Aplicações assíncronas no Android com

Coroutines & Jetpack

Nelson Glauber
@nglauber

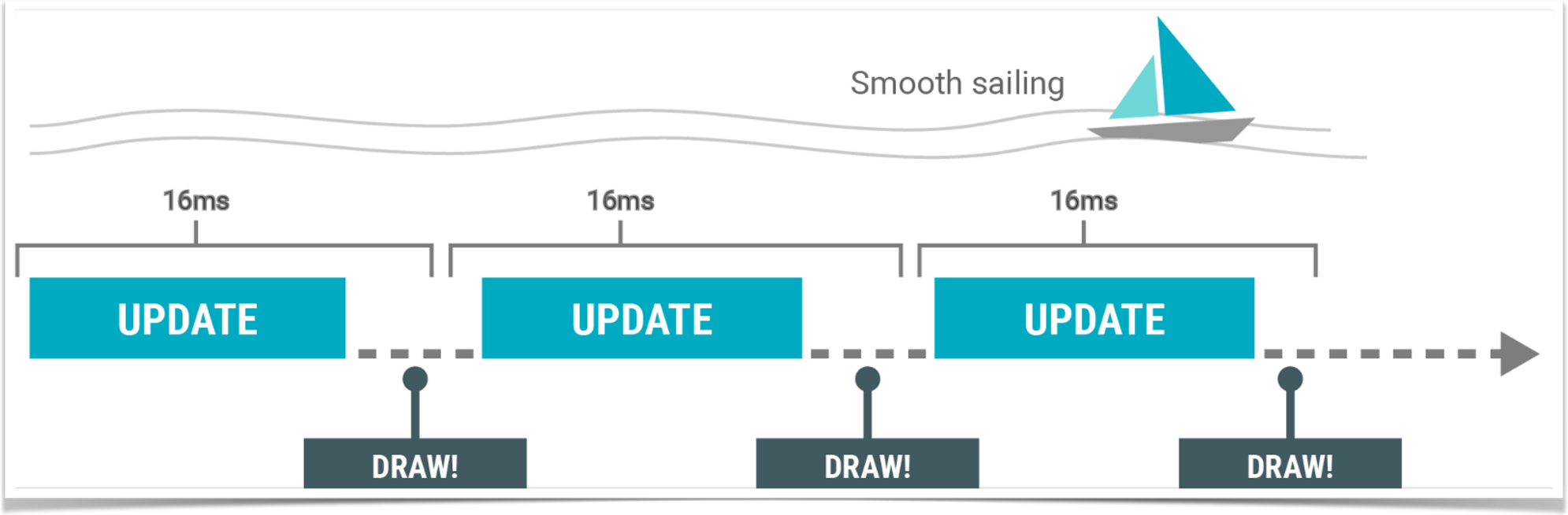
A main thread do Android

- Carregar arquivo de layout
- Desenhar as views
- Tratar os eventos de UI
- …

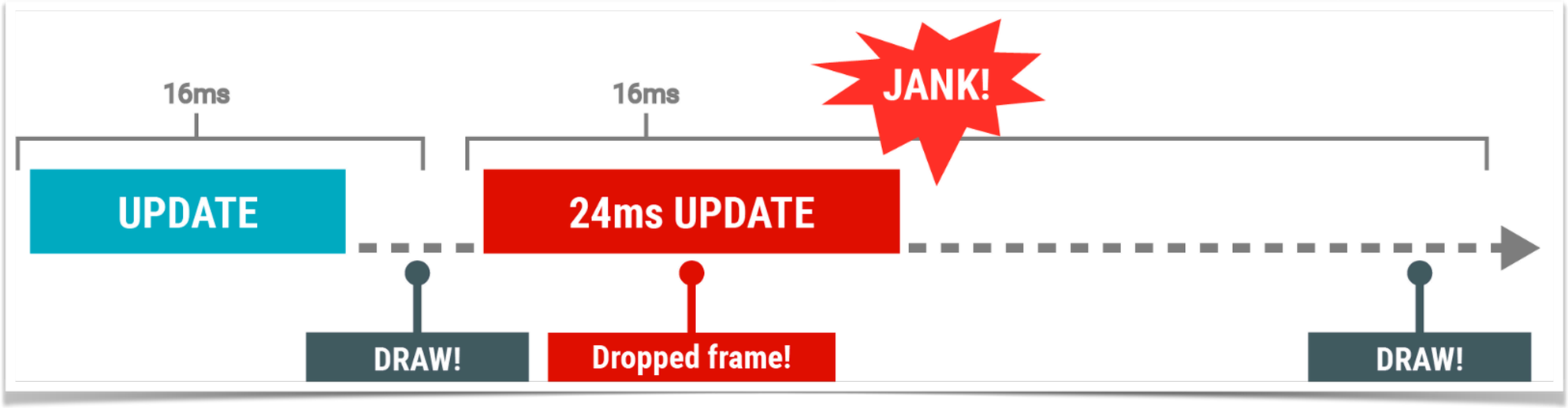
A main thread do Android

- O processamento desses eventos deve ocorrer em:
  - Menos de **16ms** para devices com taxas de atualização de **60Hz**
  - Menos de **12ms** para dispositivos com taxas de **90Hz**
  - Menos de < **8ms** para dispositivos com taxas de **120Hz**

