TAPSI

Kotlin Coroutines

Hossein Gheisary

Kotlin Coroutines

- **Asynchronous Programming**
- **Introduction to Kotlin Coroutines**
- **Structural concurrency**
- **Best Practices In Android**

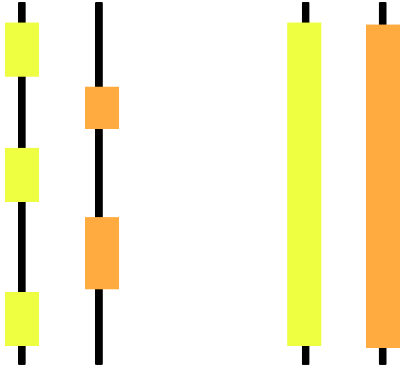
Async Programming

- more advanced features
  - networking
  - database

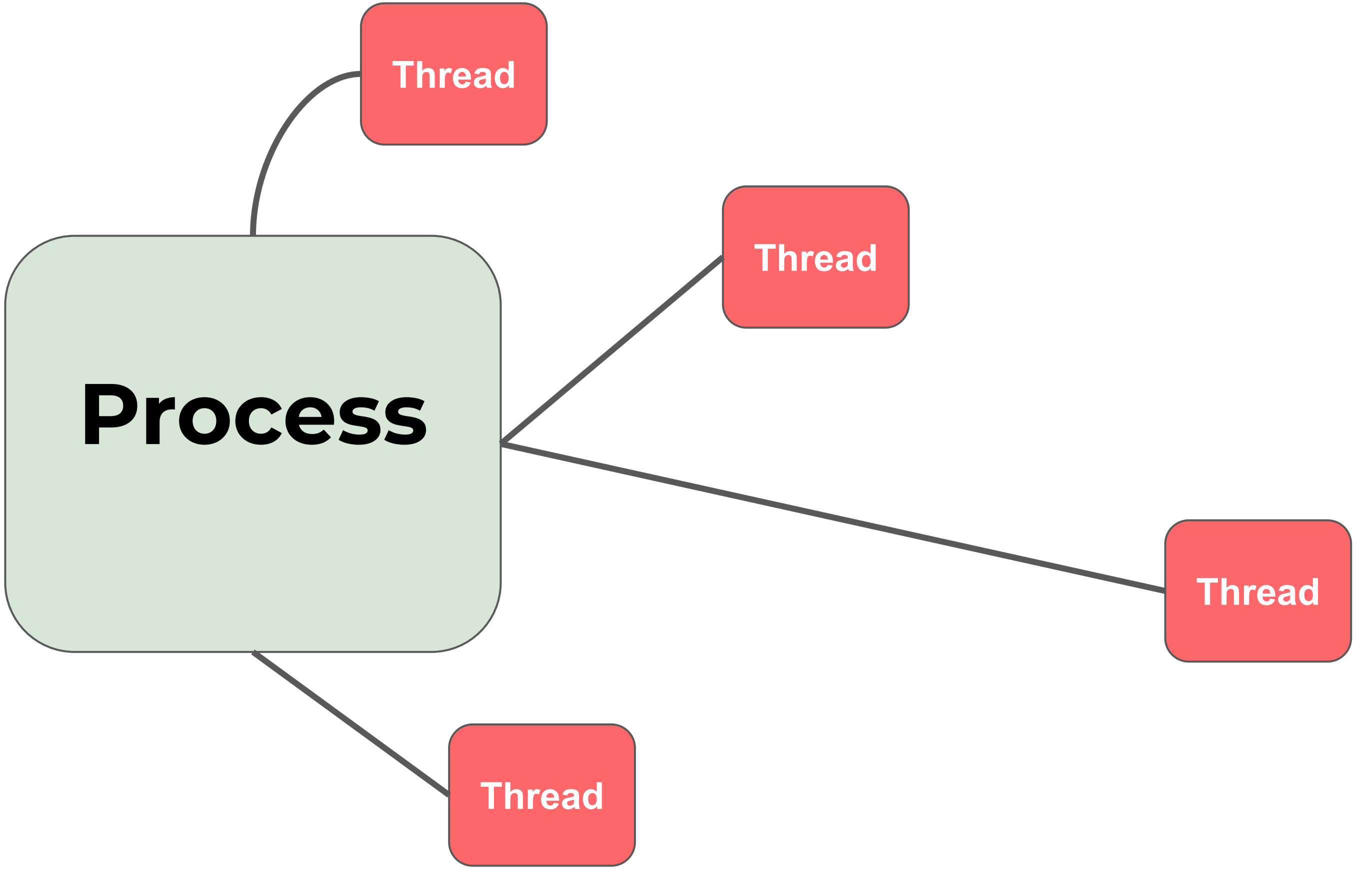
- parallel operation & concurrent operation

- Threads, AsyncTasks, RxJava, Coroutine , ...

- Efficiency
  - memory overhead
  - leaks
  - Switching time



Threads



Threads

- **A flow of execution**
- **whenever you run a java program.** → **The main thread will create**
- **Thread switching is heavy & has memory overhead**

Threads

What is Thread ?

Threads

1- Thread is a class

2- Object from Thread class (Heap & Stack memory allocation)

3- Call run(start) method

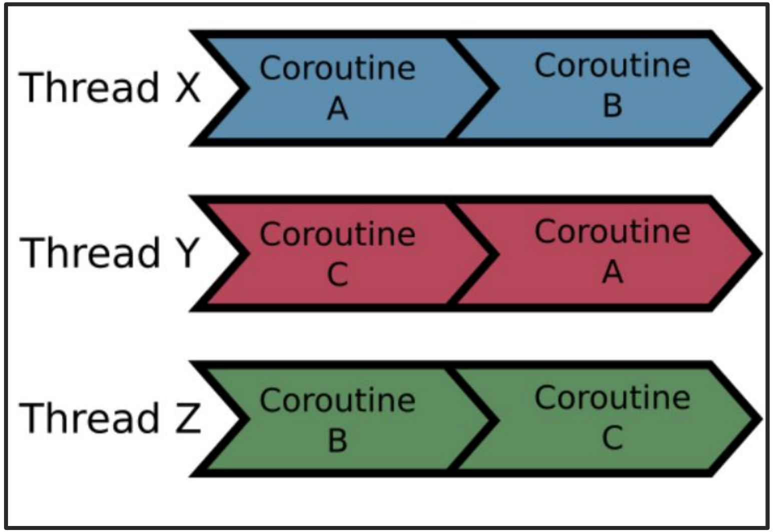
4- Jvm communicate with OS scheduler to get cpu turn

Introduction To Kotlin Coroutines

Coroutines

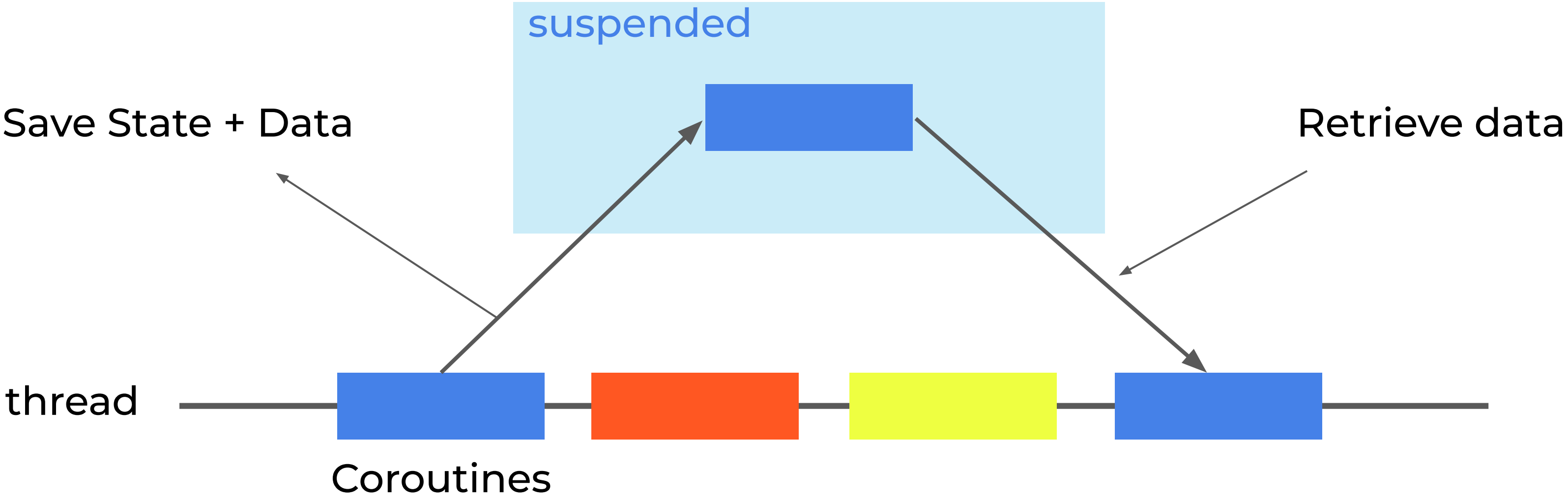
Kotlin Coroutines

- **Multiple coroutine can run on one thread**
- **A coroutine is not bound to any particular thread.**
- **may suspend execution in one thread and resume in another.**



Kotlin Coroutines

Computation can be suspended without blocking thread



Kotlin Coroutines

- **Coroutines have been stable since Kotlin 1.3 (October 2018)**
- **Unlike threads, don’t need a lot of memory, just some bytes.**
- **Suspend and resume concept**

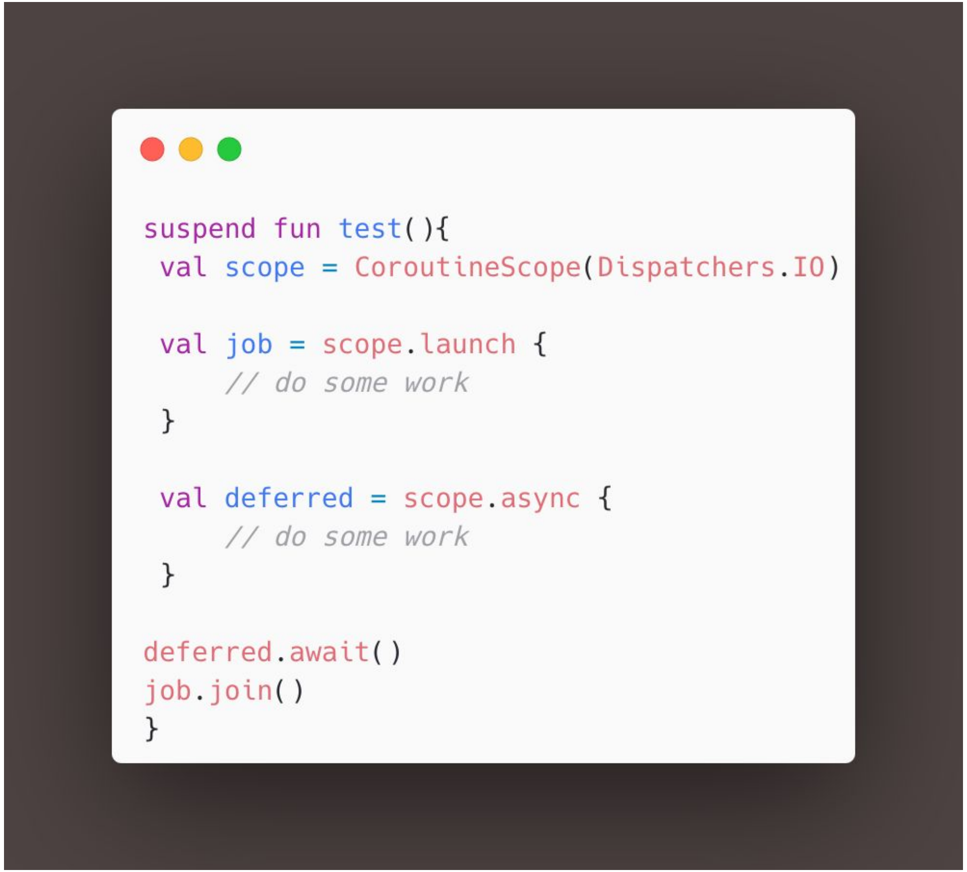
Kotlin Coroutines

Coroutines Builders

Coroutines Builders

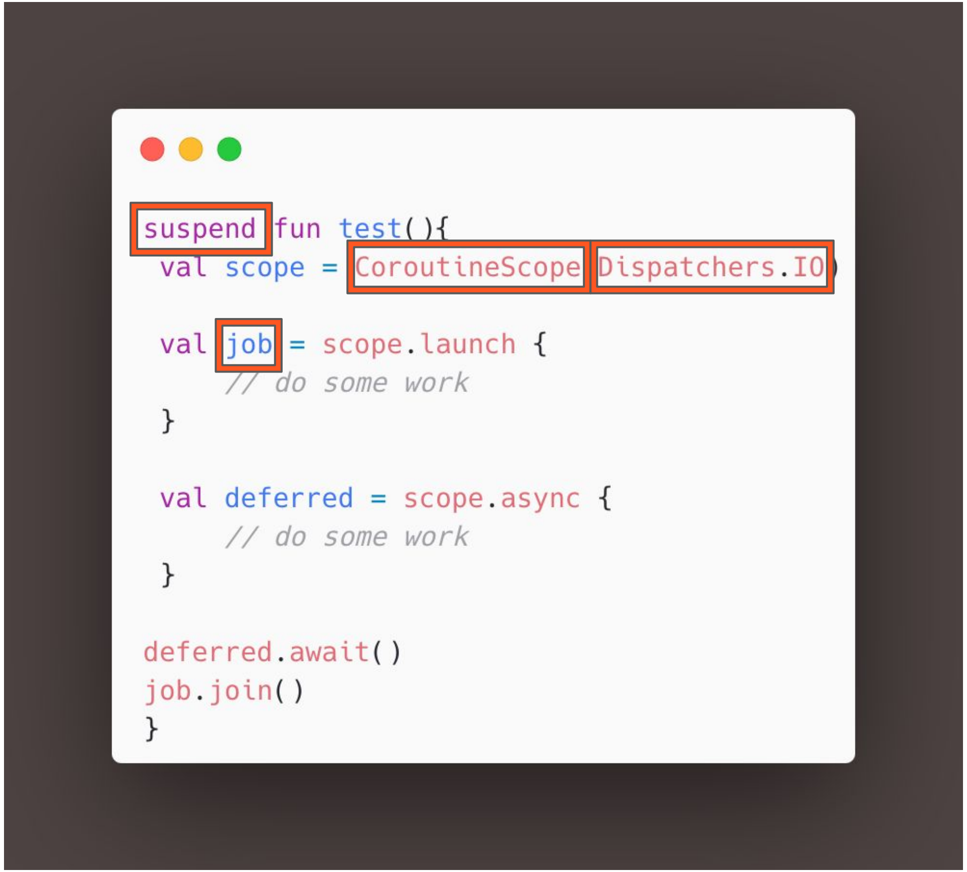
Launch

Async



```
suspend fun test(){
 val scope = CoroutineScope(Dispatchers.IO)

 val job = scope.launch {
     // do some work
 }

 val deferred = scope.async {
     // do some work
 }

deferred.await()
job.join()
}
```

