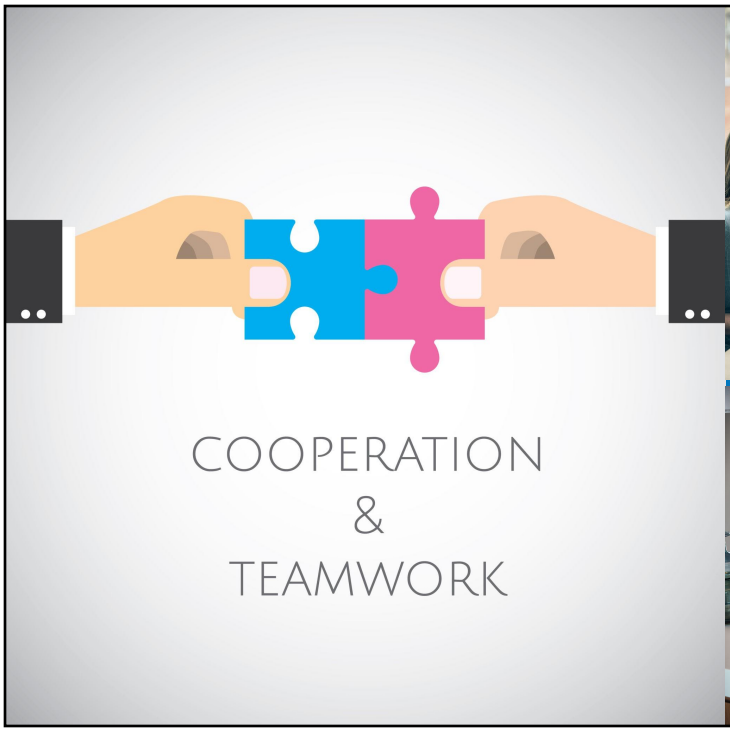
(Multi) Threading & Kotlin Coroutines



Objective:
User-Level Concurrency
(instead of *system-level*)

User Actions ⇒ UI Update

Goals

1. Keep UI highly responsive (require continuous update @ 60 fps)
2. **But also** allow the phone to perform other work (besides UI)
   a. Computations on data
   b. Fetch/Store data from/to remote database
   c. Fetch/Store data from/to local storage/filesystem

Solution: Multi-Threaded Concurrent Execution

- One dedicated thread for UI
- Other threads for performing other tasks

Topics

- Concurrency vs. Parallelism
- Concurrent Tasks
  - Concurrency control by Operating System
  - Concurrent Execution with (multiple) Threads
  - Concurrent Execution with (multiple) Coroutines
- Function calls
  - Subroutines vs. Coroutines
- Kotlin Coroutines
  - Suspending Functions
  - Dispatchers
- Prerequisites: (trailing) lambdas

General Dictionary

con·cur·rent:
run together side-by-side

Computing Dictionary (*warning: different interpretation*)

con·cur·rent:
run together one-by-one
(one at a time)

