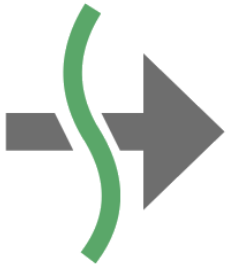
CMPS 312

Kotlin Coroutines for Asynchronous and Concurrent Programming



Dr. Abdelkarim Erradi

CSE@QU

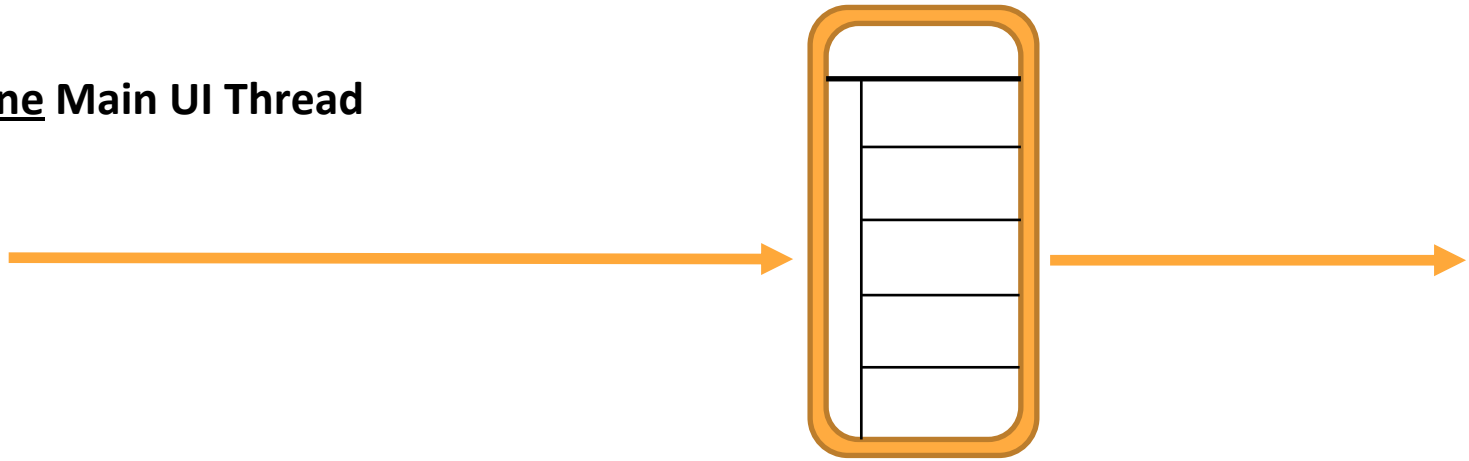
Outline

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

Coroutines Basics

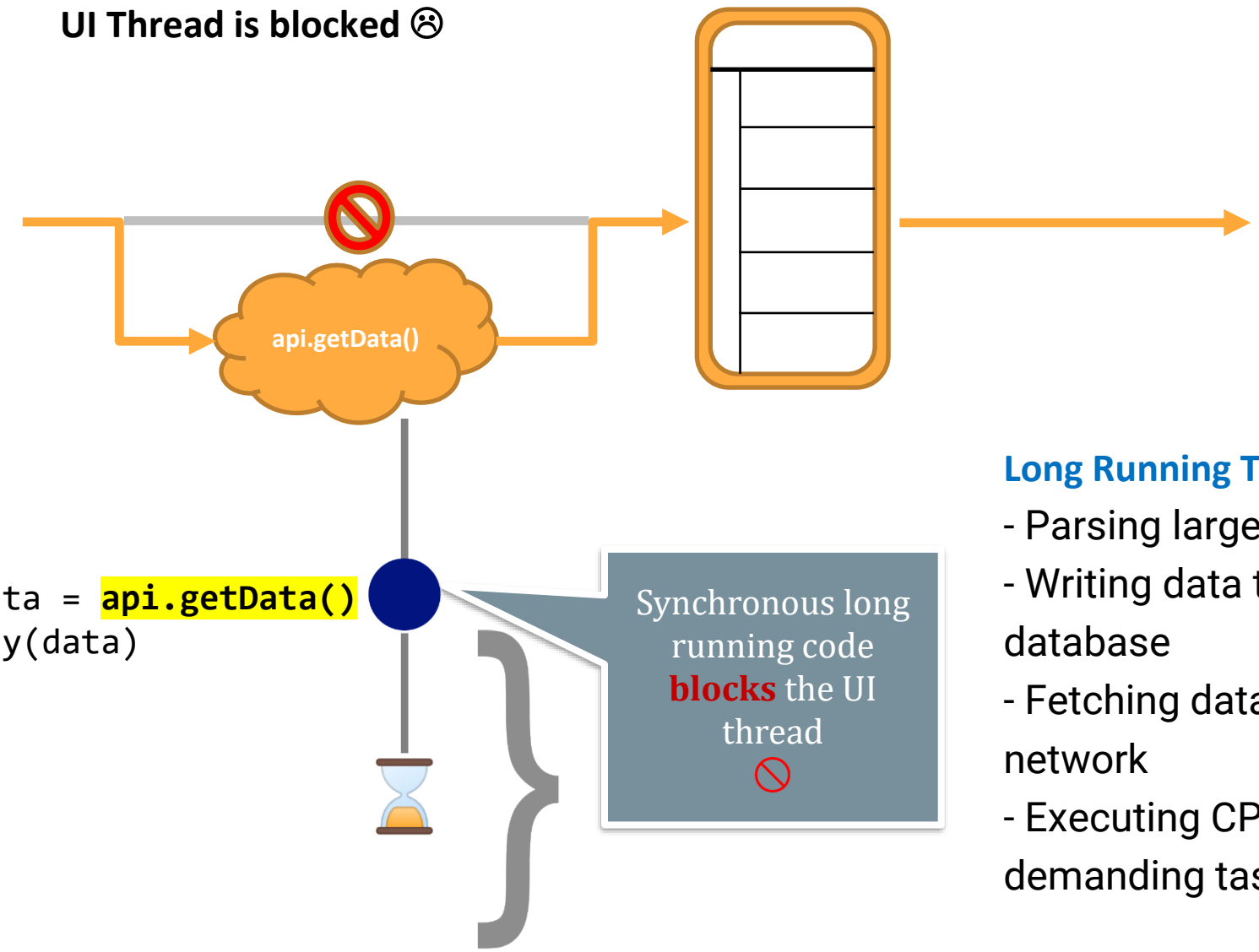
User Interface Running on UI Thread

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 UI Thread

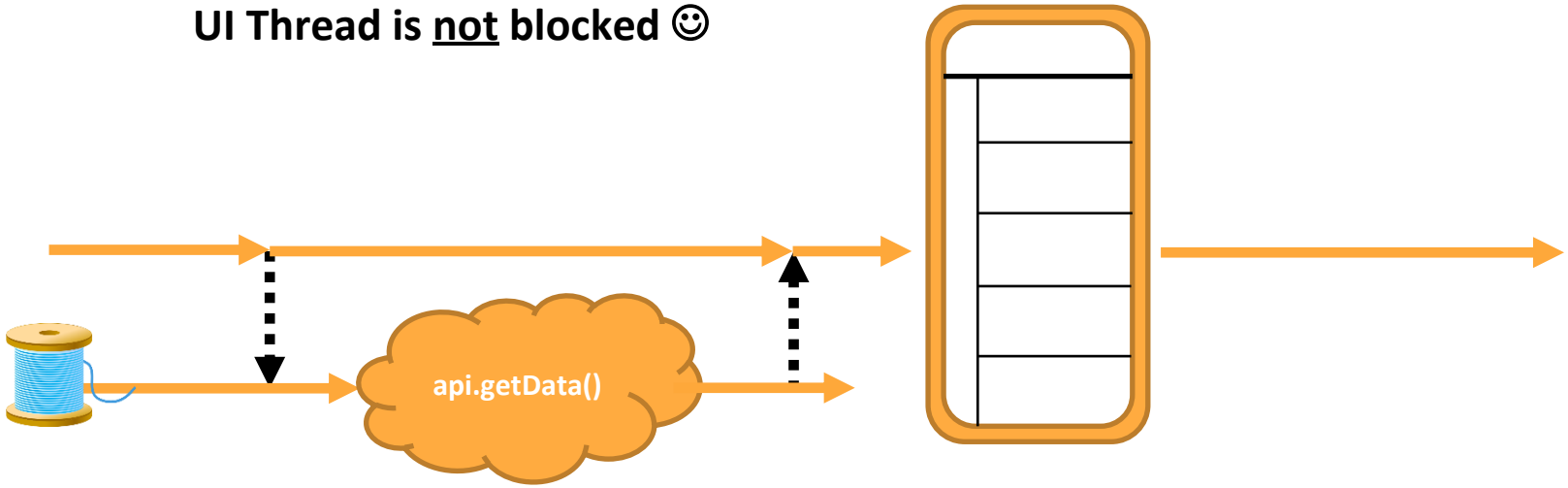


Long Running Task include:

- 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 separate thread

UI Thread is <u>not</u> blocked ☺




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

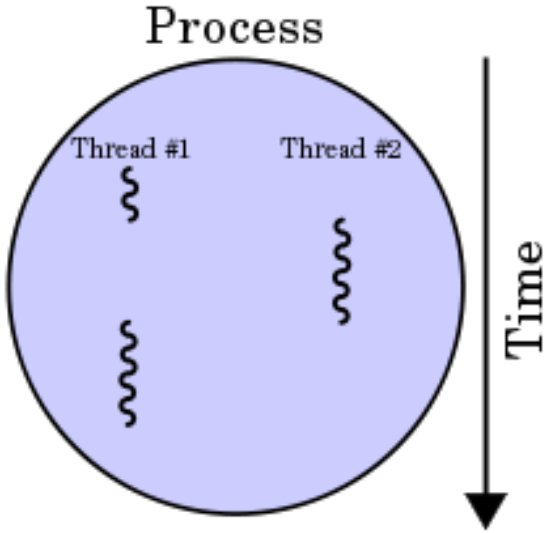
- Threads introduces many challenges related to data communication and synchronization between threads
- Plus UI can only be accessed from the main UI Thread

How to address problem of long-running task?