Coroutines Builders



```
suspend fun test(){
  val scope = CoroutineScope(Dispatchers.IO)

  val job = scope.launch {
      // do some work
  }

  val deferred = scope.async {
      // do some work
  }

deferred.await()
job.join()
}
```

Suspend Function

Coroutine Scope

Coroutine Context

Coroutine Job

Kotlin Coroutines

Suspend

Suspend

- **how the code can suspend without blocking threads ?**
- **why a suspend function won’t return until all the work that it started has completed ?**
- **what the compiler does under the hood ?**

Suspend

- Regular function

Suspend & Resume

Suspension Points

- When a coroutine suspend

The current stack frame copy and save

The return to its pool

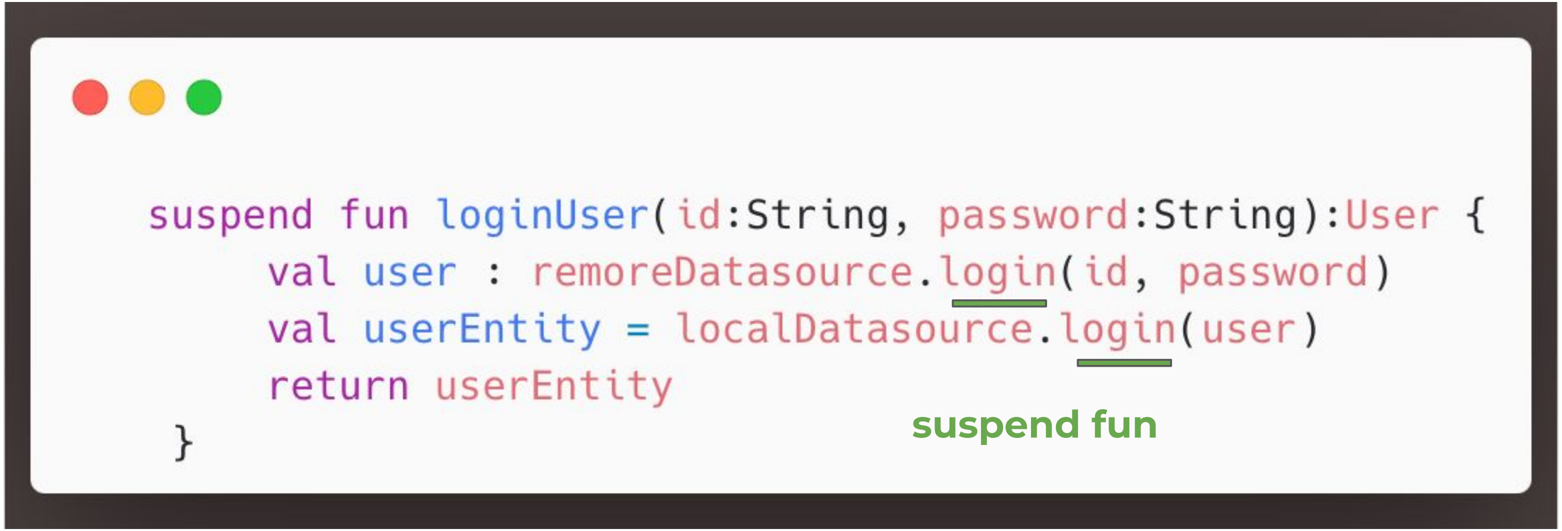
- When the suspension is over, the coroutine resumes on a free thread in the pool.

- *Kotlin compiler will create a state machine for every suspend function*

Suspend

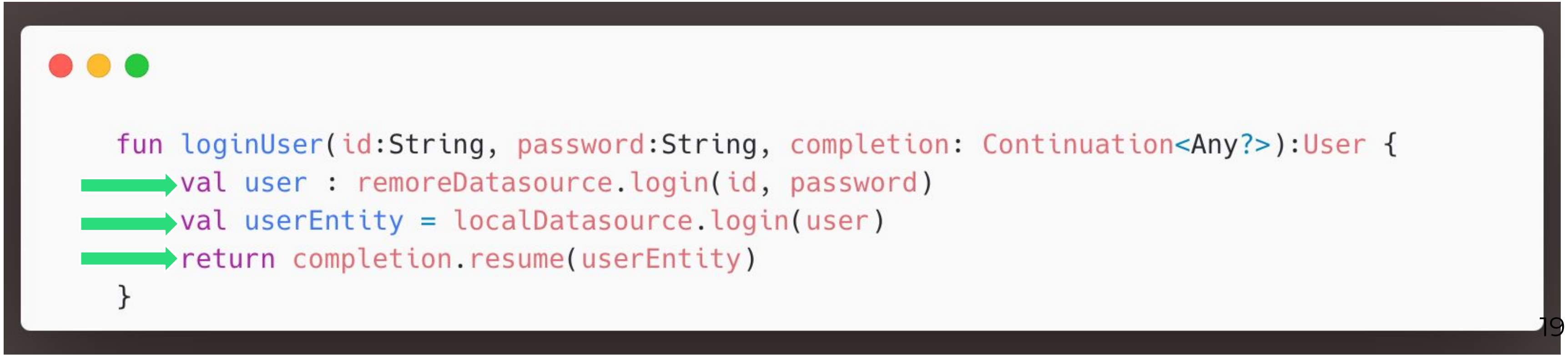
Under The Hood

Suspend



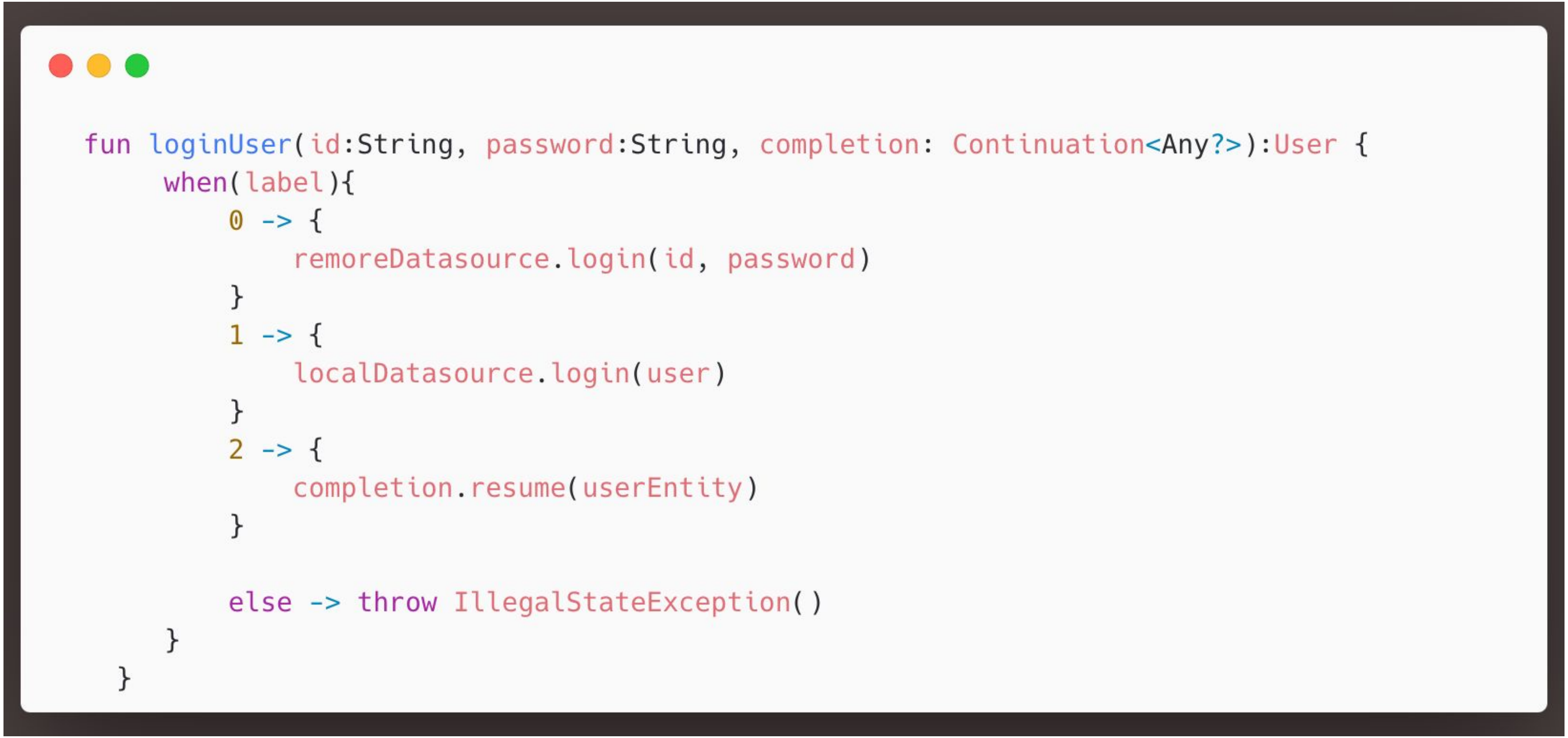
```
suspend fun loginUser(id:String, password:String):User {
    val user : remoreDatasource.login(id, password)
    val userEntity = localDatasource.login(user)
    return userEntity
 }
```

suspend fun



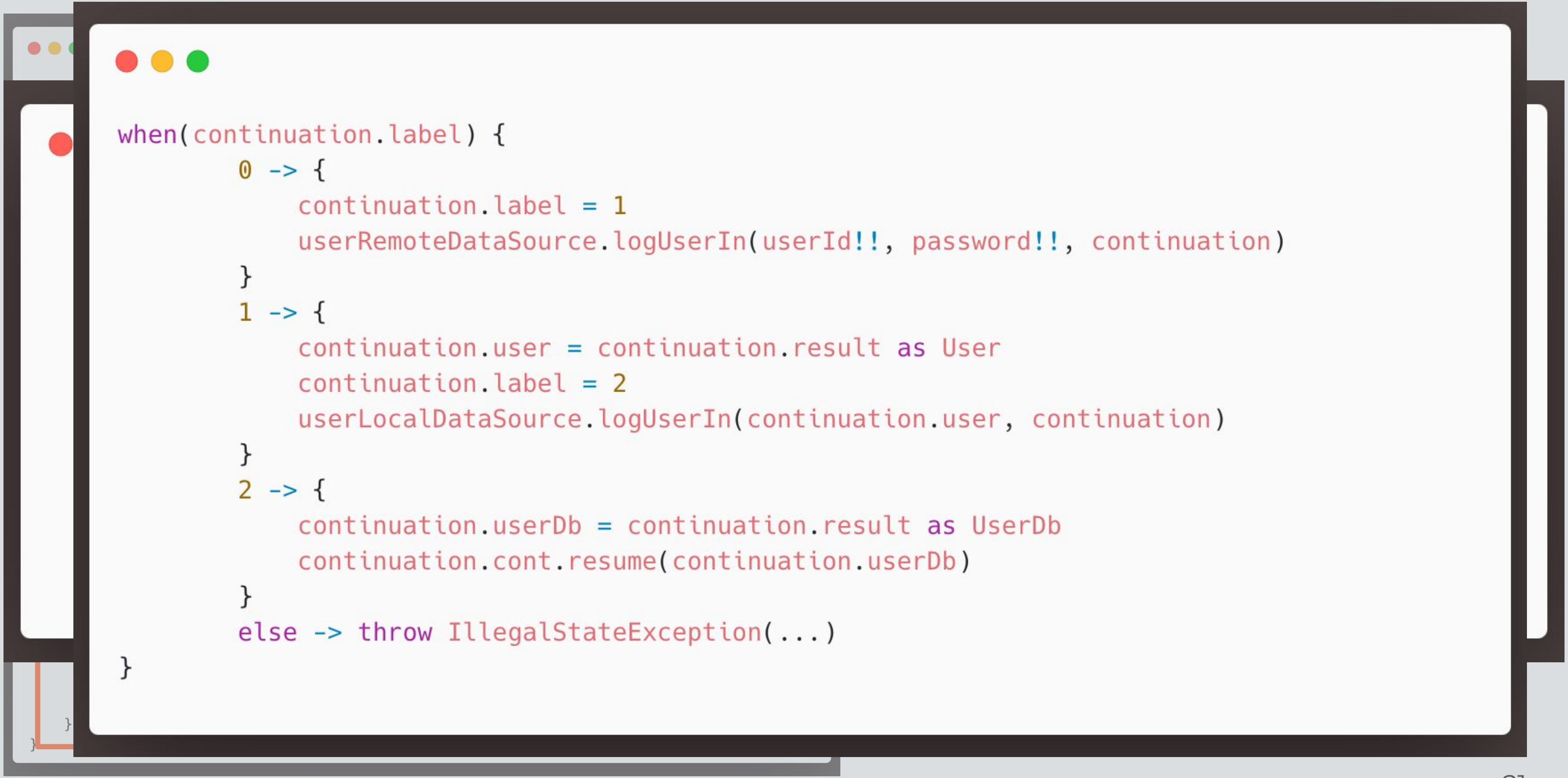
```
fun loginUser(id:String, password:String, completion: Continuation<Any?>):User {
   val user : remoreDatasource.login(id, password)
   val userEntity = localDatasource.login(user)
   return completion.resume(userEntity)
}
```

