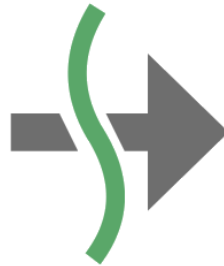


# CMPS 312

## Coroutines for Asynchronous Programming

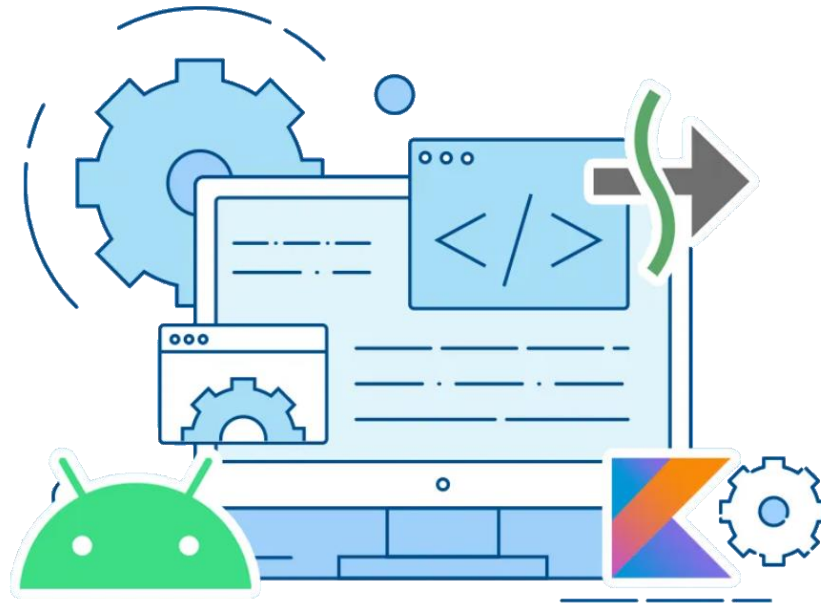


Dr. Abdelkarim Erradi  
CSE@QU

# Outline

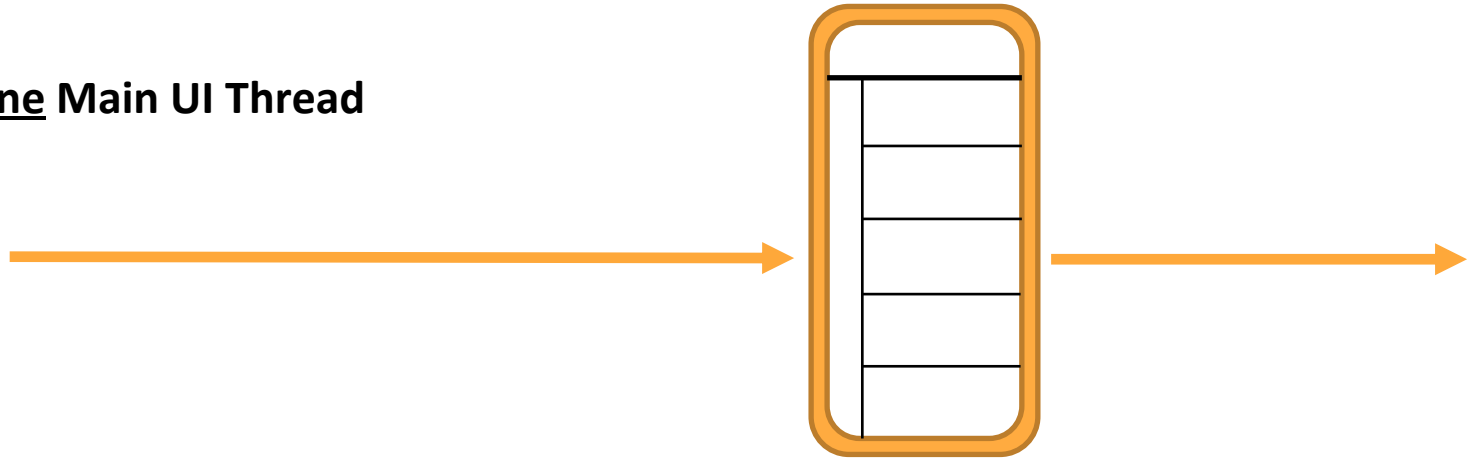
1. Coroutines Basics
2. Coroutines Programming Model
3. Flow

