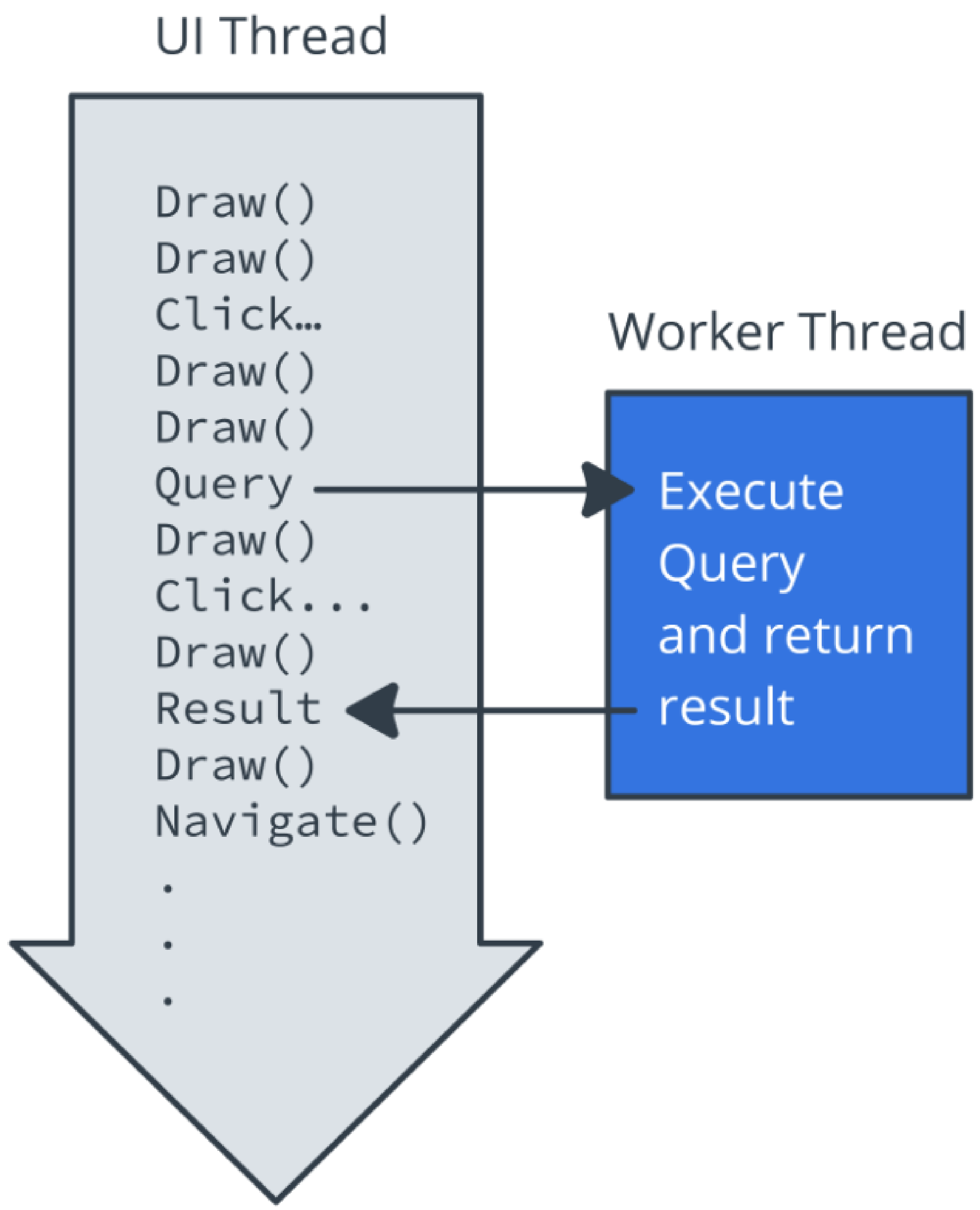
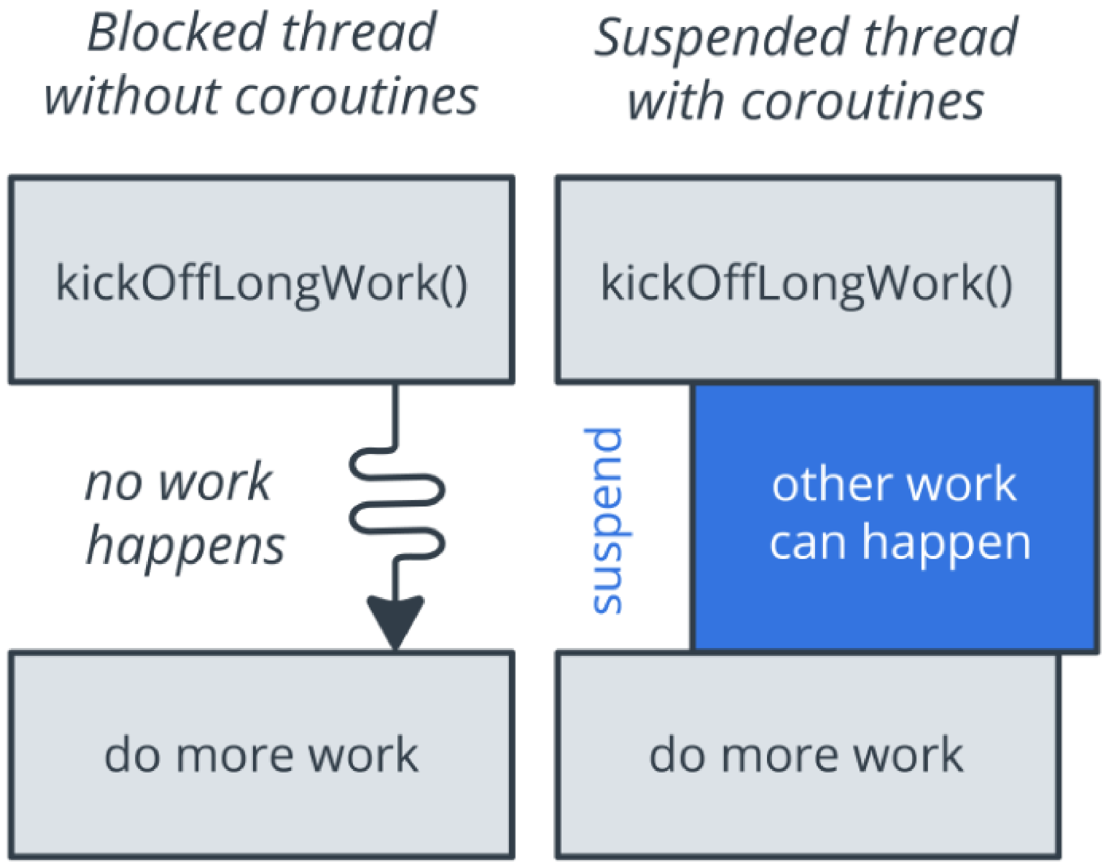
Background work with Coroutines