Using Continuation



```
fun loginUser(id:String, password:String, completion: Continuation<Any?>):User {
     when(label){
         0 -> {
             remoreDatasource.login(id, password)
         }
         1 -> {
             localDatasource.login(user)
         }
         2 -> {
             completion.resume(userEntity)
         }

         else -> throw IllegalStateException()
     }
  }
```

Using Continuation



```
when(continuation.label) {
        0 -> {
            continuation.label = 1
            userRemoteDataSource.logUserIn(userId!!, password!!, continuation)
        }
        1 -> {
            continuation.user = continuation.result as User
            continuation.label = 2
            userLocalDataSource.logUserIn(continuation.user, continuation)
        }
        2 -> {
            continuation.userDb = continuation.result as UserDb
            continuation.cont.resume(continuation.userDb)
        }
        else -> throw IllegalStateException(...)
}
```

Suspend

- **how the code can suspend without blocking threads**

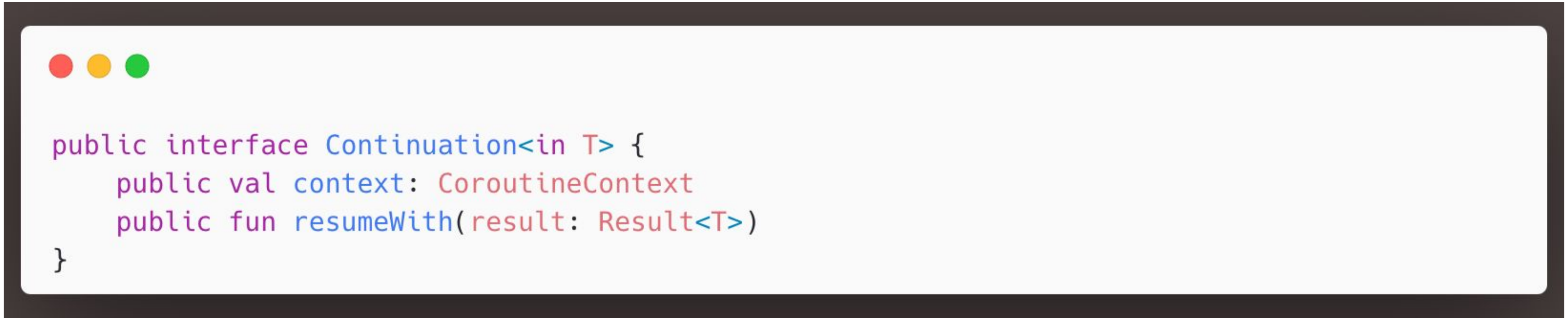
  it knows from where to continue after execution

- **why a suspend function won’t return until all the work that it started has completed**

  Continuation object (switch-case)

- **what the compiler does under the hood**

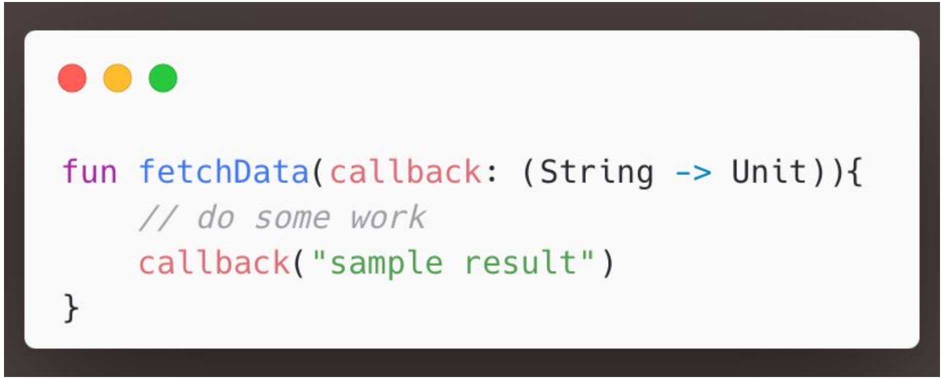
Using Continuation



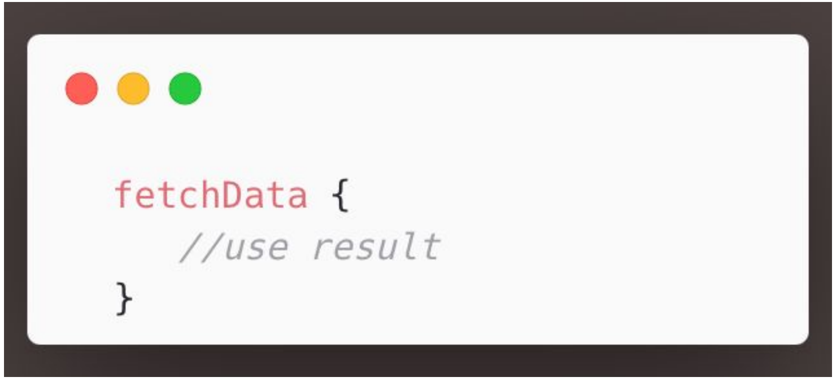
```
public interface Continuation<in T> {
    public val context: CoroutineContext
    public fun resumeWith(result: Result<T>)
}
```

- Continuation is a public interface
- Can convert the callback-based API into a suspendable function

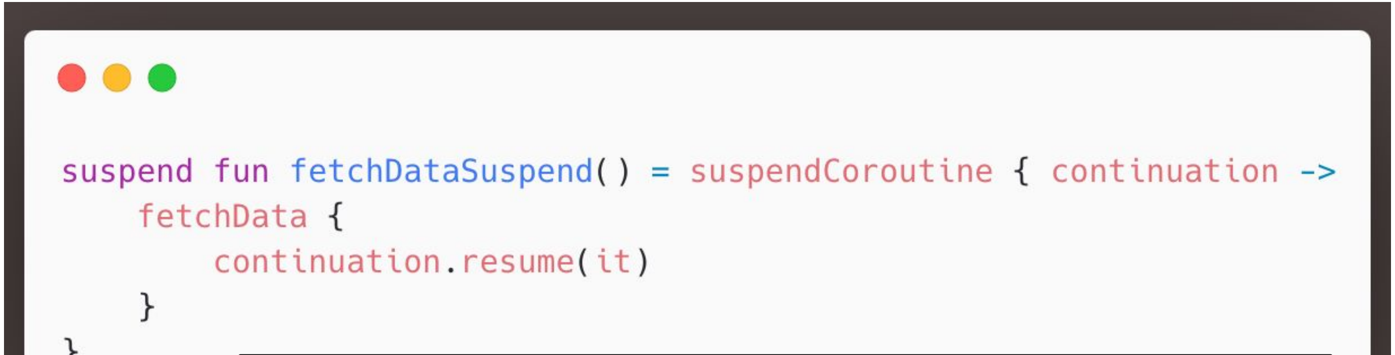
Using Continuation



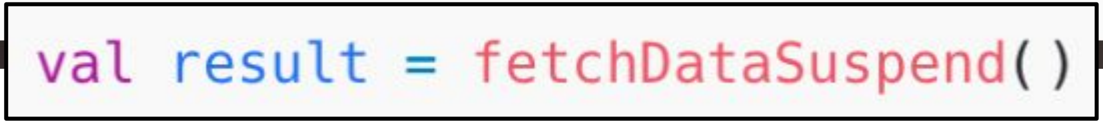
```
fun fetchData(callback: (String -> Unit)){
    // do some work
    callback("sample result")
}
```



```
fetchData {
    //use result
}
```

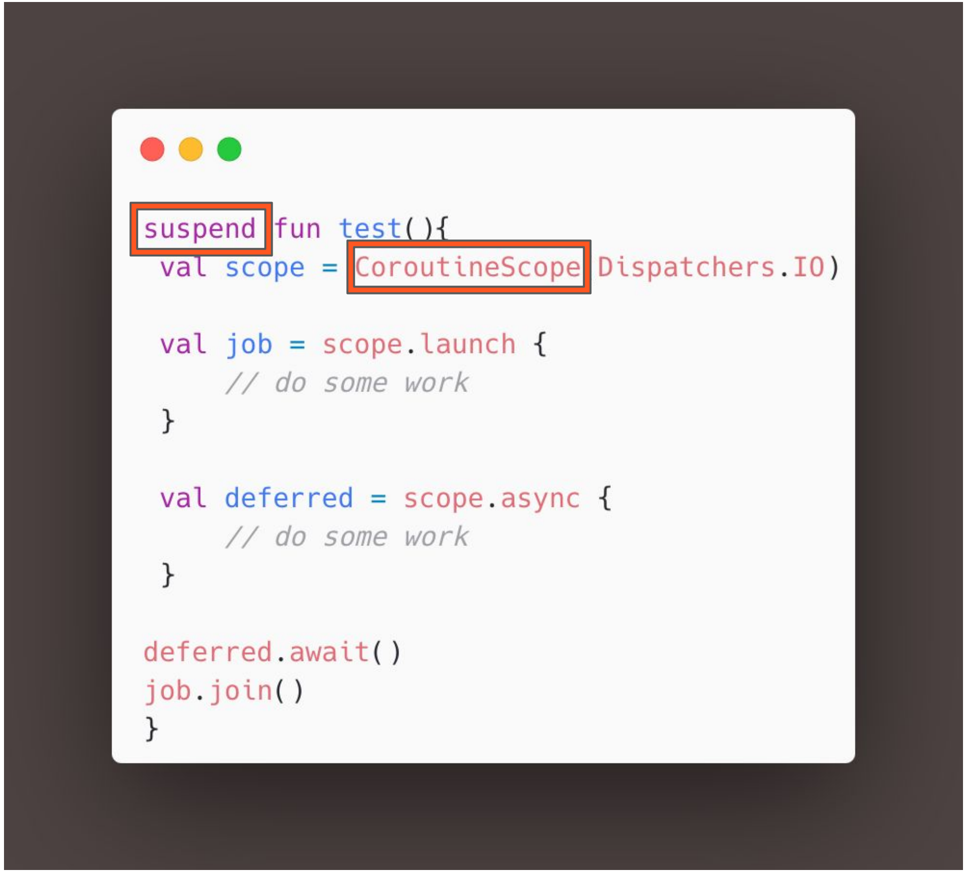


```
suspend fun fetchDataSuspend() = suspendCoroutine { continuation ->
    fetchData {
        continuation.resume(it)
    }
}
```



```
val result = fetchDataSuspend()
```

Kotlin Coroutines



```
suspend fun test(){
 val scope = CoroutineScope Dispatchers.IO)

 val job = scope.launch {
     // do some work
 }

 val deferred = scope.async {
     // do some work
 }

deferred.await()
job.join()
}
```

Suspend Function

Coroutine Scope

Kotlin Coroutines

Coroutine Scope

Coroutine Scope

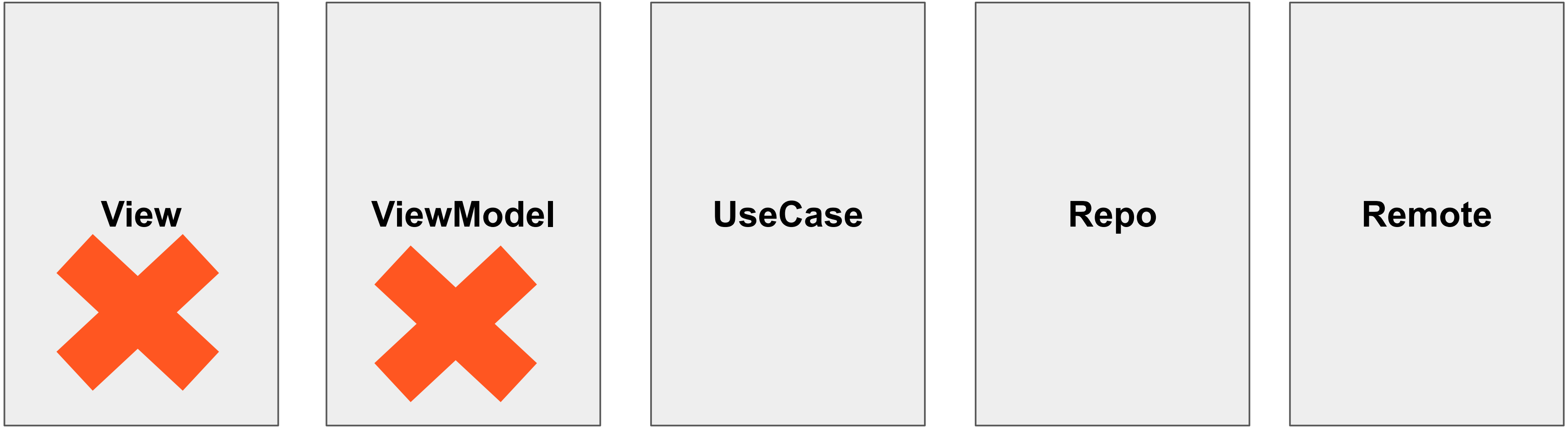
- **Start and control the lifecycle of coroutines in a particular layer of your app.**