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

Solution 1: **Use multi-threading**

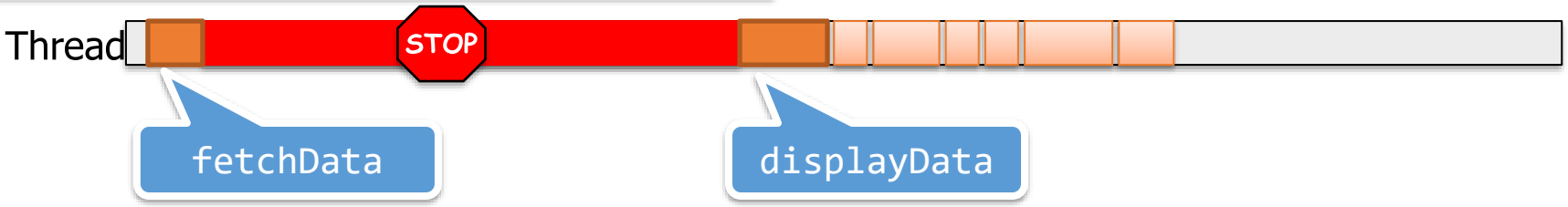
- A thread is the **unit of execution** within a process
  - When a process starts, it is assigned memory and resources
  - Each thread in the process shares that memory and resources



Synchronous vs. Asynchronous

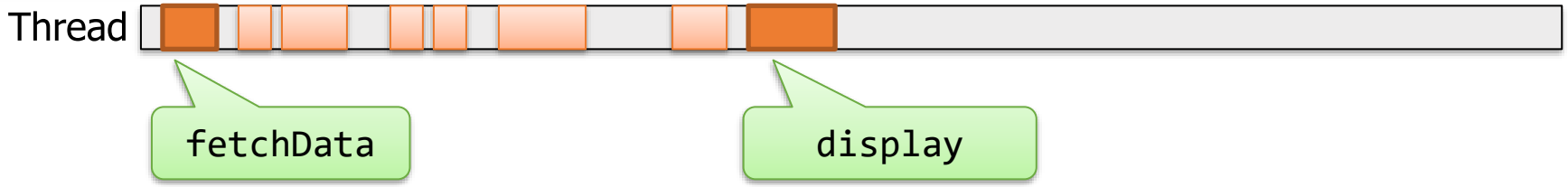
Synchronous → Wait for result before returning

```
val data = fetchData(...) // UI thread
display(data)  // UI thread
```




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

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



UI not blocked

Callback pattern

- By using callbacks, you can start long-running tasks on a background thread
- When the task completes, the callback is called to inform 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
        show(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