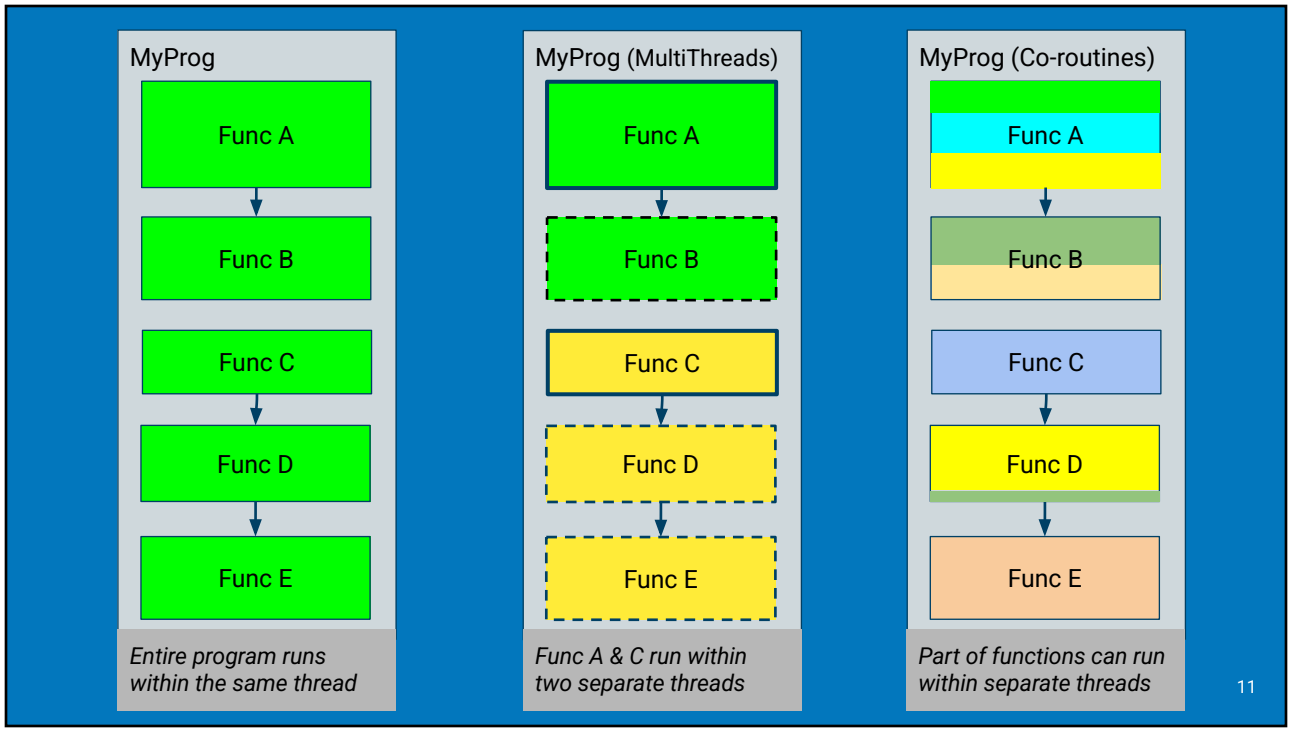
par·al·lel:
run together side-by-side

Implementation of Concurrency (*Progressive Improvements*)

- Operating System
  - Slice CPU time among multiple processes/users
  - A **single unit of work** within a program/process
  - Processes run in stop-N-go fashion (*suspended-then-resumed*)
- Multi-Threading
  - Allow users (developers) to define **multiple units of work** (within one program)
  - Each unit of work runs in stop-N-go fashion
- Co-routines (suspendable functions)
  - Allow users (developers) specify the points of stop-N-Go

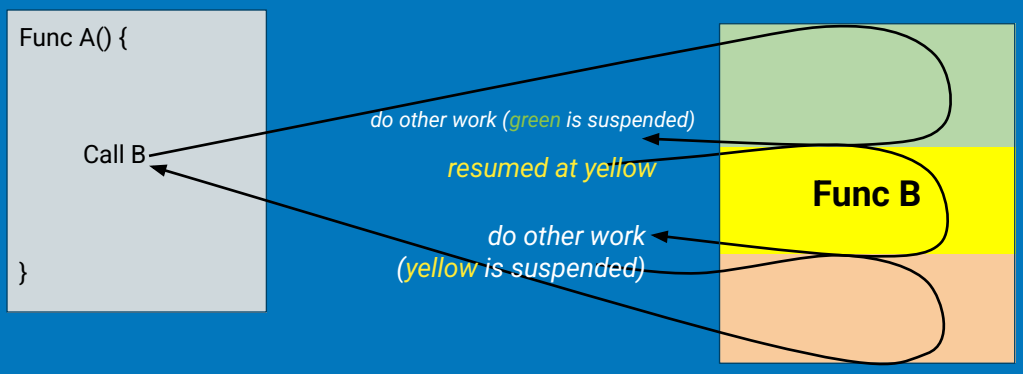
	Conc. Processes	Conc. Threads	Co-Routines
Unit of Work	The entire process	Function Call Chains	Parts of a function
Style of execution	Stop-N-Go	Stop-N-Go	Stop-N-Go
Thread Assignment	--	One (fixed) thread per function call chain	Different parts of a function can run in different threads
Where to suspend?	Decided by OS	Decided by OS or Thread Manager	Decided by developer