Threads


UI Thread
Draw()
Draw()
Click...
Draw()
Draw()
Query
Draw()
Click...
Draw()
Result
Draw()
Navigate()
.
.
.
Worker Thread
Execute Query and return result

Coroutines

- Coroutines are asynchronous and non-blocking.
- Coroutines use suspend functions to make asynchronous code sequential.



Couroutine Context

Coroutines always execute in some `CoroutineContext`: a set of various elements, mainly its `Job` and its `CoroutineDispatcher`

Scope

A coroutine's scope defines the context in which the coroutine runs.

- A scope combines information about a coroutine's `Job` and `CoroutineDispatcher`
- Scopes keep track of coroutines that are “in them”

➡️ actually just a wrapper around a `CoroutineContext`, can be seen as a "parent context"

ex: `GlobalScope`, `MainScope`, `viewModelScope`, `lifeCycleScope`

Job

Basically, a `Job` is anything that can be canceled

- Every coroutine has a `Job`, and you can use it to cancel the coroutine
- Jobs can be arranged into parent-child hierarchies
- Canceling a parent job immediately cancels all the job's children

```kotlin
fun main() {
    val job = GlobalScope.launch {
        // do something long
    }
    if (input == `^C`) job.cancel()
}
```

Dispatcher

The `CoroutineDispatcher` sends off coroutines to run on various threads

ex: `Dispatcher.Main` runs tasks on the main thread, `Dispatcher.IO` offloads blocking I/O tasks to a shared pool of threads

```
fun main() {
    GlobalScope.launch(Dispatchers.IO) {
        // do something long on IO thread
    }
}
```

Suspending

Suspend functions are only allowed to be called from a coroutine or another suspend function

```kotlin
suspend fun doSomethingLong() {
    // request server, DB, filesystem, ...
}

fun main() {
    doSomethingLong() // ⚠️ KO

    GlobalScope.launch {
        doSomethingLong() // ✅ OK
    }
}

suspend fun otherSuspendFunction() {
    doSomethingLong() // ✅ OK
}
```

Usage

```
class Repository {
   suspend fun getData() = withContext(Dispatchers.IO) {
       // execute long IO operation
   }
}

class MyViewModel: ViewModel() {
   init {
       viewModelScope.launch { // canceled when ViewModel is cleared
            repository.getData()
       }
   }
}

class MyFragment: Fragment {
    init {
        lifecycleScope.launch { /* canceled when fragment is destroyed */ }
        lifecycleScope.launch {
            whenStarted { /* starts when fragment starts */ }
            // the rest executes after the whenStarted block
        }
        lifecycleScope.launchWhenStarted { /* launches when fragment starts */ }
    }
}
```