Handling Coroutine Errors

Douglas Starnes
Author / Speaker

@poweredbyaltnet https://douglasstarnes.com

Handling Errors

This module is about when things go wrong in coroutines

Cancelling coroutines

- Single coroutines
- Entire `CoroutineScope`
- Cooperative code
- Preventing cancellation

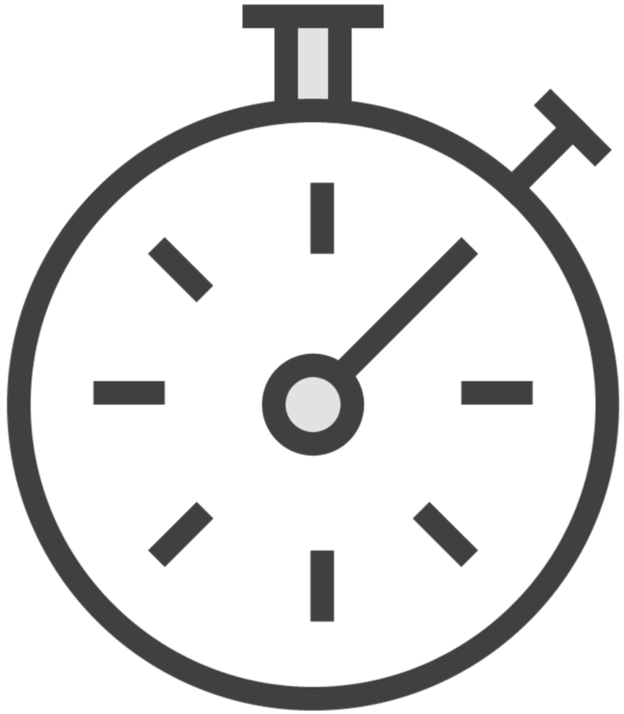
Timeouts

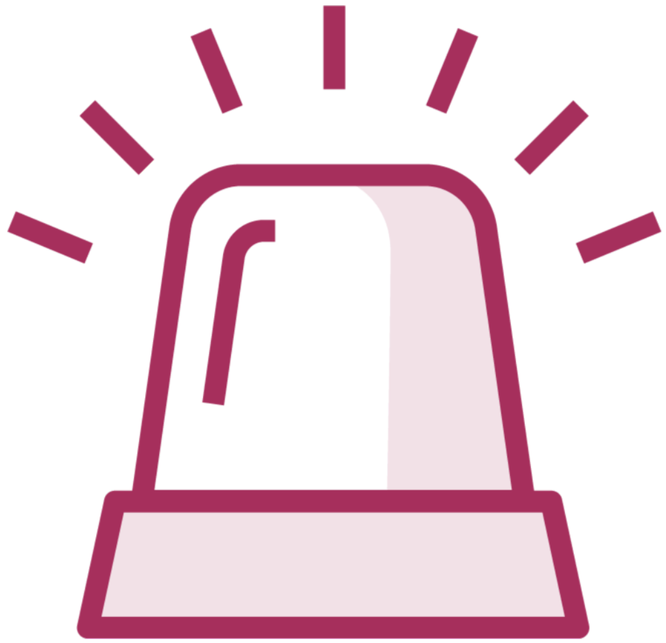
- Cancels a coroutine after a set amount of time

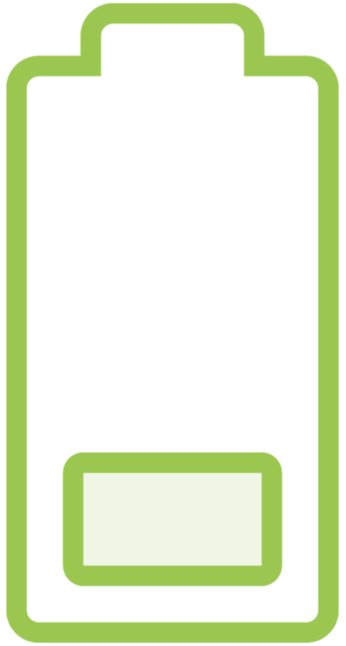
Exception handling

- Difference between `launch` and `async`

When Good Coroutines Go Bad







Cancelling Coroutines

Cancellation requires a `Job`

Call the `cancel` **method on the** `Job`

Cancelling a coroutine throws a `CancellationException`

Use `try-finally` **to clean up resources**

Cancelling Multiple Coroutines

Call `cancelChildren` **on a** `Job` **to cancel all children of a coroutine**

Call `cancel` **to cancel all children of a** `CoroutineScope`

Use the `CoroutineScope` **instead of a** `Job` **referencing the** `CoroutineScope`

Cooperative Code

Code must be cooperative to be cancelled

Prevent cooperative code from being cancelled by wrapping it in `withContext`

The `NonCancellable` **context element will prevent it from being cancelled**

Timeouts

Runs for up to a maximum duration, then cancels the coroutine

Call the `withTimeout` **function**

Timeouts throw a `TimeoutCancellationException`

`TimeoutCancellationException` **is logged, while** `CancellationException` **is not**

Catching Exceptions with `launch`

Exceptions thrown by `launch` **are not handled in a** `try-catch`

Use a `CoroutineExceptionHandler`

The `CoroutineExceptionHandler` **takes a** `CoroutineContext`**, and the thrown** `Exception`

The `CoroutineExceptionHandler` **is a parameter to** `launch`

A Tale of Two Coroutine Builders

The method used to handle exceptions in coroutines depends on how the coroutine was created

Crash Course in **async**

```
fun main() {
  runBlocking {
    val deferred = async {
      echoAsync(“I’m inside an async coroutine”)
    }
    val result = deferred.await()
    println(result)
  }
}

suspend fun echoAsync(message: String): String {
  delay(500)
  return message
}
```

Crash Course in **async**

```
fun main() {
  runBlocking {
    val deferred = async {
      echoAsync("I'm inside an async coroutine")
    }
    val result = deferred.await()
    println(result)
  }
}

suspend fun echoAsync(message: String): String {
  delay(500)
  return message
}
```

await **is a** suspend **function**

Catching Exceptions with `async`

Exceptions thrown by `async` can be handled with `try-catch`

Call `await` on the deferred object in the `try` block

Handle the exception in the `catch` block

This is the same way `try-catch` is used with regular blocking calls

Summary

Cancelling coroutines

- Cancel a single coroutine
- Cancel an entire `CoroutineScope`
- Cooperative code
- Preventing cancellation

Timeouts

Exception handling

- Use `CoroutineExceptionHandler` with `launch`
- Use `try-catch` with `async`