UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

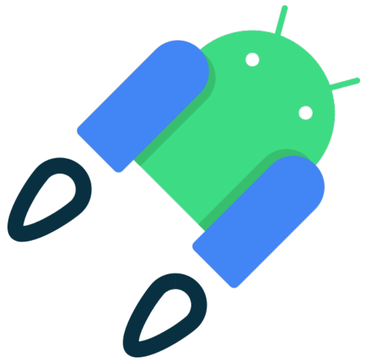
Coroutines e Acesso a Internet

QXD0102 - Desenvolvimento para Dispositivos Móveis

Prof. Bruno Góis Mateus (brunomateus@ufc.br)

Conteúdo

- Coroutines
- Android e Coroutines
- Conectando a Internet usando Coroutines



Coroutines

Coroutines

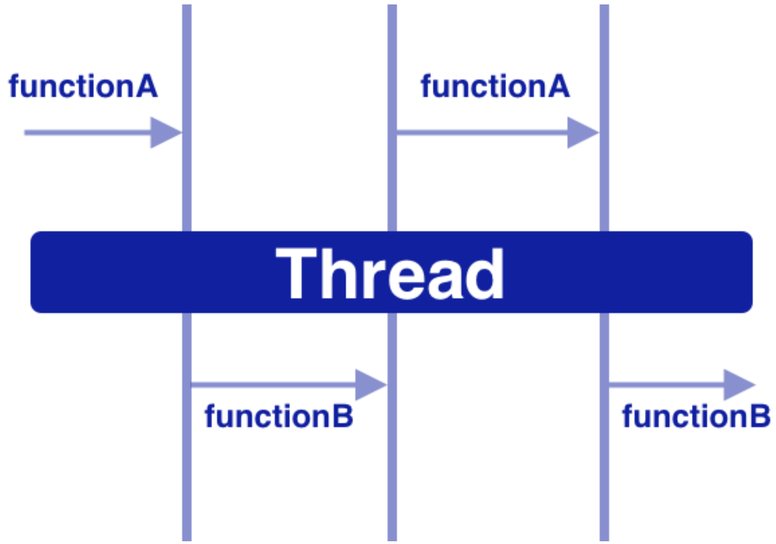
Introdução

- No últimos anos a popularidade das coroutines cresceu bastante
- Adotada por linguagens como JavaScript, C#, Python, Ruby, Go
- Adicionada inicialmente no Kotlin 1.1 (experimental)
  - Tornou-se estável na versão 1.3 do Kotlin
- Primeira linguagem a explorar esse conceito foi Simula em 1967

Coroutines

Introdução

- Coroutines = Co + Routines
  - Cooperação + Rotinas
  - Funções que cooperam uma com as outras



Coroutines

Introdução

- Coroutines são construídos por meio de funções comuns que possuem duas novas operações além de call e return
  - suspend
    - pausa a execução de uma coroutine, salvando as variáveis locais
  - resume
    - continua a execução de uma coroutine que está suspensa

Coroutines

Introdução

- Não substituem o uso de Thread
- Thread são gerenciadas pelo sistema operacional
- Coroutines são gerenciadas pelo usuário
- Vantagens
  - Melhor performance em multitasking
  - Melhor legibilidade
    - Código asíncrono escrito no “formato" síncrono

Coroutines

Exemplo prático

```
val formatter = DateTimeFormatter.ISO_LOCAL_TIME
val time = { formatter.format(LocalDateTime.now()) }

suspend fun getValue(): Double {
    println("entering getValue() at ${time()}")
    delay(3000)
    println("leaving getValue() at ${time()}")
    return Math.random()
}

fun main() {
    runBlocking {
        val num1 = async { getValue() }
        val num2 = async { getValue() }
        println("result of num1 + num2 is ${num1.await() + num2.await()}")
    }
}
```

```
entering getValue() at 22:52:25.025
entering getValue() at 22:52:25.03
leaving getValue() at 22:52:28.03
leaving getValue() at 22:52:28.032
result of num1 + num2 is 0.8416379026501276
```

