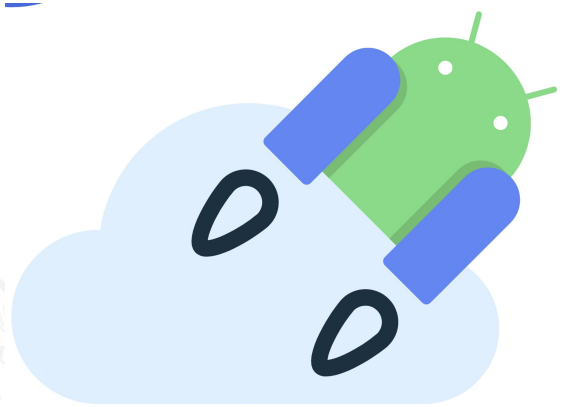
ЛЕКЦИЯ I 

# Android: Jetpack

Теодор Костадинов

# Съдържание

# What is Jetpack?



Jetpack е колекция от различни андроид библиотеки, които са създадени от самите Гугъл за да улеснят работата на разработчиците. Те варират от такива за UI, до такива за самата архитектура на апповете.

*Jetpack is a suite of libraries to help developers follow best practices, reduce boilerplate code, and write code that works consistently across Android versions and devices so that developers can focus on the code they care about.*

Jetpack библиотеките се разпознават по това, че са част от androidx пакета.

Всички Jetpack библиотеки: https://developer.android.com/jetpack/androidx/versions#version-table

# Познати Jetpack библиотеките

Ние вече сме използвали някои от Jetpack библиотеките. Например:

- Room
- Navigation Component
- WorkManager
- RecyclerView
- Activity and Fragment

# App Startup

# Context

- Вече сме виждали, че за инициализация на дадени обекти се нуждаем от *context*
- Не е препоръчително референция към *context* променливата да бъде прехвърляна между методи и класове, тъй като е предпоставка за проблеми
- Андроид предлага решение на този често проблем - изнасяне на логиката за инициализация при стартиране на приложението

```
dependencies {
    implementation "androidx.startup:startup-runtime:1.0.0"
}
```

Повече за Android Startup: https://developer.android.com/topic/libraries/app-startup

# Особености

Всеки инициализатор се дефинира като създава клас, който имплементира интерфейса `Initializer <T>`. Този интерфейс дефинира два важни метода:

- `create()` съдържа всички необходими операции за инициализиране на компонента и връща екземпляр на T
- `dependencies()` връща списък на другите обекти, от които зависи обекта, който инициализираме (библиотеки, които се очаква да са предварително заредени). Можем да използваме този метод, за да контролираме реда, в който приложението стартира инициализаторите при стартиране.

Повече за Android Startup: https://developer.android.com/topic/libraries/app-startup

# Пример

```
// Initializes WorkManager.
class WorkManagerInitializer implements Initializer<WorkManager> {

    @Override
    public WorkManager create(Context context) {
        Configuration configuration = Configuration.Builder().build();
        WorkManager.initialize(context, configuration);
        return WorkManager.getInstance(context);
    }

    @Override
    public List<Class<Initializer<?>>> dependencies() {
        // No dependencies on other libraries.
        return emptyList();
    }

}
```

Имаме context, използваме го само за инициализацията, затова е логично само тук да имаме достъп до него

Тъй като за инициализацията си WorkManager-а не разчита на други библиотеки, връщаме празен лист

Повече за Android Startup: https://developer.android.com/topic/libraries/app-startup

# Manifest

App Startup включва специален content provider, наречен InitializationProvider, който използва, за да открие и извика нашите инициализатори. При стартиране на приложението се откриват тези инициализатори, като първо се проверява <meta-data> тага под тага InitializationProvider в манифеста. След това App Startup извиква метода dependencies() за всички инициализатори, които вече е открил.

```
<provider
    android:name="androidx.startup.InitializationProvider"
    android:authorities="${applicationId}.androidx-startup"
    android:exported="false"
    tools:node="merge">
    <!-- This entry makes WorkManagerInitializer discoverable. -->
    <meta-data  android:name="com.example.WorkManagerInitializer"
          android:value="androidx.startup" />
   <meta-data  android:name="com.example.SomeOtherInitializer"
          android:value="androidx.startup" />
</provider>
```

