Kotlin

Arthur Hoskey, Ph.D.
Farmingdale State College
Computer Systems Department

Today’s Lecture

- Kotlin Coroutines
  - Overview
  - Coroutine Scope
  - Coroutine Builder
  - Coroutine Dispatcher
  - DB Operations and Coroutines
  - Coroutines and Critical Sections

Coroutine

- An instance of a suspendable computation. This means it can be stopped then restarted.
- Similar to a thread, runs code currently with other program code.
- A coroutine is not associated with a particular thread (the same coroutine can be run on different threads).
- A coroutine can suspend its execution on one thread and complete its execution on another thread.
- Coroutines take up less resources than threads.

Create a coroutine scope. Dispatcher.IO will make it run on another thread.

```
val myScope = CoroutineScope(Dispatchers.IO)
myScope.launch {
    // Code to run in coroutine
}
```

launch. Coroutine builder that launches a new coroutine. Code inside launch runs concurrently with non-coroutine code.

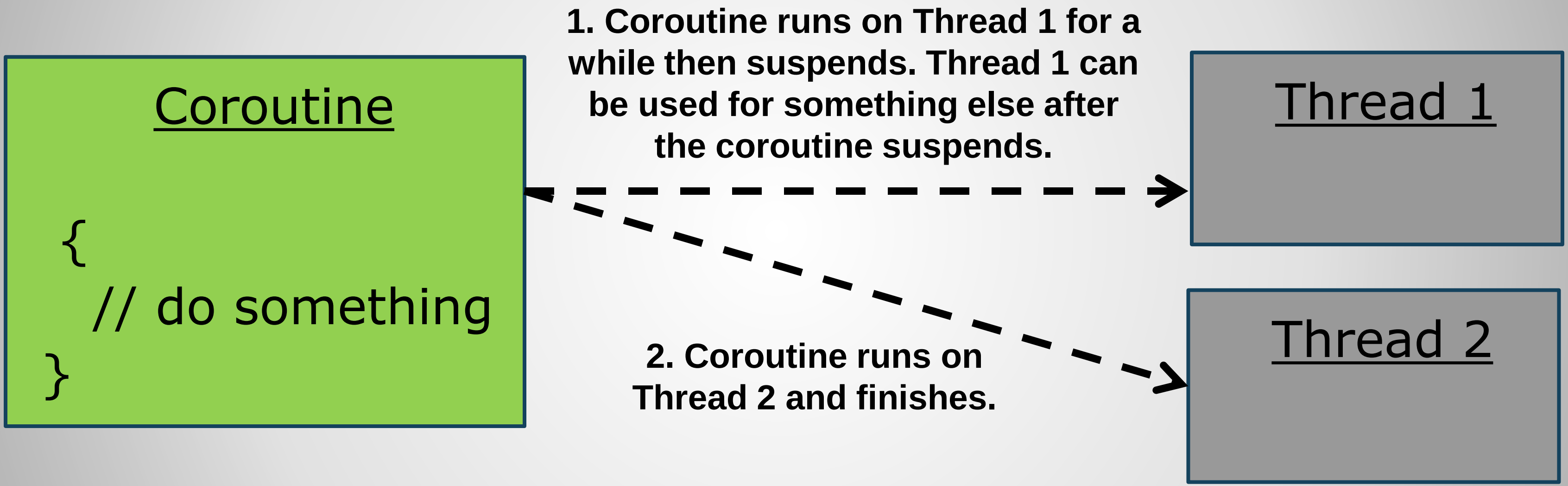
- Taken from the following link:

https://kotlinlang.org/docs/coroutines-basics.htmls

Coroutine

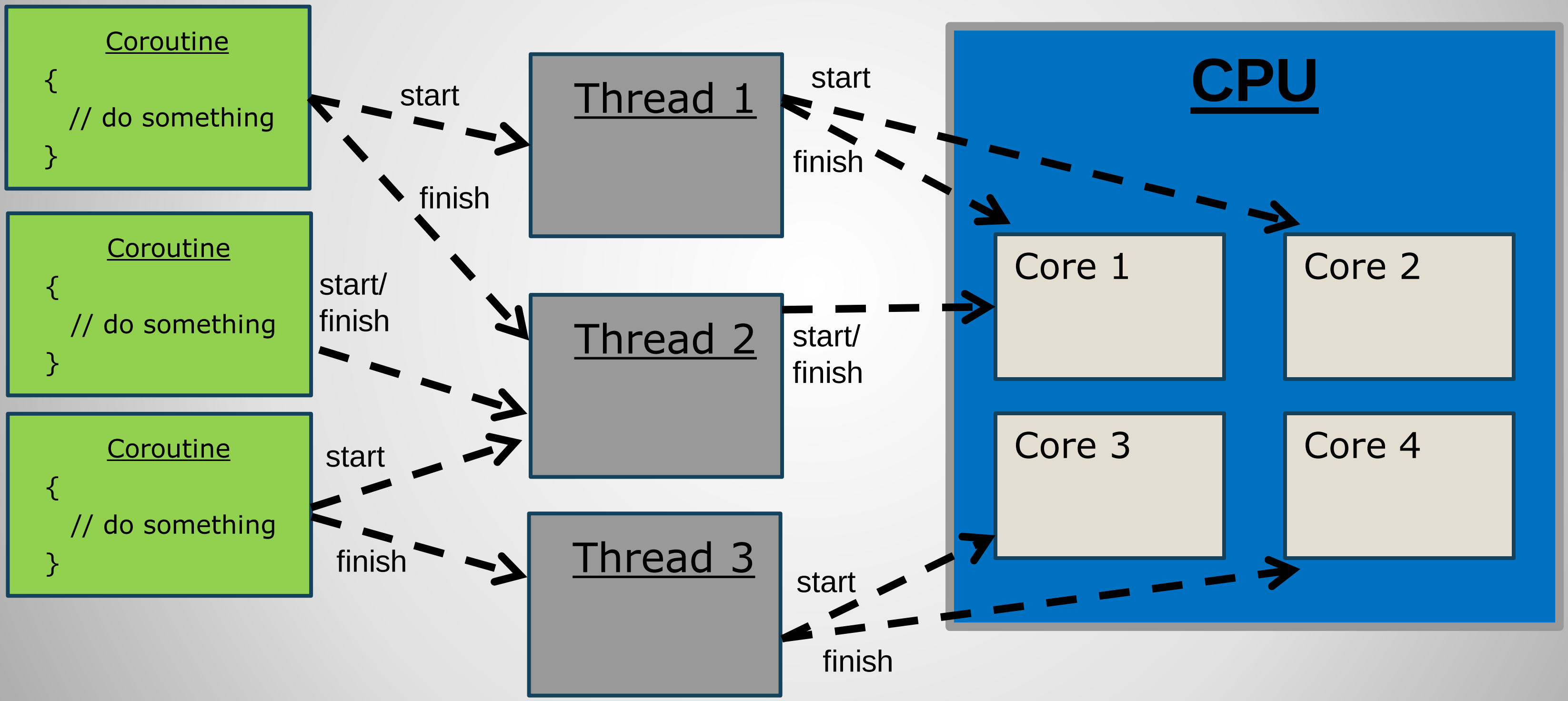
Coroutine

- In this example, the coroutine starts on Thread 1 and then suspends.
- When the coroutine resumes it runs on Thread 2 and finishes.


Coroutine
{
// do something
}
1. Coroutine runs on Thread 1 for a while then suspends. Thread 1 can be used for something else after the coroutine suspends.
Thread 1
2. Coroutine runs on Thread 2 and finishes.
Thread 2

Coroutine Runs Can Run on Different Threads

Multiple coroutines can each execute on different threads. Coroutines can start on one thread, suspend, and continue on another thread. Threads can move between cores.



Coroutines and Threads

Coroutine Example

- The code in the coroutine block runs independent of the other code.

Output
1
2

Create coroutine scope

```
fun normalMethod() {
    val myScope = CoroutineScope(Dispatchers.IO)

    myScope.launch {

        delay(2000L) // non-blocking delay for 2 seconds
        Log.d("MY_DEBUG", "2") // print after delay

    }

    Log.d("MY_DEBUG", "1") // Prints 1 before coroutine prints 2
}
```

This block creates and starts a coroutine

Coroutine code

Code outside coroutine

Coroutine Example

suspend Keyword

- A function can be decorated with the suspend keyword.
- suspend means that the function can be blocked (suspended).
- If a function is decorated with suspend it can only be called from within a coroutine.

