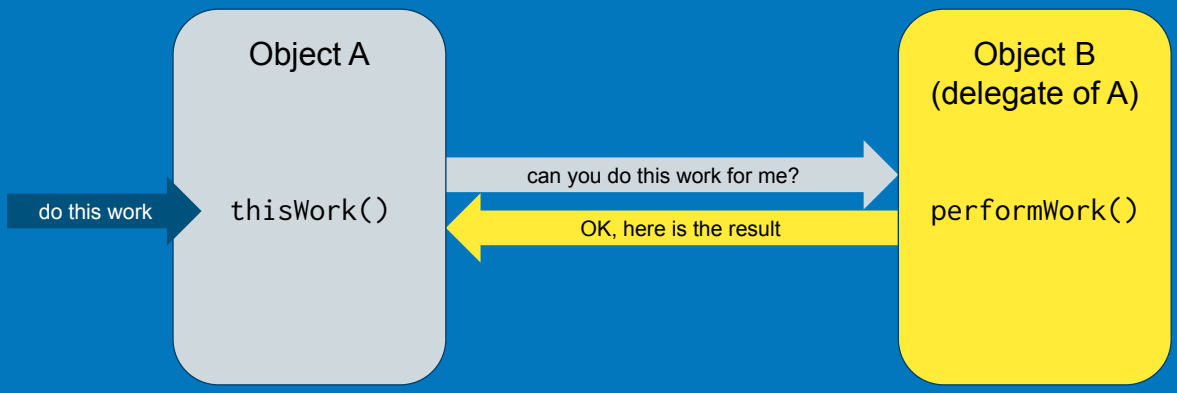
Kotlin Features for Android Development

Topics

- Delegation in Kotlin
  - Built-in delegates: lazy
- Kotlin Coroutines
- Advanced [Kotlin Flow: SharedFlow, StateFlow]

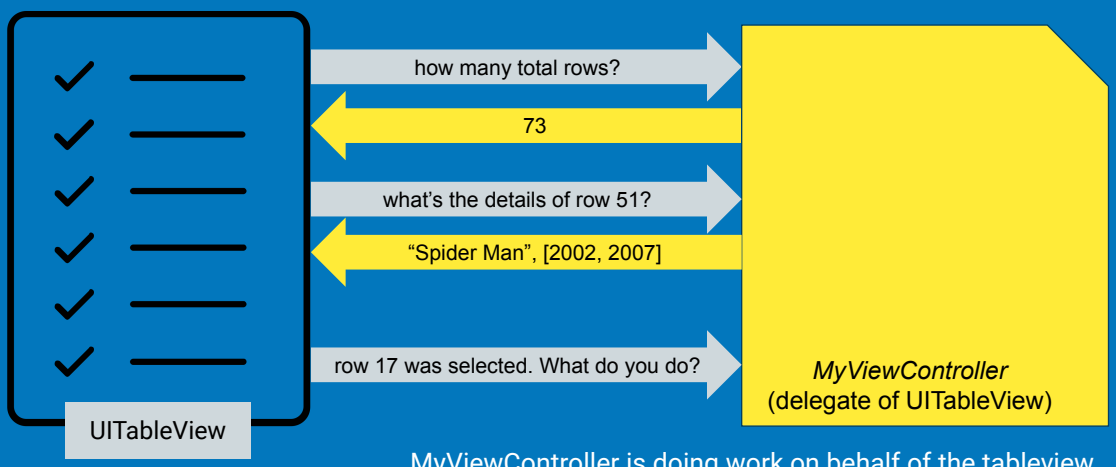
Kotlin Delegation

Delegation in general



- Object A delegates some of its work to Object B
- Work supposed to be rendered by A is actually completed by B

Delegation in iOS



MyViewController is doing work on behalf of the tableview

Delegation in Kotlin

- Three techniques for enhancing feature of a class X
  - Using inheritance/interface: requires X to implement specific functions
  - Using composition: add a subobject in X that implements those functions
  - Using Kotlin delegation (a mix of inheritance & composition)
- Kotlin provides native support for delegation via the by keyword
- Two kinds of delegation
  - Class delegation: delegate the implementation of an interface from X to an object Y
  - Property delegation: use an object Y to provide the value of a property of X

Why Delegation: Motivation

How to enhance a class to enable credit card payment?

