Syntax Institut

Modul 3 – Android App Entwicklung mit Kotlin

Coroutines

Gliederung

- Threads und Concurrency
- Kotlin Coroutines
- Suspend Funktionen
- Wie werden sie programmiert?

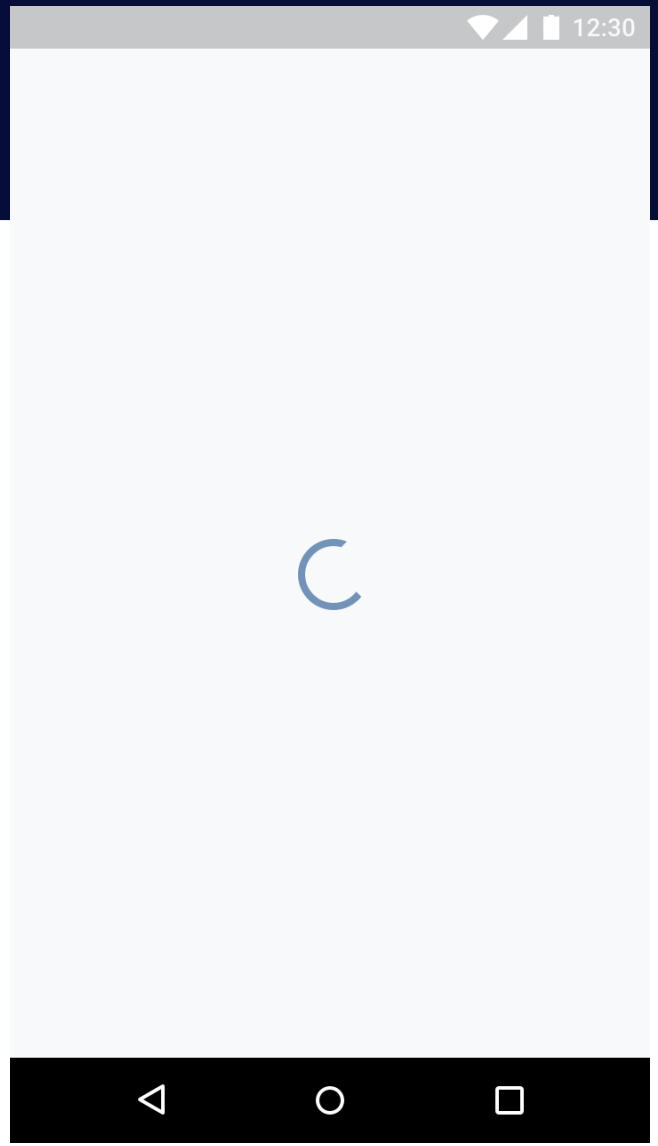
Quelle: https://blog.iamsuleiman.com/stop-using-loading-spinner-theres-something-better/

Threads

Gute Apps sind **immer ansprechbar** und bleiben nie hängen.

Deshalb wird geraten **aufwendige Arbeiten im Hintergrund** durchzuführen ohne den Rest der App zu blockieren.

z.B: Bilder von Server laden



Quelle: https://blog.iamsuleiman.com/stop-using-loading-spinner-theres-something-better/

