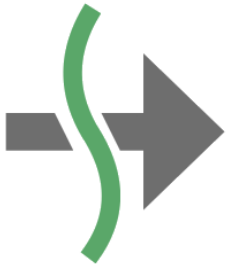
CMPS 312

Kotlin Coroutines for Asynchronous and Concurrent Programming



Dr. Abdelkarim Erradi

CSE@QU

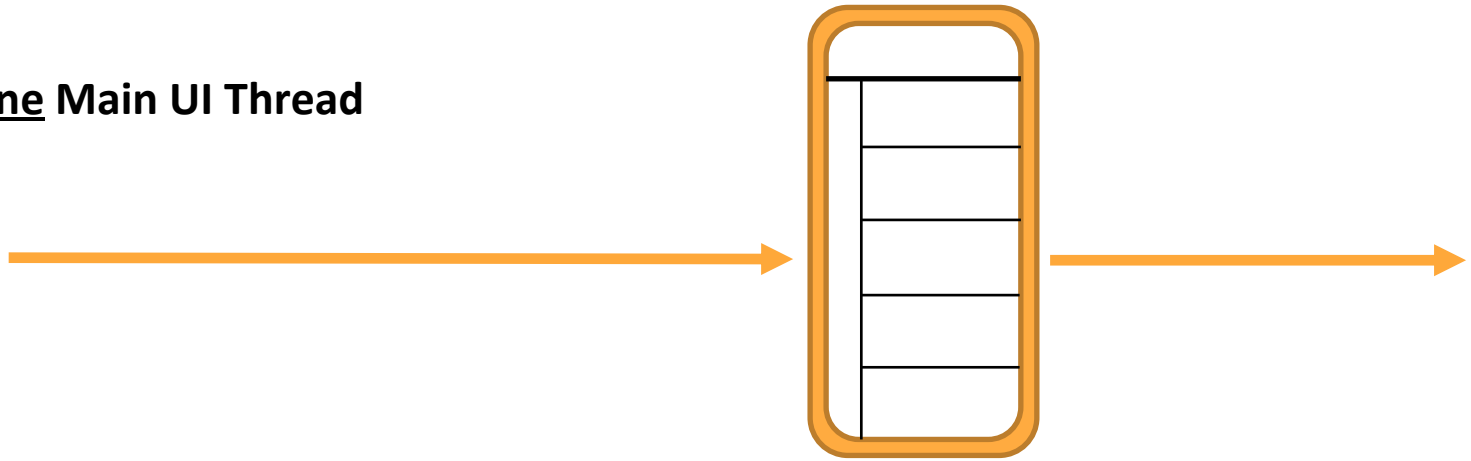
Outline

1. Coroutines Basics
2. Coroutines Programming Model
3. Flow

Coroutines Basics

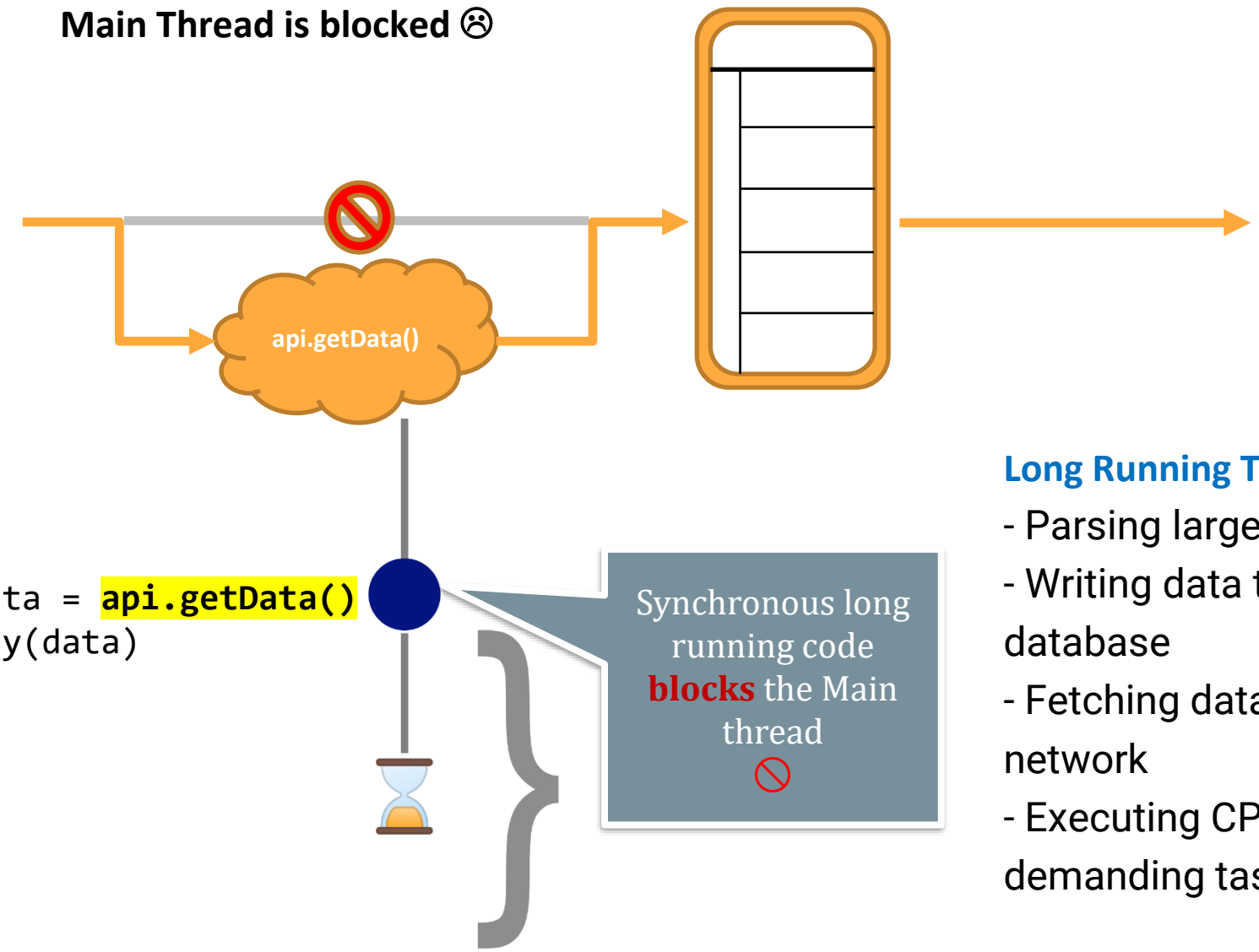
User Interface Running on the Main Thread