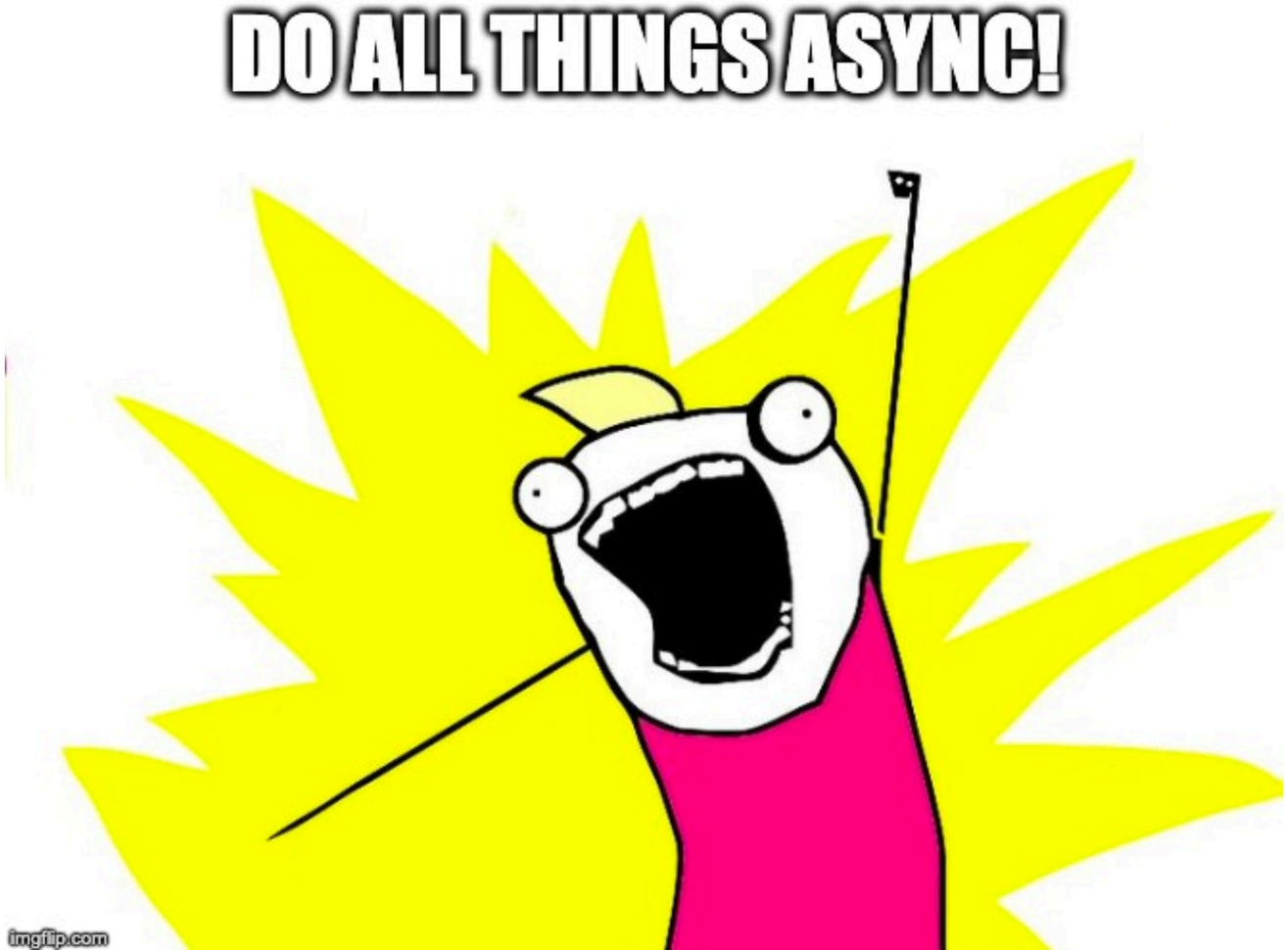
A main thread do Android



A main thread do Android



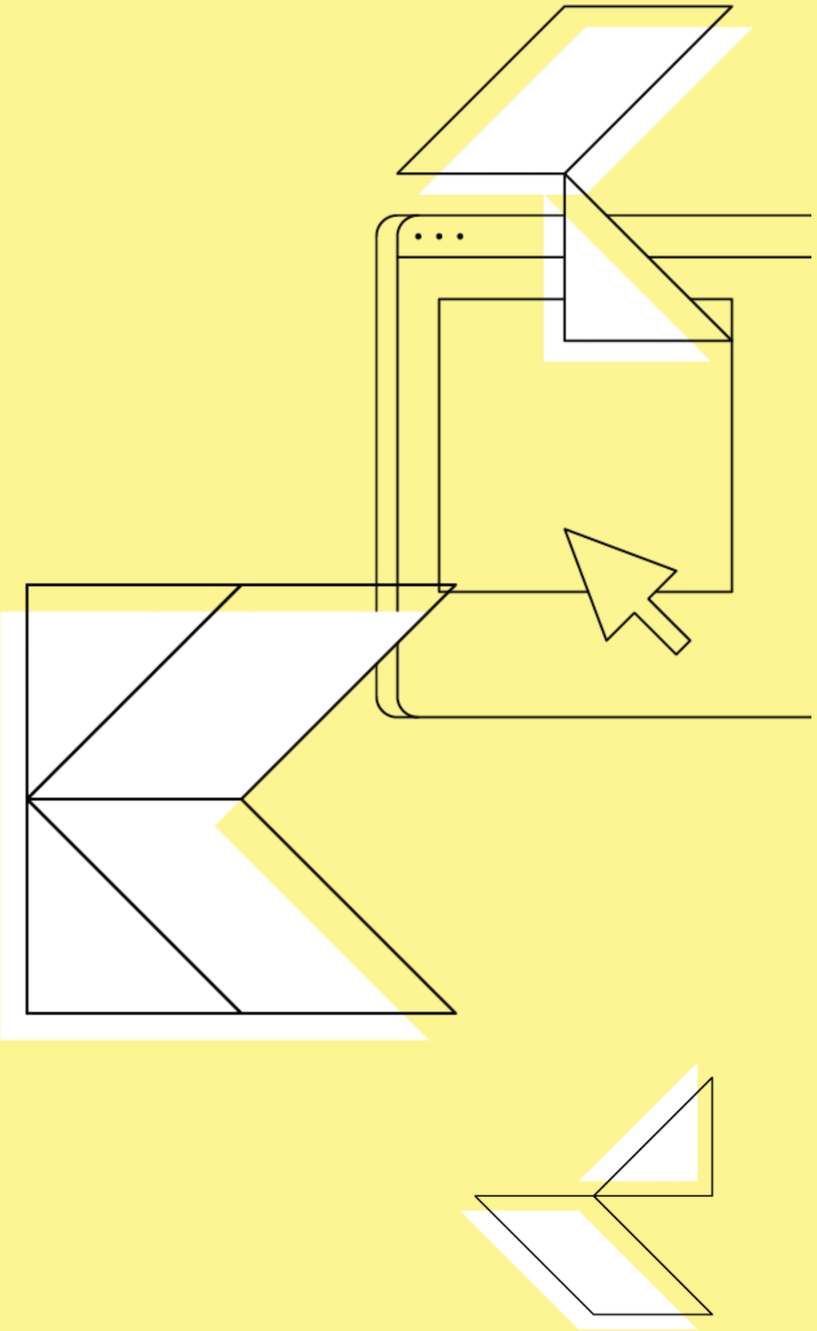
Solução?



Async no Android

- AsyncTask 👴🏽
- Thread + Handler 👷🏻
- Loaders (*deprecated*) 👴🏽
- Volley 🤦🏾‍♀️🤦🏻‍♂️
- RxJava 🧐

Coroutines



#KotlinEverywhere

Coroutines

- Essencialmente, coroutines são *light-weight threads*.
- Fácil de usar (sem mais "*callbacks hell*" e/ou centenas de operadores).
- Úteis para qualquer tarefa computacional mais onerosa (como operações de I/O).
- Permite a substituição de *callbacks* por operações assíncronas.

Dependências

```
dependencies {
    implementation "org.jetbrains.kotlinx:kotlinx-coroutines-core:1.3.5"
    implementation "org.jetbrains.kotlinx:kotlinx-coroutines-android:1.3.5"
    ...
}
```

suspend

- **Suspending functions** são o centro de tudo em Coroutines.
  - São funções que podem ser pausadas e retomadas após algum tempo.
  - Podem executar longas tarefas e aguardar o resultado sem bloquear a thread atual.
  - A sintaxe é idêntica a uma função “normal”, exceto pela adição da palavra reservada **`suspend`**.
  - Por si só, uma *suspending function* não é “assíncrona".
  - Só pode ser chamada a partir de outra *suspending function*.

```kotlin
import kotlinx.coroutines.delay

class Calculator {
    suspend fun sum(a: Int, b: Int): Int {
        delay(5_000)
        return a + b
    }
}
```

```kotlin
import kotlinx.coroutines.delay

class Calculator {
    suspend fun sum(a: Int, b: Int): Int {
        delay(5_000)
        return a + b
    }
}
```

```kotlin
import kotlinx.coroutines.runBlocking
import org.junit.*

class CalculatorUnitTest {
    @Test
    fun sum_isCorrect() = runBlocking {
        val calc = Calculator()
        assertEquals(4, calc.sum(2, 2))
    }
}
```

Principais Classes

- `Job`
- `Context`
- `Scope`
- `Dispatcher`

```
class MainActivity : AppCompatActivity() {
    private val job = Job()
    private val coroutineScope =
        CoroutineScope(job + Dispatchers.Main)

    override fun onDestroy() {
        super.onDestroy()
        job.cancel()
    }
    fun callWebService() {
        coroutineScope.launch {
            txtOutput.text = ""
            val books = withContext(Dispatchers.IO) {
                BookHttp.loadBooks()
            }
            // update the UI using books
        }
    }
    ...
```

```
class MainActivity : AppCompatActivity() {
    private val job = Job()
    private val coroutineScope =
        CoroutineScope(job + Dispatchers.Main)

    override fun onDestroy() {
        super.onDestroy()
        job.cancel()
    }
    fun callWebService() {
        coroutineScope.launch {
            txtOutput.text = ""
            val books = withContext(Dispa
                BookHttp.loadBooks()
            }
            // update the UI using books
        }
    }
    ...
```

Job

- Um **`Job`** representa uma tarefa ou conjunto de tarefas em execução.
- Pode possuir “filhos”.
- A função **`launch`** retorna um **`Job`**.
- Pode ser cancelado usando a função **`cancel`**.
- Possui um ciclo de vida (novo, ativo, completo ou cancelado)

```
class MainActivity : AppCompatActivity() {
    private val job = Job()
    private val coroutineScope =
        CoroutineScope(job + Dispatchers.Main)

    override fun onDestroy() {
        super.onDestroy()
        job.cancel()
    }
    fun callWebService() {
        coroutineScope.launch {
            txtOutput.text = ""
            val books = withContext(Dispat
                BookHttp.loadBooks()
            }
            // update the UI using books
        }
    }
    ...
```

Context

- A interface **CoroutineContext** Representa o conjunto de atributos que configuram uma coroutine.
- Pode definir a política de threading; job raiz; tratamento de exceções; nome da coroutine (debug).
- Uma coroutine herda o contexto do pai.

```
class MainActivity : AppCompatActivity() {
    private val job = Job()
    private val coroutineScope =
        CoroutineScope(job + Dispatchers.Main)

    override fun onDestroy() {
        super.onDestroy()
        job.cancel()
    }
    fun callWebService() {
        coroutineScope.launch {
            txtOutput.text = ""
            val books = withContext(Dis
                BookHttp.loadBooks()
            }
            // update the UI using book
        }
    }
    ...
```

Scope

- Uma coroutine **sempre** roda em um escopo.
- Serve como uma espécie de ciclo de vida para um conjunto de coroutines.
- Permite um maior controle das tarefas em execução.

