

# AULA 13

Gerenciamento de threads &  
Coroutines





**01**  
**THREADS**

**02**  
**COROUTINES**

**03**  
**ROOM DATABASE**

## GERENCIAMENTO DE THREADS

Quando um app é executado, o sistema cria uma thread de execução para ele chamada main thread. Ela é encarregada de despachar eventos da interface do usuário em geral. Todo nosso código, por padrão, é executado nessa thread.

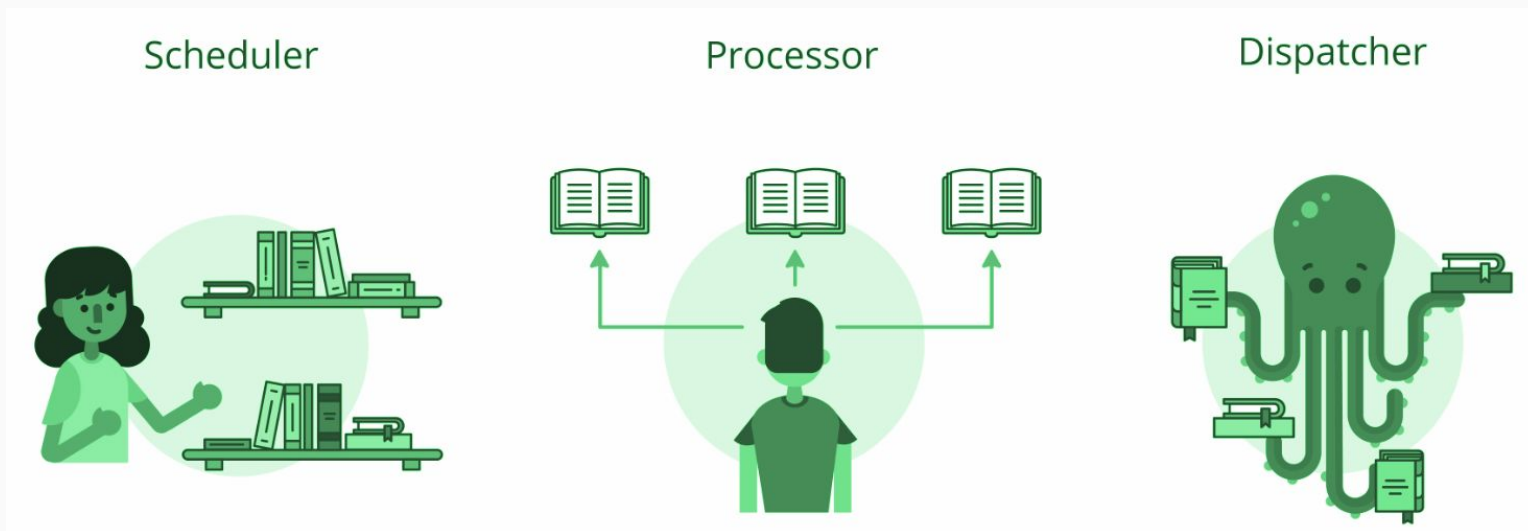


Atualmente, a maioria dos celulares tem vários processadores, cada um dos quais executa processos simultaneamente. Isso é chamado de **multiprocessamento**.

Para usar processadores com mais eficiência, o sistema operacional pode permitir que um aplicativo crie mais de um thread de execução dentro de um processo. Isso é chamado de **multi-threading**.

Uma analogia é a leitura de **vários livros ao mesmo tempo**, alternando entre os livros após cada capítulo, eventualmente terminando todos os livros. Contudo, não é possível ler mais de um livro ao mesmo tempo.

É preciso um pouco de infraestrutura para gerenciar todas essas tarefas:



**Scheduler:** leva em consideração as prioridades e garante que todos as threads sejam executadas e concluídas. Nenhum livro pode ficar na prateleira para sempre e acumular poeira, mas se um livro for muito longo ou puder esperar, pode demorar um pouco antes de ser enviado para você.