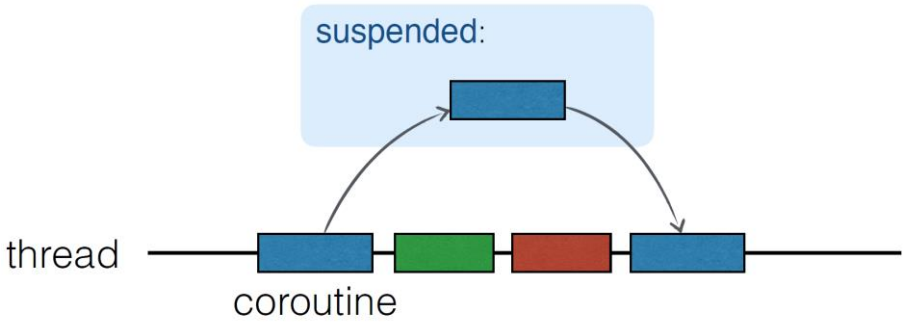
Thread Limitations

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

Better alternative are **Coroutines**

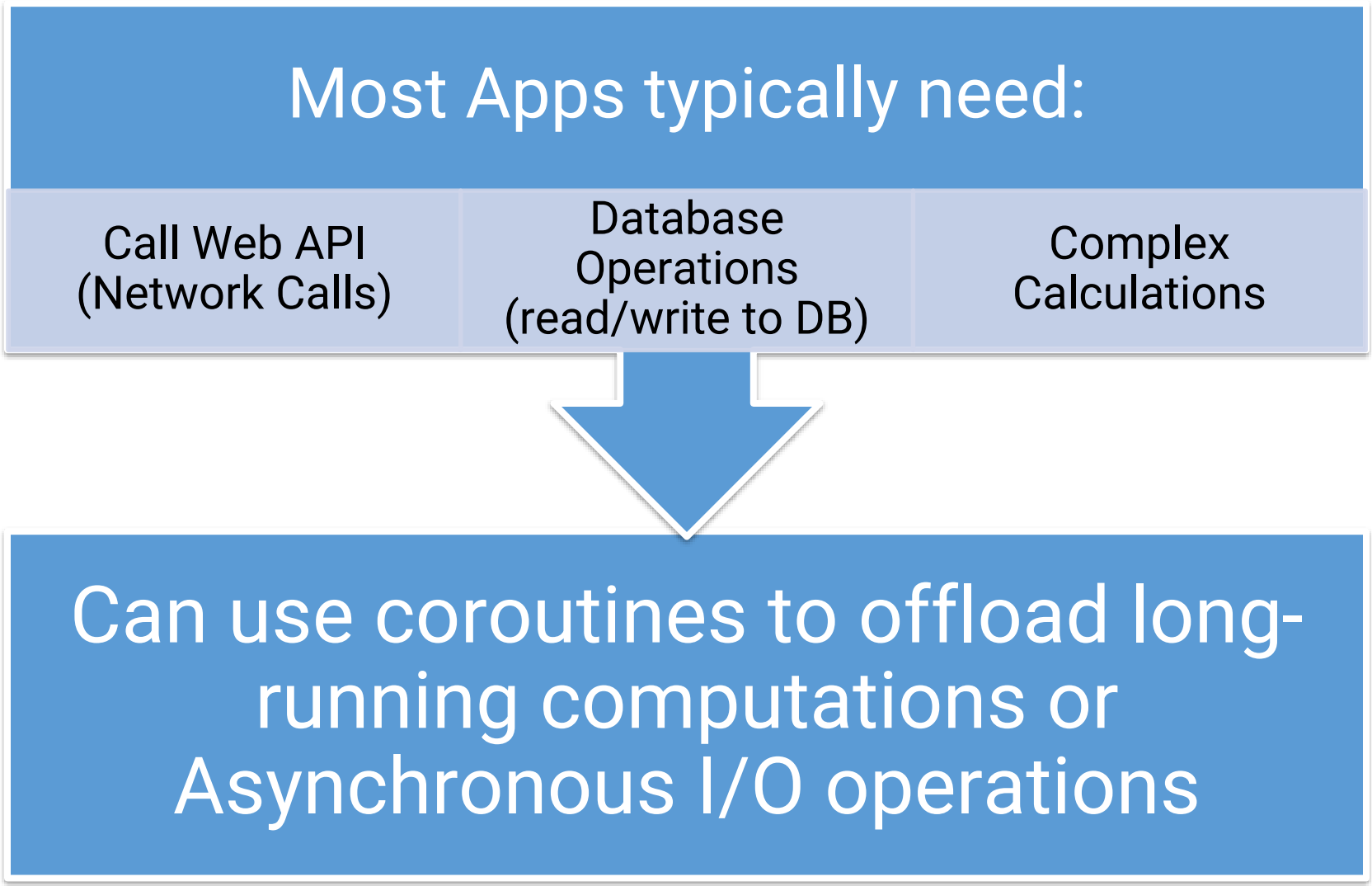
Coroutine = computation that can be suspended

Coroutines are like light-weight threads



Thread is not blocked!