Dispatcher

- Define o pool de threads onde a coroutine executará.
  - **`Default`:** para processos que usam a CPU mais intensamente.
  - **`IO`**: para tarefas de rede ou arquivos. O *pool de threads* é compartilhado com o dispatcher `Default`.
  - **`Main`** - main thread do Android.

```
CompatActivity() {
o()
eScope =
ob + Dispatchers.Main)

by() {
)


 {
aunch {
xt = ""
withContext(Dispatchers.IO) {
 loadBooks()

e UI using books
```

```
class MainActivity : AppCompatActivity() {
    private val job = Job()
    private val coroutineScope =
        CoroutineScope(job + Dispatchers.Main)

    override fun onDestroy() {
        super.onDestroy()
        job.cancel()
    }
    fun callWebService() {
        coroutineScope.launch {
            txtOutput.text = ""
            val books = withContext(Dispatchers.IO) {
                BookHttp.loadBooks()
            }
            // update the UI using books
        }
    }
    ...
```

```
class MainActivity : AppCompatActivity() {
    private val job = Job()
    private val coroutineScope =
        CoroutineScope(job + Dispatchers.Main)

    override fun onDestroy() {
        super.onDestroy()
        job.cancel()
    }
    fun callWebService() {
        coroutineScope.launch {
            txtOutput.text = ""
            val books = withContext(Dispatchers.IO) {
                BookHttp.loadBooks()
            }
            // update the UI using books
        }
    }
    ...
```

```
fun callWebService() {
    coroutineScope.launch {
        txtOutput.text = ""
        val books = withContext(Dispatchers.IO) {
            BookHttp.loadBooks()
        }
        // update the UI using books
    }
}
```

```kotlin
fun callWebService() {
    coroutineScope.launch {
        txtOutput.text = ""
        val books = BookHttp.loadBooks()
        // update the UI using books
    }
}
```

```
android.os.NetworkOnMainThreadException
   at android.os.StrictMode$AndroidBlockGuardPolicy.onNetwork(StrictMode.java:1513)
   at java.net.Inet6AddressImpl.lookupHostByName(Inet6AddressImpl.java:117)
   at java.net.Inet6AddressImpl.lookupAllHostAddr(Inet6AddressImpl.java:105)
   at java.net.InetAddress.getAllByName(InetAddress.java:1154)
   at com.android.okhttp.Dns$1.lookup(Dns.java:39)
```

```kotlin
fun callWebService() {
    coroutineScope.launch(Dispatchers.IO){
        txtOutput.text = ""
        val books = BookHttp.loadBooks()
        // update the UI using books
    }
}
```

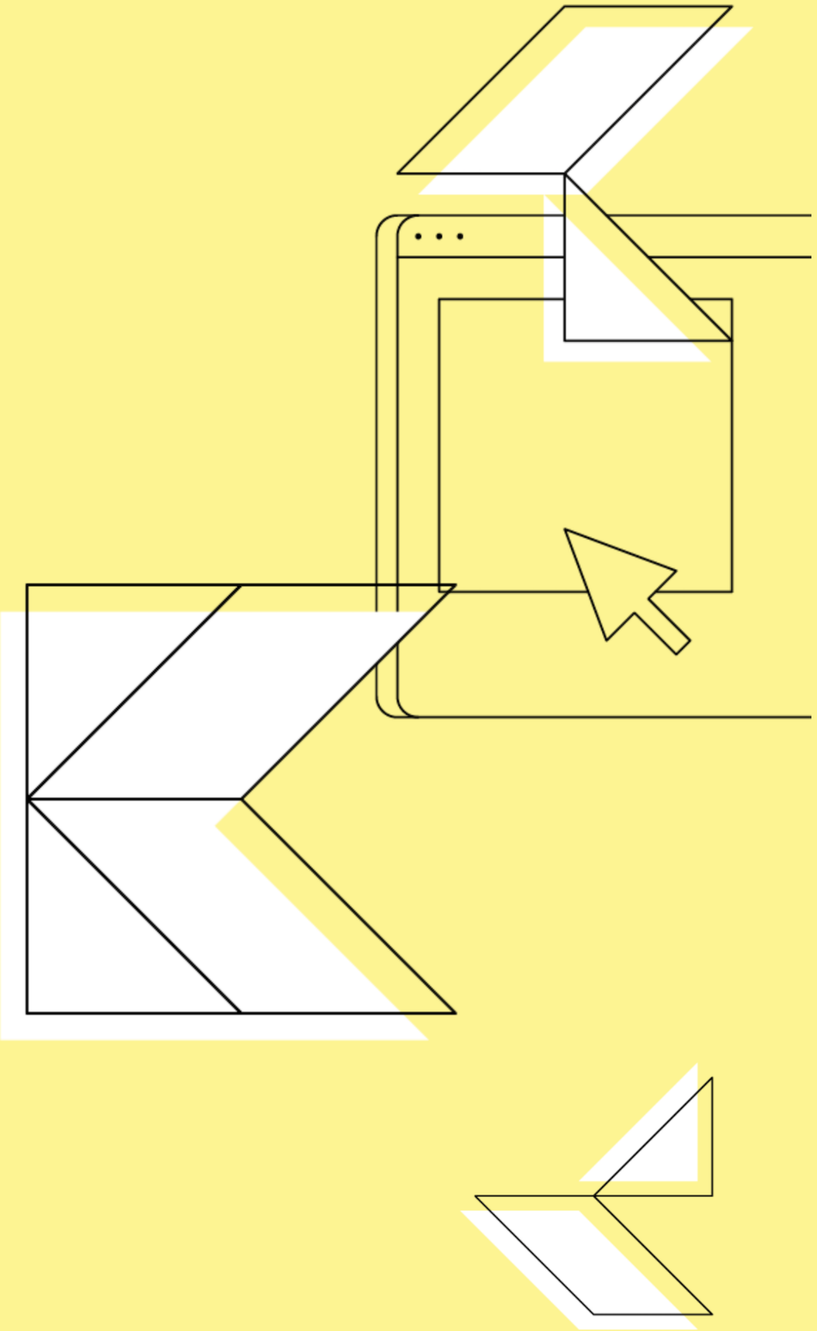
```
FATAL EXCEPTION: DefaultDispatcher-worker-1
   Process: br.com.nglauber.coroutinesdemo, PID: 26507
   android.view.ViewRootImpl$CalledFromWrongThreadException: Only the original thread that created
a view hierarchy can touch its views.
       at android.view.ViewRootImpl.checkThread(ViewRootImpl.java:7753)
       at android.view.ViewRootImpl.requestLayout(ViewRootImpl.java:1225)
```

```
fun callWebService() {
    coroutineScope.launch {
        txtOutput.text = ""
        val books = withContext(Dispatchers.IO) {
            BookHttp.loadBooks()
        }
        // update the UI using books
    }
}
```

Coroutines

- Suspending functions
- Job
- Context
- Scope
- Dispatcher

Lifecycle



#KotlinEverywhere

Lifecycle Scope

- É possível iniciar coroutines atrelada aos ciclos de vida de Activity, Fragment e View do Fragment.

- Além da função **`launch`**, podemos usar o **`launchWhenCreated`**, **`launchWhenStarted`** *e* **`launchWhenResumed`**.

```
dependencies {
    ...
    implementation "androidx.lifecycle:lifecycle-runtime-ktx:2.2.0"
}
```

```
class MyActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        lifecycleScope.launch {
            ...
        }
        lifecycleScope.launchWhenCreated {
            ...
        }
        lifecycleScope.launchWhenStarted {
            ...
        }
        lifecycleScope.launchWhenResumed {
            ...
        }
    }
}
```

```
class MyActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        lifecycleScope.launch {
            ...
        }
        lifecycleScope.launchWhenCreated {
            ...
        }
        lifecycleScope.launchWhenStarted {
            ...
        }
        lifecycleScope.launchWhenResumed {
            ...
        }
    }
}
```

```kotlin
class MyFragment : Fragment() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        lifecycleScope.launch {
            ...
        }
        lifecycleScope.launchWhenCreated {
            ...
        }
        lifecycleScope.launchWhenStarted {
            ...
        }
        lifecycleScope.launchWhenResumed {
            ...
        }
    }
}
```

```
class MyFragment : Fragment() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        lifecycleScope.launch {
            ...
        }
        lifecycleScope.launchWhenCreated {
            ...
        }
        lifecycleScope.launchWhenStarted {
            ...
        }
        lifecycleScope.launchWhenResumed {
            ...
        }
    }
}
```

```kotlin
class MyFragment: Fragment() {
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        viewLifecycleOwner.lifecycleScope.launch {
            ...
        }
    }
}
```

```
class MyFragment: Fragment() {
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        viewLifecycleOwner.lifecycleScope.launch {
            ...
        }
    }
}
```

Ambos os escopos são cancelados automaticamente.

