

Movile Next Program

Formação em Desenvolvimento *Android*

Prof. Paulo Salvatore

Tópicos abordados:

- **Architecture Principles (introdução)**
- **Kotlin Basics**
- ***Android - Application Components***
- ***Android - Architecture Components***
 - ***Android Jetpack***
 - ***Lifecycles***
 - ***ViewModel e LiveData***
 - ***Persistence - Data Storage com Room***

1. Movile Next Program

O *Movile Next* é uma iniciativa criada pelo Grupo *Movile*, em parceria com a *GlobalCode*, para capacitar desenvolvedores *Android*, *iOS* e *Backend* de nível pleno e sênior e possibilitar um avanço em carreira. Durante 4 sábados acontecerão, simultaneamente, as formações de cada especialidade com aulas presenciais ministradas por especialistas do mercado, além de atividades e desafios complementares online durante a semana.

2. Introdução

Nesse material abordaremos alguns tópicos essenciais para o desenvolvimento *Android*, os *Application Components*. Analisando aspectos avançados de cada componente individualmente e desmistificando alguns conceitos e aplicações.

Também iremos detalhar os ciclos de vida mais importantes e entender qual a verdadeira importância e quais aspectos devemos prestar atenção, visando tanto o melhor fluxo de desenvolvimento do sistema em si quanto a melhora da *User Experience* que é um dos assuntos mais importantes quando estamos falando do universo *Android*.

Por fim, falaremos do *Android Jetpack* com os componentes *Lifecycles*, *ViewModel*, *LiveData* e *Room*, que fornecem uma série de implementações de arquiteturas complexas de uma maneira bem simples e trazendo um potencial enorme ao desenvolvimento da aplicação, diminuindo o tempo de desenvolvimento necessário e melhorando a qualidade das aplicações construídas.

3. Sumário

1. Mobile Next Program	1
2. Introdução	1
3. Sumário	2
4. Architecture Principles (introdução)	4
5. Kotlin Basics	5
5.1. História e Relevância	6
5.2. Vale a pena aprender Kotlin?	6
5.3. Iniciando em Kotlin	7
5.3.1. Variáveis	8
5.3.2. Funções	9
5.3.3. Classes	9
5.3.4. Classes de Dados (data classes)	11
5.3.5. Heranças e Interfaces	12
5.4. Criando um projeto Kotlin no Android Studio	13
5.5. Principais bibliotecas	15
5.5.1. Kotlin Android Extensions	15
5.5.2. Kotlin Anko	15
5.6. Principais ferramentas	16
5.6.1. Exemplo prático	17
6. Android - Application Components	27
6.1. Activities	27
6.1.1. Ciclo de Vida de uma Activity	28
6.2. Services	30
6.2.1. Usar um Service ou uma Thread?	30
6.2.2. Classes a serem estendidas	31
6.2.2.1. Extensão das classe IntentService	31
6.2.2.2. Extensão da Classe Service	32
6.2.3. Iniciando um Service	35
6.2.4. Encerrando um Service	35
6.2.5. Execução de Service em primeiro plano	36
6.2.6. Ciclo de vida de um Service	36
6.3. Broadcast Receivers	38
6.4. Content Providers	39

6.5. Additional Components	40
6.5.1. Fragments	40
6.5.1.1. Fragments deprecated	41
6.5.1.2. Ciclo de vida de um Fragment	42
6.5.1.3. Comunicação com a Activity	44
6.5.2. Views	44
6.5.3. Layouts	44
6.5.4. Intent Filters	44
6.5.5. Resources	45
6.5.5.1. Acessando as Resources	45
6.5.6. Manifest	46
7. Android Architecture Components	46
7.1. Android Jetpack	46
7.2. Lifecycles, ViewModel e LiveData	47
7.2.1. Exemplo prático	48
7.3. Persistence - Data Storage com Room	65
7.3.1. O que precisamos entender para começar?	65
7.3.2. Arquitetura por trás do Room	66
7.3.3. Iniciando o projeto	67
7.3.4. Criando a primeira Entity	68
7.3.5. Criando o DAO para a Entity	70
7.3.6. Criando o RoomDatabase	71
7.3.7. Criando o Repository	72
7.3.8. Criando o ViewModel	74
7.3.9. Adicionando o Layout XML	75
7.3.10. Populando a base de dados	79
7.3.11. Criando uma nova Activity para adicionar palavras	81
7.3.12. Conectando os dados	84
8. Referências Bibliográficas	87
9. Vídeos Recomendados	90
10. Licença e termos de uso	91
11. Leitura adicional	92
11.1. Lifecycles	92
11.2. Room	92

4. Architecture Principles (introdução)

Durante diversos momentos do curso, alguns conceitos importantes de arquitetura, princípios de programação e desenvolvimento ágil serão apresentados. Nessa primeira aula, falaremos um pouco sobre os princípios universais, conhecidos como *Universal Principles* que se aplicam em diversos aspectos da vida.

Basicamente, quando estamos desenvolvendo uma aplicação, uma série de desafios são apresentados e na maioria das vezes lidamos diferente com determinadas situações, levando diversos aspectos em consideração que são diferentes para cada dia de trabalho. Dentro da programação, esses princípios universais estão atrelados diretamente com uma prática de desenvolvimento ágil conhecida como *Extreme Programming* (XP) que defende lançamentos frequentes em ciclos curtos de desenvolvimento, tendo como objetivo melhorar a produtividade e introduzir *checkpoints* nos quais novos requisitos podem ser adotados.

Esse princípio também diz para adotar um uso extremo de revisão de códigos, testes de unidade de tudo, evitar programar funcionalidades até que elas sejam de fato necessárias, uma estrutura de gerenciamento flexível, simplicidade e clareza no código, esperando mudanças nos requisitos a medida que o tempo passa e o entendimento do problema aumenta e por último uma forte comunicação dos desenvolvedores envolvidos com o usuário/cliente da aplicação.

Algumas dicas são fundamentais para o desenvolvimento da aplicação:

- Nunca tente fazer o app perfeito na versão 1.0:
 - Não dá certo, o desenvolvimento de um app é um processo incremental, quanto mais o tempo passa, mais vamos entendendo melhor o problema que estamos tentando resolver.
- Valide sempre aquilo que está sendo feito:
 - A importância de validar uma ideia é infinitamente superior à implementação dela.
- Pense sempre no conceito de *MVP* (*Minimum Viable Product*):
 - A menor versão funcional do projeto mostrará se a aplicação é viável ou não.
- Regra dos 20/80:
 - 80% do projeto em 20% do tempo (fase de prototipação e implementação das funcionalidades *core* do app);
 - 20% do projeto em 80% do tempo (fase de polimento e correção de bugs).

Leitura obrigatória para todo programador: 97 Journey Every Programmer should Accomplish [\[ref 1\]](#).

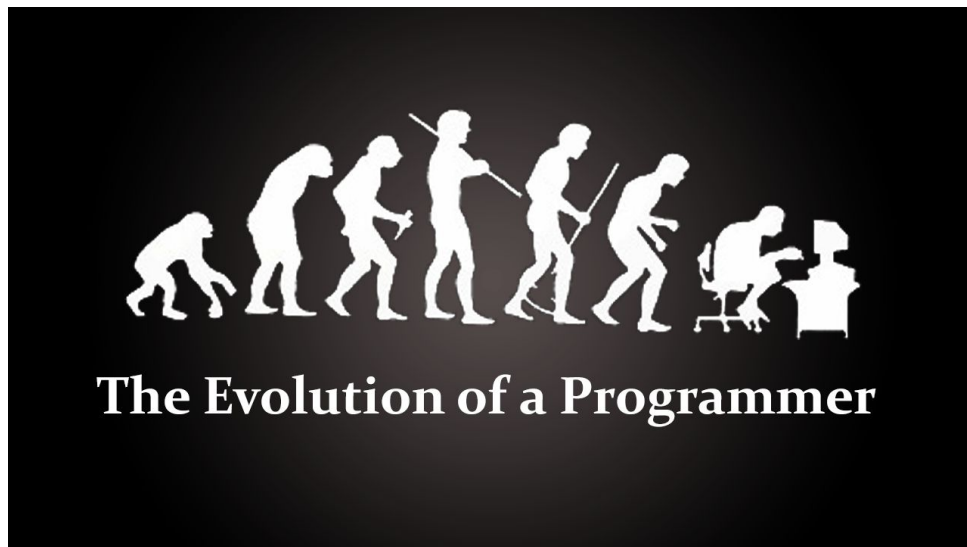


Figura 1: A evolução de um programador

Fonte: Medium - 97 Journey Every Programmer should Accomplish

Outro conceito muito importante para a programação é o *SoC* (*Separation of Concerns*), que consiste em separar o desenvolvimento em sessões distintas, fazendo uma separação em blocos, por exemplo:

Um bloco exclusivo deverá falar com o banco de dados e apenas ela ficará responsável por isso. Outro bloco será responsável pela comunicação com a interface de usuário.

Um bom exemplo disso são os padrões de arquitetura *MVC* (*Model, View e Controller*) e *MVP* (*Model, View e Presenter*), que utilizam da ideia do *SoC* para dividir as partes do código em determinados blocos feitos para realização de tarefas específicas.

A frase chave do *SoC* é: divida o projeto em CAMADAS.

5. Kotlin Basics

Desde 2017, o Google anunciou o suporte ao Kotlin, a nova linguagem de programação oficial para a plataforma Android. Desde então, os desenvolvedores iniciaram seus estudos até que hoje é uma das linguagens mais amadas da comunidade (2º lugar em 2018 pelo StackOverflow) e vem ganhando cada vez mais espaço no cenário Android. Neste capítulo iremos mostrar as características básicas da linguagem e como construir uma aplicação simples utilizando Kotlin, além das ferramentas exclusivas presentes no Android Studio que auxiliam no desenvolvimento.

5.1. História e Relevância

Antes de iniciar o desenvolvimento com a linguagem Kotlin é interação conhecer um pouquinho dela, apenas para nos ambientar um pouco melhor. A empresa russa *Jetbrains*, conhecida por suas maravilhosas *IDEs* baseadas na famosa *IntelliJ IDEA*, e também por ser criadora do *Android Studio*, iniciou o projeto *Kotlin* em 2010, só divulgado-o pela primeira vez em 2011 e se tornou open-source em 2012 [[ref 1](#)] [[ref 2](#)].

A linguagem foi criada pois o *CEO* da *Jetbrains*, Dmitry Jemerov, disse que a maioria das linguagens não tinham o que eles estavam procurando, com exceção da linguagem *Scala*, que foi descartada pelo tempo de compilação ser lento. Um dos objetivos do *Kotlin* era compilar tão rápido quanto *Java*. Em 2016 foi lançada a *Kotlin* v1.0, considerado o primeiro lançamento estável da linguagem.

Durante a *Google I/O '17*, o *Google* anunciou suporte oficial para o *Kotlin* no *Android*, o que deixou toda a comunidade de desenvolvedores (e o *Google* também) muito empolgado. Mesmo antes do suporte oficial, muitos desenvolvedores já utilizavam *Kotlin* para o desenvolvimento *Android* e diversas palestras sobre o assunto já eram dadas tentando melhorar o desenvolvimento *Android*, reduzindo os principais problemas e melhorando o fluxo de escrita de código, que pode ser muito trabalhoso no *Java* para alguns casos. **Antes de prosseguir, veja o vídeo 1** ([ir para vídeo](#)).

5.2. Vale a pena aprender Kotlin?

Atualmente existem diversos motivos que fazem valer a pena aprender desenvolver em *Kotlin*, principalmente para o cenário *Android* (que é o que estamos tratando nesta aula). O principal motivo é o ganho de produtividade, exatamente tudo que é possível fazer em *Java* é possível fazer em *Kotlin* com menos linhas de códigos, tornando o projeto mais legível, mais fácil de manter e mais fácil de estruturar também.

Um outro fator bem importante é interoperabilidade com o *Java*. Sim! O *Kotlin* conversa normalmente com o *Java*! Isso significa que você pode iniciar a escrever *Kotlin* em um projeto que esteja feito 100% em *Java* que você conseguirá executar suas bibliotecas normalmente. Não foi a toa que o *Google* adicionou suporte oficial a linguagem, isso torna a nossa vida muito mais fácil.

Outros fatores cruciais são: **Compatibilidade:** totalmente compatível com *JDK 6*, garantindo a execução em *devices* mais antigos; **Desempenho:** um aplicativo *Kotlin* é tão rápido quanto um aplicativo em *Java*, as vezes até ganhando em termos de velocidade; **Tempo de compilação:** o mesmo caso que o desempenho,

velocidade igual ou superior (em alguns casos) ao *Java*; e **Curva de Aprendizado:** para um desenvolvedor *Java* é extremamente fácil iniciar em *Kotlin*, garanto que no primeiro projeto usando a linguagem você já se sentirá apaixonado por ela e sabendo usar até os recursos avançados da linguagem e com aquele sentimento: "Por que ninguém me mostrou isso antes?" [[ref 3](#)].

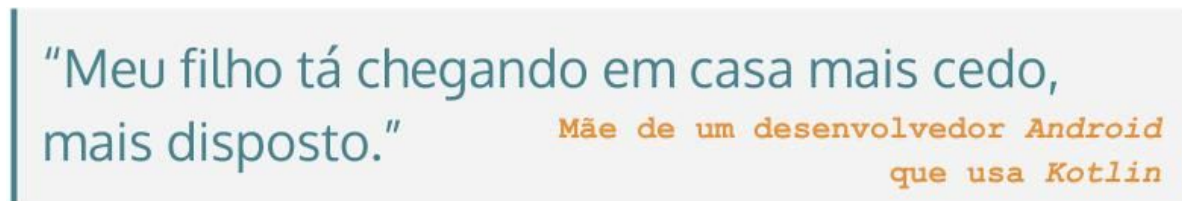


Figura 6: depoimento importante da mãe de um desenvolvedor *Android* que usa *Kotlin*

Fonte: autor

5.3. Iniciando em Kotlin

Agora que já sabemos um pouco mais sobre essa linguagem e os motivos pelos quais vale a pena dedicarmos nosso tempo a ela, vamos iniciar nos familiarizando com seus aspectos de sintaxe e estruturação e entender alguns detalhes dentro do ambiente *Android* como ferramentas disponíveis para nosso auxílio.

O *Kotlin* é uma linguagem expressiva (escreva mais com menos), tem segurança contra valores nulos, é orientada objetos (apesar de também aceitar programação funcional), utiliza algumas expressões características do paradigma funcional para resolver problemas mais simples (como expressões *lambda*, por exemplo) e também gosto de dizer que ela é inteligente e faz bastante coisa pra você. Para efeitos comparativos, gosto de falar que a linguagem *Kotlin* lembra muito a linguagem *Swift*, da *Apple* e até um pouquinho de PHP na concatenação de variáveis.

A seguir demonstrarei alguns aspectos básicos que são relevantes sabermos antes de iniciar o desenvolvimento em *Kotlin*, para mais informações consulte sempre a documentação do *Kotlin* [[ref 4](#)]. Também recomendo duas leituras adicionais sobre *Kotlin*, do próprio site oficial, a primeira é a *Kotlin - Basic Syntax* [[ref 5](#)] e a segunda é *Kotlin - Coding Conventions* [[ref 6](#)].

Uma recomendação de vídeo extremamente interessante para quem quer ter mais noção do potencial da linguagem *Kotlin* é o vídeo 2 ([ir para vídeo](#)), da *Google I/O '17*, que mostra uma introdução do *Kotlin* mostrando a linguagem mais de um aspecto multiplataforma e mostrando o que a *Jetbrains* planeja para o futuro.

5.3.1. Variáveis

Uma das primeiras coisas que precisamos saber no desenvolvimento *Kotlin* é como utilizar as variáveis que possui algumas diferenças em relação ao que estamos acostumados no *Java*.

A primeira diferença é o conceito **val** e **var** que funciona de uma maneira muito simples:

val: Essa declaração é o que estamos acostumados ao *final* no *Java*, onde uma variável *val* não irá mudar de valor no decorrer do código. Apenas o método *getter* é definido para esse tipo (que o *Kotlin* já faz automaticamente).

var: É basicamente o contrário do *val*, onde a alteração do valor da variável ocorrerá normalmente. Possui método *getter* e *setter* criados automaticamente (podendo ser sobrescritos, caso haja necessidade).

A segunda diferença é o tipo opcional, onde o *Kotlin* irá primeiro tentar entender o tipo da variável por associação de algum retorno ou pelo valor declarado e deverá ser inserido após o nome da variável com o caractere " : " (dois pontos) antes, seguindo do valor desejado para a atribuição.

A terceira diferença é a proteção contra nulo, onde no *Kotlin* é feita pela símbolo " ? " (ponto de interrogação) ao lado do tipo da variável, indicando que a variável pode receber valor nulo (durante a inicialização da variável) ou fazendo uma checagem de segurança (durante a utilização da variável). O símbolo deve obrigatoriamente estar do lado do tipo da variável.

```
val variavelNaoAceitaNull: String = "Conteúdo Final"
var variavelAceitaNull: String? = null // Pode ser alterado
```

Uma outra característica interessante é durante a utilização das variáveis, onde é possível utilizar o símbolo " ? " (ponto de interrogação) para fazer uma checagem se a variável possui um conteúdo ou não, exemplo:

```
var location: Location? = null

// A seguinte declaração:
val latitude = location?.latitude

// É a mesma coisa que:
if (location != null) {
    val latitude: Double? = location.latitude
}
```



```
}
```

Também é possível iniciar uma variável sem dizer o tipo, onde o *Kotlin* irá tentar defini-lo utilizando associação, seja do valor que foi inicializado ou do retorno de alguma função que foi chamada durante a inicialização.

```
val nome = "Paulo Salvatore" // Tipo: String
val location = Location(...) // Tipo: Location?
```

Note que no *Kotlin* o símbolo ";" (ponto e vírgula) no final das declarações é opcional.

5.3.2. Funções

Além das variáveis é importante saber como utilizar corretamente as funções. Apesar de conhecermos as funções dentro das classes como métodos, no *Kotlin*, a nomenclatura não altera. Uma característica importante é que o tipo das variáveis é sempre obrigatório, tanto os parâmetros quanto do retorno.

```
fun funcaoSemRetorno() {
}

fun funcaoComRetorno(): String {
    return "String"
}

fun funcaoComParametrosObrigatorios(nome: String,
                                     sobrenome: String) {
}

fun funcaoComParametrosOpcionais(nome: String?,
                                  sobrenome: String?) {
}
```

5.3.3. Classes

As classes possuem funcionalidades e características bem interessantes no *Kotlin*, possuindo várias facilidades e vantagens em relação ao *Java* e a outras

linguagens. Algumas declarações são bem semelhantes ao que estamos acostumados:

```
// Para classes vazias as chaves são opcionais
class Pessoa {
}
```

Se quisermos declarar o construtor da classe, podemos fazer isso direto na declaração dela, conforme o exemplo abaixo:

```
class Pessoa constructor(nome: String,
                          sobrenome: String) {
}

// Ou simplesmente sem escrever 'constructor':
class Pessoa(nome: String,
             sobrenome: String) {
}
```

É interessante observar nesse exemplo que as variáveis recebidas estão disponíveis apenas como variáveis locais que podem ser utilizadas na atribuição de propriedades ou dentro da declaração `init { }`, para acessá-la em funções da própria classe devemos criar propriedades `val` ou `var` para armazenar os valores. Dessa maneira, estamos automaticamente criando o *getter* de cada propriedade. Caso tivéssemos declarado `var`, o *setter* de cada propriedade também teria sido gerado:

```
class Pessoa(nome: String,
             sobrenome: String) {
    val nome: String = nome // [: String] -> opcional
    val sobrenome: String = sobrenome
}

// Ou simplesmente:
class Pessoa(val nome: String,
             val sobrenome: String) {
}
```

As funções das classes (que normalmente conhecemos como métodos) são declaradas normalmente e possui por padrão o acesso público e que não é necessário declarar, pois só devemos declarar caso queira que esse acesso mude:

```
class Pessoa(val nome: String,
             val sobrenome: String) {
    // Acesso public
    fun falar() {
        // ...
    }

    // Acesso forçado como private
    private fun andar() {
        // ...
    }
}
```

Além disso, as classes e as funções também são por padrão *final*, se deseja que a função esteja disponível para sobrescrita, deve-se adicionar a declaração *open* tanto na classe como na função.

```
open class Pessoa(val nome: String,
                  val sobrenome: String) {
    // Função final
    fun falar() {
        // ...
    }

    // Função disponível para override
    open fun comer() {
        // ...
    }
}
```

5.3.4. Classes de Dados (data classes)

Diversas vezes criamos classes apenas para estruturar dados. Além das classes convencionais, uma grande adição ao *Kotlin* são as *data classes*, que

carregam uma série de membros extremamente úteis quando estamos trabalhando com classes para esse propósito.