# Пример за зависимости при инициализация

```
// Initializes ExampleLogger.
class ExampleLoggerInitializer implements Initializer<ExampleLogger> {

    @Override
    public ExampleLogger create(Context context) {
        // WorkManager.getInstance() is non-null only after
        // WorkManager is initialized.
        return ExampleLogger(WorkManager.getInstance(context));
    }

    @Override
    public List<Class<Initializer<?>>> dependencies() {
        // Defines a dependency on WorkManagerInitializer so it can be
        // initialized after WorkManager is initialized.
        return Arrays.asList(WorkManagerInitializer.class);
    }
}
```

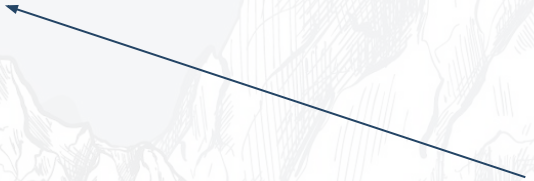
Имаме клас ExampleLogger, който за инициализацията си разчита на WorkManager

Гарантираме си, че реда в който се бъде правилен и WorkManager-а ще се инициализира пръв

Повече за Android Startup: https://developer.android.com/topic/libraries/app-startup

# Manifest

```
<provider
    android:name="androidx.startup.InitializationProvider"
    android:authorities="${applicationId}.androidx-startup"
    android:exported="false"
    tools:node="merge">
    <!-- This entry makes ExampleLoggerInitializer discoverable. -->
    <meta-data  android:name="com.example.ExampleLoggerInitializer"
          android:value="androidx.startup" />
</provider>
```



Името на класа, където сме дефинирали инициализатора

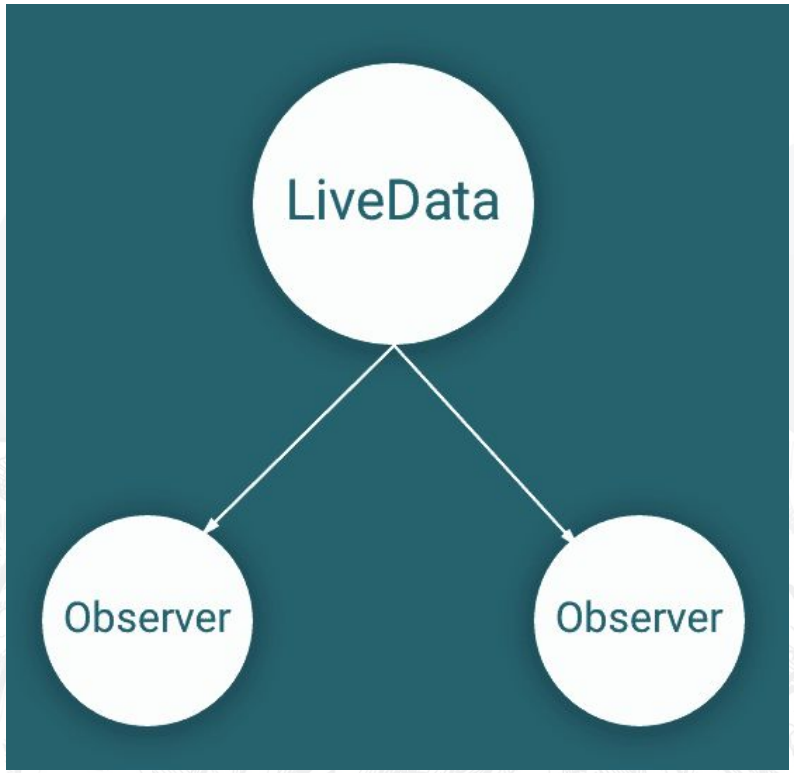
Важно е да се отбележи, че не е нужно да слагаме <meta-data> таг и за WorkManagerInitializer-а, тъй като ExampleLoggerInitializer е посочил в dependencies(), че зависи от него. Стига ExampleLoggerInitializer да е достъпен, и двата ще бъдат инициализирани.