ViewModel

#KotlinEverywhere

ViewModel Scope

A classe **`ViewModel`** possui agora a propriedade **`viewModelScope`**.

```
dependencies {
    implementation "androidx.lifecycle:lifecycle-extensions:2.2.0"
    implementation "androidx.lifecycle:lifecycle-viewmodel-ktx:2.2.0"
}
```

```
class BookListViewModel: ViewModel() {
    private val _state = MutableLiveData<State>()
    val state: LiveData<State>
        get() = _state

    fun search(query: String) {
        viewModelScope.launch {
            _state.value = State.StateLoading
            val result = withContext(Dispatchers.IO) {
                BookHttp.searchBook(query)
            }
            _state.value = if (result?.items != null) {
                State.StateLoaded(result.items)
            } else {
                State.StateError(Exception("Error"), false)
            }
        }
    }
}
```

```
class BookListViewModel: ViewModel() {
    private val _state = MutableLiveData<State>()
    val state: LiveData<State>
        get() = _state

    fun search(query: String) {
        viewModelScope.launch {
            _state.value = State.StateLoading
            val result = withContext(Dispatchers.IO) {
                BookHttp.searchBook(query)
            }
            _state.value = if (result?.items != null) {
                State.StateLoaded(result.items)
            } else {
                State.StateError(Exception("Error"), false)
            }
        }
    }
}
```

```kotlin
class BookListViewModel: ViewModel() {
    private val _state = MutableLiveData<State>()
    val state: LiveData<State>
        get() = _state

    fun search(query: String) {
        viewModelScope.launch {
            _state.value = State.StateLoading
            val result = withContext(Dispatchers.IO) {
                BookHttp.searchBook(query)
            }
            _state.value = if (result?.items != null) {
                State.StateLoaded(result.items)
            } else {
                State.StateError(Exception("Error"), false)
            }
        }
    }
}
```

```kotlin
class BookListViewModel: ViewModel() {
    private var query = MutableLiveData<String>()
    val state = query.switchMap {
        liveData {
            emit(State.StateLoading)
            val result = withContext(Dispatchers.IO) {
                BookHttp.searchBook(it)
            }
            emit(
                if (result?.items != null) State.StateLoaded(result.items)
                else State.StateError(Exception("Error"), false)
            )
        }
    }

    fun search(query: String) {
        this.query.value = query
    }
}
```

```
class BookListViewModel: ViewModel() {
    private var query = MutableLiveData<String>()
    val state = query.switchMap {
        liveData {
            emit(State.StateLoading)
            val result = withContext(Dispatchers.IO) {
                BookHttp.searchBook(it)
            }
            emit(
                if (result?.items != null) State.StateLoaded(result.items)
                else State.StateError(Exception("Error"), false)
            )
        }
    }

    fun search(query: String) {
        this.query.value = query
    }
}
```

```
class BookListViewModel: ViewModel() {
    private var query = MutableLiveData<String>()
    val state = query.switchMap {
        liveData {
            emit(State.StateLoading)
            val result = withContext(Dispatchers.IO) {
                BookHttp.searchBook(it)
            }
            emit(
                if (result?.items != null) State.StateLoaded(result.items)
                else State.StateError(Exception("Error"), false)
            )
        }
    }

    fun search(query: String) {
        this.query.value = query
    }
}
```

```
class BookListViewModel: ViewModel() {
    private var query = MutableLiveData<String>()
    val state = query.switchMap {
        liveData {
            emit(State.StateLoading)
            val result = withContext(Dispatchers.IO) {
                BookHttp.searchBook(it)
            }
            emit(
                if (result?.items != null) State.StateLoaded(result.items)
                else State.StateError(Exception("Error"), false)
            )
        }
    }

    fun search(query: String) {
        this.query.value = query
    }
}
```

```kotlin
class BookListActivity : AppCompatActivity(R.layout.activity_book_list) {

    private val viewModel: BookListViewModel …

    override fun onCreate(savedInstanceState: Bundle?) {
        viewModel.state.observe(this, Observer { state ->
            when (state) {
                is BookListViewModel.State.StateLoading -> ...
                is BookListViewModel.State.StateLoaded -> ...
                is BookListViewModel.State.StateError -> ...
            }
        })
    }
    ...
```

```kotlin
class BookListActivity : AppCompatActivity(R.layout.activity_book_list) {

    private val viewModel: BookListViewModel …

    override fun onCreate(savedInstanceState: Bundle?) {
        viewModel.state.observe(this, Observer { state ->
            when (state) {
                is BookListViewModel.State.StateLoading -> ...
                is BookListViewModel.State.StateLoaded -> ...
                is BookListViewModel.State.StateError -> ...
            }
        })
    }
    ...
```

WorkManager

#KotlinEverywhere

WorkManager

```
dependencies {
    def work_version = "2.3.4"
    implementation "androidx.work:work-runtime-ktx:$work_version"
}
```

```kotlin
class MyWork(context: Context, params: WorkerParameters) :
    CoroutineWorker(context, params) {

    override suspend fun doWork(): Result = try {
        val output = inputData.run {
            val x = getInt("x", 0)
            val y = getInt("y", 0)
            val result = Calculator().sum(x, y)
            workDataOf("result" to result)
        }
        Result.success(output)
    } catch (error: Throwable) {
        Result.failure()
    }
}
```

```kotlin
class MyWork(context: Context, params: WorkerParameters) :
    CoroutineWorker(context, params) {

    override suspend fun doWork(): Result = try {
        val output = inputData.run {
            val x = getInt("x", 0)
            val y = getInt("y", 0)
            val result = Calculator().sum(x, y)
            workDataOf("result" to result)
        }
        Result.success(output)
    } catch (error: Throwable) {
        Result.failure()
    }
}
```

```
private fun callMyWork() {
    val request =
        OneTimeWorkRequestBuilder<MyWork>()
            .setInputData(workDataOf("x" to 84, "y" to 12))
            .build()

    WorkManager.getInstance(this).run {
        enqueue(request)
        getWorkInfoByIdLiveData(request.id)
            .observe(this@MainActivity, Observer {
                if (it.state == WorkInfo.State.SUCCEEDED) {
                    val result = it.outputData.getInt("result", 0)
                    addTextToTextView("Result-> $result")
                }
            })
    }
}
```

```
private fun callMyWork() {
    val request =
        OneTimeWorkRequestBuilder<MyWork>()
            .setInputData(workDataOf("x" to 84, "y" to 12))
            .build()

    WorkManager.getInstance(this).run {
        enqueue(request)
        getWorkInfoByIdLiveData(request.id)
            .observe(this@MainActivity, Observer {
                if (it.state == WorkInfo.State.SUCCEEDED) {
                    val result = it.outputData.getInt("result", 0)
                    addTextToTextView("Result-> $result")
                }
            })
    }
}
```

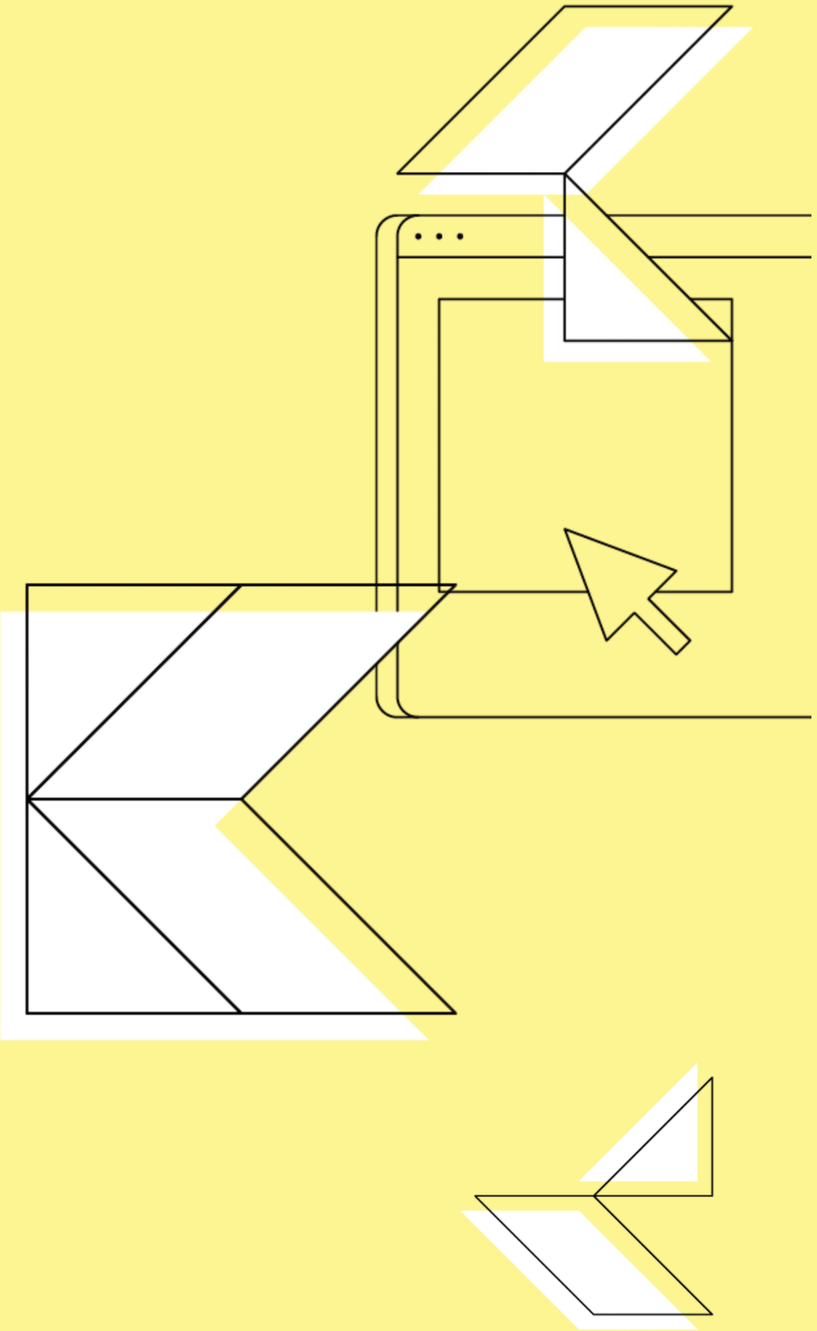
```
private fun callMyWork() {
    val request =
        OneTimeWorkRequestBuilder<MyWork>()
            .setInputData(workDataOf("x" to 84, "y" to 12))
            .build()

    WorkManager.getInstance(this).run {
        enqueue(request)
        getWorkInfoByIdLiveData(request.id)
            .observe(this@MainActivity, Observer {
                if (it.state == WorkInfo.State.SUCCEEDED) {
                    val result = it.outputData.getInt("result", 0)
                    addTextToTextView("Result-> $result")
                }
            })
    }
}
```

```
private fun callMyWork() {
    val request =
        OneTimeWorkRequestBuilder<MyWork>()
            .setInputData(workDataOf("x" to 84, "y" to 12))
            .build()

    WorkManager.getInstance(this).run {
        enqueue(request)
        getWorkInfoByIdLiveData(request.id)
            .observe(this@MainActivity, Observer {
                if (it.state == WorkInfo.State.SUCCEEDED) {
                    val result = it.outputData.getInt("result", 0)
                    addTextToTextView("Result-> $result")
                }
            })
    }
}
```

Jetpack + Coroutines

- Lifecycle provê um **`lifecycleScope`** para Activity e Fragment (e a view do Fragment).
- ViewModel possui a propriedade **`viewModelScope`**.
- WorkManager disponibiliza a classe **`CoroutineWorker`**.

