Taking Advantage of Android

Douglas Starnes
Author / Speaker

@poweredbyaltnet https://douglasstarnes.com

Coroutines and Android

Channels share data between coroutines

Basics of using coroutines with Android

- Special user interface needs

Android Jetpack

- Collection of libraries implementing best practices for Android development
- `Lifecycle`
- `ViewModel`

WorkManager

- `CoroutineWorker`

Kotlin Channels

- Data may make itself available over time
- Channels stream data between coroutines
- Data is sent by the producer, and received by the consumer
- Hot streams constantly produce data, even if there is no consumer
- Cold streams are dormant until there is a consumer (Kotlin Flow)

Channels

```
val scope = CoroutineScope(Dispatchers.Default)
val channel = Channel<Int>()

scope.launch {
  for (i in 1..10) {
    channel.send(i)
  }
}

scope.launch {
  while (true) {
    val j = channel.receive()
    println("$j")
  }
}
```

Channels

```
val scope = CoroutineScope(Dispatchers.Default)
val channel = Channel<Int>()

scope.launch {
  for (i in 1..10) {
    channel.send(i)
  }
}

scope.launch {
  while (true) {
    val j = channel.receive()
    println(“$j”)
  }
}
```

Channels

```
val scope = CoroutineScope(Dispatchers.Default)
val channel = Channel<Int>()

scope.launch {
  for (i in 1..10) {
    channel.send(i)
  }
}

scope.launch {
  while (true) {
    val j = channel.receive()
    println("$j")
  }
}
```

Channels

```
val scope = CoroutineScope(Dispatchers.Default)
val channel = Channel<Int>()

scope.launch {
  for (i in 1..10) {
    channel.send(i)
  }
}

scope.launch {
  while (true) {
    val j = channel.receive()
    println(“$j”)
  }
}
```

Channels

```
val scope = CoroutineScope(Dispatchers.Default)
val channel = Channel<Int>()

scope.launch {
  for (i in 1..10) {
    channel.send(i)
  }
  channel.close()
}

scope.launch {
  while (true) {
    val j = channel.receive()
    println("$j")
  }
}
```

Channels

```kotlin
val scope = CoroutineScope(Dispatchers.Default)
val channel = Channel<Int>()

scope.launch {
  for (i in 1..10) {
    channel.send(i)
  }
  channel.close()
}

scope.launch {
  while (!channel.isClosedForReceive) {
    val j = channel.receive()
    println(“$j”)
  }
}
```

Channels

```
val scope = CoroutineScope(Dispatchers.Default)
val channel = Channel<Int>()

scope.launch {
  for (i in 1..10) {
    channel.send(i)
  }
  channel.close()
}

scope.launch {
  for (j in channel) {
    println("$j")
  }
}
```

Channels

```
val scope = CoroutineScope(Dispatchers.Default)
runBlocking {
  val channel = scope.produce<Int> {
    for (i in 1..10) {
      send(i)
    }
    close()
  }

  scope.launch {
    channel.consumeEach {
      println(“$it”)
    }
  }
}
```

Channels

```
val scope = CoroutineScope(Dispatchers.Default)
runBlocking {
  val channel = scope.produce<Int> {
    for (i in 1..10) {
      send(i)
    }
    close()
  }

  scope.launch {
    channel.consumeEach {
      println("$it")
    }
  }
}
```

Channels

```
val scope = CoroutineScope(Dispatchers.Default)
runBlocking {
  val channel = scope.produce<Int> {
    for (i in 1..10) {
      send(i)
    }
    close()
  }

  scope.launch {
    channel.consumeEach {
      println("$it")
    }
  }
}
```