Os membros gerados automaticamente são:

- equals()/hashCode()
- toString()
- componentN() para cada propriedade (na ordem)
- copy()

Para uma classe poder ser assinalada como *data class* os requisitos abaixo devem ser cumpridos:

- O construtor primário deve ter pelo menos um parâmetro
- Todos os parâmetros do construtor primário precisam ser *val* ou *var*
- *Data classes* não podem ser *abstract*, *open*, *sealed* ou *inner*
- (Antes da v1.1) *Data classes* podem implementar apenas interfaces

```
data class Pessoa(val nome: String,  
                  val sobrenome: String) {  
}
```

5.3.5. Heranças e Interfaces

Para sinalizar que uma classe possui herança é bem simples. Apesar da declaração parecer que o *Kotlin* aceita herança-múltipla **ele não aceita**, assim com o *Java*. Sempre quando for declarar herança para uma classe, os parênteses ao lado da superclasse são obrigatórios:

```
class ExampleActivity :  
    AppCompatActivity() {  
}
```

Já para implementar alguma interface, basta colocar uma vírgula ao lado da superclasse (caso exista uma) e escrever o nome da interface a ser implementada. Isso fará com que o *Kotlin* peça pela implementação obrigatória:

```
class ExampleActivity :  
    AppCompatActivity(),  
    View.OnClickListener {  
    override fun onClick(view: View?) {  
        // ...  
    }  
}
```

```
}  
  
}
```

5.4. Criando um projeto Kotlin no Android Studio

Neste capítulo criaremos um novo projeto no *Android Studio* com suporte à linguagem *Kotlin* entendendo as principais novidades e mudanças que devemos saber.

Para começar, certifique a versão do seu *Android Studio*, onde o *Kotlin* passou a estar disponibilidade integralmente a partir da versão 3.0. Ainda assim, é possível adicionar o *Kotlin* em versões anteriores, basta instalar o *Plugin* do *Kotlin* e reiniciar a *IDE*. Caso esteja na versão 3.0 ou superior, crie um novo projeto, habilitando o suporte ao *Kotlin*, como é possível ver na figura 7.

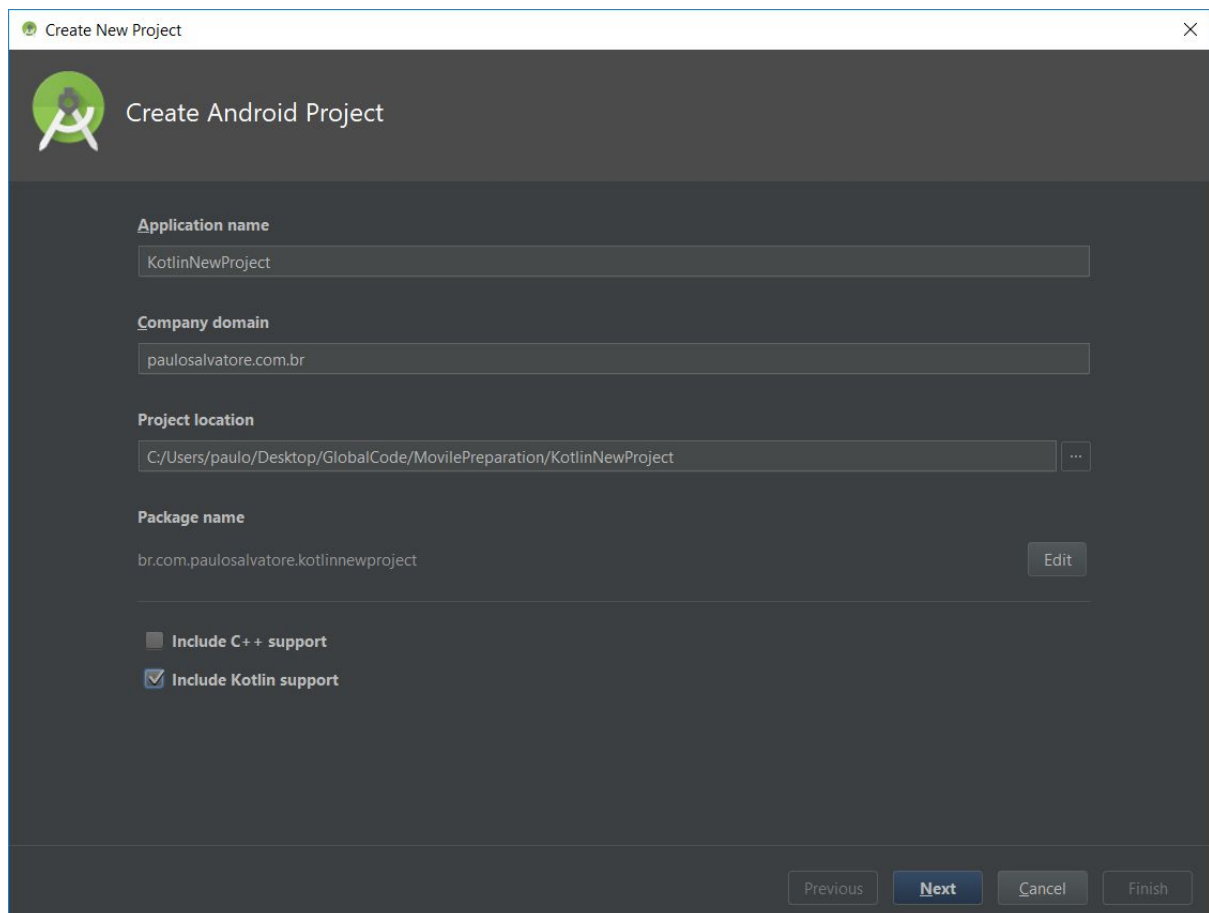


Figura 7: Criar novo projeto em *Kotlin*, habilitando o suporte

Fonte: autor - *Android Studio*

Nesse projeto, irei deixar a *API* mínima 15 e criarei uma *Empty Activity* chamada *MainActivity*, mantendo todas as configurações padrões.

Aguarde o projeto ser criado e, assim que tudo carregar, abra os arquivos 'build.gradle' do projeto e do *app*.

No *gradle* do projeto, você deverá notar as seguintes linhas:

```
buildscript {  
    ext.kotlin_version = "1.2.51"  
    dependencies {  
        classpath  
        "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"  
    }  
}
```

No *gradle* do *app*, você deverá notar as seguintes linhas:

```
apply plugin: "kotlin-android"  
  
apply plugin: "kotlin-android-extensions"  
  
//...  
  
dependencies {  
    implementation  
    "org.jetbrains.kotlin:kotlin-stdlib-jre7:$kotlin_version"  
}
```

Antes de prosseguir no novo projeto, certifique que está utilizando a versão mais atualizada do *Kotlin* (das versões disponíveis no *stable channel*) clicando no menu "Tools | Kotlin | Configure Kotlin Plugin Updates | Check for Updates Now". Instale caso tenha alguma atualização disponível.

Depois que tudo estiver atualizado, reinicie o *Android Studio*.

Note que a versão da biblioteca do *Kotlin JRE7* está depreciada, substitua para *JDK7* caso necessário.

```
implementation  
"org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlin_version"
```

Com isso definido, o projeto já está pronto para receber o suporte à linguagem de programação *Kotlin*.

5.5. Principais bibliotecas

Neste capítulo vamos entender as principais bibliotecas disponíveis no *Kotlin*, que tornam o desenvolvimento *Android* ainda mais poderoso.

5.5.1. Kotlin Android Extensions

A primeira ferramenta é o *Kotlin Android Extensions*, que atualmente já vem referenciada em todos os novos projetos de *Kotlin* no *Android Studio*. Essa ferramenta nos permite acessar *Views* sem precisar de declaração (mais fácil do que feito com *DataBinding* ou bibliotecas de *binding*), acessando-as diretamente como se fossem variáveis. Para se aprofundar mais nessa biblioteca, consulte a [referência](#) 7.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

    // Em vez de findViewById<TextView>(R.id.textView)
    // setText()
    textView.text = "Hello, world!"

    // getText()
    val name: String = textView.text as String

    // ou
    val name: String = textView.text.toString()
}
```

5.5.2. Kotlin Anko

Kotlin Anko [\[ref 8\]](#) é uma verdadeira caixa de ferramentas com várias funcionalidades distintas (com possibilidade de importar as funcionalidades separadamente) e muito poderosas. Trata-se de um projeto *open source* desenvolvido pela própria *Jetbrains* e que potencializa ainda mais o uso da linguagem para o desenvolvimento *Android*. Nesse projeto usaremos apenas o Anko Commons [\[ref 9\]](#).

Para importar a biblioteca no projeto primeiro definimos sua versão no *gradle* do projeto:

```
buildscript {  
    // ...  
    ext.anko_version = "0.10.5"  
    // ...  
}
```

E depois importamos apenas o que iremos utilizar no *gradle* do *app*, no nosso caso, apenas o **anko-commons**:

```
buildscript {  
    // ...  
    // Anko Commons  
    implementation  
    "org.jetbrains.anko:anko-commons:$anko_version"  
    implementation  
    "org.jetbrains.anko:anko-design:$anko_version"  
    // ...  
}
```

5.6. Principais ferramentas

Quando o assunto é *Kotlin*, ferramenta no *Android Studio* é o que não falta. Como a linguagem foi desenvolvida pela *Jetbrains*, mesma empresa que desenvolveu e que mantém o *Android Studio*, é comum termos diversas ferramentas muito poderosas que aumentam a produtividade e facilitam nosso trabalho.

A primeira ferramenta que falarei aqui é o *Kotlin Bytecode*, que possui uma funcionalidade muito interessante de decompilar o código *Kotlin* em *Java*, exibindo uma versão relativa do código que acabamos de escrever em *Kotlin*, em sua versão aproximada de *Java*. Para acessar essa ferramenta basta ir até o menu em "Tools | Kotlin | Show Kotlin Bytecode". Uma nova guia de acesso deve aparecer (podendo acessá-la posteriormente sem precisar ir até o menu novamente) e dentro existe uma opção *Decompile*. Certifique-se de que o arquivo *Kotlin* que quer realizar o processo está aberto e clique no botão. O *Android Studio* irá abrir uma nova janela com o arquivo decompilado.

A segunda ferramenta interessante (que é mais uma funcionalidade) é a capacidade do *Android Studio* de converter código *Java* em código *Kotlin*. Para utilizá-la basta abrir o seu arquivo *Java*, ir até o menu e selecionar a opção "Code |

Convert Java File to Kotlin File" que o *Android Studio* além de converter o código, irá alterar a extensão do arquivo para '.kt'.

Além dessa opção de conversão, quando copiamos um código *Java* e colamos em um arquivo *Kotlin*, o *Android Studio* nos pergunta se queremos que ele tente fazer a conversão, o que nem sempre funciona, depende se ele consegue encontrar as dependências certas para realizar tal ação. Geralmente tem mais eficácia em trechos pequenos de código.

5.6.1. Exemplo prático

Nesse exemplo prático, iremos construir uma aplicação utilizando uma *RecyclerView* para uma lista com diversas linguagens de programação. Para iniciar, devemos realizar a configuração das bibliotecas de *support v7* do *Android*.

Para importar a biblioteca no projeto primeiro definimos sua versão no *gradle* do projeto:

```
buildscript {  
    // ...  
    ext.support_v7_version = "27.1.1"  
    // ...  
}
```

E depois importamos apenas o que iremos utilizar no *gradle* do *app*, no nosso caso, *RecyclerView* e *CardView*:

```
buildscript {  
    // ...  
    // Support V7 (RecyclerView e CardView)  
    implementation  
    "com.android.support:recyclerview-v7:$support_v7_version"  
    implementation  
    "com.android.support:cardview-v7:$support_v7_version"  
    // ...  
}
```

Com essas implementações definidas, vamos a configuração dos *layouts*. Primeiro, crie um novo *layout* chamado 'programming_language_item.xml' e insira o seguinte conteúdo:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.v7.widget.CardView
xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="wrap_content">

  <LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal"
    android:padding="8dp">

    <ImageView
      android:id="@+id/ivMain"
      android:layout_width="96dp"
      android:layout_height="110dp"
      android:contentDescription="Main Image"
      tools:ignore="HardcodedText"
      tools:src="@drawable/ic_developer_board" />

    <LinearLayout
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:gravity="center"
      android:orientation="vertical">

      <TextView
        android:id="@+id/tvTitle"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:textSize="20sp"
        android:textStyle="bold"
        tools:text="Title" />

      <TextView
        android:id="@+id/tvLaunchYear"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
```

```
        android:textColor="#AAA"
        android:textSize="12sp"
        tools:text="Year: YYYY" />


        <TextView
            android:id="@+id/tvDescription"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:layout_marginTop="4dp"
            android:ellipsize="end"
            android:maxLines="3"
            android:textSize="14sp"
            tools:text="Description" />

    </LinearLayout>

</LinearLayout>

</android.support.v7.widget.CardView>
```

A referência '@drawable/ic_developer_board' é um *Vector Asset* adicionado manualmente que servirá como placeholder.

Referência do *asset*: 

Posteriormente, no arquivo de *layout* da *MainActivity* insira o seguinte conteúdo:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <android.support.v7.widget.RecyclerView
        android:id="@+id/recyclerView"
        android:layout_width="match_parent"
```

```
android:layout_height="match_parent"
tools:listitem="@layout/programming_language_item"/>

</android.support.constraint.ConstraintLayout>
```

Com os *layouts* configurados, vamos iniciar a preparação dos arquivos do projeto para começar a inserir o código. Nesse momento pode fechar todos os arquivos, deixando por enquanto apenas o arquivo da *MainActivity* aberto. Após isso criaremos três novos pacotes: 'adapter', 'model' e 'view'; e colocaremos a *MainActivity* dentro do pacote *view*, ficando semelhante à figura 8.

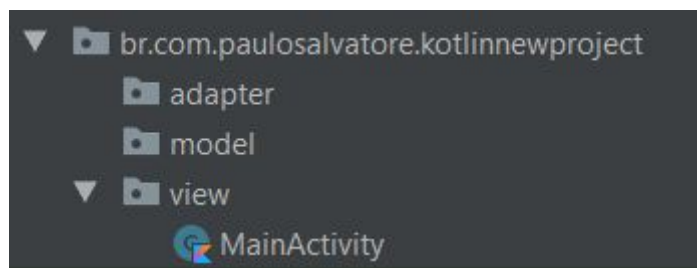


Figura 8: Pacotes iniciais da aplicação

Fonte: autor - *Android Studio*

Depois, iremos iniciar a criação das nossas classes. A primeira é a classe que conterá a estrutura dos dados que teremos, que no nosso caso é: **imagem, título, ano de lançamento e descrição**. Para isso, criaremos a nossa classe *Kotlin* dentro do pacote *model*. Certifique-se de marcar a opção *Class* na caixa de diálogo e insira o nome *ProgrammingLanguage*. O arquivo criado deverá possuir a extensão '.kt'.

Para essa classe, iremos transformá-la em uma *data class* e adicionar as propriedades citadas anteriormente, ficando assim:

```
data class ProgrammingLanguage(val imageResourceId: Int,
                                val title: String,
                                val year: Int,
                                val description: String)
```

Com a *data class* definida, vamos criar nosso *Adapter*. Crie uma classe *Kotlin* chamada *ProgrammingLanguageAdapter* dentro do pacote *adapter*. Inicie-a com o seguinte conteúdo:

```
class ProgrammingLanguageAdapter(
```

```
private val items: List<ProgrammingLanguage>,
private val context: Context,
private val listener: (ProgrammingLanguage) -> Unit
) : Adapter<ProgrammingLanguageAdapter.ViewHolder>()
```

Em seguida, antes de implementar interface, vamos criar a classe *ViewHolder* como *inner class*. O código deverá ficar assim:

```
import android.content.Context
import android.support.v7.widget.RecyclerView.Adapter
import android.view.ViewGroup
import
br.com.paulosalvatore.kotlinnewproject.model.ProgrammingLanguage

class ProgrammingLanguageAdapter(
    private val items: List<ProgrammingLanguage>,
    private val context: Context,
    private val listener: (ProgrammingLanguage) -> Unit
) : Adapter<ProgrammingLanguageAdapter.ViewHolder>() {
    class ViewHolder(itemView: View) :
        RecyclerView.ViewHolder(itemView) {
    }
}
```

Note que alterei manualmente o *import* da *RecyclerView* no momento para referenciar direto o nome *Adapter* para encurtar a declaração da interface.

Agora implemente os três membros da interface e eles já deverão referência a classe *ViewHolder* que acabou de ser criada. Antes de construirmos o conteúdo dos membros da interface, vamos criar um membro *bindView()* para a classe *ViewHolder*, que deve receber as informações de cada item e aplicar nas *Views* e por fim aplicar o *listener* também recebido. Ficando assim:

```
class ViewHolder(itemView: View) :
    RecyclerView.ViewHolder(itemView) {
    fun bindView(item: ProgrammingLanguage,
        listener: (ProgrammingLanguage) -> Unit) =
        with(itemView) {
```

```
}  
}
```

O conteúdo do método é bem simples, basta atribuir os valores às *Views* correspondentes, da seguinte maneira:

```
fun bindView(item: ProgrammingLanguage,  
            listener: (ProgrammingLanguage) -> Unit) =  
with(itemView) {  
    ivMain.setImageDrawable(ContextCompat.getDrawable(context,  
item.imageResourceId))  
    tvTitle.text = item.title  
    tvLaunchYear.text = item.year.toString()  
    tvDescription.text = item.description  
  
    setOnClickListener { listener(item) }  
}
```

Nota: Na declaração acima utilizaremos a ID de um recurso existente para carregar a imagem do item. Posteriormente podemos alterar isso para receber uma *URL* e fazer o carregamento da imagem na *web*.

Com isso podemos começar a escrever a implementação dos membros da interface. Começaremos pelo *onBindViewHolder()* que basicamente recebe uma posição, encontra o item correto e realizar o *bindView()*.

```
override fun onBindViewHolder(holder: ViewHolder, position:  
Int) {  
    val item = items[position]  
    holder.bindView(item, listener)  
}
```

Posteriormente, implementamos o *getItemCount()*.

```
override fun getItemCount(): Int {  
    return items.size  
}
```

E por fim, o membro *onCreateViewHolder()*, responsável por inflar o *layout* e chamar o *ViewHolder*.