- Emphasis on **call chains**
- Unit of work goes from coarse to fine
- They all execute in Stop-N-Go fashion



Invoking a Subroutine (Func B)

Invoking a Coroutine (Suspendable Func B)



Kotlin Example

```
val loginOk = authenticate (user, pass)
println("Authenticated? $loginOk")
if (loginOk) {
    val data = networkRequest()
    saveToLocalStorage(data)
}
else {
    logError("Unauthorized", "2022-05-18")
}
```

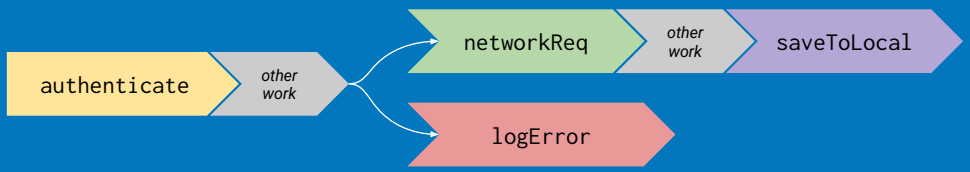
```
suspend fun authenticate(u:String, p:String): Boolean {}

suspend fun networkRequest(): List<BeeHive> {}

suspend fun saveToLocalStorate(___) { /* code here */}

suspend fun logError(reason: String, at: String) {  }
```

At least 4 times Stop-N-Go



Async Functions In Other Languages

```kotlin
// Kotlin
suspend fun authenticate(...): Boolean {}


val loginOk = authenticate (user, pass)
if (loginOk) {
   val data = networkRequest()
   saveToLocalStorage(data)
}
else {
   logError("Unauthorized", "2022-05-18")
}
```

```python
// Python
async def authenticate():
   pass

loginOk = await authenticate (user, pass)
if (loginOk) {
   val data = await networkRequest()
   await saveToLocalStorage(data)
}
else {
   await logError("Unauthorized", "2022-05-18")
}
```

```swift
// Swift
func authenticate() async -> Boolean { }

loginOk = await authenticate (user, pass)
if (loginOk) {
   val data = await networkRequest()
   await saveToLocalStorage(data)
}
else {
   await logError("Unauthorized", "2022-05-18")
}
```


Coroutines ≠ Susp. Functions

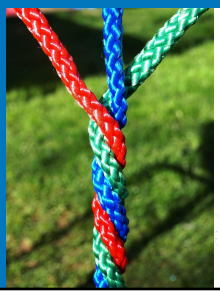
(Technical Depth)

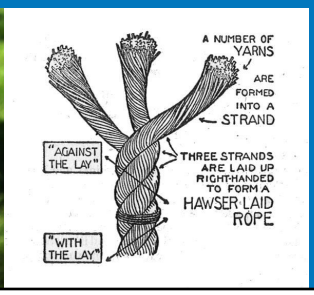
suspend(ing|able) Functions

- Kotlin functions declared with the suspend keyword
- Suspendable functions have “breakpoints” (borrowed from debugging terms) in its function body
  - These breakpoints are calls to other suspendable functions
  - At runtime a suspendable function executes until the next “breakpoint” and gets suspended
  - Function execution resumes when the the (suspend) callee completes its work
- A coroutine (coroutineScope) is a “home” for running suspendable functions
- Suspendable functions are a function that can suspend/resume coroutines

Kotlin Coroutines

- Coroutines are a *suspendable* unit of (code) execution
- In Operating Systems
  - Threads are a lightweight process (LWP)
  - One or more threads may run concurrently within a process
- In Kotlin
  - ***Coroutines are a lightweight thread***
  - One or more coroutines may run concurrently within a thread





