

The background features a complex network of thin grey lines connecting various points, some of which are solid black dots. Scattered throughout are numerous triangles of different sizes and orientations, some outlined in grey and others as solid black shapes. The overall aesthetic is technical and modern.

PROGRAMA DEV VENTURE

Desenvolvimento Android

THREADS 01
No Android

COROUTINES 02
Multi thread

PROCESSOS DE IO 03
Acesso ao Room

REPOSITORY 04
Fonte de dados

LIVEDATA 05
Padrão Observable

VIEWMODEL 06
ViewModel

Aula 16

AGENDA



01

THREADS e PROCESSOS



PROCESSOS



Um processo é uma instância de uma aplicação que está sendo executada. Cada vez que uma aplicação é executada, um processo é criado para ela.


É uma estrutura de dados que representa o contexto de execução de um programa.



THREADS

Quando um app é executado, o sistema cria uma thread de execução chamada **main thread**. Todo nosso código, por padrão, é executado nessa thread. Ela é encarregada, inclusive, de enviar/receber **eventos da interface que o usuário interage**

Uma thread só pode executar uma instrução por vez.





MAIN THREAD

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.



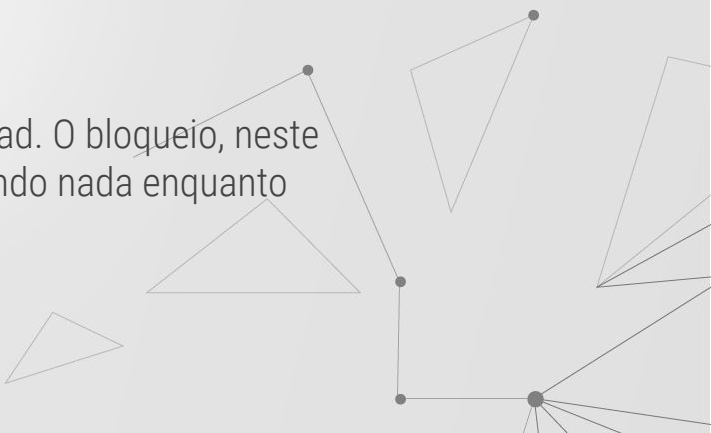


MAIN THREAD

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, é essencial evitar o bloqueio da main thread. O bloqueio, neste contexto, significa que a main thread não está fazendo nada enquanto espera algo.





MAIN THREAD

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





dojo / POKEDEX

- Observe a classe PokedexViewModel
 - Repare em como a requisição de rede é executada
 - Repare em como os dados são salvos no banco de dados
-





APPLICATION

Classe base para manter o estado global do aplicativo.

A classe Application, ou sua subclasse da classe Application, é instanciada antes de qualquer outra classe quando o processo do aplicativo.

Somente uma instância dela permanece em memória enquanto o aplicativo esta em execução, em foreground (primeiro plano) ou em background (segundo plano).






dojo / WHAT DID I LEARN

- Iniciaremos o banco de dados criando uma classe Application para nosso projeto

```
class WhatDidILearnApplication : Application() {  
    val database by lazy { LearnedItemDatabase.getDatabase(this) }  
}
```

- Atualize o manifest

```
<application  
    android:name=".WhatDidILearnApplication"  
    android:allowBackup="true"
```



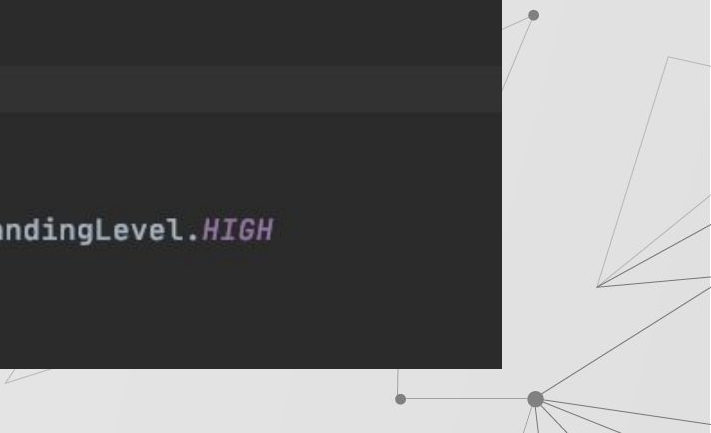


dojo / WHAT DID I LEARN

- Acesse uma instância do banco de dados e tente fazer uma operação de inserção:

```
val database = (application as WhatDidILearnApplication).database

database.
    learnedItemDao()
    .insert(
        LearnedItem(
            name = "Teste",
            description = "teste",
            understandingLevel = UnderstandingLevel.HIGH
        )
    )
```



02

COROUTINES






COROUTINES

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.