```
override fun onCreateViewHolder(parent: ViewGroup, viewType:
Int): ViewHolder {
    val view =
LayoutInflater.from(context).inflate(R.layout.programming_lan
guage_item, parent, false)
    return ViewHolder(view)
}
```

Com tudo definido, o código inteiro deve estar semelhante a isso:

```
import android.content.Context
import android.support.v4.content.ContextCompat
import android.support.v7.widget.RecyclerView
import android.support.v7.widget.RecyclerView.Adapter
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import br.com.paulosalvatore.kotlinnewproject.R
import
br.com.paulosalvatore.kotlinnewproject.model.ProgrammingLangu
age
import
kotlinx.android.synthetic.main.programming_language_item.view
.*

class ProgrammingLanguageAdapter(
    private val items: List<ProgrammingLanguage>,
    private val context: Context,
    private val listener: (ProgrammingLanguage) -> Unit
) : Adapter<ProgrammingLanguageAdapter.ViewHolder>() {
    override fun onCreateViewHolder(parent: ViewGroup,
viewType: Int): ViewHolder {
        val view =
LayoutInflater.from(context).inflate(R.layout.programming_lan
guage_item, parent, false)
```

```

        return ViewHolder(view)
    }

    override fun getItemCount(): Int {
        return items.size
    }

    override fun onBindViewHolder(holder: ViewHolder, position:
Int) {
        val item = items[position]
        holder.bindView(item, listener)
    }

    class ViewHolder(itemView: View) :
RecyclerView.ViewHolder(itemView) {
        fun bindView(item: ProgrammingLanguage,
                    listener: (ProgrammingLanguage) -> Unit) =
with(itemView) {

ivMain.setImageDrawable(ContextCompat.getDrawable(context,
item.imageResourceId))
        tvTitle.text = item.title
        tvLaunchYear.text = item.year.toString()
        tvDescription.text = item.description

        setOnClickListener { listener(item) }
    }
}
}

```

Nota: Detalhe aos *imports* que possuem nomes exclusivos ao pacote em que estão relacionados.

Para prosseguir com a construção da nossa *RecyclerView*, vamos até a nossa *MainActivity* iniciar a declaração. Precisaremos de apenas três declarações: a lista de objetos a serem exibidos; a construção e atribuição do *Adapter* criado; e a construção e atribuição do *Layout Manager* a ser utilizado, podendo realizar esse último passo em duas declarações separadas.

Iniciaremos declarando a lista de objetos a serem exibidos na *RecyclerView*, declarando-os como um membro da classe *MainActivity* que retorna os itens:


```
private fun recyclerViewItems(): List<ProgrammingLanguage> {  
    val kotlin = ProgrammingLanguage(R.drawable.kotlin,  
        "Kotlin",  
        2010,  
        "Kotlin é uma Linguagem de programação que compila  
para a Máquina virtual Java e que também pode ser traduzida  
para JavaScript e compilada para código nativo. É  
desenvolvida pela JetBrains, seu nome é baseado na ilha de  
Kotlin onde se situa a cidade russa de Kronstadt, próximo à  
São Petersburgo. Apesar de a sintaxe de Kotlin diferir da de  
Java, Kotlin é projetada para ter uma interoperabilidade  
total com código Java. Foi considerada pelo público a 2ª  
linguagem 'mais amada', de acordo com uma pesquisa conduzida  
pelo site Stack Overflow em 2018.")  
  
    return listOf(kotlin)  
}
```

Para declarar o *Adapter* podemos utilizar alguns recursos do *Kotlin* que facilitam o trabalho:

```
recyclerView.adapter =  
    ProgrammingLanguageAdapter(  
        recyclerViewItems(),  
        this) {  
        longToast("Clicked item: ${it.title}")  
    }
```

Note que dentro das chaves da declaração do *Adapter* estamos passando o *listener* para o *click*. Por enquanto temos apenas um *longToast*, declaração possível pela biblioteca *Anko*, que reduz a sua chamada. Também é possível notar a concatenação do título do item clicado, acessado através do membro *it*, disponível nesse tipo de declaração e representando o objeto da *data class*.

Por último definimos o *Layout Manager*:

```
// Lista na Vertical
```

```
val layoutManager = LinearLayoutManager(this)
recyclerView.layoutManager = layoutManager
```

Também é possível selecionar outros tipos de estruturas para mudar a forma que a sua *RecyclerView* é organizada. Para isso basta mudar a declaração do *LayoutManager* escolhendo uma das opções abaixo:

```
// Grid
val layoutManager = GridLayoutManager(this, 2)

// Grid Escalonável
val layoutManager = StaggeredGridLayoutManager(
    2, StaggeredGridLayoutManager.VERTICAL)

// Lista na Horizontal
val layoutManager = LinearLayoutManager(this,
    LinearLayoutManager.HORIZONTAL, false)
```

Rode o projeto e veja o resultado!



Figura 9: *RecyclerView App* pronto e executando!

Fonte: autor - *Android Emulator (API 27)*

Com isso terminamos nosso primeiro projeto com *Kotlin*. Podemos perceber que a linguagem torna o desenvolvimento *Android* cada vez mais simples e com

menos complicação mas mantém seu excelente desempenho e rendimento. Uma grande preocupação da *Jetbrains* é com os resultados que a linguagem obtém, deixando bem claro em cada nova ferramenta ou funcionalidade lançada ou atualizada, os impactos dela em termos de execução e também levando muito em consideração ao tamanho em *bytes* de suas bibliotecas, visando manter os arquivos *APKs* bem leves.

Nos próximos capítulos, iniciaremos um novo projeto em *Kotlin*, cobrindo mais alguns aspectos da linguagem para nos familiarizarmos ainda mais com ela e também abordando alguns outros aspectos como *Push Notification*, *Image Loading* e *Networking*.

6. Android - Application Components

Application Components [ref 2] são os blocos essenciais de uma aplicação *Android*, esses componentes são referenciados juntos pelo manifesto da aplicação (*AndroidManifest.xml*) que descreve cada um dos componentes da aplicação e como eles interagem.

6.1. Activities

Uma **Activity** [ref 3] representa uma única tela com uma interface de usuário, representando basicamente ações na tela.

Toda *Activity* precisa estar referenciada no arquivo de manifesto do Android para que o sistema possa executá-la, assim como a *Activity* que iniciará por padrão..

Uma *Activity* é implementada como uma subclasse da classe **Activity** (ou uma das subclasses existentes, como a **AppCompatActivity** *), seguindo o exemplo abaixo:

```
public class MainActivity extends AppCompatActivity {  
}
```

*A classe *AppCompatActivity* está relacionada diretamente a classe **Activity** por meio da herança.

O gerenciamento inteligente de *Activities* permite garantir, por exemplo:

- As mudanças de orientação ocorrerem sem problemas e sem interromper a experiência do usuário;
- Os dados não serem perdidos durante as transições de telas;
- O sistema eliminar processos quando for apropriado.

6.1.1. Ciclo de Vida de uma Activity

É importante ressaltar que as *Activities* possuem um ciclo de vida [ref 4] (ver figura 2), onde o sistema Android (que possui total controle sobre as telas) não te perguntará sobre o que fazer com a *Activity*, mas irá te alertar sobre o que está prestes a fazer.

Conforme o usuário navega entre as telas do seu aplicativo ou até mesmo outros aplicativos do dispositivo, a *Activity* muda de estado constantemente conforme os estados existentes em seu ciclo de vida.

A classe **Activity** fornece uma série de métodos de *callbacks* (funções chamadas quando um evento acontece) permitindo ao desenvolvedor executar determinadas ações visando melhorar o fluxo de comportamento da aplicação.

Os principais métodos *callbacks* do ciclo de vida de uma *Activity* são:

- ***onCreate()***
- ***onStart()***
- ***onResume()***
- ***onPause()***
- ***onStop()***
- ***onDestroy()***

Por exemplo, se estivermos construindo uma aplicação que reproduza vídeos podemos pausar a reprodução e encerrar a conexão com a internet quando o usuário muda para outra aplicação. Quando o usuário retornar, podemos restabelecer a conexão e reproduzir o vídeo a partir do exato momento em que ele parou.

Em outras palavras, fazer o trabalho certo no momento certo e tratando as transições de maneira apropriada faz com que a aplicação seja mais robusta e mais performática. Por exemplo, boas implementações dos *callbacks* ciclo de vida podem evitar problemas como:

- *Crashing* quando o usuário recebe uma ligação ou alterna para outro aplicativo enquanto usa sua aplicação.
- Consumir recursos valiosos do sistema enquanto o usuário não está usando.
- Perder o progresso do usuário caso ele saia do aplicativo e retorne em um momento posterior.
- *Crashing* ou perda de progresso quando o usuário altera a orientação do dispositivo entre retrato (em pé) e paisagem (deitado).

Como pode ser observado na figura 2, o ciclo de vida de uma *Activity* possui diversas etapas e é possível distinguir o que é feito exatamente em cada uma delas e em qual método de *callback* determinadas ações como a criação ou destruição da tela é feita. É possível notar, por exemplo, que uma *Activity* só é destruída caso o

sistema *Android* ou o usuário decida, porém, ela é recriada em diversos momentos, sempre que o processo é finalizado.

Mais a frente falaremos sobre persistência de dados envolvendo o ciclo de vida de uma *Activity*.

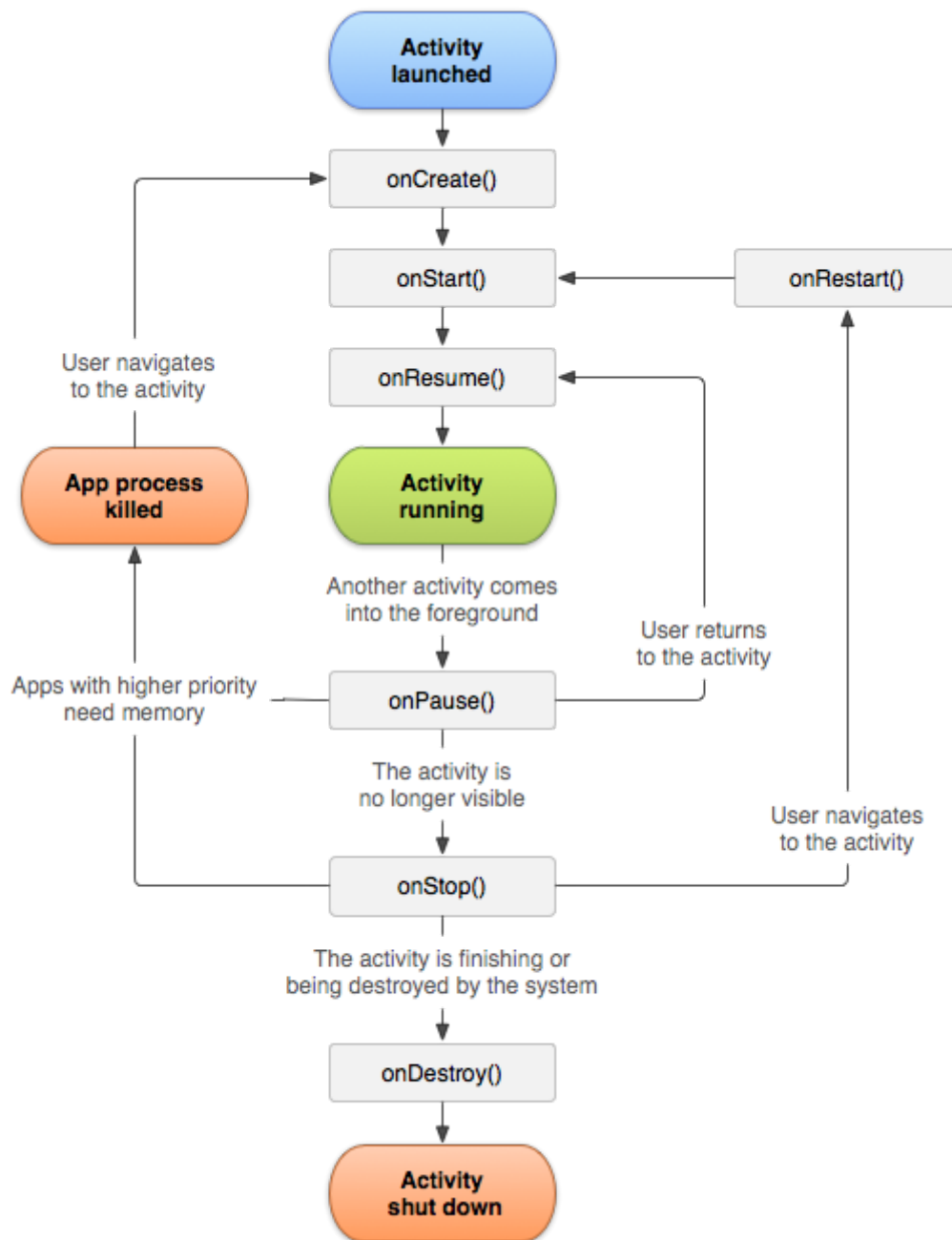


Figura 2: Ciclo de vida simplificado de uma *Activity*
Fonte: *Google Developers - Android Documentation*

6.2. Services

Um **Service** [\[ref 5\]](#) é um componente que é executado em segundo plano feito para realizar operações longas, ou seja, que ficam um bom tempo executando.

Por exemplo, um *service* pode reproduzir músicas em *background* enquanto o usuário está em uma outra aplicação, ou pode buscar dados na internet sem bloquear a interação do usuário com uma *Activity*.

Como *Activities* (e outros componentes), você deve declarar todos os *Services* no arquivo de manifesto do *Android*.

Um *Service* é implementado como uma subclasse da classe **Service** (ou uma das subclasses existentes), seguindo o exemplo abaixo:

```
public class MyService extends Service {  
}
```

Quando queremos implementar um *Service*, podemos utilizar dois métodos, o *startService()* e o *bindService()*.

O método *startService()* irá iniciar o *Service* chamando o método do ciclo de vida *Service.onStartCommand()* e fará com que o processo seja executado até que o método *stopSelf()* ou *stopService()* seja chamado.

Já o método *bindService()* também iniciará o *Service*, porém, iniciar dessa maneira irá chamar o método do ciclo de vida *Service.onBind()* e criará uma dependência entre o *Context* informado, onde o *Service* será considerado necessário enquanto o *Context* que o executou existir. Por exemplo, se o *Context* é uma *Activity* que parou, o *Service* não será requerido para continuar enquanto a *Activity* não voltar.

6.2.1. Usar um Service ou uma Thread?

Uma dúvida muito comum surge quando precisamos executar uma ação específica em segundo plano, mas não sabemos se optamos por criar um *Service* exclusivo para a ação ou uma nova *Thread* (um novo processo).

Um *Service* é simplesmente um componente que pode ser executado em segundo plano mesmo quando o usuário não está interagindo com o aplicativo. Portanto, um serviço deve ser criado somente se for realmente necessário.

Caso precise realizar trabalhos fora da *Main Thread* (processo principal), mas somente enquanto o usuário estiver interagindo com o aplicativo, você deve criar uma nova *Thread* e não um *Service*.

Por padrão, um *Service* é executado no mesmo processo que o aplicativo em que ele é declarado e na *Main Thread* do aplicativo. Portanto, se o *Service* realizar operações intensivas isso diminuirá o desempenho da *Activity*. Para evitar impacto no desempenho do aplicativo, deve-se iniciar uma nova *Thread* dentro do *Service*.

Em breve iremos discutir mais sobre o que é *Thread*, quais tipos existem e quando devemos usar cada um dos tipos disponíveis.

6.2.2. Classes a serem estendidas

Geralmente, há duas classes que podem ser estendidas para criar um serviço iniciado: ***Service*** (classe padrão para todos os *Services*) e ***IntentService*** (subclasse da classe *Service* que usa uma *WorkerThread* para lidar com todas as solicitações de inicialização, uma por vez).

6.2.2.1. Extensão das classe *IntentService*

Como a maioria dos *Services* não precisam lidar com várias solicitações simultaneamente, é melhor se o *Service* for implementado usando a classe *IntentService*.

O *IntentService* faz o seguinte:

- Cria uma *WorkerThread* padrão que executa todos os *intents* entregues a *onStartCommand()* separado da *MainThread*.
- Cria uma fila de processamento que passa um *intent* por vez à implementação *onHandleIntent()*, para que nunca seja necessário preocupar-se com várias *Threads*.
- Interrompe o *Service* depois que todas as solicitações forem resolvidas para que não seja necessário chamar *stopSelf()*.
- Fornece implementações padrão de *onBind()* que retornam como nulo.
- Fornece uma implementação padrão de *onStartCommand()* que envia o *intent* para a fila de processamento e, em seguida, para a implementação de *onHandleIntent()*.

Exemplo de implementação de *IntentService*:

```
import android.app.IntentService;
import android.content.Intent;

public class HelloIntentService extends IntentService {

    /**
```

```
* Um construtor é obrigatório e precisar chamar o super
IntentService(String)
* com um nome para a WorkerThread.
*/
public HelloIntentService() {
    super("HelloIntentService");
}

/**
* O IntentService chamará esse método pela WorkerThread
padrão com
* a intent que iniciou o Service. Quando esse método
retornar, IntentService
* irá encerrar o Service, conforme for apropriado.
*/
@Override
protected void onHandleIntent(Intent intent) {
    // Normalmente a gente executa um trabalho aqui, como
    baixar um arquivo.
    // Para esse exemplo, iremos apenas 'dormir' a aplicação
    por 5 segundos.
    try {
        Thread.sleep(5000);
    } catch (InterruptedException e) {
        // Restaura o status de interrupção.
        Thread.currentThread().interrupt();
    }
}
}
```

Acesse a documentação do Android sobre "*Extending IntentService*" para obter mais detalhes sobre essa implementação [\[ref 6\]](#).

6.2.2.2. Extensão da Classe Service

O uso da classe *IntentService* torna a implementação de um *Service* muito simples, porém, às vezes precisamos realizar várias *Threads* simultâneas em vez de processar as solicitações por meio de uma *WorkerThread*. Nesses casos, é possível estender a classe *Service* para lidar com cada *intent*.

Para efeito de comparação, o exemplo de código a seguir é uma implementação da classe *Service* que realiza exatamente o mesmo trabalho que o

exemplo anterior usando *IntentService*. Ou seja, para cada solicitação de inicialização, ela usa uma *WorkerThread* para realizar o trabalho e processa apenas uma solicitação por vez.