Rope ⇒ Strands ⇒ Yarns
Process ⇒ Threads ⇒ Coroutines

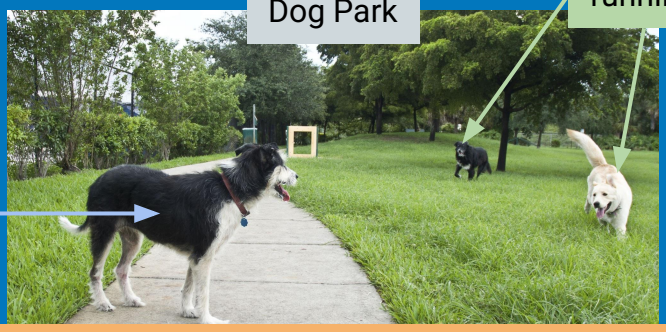
Threads vs. Coroutines

- Threads cannot be suspended
- Threads can only be blocked or unblocked
- When a thread is blocked
  - its execution context is saved by the OS
  - the thread cannot be used to execute other work

- Coroutines executes within a thread
- Coroutines can be suspended and resumed
- To suspend a coroutine, only references to local variables and the suspension location need to be saved as an object (does not require OS assistance)
  - This object is the “Continuation” object
- When a coroutine is suspended, its “host” thread can be used to execute other work or other coroutine

Coroutine: Builder & Runner

Dogs & Dog Parks



- Dogs ⇒ suspendable functions
- Dog Park ⇒ home for functions to run (or suspended)
- Park (play)ground ⇒ threads scheduled to run on CPU(s)
- The Dog Park itself can be suspended (and resumed)
- A (smaller) dog park can be built inside a (bigger) dog park

How to host suspendable functions

Four different ways of creating a “dog park”

	Where created	if a dog is suspended
`runBlocking`	From scratch	the entire park is suspended, executor is also blocked
`coroutineScope`	inside another park	the entire park is suspended, executor may switch to another park
`launch / async`	inside another park and runs concurrently with other “sibling” parks	other “sibling” parks continue to run

Use async when the “last” dog returns a “bone” to you

Coroutine Builders in Kotlin Stdlib

```
fun <R> runBlocking (context: CoroutineContext = EmptyCoroutineContext,
                     block: suspend CoroutineScope.() -> R): R

suspend fun <R> coroutineScope (block: suspend CoroutineScope.() -> R): R

fun CoroutineScope.launch(context: CoroutineContext = EmptyCoroutineContext,
                          start: CoroutineStart = CoroutineStart.DEFAULT,
                          block: suspend CoroutineScope.() -> Unit): Job

fun <R> CoroutineScope.async(context: CoroutineContext = EmptyCoroutineContext,
                             start: CoroutineStart = CoroutineStart.DEFAULT,
                             block: suspend CoroutineScope.() -> R): Deferred<R>
```

All these functions are declared to accept trailing lambdas

Coroutine Builders (Simplified)

```
// runBlocking can be invoked from an ordinary function
fun <R> runBlocking (block: suspend CoroutineScope.() -> R): R

// coroutineScope must be invoked from a suspending function
suspend fun <R> coroutineScope (block: suspend CoroutineScope.() -> R): R

// Both functions below are invoked inside a CoroutineScope
fun CoroutineScope.launch(block: suspend CoroutineScope.() -> Unit): Job
fun <R> CoroutineScope.async(block: suspend CoroutineScope.() -> R): Deferred<R>
```

- Both `runBlocking` and `coroutineScope` returns only when all their child coroutines are complete
- When one of its child coroutines is suspended the thread hosting `runBlocking` is blocked
- When one of its child coroutines is suspended the thread hosting `coroutineScope` can be reused to execute other work
- Both `launch` and `async` create a new coroutine, and they must be used inside a `CouroutineScope` such as `runBlocking` or `coroutineScope`

`runBlocking` vs `coroutineScope`

- Statements (“children”) inside them *execute sequentially*, possible suspended and resumed
- They return (finish executing) when the last statement complete
- The lifetime of these children is (collectively) managed by a `CoroutineScope` object

`runBlocking`

- It is an ordinary function
- If a child statement is suspended, the (“parent”) thread running `runBlocking` stays attached to it (i.e. the thread is blocked from doing other work)

`coroutineScope`

- It is a suspending/suspendable function
- If a child statement is suspended, the (“parent”) thread running `coroutineScope` becomes available to do other work