This function can only be called from a coroutine (since it is a suspend function)

```
suspend fun doSomething() {

    // Code for some long running operation goes here

}
```

suspend Keyword

suspend Function Example

- The code below uses a user-defined suspend function.

```
fun normalMethod() {
    val myScope = CoroutineScope(Dispatchers.IO)
    myScope.launch {
            doSomething()
    }
    Log.d("MY_DEBUG", "1") // Prints 1 before coroutine prints 2
}

suspend fun doSomething() {
    delay(2000L)
    Log.d("MY_DEBUG", "2") // print after delay
}
```

Call the user-defined suspend function from the coroutine scope

User-defined suspend function definition. This function can only be called from a coroutine scope.

suspend Function Example

Coroutine Scope

- Now on to coroutine scope…

Coroutine Scope

- A coroutine scope determines how long a coroutine can live.
- A coroutine scope does NOT start a coroutine.
- The coroutine scopes listed below are recommended because they will automatically cancel jobs according to the app's lifecycle.

- Suggested coroutine scopes to use:
  - **lifeCycleScope** – Coroutines will run according to the lifecycle of the containing activity. If the containing activity is destroyed, then the coroutines will also be destroyed.
  - **viewModelScope** – Coroutines run according to the view model lifetime. If the view model is destroyed, then the coroutines will also be destroyed. This can only be used from inside of a view model.

Coroutine Scope

Coroutine Scope Examples

- Coroutine lifetime is determined by the coroutine scope.

Life Cycle Scope. Coroutines will only be canceled when the containing activity ends.

```
lifecycleScope.launch {
   // Coroutine code here
}
```

ViewModel class

```
class MainViewModel : ViewModel() {

    fun doSomething() {

        viewModelScope.launch {
            // Coroutine code here
        }
    }
}
```

View Model Scope. Coroutines will only be canceled when the view model is destroyed.

viewModelScope can only be used from inside a view model.

Coroutine Scope Examples

<u>Other Coroutine Scopes</u>

- There are other coroutine scopes that require canceling to be sure that jobs are not running longer than they should.
- For example, there may be coroutines that are tied to the view model and should be canceled if the view model is destroyed. This would not happen using the scopes below.

- Other coroutine scopes:
  - **GlobalScope** – Coroutines will run for as long as the application is running. If the activity that the coroutines are running in is destroyed, the coroutines will still keep running. This could be very bad because the activity cannot be garbage collected (could cause memory problems).
  - **CoroutineScope** – Coroutines run in a general coroutine scope. Other scopes are derived from this scope. Need to make sure that jobs in this scope are canceled if necessary.

Other Coroutine Scopes

More Coroutine Scope Examples

Global Scope. Coroutines will only be canceled when the app itself ends

```
GlobalScope.launch {
    // Coroutine code here
}
```

Coroutine Scope. Coroutines will only be canceled when the app itself ends

```
val myScope = CoroutineScope(Dispatchers.IO)

myScope.launch {
 // Coroutine code here
}
```

Use scope variable to launch the coroutine

More Coroutine Scope Examples

Coroutine Builder

- Now on to coroutine builder…

Coroutine Builder

Coroutine Builder

- Coroutine builders create and start coroutines.
- Here are some coroutine builders:
  - **launch** – Concurrent. Creates and starts coroutines that run independent of the calling code (does NOT block the calling code). It is used for "fire and forget" type execution. This means it does not return a result. The calling code is not waiting for the coroutine to return a value to it.
  - **async** – Concurrent. Creates and starts coroutines that need to return a value. These coroutines run independent of the calling code. The calling code can call await when it gets to a point where it must have the value that it is waiting for (await will make it wait for the value to be returned).
  - **runBlocking** – Blocking. Runs coroutines but blocks all other activity on the current thread. Other activity cannot run until all coroutines in runBlocking are finished. The benefit is that it allows normal (nonsuspendable) functions to call suspendable functions. Coroutines run inside of runBlocking are executed sequentially. Bridges the gap between regular blocking code and suspendable code.

Coroutine Builder Examples

- Coroutine builder creates and starts a coroutine.

launch coroutine builder. The block following launch is the code that will execute in the coroutine.

```
lifecycleScope.launch {
   // Coroutine code here
}
```

launch coroutine builder. Starts first coroutine.

```
lifecycleScope.launch {
   // Coroutine code here

   launch {
      // Coroutine code here
   }
   launch {
      // Coroutine code here
   }
}
```

launch coroutine builder. Starts a second coroutine (uses same settings as the parent coroutine scope).

launch coroutine builder. Starts a third coroutine (uses same settings as the parent coroutine scope).

