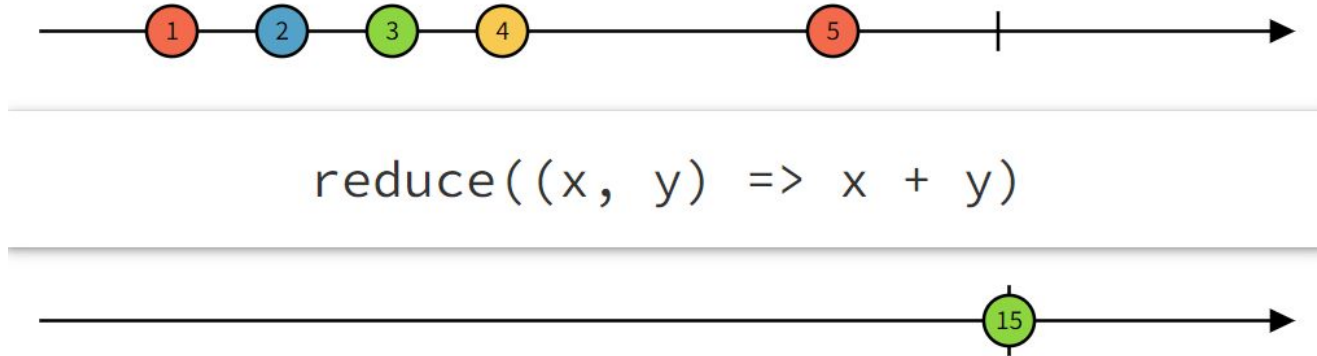
Filter

Entfernt Elemente aus dem Datenfluss, für die eine gegebene Funktion “True” zurückgibt.



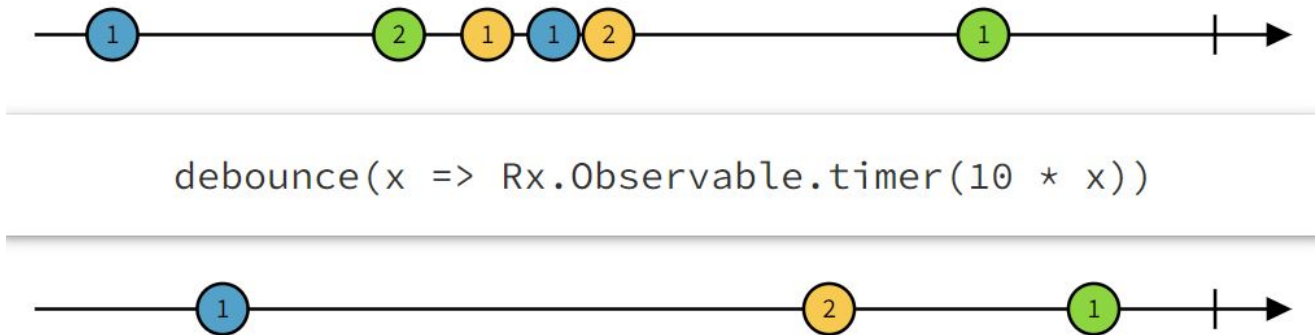
Reduce

Reduce sammelt Elemente, indem es das Ergebnis einer Funktion auf das aktuelle und vorherige Element zurückgibt.



Debounce

Filtert Elemente aus dem Datenstrom, denen innerhalb eines Zeitlimits neuere Elemente gefolgt sind.



Backpressure

Was ist Backpressure?


- Daten werden schneller emittiert als sie verarbeitet werden können.
- Muss behandelt werden, damit es nicht zu Systemabstürzen oder unkontrolliertem Verwerfen von Daten kommt.

Auf Backpressure reagieren

- Dem Produzenten mitteilen, dass weniger Daten emittiert werden sollen.
- Zwischenspeichern (Buffern) der eingehenden Daten
 - Kann kurzzeitige Spitzen abfangen
 - Strategien zum Abarbeiten der Daten und Umgang mit vollem Buffer sollten implementiert werden
- Verwerfen von Daten (Drop)

Einführung in die Nebenläufige Programmierung in Kotlin


Suspend Functions



```
1 suspend fun suspendExample() {  
2     delay(2000L)  
3     println("Suspend Function!")  
4 }
```

- Suspend Functions kennzeichnen Funktionen, die zu jeder Zeit unterbrochen und dann zu einer späteren Zeit wieder ausgeführt werden können
- Eine Funktion muss als suspend gekennzeichnet werden, wenn die Ausführung länger dauert und das Ergebnis nicht sofort vorliegt