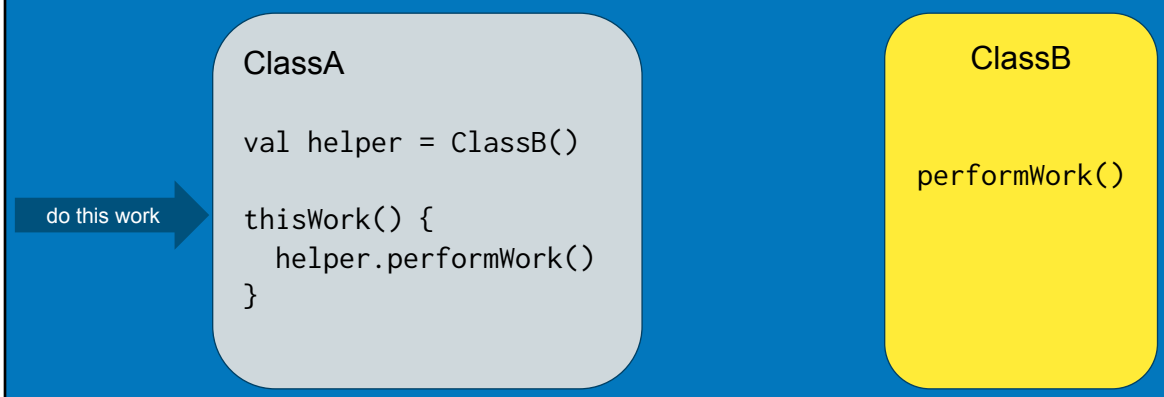
```
interface VisaPayment {
  fun pay(cardNumber: String, amount: Float)
}
```

Solution #1: Enhancement by Inheritance

```
interface VisaPayment {
  fun pay(cardNumber: String, amount: Float)
}
```

```
// Version 1: Traditional solution using inheritance
class MyAppV1: VisaPayment {
  override fun pay(cardNumber: String, amount: Float) {
    // your own code here
  }
}

fun main() {
  val app = MyAppV1()
  app.pay("XXXX YYYY ZZZZ WWWW", 12.50f)
}
```


Enhancement Via Composition



- ClassA declares a subobject of type ClassB
- Work supposed to be rendered by A is actually completed by invoking a method of ClassB

Object Composition (Refresher)

A car is composed of an engine, a transmission box, and many more...

```java
// Car.java
class Car {
    CombustionEngine engine;
    AutomaticTransmission transmission;

    void prepareForDrive() {
        engine.start()
        transmission.shiftTo(Gear.DRIVE)
    }
}
```

```kotlin
// Car.kt
class Car (val engine: CombustionEngine,
           val transmission: AutomaticTransmission) {


    fun prepareForDrive(): Unit {
        engine.start()
        transmission.shiftTo(Gear.DRIVE)
    }
}
```


Solution #2: Enhancement by Composition

```kotlin
interface VisaPayment {
  fun pay(cardNumber: String, amount: Float)
}
```

```kotlin
class PayPal: VisaPayment {
  fun pay(cardNumber: String, amount: Float) {
    // code implemented by a third-party library
  }
}
```

```kotlin
// Version 2: Solution using object Composition
class MyAppV2 (val payAgent: VisaPayment) {
  fun doPayment(cardNumber: String, amount: Float) {
    payAgent.pay(cardNumber, amount)
  }
}

fun main() {
  val service = PayPal()
  val app = MyAppV2(service)
  app.doPayment("XXXX YYYY ZZZZ WWWW", 12.50f)
}
```