# Почивка
# до 20:10

# Live Data

# Какво е LiveData?

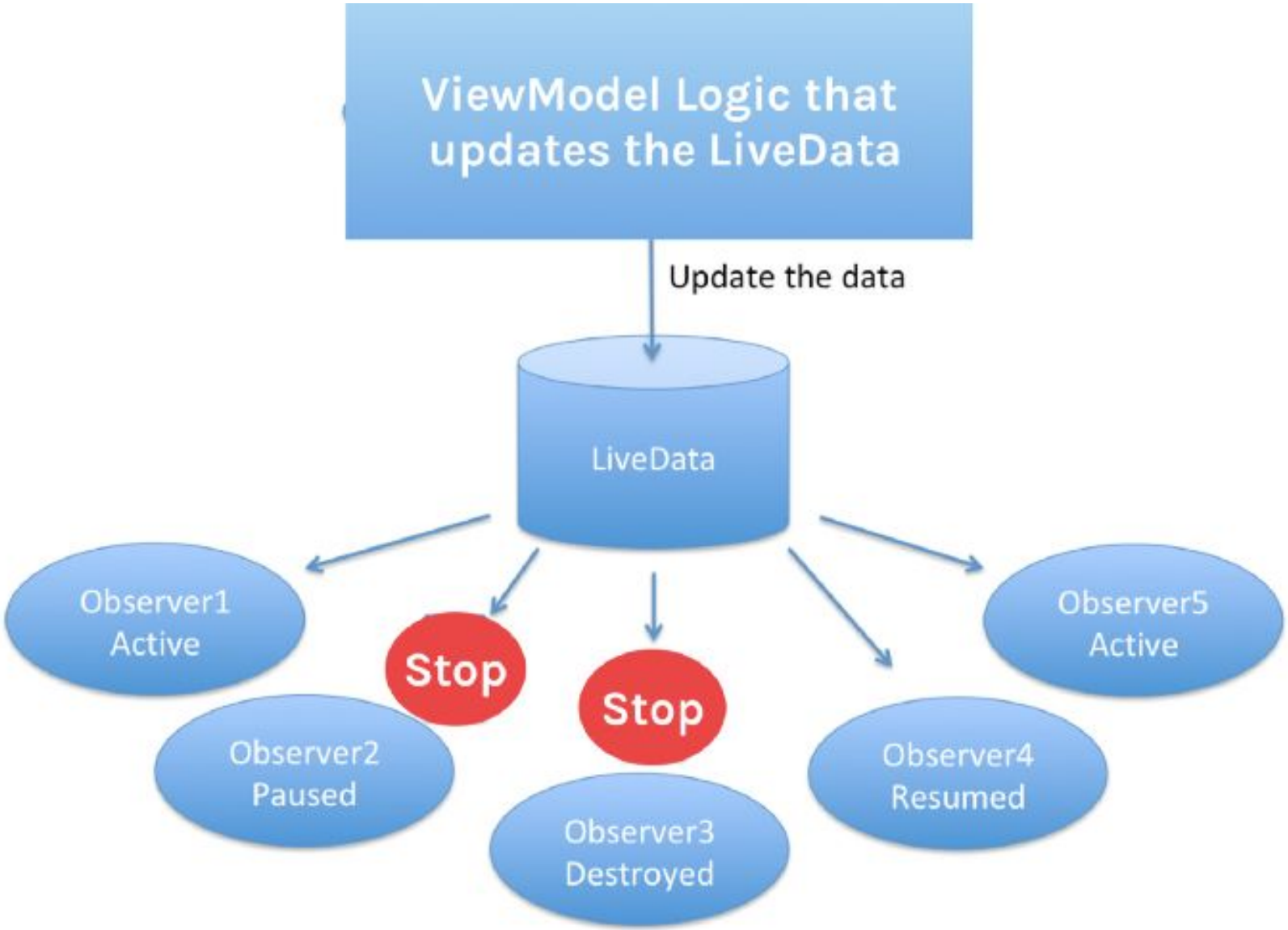
LiveData най-просто е клас, който държи данни. Може да се разглежда като wrapper клас около други обекти. Идеята му е да следи за състоянието на обектите, които държи, и да уведомява при промяна observe-ите, които са subscribe-нати за съответната LiveData.



