

Laboraufgaben zum Thema reaktive Programmierung mit Kotlin Flow (mit Lösungen)

Vorbereitung: Clone das folgende GitHub-Repository auf deinen Rechner:

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

Aufgabe 1: Theorie

Aufgabe 1.1: Reactive Manifesto

Nenne die vier Prinzipien des Reactive Manifesto und beschreibe diese kurz.

- *Antwortbereit (System antwortet zügig, mit konsistenter Geschwindigkeit)*
- *Widerstandsfähig (Komponenten sind voneinander isoliert, haben eine klare Aufgabenteilung, können automatisiert neu erzeugt werden)*
- *Elastisch (System kann auf ändernde Lasten reagieren -> Durch dynamische Erzeugung von Komponenten)*
- *Nachrichtenorientiert (System verwendet Nachrichten zur Weitergabe von Fehlern und Kommunikation)*

Aufgabe 1.2: Grundlagen Reaktiver Programmierung

Aus welchen drei Bestandteilen (Pattern oder Programmierparadigma) ist die reaktive Programmierung aufgebaut? Erläutere die einzelnen Bestandteile kurz.

1. *Datenflüsse (Daten haben eine zeitliche Reihenfolge und Menge der Daten ist unvorhersehbar)*
2. *Funktionale Programmierung (Operatoren arbeiten mit Datenflüssen, die sind Zustandslos/ haben ein/Ausgabe, keine Nebeneffekte)*
3. *Asynchrone Beobachter (Konsumenten werden durch Produzenten von Daten über Änderungen / Ende des Datenflusses informiert)*

Aufgabe 1.3: Operatoren

Nennen vier Operatoren und erläutere die Funktion.

- 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 sammelt Elemente, indem es das Ergebnis einer Funktion auf das aktuelle und vorherige Element zurückgibt.
- Filtert Elemente aus dem Datenstrom, denen innerhalb eines Zeitlimits neuere Elemente gefolgt sind.

Aufgabe 2: Suspend Functions

Aufgabe 2.1: Starten einer suspend Function

Rufe die Funktion `datasource.getLatestValue` auf und lasse den Wert in der Konsole ausgeben.

Achte darauf, die Signatur der Main-Methode nicht zu verändern.

```
val datasource = DataSource()
runBlocking {
    println( datasource.getLatestValue() )
}
```

Aufgabe 2.2: Zeitmessung

Mit der Funktion `measureTimeMillis` lässt sich die benötigte Zeit innerhalb des eingeschlossenen Blocks messen. Zur Erläuterung folgendes Beispiel:

```
val time = measureTimeMillis {  
    // run code for time measuring here  
}  
println("Time needed in milliseconds: $time")
```

Rufe die Funktion `datasource.getLatestValue` nun mehrfach auf und lasse die Werte ausgeben, ohne dass die Ausführungszeit um vielfaches steigt.

Lösung:

```
fun main() {  
    val time = measureTimeMillis {  
        runBlocking {  
            launch {  
                println( datasource.getLatestValue())  
            }  
            launch {  
                println( datasource.getLatestValue())  
            }  
            launch {  
                println( datasource.getLatestValue())  
            }  
        }  
    }  
    println("Benötigte Zeit: $time ms")  
}
```

Aufgabe 2.3: Asynchrone Verarbeitung

`datasource.getLatestValue` soll zweifach aufgerufen werden, dessen Werte anschließend Zusammenaddiert werden sollen. Dabei ist darauf zu achten, dass die Ausführung von `datasource.getLatestValue` parallel geschieht, sodass sich die Ausführungszeit nicht verdoppelt.

Lasse auch hier wieder die benötigte Zeit ausgeben.

Lösung:

```
fun main() {  
    val time = measureTimeMillis {  
        runBlocking {  
            var value1 = async { getLatestValue() }  
            var value2 = async { getLatestValue() }  
            println(value1.await() + value2.await())  
        }  
    }  
    println("Benötigte Zeit: $time ms")  
}  
  
suspend fun getLatestValue(): Int {  
    delay(1000)  
    return Random.nextInt(0, 100)  
}
```

Aufgabe 3: Arbeiten mit Kotlin Flow

Bei der Bearbeitung der Aufgaben könnte folgender Auszug aus einem Kotlin Coroutines Cheat Sheet hilfreich sein:

Flow

- Flow is an asynchronous stream of values. Uses `Emit` and `Collect` to send and collect data
- Flow runs in the context that is provided the collector
- The code inside a flow isn't run until the flow is collected
- Flows can be cancelled with `withTimeoutOrNull`
- Useful functions:
 - **transform** - customize data, such as emitting a header first
 - **take** - only take a certain amount of the flow
 - **flowOn** - change the context of preceding code
 - **zip** - combine multiple flows
 - **catch** - catch any exceptions of preceding code
 - **onCompletion** - perform any final tasks after the flow is done

Quelle: <https://dev.to/touchlab/kotlin-coroutines-cheat-sheet-5872>

Aufgabe 3.1: Flowstream ausgeben

Gegeben ist ein Flow, welcher die Fibonacci-Folge nacheinander in einem Flow als Stream emittiert.

