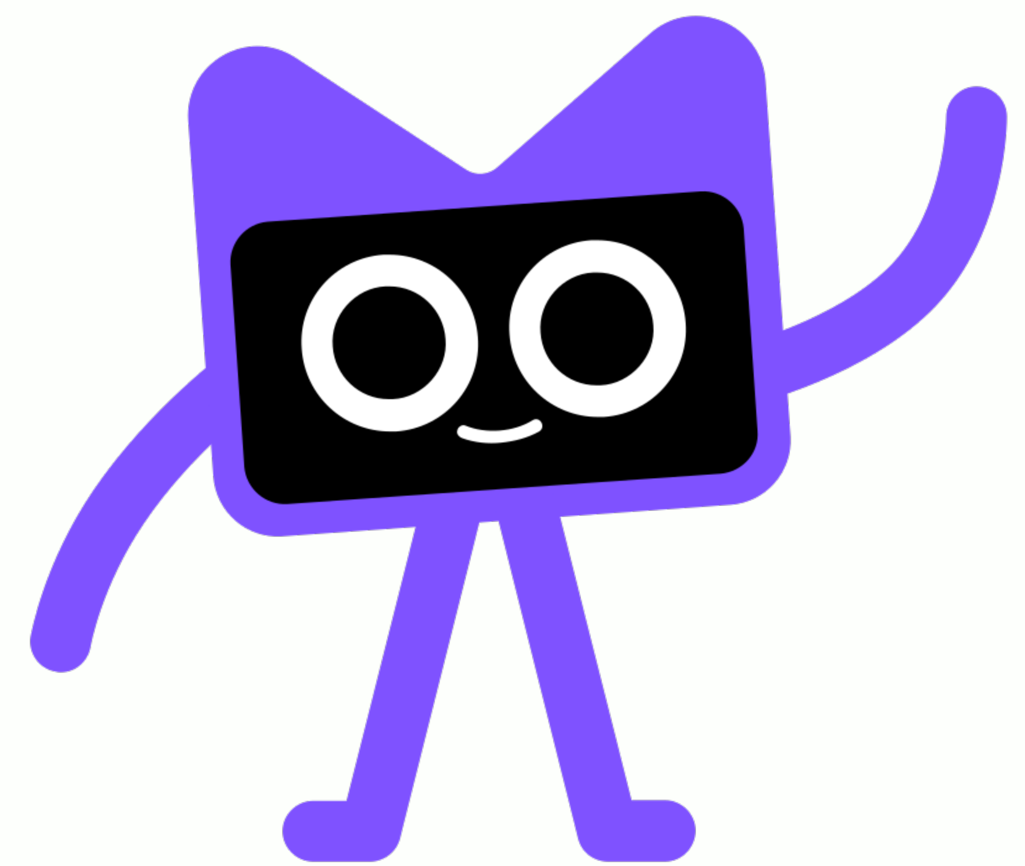
Coroutines e Acesso à Internet

QXD0276 - Desenvolvimento de Software para Dispositivos Móveis

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



Conteúdo

- Introdução
- Fundamentos das Coroutines
- Android e Coroutines
- Retrofit
- Ktor