Coroutine Builder Examples

More Coroutine Builder Examples

- Async coroutine builder example

```
val coroutineScope = CoroutineScope(Dispatchers.IO)
coroutineScope.launch(Dispatchers.IO) {

    val deferredResult = async {
        longRunningOperation()
    }
    // Do things here that do not require the async result.
    // This code runs concurrent with the async code.
    val resultFromLongRunningOperation = deferredResult.await()
    // Do things here that require the result from the async coroutine.
}

suspend fun longRunningOperation() : String{
 // Network, database, file, or other long running code here
    return "Data from longRunningOperation";
}
```

async coroutine builder. Starts a coroutine that returns a result.

Call await on deferredResult. This will suspend the coroutine until the result is returned.

More Coroutine Builder Examples

More Coroutine Builder Examples

- runBlocking creates and starts a coroutine.

```
runBlocking {
    launch {
        delay(2000)
        println("Coroutine 2")
    }
    launch {
        delay(2000)
        println("Coroutine 3")
    }

    println("Coroutine 1")
}

println("After coroutine")
```

runBlocking creates and starts a coroutine

Start a 2nd coroutine

Start a 3rd coroutine

This message is likely to appear before the two other coroutine messages since the other coroutines must be created and started

"After coroutine" will always be the last output in this example because it appears after runBlocking (cannot run this statement until AFTER runBlocking finishes executing all its coroutines)

The output will likely be:

```
Coroutine 1
Coroutine 2
Coroutine 3
After coroutine
```

Note: "After coroutine" will always be last but there is a chance the other lines may appear in a different order.