One Main UI Thread



To guarantee a great user experience, it's essential to **avoid blocking the main thread** as it used to handle UI updates and UI events

Long Running Task on the Main Thread

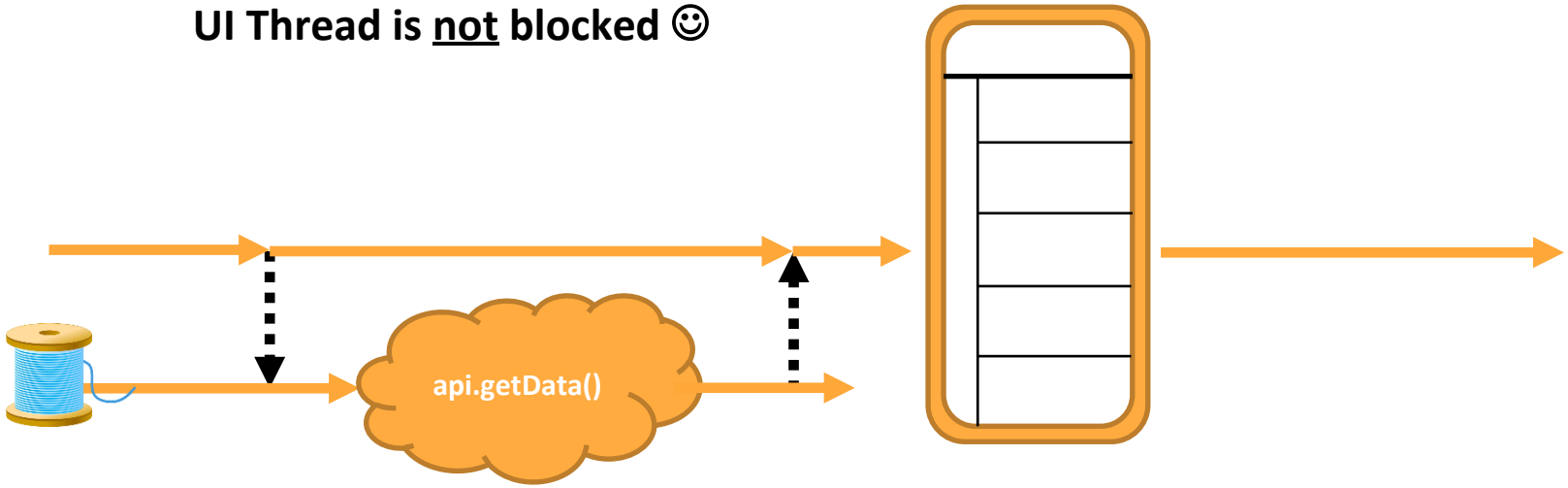


Long Running Task include:

- Parsing large JSON file
- Writing data to a database
- Fetching data from the network
- Executing CPU demanding task

Solution 1 – Run Long Running tasks on a background thread

UI Thread is <u>not</u> blocked ☺




```
var data
thread {
    data = api.getData()
}
display(data)
```

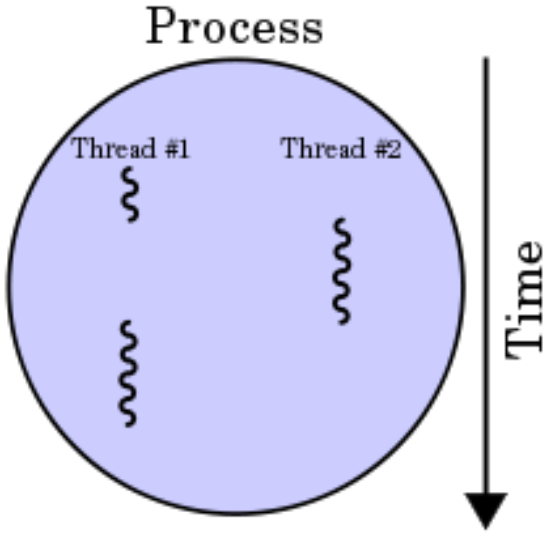
- Threads introduces many challenges related to data communication and synchronization between threads
- Plus UI can only be accessed from the Main thread

How to address problem of long-running task?

- How to create code that doesn't block Main thread?

Solution 1: **Use multi-threading**

- A thread is the **unit of execution** within a process
  - When a process starts, it is assigned memory and resources
  - Each thread in the process shares that memory and resources



Callback pattern

- By using callbacks, you can start long-running tasks on a background thread
- When the task completes, the callback is called to notify the main thread of the result

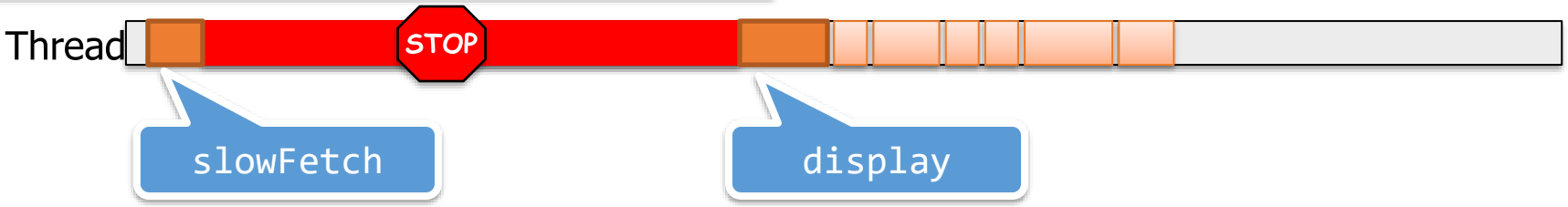
```
// Slow request with callbacks
fun makeNetworkRequest() {
    // The slow network request runs on another thread
    slowFetch { result ->
        // When the result is ready, this callback will get the result
        display(result)
    }
    // makeNetworkRequest() exits after calling slowFetch without waiting for the result
}
```

However, code that heavily uses callbacks can become hard to read and harder to reason about + poor handling of exceptions