# Coroutines Basics



# User Interface Running on the Main Thread

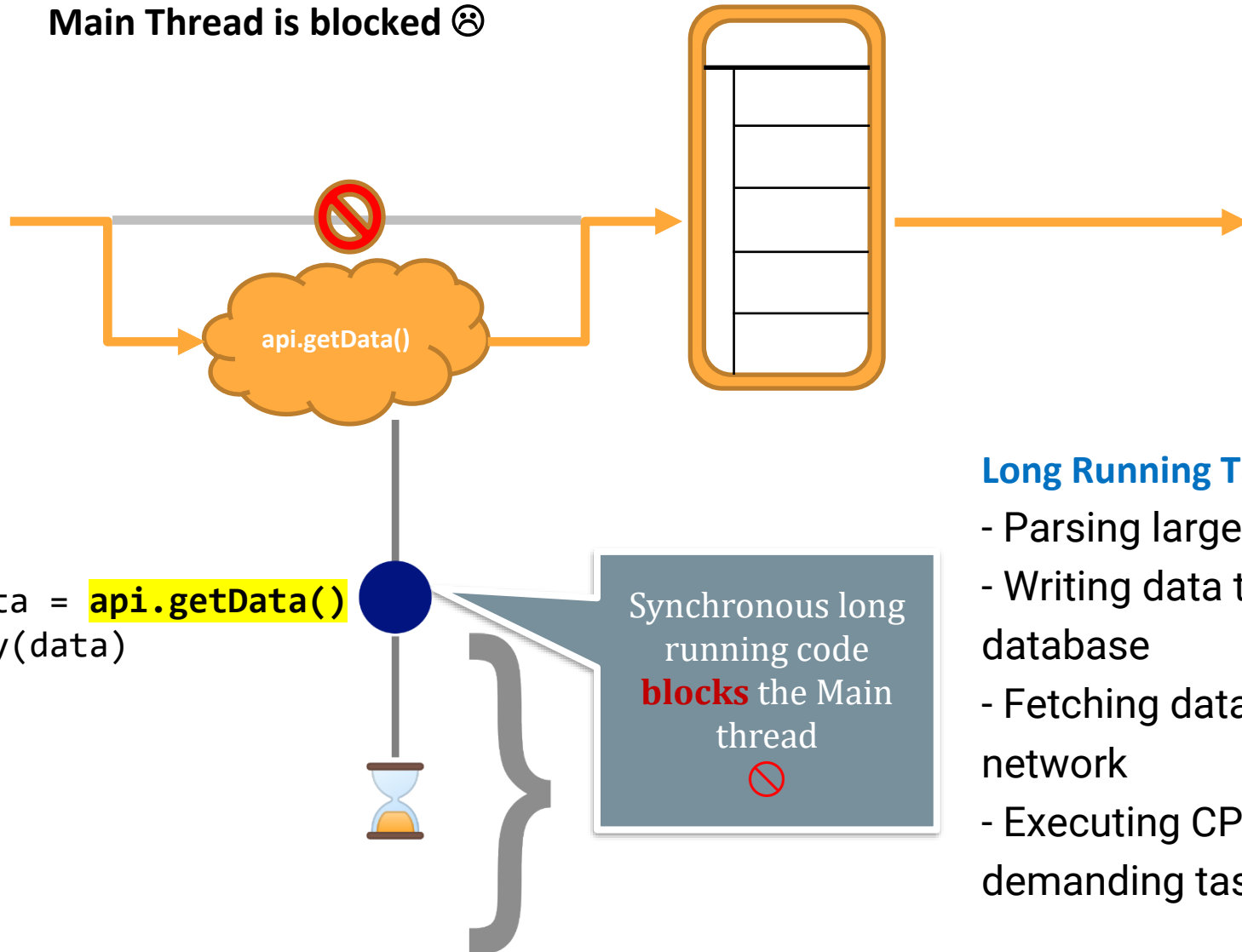
One Main UI Thread



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

Main Thread is blocked ☹️

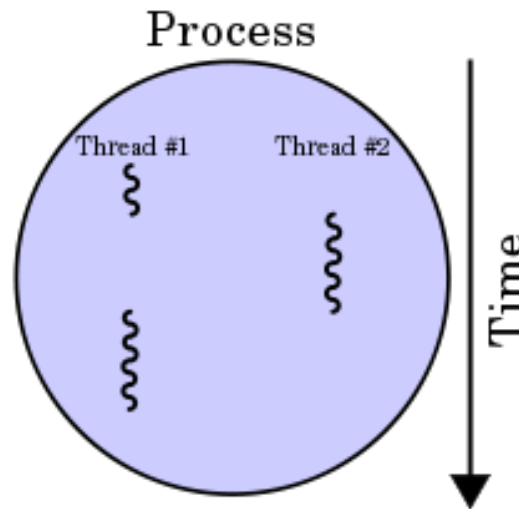


**Long Running Tasks include:**

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

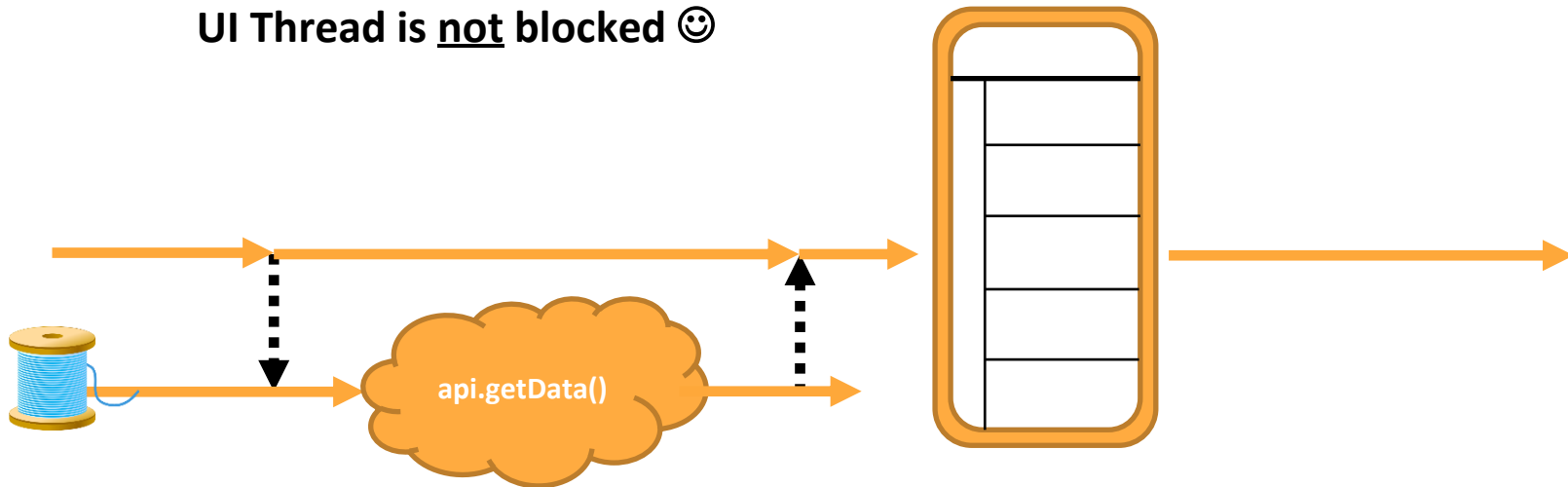
# How to address problem of long-running task?