Coroutines

- suspend
- CoroutineScope
- Coroutine builders
- Job
- Dispatchers

Coroutines

suspend

- Palavra chave que indica que uma função pode ser pausada e resumida
- Podemos executar tarefas longas e esperar pelo resultado sem causa bloqueio
- Só podem ser chamadas em uma coroutine ou em outra suspend function

```
suspend fun sample() {

}
```

Coroutines

Coroutine Scope

- Mantém o rastreio de qualquer coroutine criado usando um builder (launch or async)
- Fornece a habilidade de cancelar um coroutine há qualquer momento
- Uma coroutine não pode ser iniciado sem um escopo
- É notificado sempre que acontece uma falha

Coroutines

Coroutine builders

- Extension functions que criam e iniciam coroutines
- Ponte entre o mundo normal e o mundo das suspend functions
- Não são suspend e por isso podem ser acessados em funções normais

Coroutines

Coroutine builders

- launch
  - A maneira mais simples de criar uma coroutine
  - Inicia uma nova coroutine sem bloquear a thread atual
  - Retorna uma referência para a coroutine (Job)
  - Usado quando quando não precisamos esperar o resultado (“fire and forget”)

```
scope.launch(Dispatchers.IO){
 //New Coroutine is created
  uploadLogData()
}
```

Coroutines

Coroutine builders

- async
  - Podemos iniciar uma coroutine e recuperar o seu resultado usando a suspend function await
    - Não podemos usá-lo dentro de funções normais devido a chamada await
  - Retorna um Deffered<T>
  - Normalmente utilizado quando queremos paralelizar tarefas

```
suspend fun getUser(): User = viewModelScope{
    val result = async(Dispatchers.IO) {
        fetchUserData()
    }
    result.await()
}
```

Coroutines

Coroutine builders

- runningBlock
  - Bloqueia a thread em que é chamado até enquanto a coroutine não terminar
  - Executa a coroutine no contexto da thread em que é chamado
  - Em caso de interrupção na thread, uma InterruptedException é lançada

```
fun sampleRunBlocking(){
    println("First statement of runBlocking")
    runBlocking {
        delay(3000L)
        println("Middle  statement of runBlocking")
    }
    println("Last statement of runBlocking")
}
```

Coroutines

Job

- Um objeto que identifica unicamente uma coroutine e gerencia seu ciclo de vida
- Cancelável
- Termina ao completar a tarefa ou devido a cancelamento ou falha

Coroutines

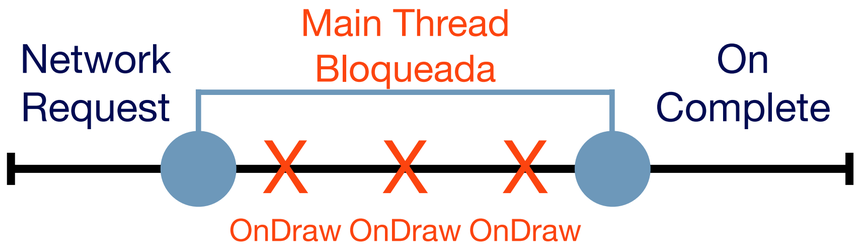
Dispatchers

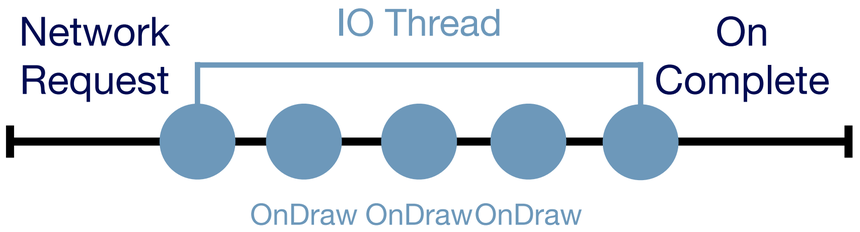
- Especificam em que thread a operação de ser executada
- No contexto do Android são 3: Main, Default e IO.