Synchronous vs. Asynchronous

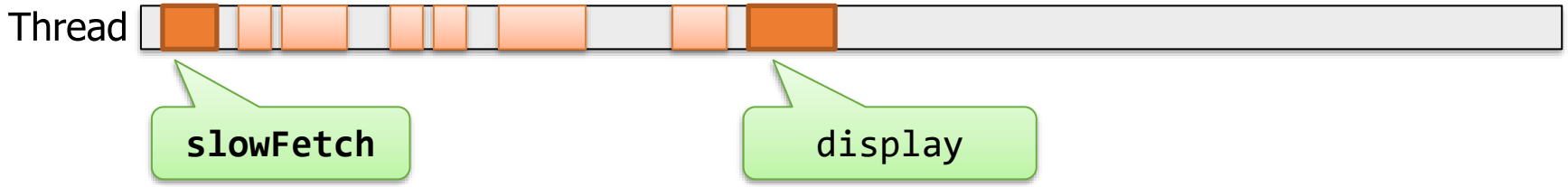
Synchronous → Wait for result before returning

```
val result = slowFetch(...) // UI Thread
display(result)  // UI Thread
```




```
// Slow request with callbacks
fun makeNetworkRequest() {
    // The slow network request runs on another thread
    slowFetch { result ->
       // When the result is ready, this callback will get the result
       display(result)
    }
}
```

Asynchronous → do an **asynchronous** call to slowFetch using backgroud thread, then update UI with the result



UI not blocked

Callback Hell…

```
// Simplified code that only considers the happy path
fun loginUser(userId: String, password: String, userResult: Callback<User>) {
    // Async callbacks
    userRemoteDataSource.loginUser { user ->
        // Successful network request
        userLocalDataSource.logUserIn(user) { userDb ->
            // Result saved in DB
            userResult.success(userDb)
        }
    }
}
```

aka “callback hell”

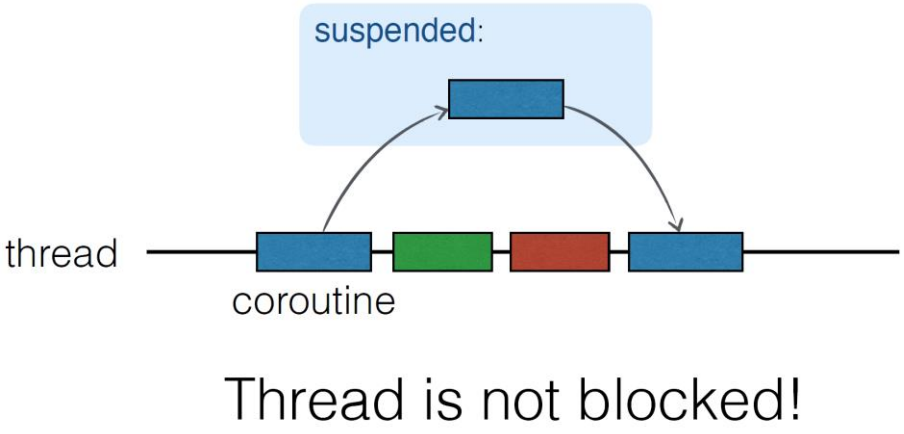
Callback hell = Heavily nested functions are hard to understand

Thread Limitations

- All **threads** are **costly** (occupy 1-2 mb)
- Some **threads** are **special** (e.g. Main UI thread) and should not be blocked

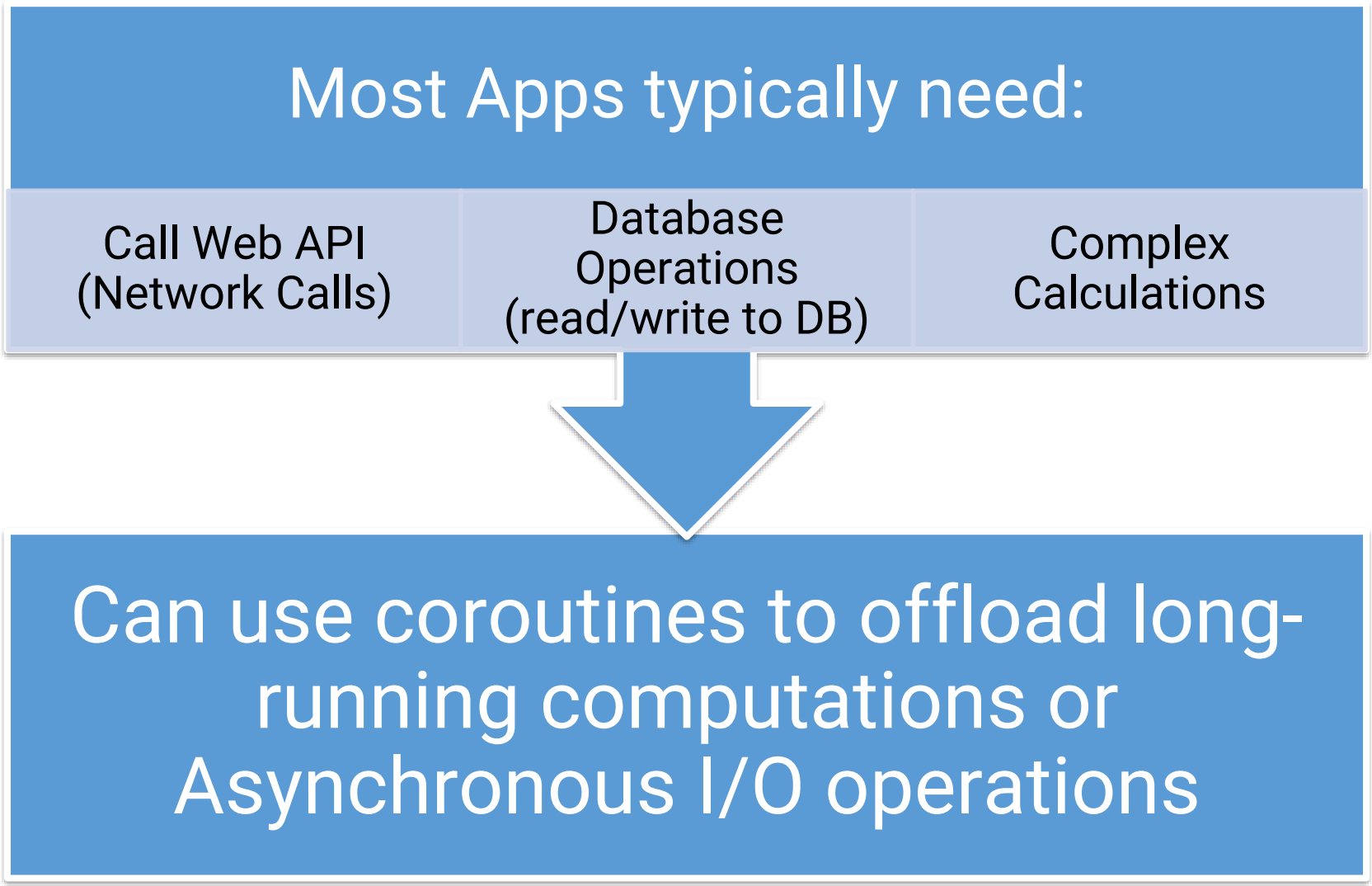
Better alternative are **Coroutines**

Coroutine = computation that can be suspended



Coroutines are like light-weight threads

Why Coroutines?