Coroutines - Parte 2



#KotlinEverywhere

Iniciando uma coroutine

- As duas formas de iniciar uma coroutine são:
  - A função **`launch`** é uma *"fire and forget"* que significa que não retornará o resultado para que a chamou (mas retornará um **`Job`**).
  - A função **`async`** retorna um objeto **`Deferred`** que permite obter o seu resultado.

launch

```
launch {
    txtOutput.text = ""
    val time = measureTimeMillis {
        val one = withContext(Dispatchers.IO) { loadFirstNumber() }
        val two = withContext(Dispatchers.IO) { loadSecondNumber() }

        addTextToTextView("The answer is ${one + two}")
    }
    addTextToTextView("Completed in $time ms")
}
```

```
The answer is 42
Completed in 2030 ms
```

async

```
launch {
    txtOutput.text = ""
    val time = measureTimeMillis {
        val one = async(Dispatchers.IO) { loadFirstNumber() }
        val two = async(Dispatchers.IO) { loadSecondNumber() }

        val s = one.await() + two.await()
        addTextToTextView("The answer is $s")
    }
    addTextToTextView("Completed in $time ms")
}
```

async

```
launch {
    txtOutput.text = ""
    val time = measureTimeMillis {
        val one = async(Dispatchers.IO) { loadFirstNumber() }
        val two = async(Dispatchers.IO) { loadSecondNumber() }

        val s = one.await() + two.await()
        addTextToTextView("The answer is $s")
    }
    addTextToTextView("Completed in $time ms")
}
```

async

```
launch {
    txtOutput.text = ""
    val time = measureTimeMillis {
        val one = async(Dispatchers.IO) { loadFirstNumber() }
        val two = async(Dispatchers.IO) { loadSecondNumber() }

        val s = one.await() + two.await()
        addTextToTextView("The answer is $s")
    }
    addTextToTextView("Completed in $time ms")
}
```

async

```
launch {
    txtOutput.text = ""
    val time = measureTimeMillis {
        val one = async(Dispatchers.IO) { loadFirstNumber() }
        val two = async(Dispatchers.IO) { loadSecondNumber() }

        val s = one.await() + two.await()
        addTextToTextView("The answer is $s")
    }
    addTextToTextView("Completed in $time ms")
}
```

```
The answer is 42
Completed in 1038 ms
```

Exceptions

- O tratamento de exceções é simples, basta tratar no lugar certo!
- A falha de um **`Job`** cancelará o seu “pai" e os demais “filhos"
- As exceções não tratadas são propagadas para o **`Job`** do escopo.
- Um escopo cancelado não poderá iniciar coroutines.

Exceptions

```
launch {
    txtOutput.text = ""
    try {
        val result = methodThatThrowsException()
        addTextToTextView("Ok $result")
    } catch (e: Exception) {
        addTextToTextView("Error! ${e.message}")
    }
}
```

Exceptions

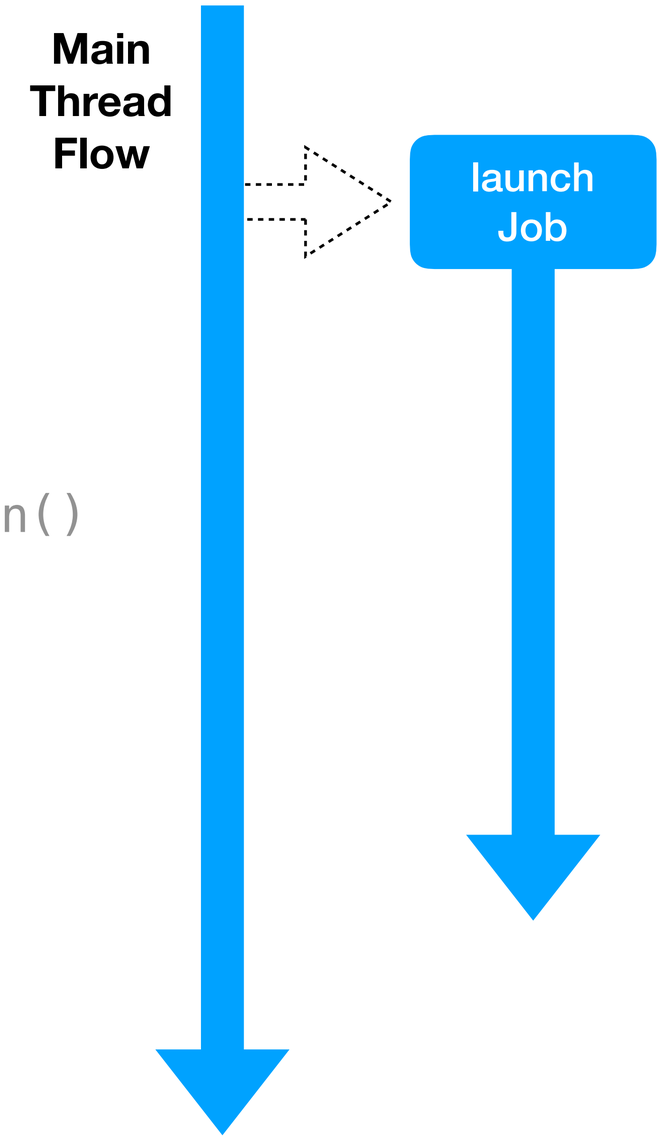
```
launch {
    txtOutput.text = ""
    try {
        launch {
            val result = methodThatThrowsException()
            addTextToTextView("Ok $result")
        }
    } catch (e: Exception) {
        addTextToTextView("Error! ${e.message}")
    }
}
```

Exceptions

```
launch {
    txtOutput.text = ""
    try {
        launch {
            val result = methodThatThrowsException()
            addTextToTextView("Ok $result")
        }
    } catch (e: Exception) {
        addTextToTextView("Error! ${e.message}")
    }
}
```

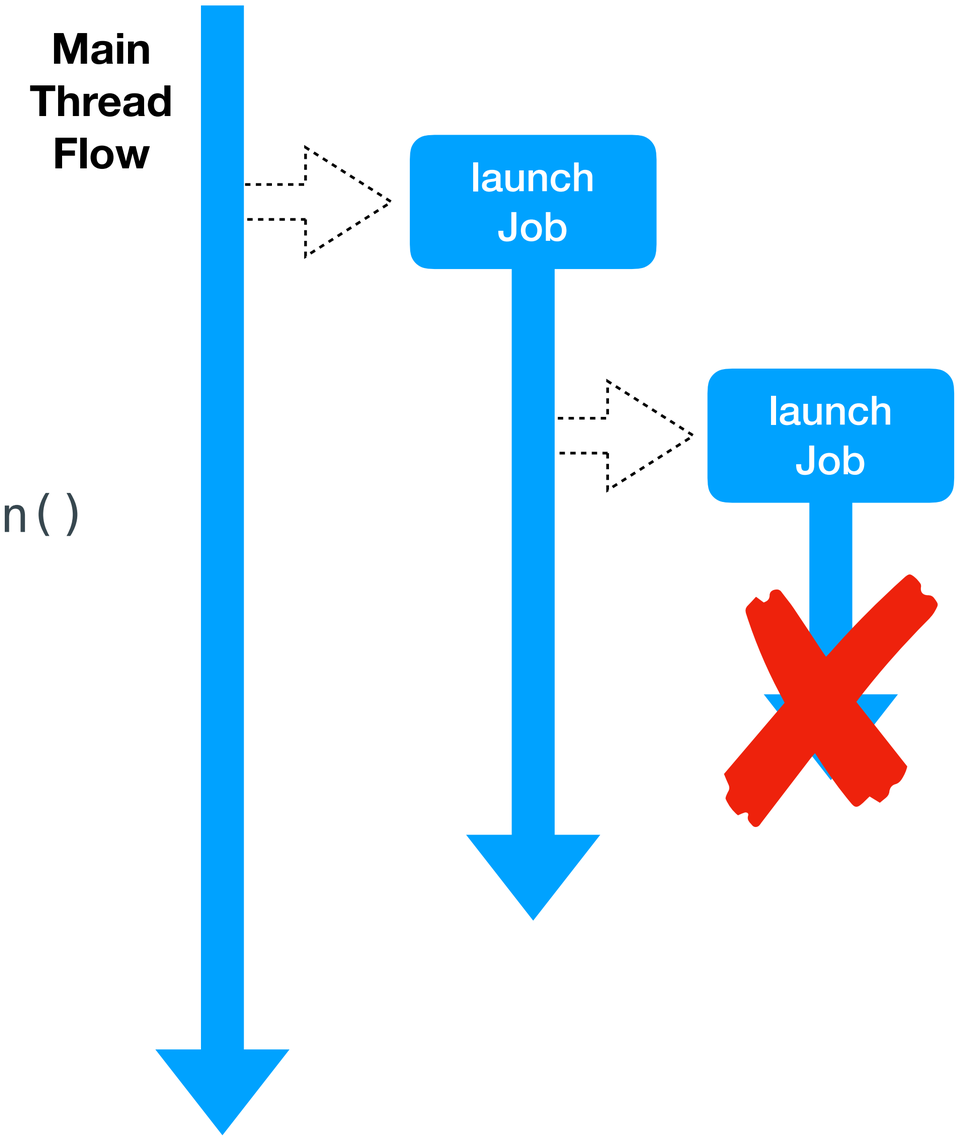
Exceptions

```
launch {
    txtOutput.text = ""
    try {
        launch {
            val result = methodThatThrowsException()
            addTextToTextView("Ok $result")
        }
    } catch (e: Exception) {
        addTextToTextView("Error! ${e.message}")
    }
}
```