Threads

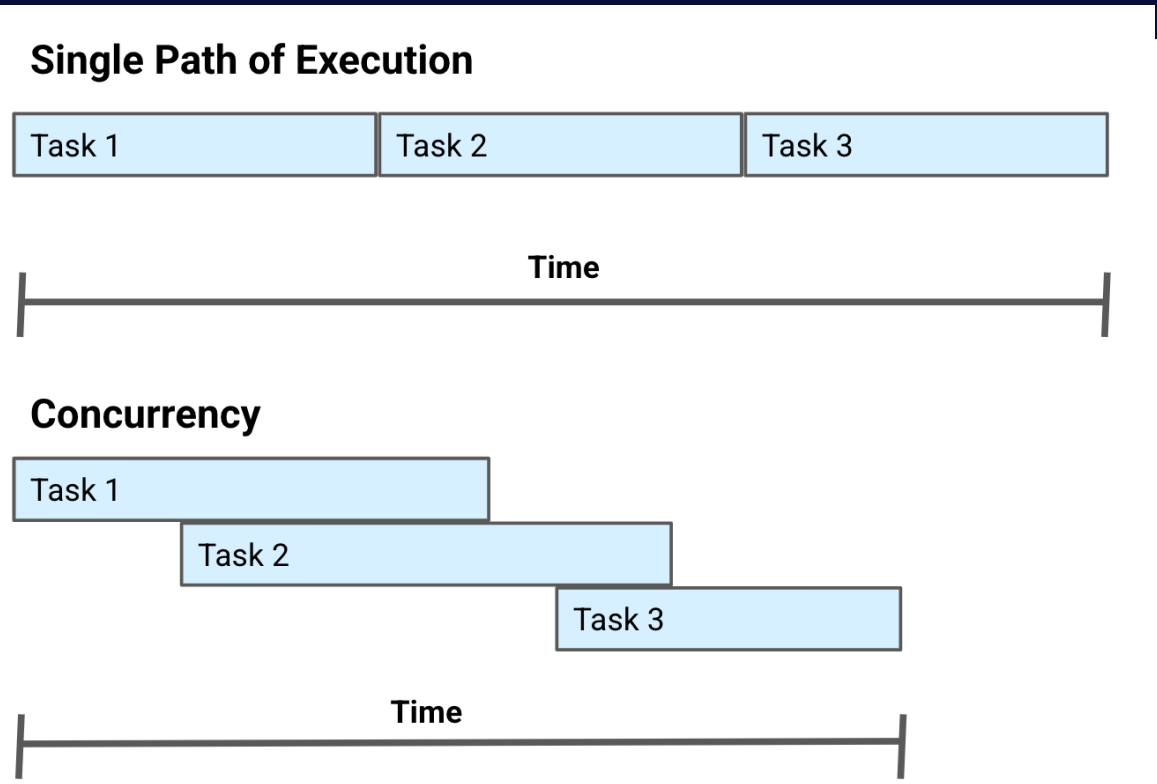
Thread – kleine Einheit an Code die Arbeitsschritt darstellt

```
var count = 0
for (i in 1..50) {
    Thread {
        count += 1
        println("Thread: $i count: $count")
    }.start()
}
```

Concurrency

„Gleichzeitigkeit“

Arbeit wird in Threads aufgeteilt und diese werden **parallel** statt nacheinander durchgeführt



Quelle: https://developer.android.com/codelabs/basic-android-kotlin-training-introduction-coroutines/img/fe71122b40bdb5e3.png

Concurrency

Probleme:

- Viele parallel laufende Threads können **Rechenintensiv** werden
- Es kann keine bestimmte **Reihenfolge** der Durchführung garantiert werden

```
s I/System.out: Thread: 3 count: 1
s I/System.out: Thread: 7 count: 6
s I/System.out: Thread: 4 count: 3
s I/System.out: Thread: 11 count: 8
s I/System.out: Thread: 10 count: 9
s I/System.out: Thread: 13 count: 11
s I/System.out: Thread: 1 count: 5
s I/System.out: Thread: 5 count: 4
s I/System.out: Thread: 15 count: 13
```

Kotlin Coroutines

Kotlin bietet mit Coroutines eine einfachere und flexiblere Möglichkeit Gleichzeitigkeit zu organisieren

```
var coroutineCount = 0
for (i in 1..50) {
    lifecycleScope.launch {  this: CoroutineScope
        coroutineCount += 1
        println("Thread: $i count: $coroutineCount")
    }
}
```

Kotlin Coroutines

Job - abbruchbare Arbeitseinheit - cancel()

CoroutineScope - eine Art Context für Coroutines

Dispatcher - bestimmt den Thread wo die Coroutine durchgeführt wird (Main, Default, IO, Unconfined)
kann mittels **withContext()** gewechselt werden
um z.B. UI Elemente zu beeinflussen

(ohne **Zuweisung** ist der Job nicht abbruchbar)

```
val myJob: Job = lifecycleScope.launch(Dispatchers.Default) {  this: CoroutineScope
    // do something
    withContext(Dispatchers.Main) {  this: CoroutineScope
        // do something that effects UI
    }
    // do another thing
}

myJob.cancel()
```