Example Slow request with coroutines

```
// Slow request with coroutines
suspend fun makeNetworkRequest() {
    // slowFetch is another suspend function so instead of
    // blocking the main thread  makeNetworkRequest will `suspend` until
    // the result is ready
    val result = slowFetch()
    // continue to execute after the result is ready
    display(result)
}
// slowFetch is suspend function that can be called using coroutines
suspend fun slowFetch(): Result { ... }
```

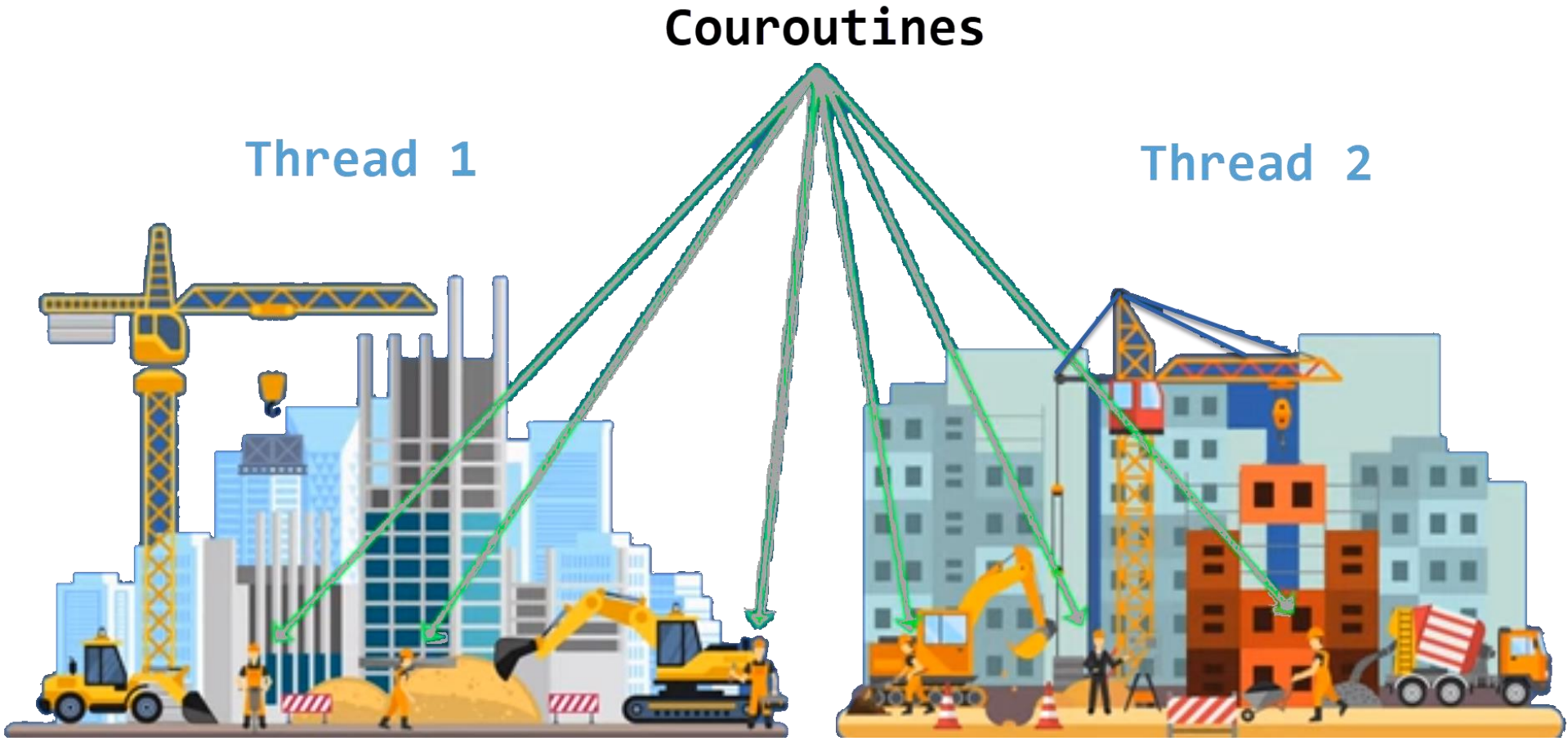
- Compared to callback-based code, coroutine code accomplishes the same result of unblocking the current thread with less code.

- Due to its sequential style, it's easier to understand + it's easy to chain several long running tasks without creating multiple callbacks

What distinguishes Coroutines from Threads? 🤔

1. Coroutines are like light-weight threads
   - Takes a block of code to run within a thread
2. Easier exception handling and **cancellation**
3. They can switch their context
   - e.g., do a Network call using the IO Thread then switch to the Main thread to update the UI
4. Simplify asynchronous programming
   - Replace callback-based code with sequential code to handle long-running tasks without blocking