- How to execute a long running tasks without blocking the Main thread?  
=> Solution 1: **Use multi-threading** 🧵 🧵 🧵
- A thread is the **unit of execution** within a process
  - It allows **concurrent** execution of tasks within an App



# Solution 1 – Run Long Running tasks on a background thread

UI Thread is not blocked 😊



```
thread {  
    val result = api.getData()  
}
```

- UI can only be accessed from the Main thread
- How to transfer the result from the background thread to the main thread?

# How to transfer the result from the background thread to the main thread?

- 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




## Limitations:

- Nested callbacks can become difficult to understand (aka **Callback Hell**)
- Difficult to cancel background tasks
- Difficult to run tasks in parallel
- Difficult to handle exceptions



# Callback Example

```
fun main() {  
    // Call the function and pass callback function  
    getUserOrders("sponge", "bob") { orders ->  
        orders.forEach { println(it) }  
    }  
}  
  
fun getUserOrders(username: String, password: String,  
    callback: (List<Order>) -> Unit) {  
    login(username, password) { user ->  
        fetchOrders(user.userId) { orders ->  
            // When the result is ready, pass it to main using the callback  
            callback(orders)  
        }  
    }  
}
```

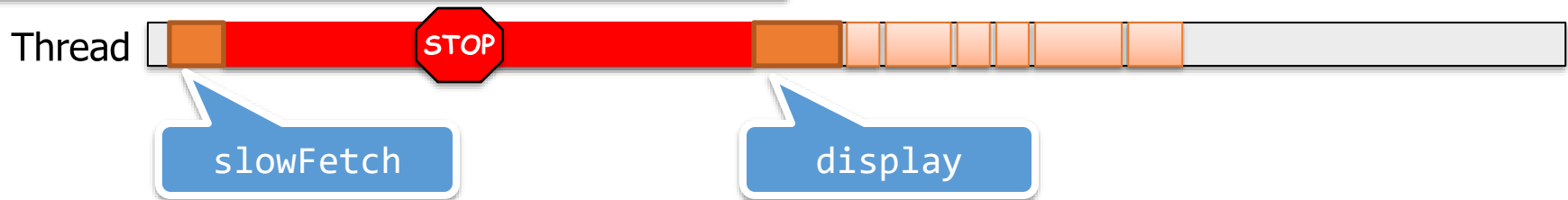


**Callback hell** = Nested callback functions are difficult to understand

# Synchronous vs. Asynchronous Functions

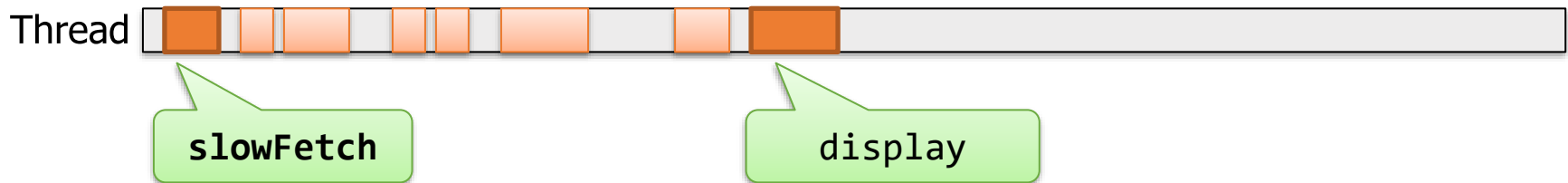
Synchronous → Wait for result before returning