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(): SlowResult { ... }
```

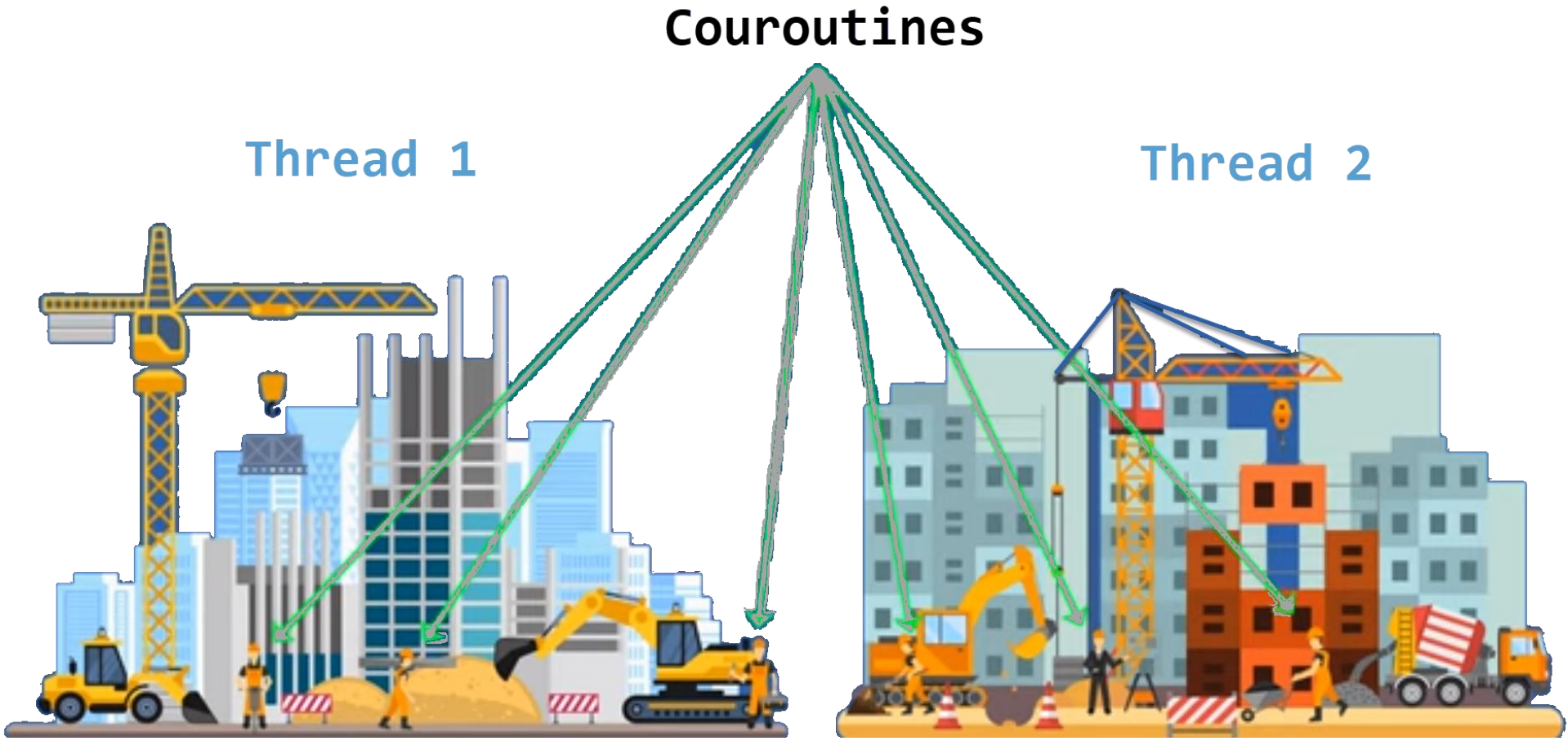
- Compared to callback-based code, coroutine code accomplishes the same result of unblocking the current thread with less code.

- 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
2. Executed within a thread
3. Coroutines are suspendable
   - Allow execution to be suspended and then resumed
4. They can switch their context
   - e.g., do a Web API call using the IO Thread then switch to the UI thread to update the UI
5. Replace callback-based code with sequential code to handle long-running tasks without blocking

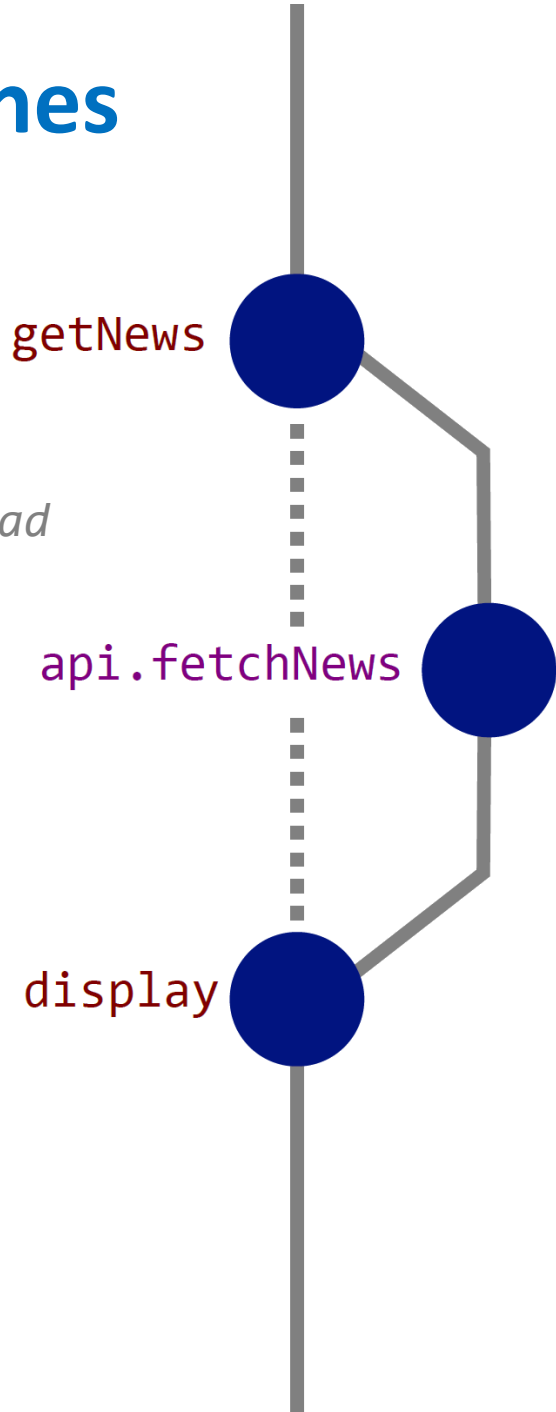
Thread vs. Coroutine



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

Async Programming with Coroutines

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



- Key benefit of Async Programming = ***Responsiveness***
  - **prevent blocking** the UI thread on long-running operations

Suspendable

Coroutines Programming Model

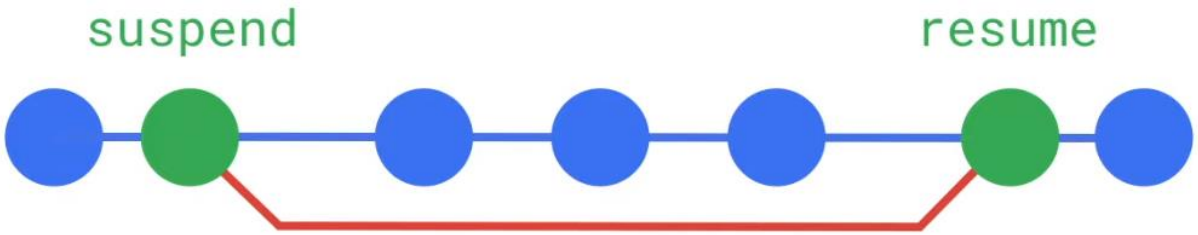
Back

Suspend function

- **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 like a normal function call, 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