Thread vs. Coroutine

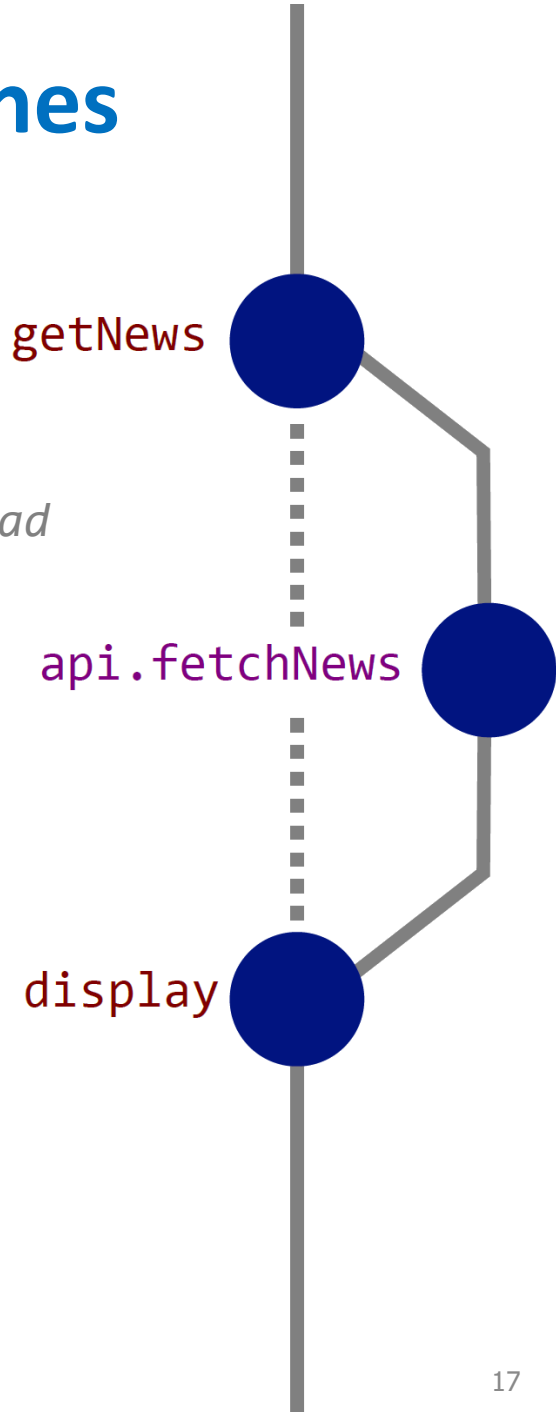


Source: https://www.youtube.com/watch?v=ShNhJ3wMpvQ

Suspendable

Async Programming with Coroutines

```
suspend fun getNews() {
    val news = api.fetchNews()  // Background thread
    withContext(Dispatchers.Main) {
        display(news)           // UI thread
    }
}
```



Key benefit of Async Programming = ***Responsiveness***

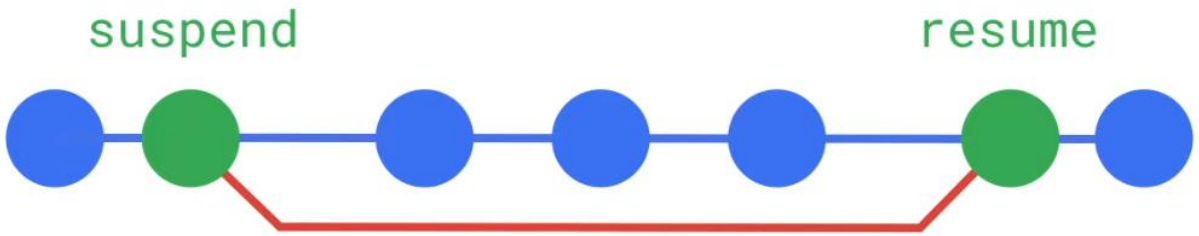
prevent blocking the UI thread on long-running operations

Callback vs. Coroutine

```
// Simplified code that only considers the happy path
fun loginUser(userId: String, password: String, userResult: Callback<User>) {
    // Async callbacks
    userRemoteDataSource.loginUser { user ->
        // Successful network request
        userLocalDataSource.logUserIn(user) { userDb ->
            // Result saved in DB
            userResult.success(userDb)
        }
    }
}
```

```
suspend fun loginUser(userId: String, password: String): User {
    val user = userRemoteDataSource.loginUser(userId, password)
    val userDb = userLocalDataSource.logUserIn(user)
    return userDb
}
```