**Dispatcher:** configura threads, ou seja, envia livros que você precisa ler e especifica um contexto para que isso aconteça. Você pode pensar no contexto como uma sala de leitura especializada separada. Alguns contextos são melhores para operações de interface do usuário e alguns são especializados para lidar com operações de entrada / saída.



No Android, o thread principal é a thread que lida com todas as atualizações da UI. Ela também é a thread padrão. A menos que seu aplicativo alterne explicitamente ou use uma classe que é executada em uma thread diferente, tudo o que o seu aplicativo faz está na main thread.

A main thread deve funcionar para garantir uma ótima experiência para quem usa as apps.

Para que seu aplicativo seja exibido para o usuário sem nenhuma pausa visível, a thread principal deve atualizar a tela pelo menos a cada **16 ms**, ou a cerca de **60 quadros por segundo**.

Portanto, no Android, é essencial evitar o bloqueio do thread de UI. O bloqueio neste contexto significa que a main thread não está fazendo **nada** enquanto espera algo.

Muitas tarefas comuns levam **mais de 16 milissegundos** para serem executadas, como buscar dados da Internet, ler um arquivo grande ou gravar dados em um banco de dados.

Por padrão operações de entrada/saída (comunicação com banco de dados, request de informações externas etc) são bloqueadas para não acontecerem na Main Thread

## COROUTINES

Uma coroutine é um padrão de projeto de simultaneidade que você pode usar no Android para simplificar o código que é executado de forma assíncrona.

No Android, as coroutines ajudam a gerenciar tarefas de longa duração que podem bloquear a linha de execução principal e fazer com que seu app pare de responder.

**Job:** Basicamente, um *job* é **qualquer coisa que pode ser cancelada**. Cada *coroutine* tem um *job* e você pode usar o *job* para cancelar a *coroutine*.

Os *jobs* podem ser organizados em hierarquias pai-filho. O cancelamento de um *job* pai cancela imediatamente todos seus filhos.

**Dispatcher:** envia *coroutines* para rodar em várias threads. Por exemplo, `Dispatcher.Main` executa tarefas na thread principal e `Dispatcher.IO` descarrega tarefas de entrada e saída.



**Scope:** o escopo de uma *coroutine* define o contexto no qual ela é executada. Um escopo combina informações sobre o *job* e o dispatcher de uma *coroutine*.

**Android conference  
talks:**

**Kotlin Coroutines  
101**



Usaremos as coroutines para gerenciar a execução das tarefas relacionadas à gestão da base de dados do app What did I Learn.

Tarefas como inserir uma informação nova no banco, listar todos os itens armazenados serão executadas com auxílio das coroutines!

## POPULANDO NOSSA BASE DE DADOS

Nosso banco de dados está vazio. Para que ele seja inicializado com algumas informações criaremos uma `RoomDatabase.Callback`, definindo um novo comportamento para o método `onCreate()`.

## POPULANDO NOSSA BASE DE DADOS

Defina, a classe privada LearnedItemDatabaseCallback:

```
private class  
LearnedItemDatabaseCallback(private val scope:  
CoroutineScope) : RoomDatabase.Callback() {
```

O *scope* de uma coroutine define o **contexto** no qual a coroutine é executada.

Ele combina informações sobre o *job* e o *dispatcher* de uma coroutine. Os *scopes* monitoram as *coroutines*.

Quando você inicia uma *coroutine*, ela está "em um escopo", o que significa que você indicou qual escopo manterá o controle da coroutine.

Nessa classe, sobrescreva o método onCreate()

```
override fun onCreate(db: SQLiteDatabase) {  
    super.onCreate(db)  
    INSTANCE?.let { database ->  
        scope.launch {  
            populateDatabase(database.learnedItemDao())  
        }  
    }  
}
```

## POPULANDO NOSSA BASE DE DADOS

Na mesma classe, defina o método `populateDatabase` criando os itens aprendidos (você pode replicar as mesmas infos do método `getAll()`)

```
suspend fun populateDatabase(dao:
LearnedItemDao) {
    val itemLearned1 = LearnedItem(
        "Kotlin - Null safety",
        "O sistema de tipos de Kotlin visa...",
        UnderstandingLevel.HIGH
    )
    dao.insert(itemLearned1)
```



A palavra-chave ***suspend*** é a maneira de Kotlin de marcar uma função, ou tipo de função, como estando disponível para *coroutines*.