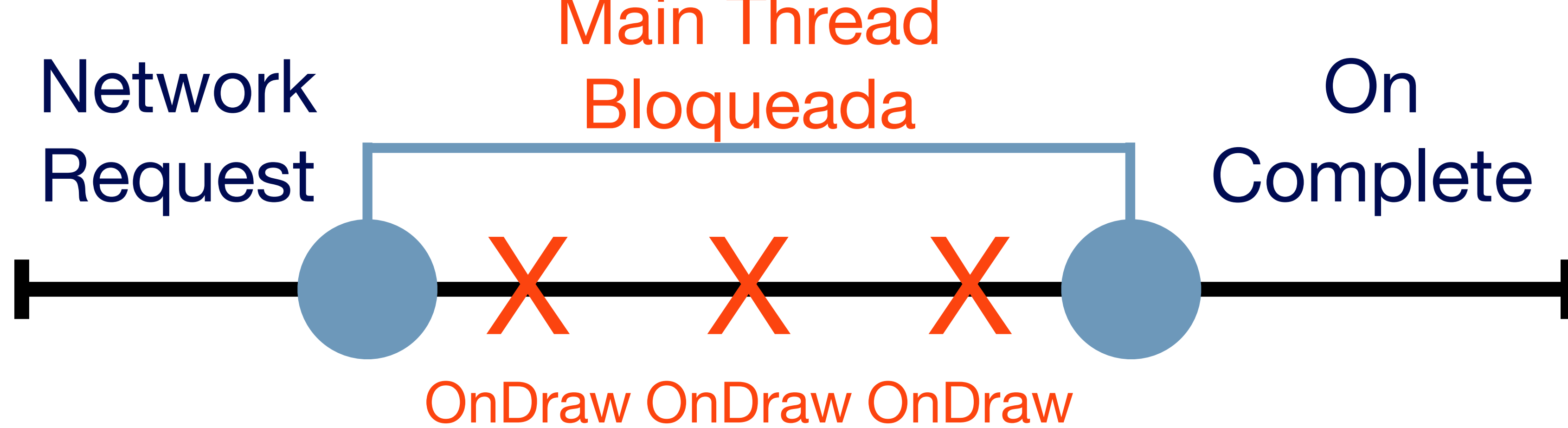
Introdução

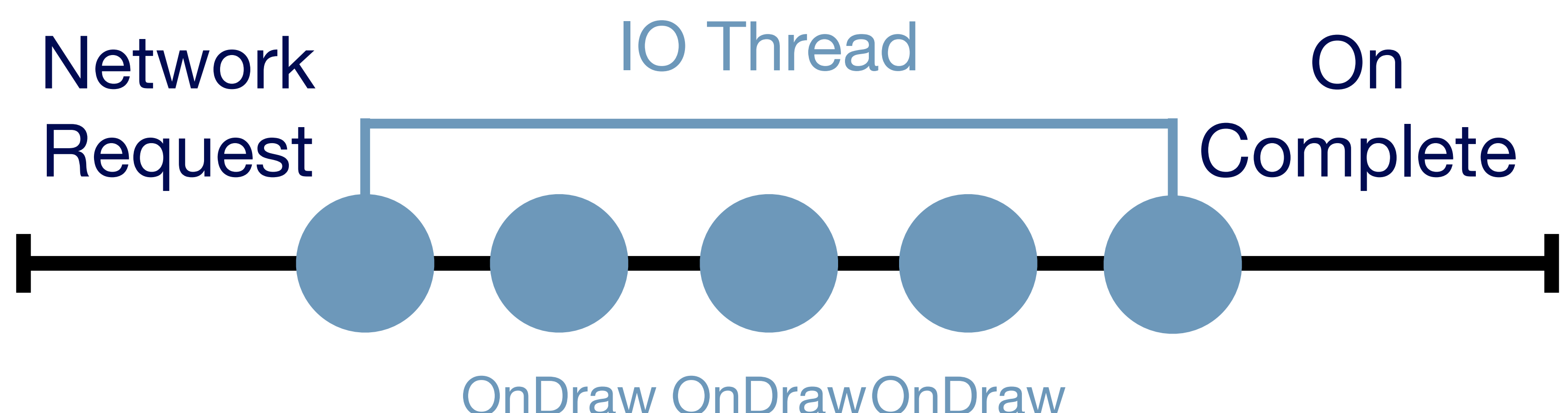
Como desenvolvedores Android temos uma missão: NUNCA bloquear a thread principal

Intrudução

- 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

Introdução





Introdução

- Uma abordagem multi-thread é necessária em alguns cenários:
  - “Parsear um json”,
  - Escrever no banco de dados
  - Recuperar informações via rede (APIs REST)
- Desafiador porque lidar com Threads não é simples
- Inicialmente AsyncTasks era o padrão, passou para RxJava e hoje são as Coroutines

Introdução

Coroutines

- 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

Introdução

Coroutines

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

```
coroutine
    loop
        while some_condition
            some_action
        yield
```

Função suspensa

Função suspensa

Função A

Thread

Função B

Função suspensa

Função suspensa

Coroutines

Introdução

- Coroutines são construídas por meio de funções comuns que possuem duas novas operações:
  - suspend
    - Pausa a execução de uma coroutine, salvando as variáveis locais
    - A pausa não bloqueia a execução da thread
  - 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

Fundamentos de Coroutines

Coroutines

```
fun main() {
    runBlocking {
        val windows = async(Dispatchers.IO) { order(Product.WINDOWS) }
        val doors = async(Dispatchers.IO) { order(Product.DOORS) }
        launch(Dispatchers.Default) {
            perform("laying bricks")
            launch { perform("install ${windows.await().description}") }
            launch { perform("install ${doors.await().description}") }
        }
    }
}
```

Coroutine builder / Coroutine scope

Coroutine builder

Suspend function

Dispatchers

Job

Suspend function

Coroutine builder

Coroutines

O Poder do suspend

- suspend indica que uma função pode ser suspensa
  - Pode ser pausada e retomada em um momento posterior
  - Permite que a thread faça outras coisas enquanto aguarda o resultado

Coroutines

O Poder do suspend

```
suspend fun order(item: Product): Product {
    println("ORDER EN ROUTE  >>> ${item.description}")
    delay(item.deliveryTime)
    println("ORDER DELIVERED >>> ${item.description} have arrived.")
    return item
}
```

Você só pode chamar uma função suspend de dentro de outra função suspend ou de dentro de um Coroutine Scope

Coroutines

O escopo

- Toda Coroutine deve ser lançada dentro de um escopo (Scope)
  - Define o ciclo de vida da Coroutine
  - Fornece a habilidade de cancelar um coroutine há qualquer momento
  - É notificado sempre que acontece uma falha

Quando o escopo é cancelado, todas as Coroutines lançadas dentro dele são automaticamente cancelada

Coroutines

🚀 Lançando coroutines

- Para iniciar uma Coroutine, usamos construtores (builders)
  - Ponte entre o mundo normal e o mundo das suspend functions
  - Extension functions que criam e iniciam coroutines
  - Não são suspend e por isso podem ser acessados em funções normais

Coroutines

Coroutine builders - launch

- 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”

Coroutines

Coroutine builders - launch

```
val windows = launch { order(Product.WINDOWS) }
val doors = launch { order(Product.DOORS) }
launch {            Job
    perform("laying bricks")
    perform("install ${windows.description}")
    perform("install ${doors.description}")
}
```

Coroutines

Coroutine builders - Job

- Um objeto que identifica unicamente uma coroutine
    - Gerencia seu ciclo de vida
- Representa uma unidade de trabalho cancelável
- Termina ao completar a tarefa ou devido a cancelamento ou falha

Coroutines

Coroutine builders - async

- async
  - Inicia uma coroutine e permite esperar por seu resultado para continuar
    - Usamos função await para suspender o código até recuperar o resultado
  - Retorna um Deffered<T> que é uma subclasse de Job
  - Usado quando você precisa do valor da tarefa para continuar (ex: buscar dados de duas APIs e combiná-los).
  - Normalmente utilizado quando queremos paralelizar tarefas

Coroutines

Coroutine builders - async

```
val windows = async { order(Product.WINDOWS) }
val doors = async { order(Product.DOORS) }
launch {                Deffered
    perform("laying bricks")
    perform("install ${windows.await().description}”)
    perform("install ${doors.await().description}")
}                                    Suspende até recuperar o resultado
```

Coroutines

Coroutine builders - runBlocking

- runBlocking
  - Bloqueia a thread em que é chamado enquanto a coroutine não terminar
  - Em caso de interrupção na thread, uma InterruptedException é lançada

Coroutines

runningBlock

```
runBlocking {
    val windows = async { order(Product.WINDOWS) }
    val doors = async { order(Product.DOORS) }
    launch {
        perform("laying bricks")
        perform("installing ${windows.await().description}")
        perform("installing ${doors.await().description}")
    }
}
```