As coroutines podem ser consideradas threads leves. No entanto, uma coroutine não está ligada a nenhuma thread em particular. Ele pode suspender sua execução em uma thread e retomar em outra.



COROUTINES

**Android conference
talks:**

**Kotlin Coroutines
101**





WHAT DID I LEARN

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

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





COROUTINES - JOB

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.





COROUTINES - DISPATCHER

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.





COROUTINES - SCOPE

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.



03

PROCESSOS DE IO





POPULANDO A 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().





dojo / CALLBACK

Defina, a classe privada LearnedItemDatabaseCallback:

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



SCOPE

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.

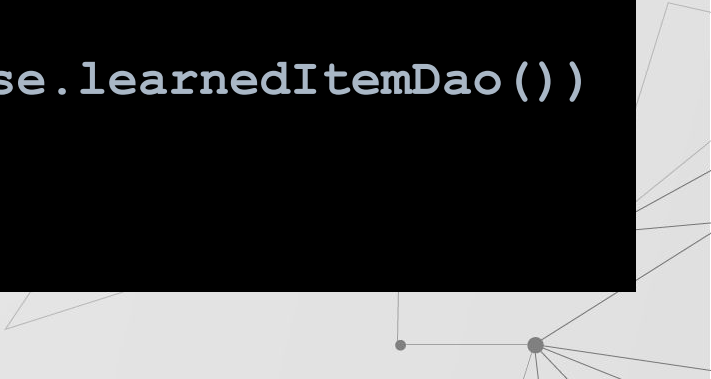




dojo / CALLBACK

Nessa classe, sobrescreva o método onCreate()

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





dojo / CALLBACK

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)
```



SUSPEND FUNCTION

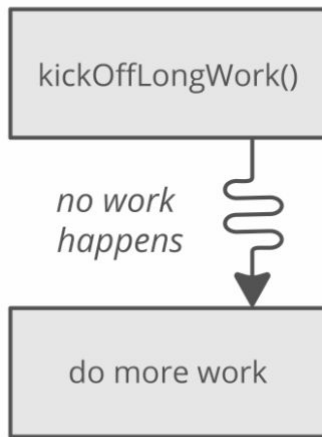
A palavra-chave **suspend** é a maneira de Kotlin de marcar uma função, ou tipo de função, como estando disponível para coroutines.

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.



SUSPEND FUNCTION

*Blocked thread
without coroutines*



*Suspended thread
with coroutines*





SUSPEND FUNCTION

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

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





dojo / ADD CALLBACK

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
```



dojo / ROOM + COROUTINES

Iniciar o banco de dados:

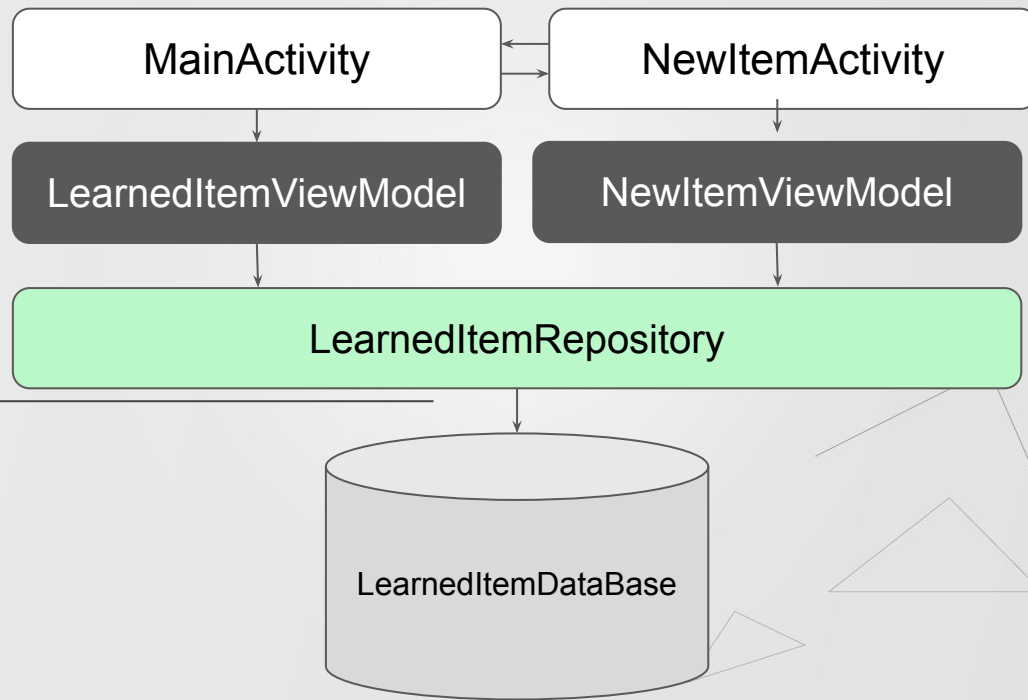
```
val database =  
LearnedItemsDatabase.getDatabase(this,  
CoroutineScope(Dispatchers.IO))
```