```
import android.app.Service;
import android.content.Intent;
import android.os.Handler;
import android.os.HandlerThread;
import android.os.IBinder;
import android.os.Looper;
import android.os.Message;
import android.os.Process;
import android.widget.Toast;

public class HelloService extends Service {
    private Looper mServiceLooper;
    private ServiceHandler mServiceHandler;

    // Manipulador que recebe as mensagens da thread
    private final class ServiceHandler extends Handler {
        public ServiceHandler(Looper looper) {
            super(looper);
        }
        @Override
        public void handleMessage(Message msg) {
            // Normalmente a gente executa um trabalho aqui,
            // como baixar um arquivo. Para esse exemplo, iremos
            // apenas 'dormir' a aplicação por 5 segundos.
            try {
                Thread.sleep(5000);
            } catch (InterruptedException e) {
                // Restaura o status de interrupção.
                Thread.currentThread().interrupt();
            }
            // Encerra o Service usando o startId, para que ele
            // não seja encerrado no meio da execução de uma
            // outra tarefa
            stopSelf(msg.arg1);
        }
    }
}
```

```
}

@Override
public void onCreate() {
    // Inicia a Thread executando o Service. Note que
    // nós criamos uma Thread separada pois o Service
    // normalmente roda no processo da MainThread, o
    // que não é muito recomendado. Também definimos
    // uma prioridade de background para que o trabalho
    // intenso da CPU não atrapalhe a MainThread.
    HandlerThread thread = new HandlerThread(
        "ServiceStartArguments",
        Process.THREAD_PRIORITY_BACKGROUND
    );
    thread.start();

    // Pega o Looper do Manipulador da Thread para usar
    // como nosso manipulador
    mServiceLooper = thread.getLooper();
    mServiceHandler = new ServiceHandler(mServiceLooper);
}

@Override
public int onStartCommand(Intent intent, int flags, int
startId) {
    Toast.makeText(this, "service starting",
Toast.LENGTH_SHORT).show();

    // Para cada requisição de início, envia uma mensagem
    // para iniciar a tarefa e enviar o startId para que
    // saibamos qual requisição estamos encerrando
    // quando finalizarmos a tarefa
    Message msg = mServiceHandler.obtainMessage();
    msg.arg1 = startId;
    mServiceHandler.sendMessage(msg);

    // Se o Service for encerrado, depois de retornar aqui,
    // reiniciamos
    return START_STICKY;
}
```

```
}

@Override
public IBinder onBind(Intent intent) {
    // Não iremos permitir o 'binding', então retornamos
    // 'null'
    return null;
}

@Override
public void onDestroy() {
    Toast.makeText(this, "service done",
Toast.LENGTH_SHORT).show();
}
}
```

Acesse a documentação do Android sobre "*Extending Service*" para obter mais detalhes sobre essa implementação [\[ref 7\]](#).

6.2.3. Iniciando um Service

É possível iniciar um *Service* a partir de uma *Activity* ou outro componente da aplicação passando uma *Intent* para o método *startService()*. O *Android* chama o método *onStartCommand()* do *Service* passando a *Intent* fornecida (nunca se deve chamar *onStartCommand()* diretamente). Exemplo:

```
Intent intent = new Intent(this, HelloService.class);
startService(intent);
```

6.2.4. Encerrando um Service

Um *Service* iniciado deve gerenciar o próprio ciclo de vida. Ou seja, o *Android* não interrompe ou elimina o *Service*, a não ser que precise recuperar a memória do sistema e o *Service* continuar em execução depois que o método *onStartCommand()* retornar. Portanto, o próprio *Service* deve ser interrompido chamando *stopSelf()* ou outro componente pode interrompê-lo chamando *stopService()*.

6.2.5. Execução de Service em primeiro plano

Um *Service* de primeiro plano é aquele com que o usuário está ativamente interagindo e não é uma opção para o *Android* eliminá-lo quando a memória estiver baixa. Um *Service* de primeiro plano deve fornecer uma notificação para a barra de status, que é colocada sob o cabeçalho "Em andamento", o que significa que a notificação não pode ser dispensada a não ser que o *Service* seja interrompido ou removido do primeiro plano.

Por exemplo, um reprodutor de música que reproduz a partir de um *Service* deve ser configurado para permanecer em execução em primeiro plano, pois o usuário está prestando atenção em sua operação explicitamente. A notificação na barra de status pode indicar a música atual e permitir que o usuário inicie uma atividade para interagir com o reprodutor de música.

Para solicitar a execução do *Service* em primeiro plano, chame `startForeground()`. Esse método usa dois parâmetros: um número inteiro que identifica de forma exclusiva a notificação e *Notification** para a barra de status. Por exemplo:

```
Notification notification = ...
startForeground(ONGOING_NOTIFICATION_ID, notification);
```

*Em breve veremos como criar Push Notifications

6.2.6. Ciclo de vida de um Service

É importante ressaltar que um *Service* possui o seu próprio ciclo de vida (ver figura 3) que deve ser observado e devidamente tratado a fim de evitar perdas de recursos. Esse ciclo nos permite monitorar as alterações no estado do *Service* e realizar tarefas em momentos adequados. O exemplo a seguir demonstra cada um dos métodos de ciclo de vida:

```
import android.app.Service;
import android.content.Intent;
import android.os.IBinder;

public class ExampleService extends Service {
    private int mStartMode; // Indica como se comportar caso o
    Service seja destruído
    private IBinder mBinder; // Interface para realizar o bind
```

```
private boolean mAllowRebind; // Indica se onRebind deve
ser usado

@Override
public void onCreate() {
    // O Service está sendo criado
}

@Override
public int onStartCommand(Intent intent, int flags, int
startId) {
    // O Service está iniciando devido a uma chamada do
startService()
    return mStartMode;
}

@Override
public IBinder onBind(Intent intent) {
    // Um componente está executando uma ação de bind ao
Service utilizando bindService()
    return mBinder;
}

@Override
public boolean onUnbind(Intent intent) {
    // Todos os componentes se desligaram utilizando
unbindService()
    return mAllowRebind;
}

@Override
public void onRebind(Intent intent) {
    // Um componente está executando uma ação de bind ao
Service utilizando bindService(),
    // depois do método onUnbind() já ter sido chamado
}

@Override
public void onDestroy() {
```

```
// O serviço não está mais sendo usado e está sendo destruído
}
}
```

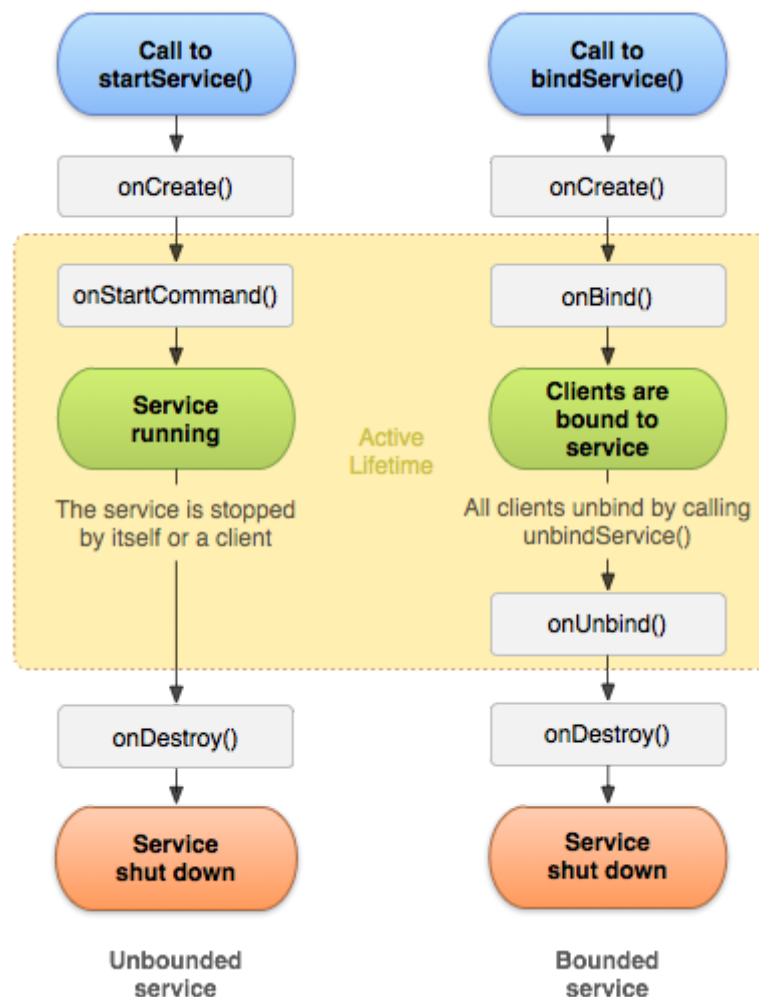


Figura 3: Ciclo de vida do *Service*. O diagrama à esquerda mostra o ciclo de vida quando o *Service* é criado com *startService()* e o diagrama à direita mostra o ciclo de vida quando o *Service* é criado com *bindService()*.

Fonte: *Google Developers - Android Documentation*

Mais informações sobre *Bounded Services* consulte a documentação [\[ref 8\]](#).

6.3. Broadcast Receivers

Broadcast Receivers simplesmente respondem a mensagens de *broadcast* de outros aplicativos ou do sistema *Android*. Além disso, aplicações também

podem iniciar eventos de *broadcast* para informar que outros aplicativos saibam que alguns dados foram baixados para o dispositivo e estão disponíveis para uso, portanto, nesse caso, o *broadcast receiver* que irá interceptar a comunicação e iniciará a ação apropriada.

Um *broadcast receiver* é implementado como uma subclasse da classe ***BroadcastReceiver*** e cada mensagem é transmitida como ***Intent***.

```
public class MyReceiver extends BroadcastReceiver {  
    public void onReceive(Context context, Intent intent) {  
    }  
}
```

Utilizamos essa implementação para detectar mudanças no sistema *Android* como informações da bateria, se o aparelho acabou de ligar ou reiniciar, se está recebendo alguma chamada e diversos outros. Além disso, conseguimos criar nossos próprios *broadcast receivers*, para tratar eventos em tempo real.

Abordaremos mais sobre essa prática mais a frente para sincronizar mensagens recebidas da nuvem e executar tarefas personalizadas. Mais informações sobre *Broadcasts* podem ser acessadas na documentação [\[ref 9\]](#).

6.4. Content Providers

Content Providers, como o próprio nome sugere, provê conteúdo, ou seja, fornece dados de uma aplicação para outra conforme alguma requisição é feita. Um *Content Provider* pode usar diferentes maneiras de armazenar seus dados que podem estar armazenados em um banco de dados, em arquivos ou até mesmo na internet.

Abordaremos mais sobre essa prática mais a frente para solicitar conteúdos armazenados no banco de dados local ou no dispositivo. Mais informações sobre *Content Providers* podem ser acessadas na documentação [\[ref 10\]](#).

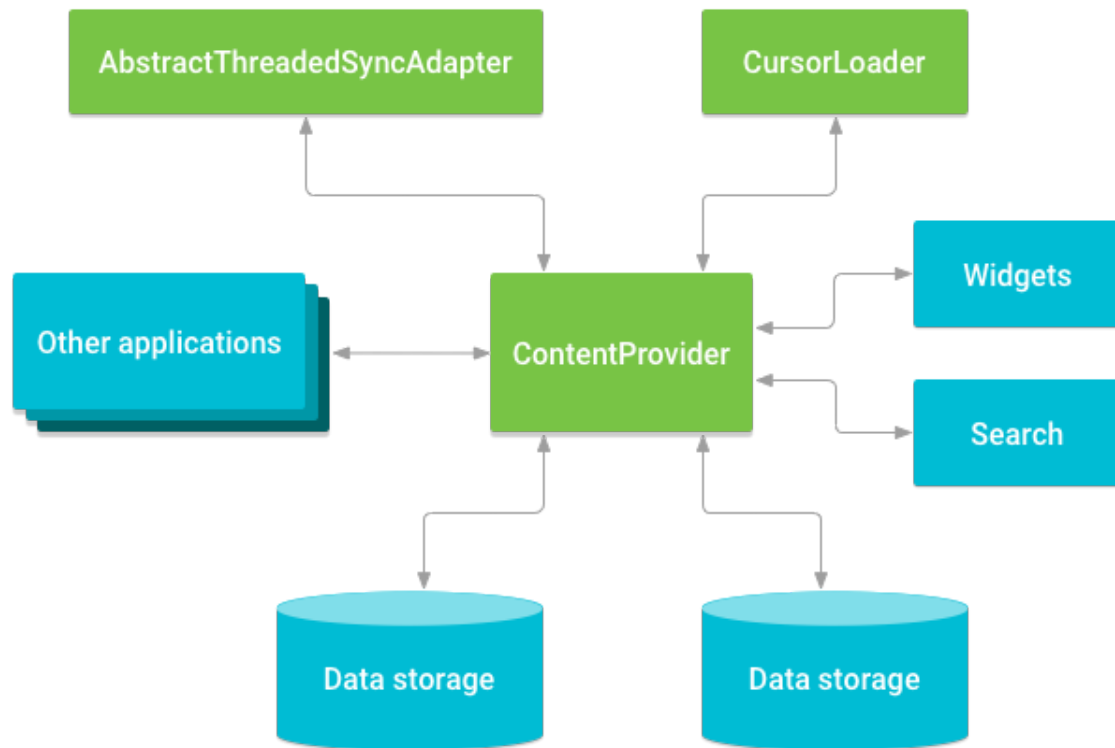


Figura 4: Relação entre *Content Providers* e outros componentes

Fonte: *Google Developers - Android Documentation*

6.5. Additional Components

6.5.1. Fragments

Um *Fragment* [ver 11] é uma parte de uma *Activity* que permite um *design* mais modular, conforme mostrado na figura 5. Podemos dizer que um *Fragment* é uma espécie de *Sub-Activity*.

É possível criar um *Fragment* estendendo a classe **Fragment** e é possível inseri-lo em uma *ActivityLayout* declarando o elemento `<fragment>`.

Também é possível estender uma das classes abaixo (que já possuem comportamentos pré-definidos bem interessantes):

- `DialogFragment` [ref 12]
- `ListFragment` [ref 13]
- `PreferenceFragment` [ref 14]
- `WebViewFragment` [ref 15]

Com a introdução do *Fragment* a partir da API 11 (*Honeycomb*) o sistema *Android* ganhou mais flexibilidade e perdeu a limitação de ter apenas uma única *Activity* na tela de cada vez. Agora é possível fazer com que essa única *Activity*

tenha vários *Fragments*, cada um com seu próprio layout, eventos e um ciclo de vida completo.

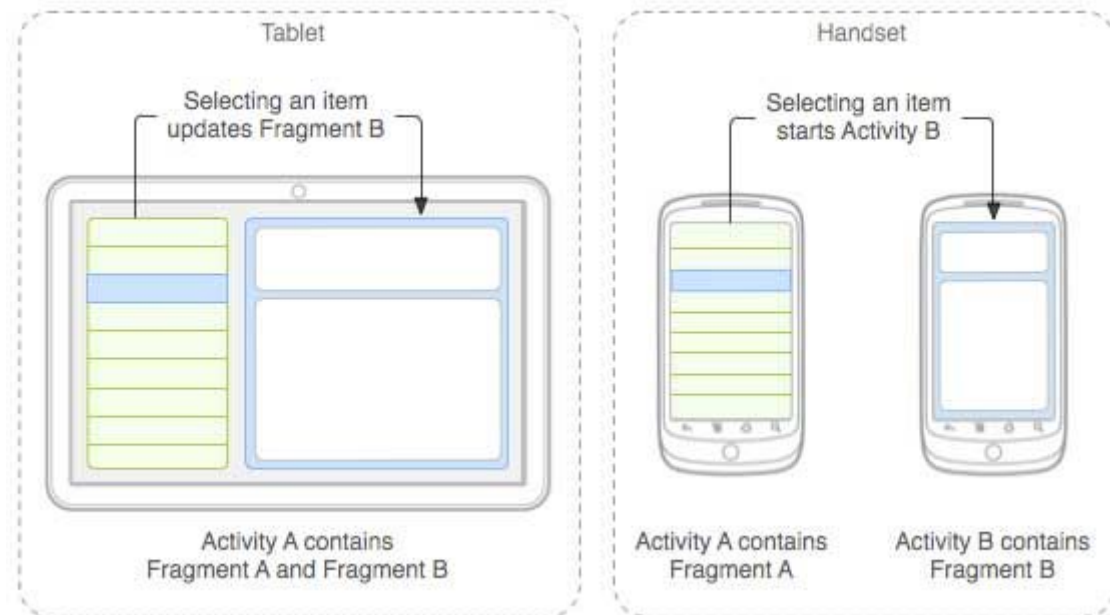


Figura 5: Um exemplo da mesma *Activity* sendo exibida em um *Tablet* e em um *Smartphone*, permitindo muito mais interatividade e controle sobre a interface de usuário.

Fonte: *Google Developers - Android Documentation*

6.5.1.1. Fragments deprecated

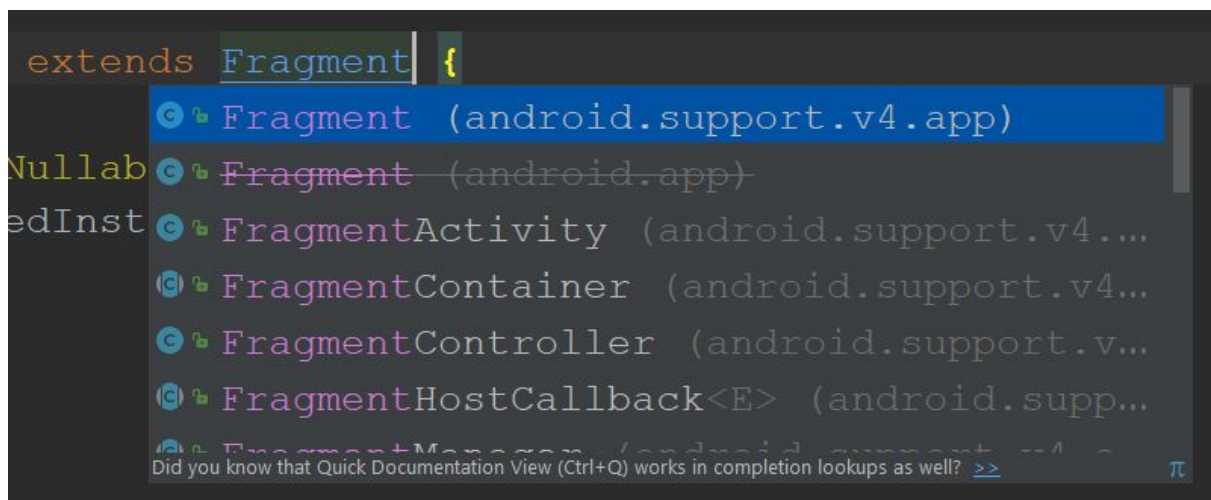
A partir da API 28 (*Android P*) a classe *Fragment* (e algumas subclasses dela citadas anteriormente) se tornou depreciada [ref 16 e ref 17] e não faz parte mais da arquitetura padrão do *Android*. Isso é uma ação comum do *Google* para alguns recursos, mas não necessariamente um problema para nós.

Quando ações como essa são tomadas, geralmente os recursos são movidos para a *Android Support Library* [ref 18], uma biblioteca de suporte preparada para oferecer uma série de recursos que não estão embutidos na estrutura padrão *Android*. Geralmente já temos uma linha de import referente ao *Support Library*.

```
implementation 'com.android.support:support-v4:27.0.0'
```

Conforme podemos observar na figura 6 e na figura 7, podemos visualizar as duas classes de *Fragment*, sendo que na API 28 em diante a classe nativa está depreciada.

Fonte: autor - *Android Studio*



Fonte: autor - *Android Studio*

Como citado anteriormente, o *Fragment* possui o seu próprio ciclo de vida, que é bastante similar ao ciclo de vida de uma *Activity* e geralmente os seguintes métodos devem ser implementados durante a criação de um *Fragment*:

- Formação em Desenvolvimento Android
por **Paulo Salvatore**

- Chamado quando é o momento de desenhar a interface pela primeira vez. Para desenhar qualquer elemento de UI devemos pegar uma *View* que é passada e corresponde a raiz do *layout* do *Fragment*.
- ***onPause()***
 - É chamado sempre que há um primeiro indício de que o usuário esteja saindo do *Fragment*, mesmo que ele não vá ser destruído. É onde geralmente devemos confirmar com o usuário qualquer informação importante que pode ser perdida.

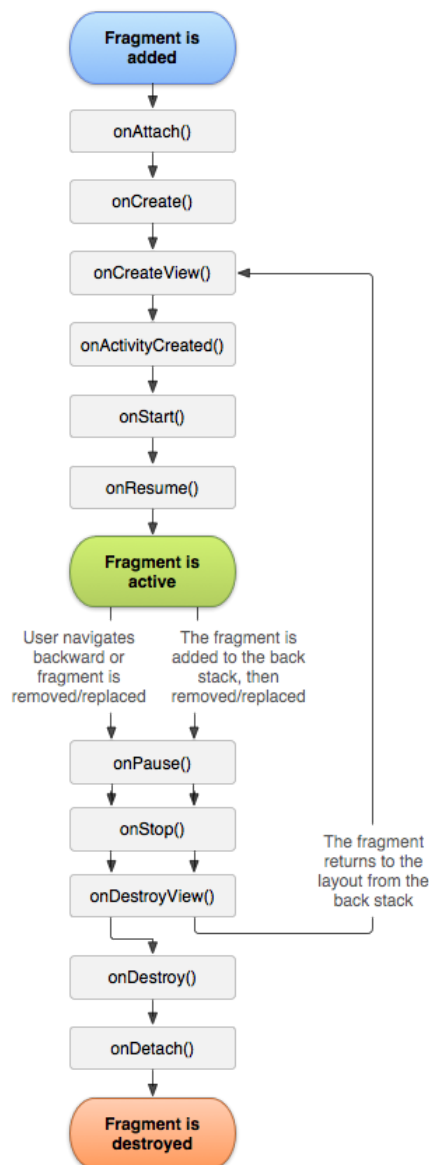


Figura 8: Ciclo de vida de um *Fragment*

Fonte: *Google Developers - Android Documentation*

6.5.1.3. Comunicação com a Activity

Apesar de um *Fragment* ser implementado como um objeto independente de uma *Activity* e poder ser usado em várias *Activities*, ele é diretamente vinculado a *Activity* que o contém no momento.

Para que o *Fragment* consiga de comunicar com a *Activity* em questão, é possível acessá-la facilmente através do método *getActivity()* e realizar facilmente tarefas como encontrar uma *View* no *layout* da *Activity*:

```
View button = getActivity().findViewById(R.id.button);
```

Do mesmo modo, a *Activity* pode buscar um *Fragment* através do *FragmentManager* usando o método *findFragmentById()* ou *findFragmentByTag()*.