Suspend function

- Suspend functions help make async code sequential



- A **suspend** function can only be called from another *suspend* function or a *coroutine* scope

Coroutine Builder

- Use **coroutine builders** to call a suspend function i.e., **create** and **start** a **coroutine**
- **runBlocking**
  - Bridge between regular and suspend functions

  (often an entry-point)
- **launch**
  - Fire and forget
- **async**
  - Expect result

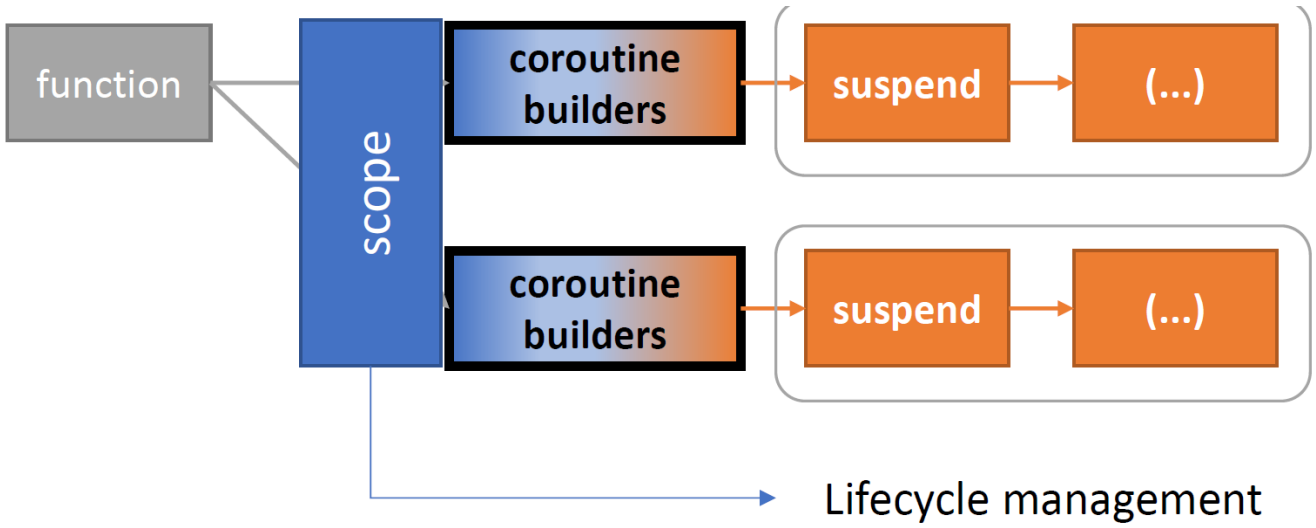
async-await

- 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 Scope

- Keep track of coroutines to allow the ability to cancel ongoing work
  - If the scope fails with an exception or is cancelled, all the children are cancelled, too
- A coroutine is **always** created in the context of a **scope**
  - A thing with a lifecycle that we want to associate with the coroutines
  - GlobalScope – app-level scope (rarely used)
  - `viewModelScope`, `lifecycleScope`



Structured concurrency

- The **lifespan** of a coroutine is constrained by a lifespan of the parent scope

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)

