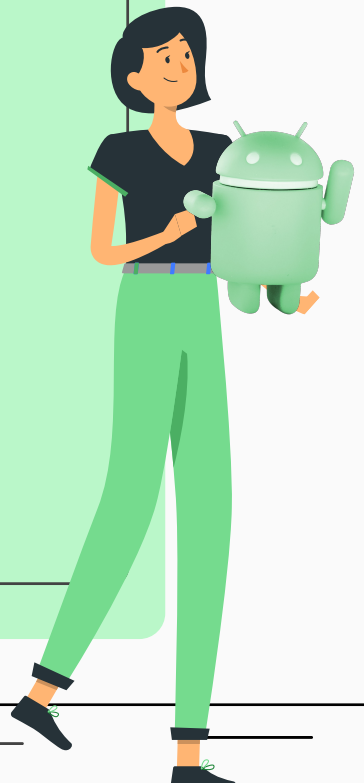
AULA 14

ViewModel e startActivityForResult()



01
VIEW MODEL

02
FLUXO DA APP

03
INSERIR DADOS

04
PROJETO FINAL

REVISÃO COROUTINES

PALAVRAS CHAVE

- COROUTINES
- JOB
- SCOPE
- DISPATCHER
- SUSPEND
- MULTI THREADING

QUESTÃO I

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.

QUESTÃO 2

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.

QUESTÃO 3

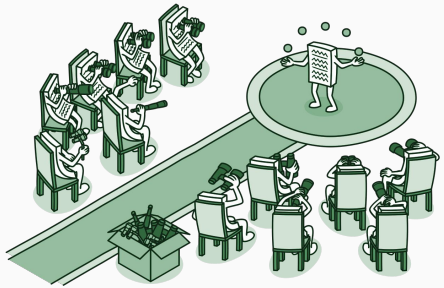
Qual é a diferença entre bloquear e suspender um thread? Marque apenas as verdadeiras:

▢ 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.

LIVEDATA

PADRÃO 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**.



PALAVRAS CHAVE

LIVEDATA

OBSERVER PATTERN

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

SEPARANDO RESPONSABILIDADES

Para isolar melhor as responsabilidades entre as camadas do nosso app, vamos criar duas classes novas:

`LearnedItemsRepository` e `LearnedItemsViewModel`

ADICIONANDO VIEWMODEL

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 LearnedItemViewModel

```
class LearnedItemViewModel(dao:
LearnedItemDao) : ViewModel() {
```

POR QUE PASSAR O LEARNED ITEM DAO COMO PARÂMETRO?

Para construir a classe ViewModel, recebemos um LearnedItemDao. Seguimos esse caminho por alguns motivos:

- Facilitar a escrita de testes
  - Quando formos testar essa classe, sabemos exatamente do quê ela precisa para ser criada. Podemos num testes passar objetos que imitem o comportamento das classes originais, para facilitar o controle do ambiente do teste que estamos criando
- Isolar responsabilidades
  - Recebendo o objeto DAO "pronto" nossa classe não precisa ter a responsabilidade de saber criar esse objeto

ADICIONANDO VIEWMODEL

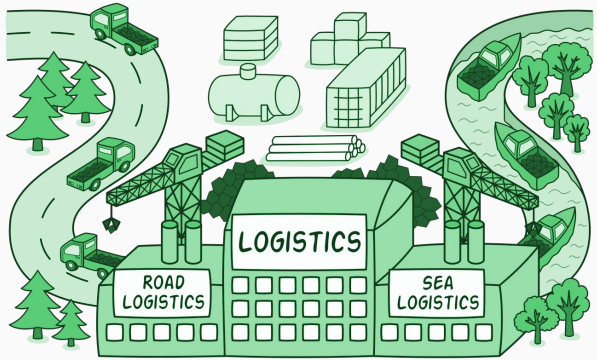
Implemente na ViewModel as tarefas relacionadas a recuperar e adicionar dados no banco:

```
val learnedItems: LiveData<List<LearnedItem>>
private var dao = dao
init {
    learnedItems = dao.getAll()
}
fun insertNewLearnedItem(item: LearnedItem) {
    viewModelScope.launch {
        dao.insert(item)
    }
}
```

ADICIONANDO VIEWMODEL

FACTORY PATTERN

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



FACTORY PATTERN

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

ADICIONANDO VIEWMODELFACTORY

Crie dentro do pacote viewmodel a classe LearnedItemViewModelFactory

```
class LearnedItemViewModelFactory(private val dao:
LearnedItemDao) : ViewModelProvider.Factory  {
   override fun <T : ViewModel?> create(modelClass:
Class<T>): T {
        if
(modelClass.isAssignableFrom(LearnedItemViewModel::class.jav
a)) {
            return LearnedItemViewModel(dao) as T
        }
        throw IllegalArgumentException("Unknown ViewModel
class")
    }
}
```

ADICIONANDO VIEW MODEL NA ACTIVITY

Precisamos vincular nosso viewmodel na activity.

```
val viewModelFactory =
LearnedItemViewModelFactory(learnedItemsDao)
val viewModel = ViewModelProvider(this,
viewModelFactory).get(LearnedItemViewModel::class.j
ava)
```

ADICIONANDO VIEW MODEL NA ACTIVITY

Refatore o código para que as informações mostradas da tela sejam puxadas a partir da viewmodel:

```
val learnedItems = viewModel.learnedItems
```