```
ExampleFragment fragment = (ExampleFragment)  
getFragmentManager().findFragmentById(R.id.example_fragment);
```

6.5.2. Views

As *Views* basicamente consistem em elementos da *UI* que são desenhados na tela para que o usuário interaja, seja visualmente ou ativamente através de ações. Sendo a classe principal, todos esses elementos basicamente consistem como subclasses. Sendo possível, inclusive, criar *Views* customizadas com comportamentos pré-definidos pelo desenvolvedor.

6.5.3. Layouts

Os *Layouts* consistem em um bloco básico para construção da tela (*UI*) de um aplicativo, contendo uma ou diversas *Views* e é a base de interação do usuário com as funcionalidades do *app*.

6.5.4. Intent Filters

Intent Filters consistem em uma ferramenta muito poderosa do *Android* que permite a habilidade de executar uma *Activity* baseado não apenas em uma requisição explícita, mas também uma requisição implícita, por exemplo:

Uma requisição explícita seria "Iniciar uma *Activity* de envio de e-mail usando o aplicativo do *Gmail*".

Já uma requisição implícita seria “Iniciar uma tela de enviar e-mails em qualquer *Activity* que consiga fazer o trabalho”. Quando a *UI* do *Android* pergunta ao usuário qual aplicativo usar para realizar uma tarefa, é o *Intent Filter* que faz isso.

6.5.5. Resources

Resources são basicamente elementos externos como *strings*, constantes, imagens, cores, definições de *layout*, estilos, animações, entre outros. Elas estão presentes no dia-a-dia de um desenvolvedor desde o início da formação e devem ser sempre organizadas da melhor forma possível. Uma boa organização das *Resources* reflete em um bom desenvolvimento de aplicativo e facilita as modificações do futuro.

Todos esses elementos externos estão separados e organizados em diversos subdiretórios localizados dentro da pasta ‘res/’ do projeto.

6.5.5.1. Acessando as Resources

Quando a aplicação *Android* é compilada, a classe **R** é gerada, contendo todas as *ResourcesIDs* para todos os elementos disponíveis no diretório ‘res/’. Você pode utilizar o acesso direto a classe **R** para acessar qualquer recurso usando o nome do subdiretório em seguida no nome da *Resource* ou diretamente pelo *ResourceID*. Exemplo:

```
TextView textView = (TextView) findViewById(R.id.textView);
```

Também é possível acessar os *Resources* padrões do *Android*, presentes na classe **R** do *Android*, utilizando a declaração ‘**android.R**’. Exemplo:

```
TextView textView = (TextView) findViewById(R.id.textView);  
textView.setText(android.R.string.ok);
```

Acessar o ‘**android.R**’ possui diversas *Resources* pré configuradas que são extremamente úteis, como *drawables* e *animations*:

```
final ImageView image = (ImageView)  
findViewById(R.id.imageView);  
Animation animation = AnimationUtils.loadAnimation(  
    this,  
    android.R.anim.fade_out
```

```
) ;  
image.startAnimation(animation) ;
```

6.5.6. Manifest

O *AndroidManifest* é um arquivo de configuração que toda aplicação deve possuir. Ele fornece configurações vitais para que o aplicativo se conforme da forma esperada executando o código escrito pelo desenvolvedor.

Esse arquivo serve para inserir diversas informações gerais da aplicação como nomear o pacote do aplicativo, descrever seus componentes (como *Activities*, *Services*, ...) declarar permissões necessárias do aplicativo, entre outros.

7. Android Architecture Components

Como é possível ver no vídeo 1 ([ir para vídeo](#)), durante a Google I/O '17 tivemos a apresentação da nova biblioteca conhecida como *Android Architecture Components*, com diversas features poderosas para o desenvolvimento *Android* em diversos aspectos. Recentemente, na Google I/O '18, tivemos o anúncio de mais alguns componentes extremamente interessantes: *Navigation*, *Paging* e *WorkManager*. Uma introdução sobre esses componentes pode ser vista no vídeo 2 ([ir para o vídeo](#))

Os principais componentes que abordaremos são o *Lifecycle*, *ViewModel*, *LiveData* e o *Room*.

A biblioteca tem como objetivo auxiliar os desenvolvedores a construir aplicações que sejam "robustas, testáveis e fáceis de manter". Resumidamente ela nos ajuda a lidar melhor com a persistência de dados em eventos envolvendo ciclos de vida e alterações de configuração, além de nos ajudar a criar um aplicativo com arquitetura melhor e evitar classes cheias de repetição de códigos que são difíceis de manter e testar (leitura adicional: [ref 19](#)).

7.1. Android Jetpack

Antes de falar sobre alguns componentes importantes para essa aula, é interessante explicar um pouco sobre o que é o *Android Jetpack*.

O *Android Jetpack* [[ref 20](#)] é um conjunto de bibliotecas, ferramentas e guias de arquitetura (ver figura 9) que buscam facilitar e acelerar a construção de bons aplicativos. Ele fornece uma boa base de códigos comumente usados em aplicações para que você não precise ficar escrevendo o óbvio e consiga focar nas reais implementações da aplicação.

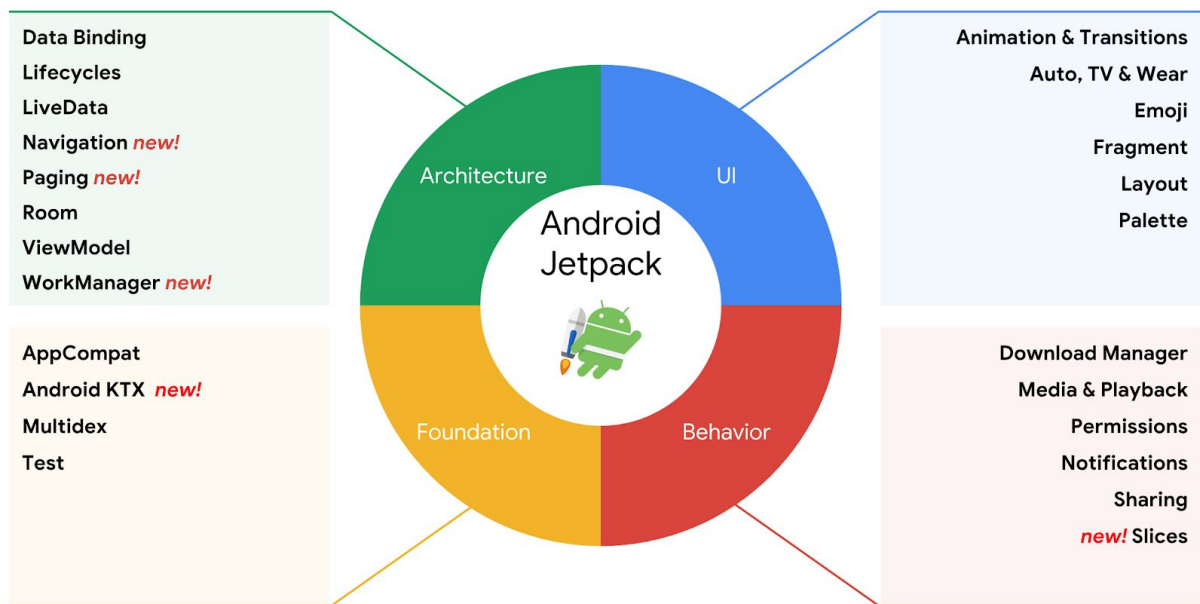


Figura 9: Componentes presentes no Jetpack anunciados na Google I/O '17 e '18
Fonte: Android Developers - Google Blog [[ref 21](#)]

Nessa aula abordaremos apenas os aspectos de arquitetura do *Android Jetpack* e falaremos sobre os aspectos principais de cada implementação. Antes de prosseguir neste capítulo, dê uma olhada no vídeo 3 ([ir para o vídeo](#)), que faz uma breve introdução sobre os componentes que trataremos aqui.

7.2. Lifecycles, ViewModel e LiveData

Os componentes *Lifecycles*, *ViewModel* e *LiveData*, quando combinados, fornecem um grande potencial para aplicação tanto em termos de funcionalidade quanto em termos de performance e robustez.

Lifecycles [[ref 22](#)] basicamente é composto de duas interfaces principais, *LifecycleOwner* e *LifecycleRegistryOwner*, sendo que ambas são implementadas nas classes *AppCompatActivity* e *SupportFragment*. É possível atrelar outros componentes para objetos que implementam essas interfaces (que chamaremos de *owners*) para detectar mudanças no ciclo de vida do seu respectivo *owner*.

ViewModel [[ref 23](#)] é uma classe responsável por preparar e manusear dados de uma *Activity* ou um *Fragment*, fornecendo uma maneira de criar e acessar objetos que estão conectados a um ciclo de vida específico (como é possível observar na figura 10). Um *ViewModel* tipicamente armazena o estado dos dados de uma *View* e se comunica com outros componentes, como *data repositories* ou a *domain layer* (leitura adicional: [ref 24](#)) que lida com o que chamamos de *business logic*.

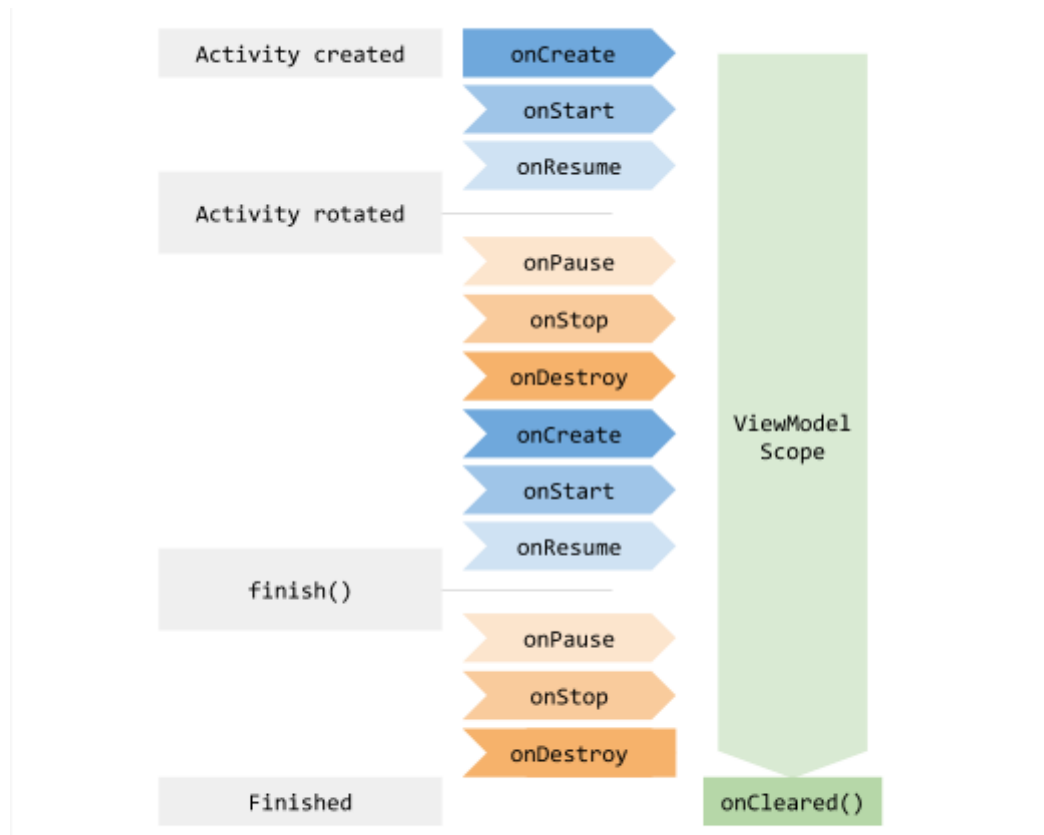


Figura 10: Relação entre o *ViewModel* e o ciclo de vida de uma *Activity/Fragment*

Fonte: *Google Developers - Android Documentation*

LiveData [ref 25] permite com que você 'observe' mudanças dos dados de vários componentes da sua aplicação sem a necessidade de criar estruturas complexas de códigos de dependência entre eles. Assim como a *ViewModel*, *LiveData* também respeita o complexo ciclo de vida dos componentes, incluindo *Activities*, *Fragments*, *Services* e qualquer outro *LifecycleOwner* da aplicação. *LiveData* gerencia *observers* acoplados pausando os acoplamentos para objetos cujo ciclo de vida foram parados e desacopla totalmente os objetos cujo ciclo de vida foi finalizado.

Se quiser praticar os componentes de arquitetura citados nesse capítulo recomendo que faça o *CodeLab Android lifecycle-aware components* [ref 26]. Usaremos esse codelab como principal referência para nosso exemplo prático do capítulo seguinte.

7.2.1. Exemplo prático

Neste capítulo desenvolveremos uma aplicação simples que utiliza os componentes citados e mostrando as suas respectivas características de declaração

e possibilidades interessantes. Crie um projeto com as configurações padrões e com uma Activity chamada ViewModelActivity.

Antes de iniciar o desenvolvimento do projeto, iremos configurar devidamente as dependências necessárias. Qualquer problema durante a configuração, consulte a documentação, disponível acessando a [referência](#) 28.

Dependências para *Lifecycle*, incluindo *LiveData* e *ViewModel*:

```
dependencies {
    def lifecycle_version = "1.1.1"

    // ViewModel and LiveData
    implementation
    "android.arch.lifecycle:extensions:$lifecycle_version"
    // alternatively - just ViewModel
    implementation
    "android.arch.lifecycle:viewmodel:$lifecycle_version" // use
    -ktx for Kotlin
    // alternatively - just LiveData
    implementation
    "android.arch.lifecycle:livedata:$lifecycle_version"
    // alternatively - Lifecycles only (no ViewModel or
    LiveData).
    // Support library depends on this lightweight import
    implementation
    "android.arch.lifecycle:runtime:$lifecycle_version"

    annotationProcessor
    "android.arch.lifecycle:compiler:$lifecycle_version"
    // alternately - if using Java8, use the following instead
    of compiler
    implementation
    "android.arch.lifecycle:common-java8:$lifecycle_version"

    // optional - ReactiveStreams support for LiveData
    implementation
    "android.arch.lifecycle:reactivestreams:$lifecycle_version"

    // optional - Test helpers for LiveData
```

```
testImplementation
"android.arch.core:core-testing:$lifecycle_version"
}
```

Note que na declaração acima existem várias implementações opcionais. Para funcionar no meu projeto irei implementar as seguintes bibliotecas:

```
dependencies {
    def lifecycle_version = "1.1.1"

    // ViewModel and LiveData
    implementation
    "android.arch.lifecycle:extensions:$lifecycle_version"
    // Support library depends on this lightweight import
    implementation
    "android.arch.lifecycle:runtime:$lifecycle_version"

    annotationProcessor
    "android.arch.lifecycle:compiler:$lifecycle_version"

    // optional - Test helpers for LiveData
    testImplementation
    "android.arch.core:core-testing:$lifecycle_version"
}
```

Para iniciar nossos testes, criamos uma *Empty Activity* e aplicamos o *layout* abaixo:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".ViewModelActivity">

    <Chronometer
```

```
        android:id="@+id/chronometer"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerVertical="true"
        android:layout_centerHorizontal="true" />

    </RelativeLayout>
```

O código da *ViewModelActivity* por enquanto será bem básico, apenas pegamos a referência do cronômetro e aplicamos o método *start()*:

```
import kotlinx.android.synthetic.main.activity_view_model.*

class ViewModelActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        chronometer.start()
    }
}
```

Com isso conseguimos fazer o primeiro teste da aplicação. Note na figura figura 11 que quando iniciamos a *Activity* o cronômetro automaticamente inicia, porém, quando alteramos a orientação do dispositivo (alterando o ciclo de vida da *Activity*) perdemos a referência do valor do cronômetro.

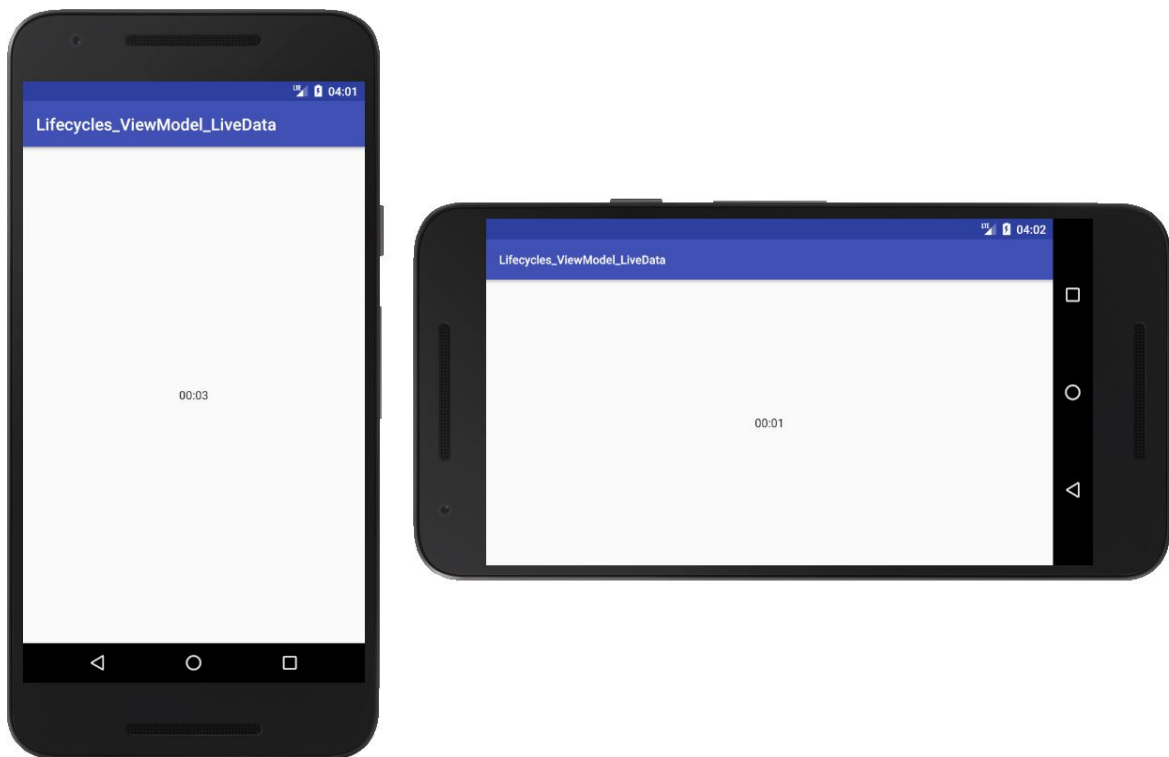


Figura 11: Perdendo referência de dados do cronômetro na troca de orientação

Fonte: autor - *Android Emulator (API 24)*

Como sabemos, existem diversas maneiras de se persistir dados no *Android* (leitura adicional: [ref 29](#)), porém, para várias abordagens precisamos escrever um grande quantidade de *boilerplate* para tratar a persistência de maneira inteligente, levando em consideração os ciclos de vida do contexto que estamos e tomando muito cuidado para evitar o *MemoryLeak*, muito comum nesse tipo de abordagem.