```
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
- Accessisble either via `lifecycle.coroutineScope` or `lifecycleOwner.lifecycleScope`
- 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)
        viewLifecycleOwner.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

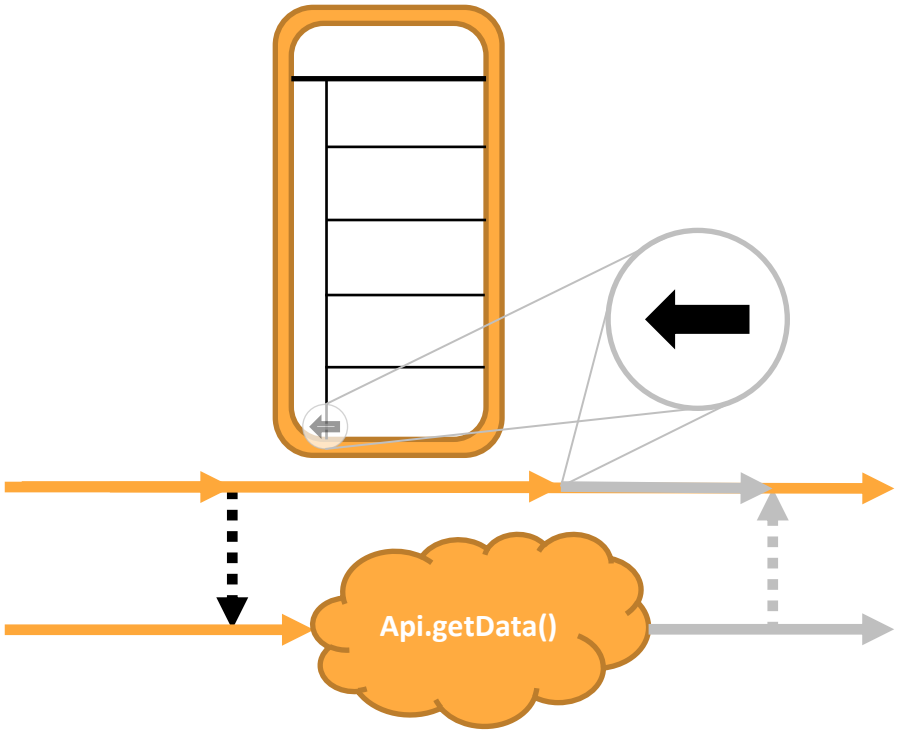

```
// Use the liveData builder function to call fetchUser()
// asynchronously and then use emit() to emit the result
val user: LiveData<User> = liveData {
    // fetchUser is a suspend function.
    val user = api.fetchUser(email)
    emit(user)
}
```