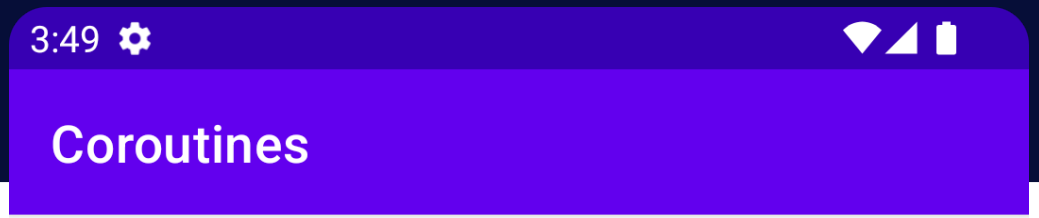
```
lifecycleScope.launch(Dispatchers.Default) {  this: CoroutineScope
    // do something
    withContext(Dispatchers.Main) {  this: CoroutineScope
        // do something that effects UI
    }
    // do another thing
}
```

suspend Funktionen

zeitintensive Funktionen die am Besten im Hintergrund laufen sollten werden mit **suspend** markiert, dadurch werden sie **pausierbar**

```
suspend fun loadImage() {
    // load Image from Server
    delay( timeMillis: 300)
}
```

Beispiel: ProgressBar



- Repository lädt Inhalte innerhalb einer Coroutine

- ProgressBar zeigt Ladevorgang an

Repository.kt

`loadImage()`

wartet 5 Sekunden und lädt anschließend einen neuen String in **_image**

`postValue()`

switcht in den Main Thread um eine Variable zu setzen

(Alternative zu **withContext()**)

```
class Repository {

    private val _image = MutableLiveData<String>()
    val image: LiveData<String>
        get() = _image

    suspend fun loadImage() {
        delay( timeMillis: 5000)
        _image.postValue( value: "stell dir vor ich bin ein Bild :)")
    }
}
```

MainViewModel.kt

`loading`

wird von der View beobachtet um zu wissen wann der Ladebalken angezeigt wird

`loadNewImage()`

startet eine neue Coroutine im **IO Thread** wo die **loading** Variable verändert wird und **loadImage** vom Repository aufgerufen wird

```
class MainViewModel : ViewModel() {

    val repository = Repository()

    val image = repository.image

    private val _loading = MutableLiveData<Boolean>( value: false)
    val loading: LiveData<Boolean>
        get() = _loading

    fun loadNewImage() {
        viewModelScope.launch(Dispatchers.IO) { this: CoroutineScope
            _loading.postValue( value: true)

            repository.loadImage()

            _loading.postValue( value: false)
        }
    }
}
```

MainActivity.kt

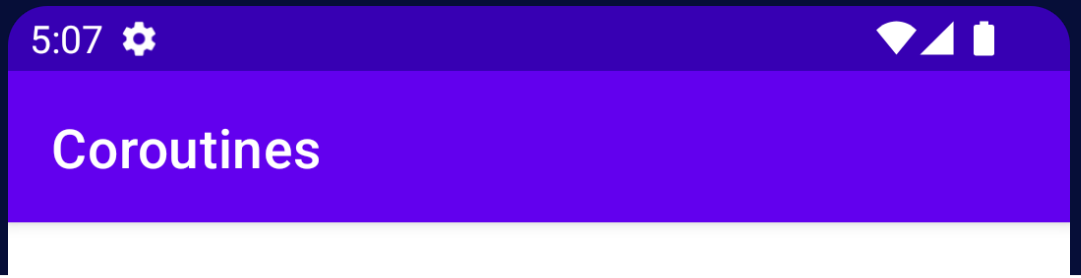
`loading`

wird beobachtet und je nach Wert wird die ProgressBar angezeigt oder ausgeblendet

```
viewModel.image.observe(
    owner: this,
    Observer { it: String!
        binding.imageText.text = it
    }
)

viewModel.loading.observe(
    owner: this,
    Observer { it: Boolean!
        if (it) {
            binding.progressBar.visibility = View.VISIBLE
        } else {
            binding.progressBar.visibility = View.GONE
        }
    }
)
```

fertig



👌

stell dir vor ich bin ein Bild :)

Coroutines

Wiederholung - Was haben wir heute gelernt?

1	Threads und Concurrency
2	Coroutines
3	Anwendung

Quelle: http://www.quickmeme.com/meme/3rpatn

Viel Spaß!