Lab 05 - Android Corutine

Software Studio - 2022 Spring

What problems do coroutines solve?

1. Long running tasks: tasks that take too long to block the main thread.

2. Main-safety: ensures that any suspend function can be called from the main thread.

Long Running tasks - tasks that take far too long for your app to stop and wait for them!

1. Fetching a webpage
2. Interacting with an API both involve making a network request.
3. Reading from a database.
4. ....

Main-safety

We consider a function `main-safe` when it doesn't block UI updates on the main thread.

On Android, every app has a main thread that is in charge of handling UI (like drawing views) and coordinating user interactions.

Any long running task should be done without blocking the main thread, so your app doesn't display what's called "jank," like frozen animations, or respond slowly to touch events.

In order to perform a network request off the main thread, a common pattern is callbacks.

```
class ViewModel: ViewModel() {
    fun fetchDocs() {
        get("developer.android.com") { result ->
            show(result)
        }
    }
}
```

`get` is called from the main thread, it will use another thread to perform the network request.

Once the result is returned from the network request, `callback` will be called from the main thread.

Coroutines are a way to simplify the code used to manage long running tasks like fetchDocs.

```
// Dispatchers.Main
suspend fun fetchDocs() {
    // Dispatchers.Main
    val result = get("developer.android.com")
    // Dispatchers.Main
    show(result)
}

// look at this in the next section
suspend fun get(url: String) = withContext(Dispatchers.IO){/*...*/}
```

Doesn’t this code block the main thread?

How does it return a result from get without waiting for the network request and blocking?

Coroutines build upon regular functions by adding two new operations. In addition to invoke (or call) and return, coroutines add suspend and resume.

- **suspend** — pause the execution of the current coroutine, saving all local variables
- **resume** — continue a suspended coroutine from the place it was paused

You can only call suspend functions from other suspend functions, or by using a coroutine builder like launch to start a new coroutine.

`Suspend and resume work together to replace callbacks.`


suspend fun fetchDocs() {
val docs = get("…")
show(docs)
}
suspend
resume
Main Thread
[stack]

`get` will suspend the coroutine before it starts the network request.

Then, when the network request completes, instead of calling a callback to notify the main thread, it can simply resume the coroutine it suspended.

When all of the coroutines on the main thread are suspended, the main thread is free to do other work.

Main-safety with coroutines

In Kotlin coroutines, well written suspend functions are always safe to call from the main thread. No matter what they do, they should always allow any thread to call them.

There’s a lot of things that are too slow to happen in the main thread.

1. Network requests
2. Parsing JSON
3. Reading or writing from the database
4. Even just iterating over large lists
5. ...

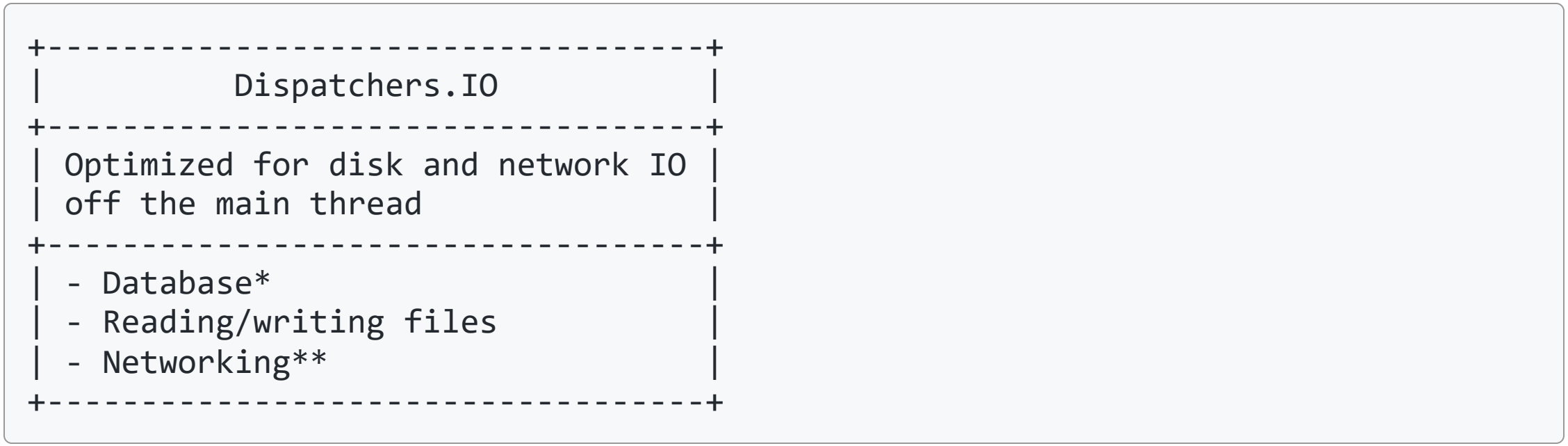
Using `suspend` doesn't tell Kotlin to run a function on a background thread. It's worth saying clearly and often that coroutines will run on the main thread.