Coroutines

Dispatchers

- O Despachante decide em qual thread o código da Coroutine será executado

Dispatcher	Uso	Exemplo de Tarefa
Dispatchers.Main	`Thread Principal. Apenas para atualizar a UI.`	`textView.text = "Dados carregados"`
Dispatchers.IO	`Tarefas de Input/Output (I/O).`	`Chamadas de Rede e Disco (arquivos/banco de dados).`
Dispatchers.Default	`Tarefas ligadas à CPU.`	`Ordenação de listas grandes, cálculos complexos.`

Coroutines

Dispatchers

```
runBlocking {
    val windows = async(Dispatchers.IO) { order(Product.WINDOWS) }
    val doors = async(Dispatchers.IO) { order(Product.DOORS) }
    launch(Dispatchers.Default) {
        perform("laying bricks")
        launch { perform("install ${windows.await().description}") }
        launch { perform("install ${doors.await().description}") }
    }
}
```

Coroutines

Dispatchers

- É perfeitamente possível executar tarefas que exigem muito da CPU ao dispatcher IO e tarefas que exigem muito de E/S ao dispatcher Default
- Dispatcher Default: possui um pool de threads com uma thread para cada núcleo do processador
  - Se o seu computador tiver 8 núcleos, seu pool de threads terá 8 threads
  - Ótima opção na maioria dos casos, especialmente quando as coroutines precisam de tempo de CPU
- Dispatcher IO: tem 64 threads por padrão
  - Se o seu computador tiver mais de 64 núcleos, o número de threads será o número de núcleos
  - Útil quando você deseja que operações de E/S sejam executadas concorrentemente

Coroutines

withContext

- Utilizado para trocar o contexto de uma coroutine
- Bastante utilizado para mudar o dispatchers de um coroutine

Coroutines

withContext

```
runBlocking {
    launch(Dispatchers.IO) {
        val windows = order(Product.WINDOWS)
        withContext(Dispatchers.Default) {
            perform("install ${windows.description}")
        }
    }
    launch(Dispatchers.IO) {
        val doors = order(Product.DOORS)
        withContext(Dispatchers.Default) {
            perform("install ${doors.description}")
        }
    }
    launch(Dispatchers.Default) {
        perform("laying bricks")
    }
}
```

Coroutines

Cancelamento

- Uma coroutine é cancelada quando a função cancel() é invocada a partir da sua Job
  - É possível chamá-la a manualmente
  - Ela pode ser invocada automaticamente por meio da propagação de cancelamento
- Quando uma coroutine é cancelada, uma exceção CancellationException é lançada

Coroutines

Cancelamento Cooperativo

- O cancelamento de coroutine é cooperativo.
    - Coroutines só reagem ao cancelamento quando cooperam:
        - suspendendo ou verificando explicitamente se houve cancelamento
            - isActive, ensureActive(), yield()
    - Sem isso, elas podem não ser canceladas

Coroutines

Cancelamento Cooperativo

- Ao ser cancelada, uma coroutine continua sua execução até atingir um ponto onde pode ser suspensa
    - Conhecido como ponto de suspensão
    - Se a ela for suspensa nesse ponto, a função que a suspendeu verifica se ela foi cancelada
    - Se tiver sido, a coroutine para e lança uma exceção CancellationException

Coroutines

Cancelamento Cooperativo

```
// ❌ Calculate without cooperating with the cancellation
suspend fun generateThumbnails(
    images: List<File>
): List<Thumbnail> {
    val result = mutableListOf<Thumbnail>()
    for (image in images) {
        val bytes = image.readBytes()
        result += createThumbnail(bytes)
    }
    return result
}
```

Operação passada, provavelmente envolvendo CPU e IO. Pode dificultar o cancelamento

Coroutines

Cancelamento Cooperativo

```
suspend fun generateThumbnails(
    images: List<File>
): List<Thumbnail> = withContext(Dispatchers.Default) {
    val result = mutableListOf<Thumbnail>()
    for (image in images) {
        ensureActive()
        val bytes = withContext(Dispatchers.IO) {
            image.readBytes()
        }
        result += createThumbnail(bytes)
    }
    result
}
```