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



```
// Slow request with callbacks  
fun makeNetworkRequest(display: (Result) -> Unit) {  
    // 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 background thread, then update UI with the result



**UI not blocked**

# Thread Limitations



**Threads** are **costly** (occupy 1-2 mb)

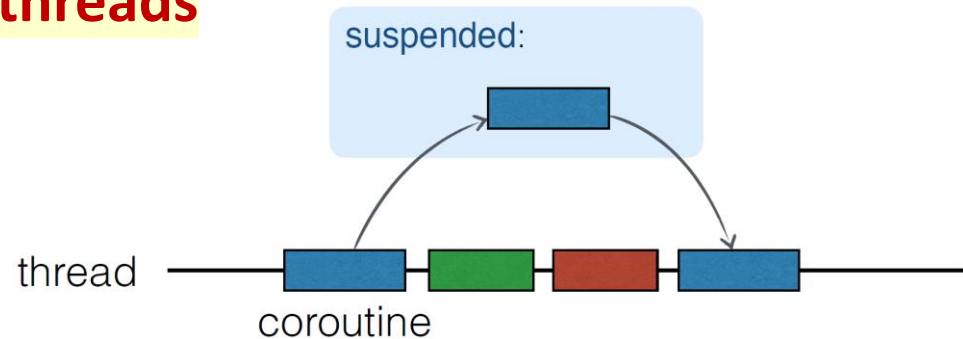
- Some **threads** are **special** (e.g. Main UI thread) and should not be blocked



Better alternative are **Coroutines**

Coroutines are like **light-weight threads**

Coroutine = computation that can be **suspended/resumed**



Thread is not blocked!

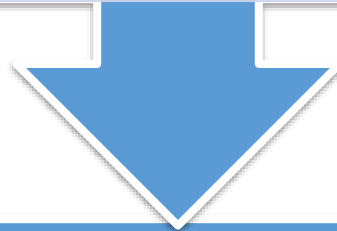
# Why Coroutines?

Most mobile apps typically need:

Call Web API  
(Network Calls)

Database  
Operations  
(read/write to DB)

Complex  
Calculations



Can use coroutines to offload long-running computations or  
Asynchronous I/O operations

# What distinguishes Coroutines from Threads? 🤔



1. Coroutines are like **light-weight** threads
  - Multiple coroutines can run within a thread
2. Easier **cancellation** of a long running coroutine
3. Easier **exception handling**
4. Easier to **run coroutines in parallel** to improve the app performance
5. Easier to **switch the coroutine execution between threads**
  - e.g., do a Network call using the IO Thread then switch to the Main thread to update the UI
6. Easier **asynchronous** programming
  - Replace callback-based code with sequential code to handle asynchronous long-running tasks without blocking

# Thread vs. Coroutine

## Couroutines

Thread 1

Thread 2



# Async Programming with Coroutines



getNews

```
newsBtn.setOnClickListener { // UI thread
    val news = getNews()
    display(news)
}
```

api.fetchNews

```
suspend fun getNews() = withContext(Dispatchers.IO) {
    ➡ return api.fetchNews() // IO thread
}
```

display

Key benefit of Async Programming = **Responsiveness**

**prevent blocking** the UI thread on long-running operations

# Callback vs. Coroutine

- Compared to callback-based code, coroutine code accomplishes the same result of unblocking the main 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

```
fun getUserOrders(username: String, password: String, callback: (List<Order>) -> Unit) {  
    login(username, password) { user ->  
        fetchOrders(user.userId) { orders ->  
            // When the result is ready, pass it to main() using the callback  
            callback(orders)  
        }  
    }  
}
```

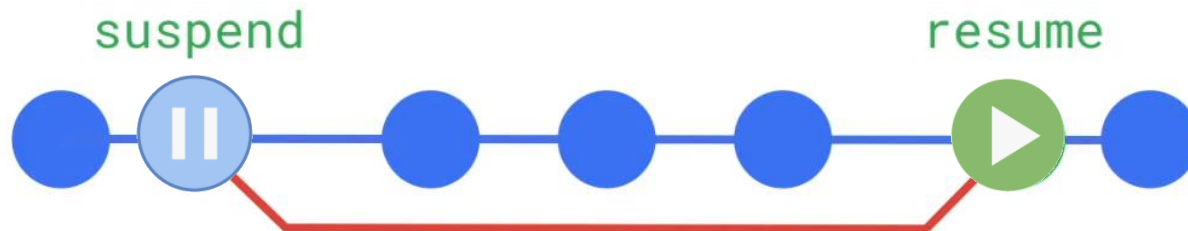