Suspend Functions



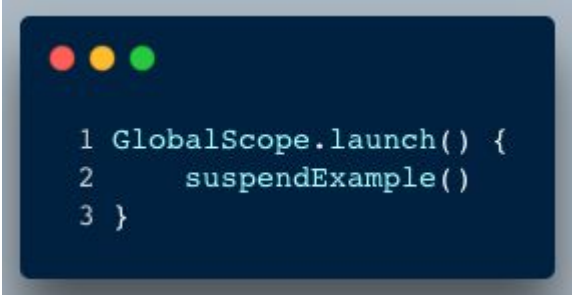
```
1 suspend fun suspendExample() {  
2     delay(2000L)  
3     println("Suspend Function!")  
4 }
```

- Dies kann beispielsweise bei aufwendigen Berechnungen oder bei Netzwerkzugriffen der Fall sein
- Suspend Functions können nur innerhalb einer Coroutine oder innerhalb einer anderen suspend Function aufgerufen werden


Coroutine - Einführung

- Coroutinen verhalten wie leichtgewichtige Threads
- Sie ermöglichen das Ausführen von suspend Functions
- Coroutinen lassen sich mittels des Coroutine Builders leicht starten
 - Vorteil: Die Verwendung von Callbacks ist nicht erforderlich
- Im Vergleich zu Threads können sehr viele Coroutines gleichzeitig gestartet werden, ohne in einen Speicherüberlauf zu geraten

Coroutine Builder



```
1 GlobalScope.launch() {  
2     suspendExample()  
3 }
```



```
1 runBlocking {  
2     suspendExample()  
3 }
```

- Der Coroutine Builder ermöglicht das Aufrufen von Suspend Functions innerhalb des Coroutine-Scopes
- Um auf das Ergebnis der suspend Function zu warten, kann auch ein blockierender Coroutine Builder aufgerufen werden

Coroutine Builder - launch

State	isActive	isCompleted	isCancelled
<i>New</i> (optional initial state)	false	false	false
<i>Active</i> (default initial state)	true	false	false
<i>Completing</i> (transient state)	true	false	false
<i>Cancelling</i> (transient state)	false	false	true
<i>Cancelled</i> (final state)	false	true	true
<i>Completed</i> (final state)	false	true	false

Quelle: <https://proandroiddev.com/kotlin-coroutine-job-hierarchy-finish-cancel-and-fail-2d3d42a768a9>

- Der Coroutine Builder launch gibt ein Job zurück
- Der Status eines Jobs kann abgefragt werden
- Wenn launch aufgerufen wird, startet der Job automatisch

Coroutine Builder - launch und async



- In einem launch muss das Ergebnis innerhalb des Coroutine-Scopes verarbeitet werden, da von außerhalb nicht zugegriffen werden kann
- Stattdessen kann async als Builder aufgerufen werden und das Ergebnis außerhalb der Coroutine mit await abgerufen werden

Laboraufgabe 1 +2

<https://github.com/neufst/Learning-Reactive-Programming-With-Kotlin-Flow>

Einführung in Kotlin Flow

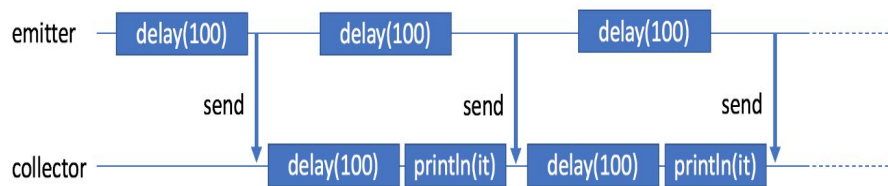
Kotlin Flow

- Kotlin Flow ist eine Möglichkeit, um in Kotlin reaktive Programmierung anzuwenden
- Kotlin Flow unterstützt den Android-Lifecycle
- Wird seit 2020 offiziell von Google als Android Architekturkomponente eingesetzt