Lança uma CancellationException se o escopo tiver sido cancelado

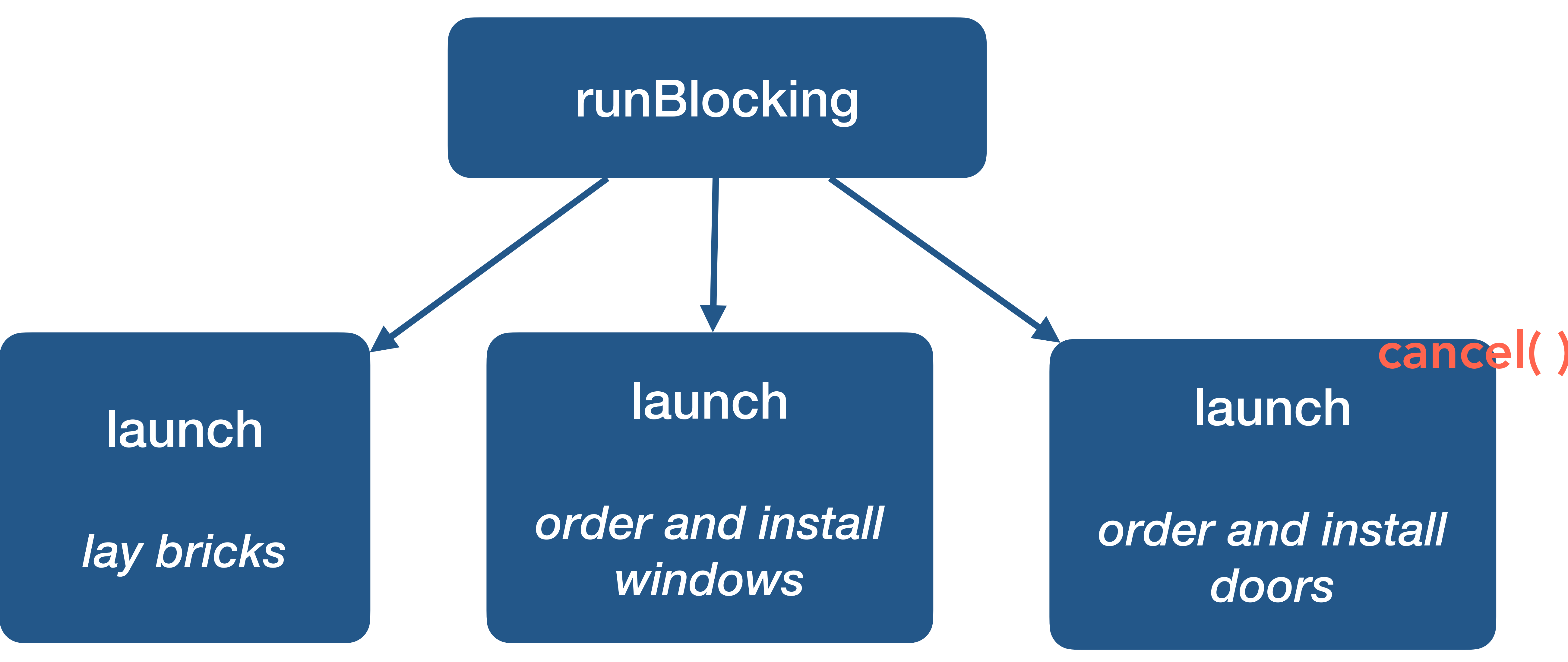
Coroutines

Concorrência estruturada

- Uma exceção lançada por uma coroutine pode levar ao cancelamento da coroutine mãe
- Ao receber uma exceção, a coroutine se cancela e propaga a exceção para sua ancestral
  - A coroutine ancestral, cancela todos seus filhos e a si mesmo
    - Continua a propagação