```
suspend fun getUserOrders(username: String, password: String) =  
    withContext(Dispatchers.IO) {  
        val user = login(username, password)  
        val orders = fetchOrders(user.userId)  
        return@withContext orders  
    }
```





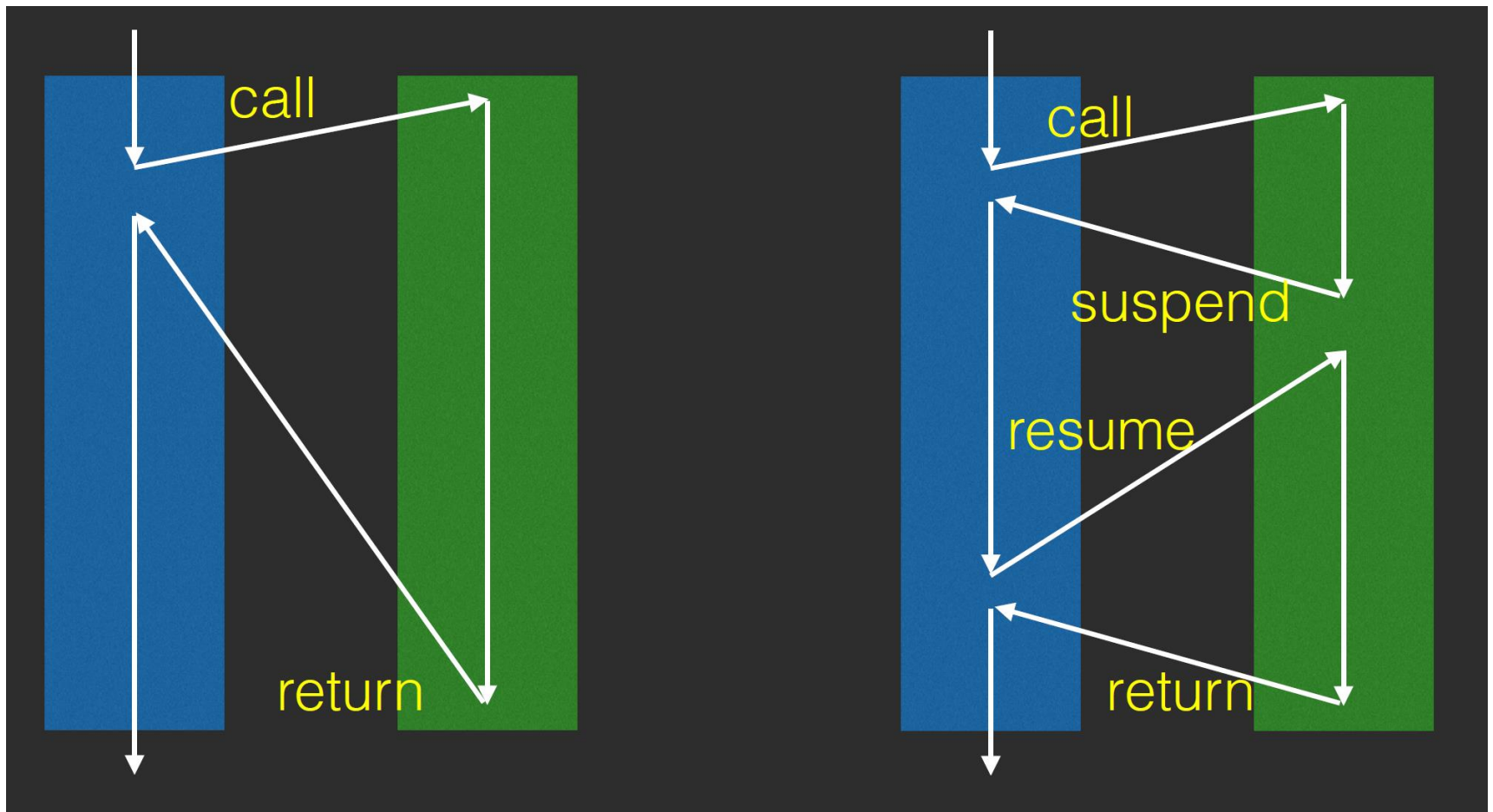
# Coroutines Programming Model



# 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



Suspend function can **suspend** at some points and later **resume** execution (possibly on another thread) when the return value is ready

# To launch a Coroutine you need a Coroutine Scope

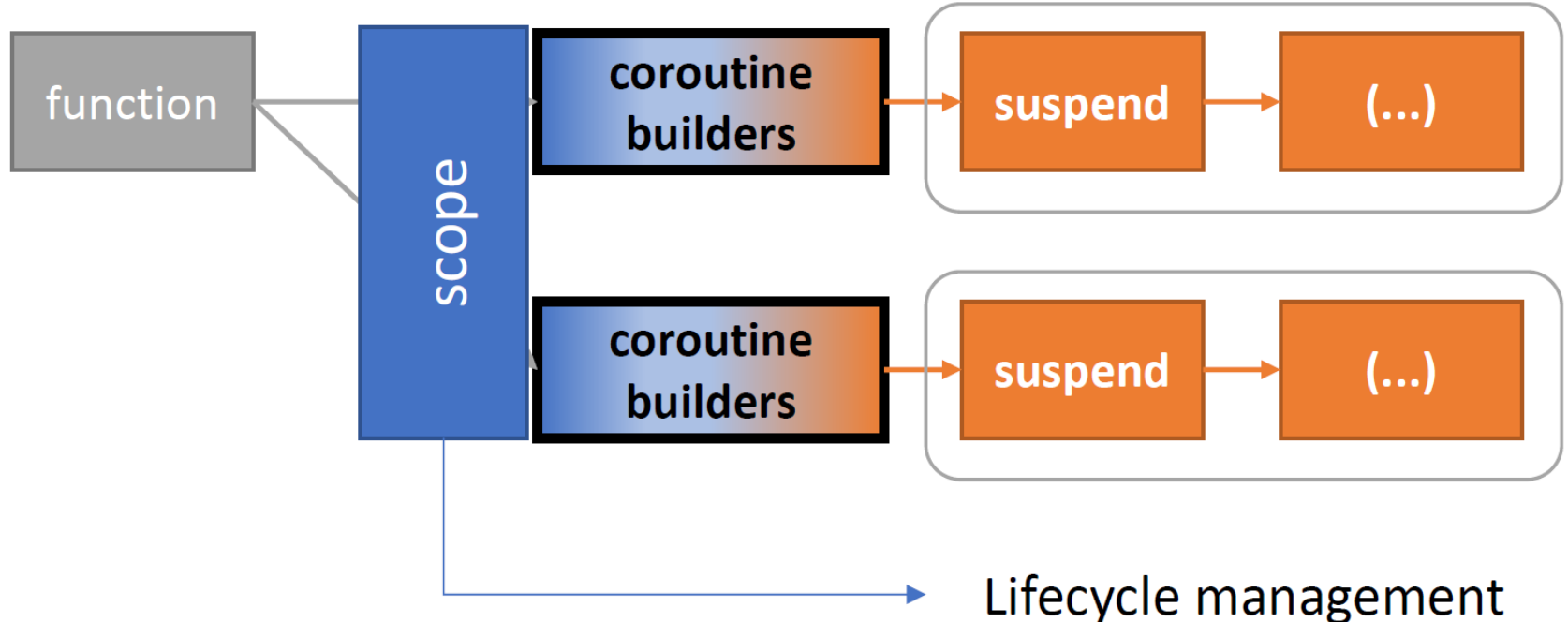
- A suspend function must be called in a coroutine
- A **Coroutine Scope** is required to create and start a coroutine using the scope's **launch** or **async** methods
- Coroutine Scope keeps track of child coroutines to allow the ability to cancel them and to handle exceptions
- Can be created as an instance of *CoroutineScope*