Android e Coroutines

Android e Coroutines

- Por padrão todos os componentes de um aplicativo Android são executados na mesma Thread, a Main Thread. (Single Thread)
- É essencial não bloquearmos a Main Thread
  - Única responsável pelas atualizações da interface com o usuário
  - Pode ocasionar o aparecimento do famoso ANR





Android e Coroutines

Multi-Thread

- É uma abordagem necessária em alguns cenários:
  - “Parsear um json”, escrever no banco de dados, recuperar informações via redes de computadores
- Desafiador porque lidar com Threads não é simples
- Inicialmente AsyncTasks era o padrão, passou para RxJava e hoje são as Coroutines

Android e Coroutines

O Callback pattern

- Usado para executar tarefas de longa duração sem bloquear a Main Thread
- Quando a tarefa é completada, ela chama a callback que informa o resultado a Main Thread

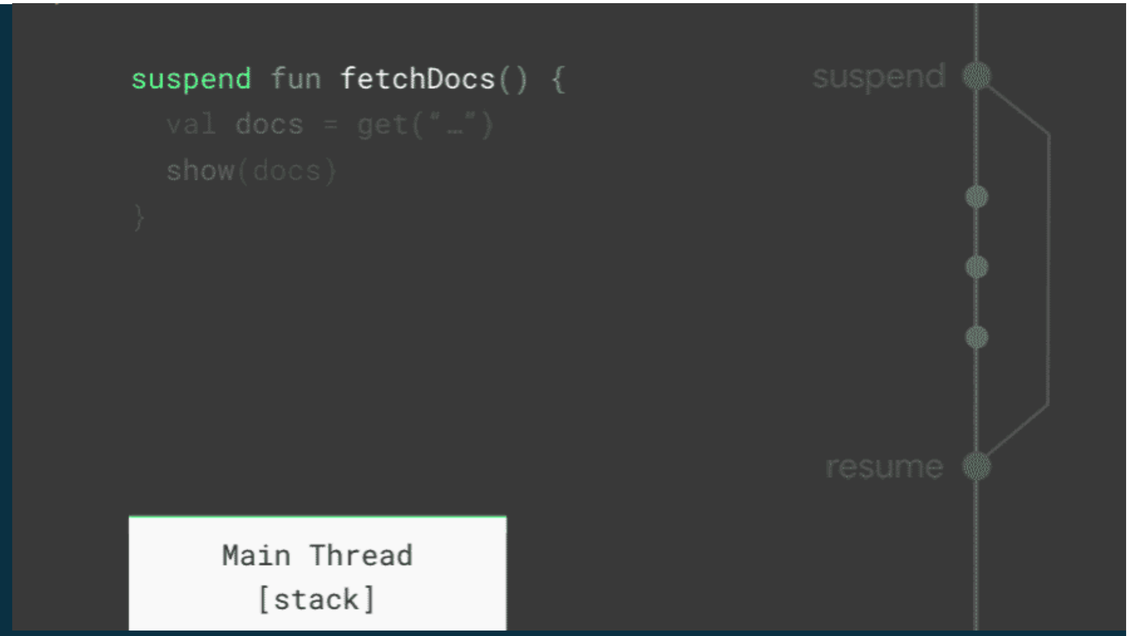
```
// Slow request with callbacks
@UiThread
fun makeNetworkRequest() {
    // The slow network request runs on another thread
    slowFetch { result ->
        // When the result is ready, this callback will get the result
        show(result)
    }
    // makeNetworkRequest() exits after calling
    slowFetch without waiting for the result
}
```

Boa solução. No entanto, o uso mais intenso de callback torna o código mais difícil de ler e consequentemente de alterar. Não permite o uso de exceptions.