# Предимства

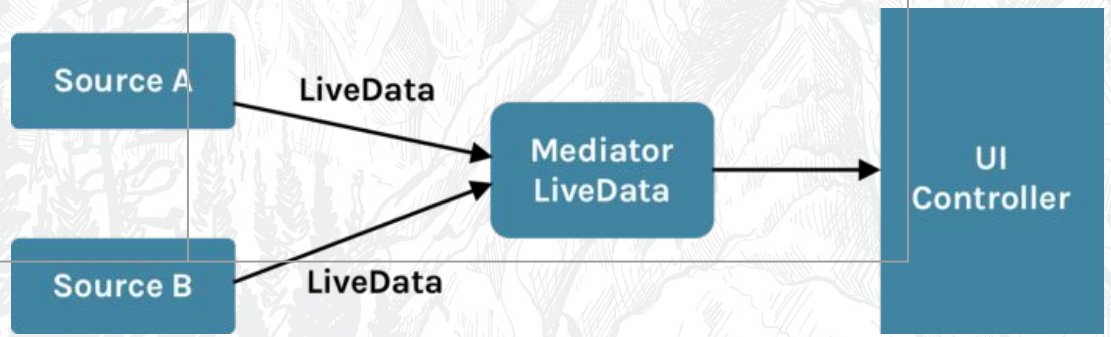
- Може да **държи данни от всякакъв тип**
- **Observable**, което означава, че наблюдателят се уведомява, когато данните, съхранявани от обекта LiveData, се променят.
- L**ifecycle-aware** - когато закачим наблюдател към LiveData, той се свързва със собственик на **Lifecycle** (обикновено **activity или fragment**). LiveData-та актуализира само наблюдатели, които са в състояние на активен жизнен цикъл, като STARTED или RESUMED
- Гаранция, че това, което вижда потребителят, е **актуално**
- **Няма memory leaks**, понеже в момента, в който lifecycle-ът, с който са свързани, се унищожи, LiveData “почиства” след себе си

# Предимства


ViewModel Logic that updates the LiveData
Update the data
LiveData
Observer1 Active
Stop
Observer2 Paused
Stop
Observer3 Destroyed
Observer4 Resumed
Observer5 Active

# Видове LiveData

| LiveData<T> | MutableLiveData<T> | MediatorLiveData<T> |
| --- | --- | --- |
| Използваме я, когато не искаме да модифицираме данните, а само да ги достъпваме. Методите setValue() и postValue() не са достъпни. | Методите setValue() и postValue() са публични. Можем да промените зададените стойности, като извикате тези методи | Можем да обединим резултатите от различни обекти в един, който да observe-аме. |



# Правилно използване на LiveData

```
val uiLiveData: LiveData<MainUiModel>
    get() = _uiLiveData

private val _uiLiveData = MutableLiveData<MainUiModel>().apply {
    viewModelScope.launch {
        userRepository.getUser()?.interest?.let {
            value = MainUiModel(
                suggestionsRepository
                    .getAllUnratedPreferenceSpecificSuggestions(it)
            )
        }
    }
}
```

# setValue(T) vs postValue(T)

- Трябва да използваме метода setValue(T), за да актуализираме данните, които държи LiveData от main нишката. Ако използваме setValue от background нишка, тогава приложението crash-ва с изключението WrongThreadException.
```
_nameLiveData.setValue("something")
```

- Ако кодът се изпълнява в worker нишка, различна от основната нишка, можете да използваме метода postValue(T), за да актуализираме данните. За разлика от setValue(T), postValue(T) може да се използва от всяка нишка, но предпочитано е от различна от main нишката, защото операцията е по-бавна от setValue.
```
_nameLiveData.postValue("something")
```

# Как да observe-аме данните?

```
uiLiveData.observe(viewLifecycleOwner) {
    //имаме новите данни
    //правим нужните промени
}
```

# Data Binding

# View Binding

- Използването на findViewById е бавен процес, който е обвързан и с писането на много код.
- Андроид предлага алтернатива, чрез своята view binding функционалност
- Тя се пуска от грейдъл файла и чрез нея за всеки лейаут се генерира обект, който съдържа всички вюта в него

```
buildFeatures {
        viewBinding true
}
```

```
@Override
public View onCreateView (LayoutInflater inflater,
                          ViewGroup container,
                          Bundle savedInstanceState) {
    binding = ResultProfileBinding.inflate(inflater, container, false);
    View view = binding.getRoot();
    return view;
}
```

# ВЪПРОСИ?

# БЛАГОДАРЯ
## ЗА ВНИМАНИЕТО!