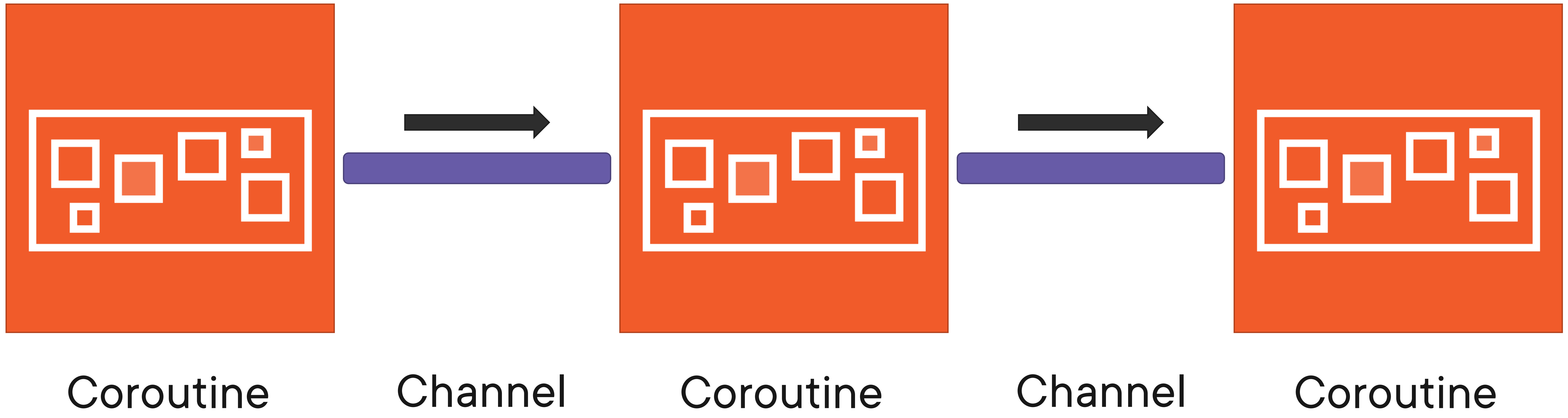
Channels

```
val scope = CoroutineScope(Dispatchers.Default)
runBlocking {
  val channel = scope.produce<Int> {
    for (i in 1..10) {
      send(i)
    }
    close()
  }

  scope.launch {
    channel.consumeEach {
      println(“$it”)
    }
  }
}
```

Channels

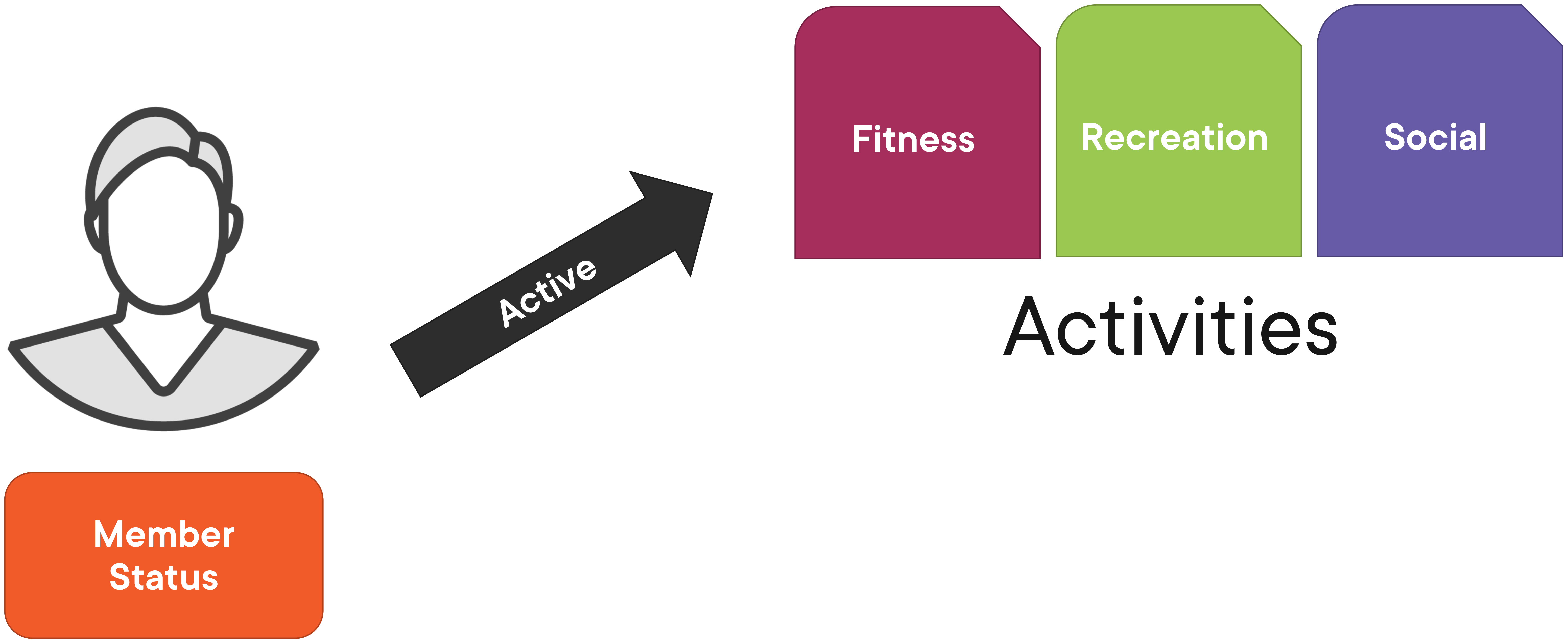
```
val scope = CoroutineScope(Dispatchers.Default)
runBlocking {
  val channel = scope.produce<Int> {
    for (i in 1..10) {
      send(i)
    }
    close()
  }

  scope.launch {
    channel.consumeEach {
      println("$it")
    }
  }
}
```