04

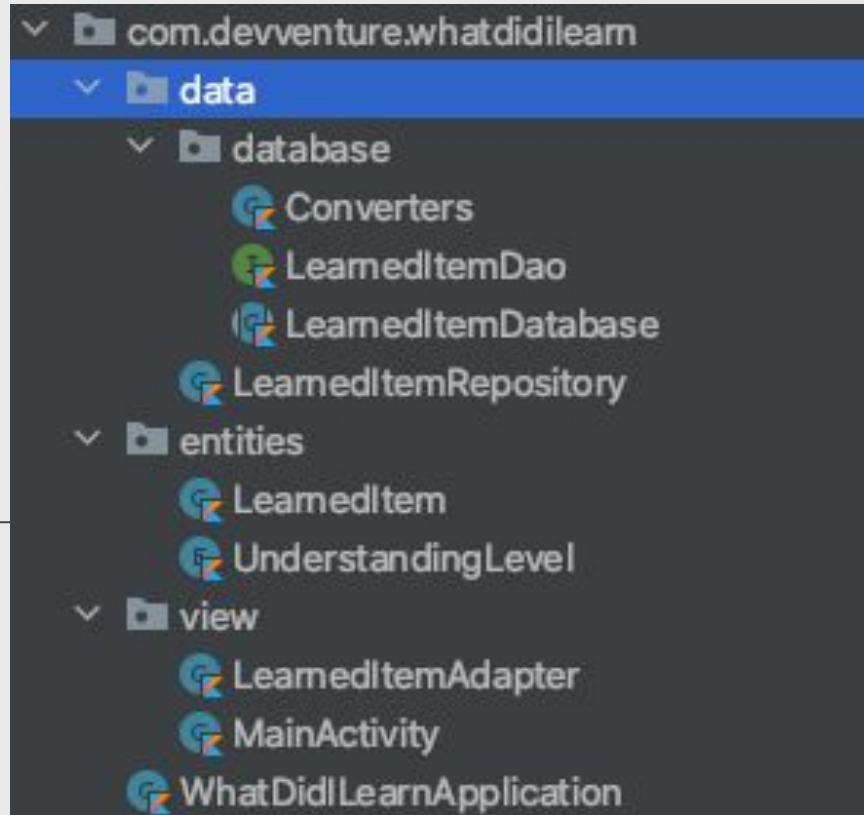
REPOSITORY



ARQUITETURA



ARQUITETURA





PADRÃO REPOSITORY

Iremos refatorar nossa app, adicionando um Repository. Ele será responsável por oferecer uma interface com os dados da app.
Ao invés de permitir o acesso direto ao banco, o Repository irá intermediar as operações:

`getAll()` -> retorna todos os itens registrados

`Insert(item: LearnedItem)` -> adiciona um dado novo na base de dados.