- **Takes a `CoroutineContext` as a parameter**

- **The coroutine context is a set of rules and configurations that define how the coroutine will be executed. (ex: which thread)**

- **Examples:** `viewModelScope` and `lifecycleScope`

Coroutine Scope



```
viewModel.getData()
```

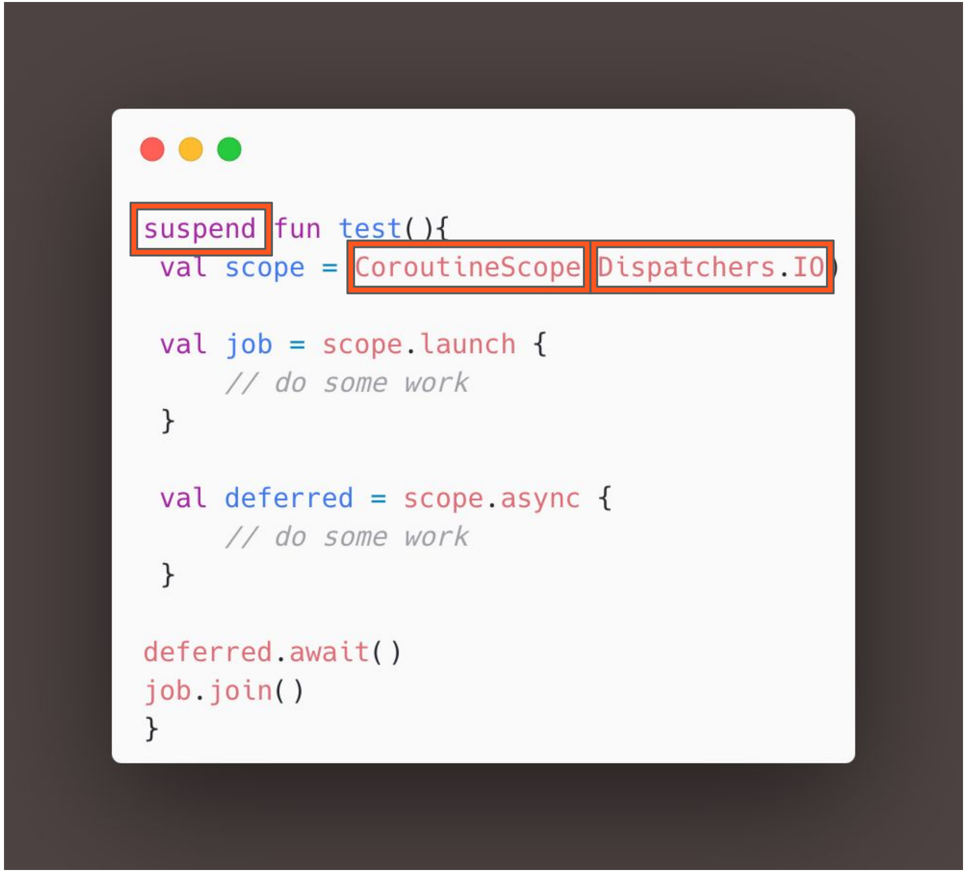
```
fun getData() {
 scope.launch{
    usecase.getData()
  }
}
```

```
suspend fun getData(){
   repo.getData()
}
```

```
suspend fun getData(){
   remote.getData()
}
```

```
@GET
suspend fun getData()
```

Kotlin Coroutines



```
suspend fun test(){
  val scope = CoroutineScope(Dispatchers.IO)

  val job = scope.launch {
      // do some work
  }

  val deferred = scope.async {
      // do some work
  }

deferred.await()
job.join()
}
```

Suspend Function

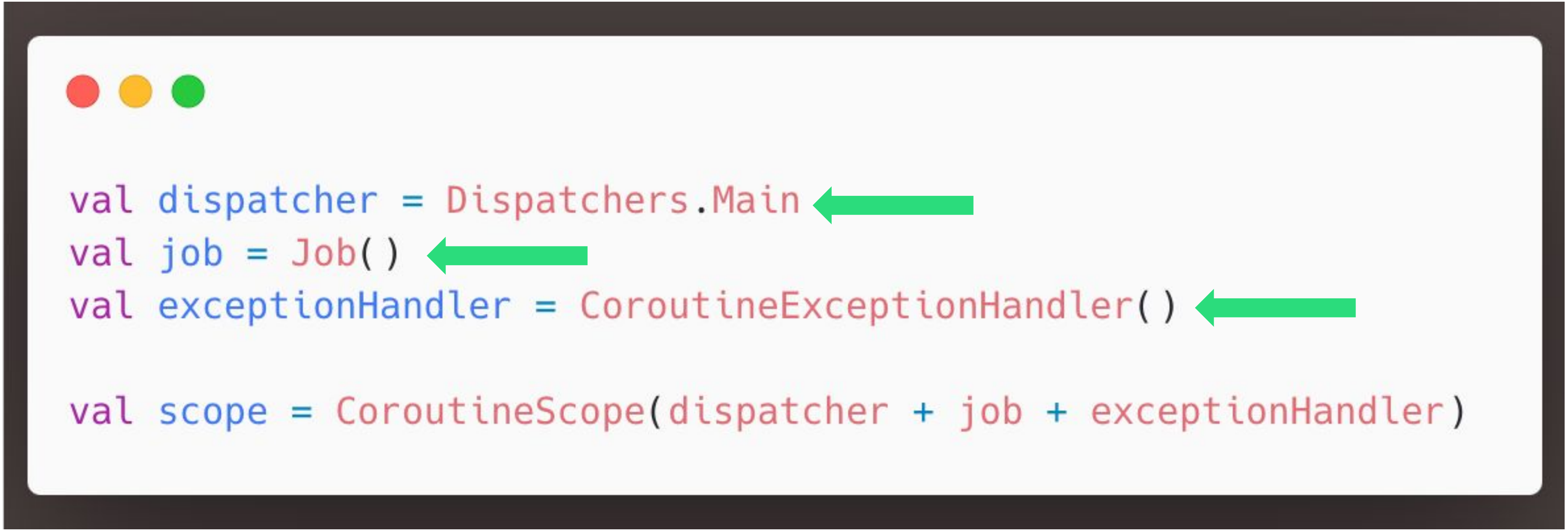
Coroutine Scope

Coroutine Context

Kotlin Coroutines

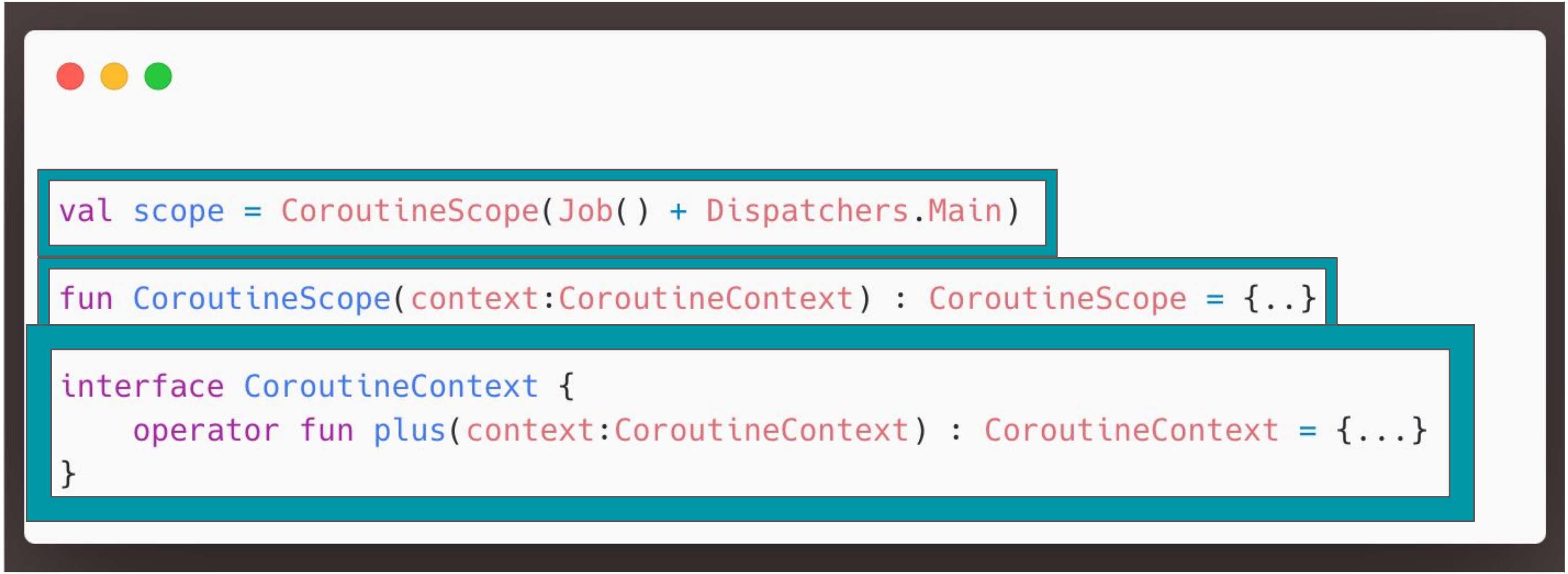
Coroutine Context

Coroutine Context



```
val dispatcher = Dispatchers.Main
val job = Job()
val exceptionHandler = CoroutineExceptionHandler()

val scope = CoroutineScope(dispatcher + job + exceptionHandler)
```

Coroutine Scope

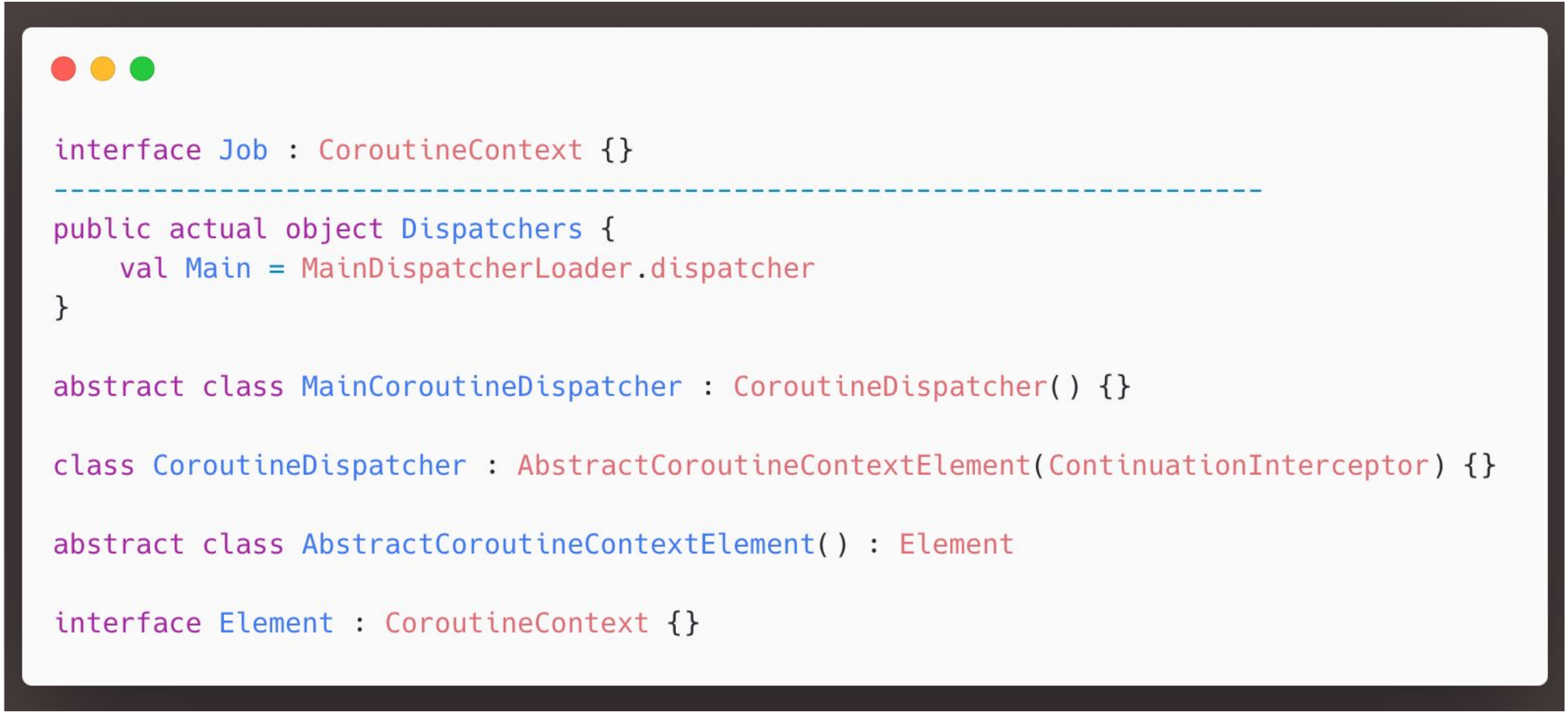


```
val scope = CoroutineScope(Job() + Dispatchers.Main)
```

```
fun CoroutineScope(context:CoroutineContext) : CoroutineScope = {..}
```

```
interface CoroutineContext {
    operator fun plus(context:CoroutineContext) : CoroutineContext = {...}
}
```