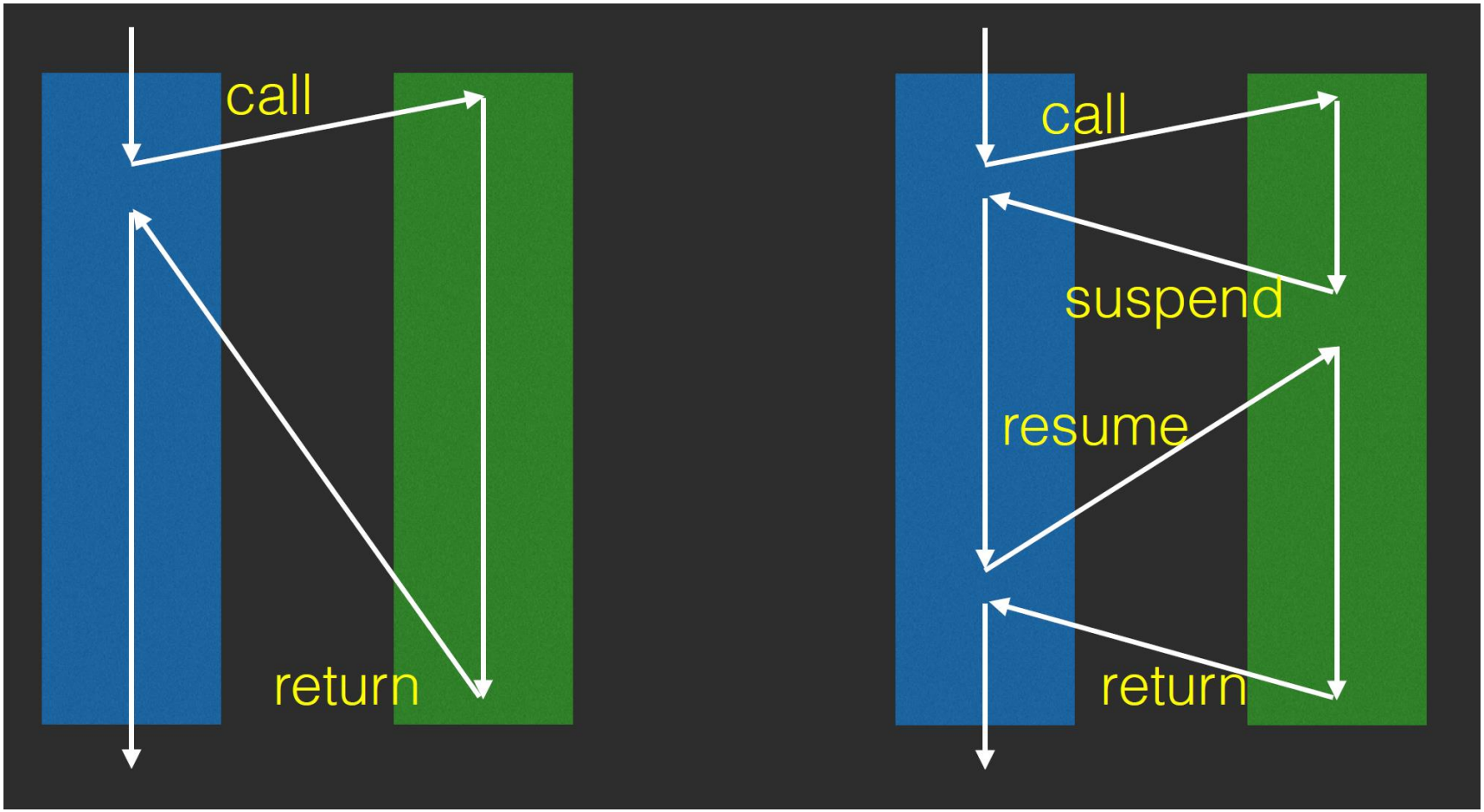
Coroutines Programming Model



Suspend function

- **Suspend** function is a function that can be **suspended** and **resumed**
  - **suspend** is Kotlin's way of marking a function available to coroutines
- When a coroutine calls a function marked suspend, instead of blocking until that function returns:
  - it **suspends** execution until the result is ready then
  - it **resumes** where it left off with the result
- While it's suspended waiting for a result, **it unblocks the thread that it's running on** so other functions or coroutines can run

Function vs. Suspend Function

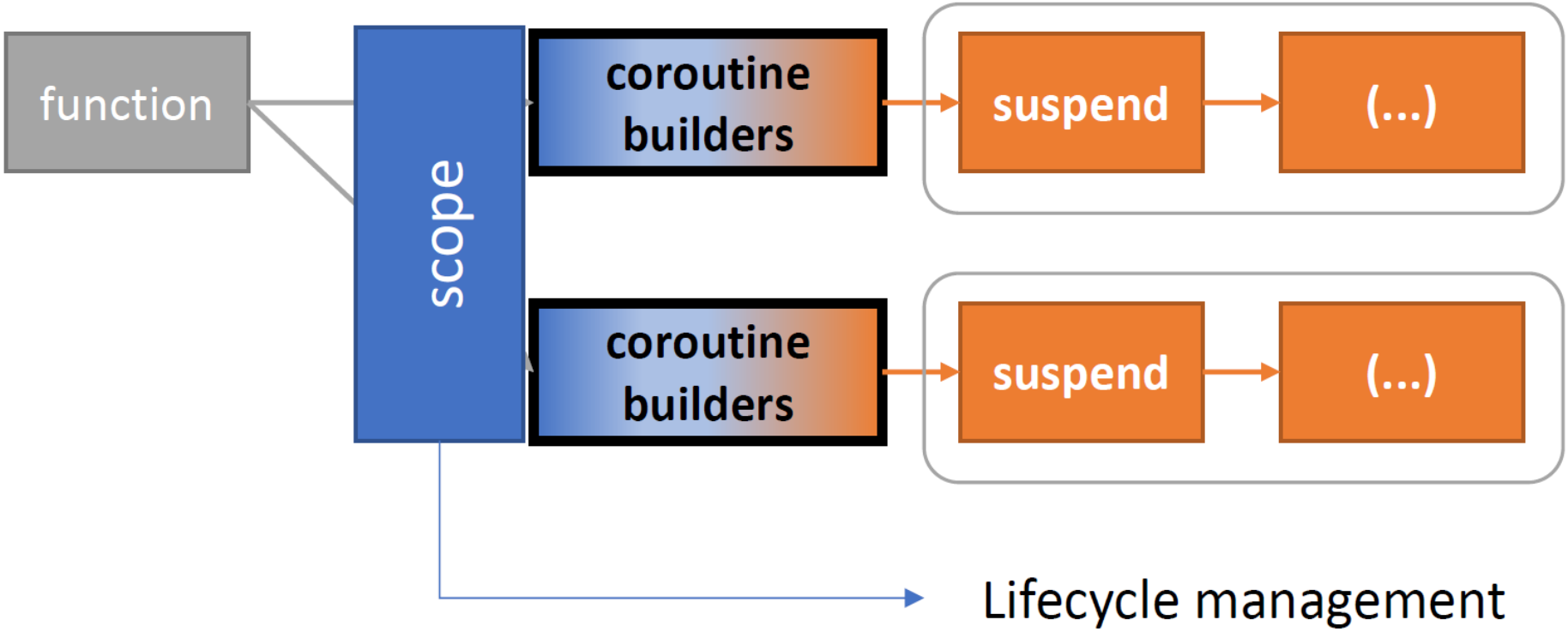


To launch a Coroutine you need a Coroutine Scope

- A suspend function must be called in a coroutine
- A Coroutine Scope is required to create and start a coroutine using the scope's **launch** or **async** methods
- Coroutine scope keeps track of coroutines to allow the ability to cancel it and to handle exceptions
- On Android you could use provided scoped:
  - *viewModelScope*, *lifecycleScope*
  - GlobalScope is an app-level scope (rarely used). It lives as long as the app does

Coroutine Scope Enable Cancellation

- A coroutine is **always** created in the context of a **scope**
  - Keep track of coroutines
  - Ability to cancel them
  - Is notified of failures



viewModelScope

- **viewModelScope** can be used in any ViewModel in the app
- Any coroutine launched in this scope is **automatically canceled** if the ViewModel is cleared (to avoid consuming resources unnecessarily)

```
class MyViewModel: ViewModel() {
    init {
        viewModelScope.launch {
            // Coroutine that will be canceled when the ViewModel is cleared
        }
    }
}
```

lifecycleScope

- **lifecycleScope** can be used in an activity/fragment
- Any coroutine launched in this scope is canceled when the Lifecycle is destroyed

```
class MyFragment: Fragment() {
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)
        lifecycleScope.launch {
            // Coroutine that will be canceled when the fragment is destroyed
        }
    }
}
```

`liveData` coroutine builder

- Use the `liveData` builder function to call a suspend function and return the result as a LiveData object


```
// Use the liveData builder function to call fetchUser()
// asynchronously and then use emit() to emit the result
val user: LiveData<User> = liveData {
    // fetchUser is a suspend function.
    val user = api.fetchUser(email)
    emit(user)
}
```

liveData with switchMap

- One-to-one dynamic transformation
  - E.g., whenever the userId changes automatically fetch the user details

```
class MyViewModel: ViewModel() {
    private val userId: LiveData<String> = MutableLiveData()

    val user = userId.switchMap { id ->
        liveData(context = viewModelScope.coroutineContext + Dispatchers.IO) {
            emit(api.fetechUserById(id))
        }
    }
}
```