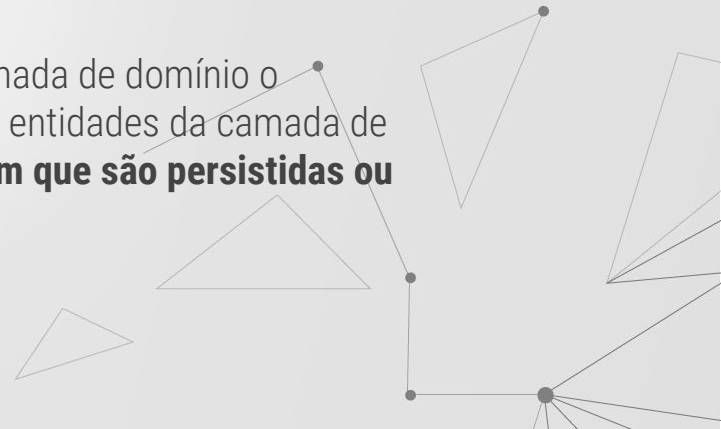


PADRÃO REPOSITORY

A utilização contribui no isolamento da camada de acesso a dados com a camada de negócio.

O Repository Pattern permite um encapsulamento da lógica de acesso a dados, tornando abstrata para as camadas que consomem os dados sua origem e funcionamento.

Com o uso desse pattern, aplicamos em nossa camada de domínio o princípio da persistência ignorante, ou seja, nossas entidades da camada de negócio, **não devem sofrer impactos pela forma em que são persistidas ou consumidas.**

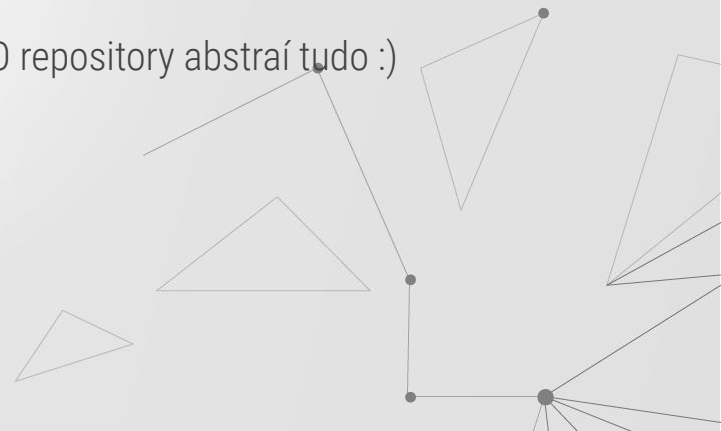




PADRÃO REPOSITORY

- Acesso à um DAO do Room?
- Acesso às SharedPreferences?
- Acesso à um sistema remoto?

Nada disso importa para o restante da aplicação. O repository abstrai tudo :)

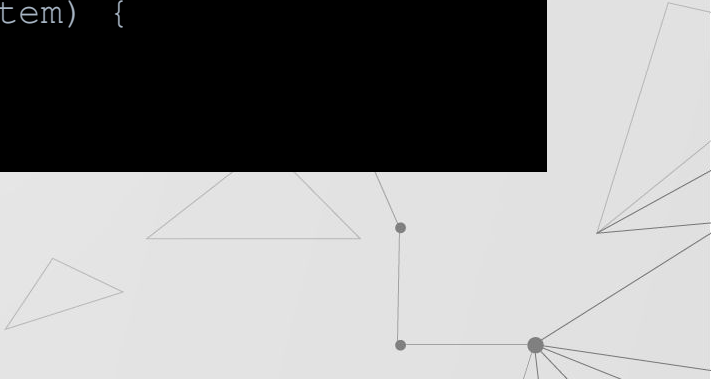




dojo / REPOSITORY

- Crie um pacote repository e dentro dele a classe LearnedItemsRepository

```
class LearnedItemsRepository(private val dao: LearnedItemDao) {  
    val learnedItems = dao.getAll()  
  
    fun insertNewLearnedItem(item: LearnedItem) {  
        dao.insert(item)  
    }  
}
```



dojo / REPOSITORY

- Tente puxar as informações para nossa lista de itens a partir do repository

```
val repository = (application as WhatDidILearnApplication).repository
val recycler = binding.learnedItemsRecyclerView
val adapter = LearnedItemAdapter()

adapter.learnedItems = repository.learnedItems
```

12: ▾ Verbose ▾ 🔍 ☒ Regex Show only selected a

```
r.java:107)
hread.java:7356) <1 internal call>
dAndArgsCaller.run(RuntimeInit.java:492)
ygoteInit.java:930)
not access database on the main thread since it may potentially lock the UI for a long period of time.
hread(RoomDatabase.java:267)
base.java:323)
m:87)
```