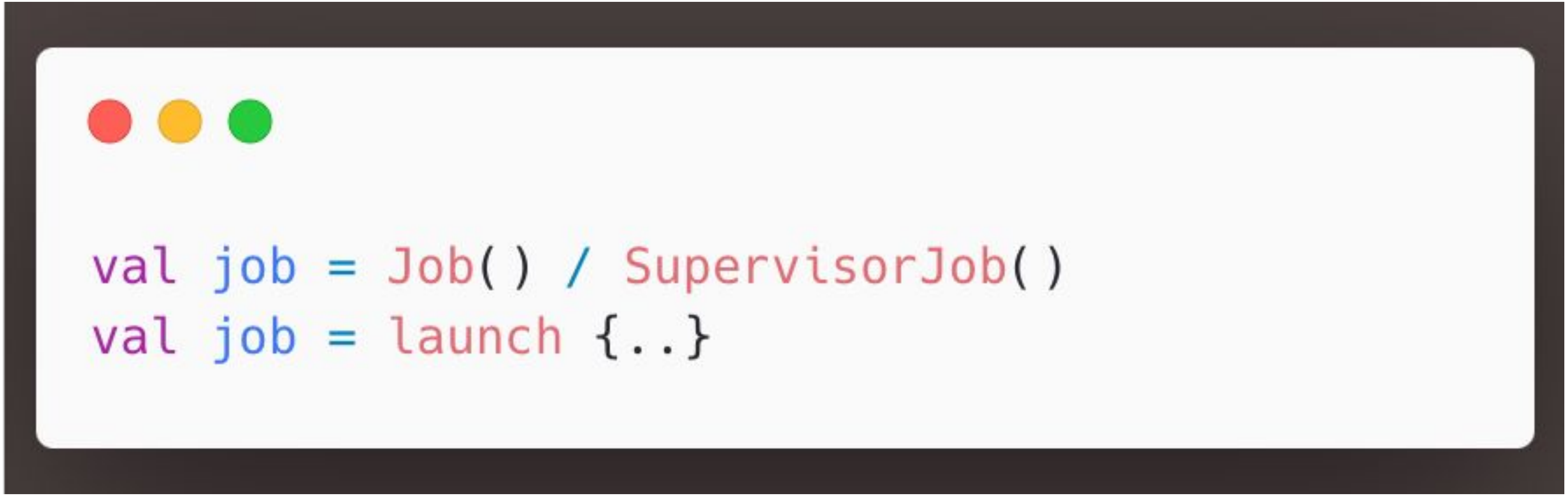
Coroutine Scope



```
interface Job : CoroutineContext {}
-------------------------------------------------------------------------
public actual object Dispatchers {
    val Main = MainDispatcherLoader.dispatcher
}

abstract class MainCoroutineDispatcher : CoroutineDispatcher() {}

class CoroutineDispatcher : AbstractCoroutineContextElement(ContinuationInterceptor) {}

abstract class AbstractCoroutineContextElement() : Element

interface Element : CoroutineContext {}
```

Job

Job

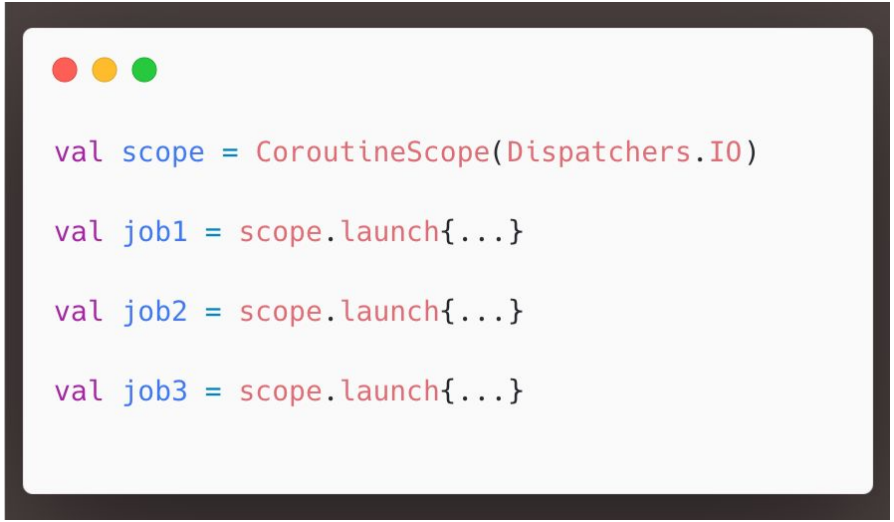


```
val job = Job() / SupervisorJob()
val job = launch {..}
```

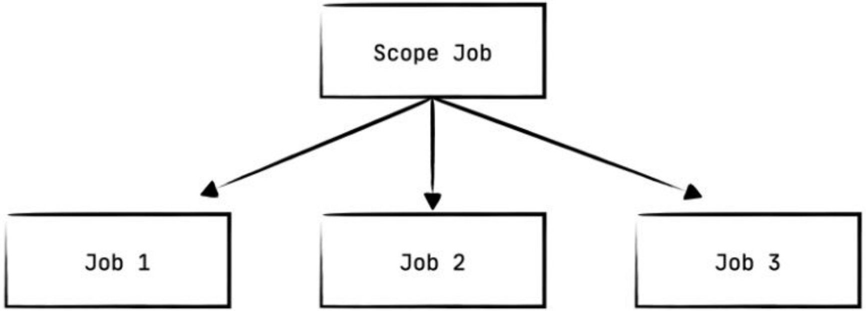
- **lifecycle, cancellation, and parent-child relations**

Job

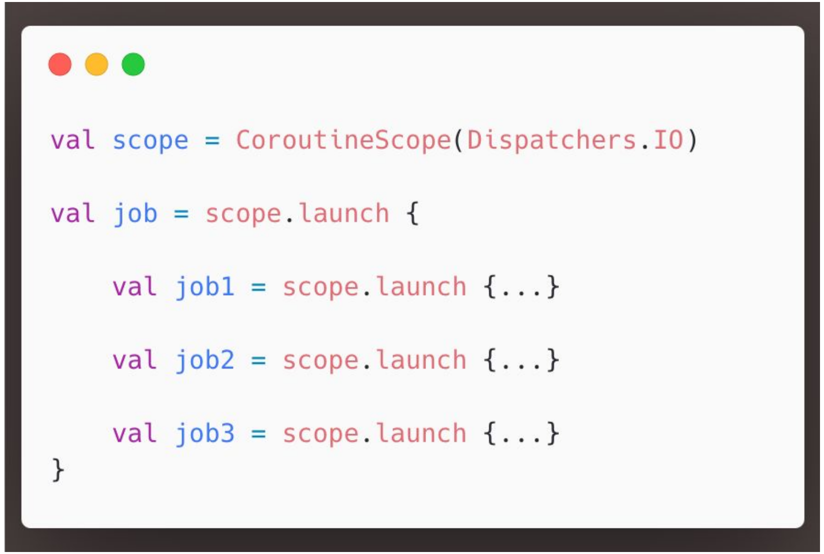
Parent-Child Relationship



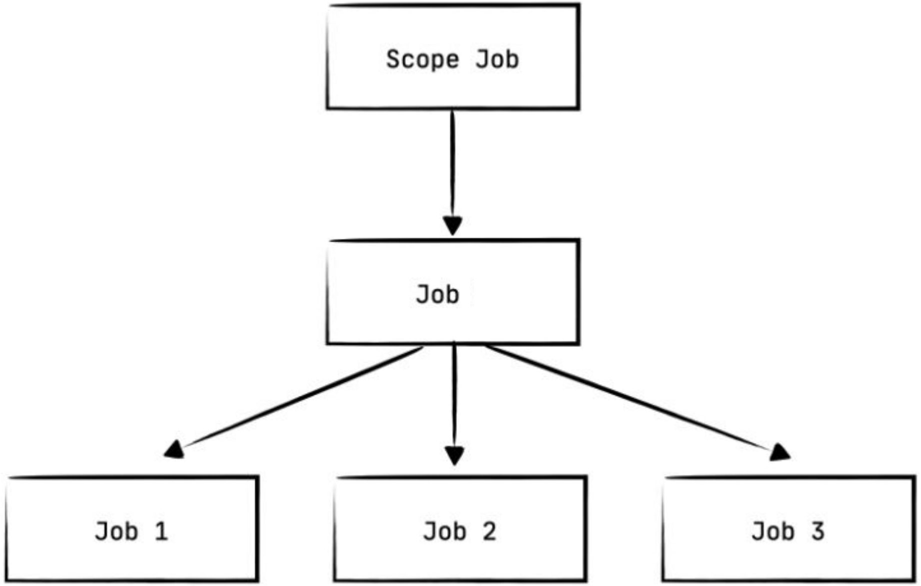
```kotlin
val scope = CoroutineScope(Dispatchers.IO)

val job1 = scope.launch{...}

val job2 = scope.launch{...}

val job3 = scope.launch{...}
```



Parent-Child Relationship



```
val scope = CoroutineScope(Dispatchers.IO)

val job = scope.launch {

    val job1 = scope.launch {...}

    val job2 = scope.launch {...}

    val job3 = scope.launch {...}
}
```



Scope Job

Job

Job 1

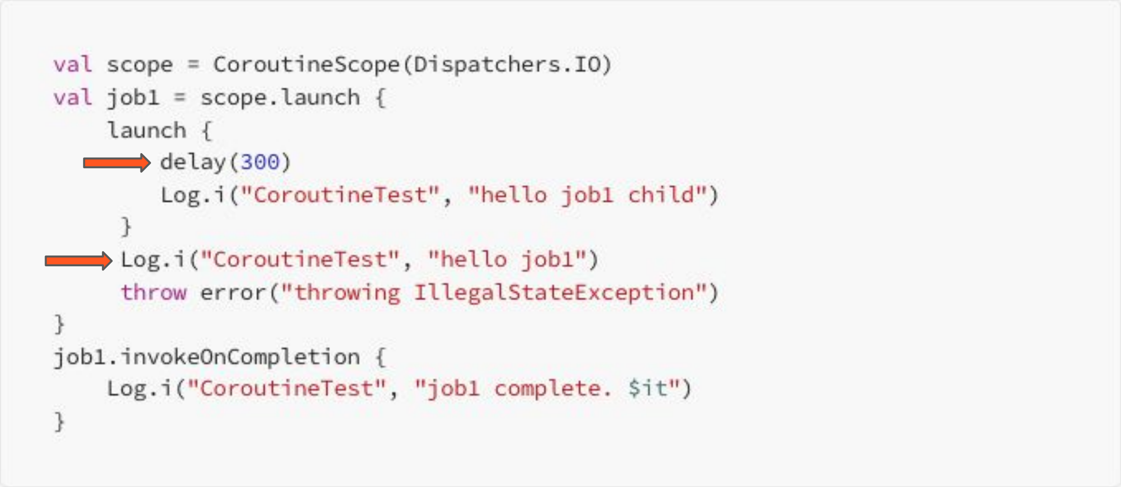
Job 2

Job 3

Job

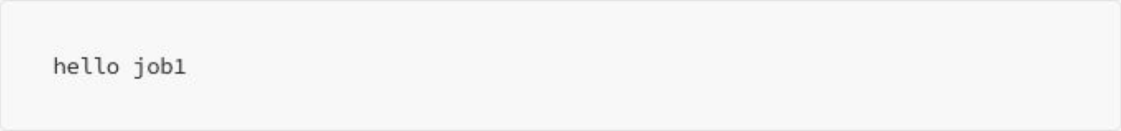
JOB	Parent Job cancel	All children cancel
	One child fail	Parent Job cancel
SUPERVISOR JOB	Parent Job cancel	All children cancel
	One child fail	Nothing happen

Job



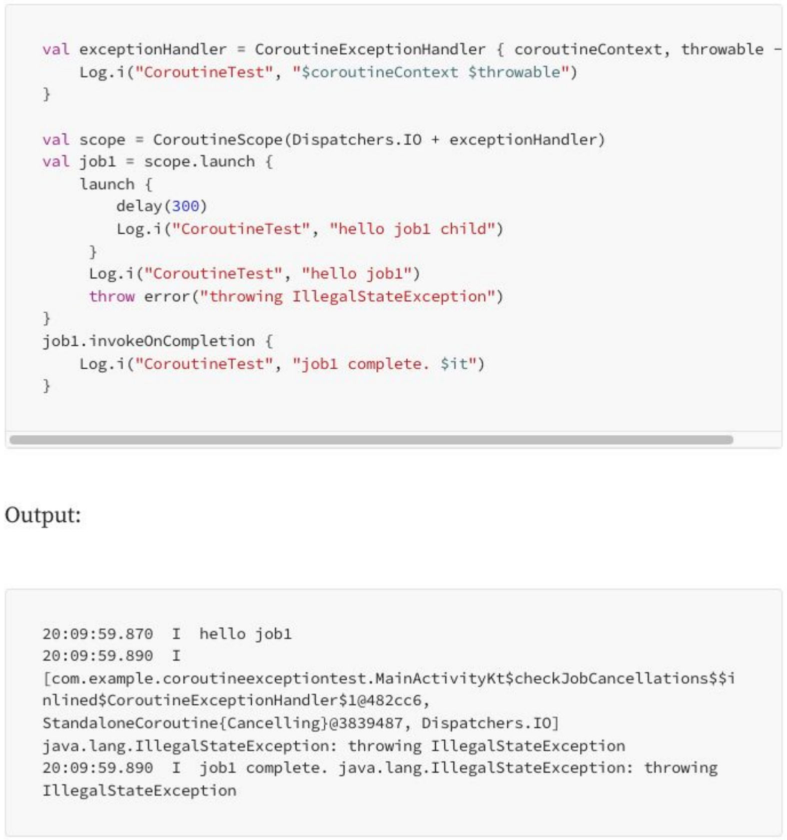
```
val scope = CoroutineScope(Dispatchers.IO)
val job1 = scope.launch {
    launch {
        delay(300)
        Log.i("CoroutineTest", "hello job1 child")
     }
     Log.i("CoroutineTest", "hello job1")
     throw error("throwing IllegalStateException")
}
job1.invokeOnCompletion {
    Log.i("CoroutineTest", "job1 complete. $it")
}
```