Quando uma *coroutine* chama uma função marcada com *suspend*, ao invés de bloquear a thread até que a função retorne, a coroutine suspende a execução até que o resultado esteja pronto.

## COROUTINE SUSPEND FUNCTION

*Blocked thread  
without coroutines*

kickOffLongWork()

*no work  
happens*

do more work

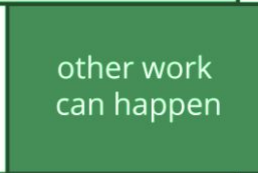
*Suspended thread  
with coroutines*

kickOffLongWork()

suspend

other work  
can happen

do more work



Atualize o método `getDatabase()` para que ele também receba um `scope`: `CoroutineScope` como parâmetro:

```
fun getDatabase(context: Context, scope:
CoroutineScope): LearnedItemsDatabase {
```

## POPULANDO NOSSA BASE DE DADOS

Ajuste a criação do banco de dados para que ela considere o callback que acabamos de criar:

```
fun getDatabase(context: Context, scope: CoroutineScope): LearnedItemsDatabase
    return INSTANCE ?: synchronized(this) {
        val instance = Room.databaseBuilder(
            context.applicationContext,
            LearnedItemsDatabase::class.java,
            "learned_item_database"
        )
        .addCallback(LearnedItemDatabaseCallback(scope))
        .build()
        INSTANCE = instance
        instance
    }
```

Iniciar o banco de dados:

```
val database =  
    LearnedItemsDatabase.getDatabase(this,  
        CoroutineScope(Dispatchers.IO))
```

Usamos *CoroutineScope(Dispatchers.IO)* para indicar que o escopo passado para a execução das atividades de inicialização do BD é do tipo entrada/saída.

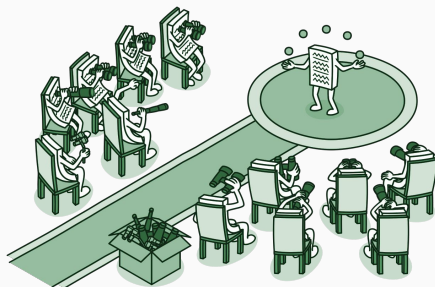
OBS: se tentarmos usar, por exemplo *Dispatchers.Main*, um erro será retornado em tempo de build, pois não deveríamos executar tarefas de entrada/saída sem nenhuma conexão com a UI nessa thread.

## LIVEDATA

LiveData é uma classe que armazena dados observáveis de forma alinhada com o ciclo de vida da aplicação.

O entendimento do ciclo de vida garante que o LiveData atualize apenas os observadores de componente do app que estão em um estado ativo válido, prevenindo erros e memory leaks.

Observer é um padrão de design comportamental que permite definir um mecanismo de assinatura para **notificar vários objetos sobre qualquer evento que aconteça com o objeto que eles estão observando.**



GET ALL

Ajuste o retorno do método `getAll()`, no `LearnedItemDao`, para que o retorno seja um `LiveData<List<LearnedItem>>`

```
@Query("SELECT * FROM learned_item ORDER  
BY item title ASC")  
fun getAll(): LiveData<List<LearnedItem>>
```

Desta forma, teremos um objeto "observável" com as informações que armazenamos no banco de dados



Para adicionar os dados recuperados do banco, no nosso adapter, vamos observar o LiveData, quando ele tiver resultados, atualizaremos o adapter:

```
val learnedItems =  
learnedItemsDao.getAll()  
learnedItems.observe(this, Observer {  
    adapter.data = it  
}))
```