Solution #3: Kotlin Class Delegation

```
interface VisaPayment {
  fun pay(cardNumber: String, amount: Float)
}
```

```
class PayPal: VisaPayment {
  fun pay(cardNumber: String, amount: Float) {
    // code implemented by a third-party library
  }
}
```

```
class MyAppV3 (val payHelper: VisaPayment): VisaPayment by payHelper {
  // No "pay" function required here
}

fun main() {
  val app = MyAppV3(PayPal())
  // this invokes the pay method provided by PayPal
  app.pay("XXXX YYYY ZZZZ WWWW", 12.50f)
}
```

name payService is insignificant

```
class MyAppV4 : VisaPayment by PayPal()

fun main() {
  val app = MyAppV4()
  app.pay("XXXX YYYY ZZZZ WWWW", 12.50f)
}
```

Playground

Summary of Kotlin Delegation

```
class MyApp : VisaPayment by PayPal()

fun main() {
  val app = MyApp()
  app.pay("XXXX YYYY ZZZZ WWWW", 12.50f)
}
```

Kotlin Delegation enables "injection" of a *function* (pay) provided by a *third-party* (PayPal) into your own code

Property Delegation

Property: Getter & Setter

```
// Person.java
class Person {
    private String name;

    public String getName() {
        return this.name;
    }

    public void setName(n:String) {
        this.name = n;
    }
}
```

```
// Person.kt
data class Person (val name:String)
```

- Properties require *getter* and *setter*
- When the actual implementation of a property is delegated to an external object/class, the class must provide getter and setter as well
  - Internally, it uses Kotlin function overloading

Property Delegation

"Borrow" setter and getter defined by an external class

```
class ProperName {
  private var myName = ""
  operator fun getValue(p: Person, z: KProperty<*>): String {
    // perform work to return the value of the property
  }

  operator fun setValue(p: Person, z: KProperty<*>, newValue:String) {
    // perform some work to return the value of the property
  }
}

class Person {
    var name:String by ProperName()
}
```

Online Playground

Summary (of Kotlin Delegation)

- A new technique for "borrowing" *functions/methods* from another class to your class
  - Without using class inheritance
  - Without using object composition
- Functions from the other class are directly "injected" into your class, making them to appear as if they are defined internally in your class
- Property delegation is a special case of borrowing getter and setter (from an external provided) for selected variable(s) in your class

Kotlin Delegation

```
// Class Inheritance
class MyClass : NameOfInterface {
  // This class must implement the interface
}
```

```
// Class Delegation
class MyClass : NameOfInterface by HisClassProvidingNewFunctions() {

}
```

```
// Property Delegation
class MyClass {
  var abc: Int by SetterAndGetterForInt()
}
```


Kotlin Coroutines (Async + Concurrent Programs)

Topics

- Function calls: **synchronous** vs. **asynchronous**
- Concurrent Execution with Threads
- Concurrent Execution with Coroutines
- Kotlin Coroutines
  - Suspending Functions
  - Coroutine Builder Functions
  - Structured Concurrency
  - Dispatchers
  - Coroutine Context
- Prerequisites: (trailing) lambdas

Why Need Concurrency?

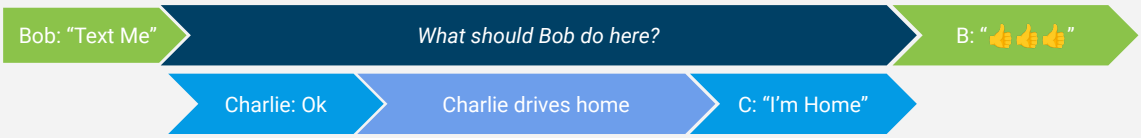
- Interactive apps (such as Android apps) rely on the existence of the *UI thread* to update the UI screen
- Sometimes your app needs to perform "heavy" work
  - Reading/Writing Database
  - Obtaining data from remote servers (weather, stock prices, event calendars, ...)
- Running these tasks on the UI thread will degrade app responsiveness
- Need different thread(s) to execute "heavy" work, but managing threads require extra overhead on the OS