```
val coroutineScope = CoroutineScope(Dispatchers.IO)  
coroutineScope.launch { }
```

- 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 enables Cancellation

- A coroutine is **always** created in the context of a **scope**. This allows **Structured Concurrency** to:
  - Keep track of coroutines
  - Ability to cancel them
  - Is notified of failures (scope can cancel child coroutines if one of them fails)



# Important properties of Structured Concurrency



- Every Coroutine needs to be **started in a Coroutine Scope**
- Coroutines started in the same scope form a hierarchy (scope is the parent and coroutines are children)
- A parent job **won't complete, until all its children have completed**
- **Cancelling a parent will cancel all children**
  - Cancelling a child won't cancel the parent or siblings
- If a **child coroutine fails**, the exception is propagated upwards and **all the incomplete siblings are cancelled** (unless if a **supervisorScope** is used)



# *viewModelScope*

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

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

# *LifecycleScope*

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

```
class MyFragment: Fragment() {  
    override fun onCreateView(view: View, savedInstanceState: Bundle?) {  
        super.onCreateView(view, savedInstanceState)  
        lifecycleScope.launch {  
            // Incomplete coroutines 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.fetchUserById(id))  
        }  
    }  
}
```

# Coroutine builder functions

A coroutine scope offers two builder functions to **create** and **start** a coroutine

- **Launch** Fire and forget

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

- **Async** Returns a value

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

# Launch vs. Async Coroutine builder functions

- **Launch** - Launches a new coroutine and returns a **job** which can then be used to **cancel** the coroutine
- **Async** — Launches a new coroutine and returns its future result (of type **Deferred**)

```
val deferred = async { viewModel.getStockQuote(company) }
```

- Can use `deferred.await()` to suspend until the result is ready
- Or call `deferred.cancel()` to cancel the coroutine

## Launch

Creates a new coroutine

Fire and forget

Executed in a scope

## Async

Creates a new coroutine

Returns a value

Executed in a scope

# Parallel Execution of Coroutines

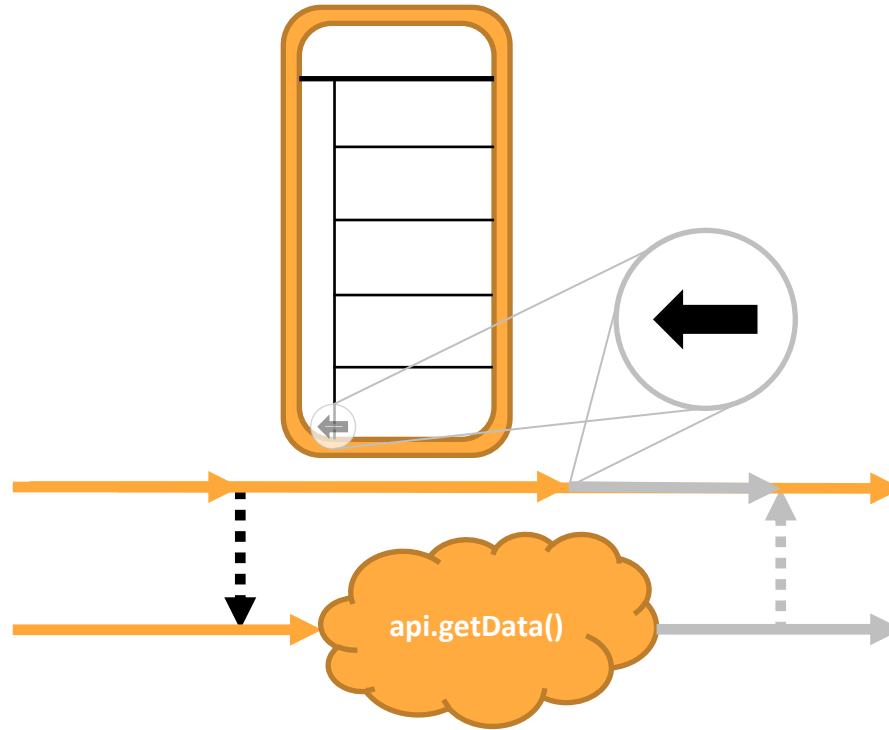
- Coroutines can be **executed in parallel** using **Async** or **Launch**
  - Parallelism is about doing lots of things simultaneously
- Async can await for the results (i.e. suspend until results are ready)