Exceptions

```
launch {
    txtOutput.text = ""
    try {
        launch {
            val result = methodThatThrowsException()
            addTextToTextView("Ok $result")
        }
    } catch (e: Exception) {
        addTextToTextView("Error! ${e.message}")
    }
}
```



Exceptions

```
launch {
    txtOutput.text = ""
    launch {
        try {
            val result = methodThatThrowsException()
            addTextToTextView("Ok $result")
        } catch (e: Exception) {
            addTextToTextView("Error! ${e.message}")
        }
    }
}
```

Exceptions

```
launch {
    txtOutput.text = ""
    val task = async { methodThatThrowsException() }
    try {
        val result = task.await()
        addTextToTextView("Ok $result")
    } catch (e: Exception) {
        addTextToTextView("Error! ${e.message}")
    }
}
```

Exceptions

```
launch {
    txtOutput.text = ""
    val task = async {
        try {
            methodThatThrowsException()
        } catch (e: Exception) {
            "Error! ${e.message}"
        }
    }
    val result = task.await()
    addTextToTextView("Ok $result")
}
```

Exceptions

```
launch {
    txtOutput.text = ""
    val task = async(SupervisorJob(job)) {
        methodThatThrowsException()
    }
    try {
        addTextToTextView("Ok ${task.await()}")
    } catch (e: Throwable) {
        addTextToTextView("Error! ${e.message}")
    }
}
```

Exceptions

```
launch {
    txtOutput.text = ""
    try {
        coroutineScope {
            val task = async {
                methodThatThrowsException()
            }
            addTextToTextView("Ok ${task.await()}")
        }
    } catch (e: Throwable) {
        addTextToTextView("Erro! ${e.message}")
    }
}
```

Exceptions

```
launch {
    txtOutput.text = ""
    supervisorScope {
        val task = async { methodThatThrowsException() }
        try {
            addTextToTextView("Ok ${task.await()}")
        } catch (e: Throwable) {
            addTextToTextView(“Error! ${e.message}")
        }
    }
}
```

Cancelamento

- Para cancelar um job, basta chamar o método **`cancel`**.
- Uma vez cancelado o job não pode ser reusado.
- Para cancelar os jobs filhos, use **`cancelChildren`**.
- A propriedade **`isActive`** indica que o job está em execução, **`isCancelled`** se a coroutine foi cancelada, e **`isCompleted`** terminou sua execução.

withTimeout

- Executa uma coroutine levantando uma **`TimeoutCancellationException`** caso sua duração exceda o tempo especificado.

- Uma vez que o cancelamento é apenas uma exceção, é possível trata-la facilmente.

- É possível usar a função **`withTimeoutOrNull`** que é similar a **`withTimeout`**, mas retorna null ao invés de levantar a exceção.

withTimeout

```
launch {
    txtOutput.text = ""
    try {
        val s = withTimeout(1300L) {
            withContext(Dispatchers.Default) {
                aLongOperation()
            }
        }
        txtOutput.text = "Result: $s..."
    } catch (e: TimeoutCancellationException) {
        txtOutput.text = "Exception! ${e.message}"
    }
}
```

withTimeout

```
launch {
    txtOutput.text = ""
    try {
        val s = withTimeout(1300L) {
            withContext(Dispatchers.Default) {
                aLongOperation()
            }
        }
        txtOutput.text = "Result: $s..."
    } catch (e: TimeoutCancellationException) {
        txtOutput.text = "Exception! ${e.message}"
    }
}
```

withTimeoutOrNull

```
launch {
    txtOutput.text = ""
    val task = async(Dispatchers.Default) {
        aLongOperation()
    }
    val result = withTimeoutOrNull(1300L) { task.await() }
    txtOutput.text = "Result: $result"
}
```

Convertendo Callbacks

- Nos bastidores, uma *suspending function* é convertida pelo compilador para uma função (de mesmo nome) que recebe um objeto do tipo **`Continuation`**.

```
fun sum(a: Int, b: Int, Continuation<Int>)
```

- **`Continuation`** é uma interface que contém duas funções que são invocadas para continuar com a execução da coroutine (normalmente retornando um valor) ou levantar uma exceção caso algum erro ocorra.

```
interface Continuation<in T> {
    val context: CoroutineContext
    fun resume(value: T)
    fun resumeWithException(exception: Throwable)
}
```

Convertendo Callbacks

```
object LocationManager {
    fun getCurrentLocation(callback: (LatLng?) -> Unit) {
        // get the location...
        callback(LatLng(-8.187,-36.156))
    }
}
```

```
LocationManager.getCurrentLocation { latLng ->
    if (latLng != null) {
        // Exibir localização
    } else {
        // Tratar o erro
    }
}
```

Convertendo Callbacks

```
suspend fun getMyLocation(): LatLng {
    return suspendCoroutine { continuation ->
        LocationManager.getCurrentLocation { latLng ->
            if (latLng != null) {
                continuation.resume(latLng)

            } else {
                continuation.resumeWithException(
                    Exception("Fail to get user location")
                )
            }
        }
    }
}
```

Convertendo Callbacks

```
suspend fun getMyLocation(): LatLng {
    return suspendCoroutine { continuation ->
        LocationManager.getCurrentLocation { latLng ->
            if (latLng != null) {
                continuation.resume(latLng)

            } else {
                continuation.resumeWithException(
                    Exception("Fail to get user location")
                )
            }
        }
    }
}
```

Convertendo Callbacks

```
suspend fun getMyLocation(): LatLng {
    return suspendCoroutine { continuation ->
        LocationManager.getCurrentLocation { latLng ->
            if (latLng != null) {
                continuation.resume(latLng)

            } else {
                continuation.resumeWithException(
                    Exception("Fail to get user location")
                )
            }
        }
    }
}
```

Convertendo Callbacks

```
suspend fun getMyLocation(): LatLng {
    return suspendCancellableCoroutine { continuation ->
        LocationManager.getCurrentLocation { latLng ->
            if (latLng != null) {
                continuation.resume(latLng)

            } else {
                continuation.resumeWithException(
                    Exception("Fail to get user location")
                )
            }
        }
    }
}
```