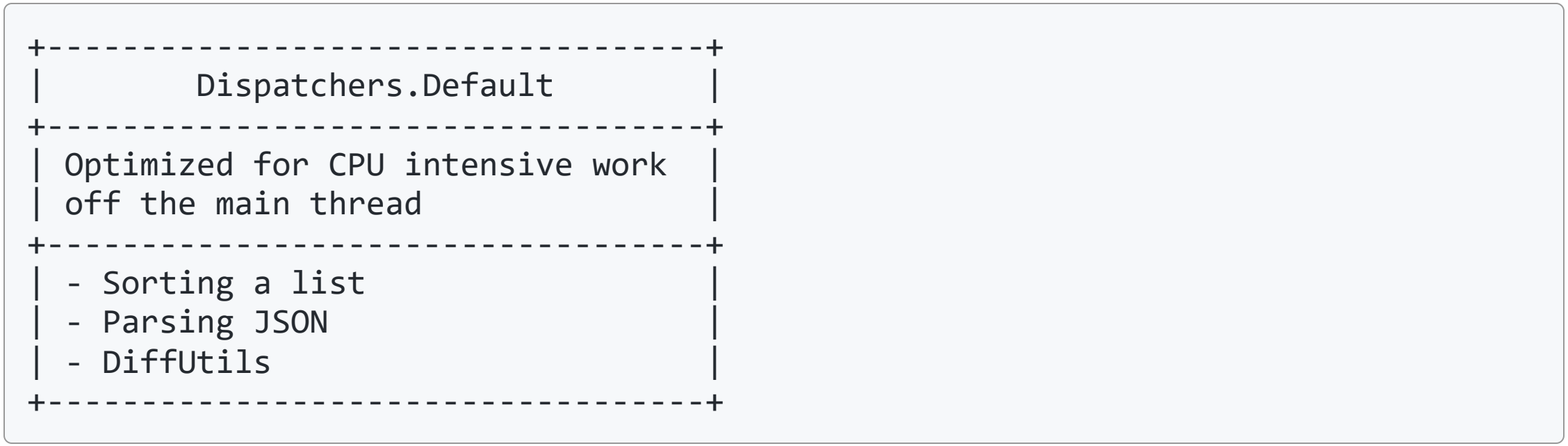
To make a function that does work that's too slow for the main thread main-safe, you can tell Kotlin coroutines to perform work on either the `Default` or `IO` dispatcher.

In Kotlin, all coroutines must run in a **dispatcher** — even when they're running on the main thread. Coroutines can **suspend** themselves, and the **dispatcher** is the thing that knows how to **resume** them.

To specify where the coroutines should run, Kotlin provides three Dispatchers you can use for thread dispatch.

```
+-----------------------------------+
|         Dispatchers.Main          |
+-----------------------------------+
| Main thread on Android, interact  |
| with the UI and perform light     |
| work                              |
+-----------------------------------+
| - Calling suspend functions       |
| - Call UI functions               |
| - Updating LiveData               |
+-----------------------------------+
```


Dispatchers.IO
Optimized for disk and network IO
off the main thread
- Database*
- Reading/writing files
- Networking**


Dispatchers.Default
Optimized for CPU intensive work
off the main thread
- Sorting a list
- Parsing JSON
- DiffUtils

To continue with the example above, let's use the dispatchers to define the `get` function.

```
// Dispatchers.Main
suspend fun fetchDocs() {
    // Dispatchers.Main
    val result = get("developer.android.com")
    // Dispatchers.Main
    show(result)
}
// Dispatchers.Main
suspend fun get(url: String) =
    // Dispatchers.Main
    withContext(Dispatchers.IO) {
        // Dispatchers.IO
        /* perform blocking network IO here */
    }
    // Dispatchers.Main
```

`withContext` lets you control what thread any line of code executes on without introducing a callback to return the result.

Because coroutines support `suspend` and `resume`, the coroutine on the main thread will be resumed with the result as soon as the withContext block is complete.

use `withContext` to make it safe to call from the main thread. This is the pattern that coroutines based libraries like `Retrofit` and `Room` follow.

Q: In `android-mars-photos`

`retrofitService.getPhotos()` is in the **main thread** or **background thread**?

```
viewModelScope.launch {
  try {
   val listResult = MarsApi.retrofitService.getPhotos()
   ...
  }
  ...
}
```

```
// Dispatchers.Main
viewModelScope.launch {
  // Dispatchers.Main
  try {
   val listResult = MarsApi.retrofitService.getPhotos()
   // Dispatchers.Main
   ...
  }
  ...
}
```

Networking libraries such as `Retrofit` and `Volley` manage their own threads and do not require explicit main-safety in your code when used with Kotlin coroutines.