```
val deferred = async { getStockQuote("Apple") }  
val deferred2 = async { getStockQuote("Google") }
```

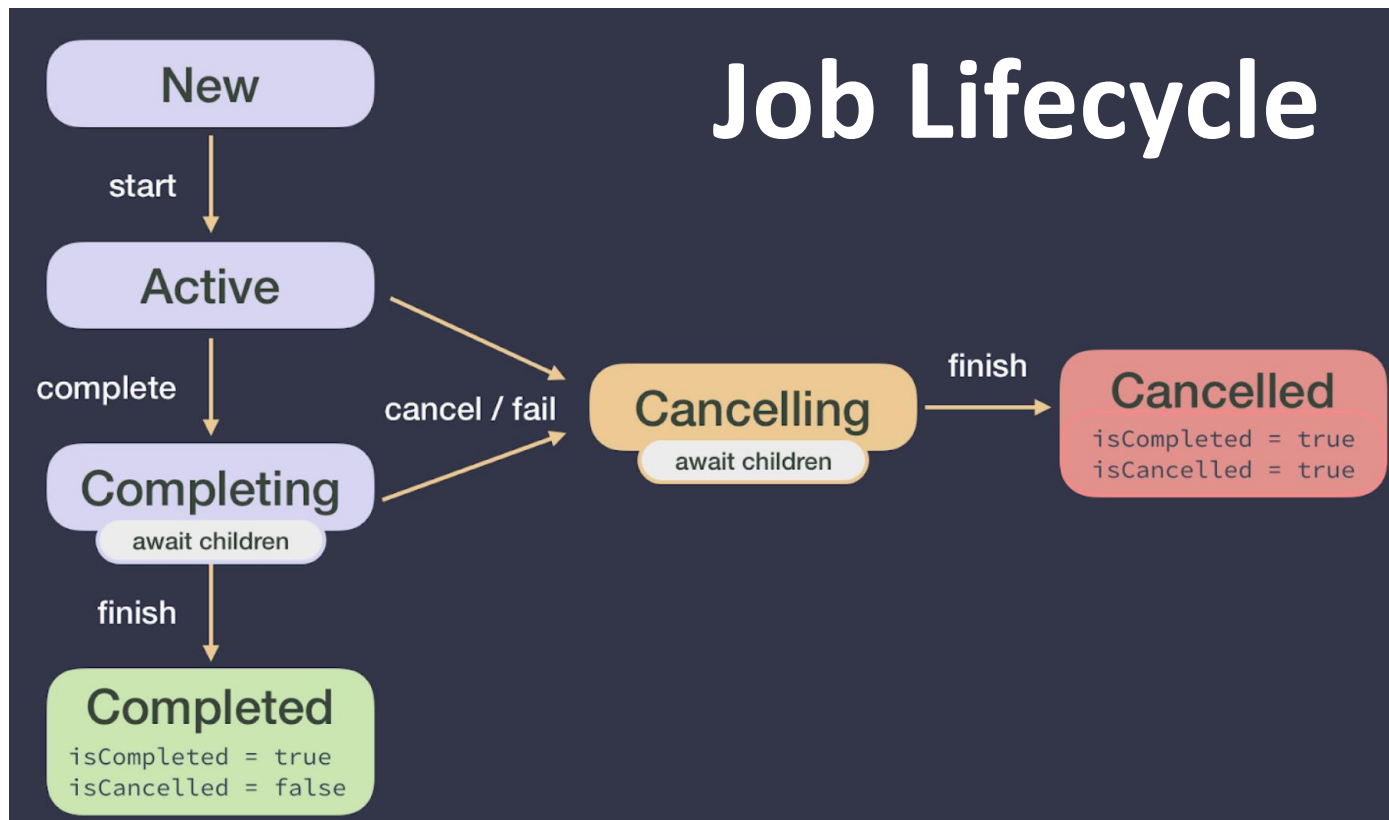
```
val quote = deferred.await()  
println(">> ${quote.name} (${quote.symbol}) = ${quote.price}")
```

```
val quote2 = deferred2.await()  
println(">> ${quote2.name} (${quote2.symbol}) = ${quote2.price}")
```

# Coroutine Cancellation



- 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()`



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

# Coroutine Cancellation

*// Create a coroutineScope and run multiple jobs*

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

```
val job1 = scope.launch { ... }
```

```
val job2 = scope.launch { ... }
```

*// Cancelling the scope cancels its children*

```
scope.cancel()
```

*// Or you can cancel a particular job*

*// First coroutine will be cancelled and the other*

*// one won't be affected*

```
job1.cancel()
```

```
val JOB_TIMEOUT = 5000L
```

*// Cancel the job after 5 seconds timeout*

*// job will be null if the job is cancelled*

```
val job = withTimeoutOrNull(JOB_TIMEOUT) {  
    fibonacci().collect {  
        print("$it, ")  
    }  
}
```



# Exception Handling

*/\* By default, **if one child failed the whole job is cancelled**  
and all incomplete sibling jobs are cancelled.*

*Unless supervisorScope is used (see example 12) \*/*