Output:



```
hello job1
```

Job



```
val exceptionHandler = CoroutineExceptionHandler { coroutineContext, throwable -
    Log.i("CoroutineTest", "$coroutineContext $throwable")
}

val scope = CoroutineScope(Dispatchers.IO + exceptionHandler)
val job1 = scope.launch {
    launch {
        delay(300)
        Log.i("CoroutineTest", "hello job1 child")
     }
     Log.i("CoroutineTest", "hello job1")
     throw error("throwing IllegalStateException")
}
job1.invokeOnCompletion {
    Log.i("CoroutineTest", "job1 complete. $it")
}
```

Output:

```
20:09:59.870  I  hello job1
20:09:59.890  I
[com.example.coroutineexceptiontest.MainActivityKt$checkJobCancellations$$i
nlined$CoroutineExceptionHandler$1@482cc6,
StandaloneCoroutine{Cancelling}@3839487, Dispatchers.IO]
java.lang.IllegalStateException: throwing IllegalStateException
20:09:59.890  I  job1 complete. java.lang.IllegalStateException: throwing
IllegalStateException
```

Structured Concurrency

every time our control splits into multiple concurrent paths, we make sure they join up again

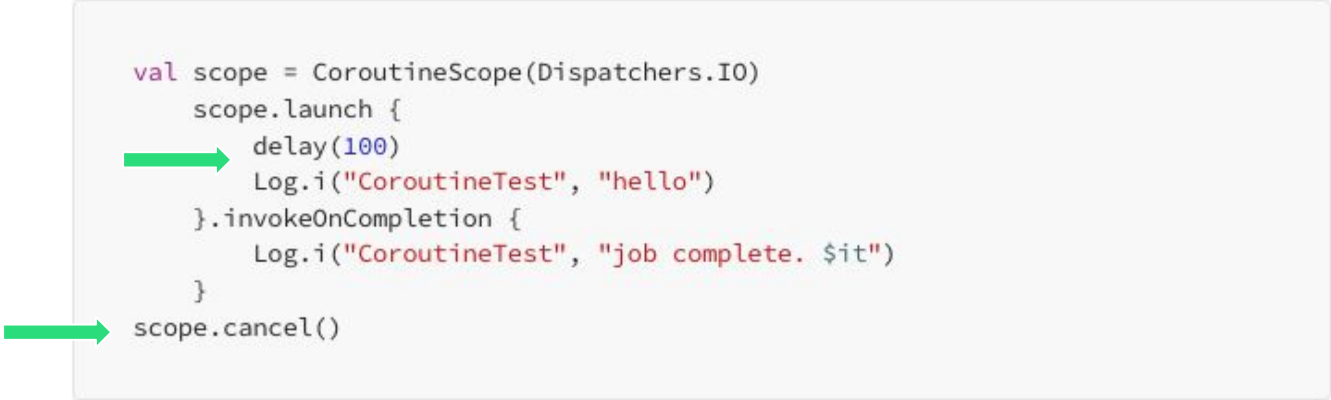
child operations are guaranteed to complete before their parents

no child operation is executed outside the scope of a parent operation

1. When a **scope cancels**, all of its **coroutines cancel**.
2. When a **suspend fun returns**, all of its **work is done**.
3. When a **coroutine errors**, its **caller or scope is notified**.”

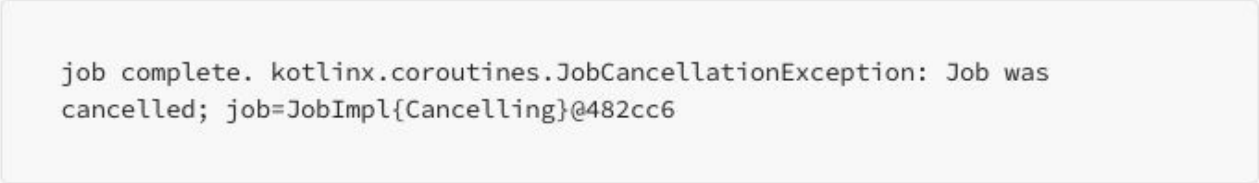
Example : ViewModelScope

Structured Concurrency



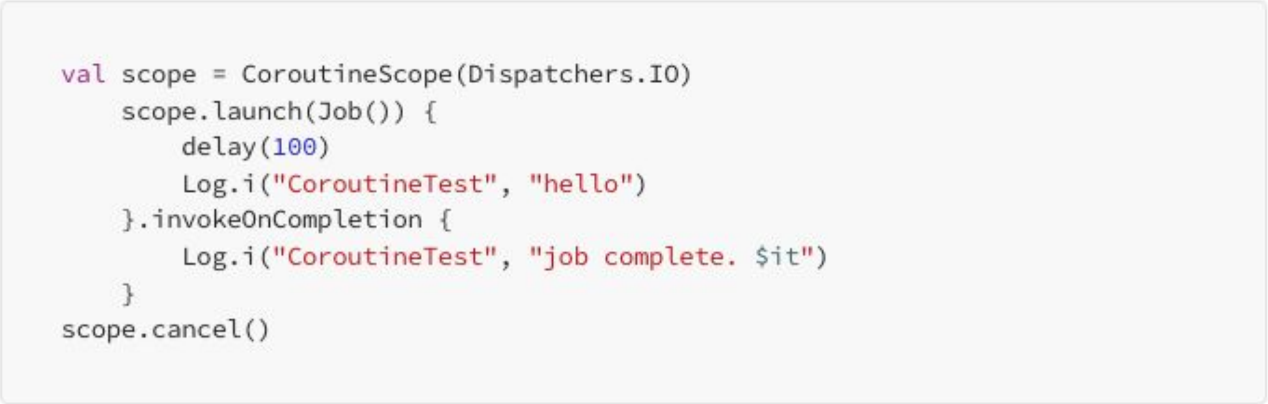
```kotlin
val scope = CoroutineScope(Dispatchers.IO)
    scope.launch {
        delay(100)
        Log.i("CoroutineTest", "hello")
    }.invokeOnCompletion {
        Log.i("CoroutineTest", "job complete. $it")
    }
scope.cancel()
```

Output:



```
job complete. kotlinx.coroutines.JobCancellationException: Job was
cancelled; job=JobImpl{Cancelling}@482cc6
```

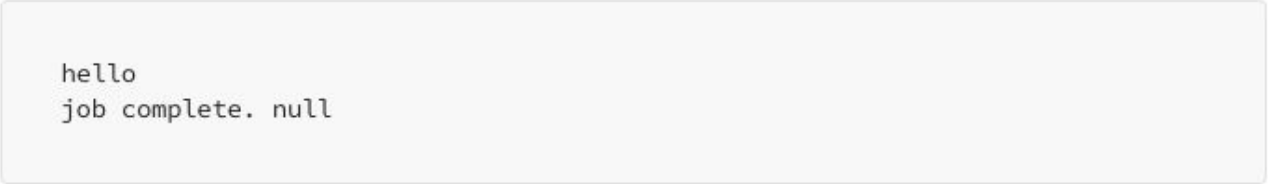
Structured Concurrency



```
val scope = CoroutineScope(Dispatchers.IO)
    scope.launch(Job()) {
        delay(100)
        Log.i("CoroutineTest", "hello")
    }.invokeOnCompletion {
        Log.i("CoroutineTest", "job complete. $it")
    }
scope.cancel()
```

Output:

The structured concurrency is broken



```
hello
job complete. null
```

Coroutine Exception Handler

Coroutine Exception Handler

Coroutine Exception Handler

Created by launch → **We have uncaught exceptions** → **Needs try-catch**

Created by async → **always catches all its exceptions and We don't have uncaught exceptions.**

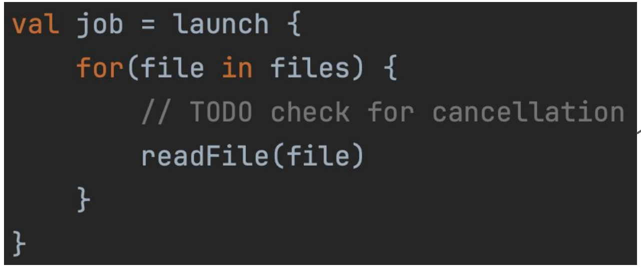
Canceling coroutine

Canceling coroutine

Canceling coroutine

Coroutines handle cancellation by throwing a special exception: CancellationException

If we just call cancel, it doesn’t mean that the coroutine work will just stop.



```kotlin
val job = launch {
    for(file in files) {
        // TODO check for cancellation
        readFile(file)
    }
}
```

isActive - withContext - delay - ...

Kotlin Coroutines

Flow

SharedFlow

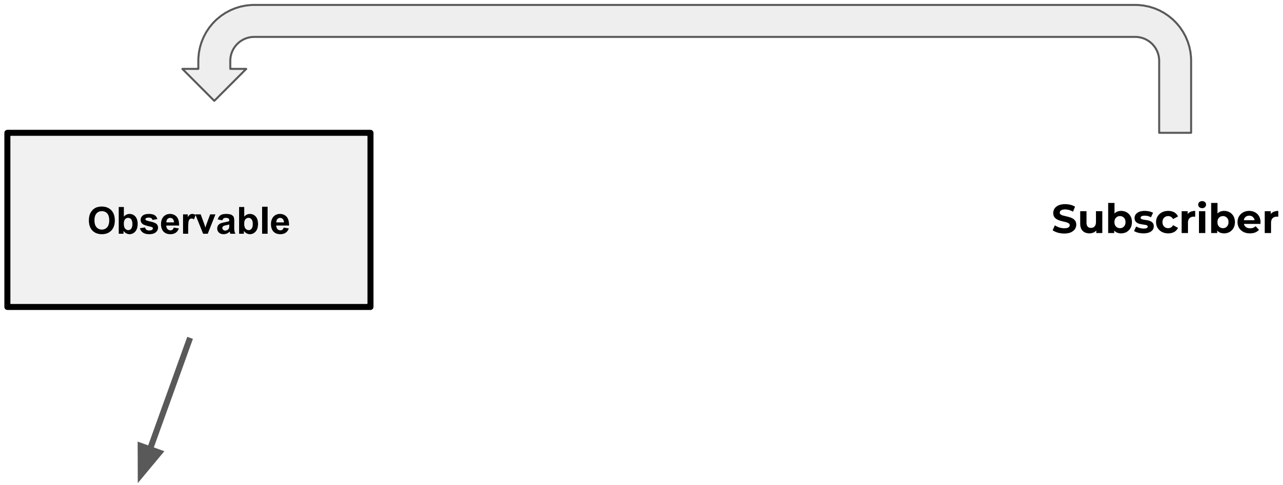
StateFlow

Kotlin Coroutines

Hot & Cold

Hot & Cold

- **observe/subscribe**
- **collect**

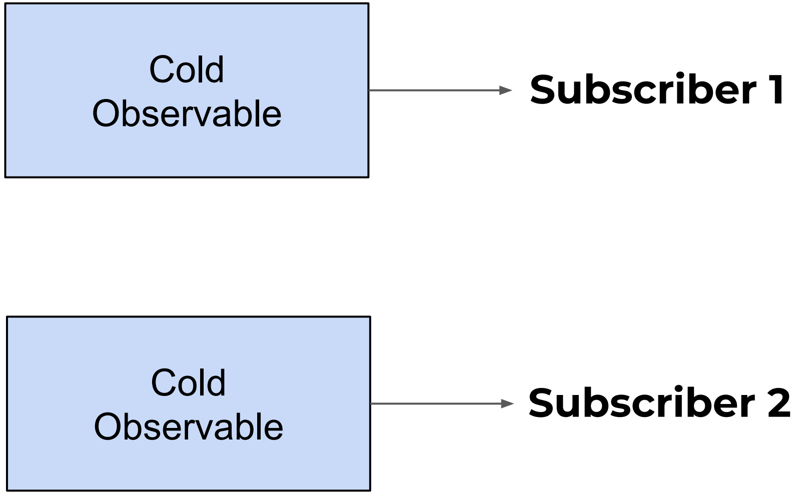


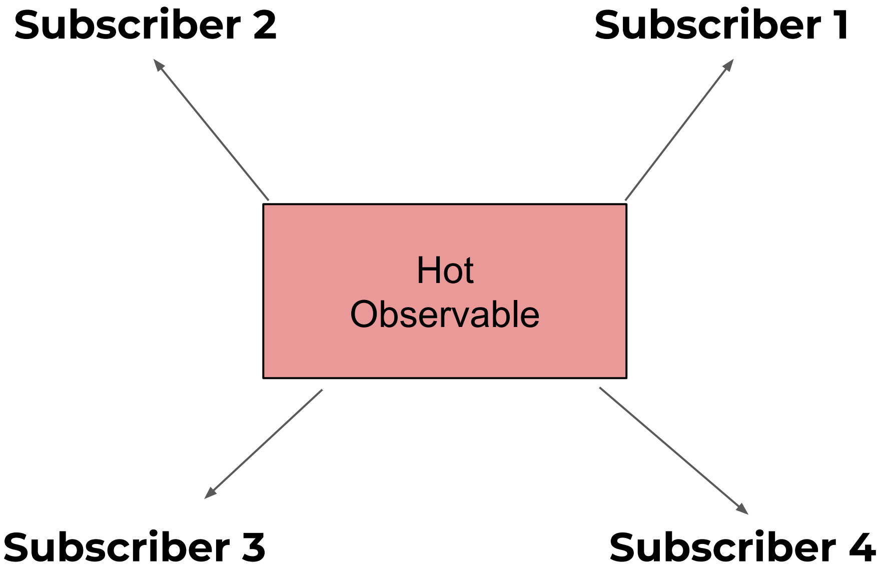
Observable

Subscriber

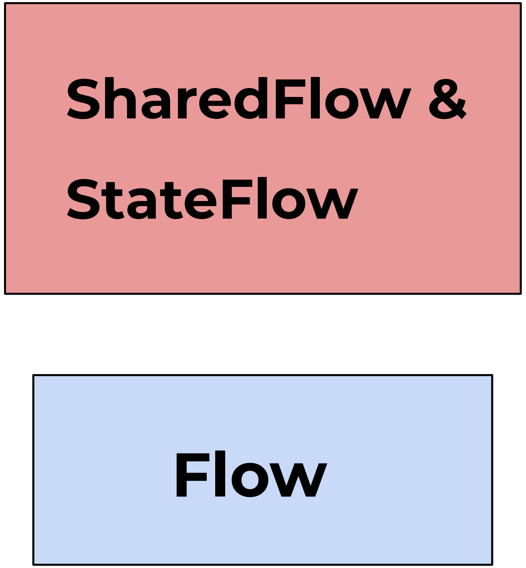
- **Single/Flowable...**
- **Flow/stateFlow/SharedFlow**

Hot & Cold


Cold
Observable
Subscriber 1
Cold
Observable
Subscriber 2


Subscriber 2
Subscriber 1
Hot
Observable
Subscriber 3
Subscriber 4

Hot & Cold



SharedFlow & StateFlow

Flow

Flow & Shared-Flow & State-Flow

Flow

A sequence of values that can be asynchronously computed and delivered over time

Shared-Flow

Allows multiple collectors to listen to the same stream of data independently.