Pipeline



Data streams are unpredictable. They do not always produce values consistently.

Carved Rock Fitness Center Check-in



Check-in without Channels

Non-active

Fitness

Recreation

Social

Android Jetpack

The Lifecycle library enables lifecycle aware components

The `Fragment` **and** `AppCompatActivity` **are already configured**

Lifecycle aware components expose the `lifecycleScope`

The `ViewModel` **persists UI state through lifecycle changes**

The `ViewModel` **exposes the** `viewModelScope`

`WorkManager` Review

Android service for scheduling background tasks

Work is defined by extending the `Worker` class

Kotlin extensions such as `workDataOf` and `OneTimeWorkRequestBuilder`

The `CoroutineWorker` class supports coroutines

NEW!

`CoroutineWorker`

Much like the `Worker` **class**

Constructor accepts the same parameters, override `doWork`**, returns a** `Result`

In `CoroutineWorker`**,** `doWork` **is a suspend method**

All work is scheduled on a coroutine, instead of a thread

A cancelled or stopped work request will cancel all coroutines

`CoroutineWorker` Context

By default `CoroutineWorker` **will use** `Dispatchers.Default`

For a different context, use the `withContext` **builder**

CoroutineWorker

```
class MyCoroutineWorker(context: Context, params: WorkerParameters)
  : CoroutineWorker(context, params) {
  suspend fun doWork() {
    // work to be done on the coroutine
    return Result.success()
  }
}
```

CoroutineWorker

```
class MyCoroutineWorker(context: Context, params: WorkerParameters)
  : CoroutineWorker(context, params) {
  suspend fun doWork() {
    // work to be done on the coroutine
    return Result.success()
  }
}
```

CoroutineWorker

```
class MyCoroutineWorker(context: Context, params: WorkerParameters)
  : CoroutineWorker(context, params) {
  suspend fun doWork() {
    withContext(Dispatchers.IO) {
      // work to be done on the coroutine
      return Result.success()
    }
  }
}
```

Summary

Channels

- Share data by communicating between coroutines
- The `produce` builder manages a channel
- A pipeline is a series of channels

Coroutines in Android

- Avoid using `runBlocking`
- Use the `Main` dispatcher for UI access

Android Jetpack

- `Lifecycle`, `ViewModel`
- Both provide a `CoroutineScope`

`CoroutineWorker`

- Schedules `WorkManager` work on coroutines

Thank you!