Lasse die ersten 12 Fibonaccizahlen in der Konsole ausgeben.

Tipp: Mittels

Lösung:

```
fun main() {
    runBlocking {
        fibonacciFlow.take(12).collect {
            println(it.toString())
        }
    }
}

val fibonacciFlow: Flow<BigInteger> = flow {
    var x = BigInteger.ZERO
    var y = BigInteger.ONE
    while (true) {
        delay(600)
        emit(x)
        x = y.also {
            y += x
        }
    }
}
```

Aufgabe 3.2: Combine

Gegeben sind zwei Flows Numbers und Letters, die unterschiedlich schnell Werte emittieren. Kombiniere diese beiden Flows in einen neuen Flow mit der Funktion `combine` und lasse die Werte ausgeben. Notiere dir die Werte und versuche das Ergebnis (grafisch) nachzuvollziehen.

Tipp: Auf der Seite <https://rxmarbles.com> werden Reaktive Operatoren grafisch dargestellt. Der hier verwendete Operator ist combine latest

Lösung:

```
numbersFlow.combine(lettersFlow) { number, letter ->
    "$number$letter"
}.collect {
    println(it)
}
```

Aufgabe 3.3: Map

Mappen einer Datenbank-Entität in eine Domain-Entität.

Mappe die Datenbankentität *Person* in die Domainentität *Person*.

Rufe den gemappten Flow mit collect auf und lasse ihn in der Konsole ausgeben.

Lösung:

```
val dataSource = DataSource()
val nameFlow = dataSource.getPerson()
runBlocking {
    nameFlow.map {
        Person(
            firstName = it.name.firstName(),
            lastName = it.name.lastName(),
            birthday = it.birthday
        )
    }.collect {
        println(it)
    }
}
```

Aufgabe 3.4: Debounce

Hier soll der Datenfluss einer simulierten Texteingabe so verarbeitet werden, dass die Eingabe erst nach einer längeren Zeit ohne weiteres Tippen ausgegeben wird.

Lösung:

```
fun main() = runBlocking<Unit> {
    simple().debounce(200).filter{x -> !x.isEmpty()}.collect { println(it) }
}
```

Aufgabe 3.5: Vergleich Flow - Async/Await

Gegeben ist ein Code, welcher Mitarbeiter abrufen und diese Aufgaben zuweist. Dieser ist mit Async/Await umgesetzt und soll jetzt stattdessen mit einem Flow umgesetzt werden.

Lösung:

```
fun main() = runBlocking<Unit> {
    order.zip(employee) { a, b -> "$b is assigned to Order $a" }
        .collect { println(it) }
```

```
}
```

```
// Das ist evtl. schon vorgegeben, so dass man sich nur um das abrufen und ausgeben des Flows kümmern muss  
val employee = flowOf("Bob", "Alice", "Peter", "Oscar")  
val order = (1..3).asFlow()
```

Aufgabe 4: Flow in Android

Vorbereitung: Starte Android Studio und öffne das Projekt **Aufgabe4**. Nachdem der Gradle-Task durchlaufen ist, führe die App erstmalig aus.

Aufgabe 4.1: MutableStateFlow

Beim Anlegen eines neuen Tasks ist es nicht möglich, Title und Description einzugeben. Stelle sicher, dass im ViewModel der UI-State des AddEditTaskScreen auf den neuen, eingegebenen Wert aktualisiert wird.

Lösung:

```
fun updateTitle(newTitle: String) {  
    // update the title of the uiState here  
    _uiState.update {  
        it.copy(title = newTitle)  
    }  
}  
  
fun updateDescription(newDescription: String) {  
    // update the description of the uiState here  
    _uiState.update {  
        it.copy(description = newDescription)  
    }  
}
```

Aufgabe 4.2: Transformation eines cold flow in einen hot flow

Beim Drehen des Gerätes wird der View-Lifecycle durchlaufen. In der Task-Ansicht der App kommt es dadurch vor, dass die Tasks nach dem Drehen erneut geladen werden. Finde eine Lösung, welches dies verhindert.

Lösung:

```
.stateIn(  
    scope = viewModelScope,  
    started = SharingStarted.WhileSubscribed(5000),  
    initialValue = TasksUiState(isLoading = true)  
)
```

Aufgabe 4.3: Combine

In der View Task Details werden keine Daten dargestellt.

Erweitere die Combine-Funktion im TaskDetailViewModel, sodass der ausgewählte Task im UiState mit enthalten ist und dargestellt wird.

Lösung:

```
val uiState: StateFlow<TaskDetailUiState> = combine(
    _userMessage, _isLoading, _isTaskDeleted, _task
) { userMessage, isLoading, isTaskDeleted, task ->
    TaskDetailUiState(
        isLoading = isLoading,
        userMessage = userMessage,
        isTaskDeleted = isTaskDeleted,
        task = task
    )
}
```