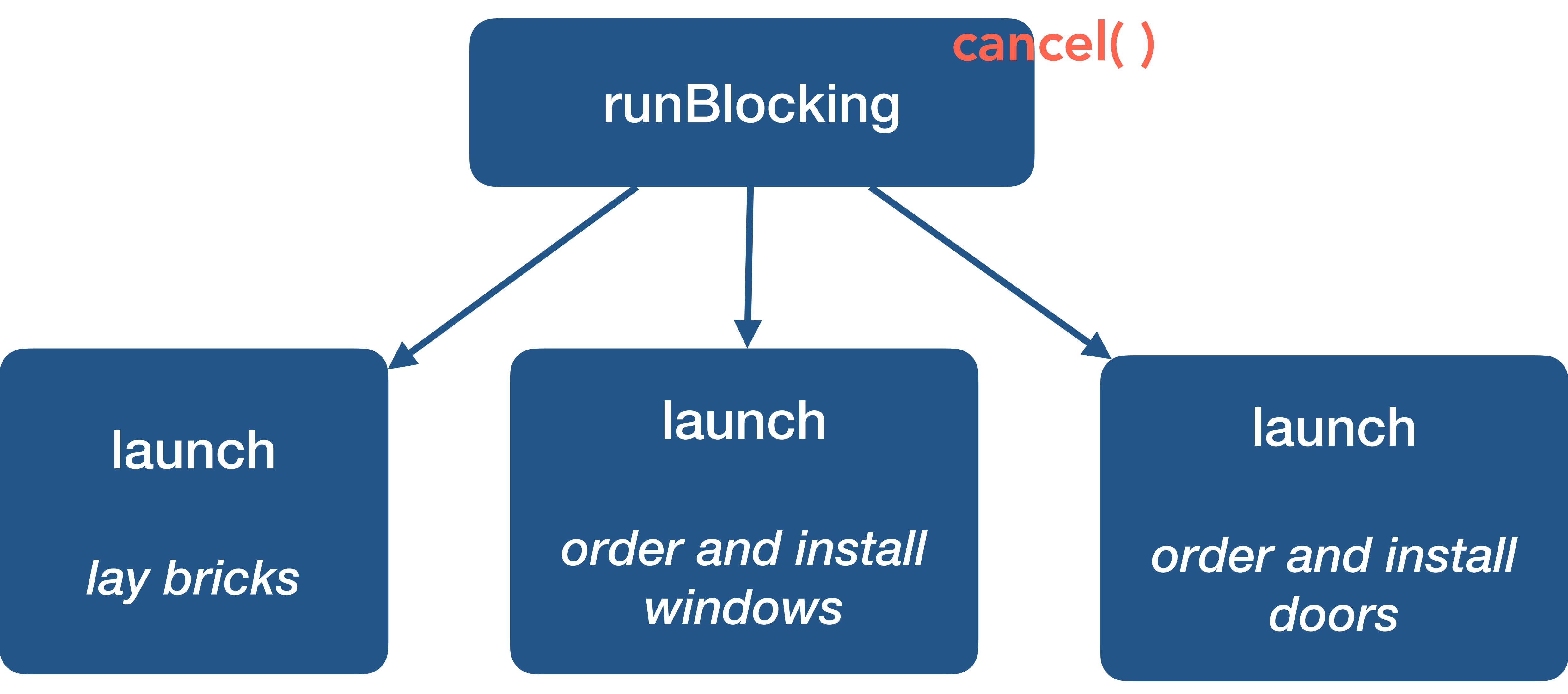
Coroutines

Concorrência estruturada



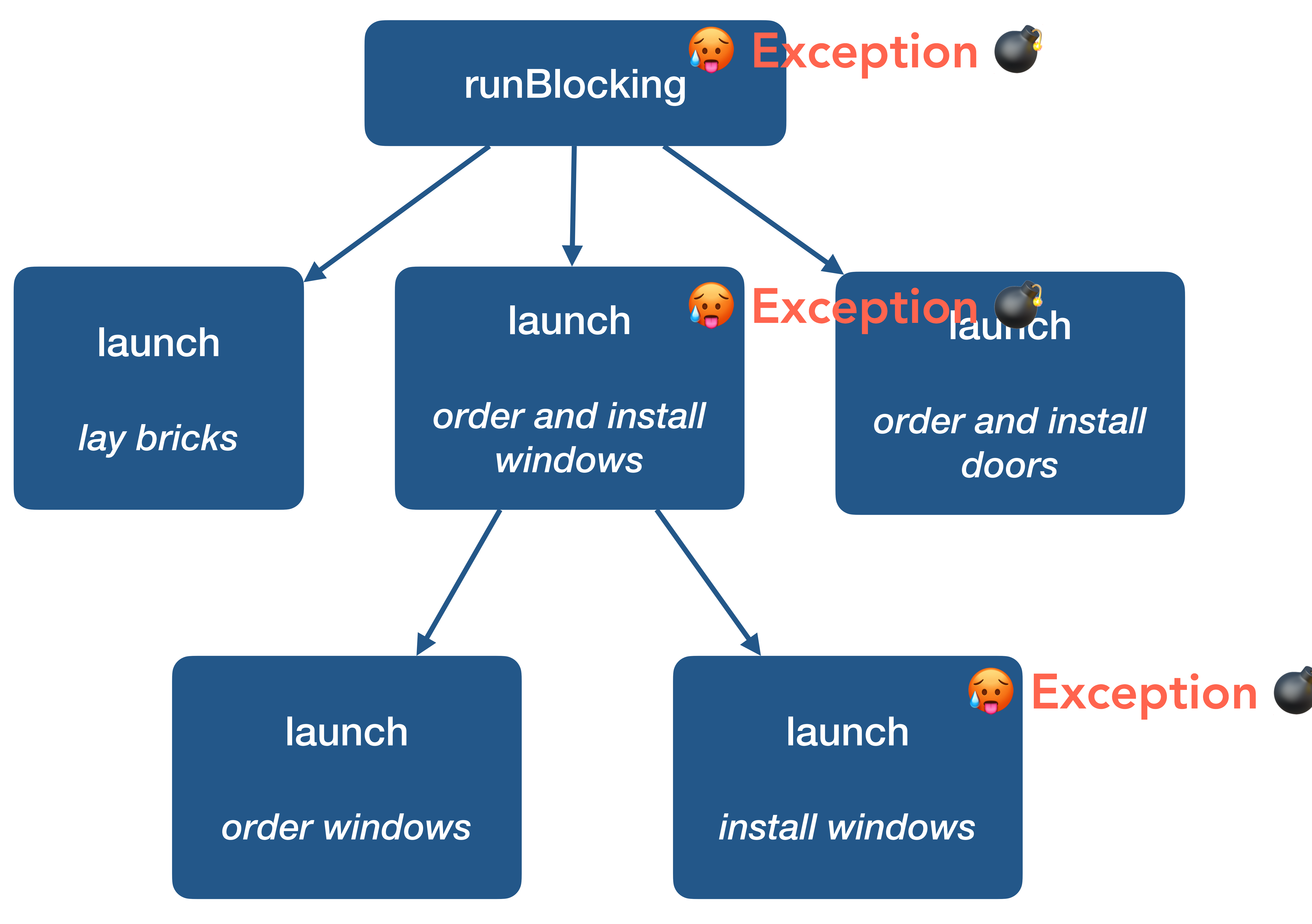
Coroutines

Concorrência estruturada



Coroutines

Concorrência estruturada



Coroutines

Concorrência estruturada

- Ao utilizar um SupervisorJob/supervisorScope, os irmão são isolados
- Outra opção é tratar a exceção localmente

```
scope.launch {
    try {
        fetch()
    } catch (e: IOException) {
        showError(e)
    }
}
```

Coroutines

Concorrência estruturada

```
fun main(): Unit = runBlocking {
    val scope = CoroutineScope(SupervisorJob())
    scope.launch {
        delay(1000)
        throw Error("Some error")
    }
    scope.launch {
        delay(2000)
        println("Will be printed")
    }
    delay(3000)
    println(scope.isActive)
}
```

Coroutines

Concorrência estruturada

```kotlin
fun main(): Unit = runBlocking {
    supervisorScope {
        launch {
            delay(1000)
            throw Error("Some error")
        }
        launch {
            delay(2000)
            println("Will be printed")
        }
    }
    println("Done")
}
```

Coroutines

Bonus: timeout

- Quando precisar lidar com timeout temos duas opções
  - withTimeout ou withTimeoutOrNull

```
fun main() {
    CoroutineScope(Main).launch {
        try {
            val result = withTimeout(2000) { fetchUserData() }
            println(result)
        } catch (e: TimeoutCancellationException) {
            println("Operation timed out")
        }
    }
}
```

Android e Coroutines

Android e Coroutines

Quando usar Coroutines?

- Operações de rede:
  - Obter dados de uma API
- Operações de banco de dados:
  - Ler e gravar dados em um banco de dados
- Computações complexas:
  - Executar tarefas que podem bloquear a thread principal
- Concorrência:
  - Executar várias tarefas simultaneamente sem bloqueio

Android e Coroutines

O Problema do Ciclo de Vida no MVVM

- Onde devemos lançar as Coroutines no Android?
  - CoroutineScope não é lifecycle-aware por padrão
  - Se lançarmos a Coroutine na Activity ou Composable, ela pode continuar rodando mesmo depois que a tela for destruída (causando crash ou vazamento de memória)
  - A Lógica de Negócio (Rede/DB) deve estar no ViewModel, mas o ViewModel não tem um Scope nativo

O Jetpack fornece um Scope pronto para uso no ViewModel

For ViewModel utilities in Compose, use implementation("androidx.lifecycle:lifecycle-viewmodel-compose:$lifecycle_version").
For Lifecycle utilities in Compose, use implementation("androidx.lifecycle:lifecycle-runtime-compose:$lifecycle_version").

