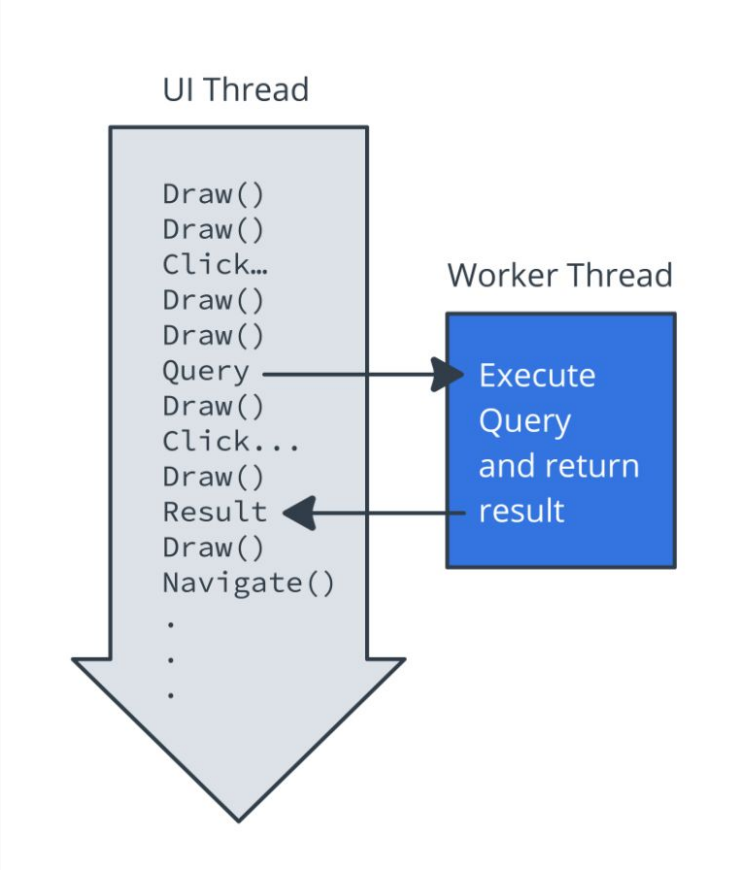
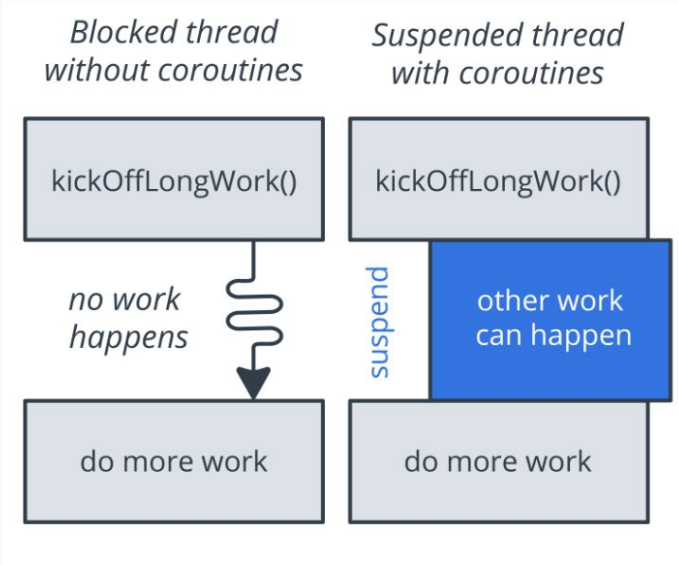
Coroutines


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

Coroutines have the following properties:

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



Using coroutines

- **Job**: Basically, a job is anything that can be canceled.
  - Every coroutine has a job, and you can use the job to cancel the coroutine.
  - Jobs can be arranged into parent-child hierarchies.
  - Canceling a parent job immediately cancels all the job's children

- **Dispatcher:** The dispatcher 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.

- **Scope:** A coroutine's *scope* defines the context in which the coroutine runs.
  - A scope combines information about a coroutine's job and dispatcher.
  - Scopes keep track of coroutines that are “in them”

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

    // scope that produces LiveData<>
    val user: LiveData<User> = liveData {
        val data = database.loadUser() // loadUser is a suspend function.
        emit(data)
    }

    // switchMap produces LiveData<> from a source LiveData<>
    private val userId: LiveData<String> = MutableLiveData()
    val user = userId.switchMap { id ->
        liveData(context = viewModelScope.coroutineContext + Dispatchers.IO) {
            emit(database.loadUserById(id))
        }
   }}
```

```
class UserDao: Dao {
    @Query("SELECT * FROM User WHERE id = :id")
    fun getUser(id: String): LiveData<User>
}

// emitSource sends the given LiveData to observers
class UserRepository {
    fun getUser(id: String) = liveData<User> {
        val disposable = emitSource(userDao.getUser(id).map { Result.loading(it) })
        try {
            val user = webservice.fetchUser(id)
            disposable.dispose() // Stop default emission
            userDao.insert(user) // update DB
            emitSource(userDao.getUser(id).map { Result.success(it) })
        } catch(exception: IOException) {
            emitSource(userDao.getUser(id).map { Result.error(exception, it) })
        }
    }}
```

```
class MyFragment: Fragment {
   init { // Notice that we can safely launch in the constructor of the Fragment.
       lifecycleScope.launch {
           whenStarted {
               // The block inside will run only when Lifecycle is at least STARTED.
               loadingView.visibility = View.VISIBLE
               val canAccess = withContext(Dispatchers.IO) { checkUserAccess() }
               loadingView.visibility = View.GONE
               if (canAccess == false) {
                   findNavController().popBackStack()
               } else {
                   showContent()
               }
           }
           // This line runs only after the whenStarted block above has completed.
       }

      lifecycleScope.launchWhenStarted {
         try {// Call some suspend functions.
         } finally { // This line might execute after Lifecycle is DESTROYED.
             if (lifecycle.state >= STARTED) {
                 // Here, since we've checked, it is safe to run any fragment transactions.
             }
         }}}
```