liveData with switchMap

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

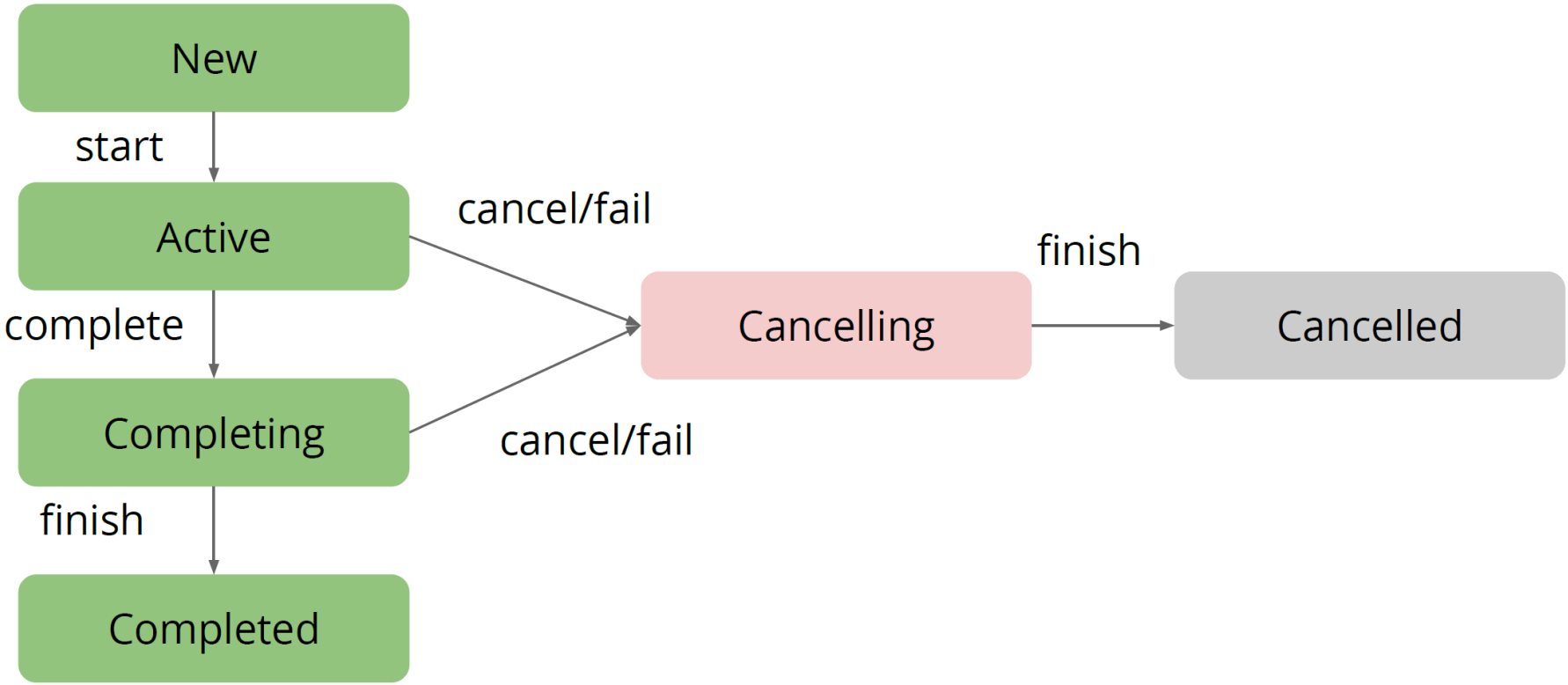
```
class MyViewModel: ViewModel() {
    private val userId: LiveData<String> = MutableLiveData()

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

Cancellable



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

Job Lifecycle

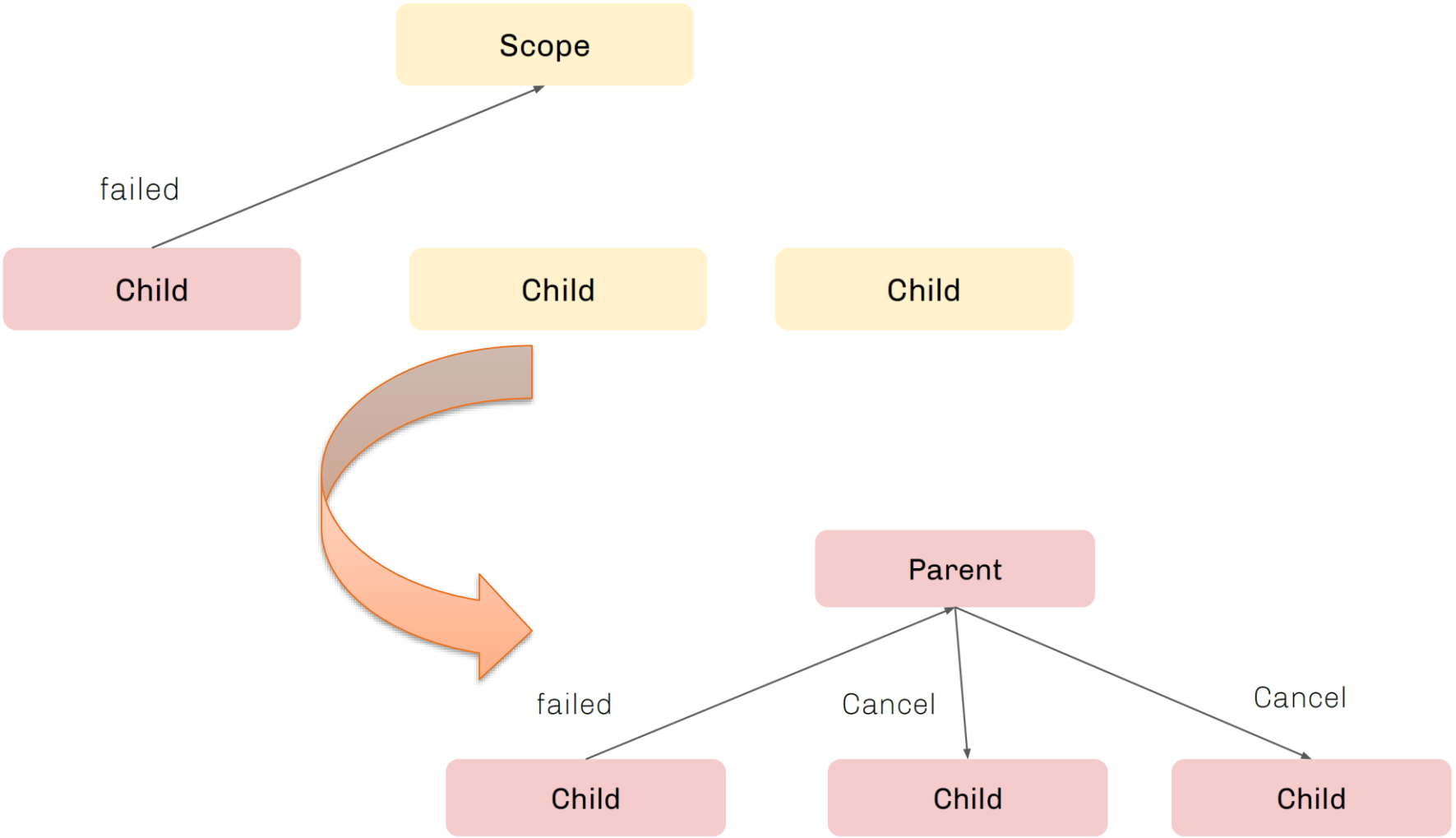


Cancellation by CoroutineScope

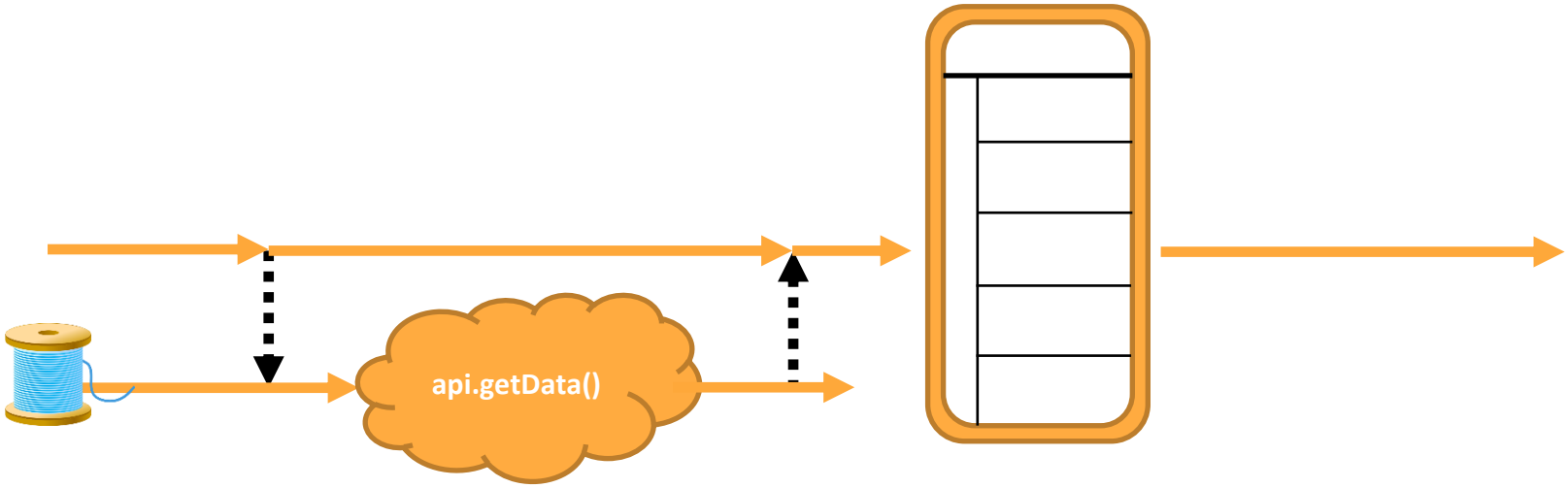
- **isActive**
  - Do an action before finishing the coroutine
- **ensureActive()**
  - Instantaneously stop work
- **yield()**
  - CPU heavy computation that may exhaust thread-pool

Cancellation due to an Exception

- The failure of a child cancels the scope & other children



Move data between Threads



Moving fetch data to background thread and result to UI thread:

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

Switch to Main Thread to update the UI thread

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

- Part 1: Coroutines, Part 2: Cancellation in coroutines, and Part 3: Exceptions in coroutines

- Coroutines codelab https://codelabs.developers.google.com/codelabs/kotlin-coroutines