Concurrent vs. Asynchronous

- Co-occurrence
- Two (or more) actions are happening about the same time

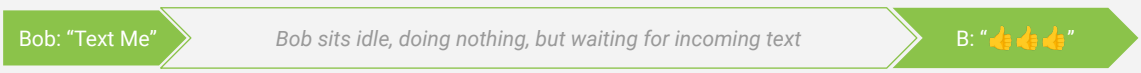
- Not synchronous, unsynchronized
- *Literal meaning*: two (or more) actions which are not happening simultaneously
- *In the context of programming*: completion mode of function calls
  - Synchronous function calls
  - Asynchronous function calls
- Asynchronous function call implies concurrency

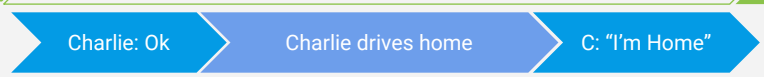
Asynchronous Actions?



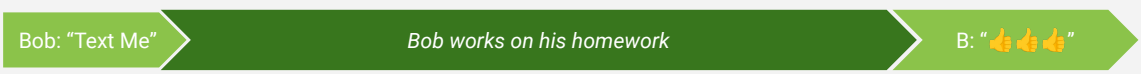
Sync. vs. Async. Function Calls

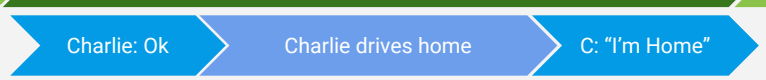
Bob sends Charlie home synchronously (NO concurrency)