Android e Coroutines

O viewModelScope

- É um CoroutineScope que está disponível automaticamente em todo ViewModel da AAC (Android Architecture Components)
- Cancelamento Automático:
  - Qualquer Coroutine lançada no viewModelScope será automaticamente cancelada quando o ViewModel for limpo
    - Quando a thread principal decidir que a tela não será mais usada

Android e Coroutines

O viewModelScope

```
class UserViewModel(
    private val repo: UserRepository
) : ViewModel() {

    val user = MutableStateFlow<User?>(null)

    fun loadUser() {
        viewModelScope.launch {
            user.value = repo.fetchUser()
        }
    }
}
```

Android e Coroutines

Combinando o Fluxo MVVM + Coroutines

- View (Composable): Dispara o evento (viewModel.loadData())
- ViewModel: Recebe o evento e lança a Coroutine no viewModelScope
- ViewModel: Chama o Repositório, que usa withContext(Dispatchers.IO)
- Repositório: Executa a tarefa em Background e retorna o resultado
- ViewModel: Recebe o resultado na thread principal e atualiza o UiState (_uiState.value = NewState)
- View (Composable): Observa o UiState e atualiza a interface

Boas práticas

Android e Coroutines

Boas práticas - Funções suspend devem ser chamadas na main thread

- Se uma classe estiver executando operações bloqueantes em uma coroutine, ela é responsável por mover a execução para fora da thread principal
- Aplicável a todas as classes do seu aplicativo, independentemente da parte da arquitetura em que ela se encontra
- Torna seu aplicativo mais escalável
    - Classes que chamam suspend functions não precisam se preocupar com qual Dispatcher usar para cada tipo de trabalho
    - Essa responsabilidade recai sobre a classe que executa a tarefa

Android e Coroutines

Boas práticas - Funções suspend devem ser chamadas na main thread

```
class NewsRepository(private val ioDispatcher: CoroutineDispatcher) {
   suspend fun fetchLatestNews(): List<Article> {
        withContext(ioDispatcher) { /* ... implementation ... */ }
    }
}
class GetLatestNewsWithAuthorsUseCase(
    private val newsRepository: NewsRepository,
    private val authorsRepository: AuthorsRepository
) {
   suspend operator fun invoke(): List<ArticleWithAuthor> {
        val news = newsRepository.fetchLatestNews()
        val response: List<ArticleWithAuthor> = mutableEmptyList()
        for (article in news) {
            val author = authorsRepository.getAuthor(article.author)
            response.add(ArticleWithAuthor(article, author))
        }
        return Result.Success(response)
    }
}
```

Android e Coroutines

Boas práticas - Injete Dispatchers

- Ao usar withContext, não fixe o dispatcher utilizado em código

```
// DO inject Dispatchers
class NewsRepository(
    private val defaultDispatcher: CoroutineDispatcher = Dispatchers.Default
) {
    suspend fun loadNews() = withContext(defaultDispatcher) { /* ... */ }
}

// DO NOT hardcode Dispatchers
class NewsRepository {
    // DO NOT use Dispatchers.Default directly, inject it instead
    suspend fun loadNews() = withContext(Dispatchers.Default) { /* ... */ }
}
```

Android e Coroutines

Boas práticas - O viewModel deve criar coroutines

- Funções Composable não devem acionar diretamente nenhuma coroutine para executar lógica de negócios
- Delegue essa responsabilidade ao ViewModel
- Facilita o teste da sua lógica de negócios
  - ViewModels podem ser testados com testes unitários, em vez de usar testes de instrumentação
- Suas coroutines sobreviverão automaticamente a mudanças de configuração

Android e Coroutines

Boas práticas - O viewModel deve criar coroutines

```kotlin
// DO create coroutines in the ViewModel
class LatestNewsViewModel(
    private val getLatestNewsWithAuthors: GetLatestNewsWithAuthorsUseCase
) : ViewModel() {

    private val _uiState =
MutableStateFlow<LatestNewsUiState>(LatestNewsUiState.Loading)
    val uiState: StateFlow<LatestNewsUiState> = _uiState

    fun loadNews() {
        viewModelScope.launch {
            val latestNewsWithAuthors = getLatestNewsWithAuthors()
            _uiState.value = LatestNewsUiState.Success(latestNewsWithAuthors)
        }
    }
}
```

Android e Coroutines

Boas práticas - A camada de dados deve expor funções suspend e flow

- As classes das camadas de dados e de negócios geralmente expõem:
  - Funções para realizar chamadas únicas
    - Utilizar suspend functions
  - Funções que notificam sobre alterações de dados ao longo do tempo
    - Devem retornar objetos do tipo Flow
- Permite que o chamador, geralmente a camada de UI, controle a execução e o ciclo de vida do que ocorre nessas camadas e cancele quando necessário

Android e Coroutines

Boas práticas - A camada de dados deve expor funções suspend e flow

```
// Classes in the data and business layer expose
// either suspend functions or Flows
class ExampleRepository {
    suspend fun makeNetworkRequest() { /* ... */ }

    fun getExamples(): Flow<Example> { /* ... */ }
}
```

Android e Coroutines

Boas práticas - Crie coroutines canceláveis

- O cancelamento em coroutines é cooperativo
  - Quando o Job é cancelado, a coroutine não é cancelada até que seja suspensa ou verifique se foi cancelada
  - Certifique-se de que ela seja cancelável

Android e Coroutines

Boas práticas - Crie coroutines canceláveis

```
someScope.launch {
    for(file in files) {
        ensureActive() // Check for cancellation
        readFile(file)
    }
}
```

Android e Coroutines

Boas práticas - Fique atento às exceções

- Exceções não tratadas em coroutines podem causar a falha do seu aplicativo
- Se houver probabilidade de ocorrerem exceções, capture-as no corpo de quaisquer coroutines criadas com viewModelScope ou lifecycleScope

Android e Coroutines

Boas práticas - Fique atento às exceções

```
class LoginViewModel(
    private val loginRepository: LoginRepository
) : ViewModel() {

    fun login(username: String, token: String) {
        viewModelScope.launch {
            try {
                loginRepository.login(username, token)
                // Notify view user logged in successfully
            } catch (exception: IOException) {
                // Notify view login attempt failed
            }
        }
    }
}
```

Retrofit

Retrofit

O que é?

- Biblioteca HTTP client type-safe criada pela Square.
- Simplifica o processo de fazer requisições de rede para APIs RESTful
  - Converte chamadas HTTP em interfaces Kotlin/Java.
- Trabalha bem com OkHttp (baixo nível) e com conversores (Moshi, Gson, kotlinx.serialization).
- Forte integração com coroutines por meio de suspend functions

Retrofit

Por que usar?