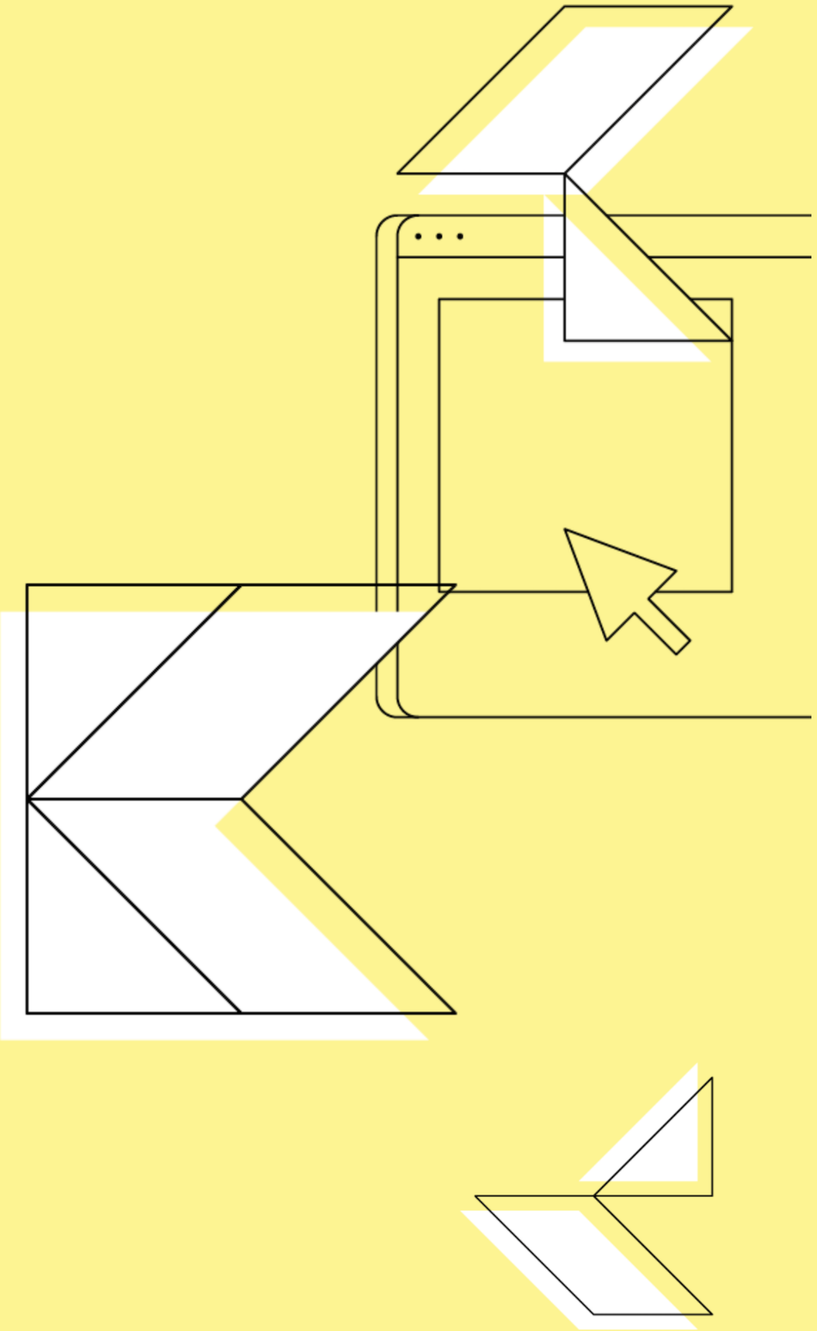
Convertendo Callbacks

```
launch{
    try {
        val latLng = getMyLocation()
        // do something
    } catch(e: Exception){
        // handle error
    }
}
```

Coroutines - Parte 2

- **`launch`** (*fire-and-forget*) e **`async`** (para obter um resultado).
- Trate as exceções no **`launch`** ou no **`async`**. Ou use **`SupervisorJob`**, **`SupervisorScope`** ou **`supervisorScope`**.
- **`cancel`** ou **`cancelChildren`** para cancelar o **`Job`** ou os jobs filhos.
- **`withTimeout`** ou **`withTimeoutOrNull`**.
- Toda *suspend function* é convertida em um *callback* usando a interface **`Continuation`**.

“Reactive Coroutines”



#KotlinEverywhere

Flow

- Flow é uma abstração de um *cold stream*.
- Nada é executado/emitido até que algum consumidor se registre no fluxo.
- Possui diversos operadores como no RxJava.

```kotlin
@FlowPreview
public interface Flow<out T> {
    public suspend fun collect(collector: FlowCollector<T>)
}

@FlowPreview
public interface FlowCollector<in T> {
    public suspend fun emit(value: T)
}
```

```
val intFlow = flow {
    for (i in 0 until 10) {
        emit(i) //calls emit directly from the body of a FlowCollector
    }
}
launch {
    intFlow.collect { number ->
        addTextToTextView("$number\n")
    }
    addTextToTextView("DONE!")
}
```

```
val intFlow = flow {
    for (i in 0 until 10) {
        emit(i) //calls emit directly from the body of a FlowCollector
    }
}
launch {
    intFlow.collect { number ->
        addTextToTextView("$number\n")
    }
    addTextToTextView("DONE!")
}
```

```
val intFlow = flow {
    for (i in 0 until 10) {
        emit(i) //calls emit directly from the body of a FlowCollector
    }
}
launch {
    intFlow.collect { number ->
        addTextToTextView("$number\n")
    }
    addTextToTextView("DONE!")
}
```

```
launch {
    (0..100).asFlow()
        .map { it * it }
        .filter { it % 4 == 0 } // here and above is on IO thread pool
        .flowOn(Dispatchers.IO) // 👆change the upstream Dispatcher
        .map { it * 2 }
        .flowOn(Dispatchers.Main)
        .onStart {  }
        .onEach {  }
        .onCompletion {  }
        .collect { number ->
            addTextToTextView("$number\n")
        }
}
```

```kotlin
class NumberFlow {
    private var currentValue = 0
    private val numberChannel = BroadcastChannel<Int>(10)

    fun getFlow(): Flow<Int> = numberChannel.asFlow()

    suspend fun sendNext() {
        numberChannel.send(currentValue++)
    }

    fun close() = numberChannel.close()
}
```

```kotlin
class NumberFlow {
    private var currentValue = 0
    private val numberChannel = BroadcastChannel<Int>(10)

    fun getFlow(): Flow<Int> = numberChannel.asFlow()

    suspend fun sendNext() {
        numberChannel.send(currentValue++)
    }

    fun close() = numberChannel.close()
}
```

```
class FlowActivity : AppCompatActivity(R.layout.activity_flow) {
    private val sender = NumberFlow()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        btnProduce.setOnClickListener {
            lifecycleScope.launch {
                sender.sendNext()
            }
        }
        lifecycleScope.launch {
            sender.getFlow().collect {
                txtOutput.append("Number: $it \n")
            }
        }
    }
    override fun onDestroy() {
        super.onDestroy()
        sender.close()
    }
}
```

```kotlin
class FlowActivity : AppCompatActivity(R.layout.activity_flow) {
    private val sender = NumberFlow()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        btnProduce.setOnClickListener {
            lifecycleScope.launch {
                sender.sendNext()
            }
        }
        lifecycleScope.launch {
            sender.getFlow().collect {
                txtOutput.append("Number: $it \n")
            }
        }
    }
    override fun onDestroy() {
        super.onDestroy()
        sender.close()
    }
}
```

```
class FlowActivity : AppCompatActivity(R.layout.activity_flow) {
    private val sender = NumberFlow()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        btnProduce.setOnClickListener {
            lifecycleScope.launch {
                sender.sendNext()
            }
        }
        lifecycleScope.launch {
            sender.getFlow().collect {
                txtOutput.append("Number: $it \n")
            }
        }
    }
    override fun onDestroy() {
        super.onDestroy()
        sender.close()
    }
}
```

Callback para Flow

```
class YourApi {
    fun doSomeCall(callback: YourListener<String>) {
        // when new value arrives
        callback.onNext("Item 1")
        // when some error happens
        callback.onApiError(Exception("Error"))
        // when we're done
        callback.onComplete()
    }

    interface YourListener<T> {
        fun onNext(value: T)
        fun onApiError(t: Throwable)
        fun onComplete()
    }
}
```

```
fun myFlow(): Flow<String> = callbackFlow {
    val myCallback = object: YourApi.YourListener<String> {
        override fun onNext(value: String) { offer(value) }

        override fun onApiError(t: Throwable) { close(t) }

        override fun onComplete() { close() }
    }
    val api = YourApi()
    api.doSomeCall(myCallback)
    awaitClose { /* do something when the stream is closed */ }
}
```