Coroutine builder functions

Use a coroutine builder to **create** and **start** a **coroutine**

- **Launch** Fire and forget

```
scope.launch(Dispatchers.IO) {
    loggingService.upload(logs)
}
```

- **Async** Return a value

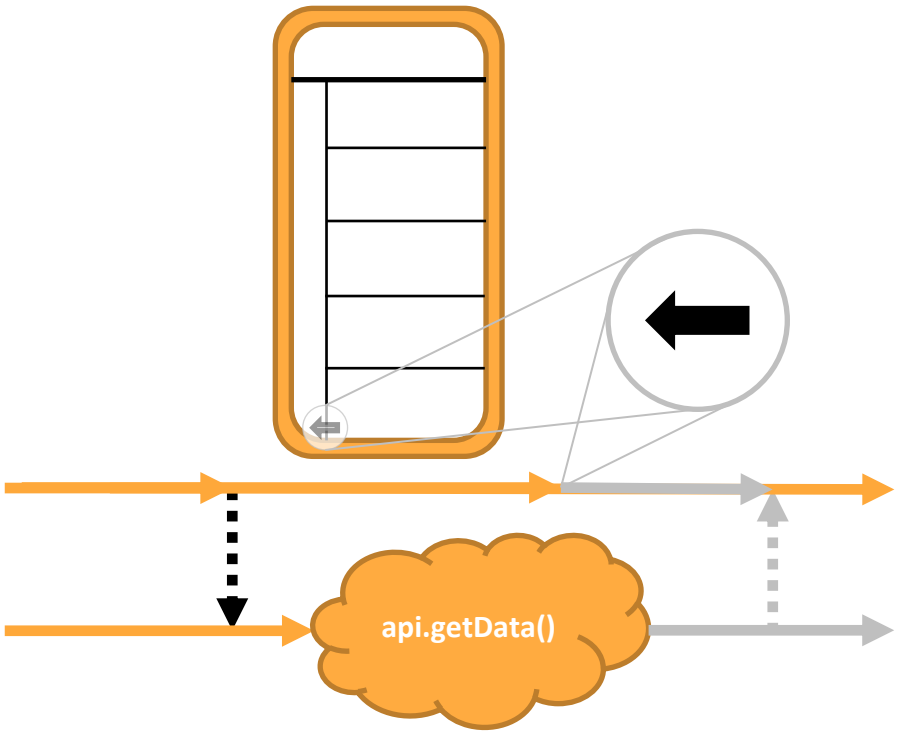
```
suspend fun getUser(userId: String): User =
    coroutineScope {
        val deferred = async(Dispatchers.IO) {
            userService.getUser(userId)
        }
        deferred.await()
    }
```

async-await

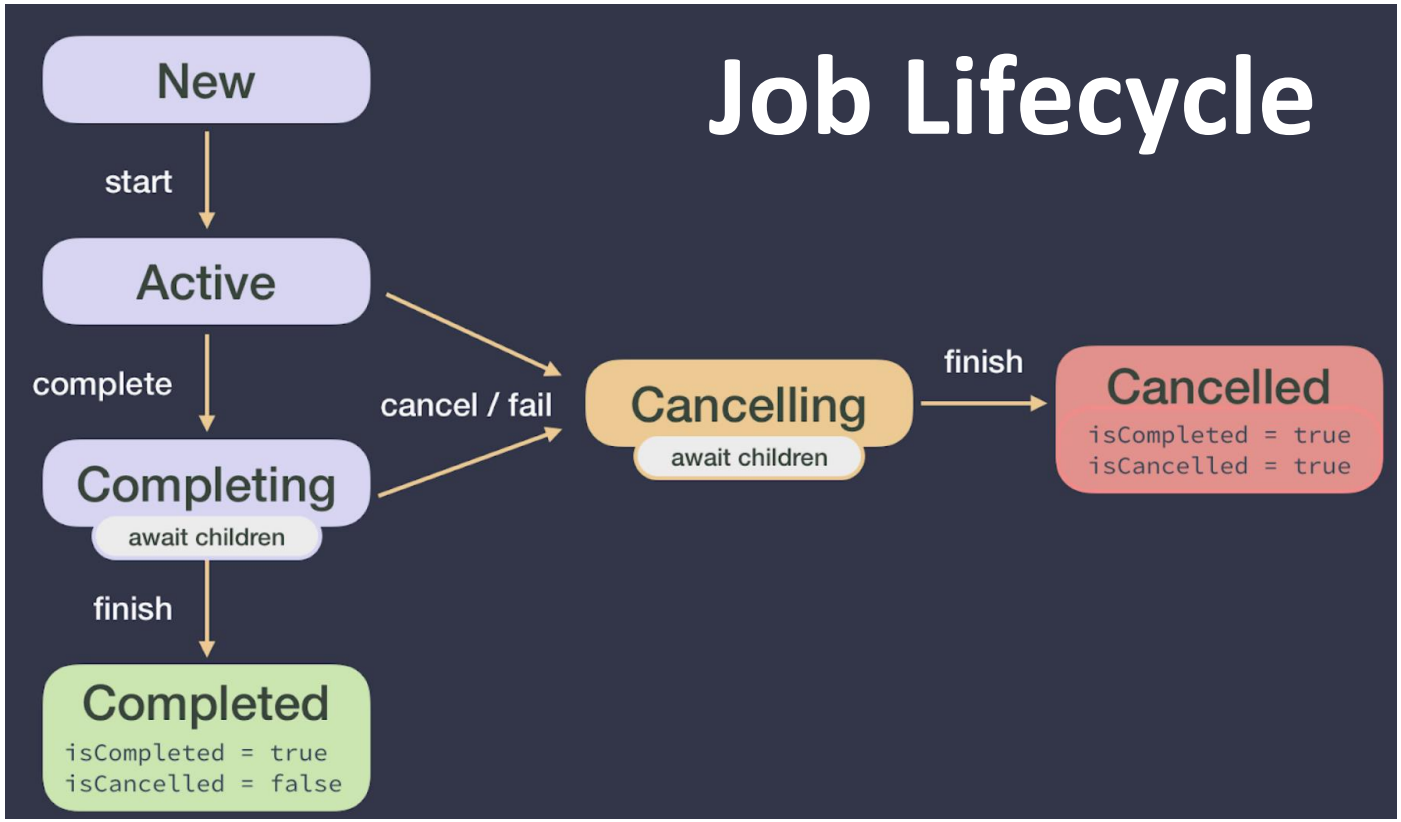
- async-await allow launching **concurrent asynchronous tasks** (executed in parallel)

```
// Download of the 2 images will be done in Parallel
suspend fun loadAndCombine(name1: String, name2: String): Image =
    coroutineScope {
        val deferred1 = async { loadImage(name1) }
        val deferred2 = async { loadImage(name2) }
        // awaits the result of the asynchronous computation
        combineImages(deferred1.await(), deferred2.await())
    }
```