- Segurança de tipos:
  - Garante a segurança de tipos, permitindo que você defina a estrutura das suas respostas de API usando data classes do Kotlin.
- Definição de API simplificada:
  - É possível definir os endpoints de uma API usando interfaces e anotações
- Conversão automática de dados:
  - Lida com a serialização e desserialização de dados
  - Por exemplo: JSON ou XML entre seu aplicativo e a API usando bibliotecas de conversão como Gson ou Moshi

Retrofit

Por que usar?

- Integração com OkHttp:
    - Utiliza o OkHttp, um cliente HTTP poderoso e eficiente, para lidar com as requisições de rede
    - Compatível com interceptors (autenticação, logging, caching)
- Amplo uso na indústria → muitos exemplos e suporte

Retrofit

Passo a passo

- Instalar as dependências
- Criar um client OkHttp
  - Opcional, porém necessário acesso a interceptors, logging …
- Criar um objeto Retrofit
- Definir modelos
- Criar uma API de Serviço
- Realizar as chamadas a API
- Tratar respostas

Retrofit

Dependências - libs.versions.toml

```
[versions]
retrofit = "3.0.0"
loggingInterceptor = “4.12.0"
converterKotlinxSerialization = "3.0.0"
kotlinPluginSerialization = "2.2.21"
kotlinxSerializationJson = “1.6.0"
```

Retrofit

Dependências - libs.versions.toml

```
[libraries]
kotlinx-serialization-json = { module =
"org.jetbrains.kotlinx:kotlinx-serialization-json", version.ref =
"kotlinxSerializationJson" }
#Retrofit
retrofit = { group = "com.squareup.retrofit2", name = "retrofit",
version.ref = "retrofit" }
converter-kotlinx-serialization = { group =
"com.squareup.retrofit2", name = "converter-kotlinx-serialization",
version.ref = "converterKotlinxSerialization" }
logging-interceptor = { group = "com.squareup.okhttp3", name =
"logging-interceptor", version.ref = "loggingInterceptor" }
```

Retrofit

Dependências - libs.versions.toml

```
plugins]
kotlin-serialization = { id =
"org.jetbrains.kotlin.plugin.serialization", version.ref =
"kotlinPluginSerialization"
```

Retrofit

Dependências - Gradle

```
plugins {
    alias(libs.plugins.kotlin.serialization)
}

dependencies {
   implementation(libs.retrofit)
   implementation(libs.kotlinx.serialization.json)
   implementation(libs.converter.kotlinx.serialization)
   implementation(libs.logging.interceptor)
}
```

Retrofit

Configurando o OkHttpClient

```
fun provideOkHttpClient(): OkHttpClient {
    val logging = HttpLoggingInterceptor().apply {
        level = HttpLoggingInterceptor.Level.BODY
    }

    return OkHttpClient.Builder()
        .connectTimeout(15, TimeUnit.SECONDS)
        .readTimeout(30, TimeUnit.SECONDS)
        .writeTimeout(15, TimeUnit.SECONDS)
        .addInterceptor(logging)
        .build()
}
```

Retrofit

Criando o Retrofit

Deve terminar com /

```
val retrofit =Retrofit.Builder()
        .baseUrl("https://api.example.com/")
        .client(okHttp)
        .addConverterFactory(MoshiConverterFactory.create())
        .build()
```

Depende da biblioteca escolhida

Retrofit

Criando os modelos com Moshi

```
import com.squareup.moshi.JsonClass

@JsonClass(generateAdapter = true)
data class Pokemon(
    val id: Int,
    val name: String,
    val type: String
)
```

Retrofit

Definindo a interface API

```
interface PokemonApi {
    @GET("pokemon")
    suspend fun listPokemons(): List<Pokemon>

    @GET("pokemon/{id}")
    suspend fun getPokemon(@Path("id") id: Int): Pokemon

    @POST("pokemon")
    suspend fun createPokemon(@Body newPokemon: Pokemon): Pokemon
}

val api = retrofit.create(PokemonApi::class.java)
```

Criando o serviço

Retrofit

Repositório

```
class PokemonRepository(private val api: PokemonApi) {
    suspend fun getAll(): Result<List<Pokemon>> {
        return try {
            val data = api.listPokemons()
            Result.success(data)
        } catch (e: Exception) {
            Result.failure(e)
        }
    }
}
```

Result é forma simples para propagar erros

Retrofit

Tratamento no viewModel

```
class PokemonViewModel(private val repo: PokemonRepository): ViewModel() {

  private val _uiState = MutableStateFlow<UiState>(UiState.Loading)
  val uiState: StateFlow<UiState> = _uiState

  fun load() {
   viewModelScope.launch {
       _uiState.update { current -> current.copy(status = Status.LOADING) }
       _uiState.update { current ->
          val res = pokemonRepository.getAll()
            current.copy(
               pokemons = res.getOrDefault(current.loadedCharacters),
               status = if (res.isSuccess) Status.READY else Status.ERROR
            )
        }
     }
  }
}
```

Retrofit

Tratando erros

```
try {
    val result = api.listPokemons()
} catch (e: HttpException) {
    // Erro HTTP (4xx, 5xx) -> e.code(), e.response()
} catch (e: IOException) {
    // Problemas de rede / timeouts
} catch (e: Exception) {
    // Geral
}
```

Retrofit

Interceptadores e autenticação

```
class AuthInterceptor(private val tokenProvider: () -> String?) :
Interceptor {
    override fun intercept(chain: Interceptor.Chain): Response {
        val token = tokenProvider()
        val request = chain.request().newBuilder().apply {
            token?.let { header("Authorization", "Bearer $it") }
            header("Accept", "application/json")
        }.build()
        return chain.proceed(request)
    }
}
```

Retrofit

Interceptadores e autenticação

```
val okHttp = OkHttpClient.Builder()
    .addInterceptor(AuthInterceptor { tokenStore.getToken() })
    .addInterceptor(HttpLoggingInterceptor().apply { level = BODY })
    .build()
```

Ktor

Ktor

O que é?