Conforme podemos ver no vídeo 2 ([ir para vídeo](#)), em 2016 o *Google* fez diversas pesquisas com vários desenvolvedores perguntando qual era a parte mais difícil no desenvolvimento *Android* e a resposta foi, de longe, a gestão dos ciclos de vida. Pensando justamente em resolver esse problema, o *Google* criou uma série de componentes de arquitetura que tem como objetivo facilitar e otimizar o processo de desenvolvimento utilizando os conceitos elevados e tornando a gestão de recursos algo menos burocrático.

A primeira modificação foi a implementação da interface *LifecycleOwner* na classe *SupportsActivity* (base para as *Activities* que criamos) permitindo gerenciar melhor o ciclo de vida e utilizar componentes como o *ViewModel*. Primeiro, para criar nosso *ViewModel*, iremos criar uma nova classe chamada *ChronometerViewModel* no mesmo pacote da nossa *Activity*. Essa nova classe deverá estender a classe *ViewModel*, conforme é possível ver na figura 12.

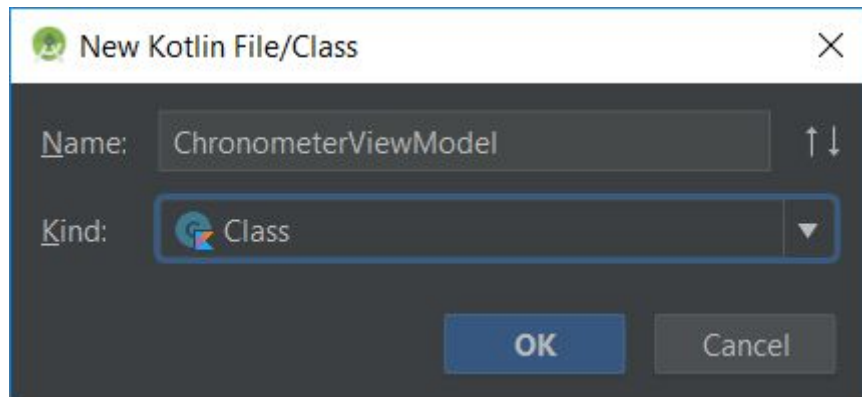


Figura 12: Criação da classe *ChronometerViewModel*

Fonte: autor - *Android Studio*

Essa nova classe possuirá basicamente as informações que queremos persistir que, no caso, será o tempo de início do cronômetro conforme o código abaixo:

```
import android.arch.lifecycle.ViewModel

class ChronometerViewModel : ViewModel() {
    var startTime: Long = 0L
}
```

Para inicializar corretamente um *ViewModel* dentro de uma *Activity*, precisamos pegar seu objeto de instância que é fornecido pela classe *ViewModelProviders*, utilizando a seguinte declaração:

```
val chronometerViewModel = ViewModelProviders
    .of(this)
    .get(ChronometerViewModel::class.java)
```

A classe *ViewModelProviders* fornecerá um novo *ViewModel* ou um que tenha sido criado anteriormente.

Com isso, podemos utilizar quaisquer métodos para obter dados que irão ser persistidos levando em consideração todo o ciclo de vida, como mostrado na figura 10.

```
if (chronometerViewModel.startTime == 0L) {
```

```
val startTime = SystemClock.elapsedRealtime()
chronometerViewModel.startTime = startTime
chronometer.base = startTime
} else {
    chronometer.base = chronometerViewModel.startTime
}

chronometer.start()
```

Portanto, o código da *ViewModelActivity* ficará assim:

```
import android.os.Bundle
import android.support.v7.app.AppCompatActivity
import kotlinx.android.synthetic.main.activity_view_model.*
import android.arch.lifecycle.ViewModelProviders
import android.os.SystemClock

class ViewModelActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        val chronometerViewModel = ViewModelProviders
            .of(this)
            .get(ChronometerViewModel::class.java)

        if (chronometerViewModel.startTime == 0L) {
            val startTime = SystemClock.elapsedRealtime()
            chronometerViewModel.startTime = startTime
            chronometer.base = startTime
        } else {
            chronometer.base = chronometerViewModel.startTime
        }

        chronometer.start()
    }
}
```

Prosseguindo com nosso exemplo, faremos agora alguns testes utilizando um *ViewModel* em conjunto com o *LiveData*. Como citado anteriormente, o *LiveData* permite com que o valor seja observado, criando um *observer*, que também leva em consideração o ciclo de vida e nos permite atualizar valores em tempo real a partir de qualquer alteração de dados detectada.

Crie uma nova *Empty Activity* chamada *LiveDataActivity* e adicione o seguinte *layout*:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".LiveDataActivity">

    <TextView
        android:id="@+id/tvTimer"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerHorizontal="true"
        android:layout_centerVertical="true" />

</RelativeLayout>
```

Com o *layout* configurado, antes de criar o conteúdo da nossa *Activity*, vamos criar uma nova classe que estenda a classe *ViewModel* (como mostrado na figura 13) e prepará-la para utilizar algumas funcionalidades interessantes do *LiveData*.

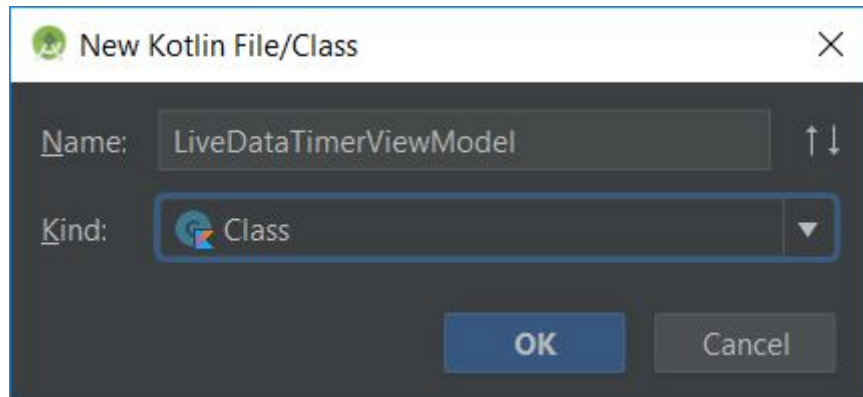


Figura 13: Criação da classe *LiveDataTimerViewModel*.

Fonte: autor - *Android Studio*

Como pode perceber, a criação do *ViewModel* continua a mesma como foi feito anteriormente. A diferença aqui é a declaração das variáveis que irão conter os dados que, em vez do tipo primitivo, iremos utilizar a subclasse *MutableLiveData<T>*, que por sua vez estende a classe *LiveData<T>*, como feito a seguir:

```
val elapsedTime = MutableLiveData<Long>()
```

Nesse exemplo iremos atualizar o valor da variável dentro do *ViewModel* através do *LiveData* e aplicaremos um *post* no valor, para que qualquer *observer* que esteja esperando uma modificação seja notificado e possa atualizar seus devidos componentes. Perceba que mesmo que o valor 'elapsedTime' irá alterar, a variável é um 'val', pois a informação armazenada agora é um objeto da classe 'MutableLiveData', que aceita valores do tipo Long.

Prepararemos o construtor da nossa classe, que será basicamente um objeto da classe *Timer* e irá atualizar o valor a cada 1000 milissegundos, publicando essa atualização para o *LiveData*. O construtor ficará assim:

```
companion object {
    private val ONE_SECOND = 1000L
}

init {
    val initialTime = SystemClock.elapsedRealtime()
    val timer = Timer()
    timer.schedule(object: TimerTask() {
```



```

        override fun run() {
            val newValue = (SystemClock.elapsedRealtime() -
initialTime) / 1000

            // setValue() não pode ser chamado de uma
            // BackgroundThread, portanto, devemos usar
            // o postValue para a MainThread.
            elapsedTime.postValue(newValue)
        }
    }, ONE_SECOND, ONE_SECOND)
}

```

Com isso, nosso *ViewModel* está pronto e o *LiveData* devidamente implementado. Agora podemos utilizá-lo em qualquer *Activity* ou *Fragment* que desejamos utilizar os valores armazenados. Na nossa *LiveDataActivity* iremos pegar normalmente o objeto *ViewModel* e iremos configurar um *observer* para o *LiveData*, que será feito de um novo método que chamaremos de *subscribe()*. Portanto, nossa *Activity* ficará assim:

```

import android.os.Bundle
import android.support.v7.app.AppCompatActivity
import android.arch.lifecycle.ViewModelProviders
import android.annotation.SuppressLint
import android.arch.lifecycle.Observer
import android.util.Log
import kotlinx.android.synthetic.main.activity_live_data.*

class LiveDataActivity : AppCompatActivity() {
    private val liveDataTimerViewModel by lazy {
        ViewModelProviders
            .of(this)

        .get<LiveDataTimerViewModel>(LiveDataTimerViewModel::class.java)
    }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
    }
}

```

```
        setContentView(R.layout.activity_main)

        subscribe()
    }

    @SuppressWarnings("SetTextI18n")
    private fun subscribe() {
        val elapsedTimeObserver = Observer<Long> {
            Log.d("ElapsedTimeObserver", "Updating timer")
            tvTimer.text = "$it segundos se passaram."
        }

        liveDataTimerViewModel.elapsedTime.observe(this,
elapsedTimeObserver)
    }
}
```

Antes de testar o funcionamento da nossa *Activity*, vamos configurá-la corretamente no manifesto para executá-la pelo *Run/Debug Configurations*.

Declare o `<intent-filter>` da *Activity*:

```
<application>
    <activity android:name=".LiveDataActivity">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />

            <category
android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
</application>
```

Criar uma nova configuração de execução no *Run/Debug Configurations*, conforme mostrado na figura 14, e em seguida execute a aplicação.

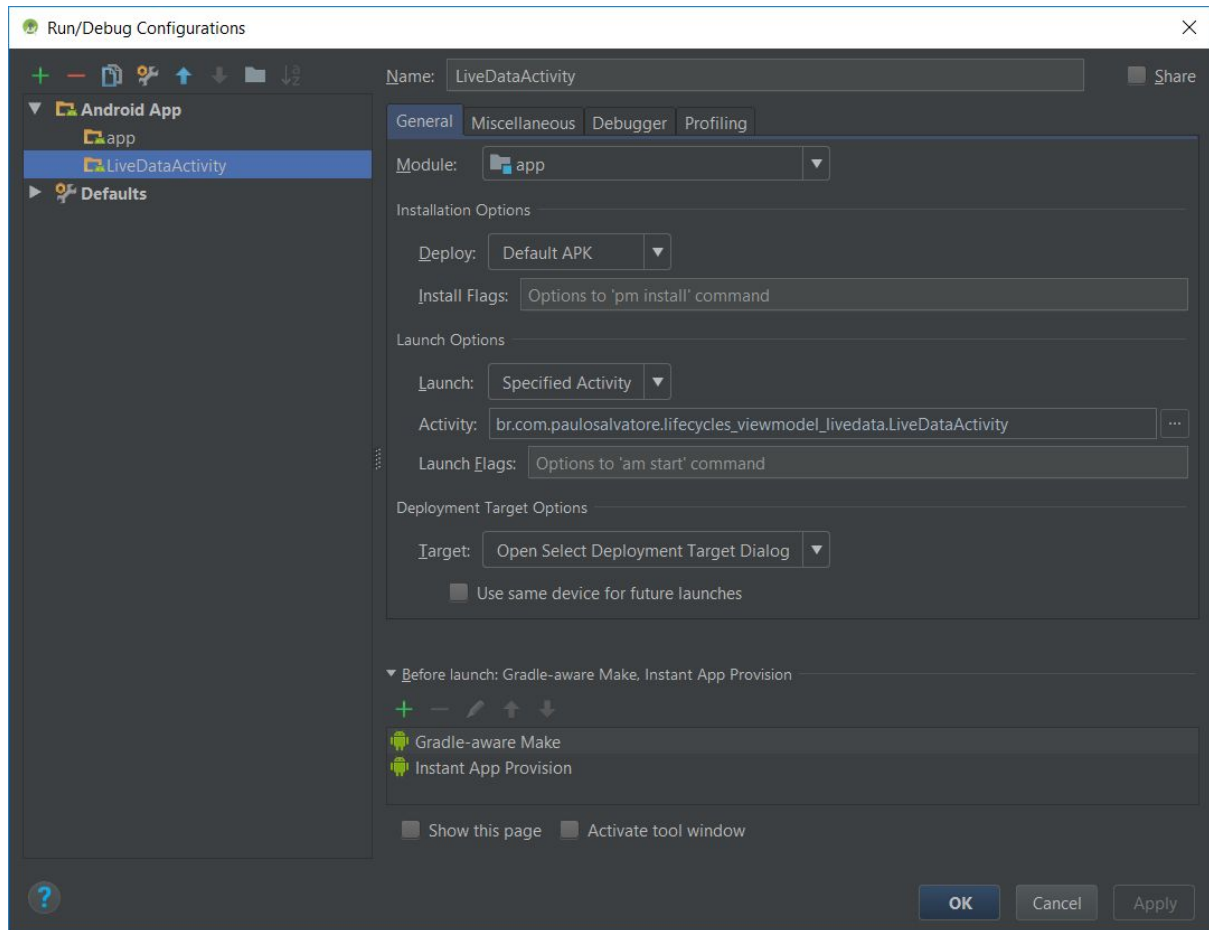


Figura 14: Configuração da *LiveDataActivity*

Fonte: autor - *Android Studio*

Como é possível observar na figura 15, a aplicação imediatamente começa a atualizar o valor conforme o *post* do *LiveData* é realizado. Note também nos registros de *logs* do *Android Studio* que a atualização do *timer* só é feita quando a aplicação está em foco. Experimente alternar para outra aplicação e note que a execução irá parar, porém, quando voltar, a aplicação prosseguirá normalmente com os dados persistidos e as mesmas lógicas de atualização.

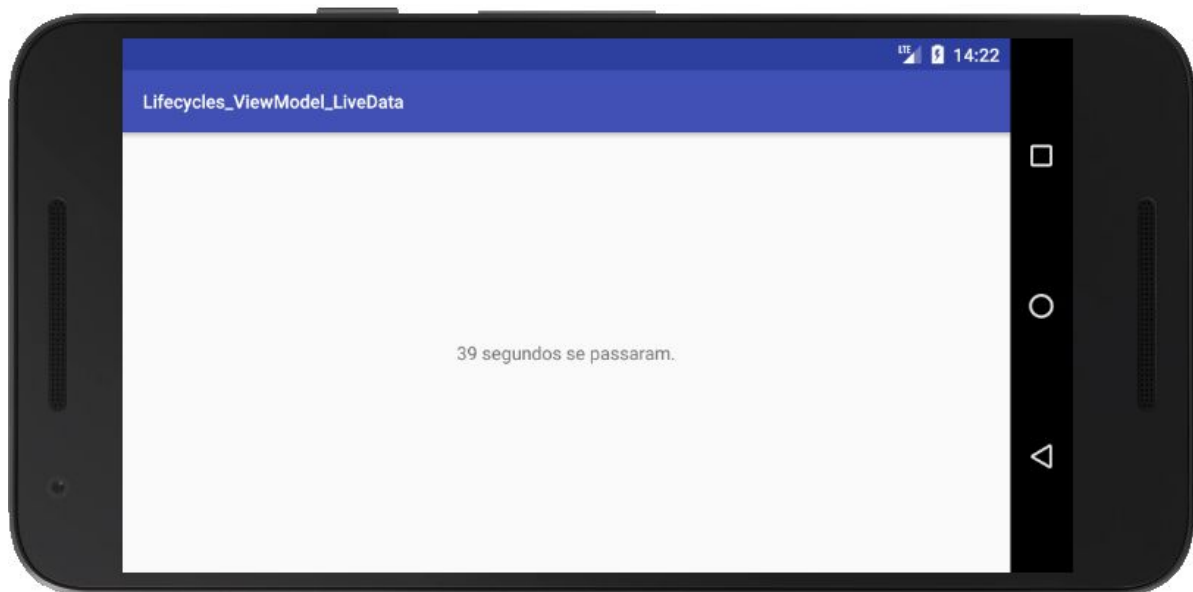


Figura 15: *LiveDataActivity* sendo executada

Fonte: autor - *Android Emulator (API 24)*

Como podemos perceber, o *ViewModel* e o *LiveData* trabalham muito bem juntos, cumprindo tudo o que prometem. Uma outra possibilidade bem interessante é utilizá-los em múltiplas atualizações de estado. Para o nosso próximo exemplo, iremos criar uma terceira *Activity* que contenha dois *Fragments*, ambos irão referenciar a mesma classe de *Fragment*, mas funcionarão de forma completamente independente. Crie uma *Empty Activity* chamada *SeekBarActivity*, um *Blank Fragment* chamado *SeekBarFragment* (sem os *factory methods* e sem os *callbacks* da interface). Não se esqueça que o *SeekBarFragment* deve estender à classe *Fragment* presente dentro do pacote de *support* do *Android*.

No *layout* da *Activity* criada, insira o seguinte código XML (certifique-se de completar o nome do pacote do *fragment SeekBarFragment* - Dica: apague o conteúdo inteiro das aspas, pressione Ctrl+Espaço e selecione opção correspondente):

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
```

```
tools:context=".SeekBarActivity">

<fragment
    android:id="@+id/fragment1"
    android:name=".SeekBarFragment"
    android:layout_width="match_parent"
    android:layout_height="0dp"
    android:layout_weight="1" />

<fragment
    android:id="@+id/fragment2"
    android:name=".SeekBarFragment"
    android:layout_width="match_parent"
    android:layout_height="0dp"
    android:layout_weight="1" />
</LinearLayout>
```

No *layout* do *Fragment*, adicione o seguinte código:

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".SeekBarFragment">

    <SeekBar
        android:id="@+id/seekBar"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />

</FrameLayout>
```

Com os layouts definidos, precisaremos criar o *ViewModel* que será responsável por atualizar os dados com o *LiveData* a cada vez que uma das *SeekBars* do *Fragment* for atualizada, disparando automaticamente uma atualização para outras *SeekBars* presentes no mesmo contexto.

Inicialmente criaremos uma classe chamada *SeekBarViewModel*, que deverá estender a classe *ViewModel* e adicionar o atributo *seekbarValue*, envolvido pela classe *LiveData*, que nos permitirá observar alterações nessa variável:

```
import android.arch.lifecycle.MutableLiveData
import android.arch.lifecycle.ViewModel

class SeekBarViewModel : ViewModel() {
    val seekbarValue = MutableLiveData<Int>()
}
```

Depois, criaremos as variáveis dentro do *SeekBarFragment* da *SeekBar* e do *ViewModel* criado anteriormente:

```
private lateinit var seekBar: SeekBar
private lateinit var seekBarViewModel: SeekBarViewModel
```

Sobrescreveremos o método *onCreateView()* para pegar as informações desejadas, armazená-las nas variáveis a escrita de valores conforme a observação feita pela *LiveData*:

```
override fun onCreateView(inflater: LayoutInflater,
    container: ViewGroup?, savedInstanceState: Bundle?): View? {
    val root = inflater.inflate(R.layout.fragment_seek_bar,
        container, false)
    seekBar = root.findViewById(R.id.seekBar)

    activity?.let {
        seekBarViewModel =
            ViewModelProviders.of(it).get(SeekBarViewModel::class.java)

        subscribeSeekBar()
    }

    return root
}
```

Por último, declaramos o método `subscribeSeekBar()` que será responsável por atualizar o valor do `LiveData` dentro do `ViewModel` e também de detectar alterações feitas neles por outras fontes de informação:

```
private fun subscribeSeekBar() {
    seekBar.setOnSeekBarChangeListener(object :
SeekBar.OnSeekBarChangeListener {
        override fun onProgressChanged(seekBar: SeekBar,
progress: Int, fromUser: Boolean) {
            if (fromUser) {
                seekBarViewModel.seekbarValue.value = progress
            }
        }

        override fun onStartTrackingTouch(seekBar: SeekBar) {}

        override fun onStopTrackingTouch(seekBar: SeekBar) {}
    })

    seekBarViewModel.seekbarValue.observe(this, Observer<Int>
{ progress ->
    seekBar.progress = progress ?: 0
})
}
```

7.3. Persistence - Data Storage com Room

A persistência de dados é um assunto de extrema importância no universo *Android*. Dentro dos *Architecture Components*, como é possível perceber, temos um componente chamado *Room Persistence Library* [\[ref 30\]](#), que propõe reinventar a maneira com que trabalhamos com o *SQLite*, evitando muito *boilerplate* a ser escrito pelos desenvolvedores, prevendo diversos erros e servindo como um grande guia de arquitetura avançada.

7.3.1. O que precisamos entender para começar?

Antes de citar os benefícios do *Room* precisamos entender um pouco dos desafios quando estamos desenvolvendo com *SQL* puro:

- Eles são relativamente de “baixo-nível” e requerem muito tempo de desenvolvimento e um grande esforço.
- *SQL Queries* puras não são verificadas durante o *compile-time*.

- Você precisa manualmente atualizar suas *queries* para refletir alterações na visualização dos seus dados. Esse processo consome um tempo desnecessário e é propenso a erros.
- Você deve sempre escrever e manter muito código *boilerplate* para converter as *queries* para objetos de dados (*data objects*).

Com esses problemas em mente, o time do *Google* construiu esse componente buscando melhorar o fluxo de desenvolvimento e torna mais fácil a maneira com que trabalhamos com esse tipo de armazenamento de dados e fornecendo acesso a implementação uma arquitetura avançada de uma maneira muito mais fácil. As principais características do *Room* são:

- Durante o *compile-time*, o *Room* valida cada *query* em relação ao *scheme* garantindo que qualquer *SQL Query* incorreta resulte em erros durante a compilação em vez de erros durante a execução.
- *Room* abstrai alguns dos detalhes “baixos” de implementação quando trabalhamos direto com o *SQL* puro.
- É possível usar o *Room* para observar alterações nos dados armazenados em um banco de dados e expôr as modificações como objetos *LiveData*, que irá atualizá-las onde for necessário levando em consideração o ciclo de vida do contexto.
- *Room* também definirá restrições de *threading*, que te ajudará a resolver problemas comuns que podem afetar negativamente o desempenho do seu *app*, como acessar dados a partir da *MainThread*.

7.3.2. Arquitetura por trás do Room

Durante nosso exemplo prático com *Room* utilizaremos uma arquitetura básica de estruturação dos nossos componentes, conforme é possível observar na figura 16.

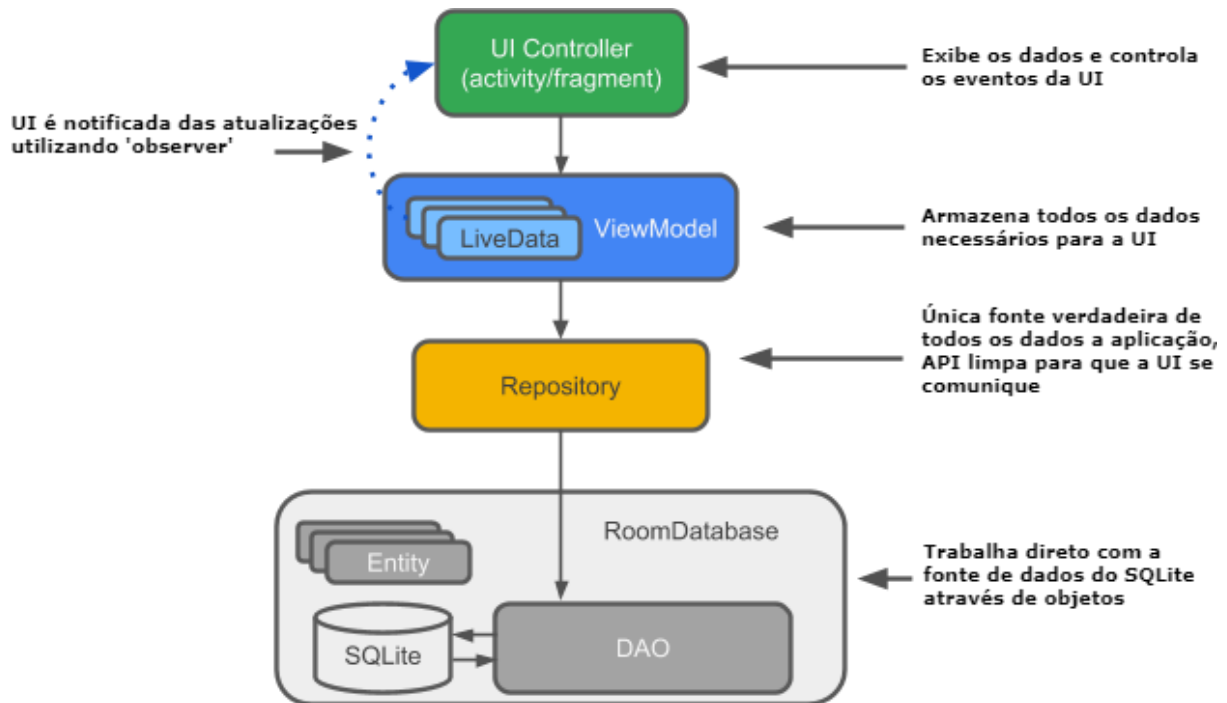


Figura 16: Diagrama básico da arquitetura que utilizamos com o *Room*

Fonte: *Google Developers - CodeLabs* (adaptado)

Traduzindo essa diagrama para nossa aplicação, basicamente criaremos as *Activities* onde o usuário irá interagir e depois criaremos todo o fluxo por trás que será responsável por armazenar os dados inseridos utilizando os componentes *ViewModel*, *LiveData* e *Room*, além do *Repository*, que não é obrigatório, porém, te permitirá trabalhar com múltiplas fontes de dados. O novo diagrama pode ser observado na figura 17.

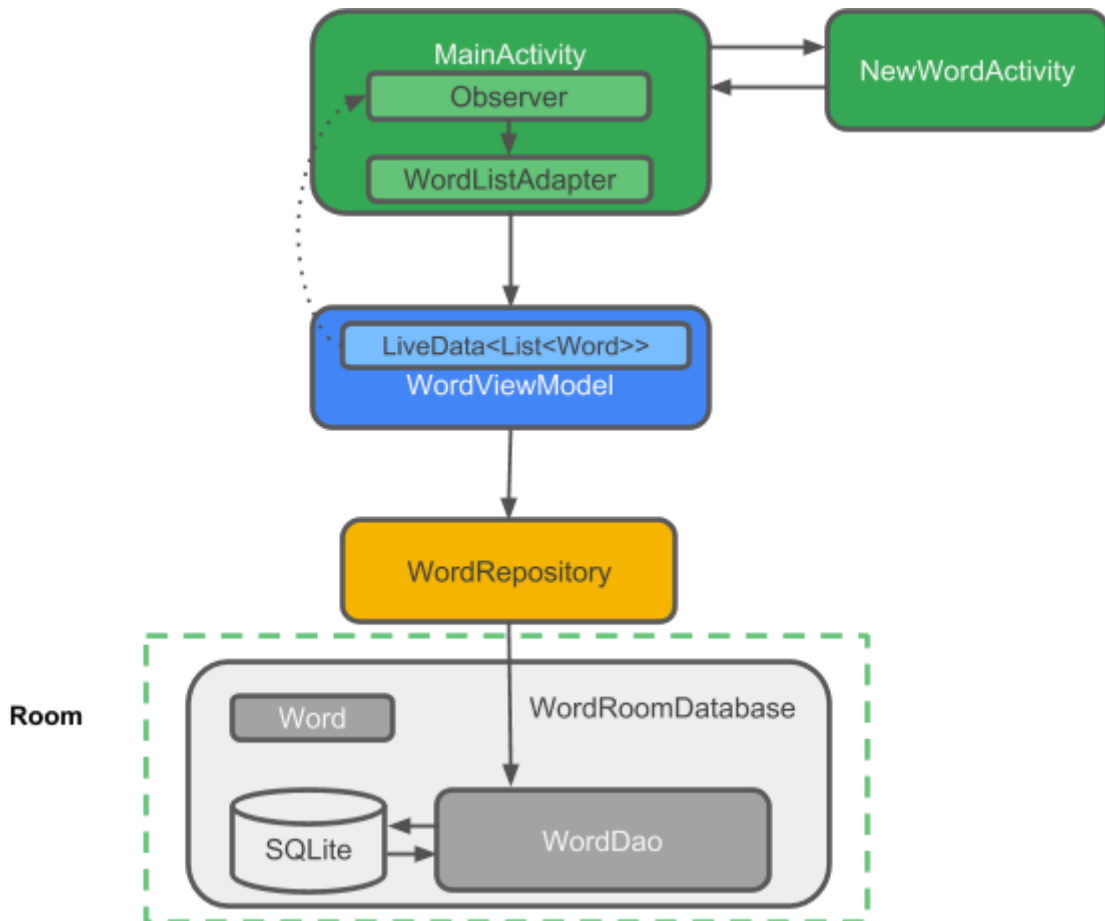


Figura 17 - Diagrama da aplicação que será desenvolvida

Fonte: Google Developers - CodeLabs

7.3.3. Iniciando o projeto

Para começar, crie um novo projeto chamado '*AndroidRoom*' sob o *company domain* 'mobile.next.com.br' em um local que achar apropriado para fazer o *backup* do projeto posteriormente. Nesse projeto não utilizaremos o suporte a *Kotlin* ou *C++*. API mínima deve ser API 20 e a nossa *MainActivity* deverá iniciar com o padrão de uma *BasicActivity*.

O projeto que você verá aqui é baseado em um *CodeLab* do *Google* algumas modificações no fluxo e nas informações apresentadas [ref 27].

Com o projeto criado, abra o arquivo do *gradle* do *app* e implemente as seguintes dependências:

```
apply plugin: "kotlin-kapt"
// ...
// Room components
```

```
implementation
"android.arch.persistence.room:runtime:$roomVersion"
kapt "android.arch.persistence.room:compiler:$roomVersion"
androidTestImplementation
"android.arch.persistence.room:testing:$roomVersion"

// Lifecycle components
implementation
"android.arch.lifecycle:extensions:$archLifecycleVersion"
kapt "android.arch.lifecycle:compiler:$archLifecycleVersion"
```

Além disso, declare o seguinte conteúdo no final do arquivo do *gradle* do projeto:

```
buildscript {
    // ...
    ext.roomVersion = '1.1.1'
    ext.archLifecycleVersion = '1.1.1'
}
```

Trabalharemos com essas versões de implementação, mas saiba que é possível encontrar as versões mais atualizadas dos componentes na própria documentação do *Google* [\[ref 31\]](#).

7.3.4. Criando a primeira Entity

Sempre que vamos iniciar um projeto no *Room* ou em qualquer outra ferramenta que envolva *SQL* devemos começar pelo planejamento da base de dados. Para essa aplicação nossa tabela será muito simples e apenas conterá uma palavra (do tipo *String*), que o usuário poderá adicionar novas posteriormente. Para isso criaremos uma *Entity*, que saberá todas a estrutura da tabela e como construir e manipular os registros dela.

Por tanto, crie um pacote chamado ``entitites`` e insira uma nova ``data class`` para nossa *Entity* e declare o atributo `word - String`, como no exemplo abaixo:

```
data class Word(val word: String)
```

Pronto! Criamos o objeto exato de um registro da nossa tabela, porém, para que o *Room* saiba que deverá olhar para ele, precisamos de algumas anotações:

- **@Entity(tableName = "word_table")**
 - Cada *Entity* representa uma entidade na tabela. O atributo *tableName* só deverá ser especificado se você quiser que ele seja diferente no nome da classe.
- **@PrimaryKey**
 - Toda entidade precisa de uma chave primária, para simplificar esse exemplo, utilizaremos a própria palavra como chave.
- **@NonNull**
 - Declare essa anotação sempre que um campo não possa ser nulo.
- **@ColumnInfo(name = "word")**
 - Especifique o nome da coluna na tabela que corresponde a variável anotada apenas se você deseja que seja diferente no nome da variável.
- Em Java, cada campo na tabela precisa de um *getter* definido. Nesse exemplo em Kotlin temos o **getWord()** definido automaticamente pois a variável é ``val``.

Seu código atualizado deverá ficar assim:

```
import android.arch.persistence.room.ColumnInfo
import android.arch.persistence.room.Entity
import android.arch.persistence.room.PrimaryKey
import android.support.annotation.NonNull

@Entity(tableName = "word_table")
data class Word(@PrimaryKey
                @NonNull
                @ColumnInfo(name = "word") // opcional
                val word: String)
```

7.3.5. Criando o DAO para a Entity

O *DAO* (*data access object*) é quem faz toda a lógica *SQL* utilizando as *queries* e sabe como transformar a sua informação (objeto) em sintaxe *SQL* e vice-versa.

O *DAO* sempre precisa ser uma interface ou uma classe abstrata e por padrão todas as *queries* são executadas em *threads* separadas (essa é uma das maiores mágicas do *Room*).

Para implementá-lo nesse exemplo, iremos começar criando a interface *WordDao*, anotando-a com o *@Dao* para identificá-la ao *Room*. Posterior a isso, iremos criar um método para inserir novos registros e um outro método para deletar todos os registros existentes. Após criar cada método, devemos anotá-lo com a anotação apropriada. O *Room* possui algumas anotações padrões (que irá gerar *SQL* para você) para apenas alguns casos, mais informações sobre esses motivos é possível encontrar no vídeo 4 ([ir para vídeo](#)) e também recomendo dar uma olhada nas anotações *RawQueries*, divulgadas na Google I/O 18" e que potencializaram a escrita de algumas *queries*, como é possível ver no vídeo 2 ([ir para vídeo](#)). O código da interface *WordDao* ficará assim:

```
import android.arch.lifecycle.LiveData
import android.arch.persistence.room.Dao
import android.arch.persistence.room.Insert
import android.arch.persistence.room.OnConflictStrategy
import android.arch.persistence.room.Query

@Dao
interface WordDao {
    /*
    Nossa chave primária é a palavra
    Default SQL Behavior para conflito é ABORT para que não
    seja possível inserir dois itens com a mesma primary key
    no banco
    Se a tabela tiver mais que uma coluna é possível usar:
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    */
    @Insert(onConflict = OnConflictStrategy.IGNORE)
    fun insert(word: Word)

    @Query("DELETE FROM word_table")
    fun deleteAll()

    @Query("SELECT * FROM word_table ORDER BY word ASC")
    fun getAllWords(): LiveData<List<Word>>
}
```

7.3.6. Criando o RoomDatabase

O *RoomDatabase* é a camada acima do banco de dados *SQLite* responsável por trabalhar com o que estávamos acostumados a fazer utilizando o *SQLiteOpenHelper*.

Para começar, vamos criar um pacote ``database`` e insira uma nova classe abstrata *WordRoomDatabase* que estende a classe *RoomDatabase*. Anotamos a nova classe com `@Database(entities = {Word::class}, version = 1)` e fornecemos um método abstrato *getter* para cada *DAO* existente, conforme o código a seguir:

```
import android.arch.persistence.room.Database
import android.arch.persistence.room.RoomDatabase

@Database(entities = [Word::class], version = 1)
abstract class WordRoomDatabase : RoomDatabase() {
    abstract fun wordDao(): WordDao
}
```

Faça com que a classe *WordRoomDatabase* seja um *singleton* para prevenir múltiplas instâncias do mesmo banco de dados.

```
companion object {
    private var instance: WordRoomDatabase? = null

    fun getDatabase(context: Context): WordRoomDatabase? {
        if (instance == null) {
            synchronized(WordRoomDatabase::class.java) {
                // Criaremos o banco de dados aqui
            }
        }

        return instance
    }
}
```

E adicione o código (abaixo do comentário demarcado) para pegar o banco de dados:

```
instance = Room.databaseBuilder(  
    context.applicationContext,  
    WordRoomDatabase::class.java,  
    "word_database"  
) .build()
```

7.3.7. Criando o Repository

O *Repository* é uma classe abstrata para acessar mais de uma fonte de dados. Essa classe não faz parte do *Architecture Components* mas é uma sugestão de boas práticas do *Google*, mantendo nossa aplicação *clean* e bem mais eficiente.

Para implementá-la neste exemplo, crie um pacote ``repositories`` e uma classe abstrata chamada *WordRepository* que irá pegar o *DAO* existente e a lista de palavras (*Words*), ficando assim:

```
import android.app.Application  
import android.arch.lifecycle.LiveData  
  
class WordRepository(application: Application) {  
    private val wordDao: WordDao  
    private val allWords: LiveData<List<Word>>  
  
    init {  
        val db = WordRoomDatabase.getDatabase(application)  
        wordDao = db!!.wordDao()  
        allWords = wordDao.getAllWords()  
    }  
}
```

Antes de prosseguir, faremos o import da biblioteca Anko Commons para o projeto, que facilitará algumas operações.

```
implementation "org.jetbrains.anko:anko-common:0.10.7"
```

Crie o método de inserção de registros na base de dados, certificando que a lógica irá executar em uma *thread* separada, que nesse caso usaremos uma *AsyncTask* comum:

```
fun insert(word: Word) {  
    doAsync {  
        wordDao.insert(word)  
    }  
}
```

7.3.8. Criando o ViewModel

Como dito anteriormente, há uma série de benefícios em utilizar o *ViewModel*, e em conjunto com o *LiveData* e o *Room*, essa lista de melhorias só aumenta. O *ViewModel* simplesmente vai cuidar e processar todos os dados que a *UI* necessita.

Além disso, dentro do *ViewModel* utilizaremos o *LiveData* que nos permitirá duas coisas fundamentais:

- Adicionar observadores em vez de submeter as modificações e apenas atualizar a *UI* quando o dado realmente trocar.
- O *Repository* e a *UI* estão completamente separados pelo *ViewModel* e não há chamadas de database do *ViewModel*, tornando o código mais fácil de manter e de testar.

Primeiro, crie uma classe chamada *WordViewModel* que estenda a classe *AndroidViewModel*. Guarde a referência do *Repository* em uma variável e o *LiveData* de todas as palavras, ficando assim:

```
import android.app.Application  
import android.arch.lifecycle.AndroidViewModel  
import android.arch.lifecycle.LiveData  
  
class WordViewModel(application: Application) :  
    AndroidViewModel(application) {  
    private var repository: WordRepository =  
        WordRepository(application)  
  
    private var allWords: LiveData<List<Word>> =  
        repository.allWords  
}
```

E para finalizar, defina o membro de inserção de uma nova palavra.