Coroutine Cancelling



When the View is destroyed (e.g., Back Button pressed).
How to cancel api.getData task?
Otherwise waste memory and battery life + possible memory leak of UI that listens to the result of getData task



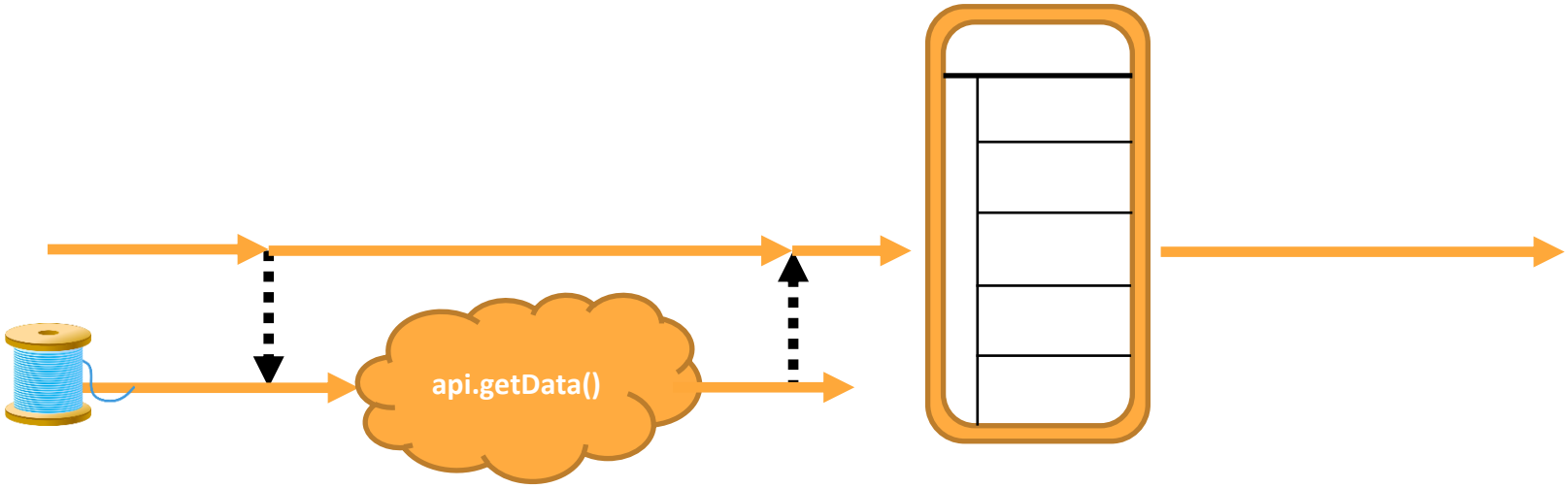

```
val job = lifecycleScope.launch(Dispatchers.Default) {
    fibonacci()
}
...
// onCancel clicked
job.cancel()
```

Coroutine Cancelling

```
// assume we have a scope defined for this layer of the app
val job1 = scope.launch { … }
val job2 = scope.launch { … }
// Cancelling the scope cancels its children
scope.cancel()

// First coroutine will be cancelled and the other
// one won’t be affected
job1.cancel()
```

Swap between threads



Perform fetch data on background thread then when the result is ready update the UI on Main thread

```
lifecycleScope.launch  {
    fibonacci(1_000_000)
    withContext(Dispatchers.Main) {
        scoreTv.text = score.toString()
    }
}
```

Switch to Main Thread to update the UI

Swap between threads

```
withContext(Dispatchers.?) {   ...   }
```

- **`withContext`** allows you to *decide where* do want to run the computation
- Use **`withContext`** to swap between different Dispatchers to execute computations on different threads:
  - **Dispatchers.IO**: Optimized for Network and Disk operations
  - **Dispatchers.Default**: used form CPU-intensive tasks
  - **Dispatchers.Main**: Used for updating the UI

Flow

Back


One-shot operation
Observers
Tweet
Android
@Android
Tens across the board. #Android10 is here and more helpful than ever.
0:05 · 07 Sep 19 · Sprinklr
374 Retweets 2,744 Likes

What is Flow?

- Stream of values (produced one at a time instead of all at once)
  - Values could be generated from *async* operations like network requests, database calls
- Can transform a flow using operators like map, switchMap, etc
- Built on top of coroutines

```
fun stream(): Flow<String> = flow {
    emit("🌊") // Emits the value upstream ☝
    emit("⚽")
    emit("🍕")
}
```

Return Flow: Stream of Data


```
object WeatherRepository {
    private val weatherConditions = listOf("Sunny", "Windy", "Rainy", "Snowy")
    fun fetchWeatherFlow(): Flow<String> =
        flow {
            var counter = 0
            while (true) {
                counter++
                delay(2800)
                emit(weatherConditions[counter % weatherConditions.size])
            }
        }
}
```

```
val currentWeatherFlow: LiveData<String> =
        WeatherRepository.fetchWeatherFlow().asLiveData()
```

Frow from Repeat API Call

- Create a Flow for repeated periodic API calls

```
private val repeatCall =
    flow<ApiResult> {
        while (true) {
            emit(api.fetchData())
            delay(10.minutes)
        }
    }
```

Flow Operations

```
val currentWeatherFlow: Flow<String> =
    dataSource.fetchWeatherFlow()
        .map { ... }
        .filter { ... }
        .dropWhile { ... }
        .combine { ... }
        .flowOn(Dispatchers.IO)
        .onCompletion { ... }
...
```

Resources

- Kotlin coroutines on Android
  - https://developer.android.com/kotlin/coroutines
- Part 1: Coroutines, Part 2: Cancellation in coroutines, and Part 3: Exceptions in coroutines
- Coroutines codelab
  - https://codelabs.developers.google.com/codelabs/kotlin-coroutines
  - https://codelabs.developers.google.com/codelabs/advanced-kotlin-coroutines