- Biblioteca multiplataforma criada pela JetBrains
- Ideal para projetos Kotlin puro / KMP / Android
- Fortemente integrada com coroutines
- Modular e altamente customizável através de plugins
- Alternativa moderna e flexível ao Retrofit

Ktor

Aspectos chave

- Assíncrono com uso Coroutines:
  - Construído com base nas coroutines
- Multiplataforma:
  - Funciona perfeitamente em Android, iOS, Desktop e Web, ideal para projetos KMP
- Kotlinx.Serialization:
  - Integra-se com kotlinx.serialization para facilitar a conversão de respostas JSON em classes de dados Kotlin

Ktor

Passo a passo

- Instalar as dependências
- Criar um client Http
    - Opcional, porém necessário acesso a interceptors, logging …
- Instalar plugins
- Definir modelos
- Realizar as chamadas a API
- Tratar respostas

Ktor

Dependências - libs.versions.toml

```
[versions]
kotlinPluginSerialization = "2.2.21"
ktor = "3.3.0"
```

Ktor

Dependências - libs.versions.toml

```
[libraries]
ktor-client-core = { module = "io.ktor:ktor-client-core",
version.ref = "ktor" }
ktor-client-android = { module = "io.ktor:ktor-client-android",
version.ref = "ktor" }
ktor-content-negotiation = { module = "io.ktor:ktor-client-content-
negotiation", version.ref = "ktor"}
ktor-serialization-kotlinx-json = { module = "io.ktor:ktor-
serialization-kotlinx-json", version.ref = "ktor"}
ktor-client-logging = { module = "io.ktor:ktor-client-logging",
version.ref = "ktor"}
```

Ktor

Dependências - libs.versions.toml

```
[plugins]
kotlin-serialization = { id =
"org.jetbrains.kotlin.plugin.serialization", version.ref =
"kotlinPluginSerialization" }
```

Ktor

Dependências - Gradle

```
plugins {
    alias(libs.plugins.kotlin.serialization)
}

dependencies {
   implementation(libs.ktor.client.core)
   implementation(libs.ktor.client.android)
   implementation(libs.ktor.content.negotiation)
   implementation(libs.ktor.serialization.kotlinx.json)
   implementation(libs.ktor.client.logging)
}
```

Ktor

Criando o HttpClient e instalando os plugins

```
val client = HttpClient(Android) {
    install(ContentNegotiation) {
        json(
            Json {
                ignoreUnknownKeys = true
                prettyPrint = true
            }
        )
    }
    install(Logging) {
        level = LogLevel.BODY
    }
    defaultRequest {
       url("https://mybaseurl/api/")
   }
}
```

Realiza a negociação de tipos de mídia entre o cliente e o servidor

Realizar a Serialização/desserialização do conteúdo

Plugin para *logging* do cliente HTTP

Determina o nível de *logging* do plugin

Ktor

Criando os modelos com Kotlinx Serialization

```
import kotlinx.serialization.Serializable

@Serializable
data class Pokemon(
    val id: Int,
    val name: String,
    val type: String
)
```

Ktor

Requisições GET/POST

```
suspend fun listPokemons(): List<Pokemon> =
    client.get(“https://api.example.com/pokemon”).body()

suspend fun getPokemon(id: Int): Pokemon =
    client.get("https://api.example.com/pokemon/$id").body()


suspend fun createPokemon(p: Pokemon): Pokemon =
    client.post("https://api.example.com/pokemon") {
        setBody(p)
    }.body()
```

Ktor

Repositório

```
class PokemonRepository(private val client: HttpClient) {

    suspend fun list(): Result<List<Pokemon>> =
        try {
            Result.success(client.get(BASE_URL + "pokemon").body())
        } catch (e: Exception) {
            Result.failure(e)
        }

}
```

Ktor

Tratamento no viewModel

```
class PokemonViewModel(private val repo: PokemonRepository): ViewModel() {

  private val _uiState = MutableStateFlow<UiState>(UiState.Loading)
  val uiState: StateFlow<UiState> = _uiState

  fun load() {
   viewModelScope.launch {
      _uiState.update { current -> current.copy(status = Status.LOADING) }
      _uiState.update { current ->
         val res = pokemonRepository.getAll()
           current.copy(
              pokemons = res.getOrDefault(current.loadedCharacters),
              status = if (res.isSuccess) Status.READY else Status.ERROR
           )
       }
    }
  }
}
```

Ktor

Tratamento de erros

```
suspend fun safeRequest(): Result<List<Pokemon>> {
    return try {
        val data = client.get("https://api.example.com/pokemon").body()
        Result.success(data)
    } catch (e: ClientRequestException) {
        // 4xx
        Result.failure(e)
    } catch (e: ServerResponseException) {
        // 5xx
        Result.failure(e)
    } catch (e: IOException) {
        // rede
        Result.failure(e)
    } catch (e: Exception) {
        Result.failure(e)
    }
}
```

Ktor

Autenticação

```
implementation("io.ktor:ktor-client-auth:$ktor_version")
```

```
val client = HttpClient(Android) {
    install(ContentNegotiation) {
        json()
    }
    install(Auth) {
      bearer {
        loadTokens {
            // Load tokens from a local storage
            BearerTokens("abc123", "xyz111")
        }
      }
    }
}
```

Ktor

Timeout e Retry

```
val client = HttpClient(Android) {
 install(HttpTimeout) {
    requestTimeoutMillis = 15000
    connectTimeoutMillis = 15000
 }

 install(HttpRequestRetry) {
    maxRetries = 3
    retryIf { _, response -> !response.status.isSuccess() }
 }
}
```

Referências

- Coroutine Essentials
- Coroutines Concurrency in Kotlin (vídeo)
- Mastering Kotlin Coroutines: Dispatchers, Jobs, and Structured Concurrency
- Kotlin Coroutines without magic: a practical guide to good and bad practices
- Mastering Kotlin Coroutines in Android
- Coroutines on Android (part I): Getting the background
- Documentação oficial do Android

Referências

- Consumindo API REST no Android com Retrofit em Kotlin — Parte 1
- Fetching Data from an API Using Ktor in Your Android App
- Consumindo REST API no Android com o Ktor

Por hoje é só