More Coroutine Builder Examples

Coroutine Dispatcher

- Now on to coroutine dispatcher…

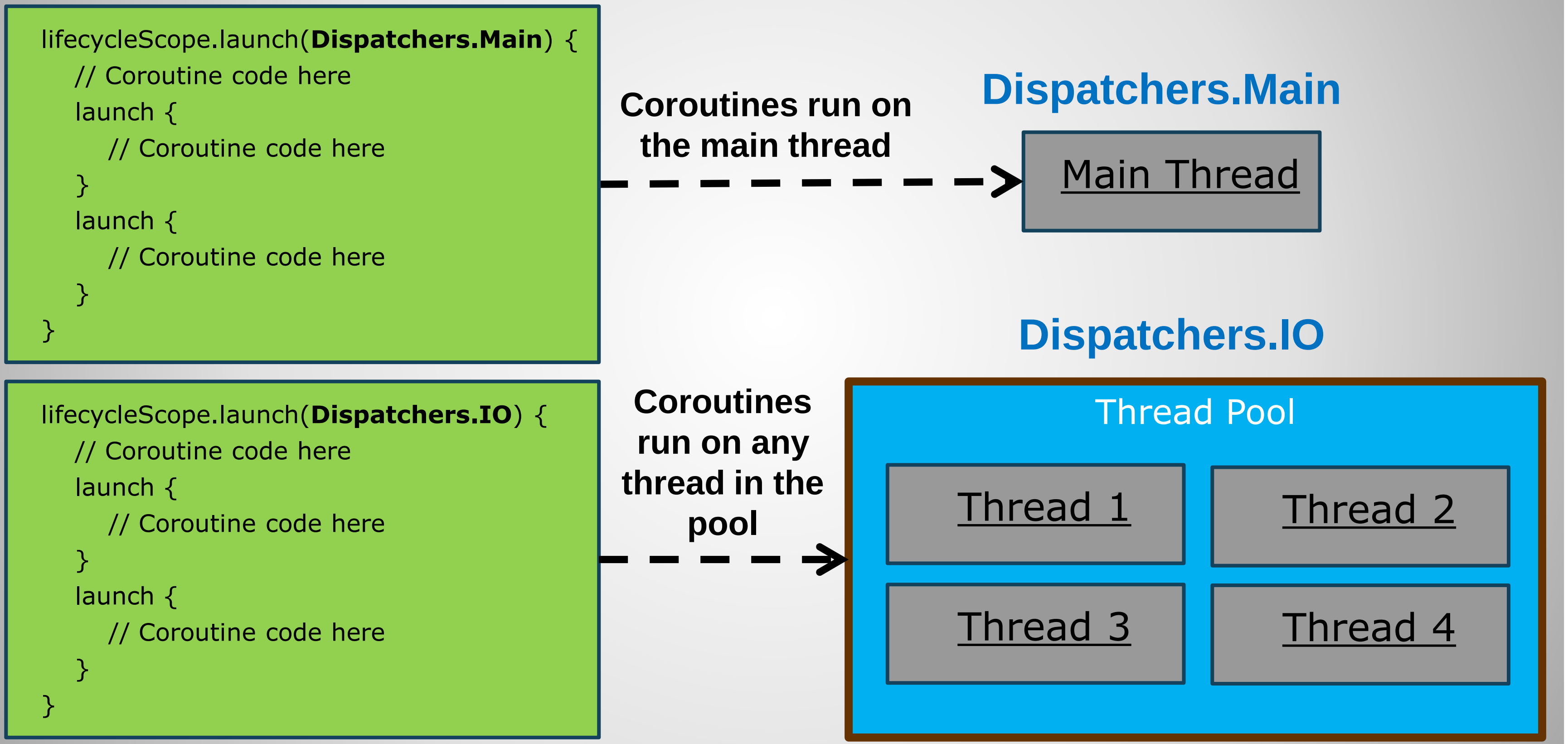
Dispatchers

Dispatchers

- Indicate which threads a coroutine can execute on.
- A coroutine builder is given a dispatcher value. It will use the dispatcher value to determine which thread(s) it can execute its coroutines on.
- Here are some dispatchers (there are others):
  - **Dispatchers.IO** – Coroutines can run on any threads from a background pool of threads (not the main thread). Use this dispatcher for long running background tasks such as network, database call, or any type of file input/output.
  - **Dispatchers.Main** – Coroutines run on the main thread.

Dispatcher Examples

- The dispatcher decides which thread(s) the coroutine can execute on.



```
lifecycleScope.launch(Dispatchers.Main) {
    // Coroutine code here
    launch {
        // Coroutine code here
    }
    launch {
        // Coroutine code here
    }
}
```

```
lifecycleScope.launch(Dispatchers.IO) {
    // Coroutine code here
    launch {
        // Coroutine code here
    }
    launch {
        // Coroutine code here
    }
}
```

Dispatcher Examples

DB Operations and Coroutines

- Now on to DB operations and coroutines…

DB Operations and Coroutines

- If you are running code that uses a database, you will get a compile error if it runs on the main thread.
- For example, use viewModelScope to run a coroutine within a ViewModel.
- viewModelScope is better than a normal coroutine scope because it is lifecycle aware (will cancel its coroutines if the ViewModel is cleared).
- For example:

viewModelScope.launch. Coroutine builder that launches a new coroutine. Code inside the block runs concurrently with non-coroutine code. Make sure to specify Dispatchers.IO

```
// Function in ViewModel
fun loadFromDB() {
    viewModelScope.launch(Dispatchers.IO) {
        // Call method to query DB here
    }
}
```

```
~~viewModelScope.launch {
    // Call method to query DB here
}~~
```

IMPORTANT! viewModelScope defaults to Dispatchers.Main. So, an error will occur unless Dispatchers.IO is passed in.

DB Operations and Coroutines

DB Operations and Coroutines – Example with Error

- The following will NOT WORK!!!

Launch uses the default dispatcher in this case since none is specified (Dispatchers.Main.immediate). The default dispatcher will run on the main thread. You will receive the following error (or something similar):

Cannot access database on the main thread since it may potentially lock the UI for a long period of time.

```
~~// Function in ViewModel
fun loadFromDB() {
    viewModelScope.launch() {
        // Call method to query DB here
    }
}~~
```

DB Operations and Coroutines

- Kotlin coroutines and critical sections…

Kotlin Coroutines and Critical Sections

Mutex

- Use Mutex to create a critical section for coroutines.
- withLock creates a block that automatically acquires and releases the lock.
- withLock can an only be used inside of a suspend function.

```
val mutex = Mutex()            <- Declare mutex variable

val myScope = CoroutineScope(Dispatchers.IO)

myScope.launch {                 Mutex variables can only be used inside
                                 suspend functions (launch is suspend)
    mutex.withLock()           <-
    {                          <- Acquires lock here
        // Critical section code goes here
    }                          <- Releasees lock here
}
```

Mutex

- End of Slides

End of Slides