Vorteile von Kotlin Flow

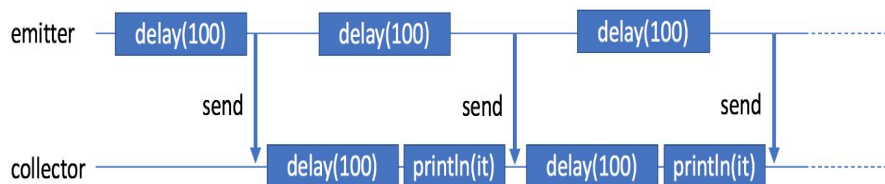
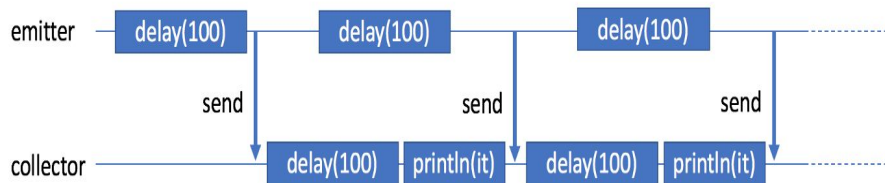
- Viele Android Bibliotheken wie Room, DataStore oder WorkManager unterstützen Kotlin Flow. Die Erstellung eigener Flows ist hierbei nicht notwendig
- Im Vergleich zu anderen Reaktiven Streams hat Flow durch ein vereinfachtes Design eine einfache Handhabung
- Wie auch Kotlin unterstützt auch Flow Kotlin-Multiplatform
- Kompatibel mit anderen reaktiven Streams

Kotlin Flow




- Ein Flow ist ein asynchroner Datenstrom, welcher nacheinander Werte ausgeben kann
- Ein Flow kann sowohl normal als auch mit einer Exception abgeschlossen werden

Kotlin Flow



- Ein Flow ist ein cold-Flow
 - Das heißt, dass die Berechnung des Flows erst ausgeführt wird, wenn dieser auch benutzt wird
- Jeder Collector führt einen eigenen Flow aus
 - Somit bekommt jeder Collector auch eigene Werte zurück

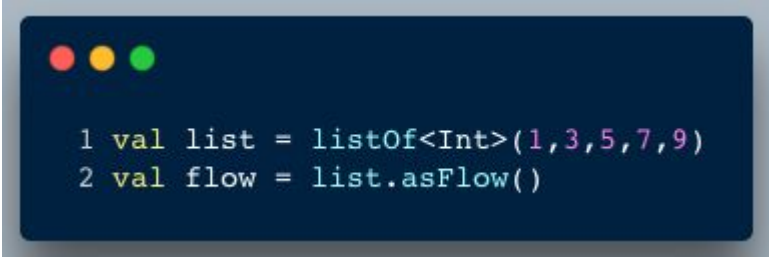
Flow Builder



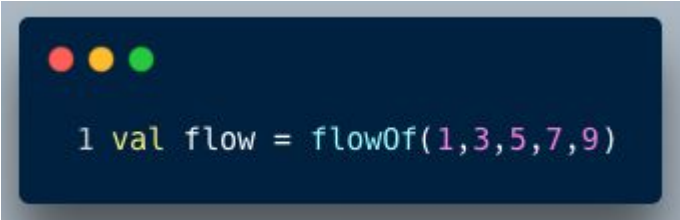
```
1 val flow: Flow<Int> = flow {  
2     for (i in 1..3) {  
3         delay(100)  
4         emit(i)  
5     }  
6 }
```

- Um einen Flow zu erstellen, muss der Datentyp zuvor festgelegt werden
- Innerhalb des Builders können suspend-functions aufgerufen werden
- Aufrufen der emit-Funktion überträgt neue Daten in den Datenstrom

Flow - Weitere Möglichkeiten der Erstellung von Flows



```
1 val list = listOf<Int>(1,3,5,7,9)
2 val flow = list.asFlow()
```



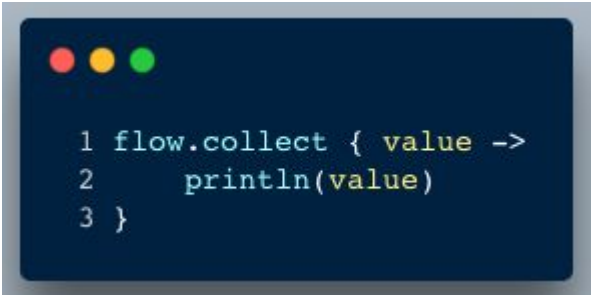
```
1 val flow = flowOf(1,3,5,7,9)
```

- Neben dem Flow Builder können Flows auch aus Sequenzen wie z.B. Listen erzeugt werden