```
fun insert(word: Word) {  
    repository.insert(word)  
}
```

7.3.9. Adicionando o Layout XML

Adicione os seguintes estilos no arquivo 'res/values/styles.xml':

```
<style name="word_title">  
    <item name="android:layout_width">match_parent</item>  
    <item name="android:layout_height">26dp</item>  
    <item name="android:textSize">24sp</item>  
    <item name="android:textStyle">bold</item>  
    <item name="android:layout_marginBottom">6dp</item>  
    <item name="android:paddingLeft">8dp</item>  
</style>
```

Adicione o novo layout em 'res/layouts/recyclerview_item.xml':

```
<?xml version="1.0" encoding="utf-8"?>  
<LinearLayout  
    xmlns:android="http://schemas.android.com/apk/res/android"  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:orientation="vertical">  
  
    <TextView  
        android:id="@+id/textView"  
        style="@style/word_title"  
        android:layout_width="match_parent"  
        android:layout_height="wrap_content"  
        android:background="@android:color/holo_orange_light"  
    />  
  
</LinearLayout>
```

Dentro do arquivo 'res/layouts/content_main.xml', insira o seguinte código:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"

    app:layout_behavior="@string/appbar_scrolling_view_behavior"
    tools:context=".MainActivity"
    tools:showIn="@layout/activity_main">

    <android.support.v7.widget.RecyclerView
        android:id="@+id/recyclerView"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:background="@android:color/darker_gray"
        android:scrollbars="vertical"
        tools:listitem="@layout/recyclerview_item" />
</LinearLayout>
```

O seu *FloatingActionButton* dentro do arquivo 'res/layout/activity_main.xml' deverá exibir um ícone de *add* (+), portanto, adicione a seguinte linha ao *FAB*:

```
app:srcCompat="@android:drawable/ic_input_add"
android:tint="#FFFFFF"
```

Para concluir a exibição dos dados, crie uma *RecyclerView* chamada *WordListAdapter* com o seguinte código abaixo:

```
import android.content.Context
import android.support.v7.widget.RecyclerView
import android.support.v7.widget.RecyclerView.Adapter
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import android.widget.TextView
```

```

class WordListAdapter constructor(context: Context) :
    Adapter<WordListAdapter.WordViewHolder>() {

    private val inflater: LayoutInflater =
        LayoutInflater.from(context)
    var words: List<Word> = emptyList()
    set(value) {
        field = value
        notifyDataSetChanged()
    }

    override fun onCreateViewHolder(parent: ViewGroup,
        viewType: Int): WordViewHolder {
        val itemView =
            inflater.inflate(R.layout.recyclerview_item, parent, false)
        return WordViewHolder(itemView)
    }

    override fun onBindViewHolder(holder: WordViewHolder,
        position: Int) {
        if (words.isEmpty()) {
            holder.wordItemView.text = "No Words"
        } else {
            holder.wordItemView.text = words[position].word
        }
    }

    override fun getItemCount() = words.size

    class WordViewHolder(itemView: View) :
        RecyclerView.ViewHolder(itemView) {
        val wordItemView: TextView =
            itemView.findViewById(R.id.textView)
    }
}

```

E depois crie uma instância na *MainActivity* da *RecyclerView* criada (certifique-se de que está dentro do método *onCreate()*):

```
RecyclerView recyclerView = findViewById(R.id.recyclerView);  
final WordListAdapter adapter = new WordListAdapter(this);  
recyclerView.setAdapter(adapter);  
recyclerView.setLayoutManager(new LinearLayoutManager(this));
```

Rode a aplicação para certificar que tudo funciona. Note que não há itens ainda pois não adicionamos nada à base de dados, conforme podemos ver na figura 18.

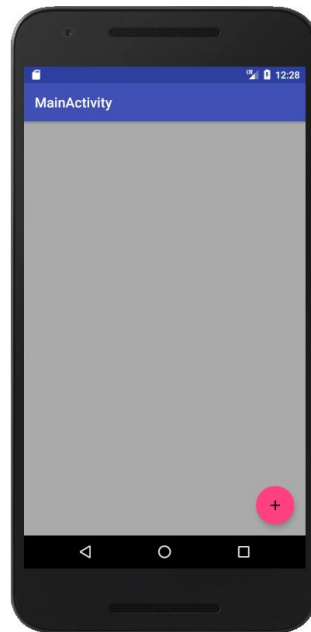


Figura 18: Tela do aplicativo desenvolvido até o momento
Fonte: autor - *Android Emulator (API 24)*

7.3.10. Populando a base de dados

Como não temos nenhum dado para testar iremos fazer isso de duas maneiras: a primeira será feito assim que o aplicativo abrir e a segunda é criando uma *Activity* para isso.

Para a primeira implementação, iremos basicamente excluir todos os dados existentes e adicionar novas informações. Como o *Room* não consegue alterar o banco de dados na *MainThread*, iremos precisar criar um *RoomDatabase.Callback* que fará o serviço para nós. Adicione o seguinte trecho dentro da classe *WordRoomDatabase*:

```
private val roomDatabaseCallback = object :
RoomDatabase.Callback() {
    override fun onOpen(db: SupportSQLiteDatabase) {
        super.onOpen(db)

        instance?.let { roomDb ->
            doAsync {
                val dao = roomDb.wordDao()

                dao.deleteAll()

                val word = Word("Movile")
                dao.insert(word)

                val word2 = Word("Next")
                dao.insert(word2)
            }
        }
    }
}
```

Com isso declarado vamos para o código que pega a instância do banco de dados (abaixo do comentário '// Criaremos o banco de dados aqui'), e adicionamos a seguinte linha antes do `.build()`:

```
.addCallback(roomDatabaseCallback)
```

7.3.11. Criando uma nova Activity para adicionar palavras

Feito isso, criaremos uma *EmptyActivity* com o nome *NewWordActivity* e adicionaremos várias informações no projeto.

No arquivo 'res/values/strings.xml':

```
<string name="hint_word">Word...</string>
<string name="button_save">Save</string>
<string name="empty_not_saved">Word not saved because it is
empty.</string>
```

No arquivo 'res/values/colors.xml':

```
<color name="buttonLabel">#D3D3D3</color>
```

No arquivo 'res/values/dimens.xml':

```
<dimen name="small_padding">6dp</dimen>
<dimen name="big_padding">16dp</dimen>
```

No arquivo 'res/layouts/activity_new_word.xml':

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context=".NewWordActivity">

    <EditText
        android:id="@+id/etWord"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginBottom="@dimen/big_padding"
        android:layout_marginTop="@dimen/big_padding"
        android:hint="@string/hint_word"
        android:inputType="textAutoComplete"
        android:padding="@dimen/small_padding"
        android:textSize="18sp"
        app:fontFamily="sans-serif-light" />

    <Button
        android:id="@+id/btSave"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:background="@color/colorPrimary"
```

```
        android:color="@color/buttonLabel"
        android:text="@string/button_save" />
    </LinearLayout>
```

No arquivo 'NewWordActivity.java':

```
import android.app.Activity
import android.content.Intent
import android.os.Bundle
import android.support.v7.app.AppCompatActivity
import kotlinx.android.synthetic.main.activity_new_word.*

class NewWordActivity : AppCompatActivity() {

    companion object {
        const val EXTRA_REPLY =
"com.example.android.wordlistsql.REPLY"
    }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_new_word)

        btSave.setOnClickListener {
            val replyIntent = Intent()

            if (etWord.text.isEmpty()) {
                setResult(Activity.RESULT_CANCELED,
replyIntent)
            } else {
                val word = etWord.text.toString()
                replyIntent.putExtra(EXTRA_REPLY, word)
                setResult(Activity.RESULT_OK, replyIntent)
            }

            finish()
        }
    }
}
```

```
}  
}
```

7.3.12. Conectando os dados

Para finalizar devemos conectar a *UI* com banco de dados salvando as novas palavras que o usuário adicionar e exibir todo o conteúdo do banco na *RecyclerView*.

Para isso, iremos na *MainActivity* realizar as declarações do *ViewModel*, adicionaremos um observador para detectar mudanças no *LiveData* e prepararemos a *Activity* principal para receber resultados da *Activity* de adição.

Na *MainActivity*, adicione as seguintes declarações no código já existente.

Variáveis da classe:

```
companion object {  
    const val NEW_WORD_ACTIVITY_REQUEST_CODE = 1  
}  
  
lateinit var wordViewModel: WordViewModel
```

Dentro do método *onCreate()*. Note que o *clickListener* do *FAB* já deve estar definido - substitua apenas o conteúdo do *onClick()*:

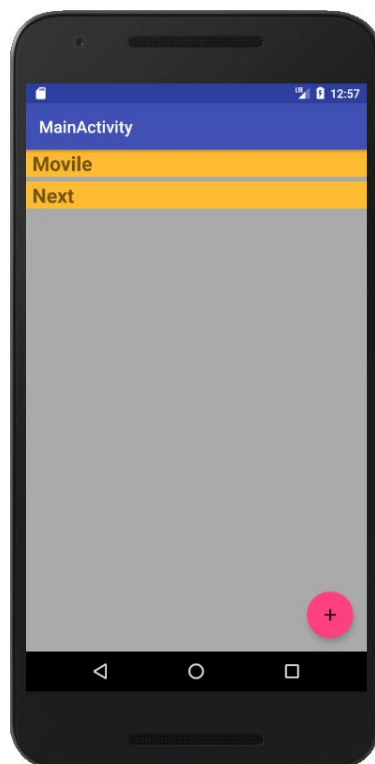
```
wordViewModel =  
ViewModelProviders.of(this).get(WordViewModel::class.java)  
wordViewModel.allWords.observe(this,  
    Observer<List<Word>> { words ->  
        words?.let {  
            adapter.words = it  
        }  
    })  
  
fab.setOnClickListener {  
    val intent = Intent(this@MainActivity,  
NewWordActivity::class.java)  
    startActivityForResult(intent,  
NEW_WORD_ACTIVITY_REQUEST_CODE)  
}
```


No final da classe:

```
override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {
    super.onActivityResult(requestCode, resultCode, data)

    if (requestCode == NEW_WORD_ACTIVITY_REQUEST_CODE &&
        resultCode == Activity.RESULT_OK) {
        data?.let {
            val word =
                Word(it.getStringExtra(NewWordActivity.EXTRA_REPLY))
            wordViewModel.insert(word)
        }
    } else {
        toast(R.string.empty_not_saved)
    }
}
```

Com isso temos o funcionamento esperado, conforme demonstrado na figura 19.



Desenvolvido por [Paulo Salvatore](#)

Figura 19: Execução da aplicação final armazenando dados com *Room* e utilizando *ViewModel* e *LiveData* para manipulação das informações.

Fonte: autor - *Android Emulator (API: 24)*

Sugestões de CodeLabs para realizar depois: [\[ref 32\]](#) e [\[ref 33\]](#).

8. Referências Bibliográficas

1. 97 Journey Every Programmer should Accomplish - <https://medium.com/@biratkirat/97-journey-every-programmer-should-accomplish-a0c53dbbfd47>
2. Android - Application Components - https://www.tutorialspoint.com/android/android_application_components.htm (Acessado em 23 de Junho de 2018)
3. Activities - <https://developer.android.com/guide/components/activities/> (Acessado em 23 de Junho de 2018)
4. Activities Lifecycle - <https://developer.android.com/guide/components/activities/activity-lifecycle> (Acessado em 24 de Junho de 2018)
5. Services - <https://developer.android.com/guide/components/services> (Acessado em 23 de Junho de 2018)
6. Extending Intent Service - <https://developer.android.com/guide/components/services#ExtendingIntentService> (Acessado em 24 de Junho de 2018)
7. Extending Service - <https://developer.android.com/guide/components/services#ExtendingService> (Acessado em 24 de Junho de 2018)
8. Bound Services - <https://developer.android.com/guide/components/bound-services> (Acessado em 24 de Junho de 2018)
9. Broadcasts - <https://developer.android.com/guide/components/broadcasts> (Acessado em 23 de Junho de 2018)
10. Content Providers - <https://developer.android.com/guide/topics/providers/content-providers> (Acessado em 24 de Junho de 2018)
11. Fragments - <https://developer.android.com/guide/components/fragments> (Acessado em 23 de Junho de 2018)
12. DialogFragment - <https://developer.android.com/reference/android/app/DialogFragment> (Acessado em 24 de Junho de 2018)
13. ListFragment - <https://developer.android.com/reference/android/app/ListFragment> (Acessado em 24 de Junho de 2018)
14. PreferenceFragment - <https://developer.android.com/reference/android/preference/PreferenceFragment> (Acessado em 24 de Junho de 2018)
15. WebViewFragment - <https://developer.android.com/reference/android/webkit/WebViewFragment> (Acessado em 24 de Junho de 2018)
16. Fragments deprecated (posted on GitHub) - <https://github.com/android/android-tx/pull/161#issuecomment-363270555>

(Acessado em 24 de Junho de 2018)

17. Fragments deprecated (posted on Twitter) -

<https://twitter.com/ianhlake/status/971455064274485248> (Acessado em 24 de Junho de 2018)

18. Android Support Library -

<https://developer.android.com/topic/libraries/support-library/> (Acessado em 24 de Junho de 2018)

19. Exploring the new Android Architecture Components library -

<https://medium.com/exploring-android/exploring-the-new-android-architecture-components-c33b15d89c23> (Acessado em 24 de Junho de 2018)

20. Android Jetpack - <https://developer.android.com/jetpack/> (Acessado em 23 de Junho de 2018)

21. Use Android Jetpack to Accelerate Your App Development -

<https://android-developers.googleblog.com/2018/05/use-android-jetpack-to-accelerate-your.html> (Acessado em 24 de Junho de 2018)

22. Handling lifecycles with lifecycle-aware components -

<https://developer.android.com/topic/libraries/architecture/lifecycle> (Acessado em 27 de Junho de 2018)

23. ViewModel Overview -

<https://developer.android.com/topic/libraries/architecture/viewmodel> (Acessado em 27 de Junho de 2018)

24. A complete idiot's guide to Clean Architecture -

<https://android.jlelse.eu/a-complete-idiot-s-guide-to-clean-architecture-2422f428946f> (Acessado em 27 de Junho de 2018)

25. LiveData overview -

<https://developer.android.com/topic/libraries/architecture/livedata> (Acessado em 27 de Junho de 2018)

26. CodeLab - Android Lifecycles -

<https://codelabs.developers.google.com/codelabs/android-lifecycles/> (Acessado em 27 de Junho de 2018)

27. CodeLab - Android Room with a View -

<https://codelabs.developers.google.com/codelabs/android-room-with-a-view/> (Acessado em 28 de Junho de 2018)

28. Adding Components to your Project

<https://developer.android.com/topic/libraries/architecture/adding-components> (Acessado em 30 de Junho de 2018)

29. Data and file storage overview -

<https://developer.android.com/guide/topics/data/data-storage> (Acessado em 28 de Junho de 2018)

30. Room Persistence Library -

<https://developer.android.com/topic/libraries/architecture/room> (Acessado em 30 de Junho de 2018)

31. Adding Components to Your Project -

<https://developer.android.com/topic/libraries/architecture/adding-components> (Acessado em 30 de Junho de 2018)

- 32. CodeLab - Build an App with Architecture Components - <https://codelabs.developers.google.com/codelabs/build-app-with-arch-component/s/> (Acessado em 30 de Junho de 2018)
- 33. CodeLab - Android Persistence - <https://codelabs.developers.google.com/codelabs/android-persistence/> (Acessado em 30 de Junho de 2018)

9. Vídeos Recomendados

1. Architecture Components - Introduction (Google I/O '17) - <https://www.youtube.com/watch?v=FrteWKKVyzI> (Acessado em 24 de Junho de 2018)
2. Android Jetpack: what's new in Architecture Components (Google I/O '18) - <https://www.youtube.com/watch?v=pErTyQpA390> (Acessado em 24 de Junho de 2018)
 - 2:30 What's new in a nutshell
 - 3:13 What's new in the Architecture layer
 - 3:33 Lifecycles
 - 8:02 Databinding
 - 9:37 Room
 - 12:57 Paging
 - 16:05 Navigation
 - 21:43 WorkManager
 - 28:30 future goals
3. Android Jetpack: Improve Your App's Architecture - <https://www.youtube.com/watch?v=7p22cSzniBM> (Acessado em 23 de Junho de 2018)
4. Architecture Components - Persistence and Offline (Google I/O '17) - <https://www.youtube.com/watch?v=MfHsPGQ6bgE> (Acessado em 24 de Junho de 2018)
- ref. Architecture Components - Solving the Lifecycle Problem (Google I/O '17) - <https://www.youtube.com/watch?v=bEKNi1JOrNs> (Acessado em 24 de Junho de 2018)

10. Licença e termos de uso

Todos os direitos são reservados. É expressamente proibida a distribuição desse material sem a permissão, por escrito, do **autor** ou da **GlobalCode Treinamentos Ltda - ME**. Mais informações sobre *copyright*:

<https://choosealicense.com/no-permission/>

Conteúdos que foram baseados em declarações providas da documentação do *Google Developers* estão licenciados sob a *Creative Commons License 3.0* (<https://creativecommons.org/licenses/by/3.0/>).

Códigos providos do *Google* estão licenciados sob a *Apache License 2.0* (<https://www.apache.org/licenses/LICENSE-2.0>).

11. Leitura adicional

11.1. Lifecycles

- The Android Lifecycle cheat sheet — part I: Single Activities - <https://medium.com/google-developers/the-android-lifecycle-cheat-sheet-part-i-single-activities-e49fd3d202ab>
- The Android Lifecycle cheat sheet — part II: Multiple activities - <https://medium.com/@JoseAlcerreca/the-android-lifecycle-cheat-sheet-part-ii-multiple-activities-a411fd139f24>
- The Android Lifecycle cheat sheet — part III : Fragments - <https://medium.com/@JoseAlcerreca/the-android-lifecycle-cheat-sheet-part-iii-fragments-afc87d4f37fd>
- Saving and restoring transient UI state - <https://developer.android.com/guide/components/activities/activity-lifecycle#saras>

11.2. Room

- Movable Blog - Room Android – menos código e menos bugs - <https://movile.blog/room-android-menos-codigo-e-menos-bugs/>
- Data and file storage overview - <https://developer.android.com/guide/topics/data/data-storage?hl=pt-br>
- Save data in a local database using Room - <https://developer.android.com/training/data-storage/room/>
- 7 Steps To Room - <https://medium.com/google-developers/7-steps-to-room-27a5fe5f99b2>
- Room Persistence Library - <https://developer.android.com/topic/libraries/architecture/room>
- Android Room with a View - <https://codelabs.developers.google.com/codelabs/android-room-with-a-view/#0>
- Android Persistence codelab - <https://codelabs.developers.google.com/codelabs/android-persistence/#0>
- ViewModel - <https://developer.android.com/topic/libraries/architecture/viewmodel>
- LiveData - <https://developer.android.com/topic/libraries/architecture/livedata>
- Android Jetpack: ViewModel - <https://youtu.be/5qLIPTDE274>
- Android Architecture Components samples - <https://github.com/googlesamples/android-architecture-components/>
- Android Architecture Counter Sample
- Here's a simple sample app that demonstrates some parts of Android Architecture Components. Plus, it's entirely written in Kotlin!
- <https://github.com/dlew/android-architecture-counter-sample>

- A collection of samples using the Architecture Components:
<https://github.com/googlesamples/android-architecture-components>
 - Room
 - Lifecycle-aware components
 - ViewModels
 - LiveData
 - Paging (preview)
 - WorkManager (alpha)