`launch` vs `async`

- They are extension functions on the `CoroutineScope` class
- They can be invoked inside a `runBlocking` or `coroutineScope` function (i.e. as a child of runBlocking/coroutineScope parent)
- They create a new coroutine (a `Job`) that *executes concurrently* with other siblings of the same parent
  - The statements inside this new coroutine *execute sequentially*

`launch`

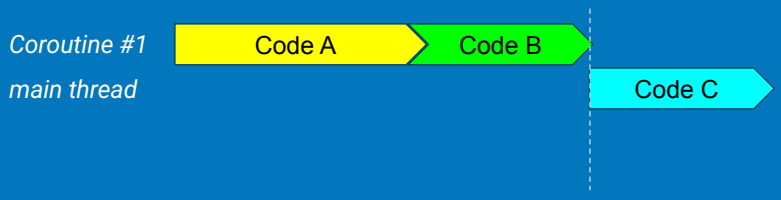
- When the coroutine finish executing, it returns a Unit (“void”)

`async`

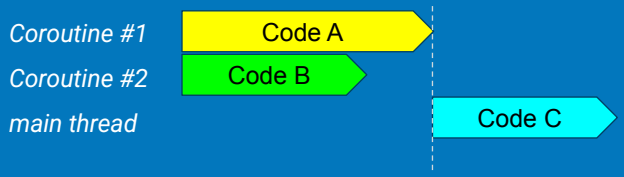
- When the coroutine finish executing, it returns a result of type `T` wrapped as **`Deferred<T>`** which can be unwrapped by calling .await

Coroutine Builders: Initiate a Coroutine

```
fun main() {
  runBlocking {
    /* Code A */
    /* Code B */
  }
  /* Code C */
}
```




```
fun main() {
  runBlocking {
    launch { /* Code A */ }
    launch { /* Code B */ }
  }
  /* Code C */
}
```



Online Playground

In both scenarios, Code C (NOT a coroutine) runs only after runBlocking() completed; ***all the coroutines also run under the main thread***

Which Kind of Date Do You Prefer?

```
runBlocking {
  launch {
    // eat the food (15 mins)
  }
  launch {
    // chat with your date (30 mins)
  }
}

// check your TikTok (10 mins)
```

Total time = _____

```
runBlocking {
  launch {
    // eat the food (15 mins)
  }
  launch {
    // chat with your date (30 mins)
  }
  launch {
    // check your TikTok (10 mins)
  }
}
```

Total time = _____

Switching Analogy

(introducing Dispatchers)

Concept	Previous Analogy	Next Analogy
Suspendable functions	Dogs	Airplanes/Trains
Running scope	Dog Park(s)	Airport(s)/Train Stations
Threads on CPU(s)	Park (play)ground	Runways/Train Tracks

Coroutine Dispatchers

- Air Traffic Controller assigns airplanes to runways for take-off or landing
- Coroutine Dispatchers let you choose which thread to run a coroutine
  - `.Main`: for UI/Non-blocking tasks
  - `.IO`: optimized for doing I/O intensive tasks (disk or network)
  - `.Default`: for CPU intensive tasks
  - Thread pools created by `newSingleThreadContext()`

Launch Coroutines on a specific Thread

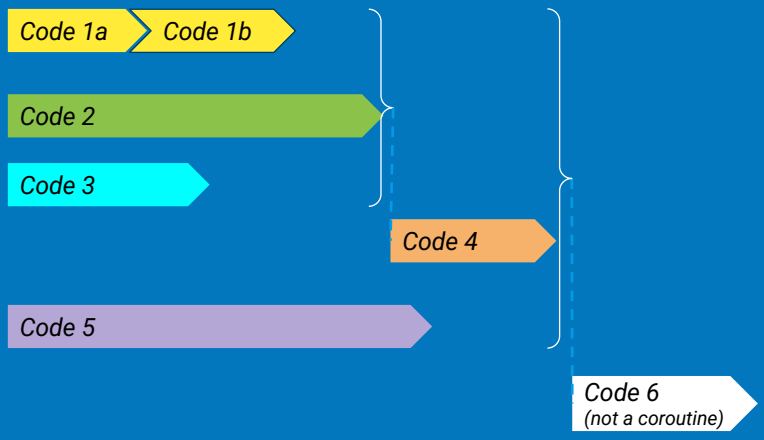
```
runBlocking {
  launch {
    // These two functions begin on the current thread
    someFunction1()
    someFunction2()
  }
  launch(Dispatchers.IO) {
    // The functions in this co-routine begin on the IO thread
    someFunction3()
  }
  launch(newSingleThreadContext("Yup!")) {
    // The functions in this co-routine begin on a newly created thread
    someFunction4()
  }
}
```