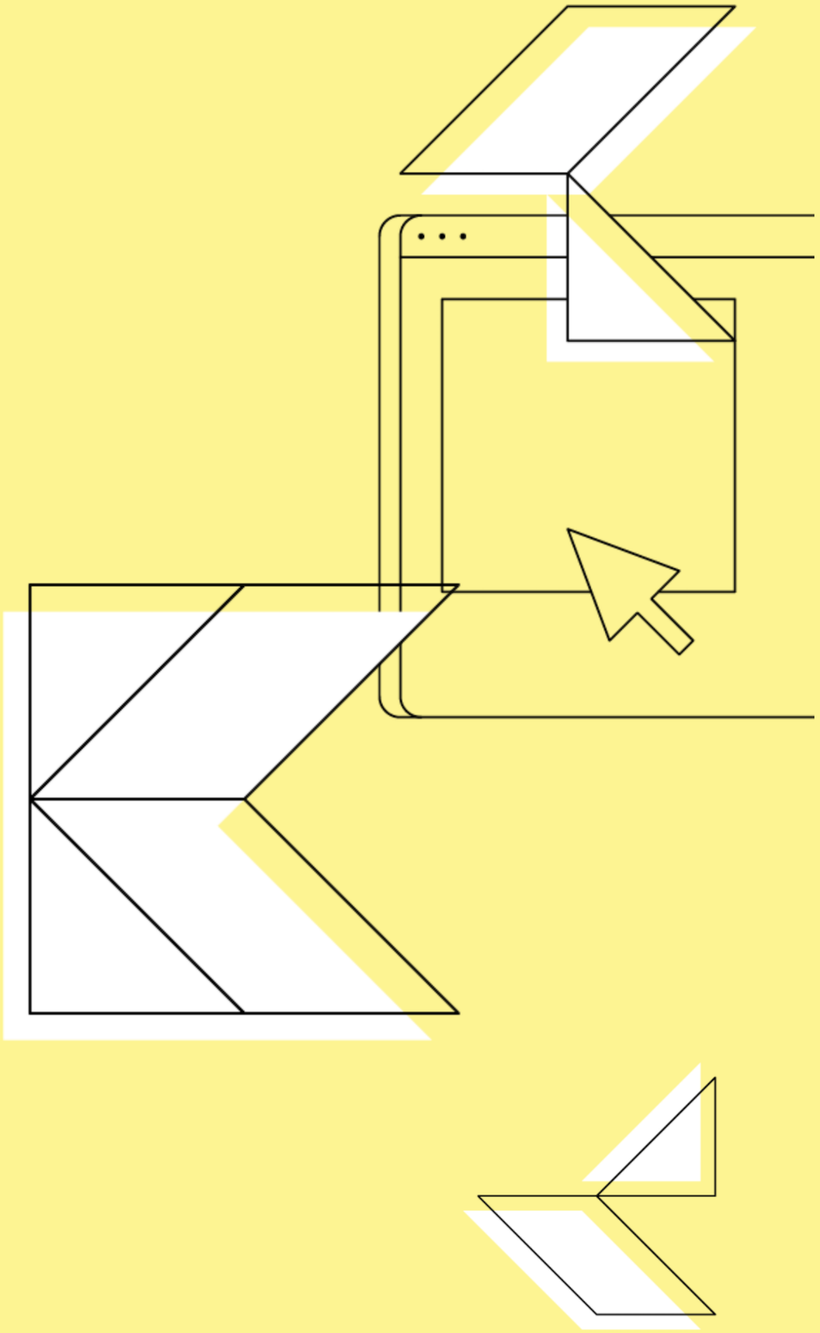
```
fun myFlow(): Flow<String> = callbackFlow {
    val myCallback = object: YourApi.YourListener<String> {
        override fun onNext(value: String) { offer(value) }

        override fun onApiError(t: Throwable) { close(t) }

        override fun onComplete() { close() }
    }
    val api = YourApi()
    api.doSomeCall(myCallback)
    awaitClose { /* do something when the stream is closed */ }
}
```

```
fun myFlow(): Flow<String> = callbackFlow {
    val myCallback = object: YourApi.YourListener<String> {
        override fun onNext(value: String) { offer(value) }

        override fun onApiError(t: Throwable) { close(t) }

        override fun onComplete() { close() }
    }
    val api = YourApi()
    api.doSomeCall(myCallback)
    awaitClose { /* do something when the stream is closed */ }
}
```

```
fun myFlow(): Flow<String> = callbackFlow {
    val myCallback = object: YourApi.YourListener<String> {
        override fun onNext(value: String) { offer(value) }

        override fun onApiError(t: Throwable) { close(t) }

        override fun onComplete() { close() }
    }
    val api = YourApi()
    api.doSomeCall(myCallback)
    awaitClose { /* do something when the stream is closed */ }
}
```

```
fun myFlow(): Flow<String> = callbackFlow {
    val myCallback = object: YourApi.YourListener<String> {
        override fun onNext(value: String) { offer(value) }

        override fun onApiError(t: Throwable) { close(t) }

        override fun onComplete() { close() }
    }
    val api = YourApi()
    api.doSomeCall(myCallback)
    awaitClose { /* do something when the stream is closed */ }
}
```

Room



#KotlinEverywhere

Room

```
dependencies {
    def room_version = "2.2.5"

    implementation "androidx.room:room-runtime:$room_version"
    kapt "androidx.room:room-compiler:$room_version"

    // 👇 Kotlin Extensions and Coroutines support for Room
    implementation "androidx.room:room-ktx:$room_version"

    ...
}
```

```
@Dao
interface BookDao {
    @Query("SELECT * FROM book")
    suspend fun getAll(): List<Book>

    @Query("SELECT * FROM book WHERE id = :id")
    suspend fun getBook(id: Long): Book

    @Insert
    suspend fun insert(book: Book): Long

    @Delete
    suspend fun delete(book: Book)
}
```

```
@Dao
interface BookDao {
    @Query("SELECT * FROM book")
    fun getAll(): Flow<List<Book>>

    @Query("SELECT * FROM book WHERE id = :id")
    fun getBook(id: Long): Flow<Book?>

    @Insert
    suspend fun insert(book: Book): Long

    @Delete
    suspend fun delete(book: Book)
}
```

```
launch {
    withContext(Dispatchers.IO) {
        val id = dao.insert(
            Book(0, "Dominando o Android", "Nelson Glauber")
        )
    }
    // Do UI stuff
}
```

```
launch {
    dao.getAll().collect { bookList ->
        lstBooks.adapter = BookAdapter(context, bookList)
    }
}
```

```
class BookFavoritesViewModel(
    repository: BookRepository
): ViewModel() {

    val favoriteBooks = repository.allFavorites().asLiveData()
}
```

Flow

```kotlin
class BookFavoritesViewModel(
    repository: BookRepository
): ViewModel() {

    val favoriteBooks = liveData {
        repository.allFavorites().collect { list ->
            emit(list)
        }
    }
}
```

Conclusão

- Coroutines vêm se tornando a forma de padrão para realizar código assíncrono no Android.
- Essa é uma recomendação do Google.
- Além do Jetpack, outras bibliotecas estão migrando (ou já migraram) pra Coroutines (ex: Retrofit, Apollo, MockK, …).

Referências #1

- Android Suspenders (Android Dev Summit 2018)
  https://www.youtube.com/watch?v=EOjq4OIWKqM

- Understand Kotlin Coroutines on Android (Google I/O 2019)
  https://www.youtube.com/watch?v=BOHK_w09pVA

- Coroutines Guide
  https://github.com/Kotlin/kotlinx.coroutines/blob/master/coroutines-guide.md

- Android Suspenders by Chris Banes (KotlinConf 2018)
  https://www.youtube.com/watch?v=P7ov_r1JZ1g

- Room & Coroutines (Florina Muntenescu)
  https://medium.com/androiddevelopers/room-coroutines-422b786dc4c5

Referências #2

- Using Kotlin Coroutines in your Android App
  https://codelabs.developers.google.com/codelabs/kotlin-coroutines

- Use Kotlin coroutines with Architecture Components
  https://developer.android.com/topic/libraries/architecture/coroutines

- Create a Clean-Code App with Kotlin Coroutines and Android Architecture Components
  https://blog.elpassion.com/create-a-clean-code-app-with-kotlin-coroutines-and-android-architecture-components-f533b04b5431

- Android Coroutine Recipes (Dmytro Danylyk)
  https://proandroiddev.com/android-coroutine-recipes-33467a4302e9

- Kotlin Coroutines patterns & anti-patterns
  https://proandroiddev.com/kotlin-coroutines-patterns-anti-patterns-f9d12984c68e

Referências #3

- The reason to avoid GlobalScope (Roman Elizarov)
  https://medium.com/@elizarov/the-reason-to-avoid-globalscope-835337445abc

- WorkManager meets Kotlin (Pietro Maggi)
  https://medium.com/androiddevelopers/workmanager-meets-kotlin-b9ad02f7405e

- Coroutine Context and Scope (Roman Elizarov)
  https://medium.com/@elizarov/coroutine-context-and-scope-c8b255d59055

- Easy Coroutines in Android: viewModelScope (Manuel Vivo)
  https://medium.com/androiddevelopers/easy-coroutines-in-android-viewmodelscope-25bffb605471

- Exceed the Android Speed Limit
  https://medium.com/androiddevelopers/exceed-the-android-speed-limit-b73a0692abc1

Referências #4

- An Early look at Kotlin Coroutine's Flow
  https://proandroiddev.com/an-early-look-at-kotlin-coroutines-flow-62e46baa6eb0

- Coroutines on Android (Sean McQuillan)
  https://medium.com/androiddevelopers/coroutines-on-android-part-i-getting-the-background-3e0e54d20bb

- Kotlin Flows and Coroutines (Roman Elizarov)
  https://medium.com/@elizarov/kotlin-flows-and-coroutines-256260fb3bdb

- Simple design of Kotlin Flow (Roman Elizarov)
  https://medium.com/@elizarov/simple-design-of-kotlin-flow-4725e7398c4c

- React Streams and Kotlin Flows (Roman Elizarov)
  https://medium.com/@elizarov/reactive-streams-and-kotlin-flows-bfd12772cda4

Referências #5

- KotlinConf 2019: Coroutines! Gotta catch 'em all! by Florina Muntenescu & Manuel Vivo
  https://www.youtube.com/watch?v=w0kfnydnFWI

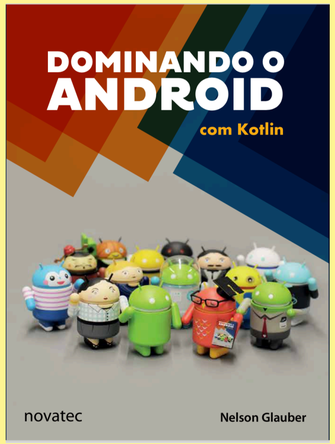
- LiveData with Coroutines and Flow (Android Dev Summit '19)
  https://www.youtube.com/watch?v=B8ppnjGPAGE

- Testing Coroutines on Android (Android Dev Summit '19)
  https://www.youtube.com/watch?v=KMb0Fs8rCRs

Obrigado!

Mokriya

Google Developers Experts



Nelson Glauber
@nglauber