05

ROOM + LIVEDATA





LIVEDATA

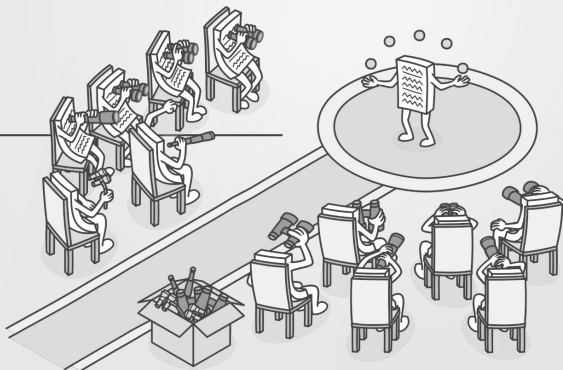
LiveData é uma classe que armazena dados observáveis de forma alinhada com o ciclo de vida da aplicação.

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

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.






dojo / GETALL

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





dojo / GETALL

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	GroupView	É uma view especial que pode conter oi	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

06

VIEW MODEL





VIEWMODEL

A classe **ViewModel** foi projetada para armazenar e gerenciar dados relacionados à IU considerando o ciclo de vida. A classe ViewModel permite que os dados sobrevivam às mudanças de configuração, como a rotação da tela.





VIEWMODEL X ANDROID VIEWMODEL

AndroidViewModel vem com o contexto do aplicativo, o que é útil se você precisar de contexto para obter um serviço do sistema ou tiver um requisito semelhante



dojo / LEARNEDITENSVIEWMOEL

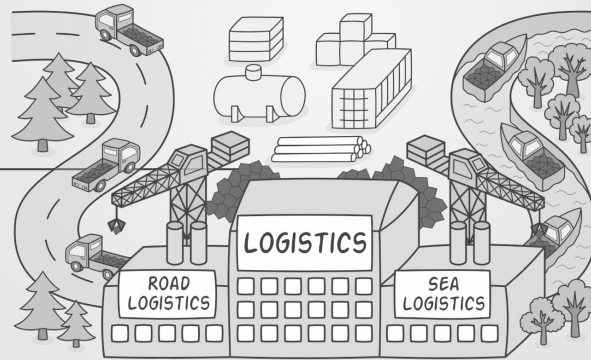
A ViewModel é a classe responsável por "segurar" os dados que a activity/fragment precisa.

- Crie um novo pacote: viewmodel e dentro dele crie a classe MainViewModel

```
class MainViewModel(  
    repository: LearnedItemRepository  
) : ViewModel() {  
    val learnedItems: LiveData<List<LearnedItem>> = repository.learnedItems  
}
```


PADRÃO FACTORY

O Factory é um padrão de design que fornece uma interface para a criação de objetos.



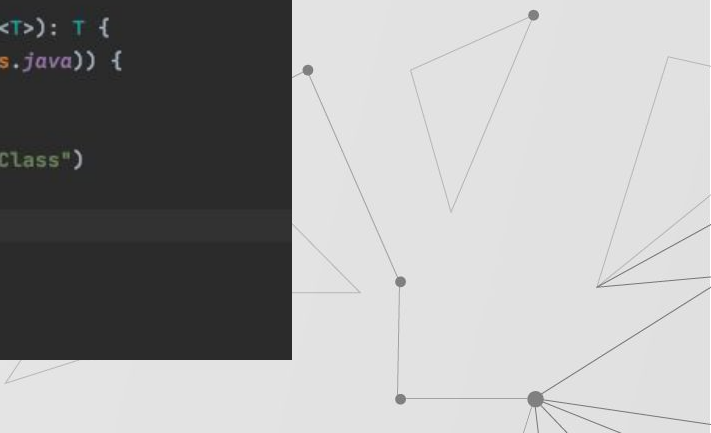


dojo / VIEWMODEL FACTORY

Usaremos a super classe ViewModelProvider.Factory para criar nossa "fábrica" de MainViewModel.