Kotlin Structured Concurrency

- Execution scopes created using `runBlocking()` or `coroutineScope()` automatically manage their children suspension and completion
- This execution scope is syntactically (and semantically) inferred from the block scope `{ /* lambda here */ }` (i.e. pair of curly braces)
- Calls to `runBlocking()`, `coroutineScope()`, `launch()`, and `async()` can be nested within each other
  - The nesting structure also indicates parent/child relationships among the couroutines

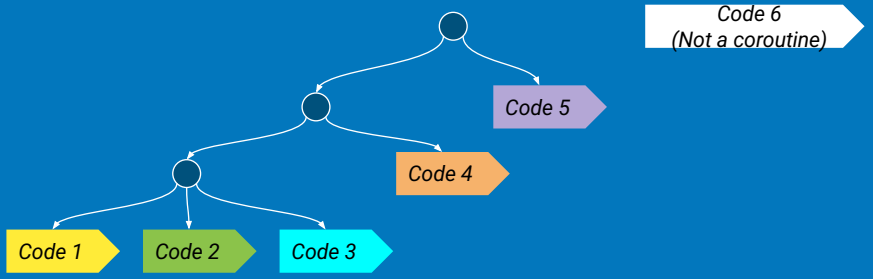
Kotlin Structured Concurrency

```
runBlocking {
  launch {
    coroutineScope {
      launch {
        // Code 1a
        // Code 1b
      }
      launch {
        // Code 2
      }
      // Code 3
    }
    // Code 4
  }
  launch {
    // Code 5
  }
}
// Code 6
```



Job/Coroutine Parent-Child Hierarchy

```
runBlocking {
  launch {
    coroutineScope {
      launch {
        // Code 1a
        // Code 1b
      }
      launch {
        // Code 2
      }
      // Code 3
    }
    // Code 4
  }
  launch {
    // Code 5
  }
}
// Code 6
```



withContext(): Switch dispatcher

```
runBlocking {
  launch(Dispatcher.IO) {
    // Code 1
    launch(Dispatcher.Main) {
      // Code 2
    }
    // Code 3
  }
}
```

Two coroutines created
- Coroutine #1 executes Code 1 and Code 3 on the IO thread
- Coroutine #2 executes Code 2 on the Main thread

```
runBlocking {
  launch(Dispatcher.IO) {
    // Code 1
    withContext(Dispatcher.Main) {
      // Code 2
    }
    // Code 3
  }
}
```

Only one coroutine created
- Code 1 and Code 3 runs on the IO thread
- Code 2 runs on the Main thread

Practical use case:
- Code 1 makes a network request to fetch data
- Code 2 uses the data to update the UI

Cooperative Coroutines

```
// Non-cooperative coroutine
val poorJob = runBlocking {
   var k = 0
   while (k < 10_000) {
     // Do network calls here
     k++
   }
}

// Elsewhere in your app
// Has to wait until
// 10_000 iterations
poorJob.cancel()
poorJob.join()
```

```
// Cooperative coroutine
val betterJob = runBlocking {
   var k = 0
   while (k < 10_000 && isActive) {
     // Do network calls here
     k++
   }
   repeat(10_000) {
     yield()
     // Do network calls here
   }
}

// Elsewhere in your app
// Cancelable at any iteration
betterJob.cancelAndJoin()
```

Both isActive and yield() are Kotlin builtin prop/function


Android conference
talks:
Kotlin Coroutines
101