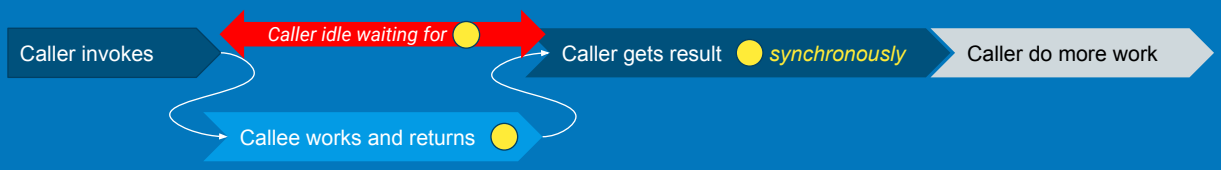
Bob sends Charlie home asynchronously

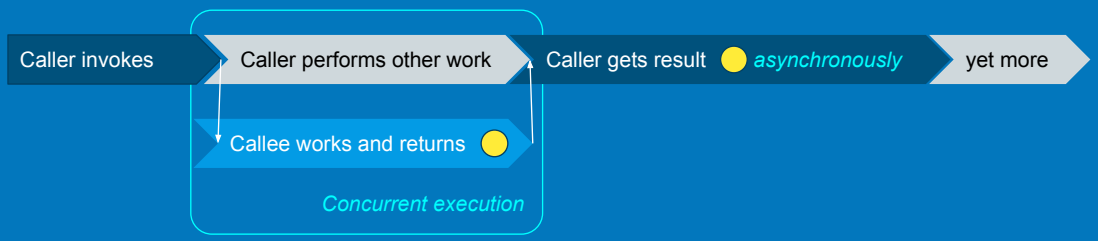




Concurrency ⇏ Parallelism
Parallelism ⇒ Concurrency
Asynchronous ⇒ Concurrency

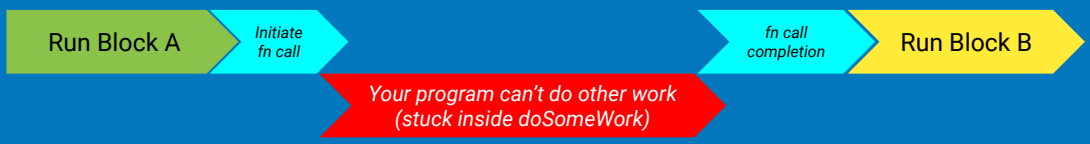
Sync vs. Async Function Calls





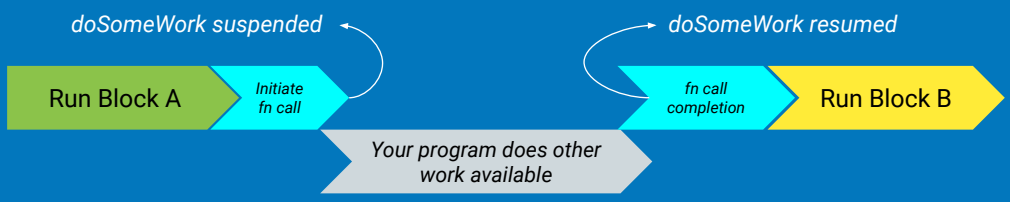
Ordinary Functions

```
fun doSomeWork(): Unit {
  /* Code block A */
  callToAnotherFunc( /* args */)
  /* Code block B */
}
```



Suspend(able) Functions

```
suspend fun doSomeWork(): Unit {
  /* Code block A */
  callToSuspendFunc( /* args */)  // => "breakpoint"
  /* Code block B */
}
```



Async Functions in Other Languages

```python
# Python
import asyncio
async def genPassword(len):
  # generate password of given length

async def resetMyPassword():
  pw = await genPassword(12)
  print(f`Your password is {pw}`)
```

```kotlin
// Kotlin
suspend fun genPassword(len: Int): String {
  // generate password of given length
}

suspend fun resetMyPassword() {
  val pw = genPassword(12)
  println("Your password is $pw")
}
```

In Kotlin:
A suspending function can only be called from another suspending function or within a co-routine scope

Online Playground

Convenient Features Provided by Coroutines

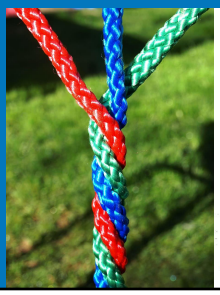
- An automated mechanism (at the language level, not at the OS level) to suspend and resume function executions
- Write an asynchronous program (almost) in the same way as a synchronous (sequential) order

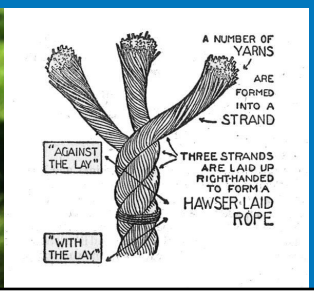
suspend(ing|able) Functions

- 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
- Suspendable functions are **NOT** a coroutine
- 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 coroutine

Threads ⇒ Lightweight Processes

Coroutines ⇒ Lightweight Threads

How lightweight are Kotlin Coroutines

- Threads require OS to allocate (& manage) execution context on stack
  - Execution of Coroutines are managed at the language level (using a simple Continuation object)
- When execution of a thread switches to another thread, OS must save and load execution context of each thread
  - Switching between coroutines does not require OS interception

Coroutine: Builder & Runner

Coroutine Building Blocks: suspend functions

```
// Kotlin
suspend fun genPassword(len: Int): String {
  // generate password of given length
}

suspend fun resetMyPassword() {
  val pw = genPassword(12)
  println("Your password is $pw")
}
```


Coroutine Runner Functions

Suspending functions can only be called inside another suspending function

- Q: How to invoke the *first* suspending function?
- A: Use one of the following builder/runner functions
  - **`runBlocking()`**: is a **non-suspending** function that creates a coroutine scope that blocks until all the ("child") coroutines complete their execution
  - **`coroutineScope()`**: similar to `runBlocking()` but is a **suspending** function itself, so when any of the ("child") coroutines is suspended, the scope is also suspended
  - **`launch()`**: starts a new coroutine concurrently with the rest of the code
  - **`async()`**: similar to `launch()` but the lambda block may return a result to the thread that launched it

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`

Runtime Behavior (of coroutine builders)

`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

- 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)