```
val exceptionHandler = CoroutineExceptionHandler { context, exception ->
    println("Exception thrown somewhere within parent or child: $exception.")
}
val job = GlobalScope.launch(exceptionHandler) {
    val deferred1 = async() { getStockQuote("Tesla") }
    try {
        val quote1 = deferred1.await()
    } catch (e: Exception) {
        println("Request failed : $e.")
    }
    val deferred2 = async() { getStockQuote("Aple") }
    try {
        val quote2 = deferred2.await()
    } catch (e: Exception) {
        println("Request failed : $e.")
    }
    val deferred3 = async() { getStockQuote("Google") }
    try {
        val quote3 = deferred3.await()
    } catch (e: Exception) {
        println("Request failed : $e.")
    }
}
```

# Exception Handling with **supervisorScope**

*/\* Because the **supervisorScope** is used. If one child failed the whole job is NOT cancelled \*/*

```
val exceptionHandler = CoroutineExceptionHandler { context, exception ->
    println("Exception thrown somewhere within parent or child: $exception.")
}
val job = GlobalScope.launch(exceptionHandler) {
```

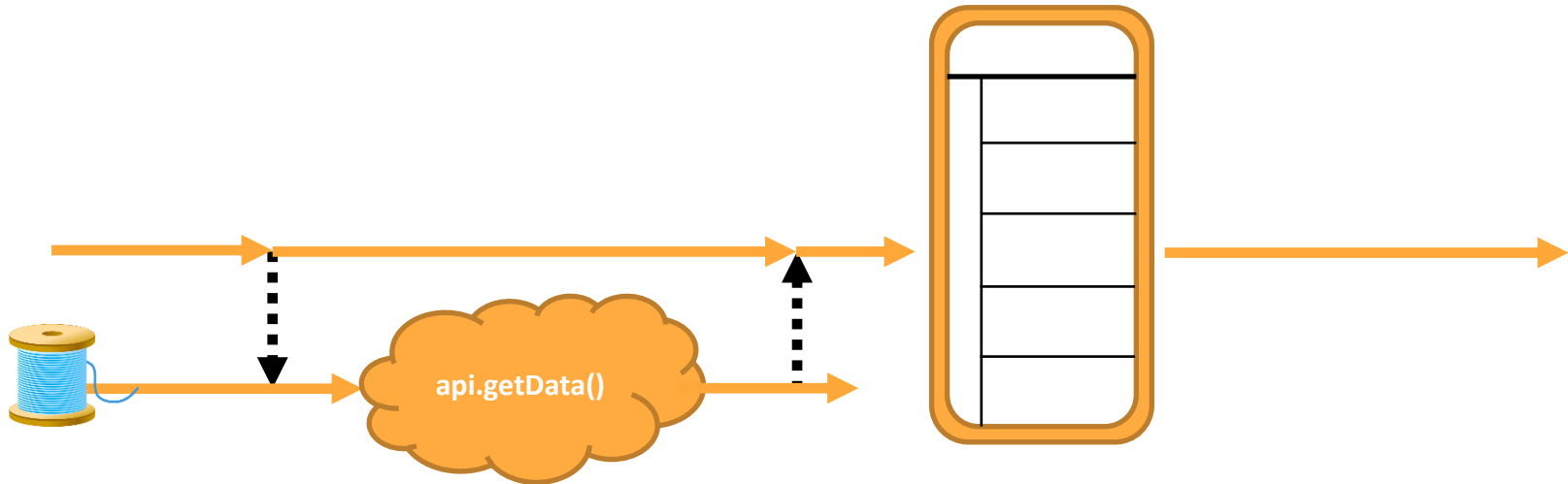
```
    supervisorScope {
```

```
        val deferred1 = async() { getStockQuote("Tesla") }
        try {
            val quote1 = deferred1.await()
        } catch (e: Exception) {
            println("Request failed : $e.")
        }
        val deferred2 = async() { getStockQuote("Aple") }
        try {
            val quote2 = deferred2.await()
        } catch (e: Exception) {
            println("Request failed : $e.")
        }
        val deferred3 = async() { getStockQuote("Google") }
        try {
            val quote3 = deferred3.await()
        } catch (e: Exception) {
            println("Request failed : $e.")
        }
    }
```

```
}
```

```
}
```

# Switch between threads



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

```
lifecycleScope.launch(Dispatchers.IO) {  
    -> val result = fibonacci(1000)  
    -> withContext(Dispatchers.Main) {  
        resultTv.text = result.toString()  
    }  
}
```

Switch to Main  
Thread to update  
the UI

# Switch 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 for CPU-intensive tasks
  - **Dispatchers.Main**: Used for updating the UI

# Flow



One-shot /  
operation

Observers

Multi-shot  
operations



# 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()
```



# Flow from Repeat API Call

- Create a Flow for repeated periodic API calls

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

# Flow Operators

- Flow has operators similar to collections such as map, filter and reduce

```
(1..5).asFlow()  
    .filter { it % 2 == 0 }  
    .map { it * it }  
    .collect { println(it.toString()) }
```

```
val result =(1..5).asFlow()  
                .reduce { a, b -> a + b }  
println("result: $result")
```

# Resources

- Kotlin coroutines
  - <https://kotlinlang.org/docs/reference/coroutines-overview.html>
  - <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>