State-Flow

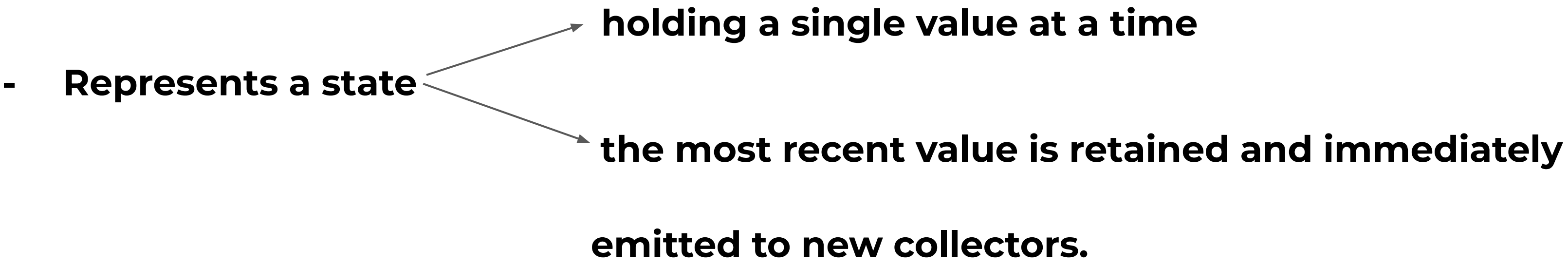
stores the last state(most recent value) and emits it to all it's collectors

Flow & Shared-Flow & State-Flow

- Broadcast a value to multiple collectors

- Multiple subscribers to the same stream of data.

- Store a certain number of previously emitted values



- Represents a state

holding a single value at a time

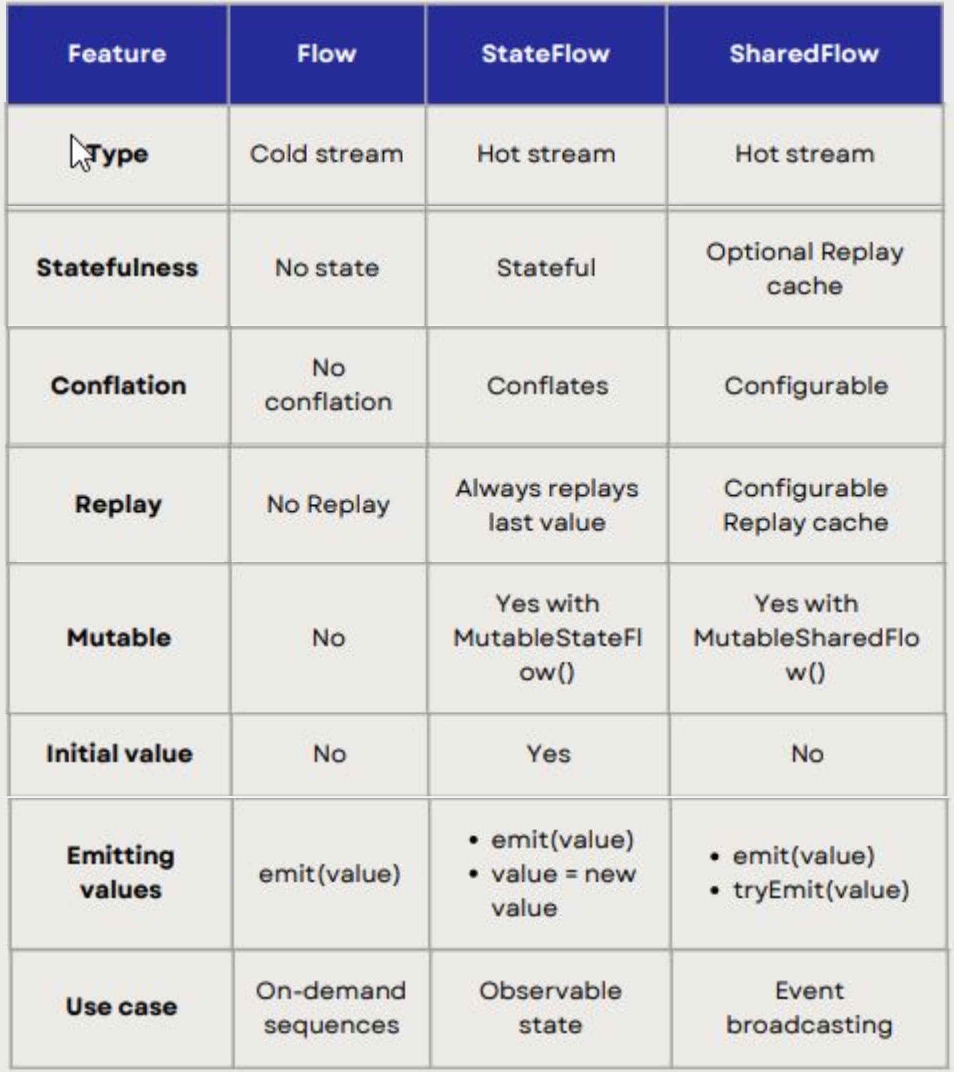
the most recent value is retained and immediately emitted to new collectors.

- Single source of truth for a state

- Automatically update all the collectors with the latest state

Flow & Shared-Flow & State-Flow

Live Datas (Stock price)	**StateFlow**
Event bus	**SharedFlow**
Chat Messaging App	**SharedFlow**



Feature	Flow	StateFlow	SharedFlow
Type	Cold stream	Hot stream	Hot stream
Statefulness	No state	Stateful	Optional Replay cache
Conflation	No conflation	Conflates	Configurable
Replay	No Replay	Always replays last value	Configurable Replay cache
Mutable	No	Yes with MutableStateFlow()	Yes with MutableSharedFlow()
Initial value	No	Yes	No
Emitting values	emit(value)	• emit(value) • value = new value	• emit(value) • tryEmit(value)
Use case	On-demand sequences	Observable state	Event broadcasting

Best Practices In Android

Inject Dispatchers

Suspend functions should be safe to call from the main thread

The ViewModel should create coroutines

Don't expose mutable types

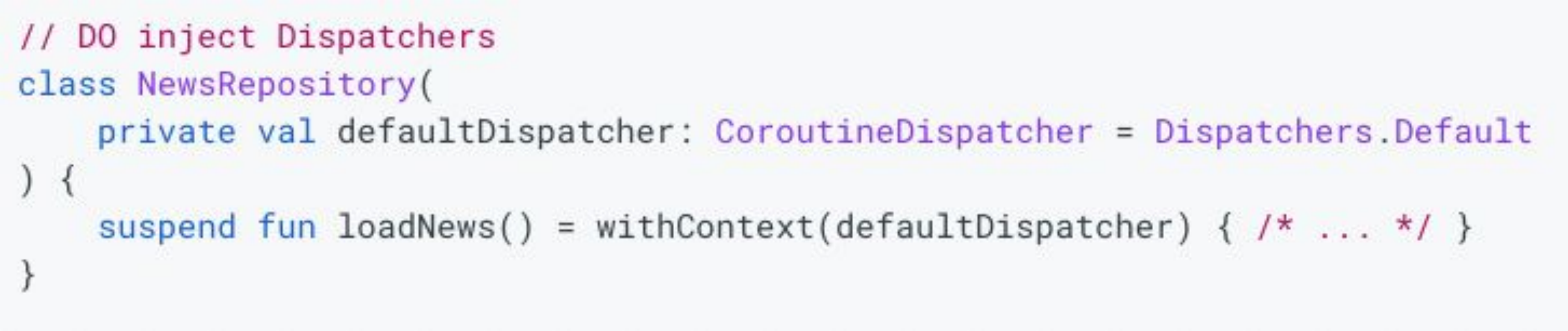
The data and business layer should expose suspend functions and Flows

Creating coroutines in the business and data layer

Avoid GlobalScope

Make your coroutine cancellable

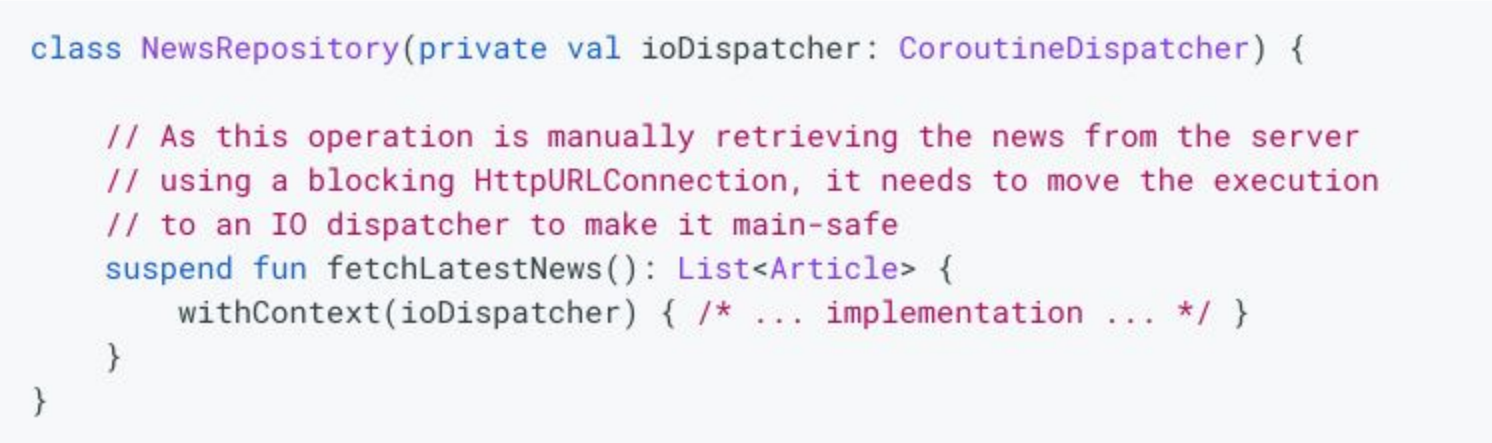
Inject Dispatchers



```kotlin
// DO inject Dispatchers
class NewsRepository(
    private val defaultDispatcher: CoroutineDispatcher = Dispatchers.Default
) {
    suspend fun loadNews() = withContext(defaultDispatcher) { /* ... */ }
}
```

testing easier as you can replace those dispatchers in unit and instrumentation tests with a test dispatcher

Suspend functions should be safe to call from the main thread



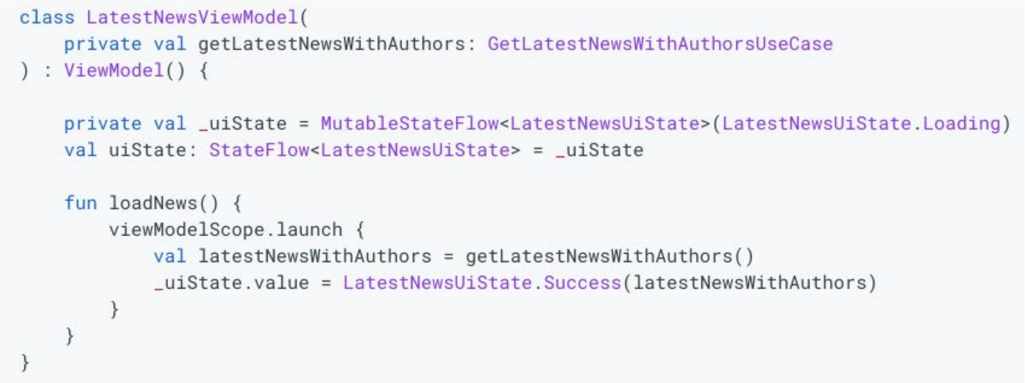
```kotlin
class NewsRepository(private val ioDispatcher: CoroutineDispatcher) {

    // As this operation is manually retrieving the news from the server
    // using a blocking HttpURLConnection, it needs to move the execution
    // to an IO dispatcher to make it main-safe
    suspend fun fetchLatestNews(): List<Article> {
        withContext(ioDispatcher) { /* ... implementation ... */ }
    }
}
```