- 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*

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

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

Some Terminologies

Important reminder: a coroutine is just *a suspendable function*

- `CoroutineScope`
  - A mechanism to manage the execution of a group of coroutines
  - The total lifetime of the (parent) scope is the lifetime of all the coroutines (combined)
- `Job`: is a handle (“reference”) to a coroutine
  - Important for cancellation and exception handling
- `Dispatchers`
  - In OS a function needs a “playground” (i.e. thread) to run
  - Likewise, a coroutine must be dispatched to a thread to run
- `CoroutineContext`: an object that maintains the execution context of a coroutine

Lambda Syntax Refresher

```
// repeat() defined in Kotlin stdlib with a trailing lambda
fun repeat(times: Int, action: (Int) -> Unit) {
  for (index in 0 until times) {
    action(index)
  }
}
```

λ

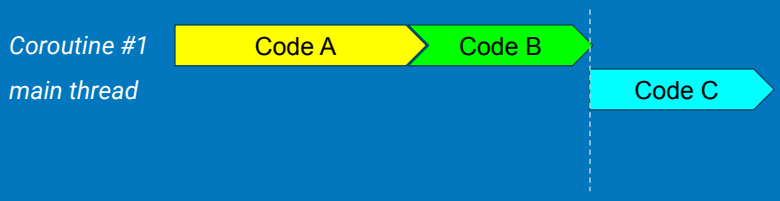
```
// Use it without lambda
fun someWork(arg: Int) {
  println ("Hello $arg")
}

fun main() {
  repeat (100, ::someWork)
}
```

```
// Use it with lambda
// inside parentheses

fun main() {
  repeat (100, {
    println("Hello $it")
  })
}
```

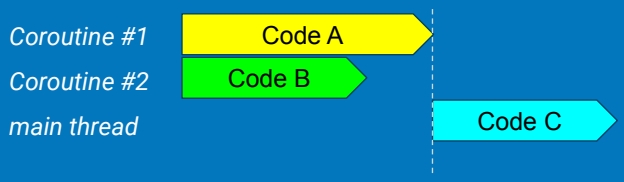
```
// Use it with lambda
// Easier to read


fun main() {
  repeat (100) {
    println("Hello $it")
  }
}
```


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 = _____

Coroutines must run within a thread

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

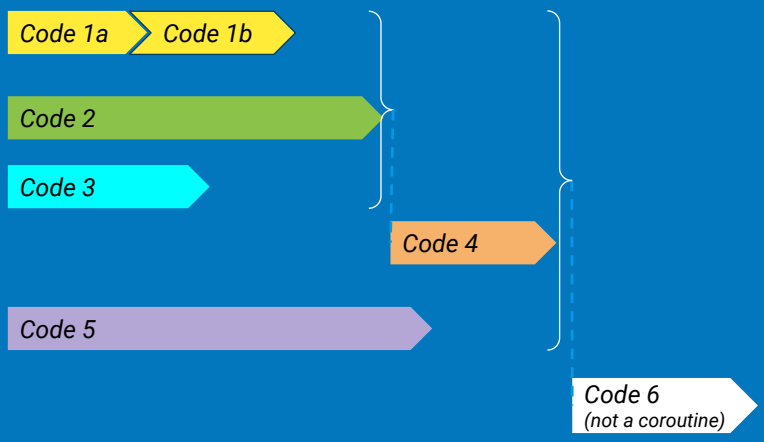
```
runBlocking {
  launch {
    // These two functions begin on the current thread
    someFunction1()
    someFunction2()
  }
  launch(Dispatchers.IO) {
    // The functions in this co-routine begin on the IO thread
    someFunction3()
  }
  launch(newSingleThreadContext(“Yup!”)) {
    // The functions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

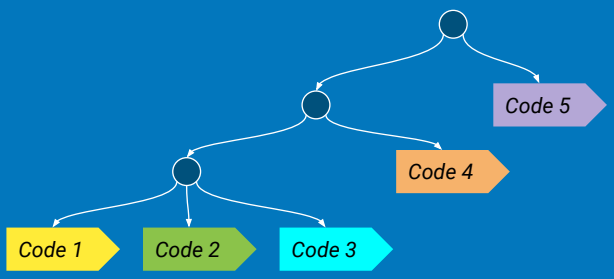
Kotlin Structured Concurrency

```
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