## DATABASE INSPECTOR

Pixel 2 API 29 > com.wcc.whatdoilearn

Databases

learned\_item

Refresh table ☐ Live updates

	item_title	item_description	item_level	item_id
1	Kotlin - Null safety	O sistema de tipos de Kotlin visa elimin	2131034298	1
2	Layout editor	O Design Editor exibe o layout em vário	2131034299	2
3	Git	É um sistema de controle de versão dis	2131034300	3
4	GridView	É uma view especial que pode conter o	2131034298	4

learned\_item\_database

- learned\_item
  - item\_title : TEXT, NOT NULL
  - item\_description : TEXT, NOT NULL
  - item\_level : INTEGER, NOT NULL
  - item\_id : INTEGER, NOT NULL
- room\_master\_table

## **PALAVRAS CHAVE**

**COROUTINES**

**JOB**

**SCOPE**

**DISPATCHER**

**SUSPEND**

**MULTI THREADING**

**Marque a opção FALSA em relação às coroutines**

- ☐ Elas reduzem o tamanho da APK gerada
- ☐ Elas são executados de forma assíncrona.
- ☐ Elas podem ser executadas em um thread diferente do thread principal.
- ☐ Eles podem ser escritos e lidos como código linear.

## O que é uma suspend function?

- Uma função comum anotada com a palavra-chave suspend.
- Uma função que pode ser chamada dentro de uma coroutine.
- Enquanto uma função de suspensão está em execução, a thread de chamada é suspensa.
- As funções de suspensão devem sempre ser executadas em segundo plano.

**Qual é a diferença entre bloquear e suspender um thread? Marque apenas as verdadeiras:**

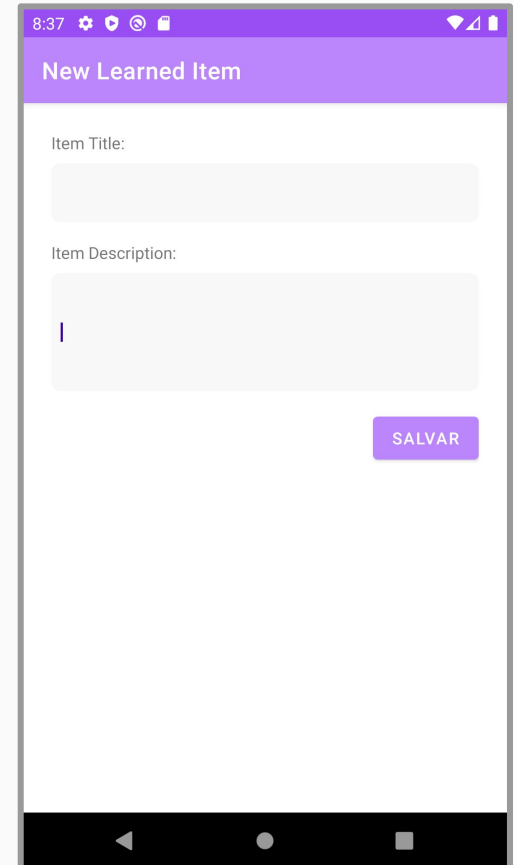
- ☐ Quando a execução é bloqueada, nenhum outro trabalho pode ser executado na thread bloqueado.
- ☐ Quando a execução é suspensa, a thread pode realizar outro trabalho enquanto aguarda a conclusão do trabalho transferido.
- ☐ Suspender apagar a thread de execução temporariamente
- ☐ Seja bloqueada ou suspensa, a execução ainda está aguardando o resultado da coroutine antes de continuar.

## DESAFIO - ADD ITEM APRENDIDO

Ao clicar no botão, se a os campos Item Title e Item Description estiverem preenchidos, salve essas informações no banco de dados.

Dica:

Para validar os campos, use o método isEmpty()



The screenshot shows a mobile application interface with a purple header bar containing the title "New Learned Item". Below the header, there are two text input fields. The first field is labeled "Item Title:" and the second field is labeled "Item Description:". Both fields are currently empty. At the bottom right of the form, there is a purple button with the text "SALVAR" in white capital letters. The top status bar of the phone shows the time as 8:37 and various system icons. The bottom navigation bar of the phone is visible at the very bottom.