NEW LEARNED ITEM VIEW MODEL

Para que as informações digitadas sejam salvas no banco, crie uma nova viewmodel para cuidar dessa responsabilidade:

```
class NewLearnedItemViewModel(private var dao:
LearnedItemDao): ViewModel() {
```

SUSPEND FUNCTION

Verifique o tipo da função insert que está definida no DAO.
Atualize para suspend function.

```
@Insert
suspend fun insert(item: LearnedItem)
```

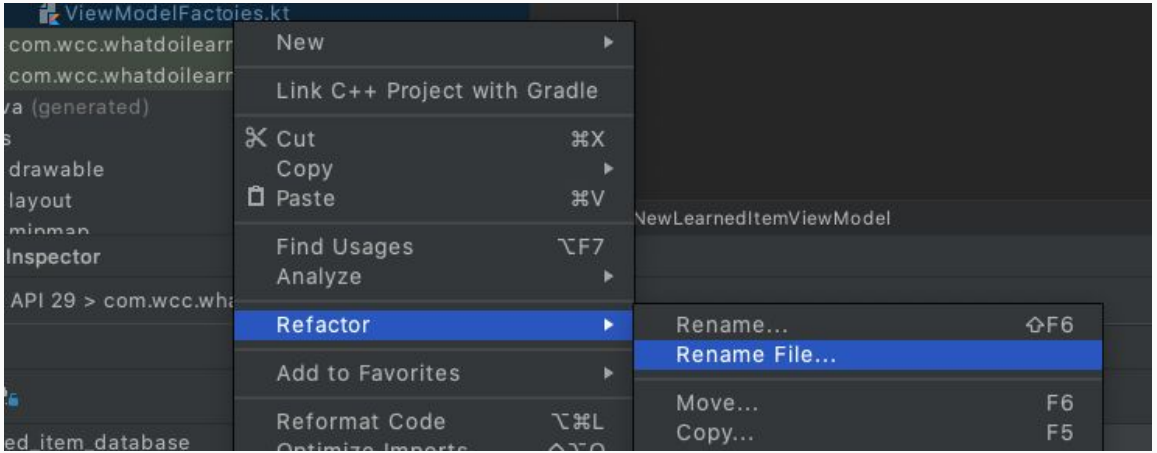
NEW LEARNED ITEM VIEW MODEL

Adicione o método insertNewLearnedItem para capturar as informações "cruas" da tela e adicioná-las no banco de dados.

```
fun insertNewLearnedItem(itemTitle: String, itemDescription:
String) {
    viewModelScope.launch {
        val item = LearnedItem(itemTitle, itemDescription,
UnderstandingLevel.HIGH)
        dao.insert(item)
    }
}
```

VIEW MODEL FACTORY

Renomeie o arquivo LearnedItemViewModelFactory para **ViewModelFactories** (vamos adicionar nossas factories ali)



VIEW MODEL FACTORY

Atualize o arquivo com mais uma factory. Desta vez, a NewLearnedItemViewModelFactory

```
class NewLearnedItemViewModelFactory(private val dao:
LearnedItemDao) : ViewModelProvider.Factory  {
   override fun <T : ViewModel?> create(modelClass:
Class<T>): T {
        if
(modelClass.isAssignableFrom(NewLearnedItemViewModel::class.
java)) {
            return NewLearnedItemViewModel(dao) as T
        }
        throw IllegalArgumentException("Unknown ViewModel
class")
```

SALVAR ITEM

No clique do botão salvar, chame o método insertNewLearnedItem da viewmodel:

```
binding.saveButton.setOnClickListener {
val title = binding.titleEditText.text.toString()
val description = binding.descriptionEditText.text.toString()
viewModel.insertNewLearnedItem(title, description)
```

VOLTANDO PARA MAIN ACTIVITY

Também no clique no botão adicione a ação de voltar para main activity

```
private fun navigateToMainActivity(){
    val intent = Intent(this, MainActivity::class.java)
    startActivity(intent)
}
```

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.

REPOSITORY

Crie um pacote repository e dentro dele a classe LearnedItemsRepository

```
class LearnedItemsRepository(private val dao: LearnedItemDao) {
    val learnedItems = dao.getAll()

    suspend fun insertNewLearnedItem(item: LearnedItem) {
        dao.insert(item)
    }
}
```

REPOSITORY

1. Refatore os viewmodels e activities para usarem o repository
2. Rode o projeto e veja que o comportamento continua o mesmo

PALAVRAS CHAVE

- VIEWMODEL
- LIFECYCLE
- REFATORAR
- ARQUITETURA

QUESTÃO I

Uma ViewModel é:

- ▢ Responsável por cuidar da navegação da aplicação
- ▢ É quem cuida dos dados que serão mostrados nas actvities/fragments
- ▢ Uma ferramenta de gestão de factories
- ▢ Usa o padrão observable para atualizar dados

DESAFIO

Nosso ainda não suporta a opção de se indicar o nível de entendimento do tópico adicionado.

Incremente a interface para capturar essa informação (você pode usar um RadioButton, por exemplo) e depois gravá-la na base de dados.

DESAFIO

Adicione um botão para remover itens na tela inicial.

PROJETO FINAL