- Crie dentro do pacote viewmodel a classe ViewModelFactory

```
class MainViewModelFactory(  
    private val repository: LearnedItemRepository  
) : ViewModelProvider.Factory {  
    override fun <T : ViewModel?> create(modelClass: Class<T>): T {  
        if(modelClass.isAssignableFrom(MainViewModel::class.java)) {  
            return MainViewModel(repository) as T  
        }  
        throw IllegalArgumentException("Unknown ViewModel Class")  
    }  
}
```



dojo / VIEWMODEL

- Precisamos vincular nosso viewmodel na activity.

```
val repository = (application as WhatDidILearnApplication).repository
val viewModelFactory = MainViewModelFactory(repository)
val viewModel = ViewModelProvider(this, viewModelFactory).get(MainViewModel::class.java)

val recycler = binding.learnedItemsRecyclerView
val adapter = LearnedItemAdapter()

val items = viewModel.learnedItems
items.observe(this, { it: List<LearnedItem>!
    adapter.learnedItems = it
})
```

VIEWMODEL





EXERCÍCIOS

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.
-





EXERCÍCIOS

O que é uma suspend function?

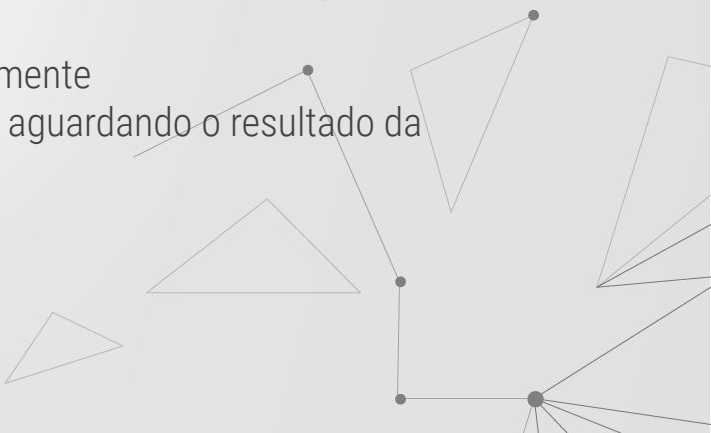
- Uma função comum anotada com a palavra-chave suspend.
 - Uma função que pode ser chamada dentro de uma coroutine.
 - Enquanto uma função de suspensão está em execução, a thread de chamada é suspensa.
 - As funções de suspensão devem sempre ser executadas em segundo plano.
-





EXERCÍCIOS

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.
 - ☐ Suspende a thread de execução temporariamente
 - ☐ Seja bloqueada ou suspensa, a execução ainda está aguardando o resultado da coroutine antes de continuar.
- 

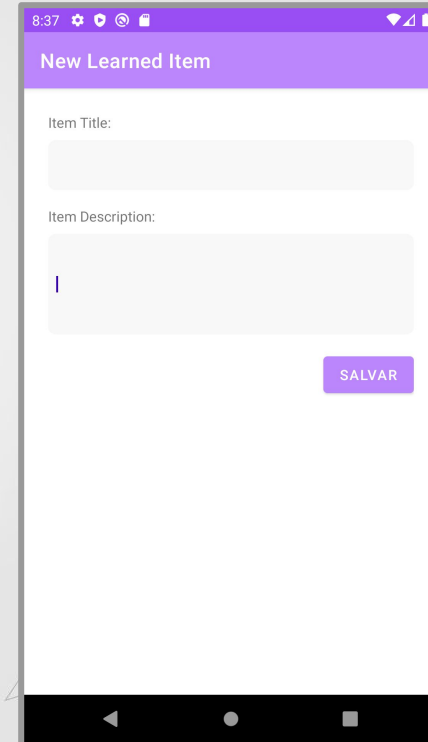
DESAFIO

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

Dica:

Para validar os campos, use o método isEmpty()

Você pode usar um Toast para dar o feedback de validação



The screenshot shows a mobile application interface on a smartphone. At the top, there is a purple header bar with the text "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. To the right of the "Item Description:" field, there is a purple button with the text "SALVAR" in white. The status bar at the top of the phone shows the time as 8:37 and various icons. The bottom of the phone shows the standard Android navigation bar.

Palavras chave da aula de hoje:

- COROUTINES
- SCOPE
- SUSPEND FUNCTIONS
- THREADS
- JOB
- DISPATCHER
- MULTI THREADING
- PROCESSOS

CREDITS: This presentation template was created by **Slidesgo**, including icons by **Flaticon**, and infographics & images by **Freepik**.

Please keep this slide for attribution.