The ViewModel should create coroutines

- **Views shouldn't directly trigger any coroutines to perform business logic**

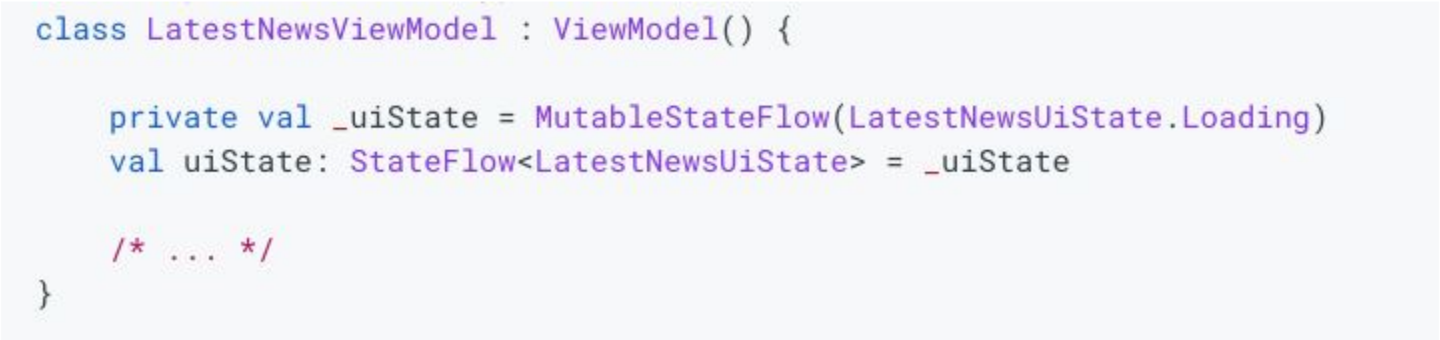
- **your coroutines will survive configuration changes automatically**

- **Views should trigger coroutines for UI-related logic**



```kotlin
class LatestNewsViewModel(
    private val getLatestNewsWithAuthors: GetLatestNewsWithAuthorsUseCase
) : ViewModel() {

    private val _uiState = MutableStateFlow<LatestNewsUiState>(LatestNewsUiState.Loading)
    val uiState: StateFlow<LatestNewsUiState> = _uiState

    fun loadNews() {
        viewModelScope.launch {
            val latestNewsWithAuthors = getLatestNewsWithAuthors()
            _uiState.value = LatestNewsUiState.Success(latestNewsWithAuthors)
        }
    }
}
```

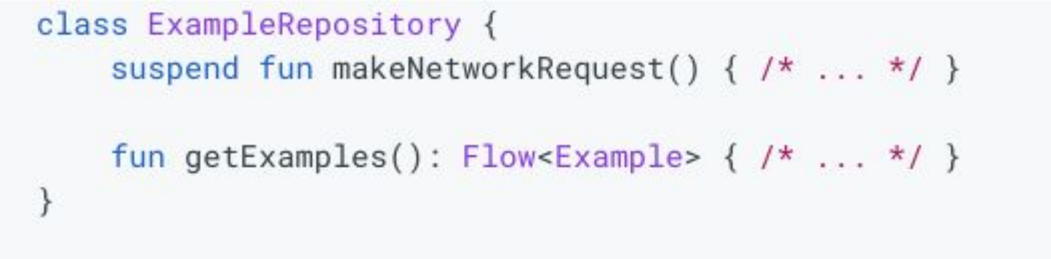
Don't expose mutable types



```
class LatestNewsViewModel : ViewModel() {

    private val _uiState = MutableStateFlow(LatestNewsUiState.Loading)
    val uiState: StateFlow<LatestNewsUiState> = _uiState

    /* ... */
}
```

The data and business layer should expose suspend functions and Flows

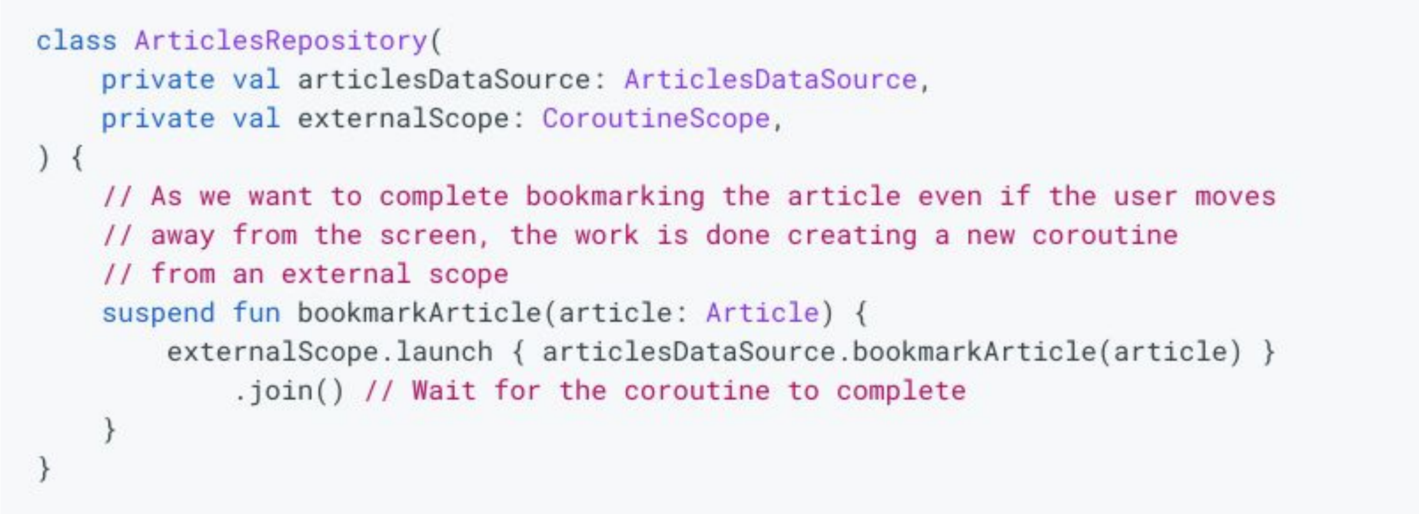
Classes in these layers should expose **suspend functions for one-shot calls** and **Flow to notify about data changes**.



```
class ExampleRepository {
    suspend fun makeNetworkRequest() { /* ... */ }

    fun getExamples(): Flow<Example> { /* ... */ }
}
```

Creating coroutines in the business and data layer

If the work to be done is relevant as long as the app is opened, and the work is not bound to a particular screen, then the work should outlive the caller's lifecycle.

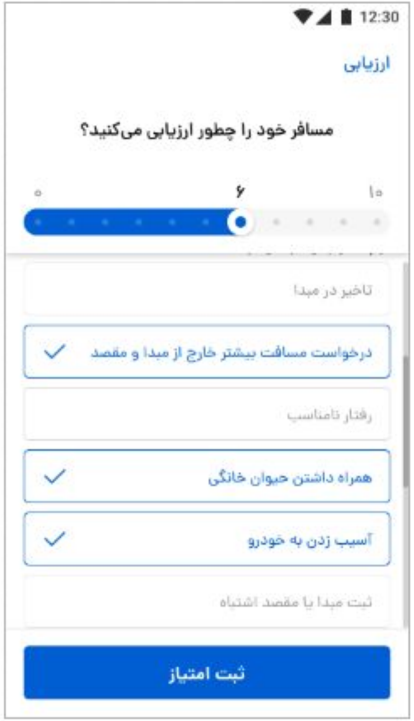


```
class ArticlesRepository(
    private val articlesDataSource: ArticlesDataSource,
    private val externalScope: CoroutineScope,
) {
    // As we want to complete bookmarking the article even if the user moves
    // away from the screen, the work is done creating a new coroutine
    // from an external scope
    suspend fun bookmarkArticle(article: Article) {
        externalScope.launch { articlesDataSource.bookmarkArticle(article) }
            .join() // Wait for the coroutine to complete
    }
}
```

`externalScope` should be created and managed by a class that lives longer than the current screen, it could be managed by the `Application` class or a `ViewModel` scoped to a navigation graph.

Creating coroutines in the business and data layer

`Example: Driver Optimistic NPS`



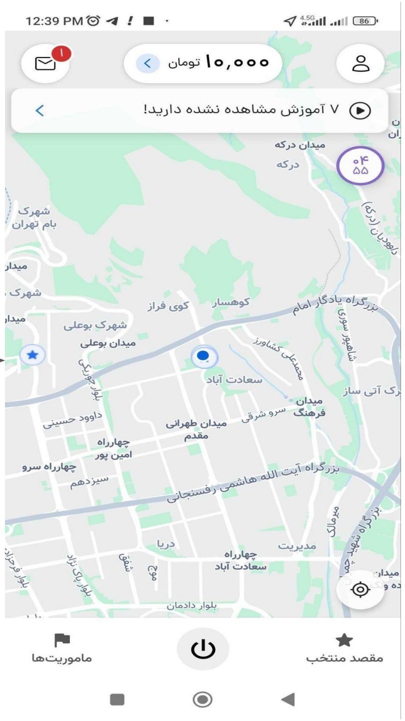
ارزیابی

مسافر خود را چطور ارزیابی می‌کنید؟

۰ ۶ ۱۰

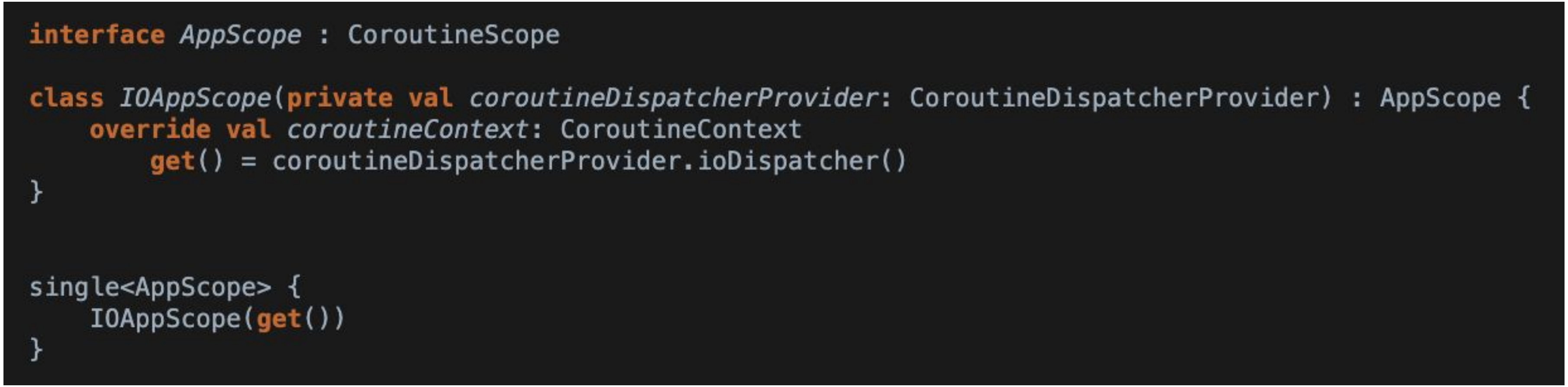
- تاخیر در مبدا
- ✓ درخواست مسافت بیشتر خارج از مبدا و مقصد
- رفتار نامناسب
- ✓ همراه داشتن حیوان خانگی
- ✓ آسیب زدن به خودرو
- ثبت مبدا یا مقصد اشتباه

ثبت امتیاز



Creating coroutines in the business and data layer

Example Of injecting external scope : Driver Optimistic NPS

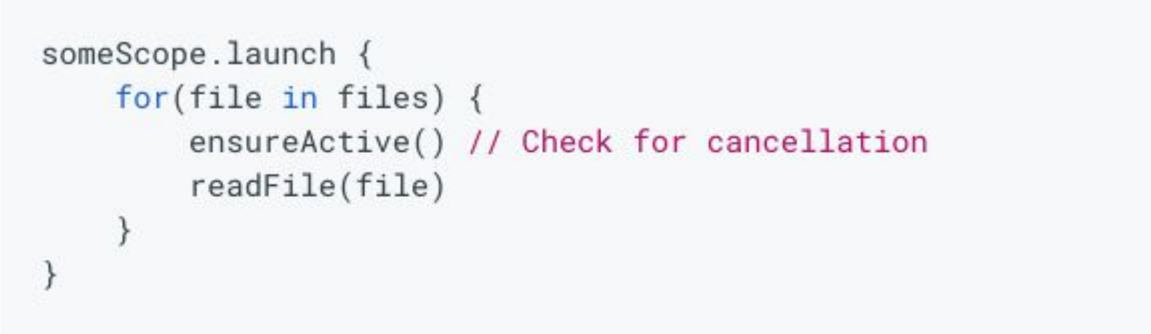


```
interface AppScope : CoroutineScope

class IOAppScope(private val coroutineDispatcherProvider: CoroutineDispatcherProvider) : AppScope {
    override val coroutineContext: CoroutineContext
        get() = coroutineDispatcherProvider.ioDispatcher()
}


single<AppScope> {
    IOAppScope(get())
}
```

Avoid GlobalScope

- **Makes testing very hard as your code is executed in an uncontrolled scope, you won't be able to control its execution.**

Make your coroutine cancellable



```
someScope.launch {
    for(file in files) {
        ensureActive() // Check for cancellation
        readFile(file)
    }
}
```

Thanks!