Android e Coroutines

```
// Slow request with coroutines
@UiThread
suspend fun makeNetworkRequest() {
    // slowFetch is another suspend function so instead of
    // blocking the main thread  makeNetworkRequest
    // will `suspend` until the result is ready
    val result = slowFetch()
    // continue to execute after the result is ready
    show(result)
}

// slowFetch is main-safe using coroutines
suspend fun slowFetch(): SlowResult { ... }
```



Kotlin Coroutines permite a conversão de código baseado em callback para o código em estilo sequencial

Android e Coroutines

Coroutines

- Leves computacionalmente
- Mecanismo de cancelamento embutido
- Poucas chance de vazamento de memória
- Bibliotecas do Jetpack dão suporte a coroutines
- Main-safety

Android e Coroutines

Main-safety com Coroutines

- Suspend functions bem escritas sempre podem ser chamadas na Main Thread
- Ao marcar uma função como suspend, não estamos informando que a função deve executar em background
  - Podemos fazer com que uma Coroutine não execute na Main Thread
  - Devemos usar diferente Dispatchers

Android e Coroutines

Dispatcher	Descrição	Operações
Dispatchers.MAIN	Main Thread do Android. Usada para interagir com a UI e realizar operações leves	• Chamar funções suspend • Chamar funções ligadas a UI • Atualizar LiveData
Dispatchers.IO	Otimizado para operações de entrada e saída no disco e na rede fora da Main Thread	• Banco de dados* • Ler e escrever arquivos • Operações em rede**
Dispatchers.Default	Otimizado para operações de alto custo de CPU fora da Main Thread	• Ordenar listas • Parsear Json • DiffUtils

*Room prover de forma automática main-safety ao utilizarmos suspend functions, RxJava ou LiveData
** Bibliotecas como Retrofit e Volley gerenciam suas próprias Thread e por essa razão não exigem código main-safety quando se usa coroutines

Android e Coroutines

```
// Dispatchers.Main
suspend fun fetchDocs() {
    // Dispatchers.Main
    val result = get("developer.android.com")
    // Dispatchers.Main
    show(result)
}

suspend fun get(url: String) =
    // Dispatchers.Main
    withContext(Dispatchers.IO) {
        // Dispatchers.IO
        /* perform blocking network IO here */
    }
    // Dispatchers.Main
}
```

Android e Coroutines

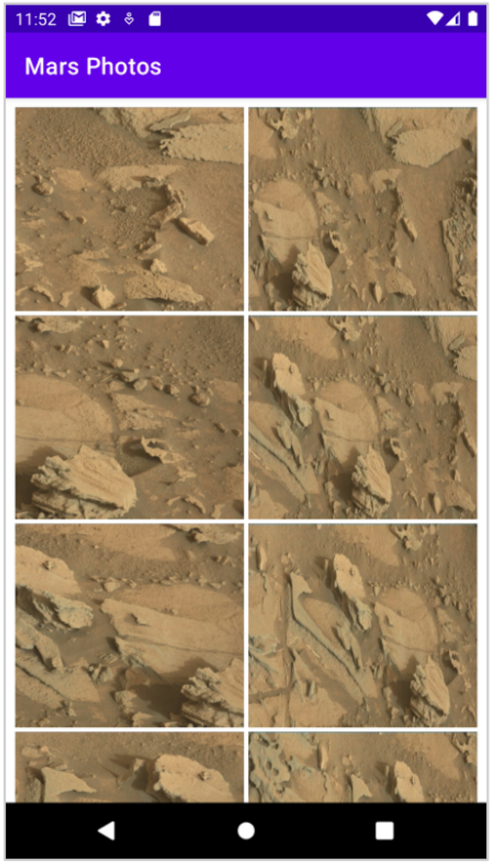
Onde devemos iniciar uma coroutine?

- Geralmente, faz sentido associar o CoroutineScope com uma tela
  - Evita trabalho extra com Activities ou Fragments que não são mais relevantes para o usuário
  - Ao sair de uma tela, o CoroutineScope cancela a coroutine
- Ao integrar Coroutines com os components da arquitetura Android, tipicamente iremos iniciar coroutines no ViewModel

Conectando a Internet usando Coroutines

Conectando a Internet usando Coroutines

- Codelabs:
  - Recuperando dados na Internet
  - Carregando e mostrando imagens vindas da Internet



Por hoje é só