Flow Collect



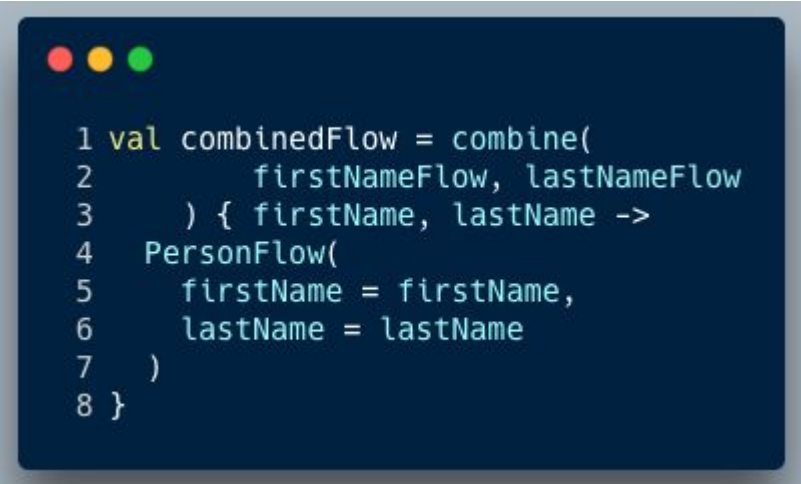
```
1 flow.collect {  
2     println(it)  
3 }
```



```
1 flow.collect { value ->  
2     println(value)  
3 }
```

- Innerhalb des Collects kann über das `it` auf den Wert zugegriffen werden
- Alternativ kann der Wert auch benannt werden

Flow-Combine



```
1 val combinedFlow = combine(  
2     firstNameFlow, lastNameFlow  
3 ) { firstName, lastName ->  
4     PersonFlow(  
5         firstName = firstName,  
6         lastName = lastName  
7     )  
8 }
```

- Beliebige viele Flows können zu einem Flow kombiniert werden
- Der neue Flow wird emitted, wenn alle abhängigen Flows einen Wert liefern
- Hilfreich für UI-States

Shared Flow (Hot-Flow)

- Shared Flow
 - Multicast-Stream
 - Übertragene Daten sind für alle Subscriber gleich
 - Berechnung wird unabhängig von den Subscribern durchgeführt
 - Auch bei mehreren Subscribers wird die Berechnung nur einmal ausgeführt
 - Ein collect läuft unendlich und ist nie abgeschlossen
 - Übertragene Werte können gecached werden
- Mutable Shared Flow
 - Wie Shared Flow
 - Stellt die emit-Funktion bereit

State Flow

- Verhält sich ähnlich wie ein Shared Flow
- Es gibt immer nur einen Wert
- Initialer Wert wird beim Erstellen festgelegt
- Kein caching möglich, es wird immer der letzte Wert verwendet
- Kann z.B. als UI-State verwendet werden

Flows in Android UI - Architekturbeispiel

```
1 data class UiState(  
2     val title: String = "",  
3     val description: String = "",  
4 )  
5  
6 class UiViewModel {  
7     private val _uiState = MutableStateFlow(UiState())  
8     val uiState: StateFlow<UiState> = _uiState.asStateFlow()  
9  
10    fun updateTitle(newTitle: String) {  
11        _uiState.update{  
12            it.copy(title = newTitle)  
13        }  
14    }  
15 }
```

```
1 @Composable  
2 fun AddEditTaskScreen(viewModel: ViewModel) {  
3     val uiState by viewModel.uiState.collectAsStateWithLifecycle()  
4 }
```

Flows in Android UI

- Worauf müssen wir achten?
 - Wechselt die App im Hintergrund, sollte das Collecting für den UI-State unterbrochen werden (App-Lifecycle-Awareness)
 - Wird der Bildschirm gedreht, sollte das Collecting nicht erneut aufgerufen werden, sondern die Werte zwischengespeichert werden
- Verwenden von State Flow
 - mit `asStateFlow()` kann ein `mutableStateFlow` in ein `StateFlow` transformiert werden
 - mit `stateIn()` kann ein Flow (cold) in einen State Flow (hot) transformiert werden
 - Verwenden eines Timeouts mit `SharingStarted.WhileSubscribed(5000)`
 - Stoppt das Collecting, wenn keine Subscriber mehr vorhanden sind
 - Verzögert das Beenden des Collecting um 5 Sekunden, um bei einer Bildschirmdrehung unnötige Verzögerungen zu vermeiden

Laboraufgabe 3 + 4

<https://github.com/neufst/Learning-Reactive-Programming-With-Kotlin-Flow>